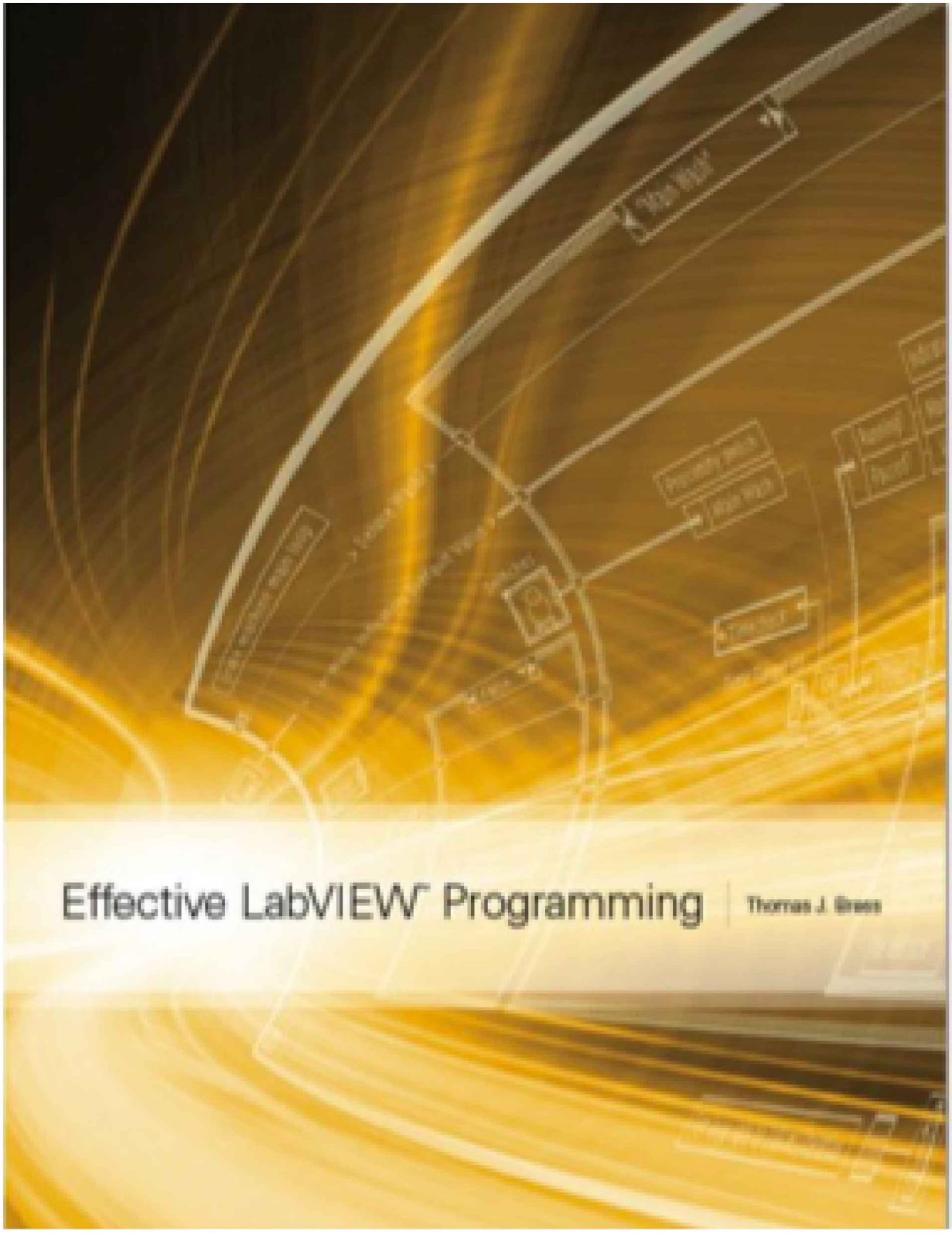

Effective LabVIEW® Programming
Thomas J. Bress

Effective LabVIEW Programming

Thomas J. Bress



ISBN-10: 1-934891-08-8
ISBN-13: 978-1-934891-08-7

Publisher: Tom Robbins
Development Manager: Gretchen Edelmon
Project Manager: Catherine Peacock

Library of Congress Control Number: 2013940699

© 2013 National Technology and Science Press.

All rights reserved. Neither this book, nor any portion of it, may be copied or reproduced in any form or by any means without written permission of the publisher.

NTS Press respects the intellectual property of others, and we ask our readers to do the same. This book is protected by copyright and other intellectual property laws. Where the software referred to in this book may be used to reproduce software or other materials belonging to others, you should use such software only to reproduce materials that you may reproduce in accordance with the terms of any applicable license or other legal restriction.

Neither the author nor the publisher makes any warranties of any kind, including without limitation any warranty as to the sufficiency of the book or of any information, theories, or programs contained or described in the book will not infringe any patent or other intellectual property right.

In no event shall the publisher or author be liable for any direct, indirect, special, incidental, cover, economic or consequential damages arising out of this book or any possibility of such damages, and even if caused or contributed to by the negligence of the publisher, author, or others.

National Instruments, LabVIEW and NI-DAQ are trademarks of National Instruments. National Instruments is not affiliated with the author, and does not authorize, sponsor, endorse or otherwise approve the contents of this publication. National Instruments is not liable for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause by the use of any information in this publication.

All other trademarks or product names are the property of their respective owners.

Additional Disclaimers:

The reader assumes all risk of use of this book and of all information, theories, and programs contained or described in it. This book may contain technical inaccuracies, typographical errors, other errors and omissions, and out-of-date information. Neither the author nor the publisher assumes any responsibility or liability for any errors or omissions of any kind, to update any information, or for any infringement of any patent or other intellectual property right.

Neither the author nor the publisher makes any warranties of any kind, including without limitation any warranty as to the sufficiency of the book or of any information, theories, or programs contained or described in it, and any warranty that use of any information, theories, or programs contained or described in the book will not infringe any patent or other intellectual property right. THIS BOOK IS PROVIDED "AS IS." ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY AND ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, ARE DISCLAIMED.

No right or license is granted by publisher or author under any patent or other intellectual property right, expressly, or by implication or estoppel.

IN NO EVENT SHALL THE PUBLISHER OR THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, COVER, ECONOMIC, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THIS BOOK OR ANY INFORMATION, THEORIES, OR PROGRAMS CONTAINED OR DESCRIBED IN IT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, AND EVEN IF CAUSED OR CONTRIBUTED TO BY THE NEGLIGENCE OF THE PUBLISHER, THE AUTHOR, OR OTHERS. Applicable law may not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you.

This book is dedicated to spaghetti coders everywhere.

Contents

Preface xi

Introduction xiii

1 LabVIEW Basics 1

1.1 Anatomy of a VI 3
1.2 A Simple Example: `Hello World.vi` 3
1.3 Controls, Indicators, and Wires 4
1.4 Dataflow 8
1.5 A More Substantial Example: `Clean Up Folder.vi` 11
1.6 Arrays and Clusters 13
1.7 Creating Sub-VIs 17
1.8 Case Structures 20
1.9 The Standard Sub-VI Pattern with Error Handling 24
1.10 The `Move Or Delete` Sub-VI 25
1.11 The `Right File Type` Sub-VI 27
1.12 Loops 27
1.13 Shift Registers and Feedback Nodes 34
1.14 Error Handling 35
1.15 Developing the `Clean Up Folder` VI 39
1.16 Passing Data by Reference 42
1.17 File I/O 43
1.18 The `Simulated Data Acquisition` VI 48
1.19 Loop Timing 49
1.20 The `Data Over Limit` sub-VI 51
1.21 The `Create Log String` sub-VI 52
1.22 Absolute and Relative File Paths 53
1.23 The `Append To Log File` Sub-VI 54
1.24 Property Nodes and Invoke Nodes 56
1.25 Developing the `Simulated Data Acquisition` VI 57
1.26 Elements of LabVIEW Style 59

2 Functional Specifications 61

2.1 The `Multitest` VI 61
2.2 State Transition Diagram for the `Multitest` VI 62
2.3 The Car Wash Controller 63
2.4 State Transition Diagrams for the Car Wash Controller 66

3 Problem Set 1 68
3.1 Car Wash Controller Exercises 68
3.2 Traffic Light Controller Exercises 68
3.3 Security System Controller Exercises 71
3.4 Boiler Controller Exercises 74
3.5 Car Wash 2 Controller Exercises 86

4 Classic State Machines 90
4.1 State Machine Elements 90
4.2 State Machine Structure 92
4.3 Placing Elements in the State Machine 96
4.4 Building a State Machine From Scratch: The `Multitest` VI 97

5 Action Engines 106
5.1 Anatomy of an Action Engine 106
5.2 Feedback Node Action Engines 107
5.3 Example: Creating a Counter 110
5.4 The State Variable Method 111

6 The Car Wash Timer 115
6.1 The Functions of the Car Wash Timer 116
6.2 Inputs of the Car Wash Timer 116
6.3 Outputs of the Car Wash Timer 117
6.4 Icon and Front Panel of the Car Wash Timer 117
6.5 State Variables of the Car Wash Timer 118
6.6 The *Timecheck* Action of the Car Wash Timer 118
6.7 The *Reset Timer* Function of the Car Wash Timer 120
6.8 The Kernel of the Car Wash Timer 121
6.9 Providing Storage for the Car Wash Timer 123

7 Problem Set 2 125
7.1 Car Wash 2 Controller Exercises 125
7.2 Boiler Controller Exercises 126
7.3 Traffic Light Controller Exercises 127
7.4 Security System Controller Exercises 129

8 The Classic State Machine Car Wash Controller 132
8.1 Determine the States 133
8.2 Plan the State Transitions 134
8.3 Create the State Constant 134
8.4 Build the Front Panel 135
8.5 Add the Main Loop and Case Structure 135
8.6 Add an Error or Stop Check 139
8.7 Develop Individual States 142
8.8 Develop the *Initialize* State 142
8.9 Develop the *Unlock Selector* State 145
8.10 Develop the *Wait* State 145
8.11 Develop the *Lock Selector* State 146
8.12 Develop the Car Wash States 147
8.13 Develop the *Shutdown* State 151

9 Problem Set 3 152
9.1 Traffic Light Controller Exercises 152
9.2 Security System Controller Exercises 152

9.3 Boiler Controller Exercises 166
9.4 Car Wash 2 Exercises 166

10 Event-Driven State Machines 167
10.1 Event Structures 168
10.2 Anatomy of an Event Structure 168
10.3 Configuring an Event Structure 170
10.4 Event Structure Quirks and Caveats 171
10.5 Event Structure Guidelines 175
10.6 Example: Building an Event-Driven State Machine 175

11 The Event-Driven Car Wash Controller 178
11.1 Adding the Event Structure to the *Wait* State 178
11.2 The STOP Button 180

12 Problem Set 4 182
12.1 Traffic Light Controller Exercises 182
12.2 Security System Controller Exercises 182
12.3 Boiler Controller Exercises 182
12.4 Car Wash 2 Exercises 183

13 Queued State Machines 184
13.1 The Power of Queues 185
13.2 Queue Functions 185
13.3 By-Value versus By-Reference 188
13.4 Building a Queued Event-Driven State Machine 191

14 The Queued Car Wash Controller 197
14.1 Building a Non-Square Icon 199
14.2 Examining the Queued Car Wash Controller 201
14.3 The Car Wash States 206

15 Data Acquisition with the Multitest VI 210
15.1 Strain Gages and Bridge Circuits 210
15.2 Data Acquisition in LabVIEW 212
15.3 Functional Specification 212
15.4 State Transition Diagram 212
15.5 Building the DAQ Task 214
15.6 Developing the `Multitest` Data Acquisition VI 217

16 Problem Set 5 224
16.1 Traffic Light Controller Exercises 224
16.2 Security System Controller Exercises 224
16.3 Boiler Controller Exercises 225
16.4 Car Wash 2 Exercises 225
16.5 DAQ State Machine Exercises 226

17 Preparing for the CLD Exam 231
17.1 Documentation 232
17.2 Style 234
17.3 Functionality 235
17.4 Preparing for the CLD Exam 236
17.5 Time Management 237
17.6 Practicing for the Exam 238

18 Problem Set 6 239

19 Problem Set 7 246

20 State Machines with Arguments 254

20.1 The Data Highway 255
20.2 Using Arguments to Pass Information 257
20.3 Passing Arguments Using Clusters 257
20.4 Passing Arguments Using Strings 260
20.5 Designing Parsing Sub-VIs 260
20.6 The String-Based `Multitest` VI 263
20.7 Disadvantages of Using Strings 266
20.8 Improving the String-Based State Machine 268

21 Car Wash Controller with Arguments 274

21.1 Car Wash Controller with Time and Cycle Arguments 275
21.2 Developing Parsing Routines for the Car Wash Controller 276
21.3 Developing a Generic *Cycle* State 278
21.4 Implementing the Car Wash Controller with Arguments 281

22 Problem Set 8 287

22.1 DAQ State Machine Exercises 287
22.2 Car Wash Controller Exercises 287
22.3 Traffic Light Controller Exercises 288
22.4 Security System Controller Exercises 288
22.5 Boiler Controller Exercises 289
22.6 Car Wash 2 Controller Exercises 294

23 Data Encapsulation 298

23.1 LabVIEW Objects and Classes 302

24 Problem Set 9 313

24.1 Traffic Light Controller Exercises 313
24.2 Boiler Controller Exercises 313
24.3 Car Wash 2 Controller Exercises 314

25 Inheritance and Composition 316

25.1 Inheritance and Composition 318
25.2 Creating the `Cycle` Class 320
25.3 Data Member Access Sub-VIs 321
25.4 Dynamic Dispatch Sub-VIs 324
25.5 Static Dispatch Sub-VIs 327
25.6 Access Scope 329
25.7 The Completed `Cycle` Class 331
25.8 Creating the Child Classes 332
25.9 Developing Override VIs 333
25.10 The Factory Method 338
25.11 Dynamic Loading of Classes 342
25.12 The Car Wash Controller with Inheritance and Composition 345
25.13 Why Bother with Classes? 347

26 Problem Set 10 348

26.1 Car Wash 2 Exercises 348
26.2 Traffic Light Controller Exercises 351
26.3 Boiler Controller Exercises 353

27 Producer–Consumer State Machines 368

27.1 Stopping Parallel Loops 369
27.2 The Producer–Consumer Architecture 371
27.3 The Producer–Consumer (Events) Architecture 372
27.4 User Events 377
27.5 Producer–Consumer Pattern with User Events 383
27.6 Value(Signalling) or User Events? 385
27.7 Error Handling in the Producer–Consumer Architecture 385

28 Combatting Race Conditions 387

28.1 Guarding Against Race Conditions 393
28.2 The Producer–Consumer Car Wash Controller 397
28.3 The Producer–Consumer Car Wash Controller with User Events 400
28.4 Car Wash Race Condition 401
28.5 Semaphores and the Producer–Consumer Car Wash Controller 402
28.6 Breaking the Deadlock 404
28.7 Living with the Race Condition 408

29 Extending the Language 411

29.1 The `Switched Queue` Class 412
29.2 Data Value References 414
29.3 Data Accessor Methods of the `Switched Queue` Class 418
29.4 `Switched Queue` Method Sub-VIs 419
29.5 Using the Switched Queue 426
29.6 The Pros and Cons 427

30 Problem Set 11 429

30.1 Security System Controller Exercises 429
30.2 Car Wash 2 Exercises 437

31 Wrapping It Up 438

31.1 The Classic State Machine 438
31.2 Event-Driven State Machine 438
31.3 Queued State Machine 439
31.4 State Machines with Arguments 439
31.5 State Machines with Classes 440
31.6 Producer–Consumer State Machines 440
31.7 Coupling 440
31.8 Selecting an Architecture 441
31.9 How Can You Improve? 442
31.10 Where to Go from Here 443

A Sample CLD Exam— Traffic Light Controller 444

B Sample CLD Exam— Security System 446

C Sample CLD Exam—Boiler Controller 449
D Sample CLD Exam—Car Wash Controller 2 454
E Sample CLD Exam—Sprinkler Controller 458
F Sample CLD Exam—ATM Controller 463
G Problem Set 1 Solutions 469
H Problem Set 2 Solutions 493
I Problem Set 3 Solutions 502
J Problem Set 4 Solutions 529
K Problem Set 5 Solutions 535
L Problem Set 6 Solutions 556
M Problem Set 7 Solutions 570
N Problem Set 8 Solutions 592
O Problem Set 9 Solutions 622
P Problem Set 10 Solutions 634
Q Problem Set 11 Solutions 678
Index 696

Preface

Many new users come to LabVIEW with some experience in another programming language. They already know about loops, arrays, conditional statements, subroutines, file I/O, etc. Because of this, many expect that LabVIEW will be easy to learn. After all, it's just another programming language: memorize some syntax rules, figure out how loops and conditional statements work, keep a list of commonly used functions handy, and they should be fluent in no time, right?

Yet they find that it is harder than that. Beginning and intermediate LabVIEW programmers often suffer from "blank page" syndrome. They understand the basic elements and structures of LabVIEW, but they have difficulty knowing where or how to start a new project. They often find they easily can produce functional code but struggle to produce maintainable code. The code produced is often difficult to read—even by the original developer. The problem is that most beginning and intermediate LabVIEW programmers simply don't know what a good LabVIEW program looks like. LabVIEW programmers often produce spaghetti code because, while they understand LabVIEW programming elements, they do not understand how to put them together effectively. Since LabVIEW is a graphical programming language, spaghetti code really does look like spaghetti (Figure 0.1). Spaghetti code may be temporarily functional, but woe to the programmer who needs to extend its functionality—even if that programmer is the original author.

LabVIEW programmers often exist in isolation. Usually there is a single LabVIEW programmer in a work group, leading to something that might be called "the lone wolf syndrome." Lone wolves have no LabVIEW frame of reference, no experienced co-workers to review their code, and no contact with the greater LabVIEW community. As long as the lone wolf produces functional code—no matter the quality—he or she is considered to be the local LabVIEW guru.

In many ways, LabVIEW is easier to learn than a text-based language, but few other languages follow the dataflow paradigm. Many programming concepts carry over to LabVIEW directly, but the elements of good LabVIEW style and proven LabVIEW design patterns must be learned from scratch. Programmers must learn to "think" in LabVIEW.

Coaching by an experienced developer is the best way to learn style and design in any computer language. When a mentor is unavailable, the next best way is to learn through books. Unfortunately, the number of LabVIEW books is fairly small. The few general LabVIEW books aimed at beginners tend to describe the basic elements of LabVIEW rather than describe how to integrate those elements into an effective LabVIEW program.

This book is different. It does not describe every LabVIEW function in excruciating detail. In fact, the set of LabVIEW functions used in this book is quite small.

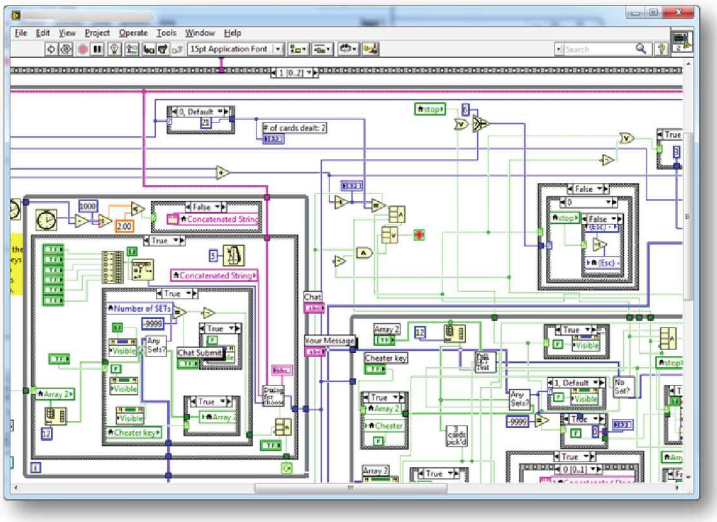


Figure 0.1: Spaghetti code

This book demonstrates what good LabVIEW programs look like. It does this using core LabVIEW functions and common design patterns for a common project drawn from the Certified LabVIEW Developer exam. These patterns build on each other. They provide a firm starting point for most beginning and intermediate projects.

My hope is that new LabVIEW users will absorb these design patterns and implement them in their earliest projects. In this way, they may avoid joining the ranks of LabVIEW spaghetti coders. I also hope that current spaghetti coders will be inspired to re-examine their body of work and re-evaluate their programming practices. It takes humility to admit that your code may not be as good as you once thought, and it takes effort to change your programming habits. But in return, this investment in effort and humility will take your LabVIEW coding to the next level. It did for me.

My final hope for this book is that it will serve as a bridge between the many LabVIEW lone wolves and the greater LabVIEW community. Good programming style and effective design patterns are not developed overnight. They are formed through many years of reflection on the practical experience of the LabVIEW community. By adopting these practices and patterns, the lone wolf can connect with the community, benefit from its experience, and strengthen the community in return.

I would like to thank a number of people who helped me bring this book into being. Andrew Watchorn at National Instruments listened to my ideas for a book that I would like to write "someday" and immediately called NTS Press, setting this book into motion. Thanks to Tom Robbins, my publisher, for shepherding this book with patience. Special thanks to Stephen Mercer, Fabiola De La Cueva, Justin Petry and Jared Kirschner for reviewing my manuscript and vastly improving the final product. Also, I would like to thank Jason Goode, Joe Silva, and David Schlickeisen of NI for producing the cover. Thanks also to Rose Kernan and her crew for putting the book into the final form you see here. Finally, thanks to my wife Heather and my boys, Nathan and Zachary, for allowing me to spend time on this book that would have been better spent with them.

Introduction

Who Should Read This Book

This book is suitable for all beginning and intermediate LabVIEW programmers. If you are already a Certified LabVIEW Architect, this book might be too basic for you. Everyone else should find it valuable. This book is unique in its approach. Rather than presenting detailed instructions on how to navigate the LabVIEW user interface, this book focuses on how to use the dataflow paradigm of LabVIEW to create effective programs that are readable, scalable, and maintainable. This book will teach you what good LabVIEW programs look like and, in the process, will teach you how to "think" in LabVIEW.

The first chapter covers everything you need to know to read and understand the programs used, so even absolute beginners can benefit from this book. When new LabVIEW functions are first introduced, they are accompanied by excerpts from the LabVIEW help files for ease of reference. Beginning and intermediate LabVIEW programmers will find design patterns that they can use right away in their own programs. Beginning programmers will have the greatest benefit, because the material in this book is not easily found in other sources. The beginner will "jump start" their understanding of dataflow and produce better code faster than they would otherwise. Intermediate programmers will benefit as well and hopefully will learn techniques that inspire them to revisit and revise their previous work. LabVIEW programmers preparing for the Certified LabVIEW Developer (CLD) exam will especially find this book useful, since it contains full solutions for all previously published CLD exams.

Intermediate and advanced users who have not yet made the leap to LabVIEW classes will also find this book useful. LabVIEW classes are used extensively in the last third of this book. Beginning programmers will perhaps see the greatest benefit because the material in this book is not easily found in other sources. This book provides examples of how to take advantage of data encapsulation, inheritance, composition, and dynamic dispatch when using LabVIEW classes. Intermediate and advanced users will also appreciate the discussion of parallel loop architectures and race conditions, as well as the example of how to use LabVIEW classes to extend the language.

How This Book Is Organized

This book follows a "teach by showing, learn by doing" approach. National Instruments offers multiple levels of certification for aspiring LabVIEW programmers. The Certified LabVIEW Developer (CLD) is the mid-level LabVIEW certification. Certified LabVIEW Developers must pass a four-hour programming exam in which they demonstrate an ability to create functional and well-documented LabVIEW code.

Over the years, National Instruments has published seven CLD exams. One of these is a controller for a car wash, which is used as a demonstration project throughout this book. The chapters in this book tend to follow a three-part pattern:

- A chapter introduces a new concept and demonstrates its use in a simple program (the `Multitest` VI)
- The next chapter shows how this concept can be used in a more complex program, such as the car wash controller
- A problem set is presented in the following chapters, allowing you to implement the new concept using problems based on the six other released CLD exams

This process is repeated throughout the book, focusing on core LabVIEW structures and functions that are the basis of all effective LabVIEW programs.

- Chapter 1 introduces all of the LabVIEW basics needed to read and understand the programs presented in this book. Chapter 2 introduces functional specifications and state diagrams. The functional specifications of the `Multitest` VI and of the car wash controller are presented. Problem Set 1 (Chapter 3) covers the basic LabVIEW material and asks you to develop sub-VIs used in the later problem sets.
- Chapter 4 introduces the classic state machine architecture and demonstrates it using the simple `Multitest` VI. Chapter 5 introduces action engines (also known as functional global variables). Chapter 6 uses an action engine to create a timer for the car wash controller. Problem Set 2 (Chapter 7) gives you an opportunity to create action engines for some of the other CLD exams. Chapter 8 demonstrates how to use the state machine architecture and an action engine to create a fully functioning solution for the car wash controller. Problem Set 3 (Chapter 9) gives you an opportunity to create state machine solutions for some of the other CLD exams.
- Chapter 10 introduces event structures and demonstrates their use in the `Multitest` VI. Chapter 11 upgrades the state machine implementation of the car wash controller to include an event structure. Problem Set 4 (Chapter 12) explores the use of event structures in some of the other CLD exams.
- Chapter 13 introduces queues and demonstrates their use in the `Multitest` VI. Chapter 14 upgrades the event-driven car wash controller to include the use of a queue.
- Chapter 15 applies queues, event structures, and the state machine architecture to a data acquisition program. This chapter discusses how data acquisition tasks

are created and used in a LabVIEW program. A program is developed that allows the user to null, calibrate, and read a strain gage in a quarter-bridge circuit. Problem Set 5 (Chapter 16) explores the use of queues in some of the other CLD exams.

- Chapter 17 explains how to use the material presented in the previous chapters to prepare for the Certified LabVIEW Developer exam. Problem Set 6 (Chapter 18) and Problem Set 7 (Chapter 19) provide full walk-throughs of two of the published CLD exams that were not used in the prior problem sets.
- Chapter 20 expands on the topic of state machines with arguments and string-based queues. These are demonstrated using the `Multitest` VI. Chapter 21 upgrades the queued car wash controller by adding arguments and a string-based queue. Problem Set 8 (Chapter 22) explores the use of arguments in some of the other CLD exams.
- Chapter 23 introduces LabVIEW classes and discusses data encapsulation using classes. The car wash controller with arguments is upgraded by replacing the action engine with a LabVIEW class. Problem Set 9 (Chapter 24) explores the use of LabVIEW classes to encapsulate data in some of the other CLD exams.
- Chapter 25 introduces the concepts of inheritance and composition in LabVIEW classes. The car wash controller with LabVIEW classes is upgraded to take advantage of inheritance and composition. Problem Set 10 (Chapter 24) explores the use of inheritance and composition in some of the other CLD exams.
- Chapter 27 introduces parallel loop architectures and demonstrates the producer–consumer design pattern using the `Multitest` VI. Chapter 28 discusses race conditions in parallel loop architectures and how to combat them, demonstrating these techniques using the car wash controller. Chapter 29 uses LabVIEW classes to extend the language and create a new form of queue that can be used to defeat the race conditions shown in Chapter 28. Problem Set 11 (Chapter 30) explores the use of parallel loop architectures in some of the other CLD exams.
- Chapter 31 provides a look back at the material covered in this book. The appendices present the functional specifications of the CLD exams used in the problem sets, as well as providing full solutions to all of the exercises in the problem sets.

The website that accompanies this book (http://www.ntspress.com/publications/effective-labview-programming/user-resources/) contains all of the programs presented in the main body of the text as well as solutions to the exercises presented in the problem sets. The programs (or VIs as they are called in LabVIEW) can be found in the "All VIs" folder on the site. This folder contains subfolders dedicated to all of the major projects used in this book: the ATM controller, the Boiler controller, the Car Wash controller, the Car Wash 2 controller, the DAQ State Machine, the Multitest State Machine, the Security System controller, the Sprinkler controller, and the Traffic Light controller. Each of these folders contains all of the files related to the various versions of these projects presented in this book.

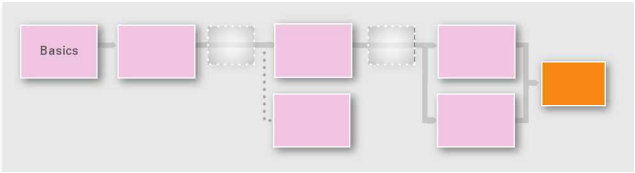
LabVIEW Projects also have been provided to make it easier to navigate the codes in the "All VIs" folder. A LabVIEW Project is a library that can be used to provide a virtual organization of VIs and other material without changing the location of the codes. The LabVIEW Projects are used to present selected VIs and all of their supporting sub-VIs and type definitions in one easily navigated package.

The website contains four folders of LabVIEW Projects. The LabVIEW Projects in the "Problem Set Solutions" folder contain the solutions to the exercises presented in this book. There is one LabVIEW Project for each problem set. The individual exercise solutions are contained in labeled virtual folders in the project. Clicking on the entries in these folders will automatically open the solutions to the corresponding exercises.

The website also contains a folder of LabVIEW Projects called "VIs Organized By Chapter." The LabVIEW Projects in this folder are each dedicated to a separate chapter in the book. If you would like to examine a VI described in a chapter, navigate to the LabVIEW Project for that chapter and double-click on the desired VI to open it.

There is also a folder called "VIs Organized By Topic." This folder groups all of the example codes and the problem set solutions by topic. For example, the "Classic State Machine" subfolder contains LabVIEW Projects dedicated to classic state machine implementations of the Boiler controller, the Car Wash controller, the Car Wash 2 controller, the Multitest State Machine, the Security System controller, and the Traffic Light controller. This set of LabVIEW Projects is very useful to see how a specific feature or pattern can be used in a variety of programming projects.

The website also contains a folder of LabVIEW Projects called "Practical CLD Exam Solutions." These projects are dedicated to VIs that satisfy the requirements of the Certified LabVIEW Developer exam at a level of complexity that is appropriate to the four-hour time limit of the exam. The original CLD exam functional specification documents also can be found on the website in the "Original CLD Exam Documents" folder.



1

LabVIEW Basics

LabVIEW is a graphical programming environment that can be used to quickly create applications with professional user interfaces. It is widely used by engineers, scientists, and students to develop measurement, test, and control applications. LabVIEW is well-suited to this purpose. The user interface is created automatically as the code is developed and easily can be customized. LabVIEW has unrivaled integration with thousands of hardware devices and comes with hundreds of built-in libraries for analysis and data visualization. LabVIEW is scalable across many different platforms, targets, and Operating Systems (OSs). It is also inherently parallel, and applications can scale automatically to take advantage of multicore CPUs with little or no additional programming effort. Technically speaking, "LabVIEW" is the name of the graphical development environment and "G" is the name of the graphical programming language used in this environment. But since G code can only be developed from within the LabVIEW environment, the term "LabVIEW" is commonly used to refer to both the development environment and the programming language. This is how the term will be used in this book.

LabVIEW stands for Laboratory Virtual Instrumentation Engineering Workbench. As its name implies, LabVIEW is most often used for measurement, test, and control applications, but it is a full-featured programming language. It has the standard elements of a typical programming language: a wide range of data types, loops, conditional statements, data structures, arrays, string manipulation functions, file I/O functions, etc. LabVIEW is event-driven and object-oriented. LabVIEW compiles directly to machine code. LabVIEW compiles in the background: it does not need to be explicitly compiled in a separate step.

But even though LabVIEW has many things in common with typical programming languages, it has two unique aspects that make it different from almost any other language: a graphical programming interface and a dataflow programming paradigm. Each of these aspects has a distinct learning curve. The graphical nature of LabVIEW is its most obvious difference from other programming languages. LabVIEW is not text-based. A LabVIEW program (called a "virtual instrument" or VI) consists of a front panel viewable by the user and a block diagram viewable by the developer. The block diagram contains icons and terminals connected by wires. VIs are created by

selecting function icons from a palette and then dropping and dragging them on the block diagram. The function icons have input and output terminals, indicating points where data enters or exits the functions represented by the icons. The cursor is used as a wiring tool, connecting the icons with paths that represent the flow of data through the program.

The flow of data from one function to the next dictates the order of execution of the functions on the block diagram. This is known as dataflow, and the dataflow paradigm is the second unique aspect of LabVIEW. In most text-based languages, the order of execution depends on the order of the text in the program. When one line is finished, the next line executes in a serial fashion. This is not the way that LabVIEW works. LabVIEW is inherently parallel. Data travels along wires in parallel. Functions execute as soon as they have all of their input data. Functions that have no dataflow dependence automatically execute in parallel with each other. Most text-based languages also use named variables that can be called anywhere within their scope simply by invoking their name. This is an alien concept in LabVIEW because dataflow is inherently anonymous. Data travels from one function to another along wires without ever being named. Data is not stored unless the developer makes an effort to store it.

These two aspects of LabVIEW make it easy to learn but difficult to master. Graphical programming languages are easy to learn because graphical languages don't have the syntax issues of textual languages. Functions are laid out in palettes, and it is easy to browse through the palettes to select the desired function. Function inputs and outputs are easily read by hovering the mouse over the icon terminals. The order of inputs and ouptuts does not need to be memorized, and LabVIEW prevents wires of one data type from being connected to terminals of a different data type. Constructing LabVIEW code is quick and easy once the basic mechanics of navigating the graphical interface have been mastered.

The dataflow aspect of LabVIEW is another story. Simple VIs with simple dataflow present little difficulty to the beginning LabVIEW programmer. But when a program becomes even moderately complex, it can suddenly become very difficult for beginning and intermediate LabVIEW programmers to create effective VIs that are easy to read, easy to maintain, and easy to scale. The problem seems to lie with the graphical nature of LabVIEW because the resulting VIs look like a pile of spaghetti (see Figure 0.1 for a particularly egregious example). But the problem really lies in a lack of understanding of the dataflow nature of LabVIEW. Effective dataflow programming requires different techniques and different architectures than are found in traditional text-based programming languages.

LabVIEW has been in use since 1986, but there have not been many books published about LabVIEW programming. Most of the books that have been published address the graphical programming learning curve and focus on the mechanics of creating LabVIEW applications. This book is different. The focus of this book is dataflow thinking. This book will help you internalize the techniques and style that will help you become an effective LabVIEW programmer.

Before we can focus on dataflow thinking, we need to review the basic elements of LabVIEW programming. In order to keep our review of LabVIEW brief and focused, two guiding principles will be followed in this chapter. First, we will only

review the elements of LabVIEW that are used in this book. This book does not cover advanced math functions, data acquisition functions, instrument control, signal processing functions, etc. The programs in this book focus on core LabVIEW structures and functions that are the bedrock of any LabVIEW program, so that will be the focus of this chapter. Second, the review in this chapter will focus primarily on *reading* LabVIEW code rather than *writing* it. The purpose of this book is to demonstrate what effective LabVIEW code looks like, it is not a step-by-step tutorial on how to build LabVIEW code detailing every mouse-click and menu-pick.

This book is self-contained—to a point. If you are a new LabVIEW programmer, this chapter will teach you everything you need to read and understand the programs presented in this book. But if you want to build these programs yourself, then you will likely need some outside help. LabVIEW has many built-in tutorials and example programs, and individual functions have extensive help files. There are also many books and online resources that can help you master LabVIEW's graphical interface. If you are already an experienced LabVIEW user, you can skip this chapter and move on to the next.

1.1 Anatomy of a VI

LabVIEW programs are called "virtual instruments" or "VIs," because they are intended to look and feel like physical measurement instruments, such as digital multimeters or oscilloscopes. Every VI has three core elements: a front panel, a block diagram, and an icon/connector pane. These elements are created when a new VI is created. The front panel and block diagram are created as two separate windows. The front panel contains the user interface. Knobs, buttons, dials, graphs, and text displays are all contained in the front panel. The block diagram contains the code. When a user interface element such as a button or a graph is placed on the front panel, a corresponding terminal appears on the block diagram. These terminals then can be used in the block diagram as sources or sinks of information. Data flows from controls and to indicators.

The icon/connector pane appears in the upper-right corner of the front panel. An icon is a developer-generated image that represents a VI. If the VI is used as a sub-VI in a larger program, the icon of the sub-VI appears on the block diagram of the calling VI. The connector pane is a template of the inputs and outputs of the VI. It defines how the wiring terminals of the icon of a VI are mapped to the front panel controls and indicators and the VI. The icon and connector pane appear side-by-side in the upper-right corner of the front panel, and the icon alone appears in the upper-right corner of the block diagram.

1.2 A Simple Example: `Hello World.vi`

An example of a "Hello, world!" program can be seen in Figure 1.1. The front panel is on the left of the figure, and the block diagram is on the right. The connector pane and

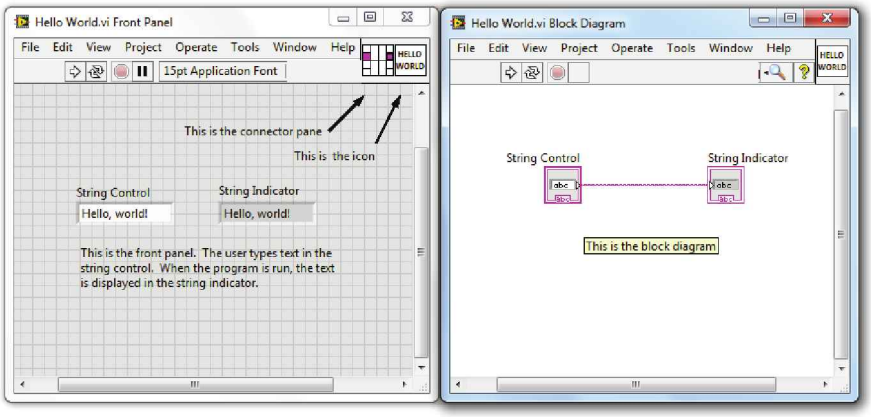


Figure 1.1: An example of a VI

icon are in the upper-right corner of the front panel. The purpose of this program is very simple. The user types text into the box on the front panel labelled *String Control*. When the program is run, the text typed by the user will appear in the box labeled *String Indicator*. If the user types `Hello, world!` into the control and then runs the VI, the indicator will display the text `Hello, world!`.

The user sees and interacts with the front panel when the VI is running. The code itself resides on the block diagram. The separation of the user interface from the code is one of LabVIEW's strengths. Functional user interfaces can be developed rapidly, since the front panel objects are all drag-and-drop and can be customized directly or programatically.

The front panel and the block diagram of a VI are connected by controls and indicators. Every control and indicator in a VI appears on both the front panel and on the block diagram. The default appearance of a selection of front panel controls is shown in Figure 1.2. Controls are shown on the left, and indicators are shown on the right. Controls are manipulated by the user to enter information into a VI, and indicators display information to the user. In both the front panel and block diagram of the "Hello, world" VI shown in Figure 1.1, the string control is on the left and the string indicator is on the right. Information entered into the control by the user is sent to the indicator via the wire that connects them in the block diagram.

1.3 Controls, Indicators, and Wires

Controls and indicators on the block diagram may be viewed as icons or terminals. Icons are larger and more detailed. Terminals are smaller and take up less area on the screen. The default view in LabVIEW is the icon view, and that is the view that will be used in the VIs in this book. However, the native LabVIEW help file system uses the terminal view when it discusses the inputs and outputs of functions. Since the

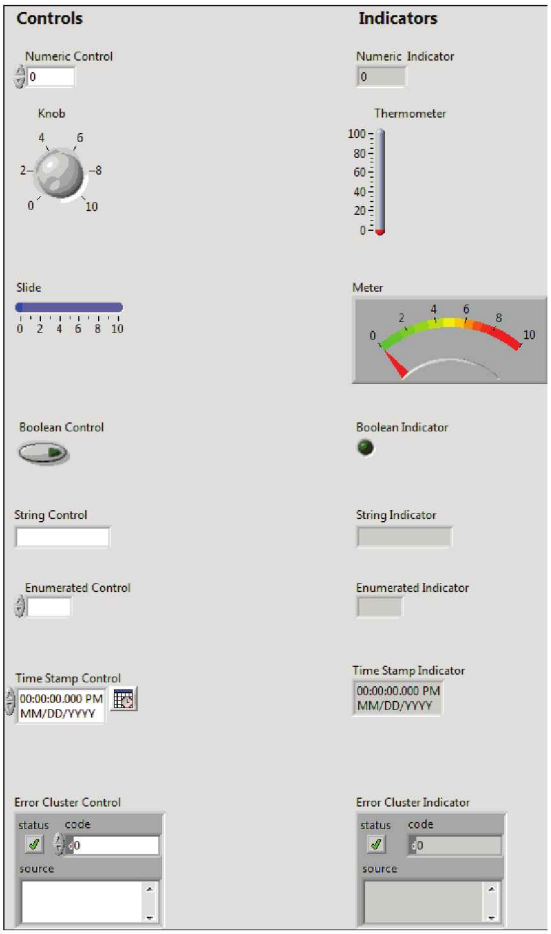


Figure 1.2: Default appearance of front panel controls and indicators in LabVIEW

help file figures in this book are based on the LabVIEW help files, they also use the terminal view.

Examples of the icon view and terminal view are shown in Figure 1.3. This figure shows two sets of block diagram terminals, with each set corresponding to the front panel objects shown in Figure 1.2. The icon view of the controls and indicators are on the left side of Figure 1.3, while the terminal view is on the right.

Icons have a visual cue of the front panel appearance of the item. The icon of a double-precision thermometer indicator has a small thermometer glyph, and the icon of a double-precision meter indicator has a meter glyph. Terminals have no visual cues about the appearance of the control or indicator on the front panel. For example, the numeric control, knob, and slide shown in Figure 1.2 are all double-precision numeric controls. The icons of these controls are all different, and they reflect the appearance of the control on the front panel seen in the left column of Figure 1.3. In

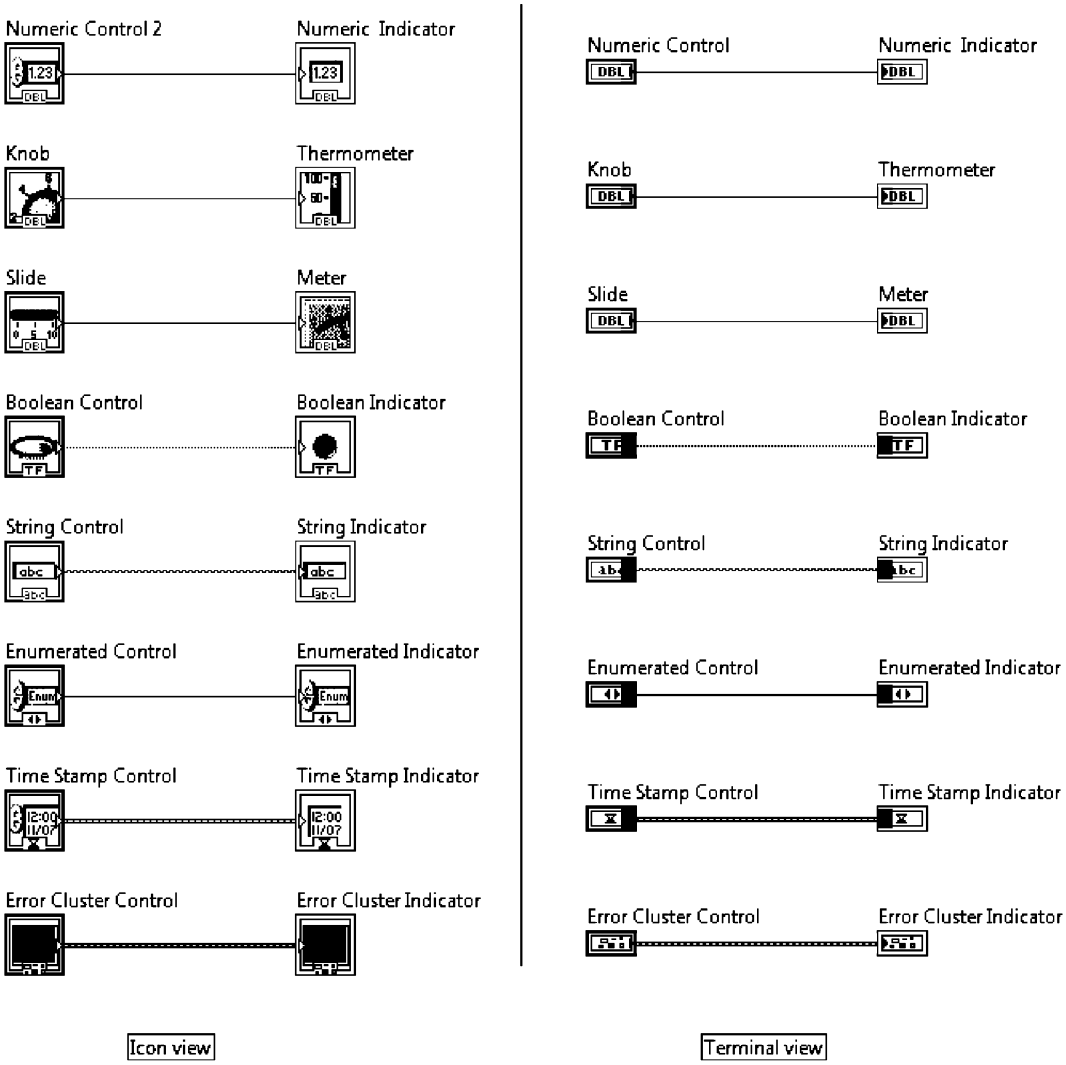


Figure 1.3: Icon and terminal view comparison of controls and indicators in LabVIEW

the terminal view, all of these controls are identical, as seen in the right column of column of Figure 1.3.

Whether you use icon view or terminal view, each control and indicator in a VI must have a unique label. The label is visible by default in both the front panel and block diagram. Controls and indicators also have optional captions that can only be displayed on the front panel. Long, descriptive captions are useful for improving readability of front panel, while short labels conserve space on the block diagram.

The data type of a terminal or wire on the block diagram is indicated by its color. Orange represents floating-point data, while blue represents integer data. Booleans

are green, and strings are pink. For example, the terminals and connecting wire in the "Hello, world!" example shown in Figure 1.1 are pink, indicating that they have all have the string data type. Other examples of data types and their associated colors are shown in Figure 1.3.

Controls, indicators, and other terminals can be wired only if the data type of the wire matches the data type of the terminal. If they don't match, the wire will break, and the VI won't run. Some data types, such as the numeric data type, are available in a variety of representations. Floating-point data types are available in extended precision, double precision, and single precision. Integer data types can be signed or unsigned and can be 8-bits, 16-bits, 32-bits, or 64-bits in width.

Wires and terminals with different numeric representations can be wired together without breaking the VI, but LabVIEW will automatically convert the data behind the scenes. This automatic data conversion is indicated on the block diagram by a coercion dot, which is illustrated in Figure 1.4. The top portion shows a code fragment that calculates the area of a rectangle. The length of the rectangle is represented by a double-precision, floating-point number (DBL), and the width is represented by a 32-bit signed integer (I32). The two numbers are multiplied together to calculate the area of the rectangle, which is displayed by a double-precision, floating-point indicator.

Notice the small red dot where the width is wired to the **`Multiply`** function. This is a coercion dot. A coercion dot indicates that there is a mismatch between the data type of the input wire and the data type expected by the input terminal and that an automatic data type conversion has taken place. In Figure 1.4, the I32 value will be converted to a double-precision, floating-point value to match the representation of the Area indicator.

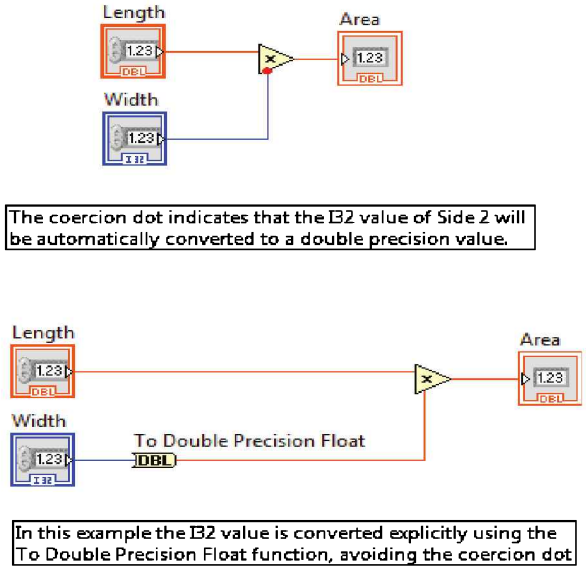


Figure 1.4: Coercion dots indicate an automatic conversion of data type

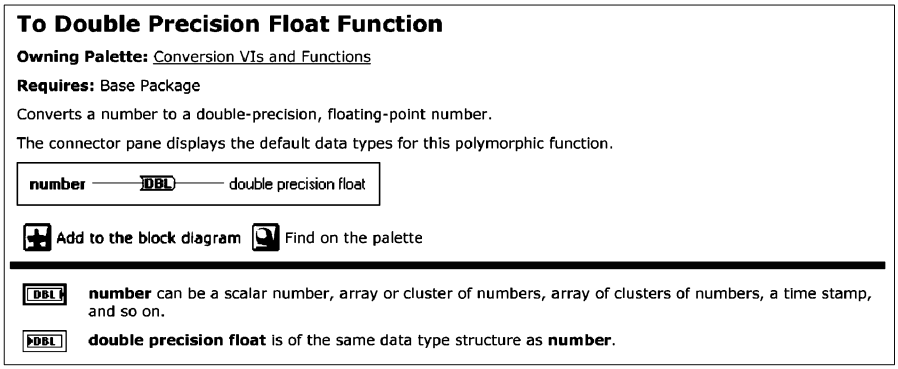


Figure 1.5: Help file for the `To Double Precision Float` function

Coercion dots should be avoided, because potentially important decisions about data type conversions are being left to LabVIEW rather than to the developer. A better approach is shown in the bottom half of Figure 1.4. In this version of the code, the conversion from I32 to double precision is performed explicitly by the **`To Double Precision Float`** function whose help file is shown in Figure 1.5. The data type of the wire now matches the data type of the terminal, avoiding a coercion dot at the **`Multiply`** input terminal.

Boolean controls have a special property known as *mechanical action*. In the physical world, Boolean controls correspond to switches and buttons, and these devices have different mechanical behaviors. When you turn a light switch on or off, the light responds as soon as the switch is flipped and the switch stays in its new position until it is flipped again. This corresponds to the LabVIEW mechanical action "switch when pressed." A buzzer will sound only as long as you are holding the button down and will stop as soon as you release it. This mechanical action is called "switch until released." In the Windows operating system, if you click and hold the mouse button over the red box with the white "X" in the upper-right corner of a window, nothing will happen. But as soon as you release the mouse button, the window will close. This is known as "switch when released." These mechanical actions can be incorporated into LabVIEW Boolean controls as well. Boolean switches in LabVIEW can be assigned any of these mechanical actions. These actions can take place immediately (*switching actions*), or they can be intentionally delayed to take place only after the contents of the Boolean control have been read by LabVIEW (*latching actions*).

1.4 Dataflow

In a dataflow language, the flow of the data itself determines the execution order of the program. The basic rule of dataflow is that any program element may execute once it has all of its inputs. The dataflow of the "Hello, world!" program shown in Figure 1.1 is very simple. When the VI starts, the control can execute because it has

no inputs, but the indicator cannot immediately execute because its input data is undetermined. The data from the string control then travels along the wire from the string control to the string indicator. When the indicator receives its input data, it executes immediately and displays the text it received from the control. The VI is laid out from left-to-right, but this is simply a convention. The data doesn't automatically travel from left-to-right—the rules of dataflow determine the program execution.

Figure 1.6 shows a VI that calculates the area of a square given the length of the side of the square. This VI consists of three elements: a control containing the length of the side of the square, the LabVIEW **`Square`** function, and a numeric indicator. Of the three program elements, only the numeric control has no inputs. The control executes immediately when the VI starts. The **`Square`** function and the numeric indicator both have inputs so they do not execute immediately when the VI starts. The data from the control travels along the wire to the **`Square`** function. When the data arrives at the input to the **`Square`** function, the function executes. The result of this operation then travels along the second wire to the numeric indicator. The indicator then executes, displaying the area of the square.

Figure 1.7 shows a slightly more complex VI. This VI calculates the length of the hypotenuse c of a right triangle given the lengths of the two perpendicular sides a and b. It does this using the Pythagorean theorem, which states that the length of the hypotenuse is equal to the square root of the sum of the squares of the other two sides, or $c = \sqrt{a^2 + b^2}$.

When the VI begins, the only items that can execute are the controls containing the values of a and b. The other items on the block diagram are waiting for input data. The two **`Square`** functions execute once they receive their input data from the a and b controls. The **`Add`** function executes once it receives data from the two

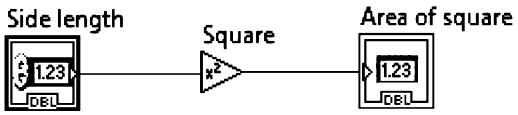


Figure 1.6: Calculating the area of a square

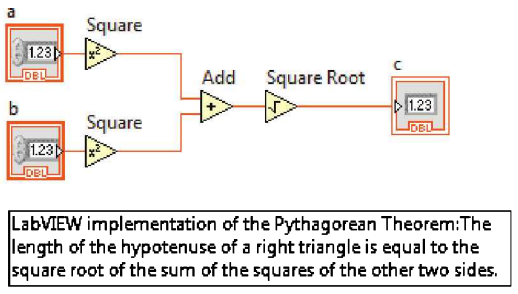


Figure 1.7: Calculating the hypotenuse of a right triangle

Square functions. The **Square Root** function can only execute after it receives its input data from the **Add** function. The indicator then executes and displays the value passed to it by the **Square Root** function.

This VI is interesting, because it has two parallel wires feeding the **Add** function. What is the exact order of execution of the functions in these two wire paths? Dataflow dictates that the **Square** functions must execute after the numeric controls that feed them, and that both **Square** functions must execute before the **Add** function. But that is all we can say. Any of the following scenarios are possible:

- The control containing the value of a could execute first, followed by the **Square** function in the top wire path. The control containing the value of b could execute next, followed by the **Square** in the bottom wire path.
- The control containing the value of b could execute first, followed by the **Square** function in the bottom wire path. The control containing the value of a could execute next, followed by the **Square** in the top wire path.
- The control containing the value of a could execute first, followed by the control containing the value of b. Then the top **Square** function could execute, followed by the bottom **Square** function.
- The control containing the value of a could execute first, followed by the control containing the value of b. Then the bottom **Square** function could execute, followed by the top **Square** function.
- The control containing the value of b could execute first, followed by the control containing the value of a. Then the top **Square** function could execute, followed by the bottom **Square** function.
- The control containing the value of b could execute first, followed by the control containing the value of a. Then the bottom **Square** function could execute, followed by the top **Square** function.

Now let's assume that the VI shown in Figure 1.7 is executing on a multicore computer and that operations can happen simultaneously if they run on different cores. This allows even more possibilities:

- The two controls execute simultaneously, then the two **Square** functions execute simultaneously
- The two controls execute simultaneously, then the top **Square** function executes, followed by the bottom **Square** function
- The two controls execute simultaneously, then the bottom **Square** function executes, followed by the top **Square** function
- The control containing the value of a executes, the top **Square** function and the control containing the value of b execute simultaneously, and then the bottom **Square** function executes
- The control containing the value of b executes, the bottom **Square** function and the control containing the value of a execute simultaneously, and then the top **Square** function executes

- The control containing the value of a executes, the control containing the value of b, and then both **Square** functions execute simultaneously
- The control containing the value of b executes, the control containing the value of a, and then both **Square** functions execute simultaneously

This gives us thirteen possible ways to execute the VI shown in Figure 1.7 but all of them lead to the same result. Which one does LabVIEW actually execute? We don't know, but it doesn't matter, since they all give the correct result. LabVIEW is free to execute any of the thirteen versions, selecting the one that gives optimal performance.

This is the beauty of the dataflow model. It is inherently parallel. On a single-core processor, LabVIEW would determine which parallel operation happened first. On a multicore processor, the parallel operations might be carried out simultaneously. In LabVIEW, operations coded in parallel are automatically carried out in parallel. In the hypotenuse VI shown in Figure 1.7, we specify just enough dataflow to ensure that we get the right answer. The sides must be squared before they are added, then the square root of the sum must be taken. We don't care which side is squared first, and we don't care in what order the controls are read. The dataflow model used by LabVIEW lets the developer specify the important aspects of the flow of data in a problem. The unimportant elements of execution order are left to LabVIEW to execute as it sees fit.

1.5 A More Substantial Example: `Clean Up Folder.vi`

LabVIEW VIs can be printed to assist in the production of documentation. Since LabVIEW is graphical, printing a VI can generate a large number of image files as well as text files. LabVIEW generates image files in bitmap format with the file extension `bmp` and rich text format files with the extension `rtf`.

Imagine that you were documenting a LabVIEW VI and that you store all of these image and text files in a documentation folder and subfolders. When you are finished documenting the VI, you are left with with a large number of `bmp` and `rtf` files that are no longer needed. You would like to develop a `Clean Up Folder` VI that would search through the documentation folder and all of its subfolders and eliminate the unwanted files. The VI would accept a list of file extensions to target, and it would also accept input directing it to either delete the files or move them to another location.

The `Clean Up Folder` VI is developed along the following lines. The main VI searches through the target folder and its subfolders and generates a list of all the files they contain. A sub-VI then examines each file in turn to see if it has one of the targeted file extensions. If it does, a second sub-VI either moves or deletes the file as indicated by the user.

The front panel of the `Clean Up Folder` VI is shown in Figure 1.8. The front panel consists of four controls and no indicators. Counting from the top down, the first and third controls are path controls that allow the user to specify a path to a folder

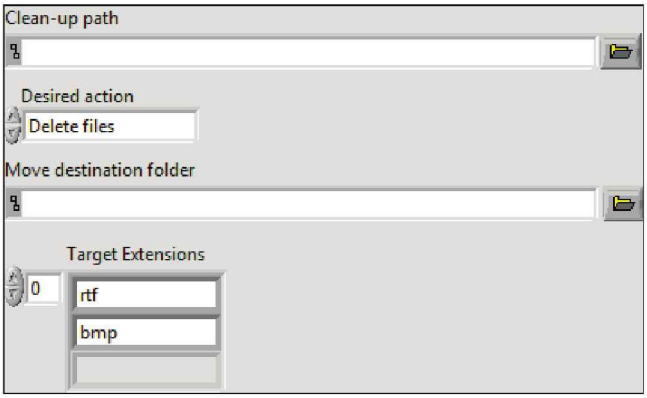


Figure 1.8: Front panel of the `Clean Up Folder` VI

or file. The path can be entered manually or can be selected via a navigation window that is brought up by clicking on the folder icon at the far right of the control.

Enumerated Controls

The second control from the top of Figure 1.8 is an enumerated control. The enumerated data type, also known as an enum, can be thought of as a pre-defined set of n items that are mapped to the set of integers $0 \ldots n-1$. Enums are useful because they improve program readability. When an enum is used in a VI, the items are displayed to the human reader, but the numbers mapped to the items are used by the VI. In the `Clean Up Folders` VI, an enumerated control is used on the front panel to allow the user to select one of the two possible clean-up actions: *Move files* and *Delete files*. These actions are mapped to the integers 0 and 1 respectively, as shown in Figure 1.9.

Type Definitions

The data type of an enumerated control is defined by its set of items. Enumerated controls with different sets of items have different data types, and enumerated controls that share the same set of items have the same data type. It is good programming practice to define the data type of an enumerated control by creating a type definition file.

A type definition allows the user to create a special kind of customized control. When a control is saved as a type definition, all copies of that control are referenced back to the type definition file. If the type definition file is changed, then all copies of the control created from that type definition are automatically updated to match the changed type definition file. This is a powerful tool for maintaining code, as type definitions help future-proof code. Changes made to the type definition file are automatically propagated to all copies of the control, reducing both the maintenance

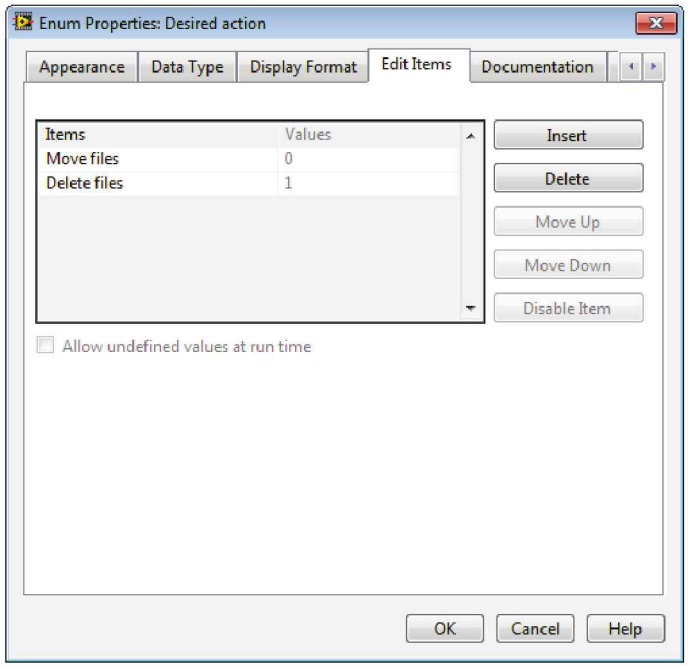


Figure 1.9: Defining the items in an enumerated control

workload and the chance for making errors. A block diagram item that is tied to a type definition file is displayed with a black triangle in the upper-left corner icon.

You can create three kinds of controls in LabVIEW: custom controls, type definitions, and strict type definitions. Custom controls are used to enforce the appearance of controls only. Copies of custom controls will all look alike, but they are not tied to the original control file. If the custom control file is changed, existing instances of the control are not updated. Type definitions enforce the data type of the control. Copies of the control will all have the same data type as the original, and all copies update if the original file is updated. However, individual copies of the type-defined control can have different appearances. Strict type definitions enforce both the data type and the appearance of the control. Strict type definitions combine the effects of the custom control and the type definition.

In the `Clean Up Folders` VI, the enumerated control containing the *Move files* and *Delete files* elements will be saved as a type-definition file. Even though there will be only one copy of this control in the VI, it is good programming practice to always create type definitions for custom enumerated controls.

1.6 Arrays and Clusters

The control containing the list of file extensions in Figure 1.8 is an array control. An array is an ordered group of elements that share the same data type the same

dataflow direction. Array elements must be all controls, all constants, or all indicators. An individual element in the array is accessed by its index, which is an integer that specifies the location of the element in the array. The number of indices needed to specify an element is called the dimension of the array. Indices in LabVIEW are zero-based, where the first item in any dimension of an array has the index 0. If a one-dimensional array has ten elements, the indices of the array run from 0 to 9.

The block diagram appearance of arrays is illustrated in Figure 1.10. Arrays are easily distinguishable in the icon view, because the central glyph of the icon is replaced with an array glyph. The data type of the array can be determined by the color of the array icon. In terminal view, arrays have distinctive borders and brackets

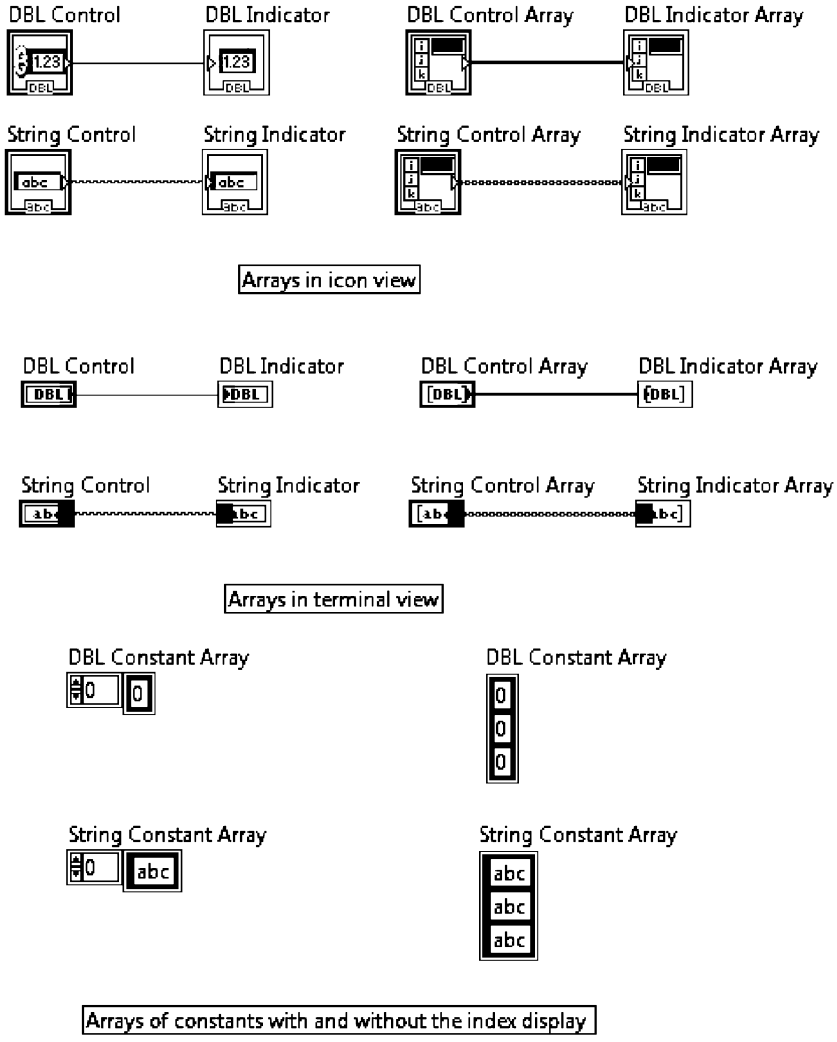


Figure 1.10: Examples of array icons for double-precision and string controls and indicators

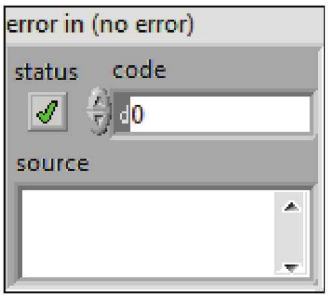


Figure 1.11: The standard LabVIEW error cluster

around the central text of the terminal. In both icon and terminal view, wires carrying one-dimensional arrays are thicker than wires carrying scalars. Arrays of constants are also shown in Figure 1.10. The array constant can be stretched to show multiple elements of the array at once. An index display can be made visible if desired to help navigate through the array constant. Arrays of constants only can be displayed on the block diagram of a VI.

Arrays are used to aggregate data with the same data type. Clusters are used to aggregate data with mixed data types. Arrays and clusters are similar in many ways. Like arrays, the items in a cluster must either be all controls or all indicators. Like arrays, clusters allow multiple pieces of data to travel on one wire simultaneously. Unlike arrays, the number and data types of a cluster are fixed when the cluster is created. Arrays are open-ended, and their length can change as a VI runs. This is not true for clusters. The values stored in a cluster can be overwritten as a VI runs, but the number of elements and their data types cannot change.

The front panel of the `Clean Up Folders` VI doesn't contain any cluster controls or indicators, but there is a very important cluster that will be used extensively by this VI: the error cluster. The LabVIEW error cluster is shown in Figure 1.11. The error cluster consists of three elements: a status Boolean control, a 32-bit integer error code numeric control, and an error source string control. The status Boolean indicates if an error is present, the integer contains the error code, and the source string indicates where the error occurred. If the status Boolean is True, then an error has occurred. The error code and the source of the error are recorded in the remaining two elements of the cluster. The error cluster also can be used to record a warning. If the status Boolean is False but the error code is not zero, a warning has occured. The error code and source string are used to store both the warning type and the source of the warning.

In order to read a cluster element, it must be unbundled from the cluster. LabVIEW provides two functions for doing this: **`Unbundle`** and **`Unbundle By Name`**. These two functions are illustrated in Figure 1.12. In the top portion, the error cluster elements are accessed using the **`Unbundle`** VI. Notice that all three elements are unbundled and only the data types of the elements are shown. The labels of the individual elements do not appear. The lower portion of the figure accesses the cluster elements using the **`Unbundle By Name`** function. This function has two advantages over the

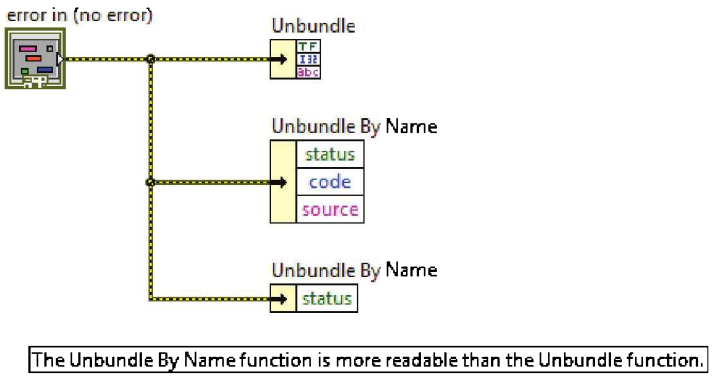


Figure 1.12: Using the `Unbundle` and `Unbundle By Name` functions

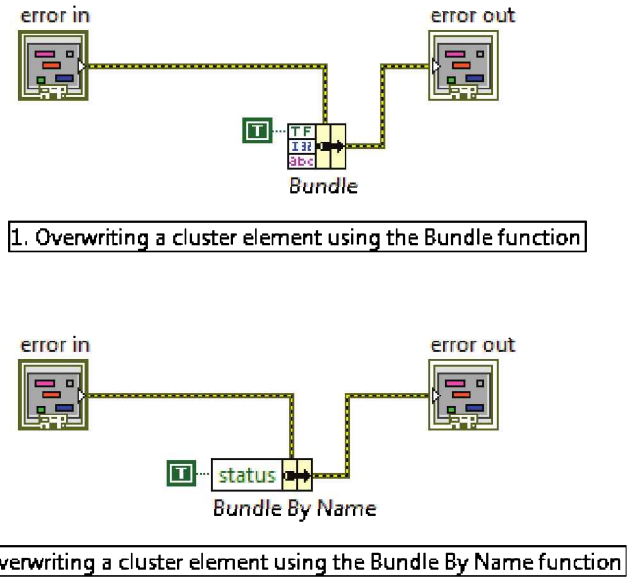


Figure 1.13: Using the `Bundle` and `Bundle By Name` functions

`Unbundle` function. First, the labels of the individual cluster elements are displayed instead of the data types. This makes the VI much easier to read, especially if a cluster contains multiple elements with the same data type. Second, the **`Unbundle By Name`** function can be resized to display only the items that need to be unbundled. This also improves code readability.

Data can be written to a cluster with the use of a bundling function. LabVIEW provides two functions shown in Figure 1.13 for this purpose: **`Bundle`** and **`Bundle By Name`**. These functions behave very much like their unbundling counterparts.

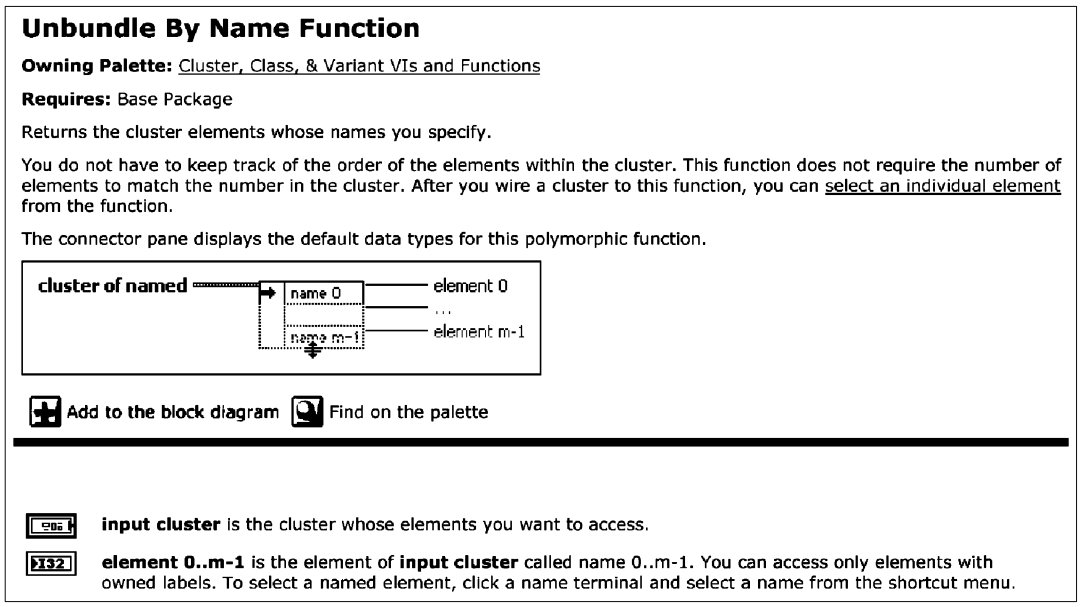


Figure 1.14: Help file for the `Unbundle By Name` function

The **`Bundle`** function accesses all of the cluster elements at once and displays only the data types of the elements. The **`Bundle By Name`** function displays the element labels and can be resized to access only the desired elements.

The bundling functions only can be used on pre-defined clusters. They do not create new clusters: they overwrite the data of existing clusters. The existing cluster must be wired to the central input of the bundling functions, as seen in Figure 1.13. Cluster elements that are not specifically overwritten by the bundle inputs retain the values they had in the cluster wired to the central input.

It is good LabVIEW style to use **`Unbundle By Name`** and **`Bundle By Name`** instead of **`Unbundle`** and **`Bundle`** whenever you can. It saves space on the block diagram and improves readability. The help file for the **`Unbundle By Name`** function can be found in Figure 1.14, and the help file for the **`Bundle By Name`** function can be found in Figure 1.15.

1.7 Creating Sub-VIs

Before developing the block diagram of the `Clean Up Folder` VI, we will develop the necessary sub-VIs. Sub-VIs are VIs that are used on the block diagrams of other VIs. They are analogous to subroutines in text-based programming languages. The icon and connector pane of a sub-VI are vitally important. The icon identifies the sub-VI when it appears on the block diagram of another VI. The connector pane dictates how the wiring terminals on the icon are mapped to the front panel controls and indicators of the sub-VI.

Bundle By Name Function

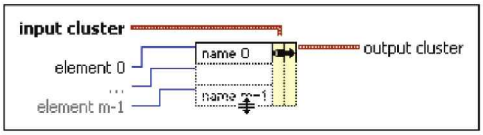
Owning Palette: Cluster, Class, & Variant VIs and Functions

Requires: Base Package

Replaces one or more cluster elements. This function refers to cluster elements by name instead of by their position in the cluster.

After you wire the node to an input cluster, right-click the name terminals to select elements from the shortcut menu. You also can use the Operating tool to click the name terminals and select from a list of cluster elements. All inputs are required. The connector pane displays the default data types for this polymorphic function.

Details



Add to the block diagram Find on the palette

input cluster is the cluster whose elements you want to replace. At least one element of **input cluster** must have an owned label. You must always wire the **input cluster** terminal.

element 0..m-1 are the elements of **input cluster** you want to replace by name. You only can replace elements with owned labels. Select the correct **element 0..m-1** by clicking a name terminal and selecting a name from the shortcut menu.

output cluster is the cluster whose value is the resulting cluster.

Bundle By Name Details

When you use the Bundle by Name function on nested clusters, you can display both the element name and the owning cluster name for elements in the nested cluster by right-clicking the function and selecting **Show Full Names**. This is useful when element names in nested clusters are identical or similar.

Figure 1.15: Help file for the `Bundle By Name` function

When a new VI is created, it is given a default icon and a standard connector pane. The default icon can be seen in Figure 1.16. This figure shows the icon editor, which can be used to create a custom icon for the VI. It is good LabVIEW style to create a descriptive custom icon for every sub-VI. The icon editor has many graphics features that can be used to construct an icon: layers with transparency control, standard drawing tools, color palettes, many fonts, text style and size controls, and a built-in library of pre-constructed glyphs. In this book, though, a simpler approach is taken. Most of the icons in this book consist of black text on a white background surrounded by a black rectangle. Typical text settings can be seen in Figure 1.16. This style was chosen for simplicity and for its ability to be reproduced when printed in black and white. Most of the sub-VI icons used in this book are square, but icons with other shapes can be constructed. This will be discussed further in Chapter 14.

The inputs and outputs of a sub-VI are defined by the connector pane. The connector pane appears to the left of the icon in the upper-right corner of the front panel of a VI. There are many choices of connector pane patterns, as shown in Figure 1.17. Each pattern can be flipped and rotated, presenting the developer with a wide variety of connector pane possibilities.

There is a clear preference in the LabVIEW community for one particular connector pane pattern, the $4 \times 2 \times 2 \times 4$ pattern shown in Figure 1.18. This is the default

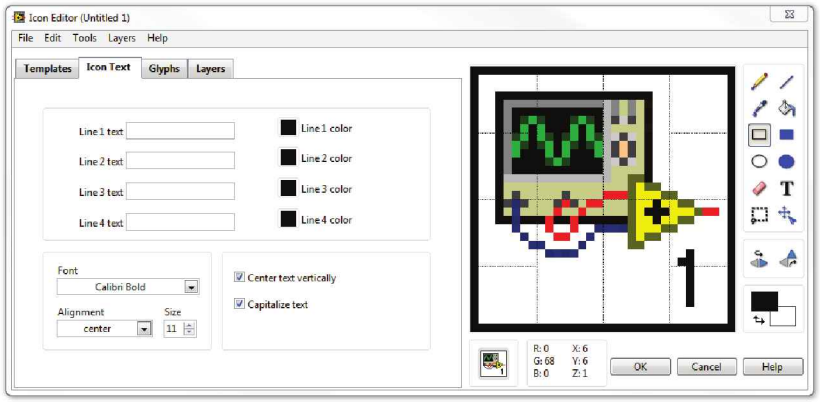


Figure 1.16: The Icon Editor

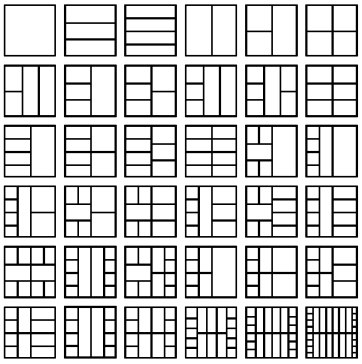


Figure 1.17: Connector pane patterns

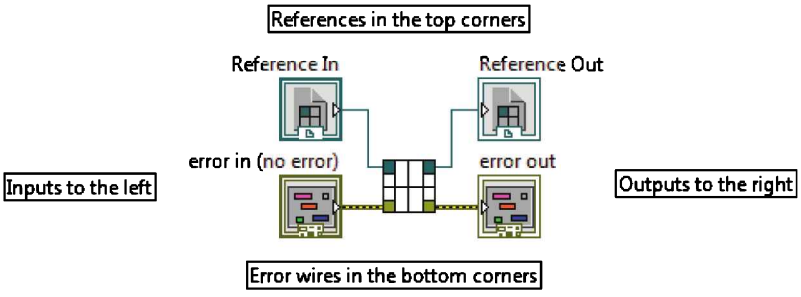


Figure 1.18: The standard 4 × 2 × 2 × 4 connector pane pattern

pattern for new VIs in LabVIEW. Experience has shown that this pattern strikes a good balance between terminal size and the number of terminals. Patterns with more terminals can be difficult to wire, and patterns with fewer terminals often get filled too quickly. Choose a connector pane pattern with more terminals than are needed. This allows room for adding inputs or outputs in the future without changing the connector pane. Changing the connector pane pattern of a sub-VI will cause its calling VIs to break. Adding new inputs or outputs to a sub-VI without changing its connector pane preserves its compatibility with its calling VIs.

Figure 1.18 shows the standard wiring convention for the preferred connector pane pattern. The terminals on the left of the connector pane are typically used as inputs, while the terminals on the right are typically used as outputs. The lower corner terminals are usually used for error input and output. The upper corner terminals are usually reserved for file paths or references. In data acquisition VIs, such as those discussed in Chapter 15, the upper corners of the connector pane are usually reserved for data acquisition task inputs and outputs.

Style Guide: Use the preferred connector pane if possible, and adopt the input/output, error terminal, and reference terminal conventions if at all possible. This will make your code easier to read by other LabVIEW users.

1.8 Case Structures

Many of the native LabVIEW functions have error input and output terminals. Most of these functions are designed to forego execution if an error is present at the error input terminal. The error is passed straight to the error output terminal instead. Most of the custom sub-VIs in this book follow the same design principle. In order to do this, the sub-VI must first determine if an error is present at its error input terminal. The status Boolean of the error cluster must be checked to see if it is True or False. This can be done using a case structure.

A case structure is a rectangular frame with a selector terminal, which is a small rectangle containing a question mark found on the left edge. The value wired to the selector terminal determines which case of the case structure will execute.

The case structure can be thought of as a stack of frames. Each frame corresponds to a possible value of the selector terminal, which is displayed in the case selector in the middle of the top edge of the case structure. For example, if a Boolean is wired to the selector terminal, the possible values of the Boolean are True and False. The case structure would then have two cases: *True* and *False*. The code in the *True* case executes if the value on the wire connected to the selector terminal is True, and the code in the *False* case executes if the value on the wire connected to the selector terminal is False. This is shown in Figure 1.19.

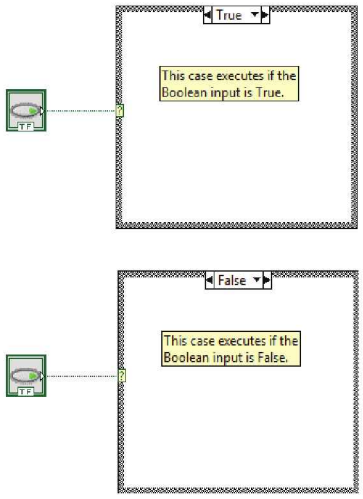


Figure 1.19: The LabVIEW case structure

The data type of the value wired to the selector terminal does not have to be Boolean. It also can be a string, integer, enumerated type, or error cluster. Booleans, enumerated types and error clusters have a fixed number of possible values. But strings and integers can have a huge number of possible values limited only by the size their data types. If a string or an integer is used as the input to a case structure, a default case must be specified. The default case will execute if a value is sent to the selector terminal that does not match one of the existing cases. Suppose an integer is wired to the selector terminal of a case structure and that cases are defined for the integers 0, 1, 2 and 3, where with the 0 case is designated as the default case. If a value of 4 is sent to the selector terminal, case 0 will execute, since case 4 is not defined and case 0 is the default. Boolean, enumerated type, and error-cluster inputs do not require the case structure to have a default state, but a default can be designated if desired.

Enumerated controls or constants are commonly used as inputs for case structure selector terminals. Each case of the case structure corresponds to a possible value of the enumerated constant. If an enumerated constant or control is wired to the selector terminal, the case selector menu at the top of the structure displays the enumerated items instead of the integer values. This makes the program more readable, as shown in Figure 1.20.

Error wires are commonly used as inputs to case structures. If an error wire is connected to the selector terminal, the case is configured to contain two cases: *Error* and *No Error*. The status Boolean of the error value is used to determine which case exectutes. If the status Boolean is True, the *Error* case executes If the status Boolean is False, the *No Error* case executes. When viewing the block diagram, the color of the border of the error-based case structure changes when the viewed case changes. The *No Error* case has a green border and the *Error* case has a red border, as shown in Figure 1.21.

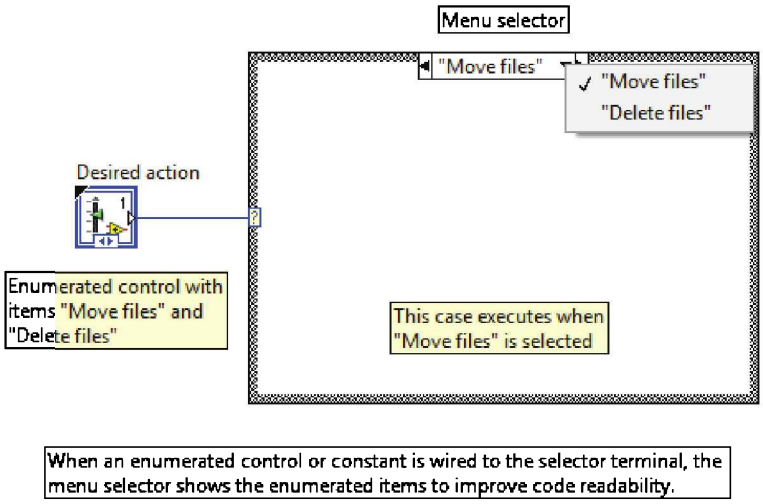


Figure 1.20: Case structures can be automatically configured to create cases matching the values of an enumerated constant or control

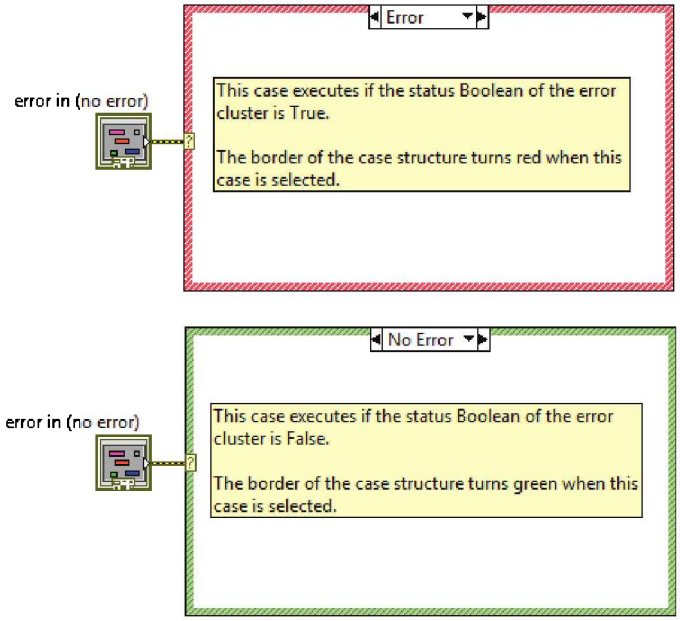


Figure 1.21: Case structures check the status Boolean if an error wire is connected to the selector terminal

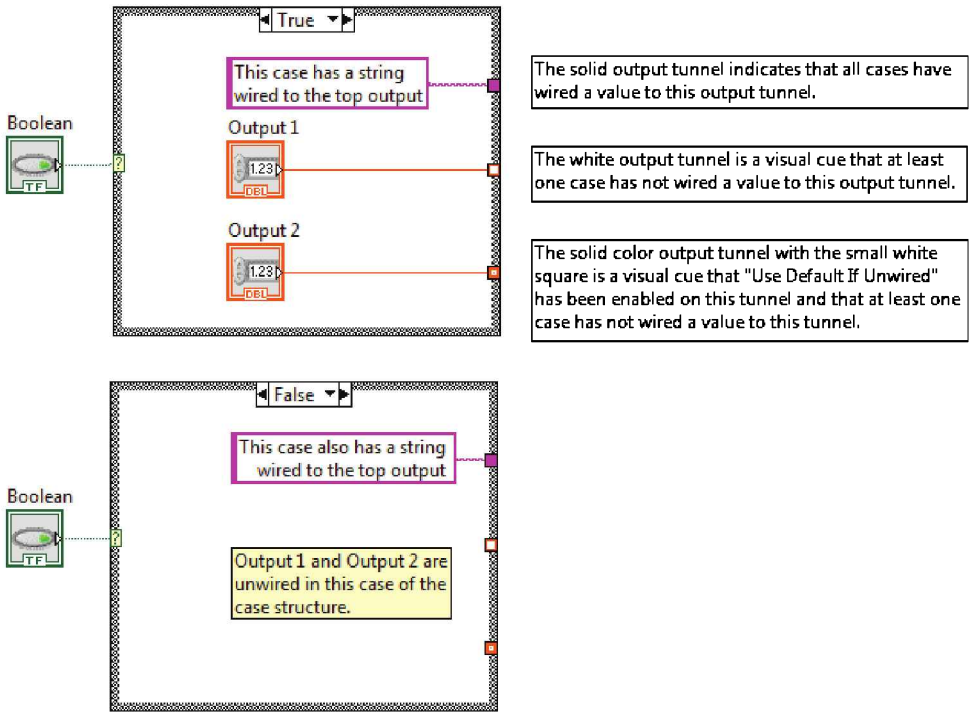


Figure 1.22: The appearance of an output tunnel signals whether or not it has been wired in every case of the case structure

Inputs to case structures (including the value wired to the selector terminal) are available to every case in the structure. The inputs can be used or ignored by individual cases without consequence. This is not true for outputs of a case structure. Every case in a case structure must output a value to every output tunnel of the case structure. If a case does not send a value to the output tunnel, the run arrow of the VI will be broken unless the *Use Default If Unwired* option has been selected for that tunnel. If this has been selected, the default value of the data type of the tunnel will be passed to the tunnel in any cases that leave the tunnel unwired.

The visual appearance of an output tunnel signals whether or not it has been wired in every case of the case structure. Figure 1.22 illustrates this by showing both the True and False cases of Boolean-input case structure. There are three outputs in the case structure. The top output is wired in both cases, so the output tunnel is a solid color. The middle output is wired in the True case, but, it is unwired in the False case. The output tunnel is white, indicating that at least one case has not wired a value to the tunnel. The bottom output tunnel is also wired in the True case and unwired in the False case. This output tunnel is solid with a small white square, indicating that the *Use Default If Unwired* option has been selected for this tunnel.

Style Guide: The *Use Default If Unwired* option should be used sparingly. It can degrade readability and could lead to subtle bugs in your VI.

1.9 The Standard Sub-VI Pattern with Error Handling

We now develop a template for a sub-VI with standard LabVIEW error handling. This sub-VI template will only execute if no error is present at its error input terminal. If an error is present, it will forego execution and pass the error to its error output terminal. This can be accomplished as shown in Figure 1.23. The error input terminal of the sub-VI is wired to the selector terminal of a case structure.

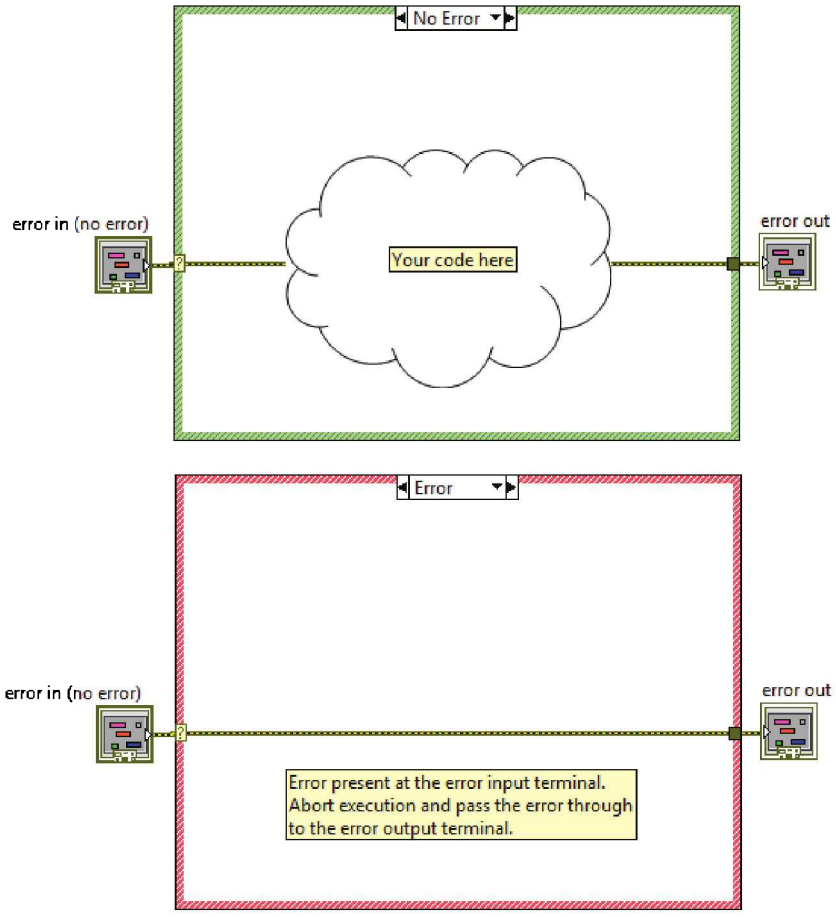


Figure 1.23: Template for a standard sub-VI with error handling

The working code of the sub-VI is placed in the *No Error* case of the case structure. If no error is present, the working code executes normally. But if an error is detected at the error input terminal, the *Error* case of the case structure will execute. The working code of the sub-VI is bypassed, and the error is passed directly to the error output terminal.

This pattern will be used throughout this book. Whenever a sub-VI built on this pattern is presented, only the *No Error* case will be shown. The *Error* case will always consist of a single wire connecting the error input and output terminals.

1.10 The `Move Or Delete` Sub-VI

We can now develop the sub-VIs needed for the `Clean Up Folder` VI. The icon for the `Move Or Delete` sub-VI is shown in Figure 1.24. This sub-VI accepts a target file path, a desired action enumerated control, and a destination path as inputs. If the enumerated control is set to the *Delete file* element, the targeted file is deleted. If the enumerated control is set to the *Move file* element, the file is moved to the folder specified by the destination path input.

The block diagram of the `Move Or Delete` sub-VI is shown in Figure 1.25. This sub-VI uses the standard error-handling template. If no error is present, the

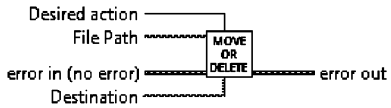


Figure 1.24: Icon of the `Move Or Delete` sub-VI

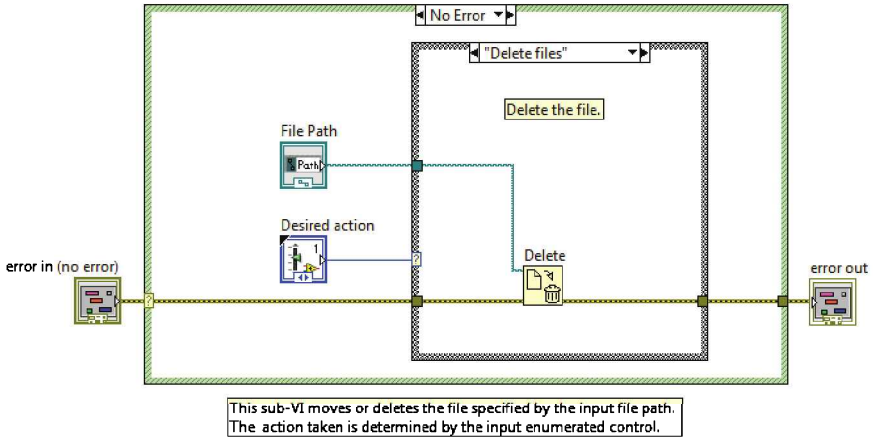


Figure 1.25: Block diagram of the `Move Or Delete` sub-VI, delete the file

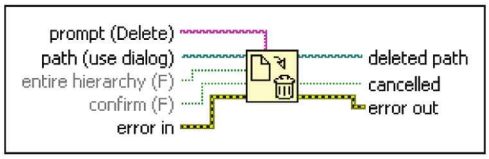
Delete Function

Owning Palette: Advanced File VIs and Functions

Requires: Base Package

Deletes the file or directory specified by **path**. This function does not work for files inside an LLB.

Details



Add to the block diagram Find on the palette

prompt is the message that appears above the list of files and directories, or folder, in the file dialog box.

path specifies the absolute path to the file or directory you want to delete. If **path** is empty (default), the function displays a dialog box from which you can select a file or directory. If you specify an empty or relative path, this function returns an error. **cancelled** is TRUE if you cancel the dialog box.

entire hierarchy determines whether this function deletes all contents in the folder. The default is FALSE. If **entire hierarchy** is TRUE, this function deletes all files and folders within the specified path.

If **confirm** is TRUE, the function displays a dialog box and asks you to confirm the delete operation. The default is FALSE.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

deleted path is the path of the deleted file or folder. **deleted path** returns `<Not A Path>` if an error occurred.

cancelled is TRUE if you cancel the file dialog box. Otherwise, **cancelled** is FALSE, even if this function returns an error.

error out contains error information. This output provides standard error out functionality.

Delete Details

If **path (use dialog)** specifies a directory that is not empty or if you do not have write permission for the file or directory specified by **path (use dialog)** and its parent directory, this function does not remove the directory and returns an error. Set **entire hierarchy** to TRUE to set the function to delete non-empty directories. Use the Set Permissions function to set the permissions for a file or directory.

Figure 1.26: Help file for the `Delete` function

sub-VI reads the value of the enumerated control containing the desired action. This value is wired to the selector terminal of a case structure. If the enumerated control has the value *Delete file*, the file is deleted using the native **`Delete`** function. The help file for this function is shown in Figure 1.26.

If the enumerated control has the value *Move file*, the file is moved as shown in Figure 1.27. The file is moved to the destination folder using the native **`Move`** function, whose help file is shown in Figure 1.28. The **`Move`** function is configured in this example to overwrite any previous versions of the file in the destination folder. The terminal for the destination path is placed in the *Move file* case of the case structure, since it is only needed by that case.

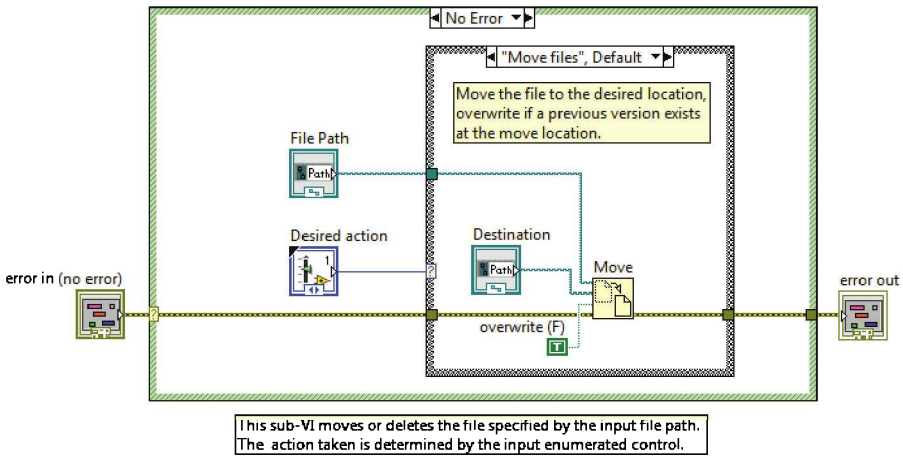


Figure 1.27: Block diagram of the `Move Or Delete` sub-VI, move the file

1.11 The `Right File Type` Sub-VI

The `Clean Up Folder` VI also needs a sub-VI to determine if a given file has the correct file extension. This task will be performed by the `Right File Type` sub-VI whose icon is shown in Figure 1.29. This sub-VI accepts a file path and an array of target extensions as inputs. The sub-VI outputs a CORRECT EXTENSION? Boolean indicator. If the extension of the file specified by the input path is found in the array of target extensions, the Boolean indicator output is True. Otherwise, the Boolean output is False. The input file path is also passed through the sub-VI as an output for wiring convenience in the `Clean Up Folder` VI.

The block diagram of the `Right File Type` sub-VI is shown in Figure 1.30. This sub-VI uses the native **`Get File Extension`** function to extract the file extension from the given file at the given path. The help file for this function is shown in Figure 1.31. The file extension is output from this function as a string. This string is then wired to the native **`Search 1D Array`** function, and this help file is shown in Figure 1.32. The array of target extensions is also wired to this function. The search function outputs the index of the matching element in the target extensions array. If the extension is not found in the array, the **`Search 1D Array`** returns a value of `-1`. The `Right File Type` sub-VI checks to see if this index is greater than or equal to zero. If it is, a match was found in the target array, and the output Boolean is set to True. If the index is negative, no match was found in the target array, and the output Boolean is set to False.

1.12 Loops

We are almost ready to begin developing the `Clean Up Folder` VI. Before we do, we need to discuss loops and error handling. LabVIEW provides two structures for

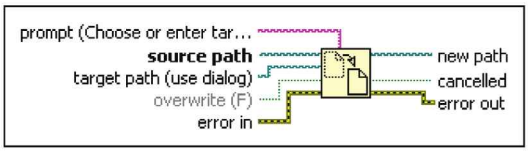
Move Function

Owning Palette: Advanced File VIs and Functions

Requires: Base Package

Moves the file or directory that you specify in **source path** to the location that you specify in **target path (use dialog)**. If you move a directory, this function moves all the contents of the directory recursively to the new location. You cannot use this function to move files into or out of an LLB.

Details



Add to the block diagram Find on the palette

prompt (Choose or enter target path for move) is the message that appears above the list of files and directories, or folder, in the file dialog box.

source path specifies an absolute path to the file or directory on which you want to operate. If you specify an empty or relative path, this function returns an error.

target path (use dialog) is the new absolute path for the file or folder on which you want to operate, including the new file or folder name. If **target path** is empty (default), the function displays a dialog box from which you can select a file or folder. If you specify an empty or relative path, this function returns an error. **cancelled** is TRUE if you cancel the dialog box.

overwrite determines whether the function replaces existing files or folders in the target path. If you overwrite a folder that contains unique files, LabVIEW moves those files to the new folder. **(Mac OS X and Linux)** LabVIEW does not overwrite read-only files, even if you set this parameter to TRUE.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

new path specifies the new location of the file or directory. If the operation is unsuccessful, this function sets **new path** to `<Not A Path>`.

cancelled is TRUE if you cancel the file dialog box. Otherwise, **cancelled** is FALSE, even if this function returns an error.

error out contains error information. This output provides standard error out functionality.

Move Details

If you do not have write permission for the file or directory that you specify in **target path (use dialog)**, this function does not move the file or directory and returns an error. Use the Set Permissions function to set the permissions for a file or directory.

Figure 1.28: Help file for the `Move` function

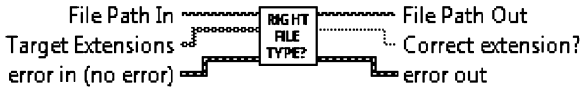


Figure 1.29: Icon of the `Right File Type` sub-VI

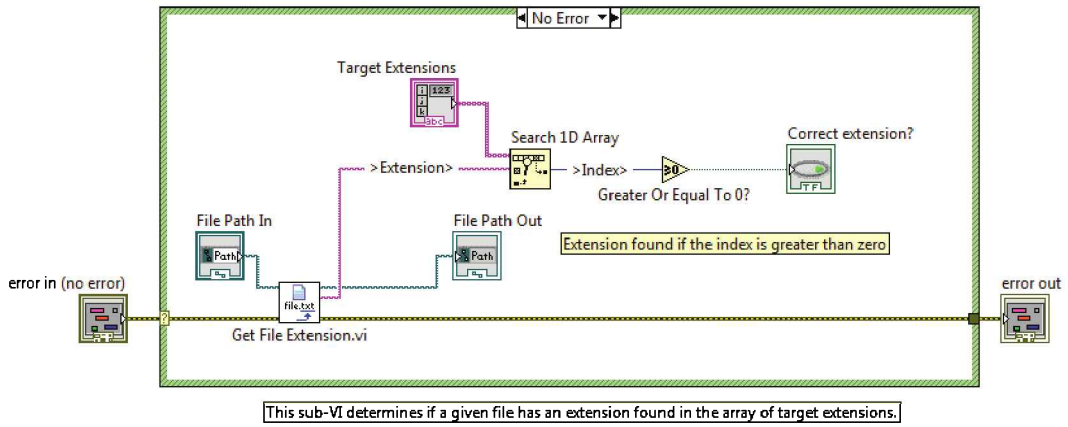


Figure 1.30: Block diagram of the `Right File Type` sub-VI

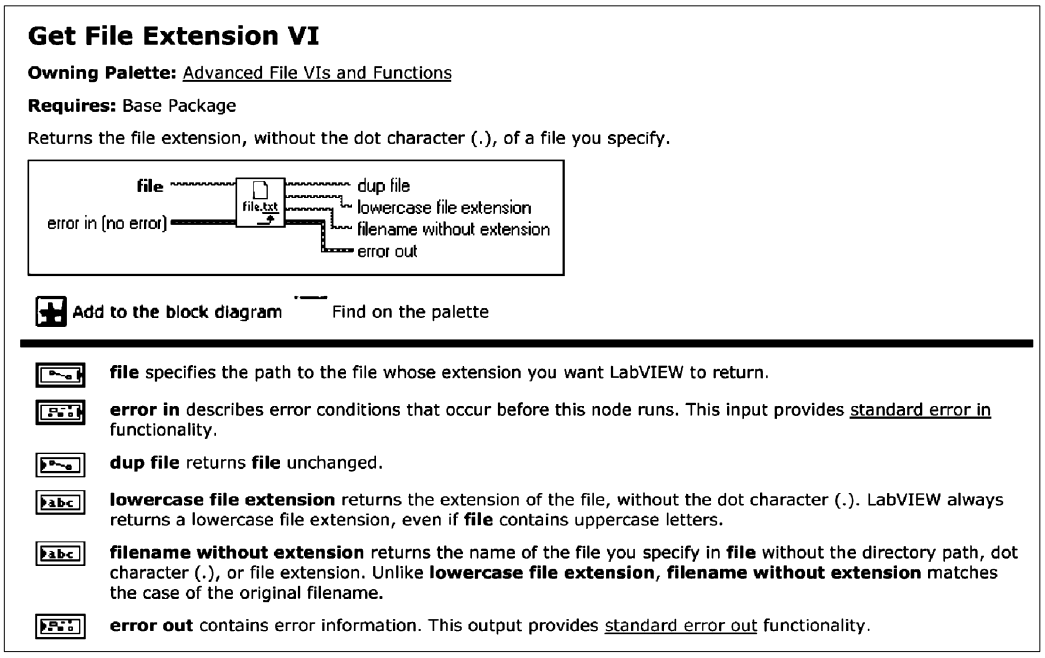


Figure 1.31: Help file for the `Get File Extension` function

repeatedly executing portions of code: **`While`** loops and **`For`** loops. These two loop structures are similar—the main difference between the two is in how the loops are terminated. **`While`** loops are typically used when the developer doesn't know ahead of time how many times the loop will be executed. For instance, a loop may execute until the user decides to terminate it or if an error occurs. **`For`** loops are typically used when the number of loop iterations is known before the loop executes.

Search 1D Array Function

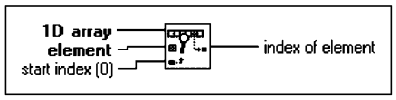
Owning Palette: Array Functions

Requires: Base Package

Searches for an **element** in a **1D array** starting at **start index**. Because the search is linear, you need not sort the array before calling this function. LabVIEW stops searching as soon as the element is found.

The connector pane displays the default data types for this polymorphic function.

Details



Add to the block diagram Find on the palette

1D array can be a 1D array of any type.

element is the value to search for in the input array. The representation of **element** should match the representation of **1D array**.

start index must be numeric. The default is 0.

index of element is the index where **element** is found. If the function does not find **element**, **index of element** is –1.

Search 1D Array Details

You cannot use this function to retrieve the index of a value that is not an element of the array. For example, if you have an array of two elements (0.0 and 1.0), this function does not find the index of the value 0.5, as that value is not an element of the array. Use the Threshold 1D Array to find a fractional index.

This function only finds a string if the **element** you specify matches an array element exactly. For example, if you have an array of two elements (`upper limit` and `lower limit`), this function does not find the index of the value `limit`, because `limit` does not match an array element exactly. To search a string for any occurrence of a regular expression, such as in the previous example, use the Match Regular Expression function.

Figure 1.32: Help file for the `Search 1D Array` function

`While` Loops

An example of a **`While`** loop is shown in Figure 1.33. The loop is designated by the gray arrow-like rectangular border. Code within the rectangle will be repeated when the loop iterates. Every **`While`** loop is created with two terminals: an iteration counter and a conditional terminal. The iteration counter acts like a control and outputs the current loop iteration as a signed 32-bit integer. LabVIEW starts all loop counts at zero, so the value of the iteration counter is zero during the first loop iteration. The iteration counter appears in the lower-left corner of the loop when the loop is created, but it can be moved anywhere within the loop.

The conditional terminal is used to determine when the loop ends. This terminal appears in the lower-right corner of the loop when the loop is created, but it too can be moved anywhere within the loop. The default form of the conditional terminal is *Stop if True*, which looks like a small stop sign, as seen in the left loop of Figure 1.33. In this case, the terminal acts like an indicator and accepts a Boolean input. The value on the Boolean input wire is checked at the end of each loop iteration. If the value is True, the loop terminates. If the value is False, the loop continues. The terminal can be changed to *Continue if True* by clicking on the terminal or by selecting *Continue if True* from the

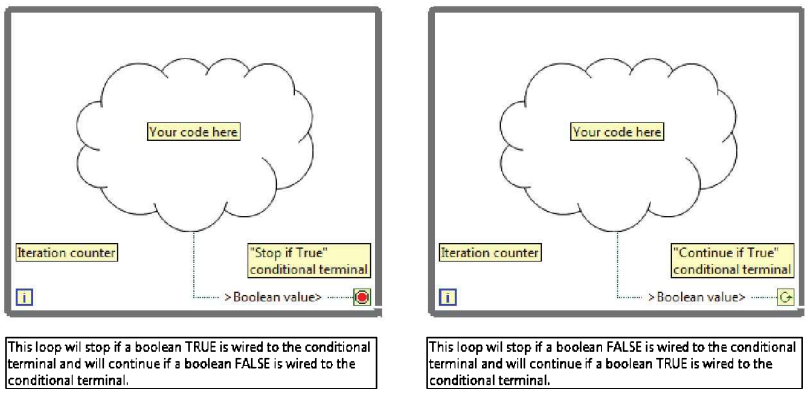


Figure 1.33: `While` loops in LabVIEW, where the code inside the loop repeats until the condition is satisfied

right-click menu of the terminal. In this form, the terminal looks like a green, circular arrow, as seen in the right loop of Figure 1.33. In this case, the loop will continue if the Boolean input is True and will terminate if the Boolean input is False.

`For` Loops

An example of a **`For`** loop is shown in Figure 1.34. The **`For`** loop box looks like a stack of paper with the top sheet dog-eared in the lower-right corner. Every **`For`** loop is created with two terminals: an iteration counter and a count terminal. The iteration counter is identical to the one found in a **`While`** loop. The iteration counter is a control that outputs the current loop iteration number with the first iteration indexed at zero. It appears in the lower-left corner of the loop when the loop is created, but it can be moved anywhere within the loop.

The count terminal appears in the upper-left corner of the loop and is designated by the letter *N*. Unlike the iteration counter, it cannot be moved. The count terminal is an indicator that accepts a signed 32-bit integer. The number wired to the count terminal determines the number of times the loop will run. Keep in mind that the iteration counter is zero-indexed. In Example 1 of Figure 1.34, a value of 10 is wired to the count terminal. This means that the loop will run ten times, but the value of the iteration counter will run from 0 to 9.

Loops are often used to operate on the elements of an array. The first array element is used in the first loop iteration, the second array element is used in the second loop iteration, and so on. This scenario is easily handled in LabVIEW using **`For`** loops and auto-indexing. If an array is wired as an input to a **`For`** loop and no input is wired to the count terminal, the loop will run once for every element in the array. With each loop iteration, the corresponding array element is fed into the loop. This auto-indexing is shown in Example 2 of Figure 1.34. Notice that the wire from the array to the loop is thick, indicating that the value on the wire is a one-dimensional

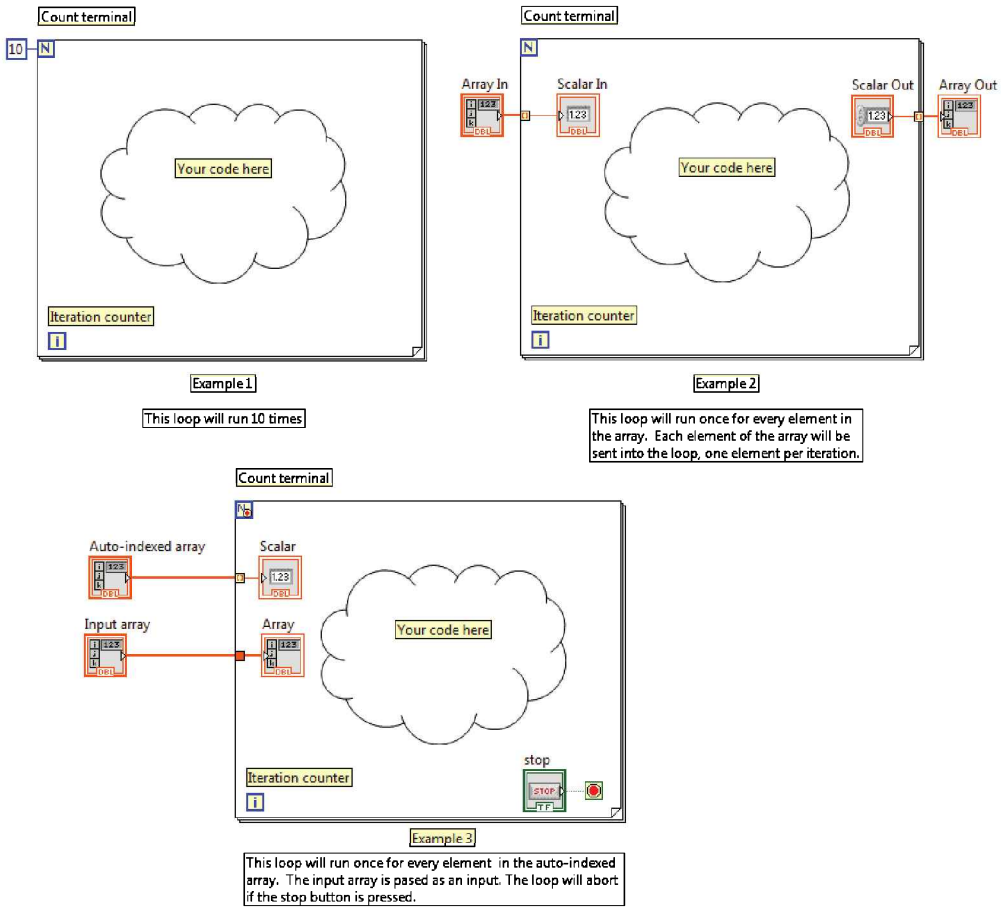


Figure 1.34: `For` loops in LabVIEW

array. As the wire crosses the loop boundary, it becomes thin, indicating that the value on the wire is a scalar. Auto-indexing also can be used on an output tunnel to build an array. Example 2 of Figure 1.34 also shows a scalar output wire crossing the right edge of the loop. The wire thickens as it exits the loop, indicating that an array of the output scalars will be passed on that wire after the loop ends.

In both of these cases, note the red square brackets "[]" in the input and output tunnels of the loop. The brackets indicate that auto-indexing is being used. Auto-indexing can be enabled or disabled by right-clicking on the desired tunnel and selecting the correct indexing mode. If auto-indexing is disabled, the tunnel behaves normally. For example, Example 3 of Figure 1.34 shows two array inputs to the loop. The auto-indexed array input tunnel has the bracket icon, indicating that this array determines the number of loop iterations. As the wire passes through the tunnel, it changes from an array to a scalar, indicating that each element of the array will be

passed into the loop one element per iteration. In contrast, the input array tunnel is solid. This indicates that auto-indexing is not enabled for this tunnel. The entire array is passed into the loop as an input. The wire stays thick as it passes through the tunnel, indicating that the entire array is carried into the loop. Since the value of the input does not change after the loop starts running, the array can be read by any loop iteration and remains unchanged throughout the execution of the loop.

A conditional terminal also can be added to a **For** loop. The loop will then execute for the prescribed number of iterations, just like a standard **For** loop, but the loop will be aborted if the conditional terminal is satisfied. The conditional terminal can be configured as *Stop if True* or *Continue if True*, just like a **While** loop. Conditional terminals are often added to **For** loops to abort the loop if an error is detected.

Style Guide: Even though it is possible to move the iteration counter and the conditional terminal in both **While** loops and **For** loops, don't do this unless it is necessary. Loops are easier to read when these terminals are in their default positions.

Loop Output Modes

Loop output tunnels can have one of four different operating modes: *Last Value*, *Indexing*, *Concatenating* and *Conditional*. **While** loop output tunnels by default have the *Last Value* tunnel mode. When the loop ends, the tunnel outputs the value wired to it during the last loop iteration. **For** loop output tunnels by default have the *Indexing* tunnel mode. This is the auto-indexing behavior described previously. An auto-indexed output tunnel collects each value wired to it and forms an output array of all the values. The output array is one dimension greater than the value wired to it inside the tunnel. If a scalar is wired to an indexed tunnel, the output is a one-dimensional array. If a one-dimensional array is wired to an indexed tunnel, then the output is a two-dimensional array.

Output tunnels also can have a *Concatenating* mode, where arrays are wired to concatenating tunnels. Wiring a scalar will result in an error. A concatenating tunnel collects the arrays wired to it each loop iteration and produces a concatenated output array. The output array has the same dimension as the array wired to the concatenated tunnel from within the loop. The difference between indexed and concatenated tunnels is illustrated in Figure 1.35. A two-dimensional array is passed to a **For** loop using an auto-indexed input tunnel. Each row of the array is passed into the loop—one row per loop iteration. Each row is then passed to an indexed output tunnel. The output of this tunnel is a two-dimensional array the same size as the input array. Each row is also passed to a concatenated tunnel. This tunnel concatenates each row and produces a one-dimensional array with the same number of elements as the input array. For example, if the input array had three rows and two columns, the output of the concatenated array would be a one-dimensional array with six elements.

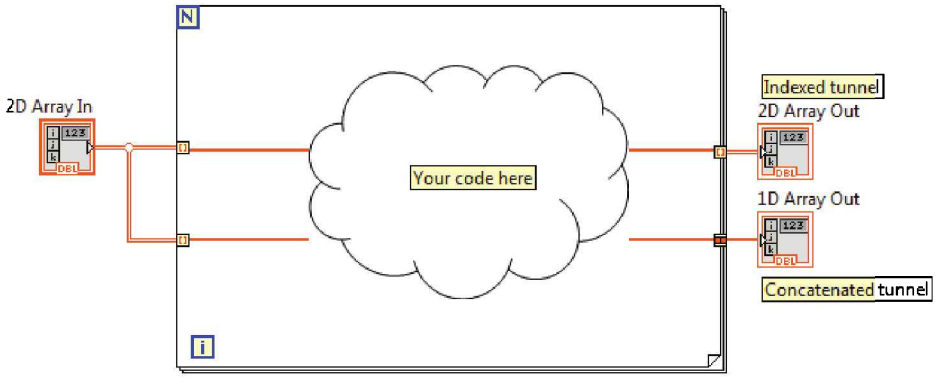


Figure 1.35: Comparing the *Indexed* and *Concatenated* tunnel modes

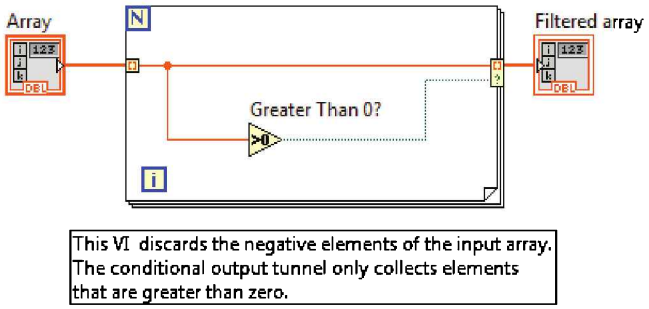


Figure 1.36: Example of a conditional loop output tunnel

Loop output tunnels can also have a *Conditional* operating mode. A conditional tunnel has two inputs: indexing and Boolean selection. A conditional tunnel acts like an indexing tunnel, but it only collects elements when the selection input is True. The output array will have the same dimension as the input array, but the number of elements may be smaller than that of the input array. This is demonstrated in Figure 1.36. Each element of the input array is passed into a **`For`** loop using an auto-indexed tunnel. Each element is then passed to a conditional output tunnel. If the element is greater than zero, a True value is wired to the conditional tunnel selector terminal, otherwise the value is False. The conditional output tunnel only collects the positive elements of the input array. Elements less than or equal to zero are discarded.

1.13 Shift Registers and Feedback Nodes

It is often necessary to store information generated by one loop iteration for use in future loop iterations. This is straightforward in most text-based programming

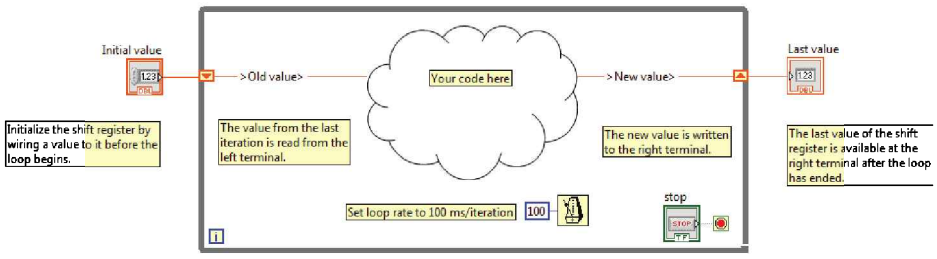


Figure 1.37: An example of how to use a shift register, where the old value is read from the terminal on the left and the new value is written to the terminal on the right

languages, since data can be assigned to named variables and recalled at any time. But this is not as easy in a dataflow language where the data is not named. Data passes along wires from one function to another anonymously and is not stored by the wires themselves. If data needs to be stored inside a loop, the storage mechanism needs to be added to the loop explicitly.

`While` loops and **`For`** loops use shift registers for data storage, as demonstrated in Figure 1.37. A shift register consists of a matching pair of terminals on the left and right edges of a loop. The terminal on the left edge of the loop has a downward pointing arrowhead and the terminal on the right edge of the loop has an upward pointing arrowhead. This visually suggests that the data "travels around the loop" from the right shift register to the left shift register. This is a good way of thinking about how a shift register operates. Data that is written to the right-hand terminal in one iteration of the loop "travels around the loop" and is available at the left-hand terminal during the next loop iteration. The shift register acts as a temporary storage device, making the data available to the next loop iteration.

The shift register can be initialized by wiring a value to it before the loop executes, as shown in Figure 1.37. The value wired to the left shift register from outside the loop will be the value of the shift register during the first loop iteration. If the shift register is not initialized, it will deliver the default value of the data type of the shift register during the first loop iteration. For example, if an uninitialized shift register has an integer data type, the value delivered during the first iteration will be 0, since 0 is the default value of the integer data type. The data type of the register is determined by the data type of the wire running to the right-hand terminal or by the data type used to initialize the shift register. Uninitialized shift registers also can be used to store information in sub-VIs from one call to the next. This will be covered in greater detail in Chapter 5.

1.14 Error Handling

Error handling is an important topic to consider in all VIs. In LabVIEW, when an error occurs, the error cluster is usually passed from one function to the next until the error

is finally passed to an error-handling routine and the program terminates. This is known as error trapping. Errors are trapped by maintaining a "tight" error chain in a VI. If an error chain is tight, any error generated in the VI will be captured and passed along to the error handling routine at the end of the VI as quickly as possible. A tight error chain has the following characteristics.

- It runs without breaks from the beginning of the VI to the end.
- Every LabVIEW function with error inputs and error outputs is wired into the chain, either along the main error wire or through parallel error wires that are then merged into the main error wire.
- Loops are constructed in such a way that warnings as well as errors are captured. It is also recommended that loops check for errors during each iteration and abort if an error is detected.
- Error wires are merged in such a way that the highest priority errors are passed to the main error wire.
- The error chain ends with an error-handling routine. All trapped errors and warnings are routed to this error-handling routine.

The codes in this book use a very simple error-handling strategy based on the LabVIEW **`General Error Handler`** function. This function will be placed at the end of the main error wire in each VI. If an error occurs, it will be passed on and captured by the **`General Error Handler`** function, and a dialog box describing the error will be displayed. The code will then shut down. The help file for the **`General Error Handler`** function is shown in Figure 1.38.

Error Handling in Loops

The main code of the `Clean Up Folder` VI will execute in a loop, and loops require special consideration for error handling. Figure 1.39 shows how to terminate a **`While`** loop if an error is detected. In this example, the `Error Prone` sub-VI is placed within the loop. The error wire from the `Error Prone` sub-VI and the value of the STOP button are used as inputs to the **`Or`** function—the result of which is wired to the conditional terminal of the loop. The loop will stop if the STOP button is pressed or if an error is generated.

When an error wire is used as an input to a logical function, the status Boolean of the error cluster is unbundled from the error wire and is used as an input to the logical function. This visual shorthand was also seen in Figure 1.21 when an error wire was used as an input for a case structure. If an error is detected in Figure 1.39, the status Boolean will be True, stopping the loop.

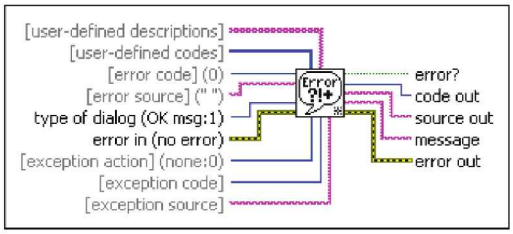
The error cluster is placed on a loop shift register, passing the cluster around the loop each iteration. The reason for doing this is to trap warnings as well as errors. Recall that a warning carries error code and source information while the status Boolean is False. A warning will not cause the loop to stop, but warnings should be trapped for later processing. If an error occurs later in the program, the warning will be overwritten

General Error Handler VI

Owning Palette: Dialog & User Interface VIs and Functions

Requires: Base Package

Indicates whether an error occurred. If an error occurred, this VI returns a description of the error and optionally displays a dialog box.



Add to the block diagram Find on the palette

[user-defined descriptions] is an array of descriptions of user-defined codes. If an incoming error matches one in **user-defined codes**, the VI returns the corresponding description from **user-defined descriptions** in **message**.

[user-defined codes] is an array of numeric error codes you can use to define error codes and messages for your own VIs. The VI searches this array after searching an internal database of error codes. Error codes -8999 through -8000, 5000 through 9999, and 500,000 through 599,999 are reserved for you to define your own error messages.

[error code] is a numeric error code. If **error in** indicates an error, the VI ignores **error code**. If not, the VI tests it. A nonzero value signifies an error.

[error source] is an optional string you can use to describe the source of **error code**.

type of dialog determines what type of dialog box to display, if any. Regardless of its value, the VI outputs the error information and **message** describing the error.

error in describes error conditions that occur before this node runs.

This input contains **status**, **code**, and **source**, which provide standard error in cluster element functionality.

[exception action] is a way for you to create exceptions to error handling. The VI performs the **exception action** if the **error code** and **error source** match the **exception code** and **exception source**. If you use the default value for an **exception source**, only the **exception code** must match for the VI to perform the **exception action**.

[exception code] is the error code that you want to treat as an exception. The default is 0.

[exception source] is the error message that you want to use to test for an exception. The default is an empty string.

error? indicates whether an error occurred. If this VI finds an error, it sets the parameters in the error cluster.

code out is the error code indicated by **error in** or **error code**.

source out indicates the source of the error. The **source out** string is a more descriptive string than the **source** string in the **error in** input.

message describes the error code that occurred, the source of the error, and a description of the error. If the VI does not return a description of the error, you can take several actions to find the error code description. If more than one description exists for the same error code, the VI displays all the descriptions, separated by `or`.

error out contains error information. This output provides standard error out functionality.

Figure 1.38: Help file for the `General Error Handler` function

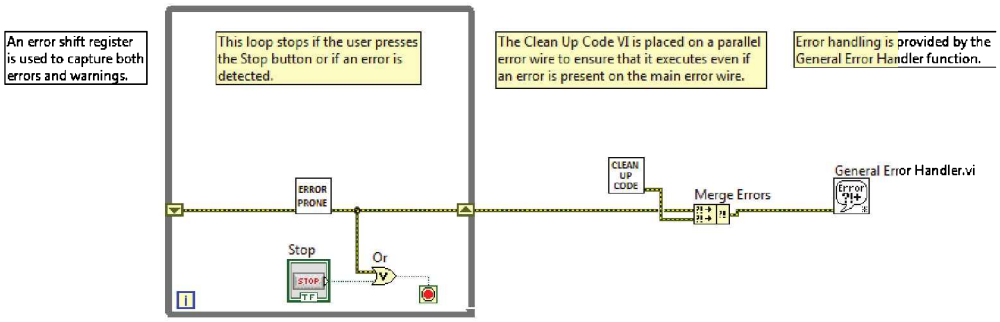


Figure 1.39: An example of how to stop a loop on detection of an error, where the error cluster is placed on a shift register to trap warnings as well as errors

by the error, but if no errors occur, the warning eventually will be carried to the error-handling code at the end of the error chain.

Merging Errors

If functions execute in series, it is typical to wire their error terminals together. This establishes a dataflow relationship between the functions and is a visually clean way of laying out the code. But many native LabVIEW functions are written in such a way that they do not execute if an error is present at their error input terminals. Instead, they pass the error on to their error output terminals. But what if you want a sub-VI to execute even when there is an error present? In Figure 1.39, the `Shut Down Code` sub-VI contains code necessary to shut down the system safely. If the `Shut Down Code` sub-VI has the standard LabVIEW error-handling behavior, it will not execute if an error is present at its error input terminal. Wiring the error output of the loop shift register directly to the error input of the `Shut Down Code` sub-VI is not a good idea in this case. If the `Error Prone` sub-VI generates an error, the `Shut Down Code` sub-VI will not run. It is better to place the `Shut Down Code` sub-VI on a parallel error wire and merge the two error wires using the **`Merge Errors`** function whose help file is shown in Figure 1.40.

In LabVIEW, wires carry only one instance of a data type at a time. Since errors are propagated as clusters along wires, this means that only one error cluster can travel along a wire at a time. When the **`Merge Errors`** function is placed on the block diagram, it only has two input nodes, but the function can be resized with the cursor to create additional terminals. The **`Merge Errors`** function searches the nodes in order from top to bottom looking for an error. It stops at the first error found, and passes it on to the *error out* terminal. If no errors are detected, it passes on the first warning found, passing a clear error cluster if no errors or warnings are found.

The fact that the **`Merge Error`** function can only pass one error creates an interesting situation if multiple errors are present. When a developer merges error

Merge Errors Function

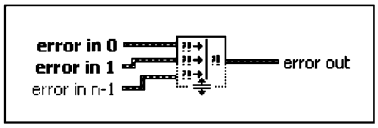
Owning Palette: Dialog & User Interface VIs and Functions

Requires: Base Package

Merges error I/O clusters from different functions.

This function looks for errors beginning with the **error in 0** parameter and reports the first error found. If the function finds no errors, it looks for warnings and returns the first warning found. If the function finds no warnings, it returns no error. Use exception control to treat what is normally an error as no error or to treat a warning as an error.

Details



Add to the block diagram Find on the palette

error in 0 describes the first error condition to check for errors.

error in 1 describes the second error condition to check for errors.

error out contains the first error found in **error in 0**, **error in 1**, and continuing in descending order for each input that is wired. If the VI finds no errors, **error out** contains the first warning found.

Merge Errors Details

When you place this function on the block diagram, it has two inputs available. You can add additional inputs to the node by right-clicking an input and selecting **Add Input** from the shortcut menu or by resizing the node. To remove inputs, right-click an input and select **Remove Input** from the shortcut menu or use resizing handles to resize the node.

Figure 1.40: Help file for the **`Merge Errors`** function

wires on a block diagram it is usually clear which error wire has a higher priority. In Figure 1.39, an error generated by the `Error Prone` VI is likely more interesting than an error generated by the `Shut Down Code` VI. Assume that the `Error Prone` VI deals with hardware, so an `Error Prone` error is likely a hardware error. The `Shut Down Code` VI is more likely a software-only routine, so its errors are likely to be more rare. Given the choice, we would rather pass on a likely hardware error than an unlikely software error. The main error wire is connected to the top terminal of the **`Merge Errors`** function to ensure that the `Error Prone` error will be passed on if multiple errors are detected.

1.15 Developing the `Clean Up Folder` VI

We can now develop the block diagram of the `Clean Up Folder` VI. This VI will first create an array of paths to the files contained in the target folder and all of its subfolders. It will then iterate through this array and pass each file to the `Right File Type` sub-VI to determine whether or not the file has a targeted extension. If it does

have a targeted extension, the file will be deleted or moved according to the value of the enumerated control set by the user.

The block diagram of the `Clean Up Folder` VI is shown in Figures 1.41 and 1.42. The VI uses the native **`Recursive File List`** function to recursively search through the target folder and its subfolders, see the help file in Figure 1.43. The output of this function is an array of file paths found in these folders. The array of file paths is then wired to an auto-indexing input tunnel of a **`For`** loop. The loop iterates once per array element, passing each file path individually to the interior of the loop. The individual file paths are then passed to the `Right File Type` sub-VI developed in Figure 1.30. This sub-VI outputs a Boolean signifying if the file has an

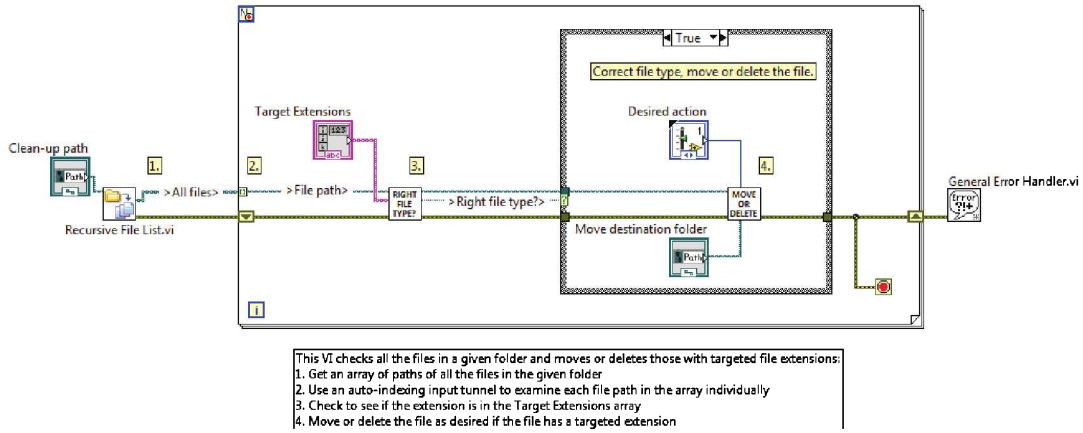


Figure 1.41: The block diagram of the `Clean Up Folder` VI file with a target extension

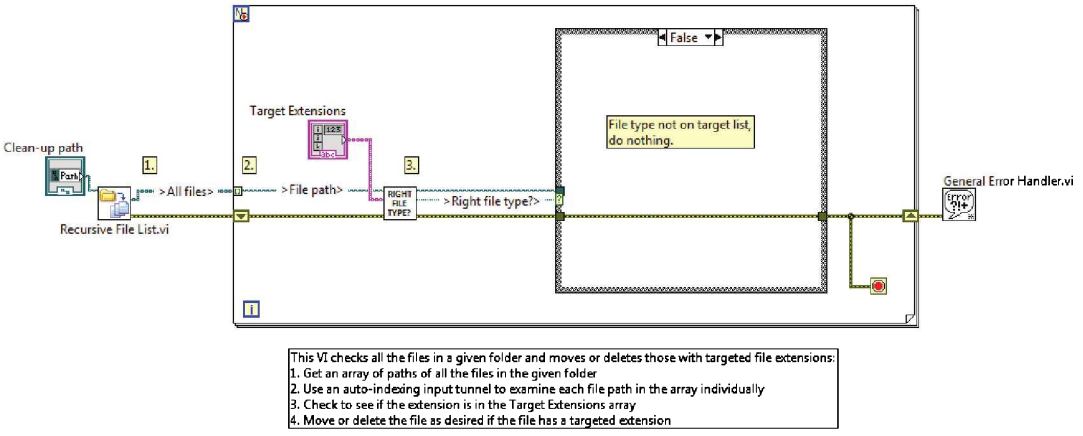


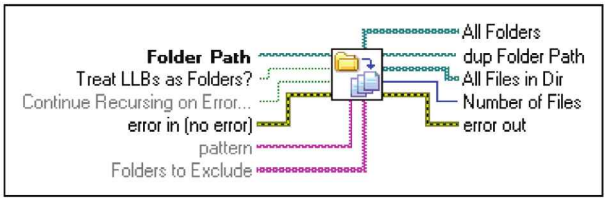
Figure 1.42: The block diagram of the `Clean Up Folder` VI file without a target extension

Recursive File List VI

Owning Palette: Advanced File VIs and Functions

Requires: Base Package

Lists the contents of a folder or LLB. In order for this VI to work correctly, the LLB must have a `.llb` file extension.



Add to the block diagram Find on the palette

Folder Path specifies the path to the folder or LLB whose contents you want LabVIEW to return.

Treat LLBs as Folders? specifies whether to list LLBs as files or folders. If **Treat LLBs as Folders?** is TRUE, this VI includes LLBs in the list of folders. If **Treat LLBs as Folders?** is FALSE, this VI includes LLBs in the list of files. The default is FALSE. If **Folder Path** specifies the path to an LLB, this VI ignores this input and returns all files, including LLBs, in **All Files in Dir**.

Continue Recursing on Error? specifies whether to continue recursing directories if an error occurs. If **Continue Recursing on Error?** is TRUE, the VI continues recursing directory contents after the error occurs. If **Continue Recursing on Error?** is FALSE, the VI stops recursing directory content when the error occurs.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

pattern specifies the pattern for files for which you want to search in **Folder Path**. You can use the question mark character (?) to match any single character. You can use the asterisk character (*) to match any sequence of one or more characters. For example, if you want this VI to find a list of the VIs in a folder, specify `*.vi` for **pattern**.

Folders to Exclude specifies folder names you want to exclude during the recursion. Any folder you exclude does not appear in the **All Folders** array. This input is case insensitive.

All Folders returns a list of all subfolders, and shortcuts to folders in **Folder Path**.

dup Folder Path returns **Folder Path** unchanged.

All Files in Dir returns a list of all files in **Folder Path**. If **Treat LLBs as Folders?** is set to TRUE, the list of files includes LLBs. The list also includes shortcuts to files.

Number of Files returns the number of files in the **All Files in Dir** array.

error out contains error information. This output provides standard error out functionality.

Figure 1.43: Help file for the `Recursive File List` function

extension found in the target list of extensions. The output Boolean is wired to the selector of a case structure. If the file does have a targeted extension, the file path is passed to the `Move or Delete` sub-VI shown in Figures 1.25 and 1.27. The file is moved or deleted according to the value of the DESIRED ACTION enumerated control. If the file does not have a targeted extension, no action is taken, as shown in Figure 1.42.

Let's examine the error handling in the `Clean Up Folder` VI. The **`For`** loop has a conditional terminal. The main error wire is connected to the conditional terminal, so the loop will stop if an error is detected. The error wire is also connected to a shift register on the loop, so any warnings generated in the loop will be captured and passed on to the next loop iteration. The error wire terminates at the **`General Error Handler`** function, which displays any errors to the user when the VI shuts down.

1.16 Passing Data by Reference

References present an interesting twist on LabVIEW's dataflow philosophy. LabVIEW typically operates on a by-value basis—not a by-reference basis. This means that data values travel on wires—not references to data values. Suppose that the number 3 was carried on a wire that splits into two wires. Each wire would then carry the value of the number 3. Each wire is independent of the other, because each carries a copy of the number 3. If the value on one of the two wires was incremented to 4, it would not affect the value of 3 on the other wire. Each wire carries an independent copy of the data carried on the original wire.

References follow the strict rules of dataflow, but they achieve a different result. A reference can be thought of as a pointer to a location in memory that stores a piece of data. Suppose a wire carries a reference to a memory location that contains the value 3. If this reference wire splits, both wires will carry copies of the reference to the memory location that contains the number 3. The numerical values of the references on the two wires are still independent. If the reference value on one wire is incremented to point to a different memory location, this will not affect the reference value on the other wire. In this sense, the rules of dataflow are strictly followed.

However, if one of the two reference wires is used to write the value 4 to the referenced memory location, the reference on the other wire will now point to a memory location that contains the value 4. In a way, the split reference wires are not independent of each other. Changes made to the referenced data by one wire are "seen" by the other wire.

References are commonly used by native LabVIEW functions and will be used throughout this book.

- File input and output functions, introduced in this chapter
- Queues, introduced in Chapter 13
- Data acquisition tasks, introduced in Chapter 15
- User events, introduced in Chapter 27
- Data value references, introduced in Chapter 29

Sub-VIs that accept references as inputs often pass these references through as outputs for block-diagram wiring convenience. The upper-left and -right corner

terminals of the sub-VI connector pane are typically used for the reference input and output, as shown in Figure 1.18.

1.17 File I/O

In the `Clean Up Folder` VI and its sub-VIs, file paths were used to manipulate files without opening them. The files were treated as black boxes, and their contents were never read or modified. LabVIEW functions for file input and output, or "file I/O" as it is commonly called, typically use file references rather than file paths to read or modify the contents of files.

Text files are commonly used in the examples and problem sets in this book. Text files are human-readable files primarily made up of the characters produced on a standard keyboard known as ASCII characters. The text-file format is very universal. Most major software packages that deal with words and numbers can read and write text files.

There are three basic operations involved in file I/O: opening the file, reading data from or writing data to the file, and closing the file. These steps are illustrated in the `Append To Text File` sub-VI icon shown in Figure 1.44 and block diagram shown in Figure 1.45. This example shows a LabVIEW sub-VI that opens or creates a

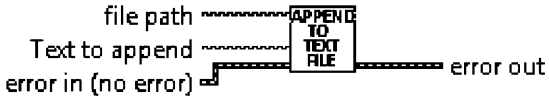


Figure 1.44: Icon of the `Append To Text File` sub-VI

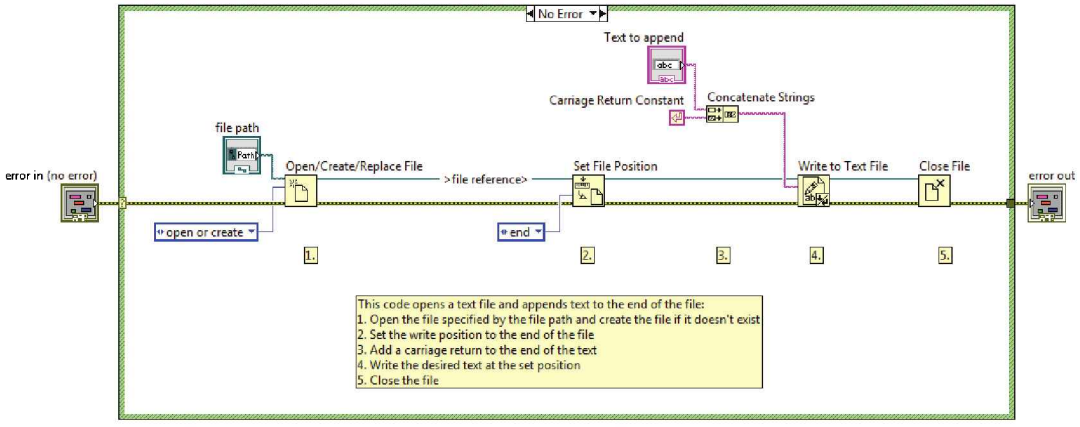


Figure 1.45: Block diagram of the `Append To Text File` sub-VI

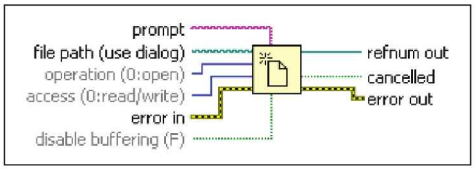
Open/Create/Replace File Function

Owning Palette: File I/O VIs and Functions

Requires: Base Package

Opens an existing file, creates a new file, or replaces an existing file, programmatically or interactively using a file dialog box. This function does not work for files inside an LLB.

Details Example



Add to the block diagram Find on the palette

prompt is the message that appears above the list of files and directories or folder in the file dialog box.

file path (use dialog) is the absolute path to the file. If you do not wire **file path (use dialog)**, the function displays a dialog box from which you can select a file. If you specify an empty or relative path, this function returns an error.

operation is the operation to perform. Error 43 occurs if you cancel the dialog box.

0	**open** (default)—Opens an existing file. Error 7 occurs if the file cannot be found.
1	**replace**—Replaces an existing file by opening the file and setting its end of file to 0.
2	**create**—Creates a new file. Error 10 occurs if the file already exists.
3	**open or create**—Opens an existing file or creates a new file if one does not exist.
4	**replace or create**—Creates a new file or replaces a file if it exists. This VI replaces a file by opening the file and setting its end of file to 0.
5	**replace or create with confirmation**—Creates a new file or replaces a file if it exists and you give permission. This VI replaces a file by opening the file and setting its end of file to 0.

access specifies how you plan to access the file. The default is read/write.

0	read/write
1	read-only
2	write-only

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

disable buffering specifies if the file opens without buffering. The default is FALSE. If you want to read or write a data file to a Redundant Array of Independent Disks (RAID), consider opening the file without buffering to speed up data transfers. To disable buffering, wire a TRUE value to the **disable buffering** input.

Note If you have a small amount of data to transfer, you might not notice a difference if you disable buffering.

(Mac OS X and Linux) LabVIEW ignores this input.

refnum out is the reference number of the open file. The value is `Not A Refnum` if the file cannot be opened.

cancelled is TRUE if you cancel the file dialog box or if you do not select the replacement in an advisory dialog box.

error out contains error information. This output provides standard error out functionality.

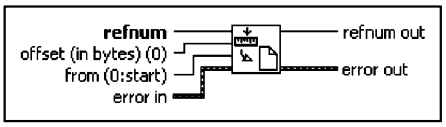
Figure 1.46: Help file for the `Open/Create/Replace File` function

Set File Position Function

Owning Palette: Advanced File VIs and Functions

Requires: Base Package

Moves the position of the current file mark of the file identified by **refnum** to the position indicated by **offset (in bytes)** according to the mode in **from**.



Add to the block diagram Find on the palette

refnum is the file refnum associated with the file whose file mark you want to move.

offset (in bytes) specifies how far from the location specified by **from** to set the file mark.

from, together with **offset (in bytes)**, specifies where to set the file mark. If you wire **offset (in bytes)**, **from** defaults to 0, and the offset is relative to the beginning of the file. If you do not wire **offset (in bytes)**, **offset (in bytes)** defaults to 0, **from** defaults to 2, and the operation starts at the current file mark.

0	**start**—Sets the file mark **offset (in bytes)** bytes from the beginning of the file. If **from** is 0, **offset (in bytes)** should be positive or 0, which sets the file mark at the beginning of the file.
1	**end**—Sets the file mark **offset (in bytes)** bytes from the end of the file. If **from** is 1, **offset (in bytes)** should be negative or 0, which sets the file mark at the end of the file.
2	**current**—Sets the file mark **offset (in bytes)** bytes from the current file mark.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

refnum out returns **refnum**.

error out contains error information. This output provides standard error out functionality.

Figure 1.47: Help file for the `Set File Position` function

text file with a given path, appends text to the end of the file with an added carriage return, and then closes the file.

The **`Open/Create/Replace File`** function accepts a file path as an input. It also accepts an enumerated input that specifies whether the file should be opened, created if doesn't exist, replaced if it does exist, or some combination of these options. The output of this function is a reference to the opened file—not the file itself. The help file for this function is shown in Figure 1.46.

The reference generated by the **`Open/Create/Replace File`** function is then passed on to the **`Set File Position`** function where the help file is shown in Figure 1.47. This function sets the position of the file mark in the referenced file. The file mark determines the starting position of the next read or write operation on that file. In Figure 1.45, the file mark position is set to the end of the file.

The file reference is then passed to the **`Write to Text File`** function to write the input text to the referenced file at the marked position. The help file for this function is shown in Figure 1.48. Before the text is written to the file, a carriage return is added

Write to Text File Function

Owning Palette: File I/O VIs and Functions

Requires: Base Package

Writes a string or an array of strings as lines to a file. This function does not work for files inside an LLB.

Use the Set File Position function if you need to perform random access file reads or writes.

Details Example



Add to the block diagram Find on the palette

prompt is the message that appears above the list of files and directories or folder in the file dialog box.

file can be a refnum or absolute file path. If you wire a path to the **file** input, the function opens or creates the file before writing to it and replaces any previous file contents. If you wire a file refnum to the **file** input, writing begins at the current file position. To append to an existing file, set the file position to the end of the file by using the Set File Position function. The default is to display a file dialog box and prompt you to select a file.

If you specify an empty or relative path, this function returns an error.

text is the data the function writes to the file. **text** is a string or an array of strings.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

refnum out is the refnum of the file that the function read. You can wire this output to another file function, depending on what you want to do with the file. The default is to close the file if it is referenced by a file path or selected from the file dialog box. If **file** is a refnum or if you wire **refnum out** to another function, LabVIEW assumes that the file is still in use until you close it.

cancelled is TRUE if you cancel the file dialog box. Otherwise, **cancelled** is FALSE, even if this function returns an error.

error out contains error information. This output provides standard error out functionality.

Write to Text File Details

This function appends platform-dependent, end-of-line (EOL) characters to the elements of an array even if you right-click the function and remove the checkmark next to the **Convert EOL** shortcut menu item.

Figure 1.48: Help file for the `Write to Text File` function

with the **`Concatenate Strings`** function, and this help file is shown in Figure 1.49. This ensures that the next text added to the file begins on a new line.

The file is closed with the **`Close File`** function. The **`Close File`** function has an interesting property. Most LabVIEW functions with error terminals are designed to not execute if an error is present at the error input terminal. Instead of executing their normal functions, they simply pass the error to their error output terminals. The **`Close File`** function is an exception to this rule. This function will close the referenced file even if an error is present at the error input terminal. The help file for this function can be found in Figure 1.50.

Concatenate Strings Function

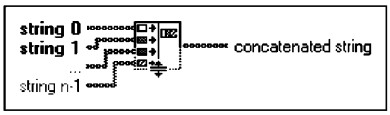
Owning Palette: String Functions

Requires: Base Package

Concatenates input strings and 1D arrays of strings into a single output string. For array inputs, this function concatenates each element of the array.

Add inputs to the function by right-clicking an input and selecting **Add Input** from the shortcut menu or by resizing the function.

Details Example



Add to the block diagram Find on the palette

string 0..n-1 are the strings you want to concatenate.

concatenated string contains the concatenated input strings in the order you wire them to the node from top to bottom.

Concatenate Strings Details

You can use this function to concatenate the output from Picture Functions VIs so that they draw on a single picture control. The pictures are drawn in order from top to bottom.

Example

Refer to the Concatenate Strings VI in the `labview\examples\general\functions\String` directory for an example of using the Concatenate Strings function.

Figure 1.49: Help file for the `Concatenate Strings` function

Close File Function

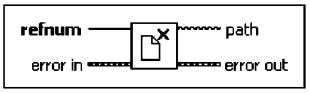
Owning Palette: File I/O VIs and Functions

Requires: Base Package

Closes an open file specified by **refnum** and returns the path to the file associated with the refnum.

Error I/O operates uniquely in this function, which closes the file regardless of whether an error occurred in a preceding operation. This ensures that files are closed correctly.

Examples



Add to the block diagram Find on the palette

refnum is the file refnum associated with the file you want to close.

error in describes error conditions that occur before this node runs. With the following exception, this input provides standard error in functionality.

This node runs normally *even if* an error occurred before this node runs.

path is the corresponding path to **refnum**.

error out contains error information. This output provides standard error out functionality.

Figure 1.50: Help file for the `Close File` function

1.18 The `Simulated Data Acquisition` VI

LabVIEW is often used for data acquisition. For our final example, we develop the `Simulated Data Acquisition` VI. The front panel is shown in Figure 1.51. This VI uses a random number generator to simulate single-point data acquisition at a rate of ten samples per second until the user presses the STOP button or an error occurs. The STOP button is reinitialized to False when the VI starts. The data is plotted in real time on a waveform chart, and the chart is automatically cleared of past data when the VI starts.

The VI checks the simulated data as it is collected to see if it exceeds a user-defined limit. The user is able to change the limit when the VI is running. If a simulated data point exceeds the limit, it is written to a comma-separated value (CSV) text file along with the date, time, and current value of the user-defined limit in the following order: date, time, limit value, data value. Individual records in the log file are separated by carriage returns. The log file is named `Log.txt`, and it is stored in the same directory as the `Simulated Data Acquisition` VI. The VI also keeps a running count of the number of data points that have exceeded the limit and displays the current value of the count in a numeric indicator on the front panel.

The data is generated by the `Get Simulated Data` sub-VI. This icon is shown in Figure 1.52 and block diagram is shown in Figure 1.53. This sub-VI uses the **`Random Number (0-1)`** function to generate a pseudorandom number between zero and one. The help file for this function is shown in Figure 1.54.

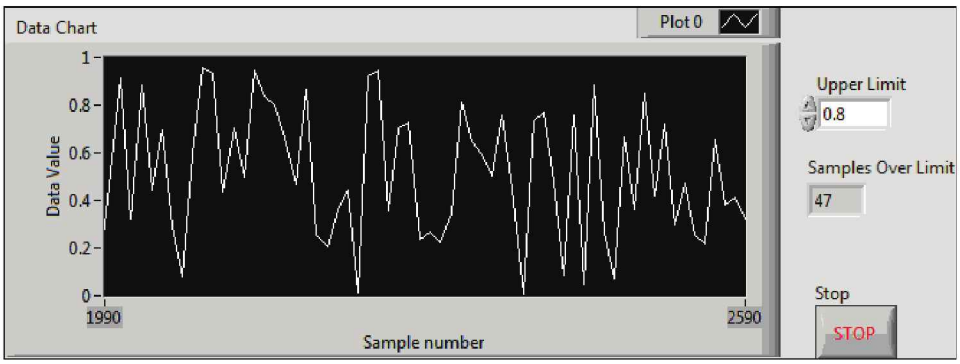


Figure 1.51: Front panel of the `Simulated Data Acquisition` sub-VI



Figure 1.52: Icon of the `Get Simulated Data` sub-VI

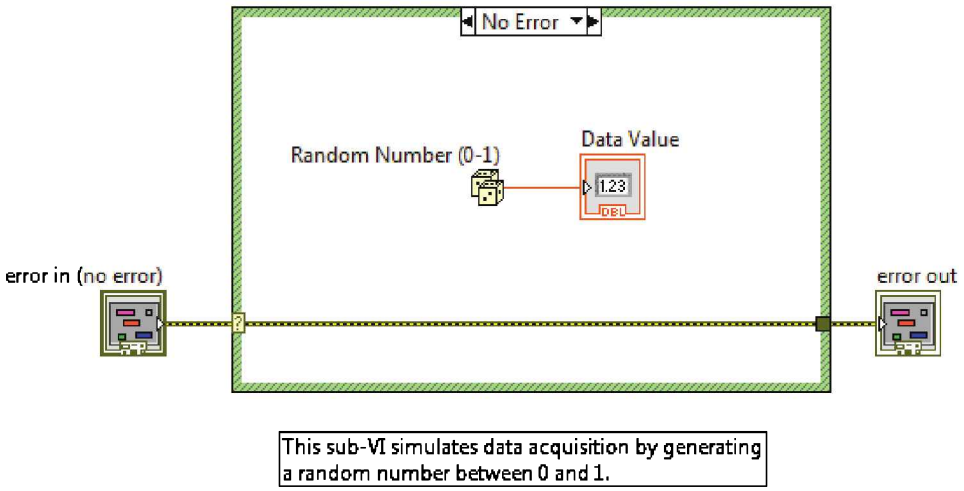


Figure 1.53: Block diagram of the `Get Simulated Data` sub-VI

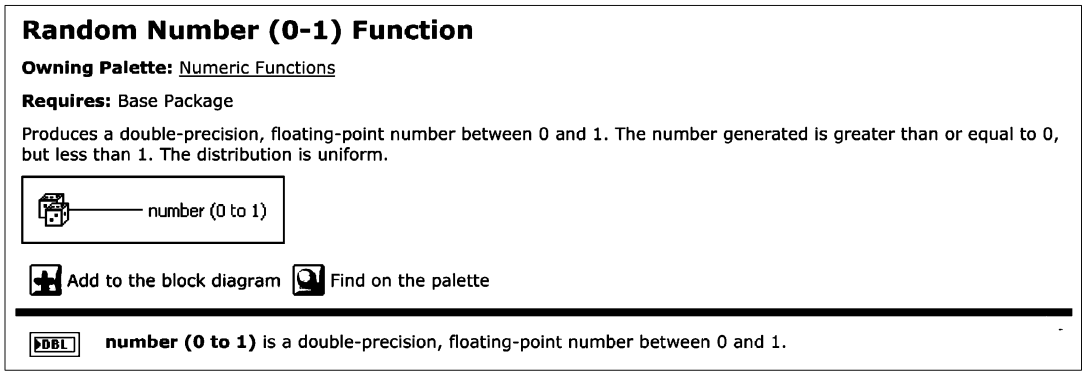


Figure 1.54: Help file for the `Random Number (0-1)` function

1.19 Loop Timing

The specification of the `Simulated Data Acquisition` VI states that the data will be taken ten times per second until the user presses the STOP button or an error is generated. This can be accomplished by placing the `Get Simulated Data` sub-VI in a **`While`** loop. The loop timing needs to be regulated to meet the requirement that the data is generated ten times per second or once every 100 milliseconds.

Loop timing can be performed using the **`Wait Until Next ms Multiple`** function. This help page is shown in Figure 1.55. This function accepts an integral

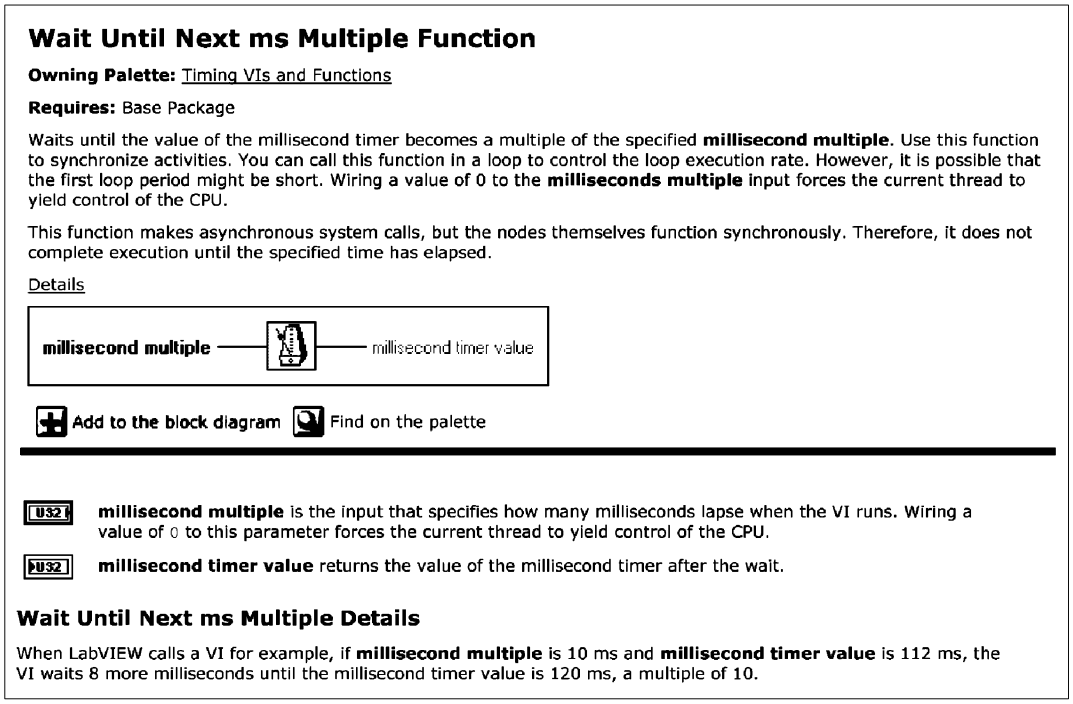


Figure 1.55: Help file for the `Wait Until Next ms Multiple` function

number of milliseconds as an input. When this function is placed within a loop, it provides a variable amount of delay to ensure that the period of the loop is equal to the specified number of milliseconds. An examples of how to use the **`Wait Until Next ms Multiple`** VI can be found in Figures 1.37 and 5.3.

The variability of the delay is a key feature of this function, since the execution time of the code within the loop also may be variable. For instance, suppose that you want the loop to run at 100 milliseconds per iteration. This would be achieved by placing the **`Wait Until Next ms Multiple`** VI inside the loop and wiring a value of 100 to the millisecond input. If the code within the loop took 50 milliseconds to execute during one iteration of the loop, the **`Wait Until Next ms Multiple`** function would add an extra 50 milliseconds of delay after the code executed to bring the total to 100 milliseconds. If the code took 75 milliseconds to execute during the next iteration, the **`Wait Until Next ms Multiple`** function would add 25 milliseconds of delay to again bring the total to 100 milliseconds. The **`Wait Until Next ms Multiple`** function will continue to regulate the loop to 100 milliseconds per iteration as long as the code inside the loop takes less than 100 milliseconds to execute. If the code within the loop exceeds 100 milliseconds, the **`Wait Until Next ms Multiple`** function will add a delay to bring the total iteration time up to the next multiple of 100 milliseconds. For example, if the code within

the loop took 110 milliseconds to execute, the **Wait Until Next ms Multiple** function would add 90 milliseconds of delay to bring the total to 200 milliseconds. If the code within the loop took 390 milliseconds to execute, the **Wait Until Next ms Multiple** function would add 10 milliseconds of delay to bring the total to 400 milliseconds.

The **Wait Until Next ms Multiple** function has another valuable feature. When this function adds its delay, it also releases the CPU to perform other tasks during the delay. It essentially puts LabVIEW to sleep during the delay, allowing the CPU to take care of other processes on the host computer.

1.20 The Data Over Limit sub-VI

The simulated data needs to check each loop iteration to see if it exceeds the user-defined limit. This will be accomplished with the Data Over Limit sub-VI with the icon shown in Figure 1.56. This sub-VI accepts the current data point and limit value as inputs, and it outputs a Boolean value that indicates whether the data exceeds the limit or not. This sub-VI passes the data value and limit through as outputs for wiring convenience. The block diagram of this sub-VI is shown in Figure 1.57.

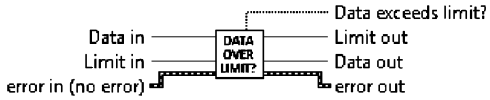


Figure 1.56: Icon of the Data Over Limit sub-VI

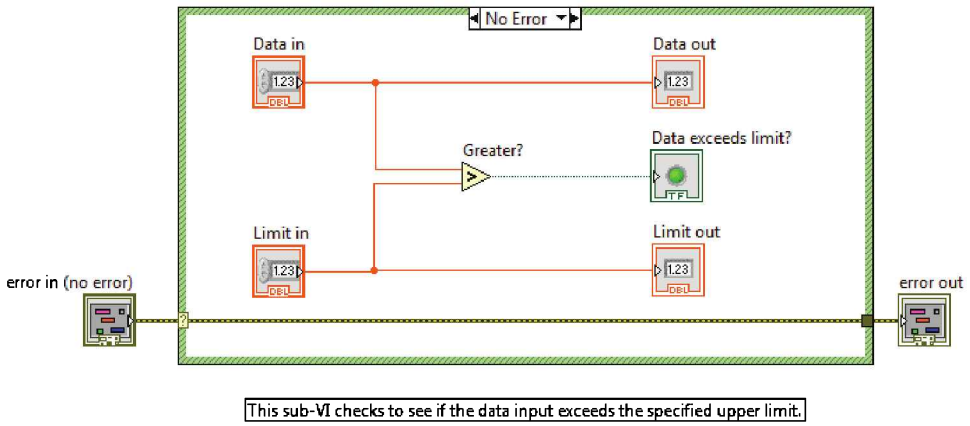


Figure 1.57: Block diagram of the Data Over Limit sub-VI

1.21 The `Create Log String` sub-VI

Whenever the simulated data exceeds the user-defined limit, it is written to a log file as a comma-separated string with the following format: date, time, current limit value, data value. The `Create Log String` sub-VI icon is shown in Figure 1.58, and it will create this string. This sub-VI accepts a time stamp, the limit value and the data value as inputs, and outputs the log string in the correct format.

The block diagram of the `Create Log String` sub-VI is shown in Figure 1.59. The log string is created using the **`Format Into String`** function. This help file is shown in Figure 1.60. The **`Format Into String`** uses a format string with tokens to create an output string. The inputs to the function are mapped to the tokens in the format string. The format statement used here is `%s, %s %f, %f`, meaning that the function expects two string inputs representing the date and the time followed by two floating-point numerical inputs representing the limit value and the data value. The time stamp input is a native LabVIEW data type converted to a date string and a time string using the **`Get Date/Time String`**. This help file is shown in Figure 1.61.

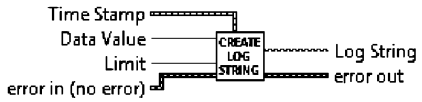


Figure 1.58: Icon of the `Create Log String` sub-VI

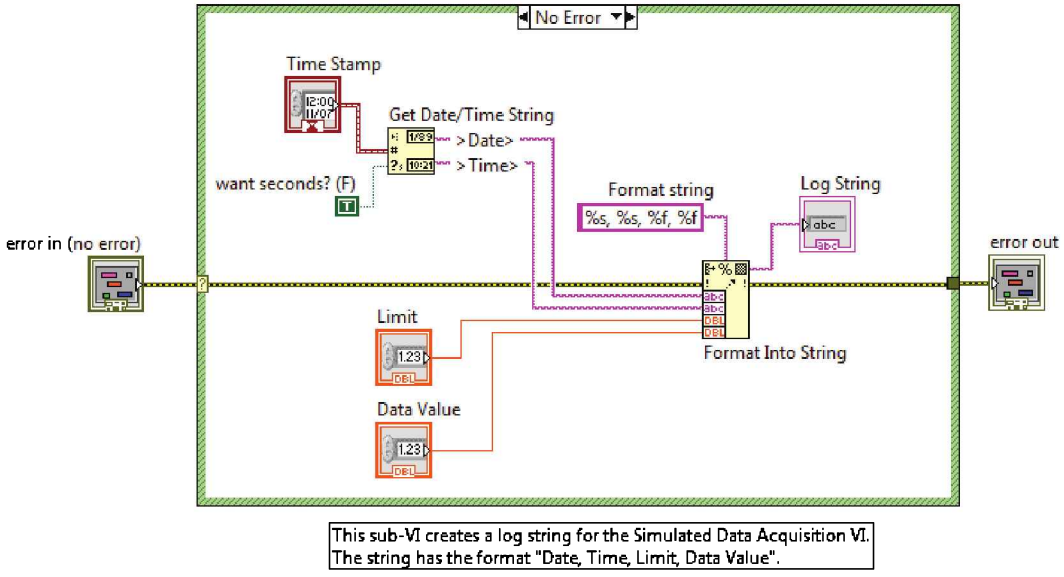


Figure 1.59: Block diagram of the `Create Log String` sub-VI

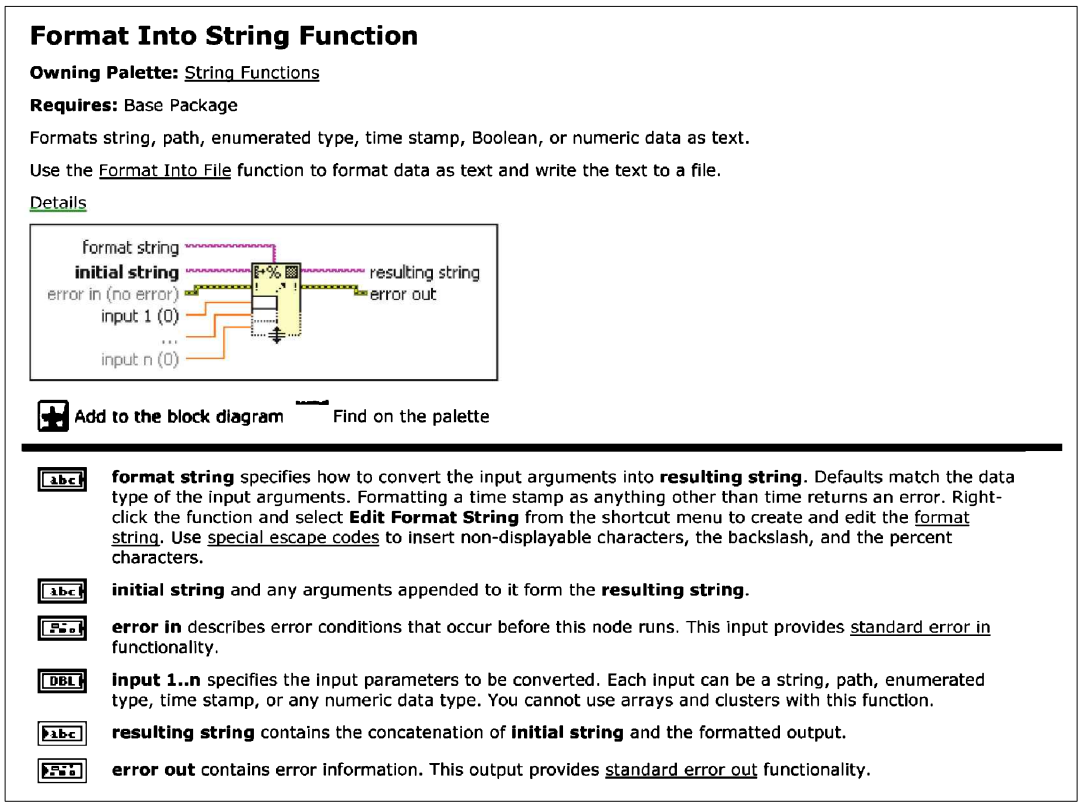


Figure 1.60: Help file for the `Format Into String` function

These strings are wired to the top two inputs of the **`Format Into String`** function. The upper limit and data values are wired to the bottom two inputs. The format string wired to the **`Get Date/Time String`** function separates the tokens with commas, so the output string will also separate the input values with commas.

1.22 Absolute and Relative File Paths

The path controls used in the `Clean Up Folder` VI shown in Figure 1.41 fully specify the target folder and the destination folder. Fully specified paths are known as absolute paths. An absolute path contains all of the information needed to find a particular folder or file. File paths can be relative rather than absolute. A relative file path is the path to a given file relative to some other file or directory. For example, the log file in the `Simulated Data Acquisition` VI is named `Log.txt`, and it resides in the same directory as the `Simulated Data Acquisition` VI. Even though the name of the log file is known, it cannot be located unless the directory of the main application is also known.

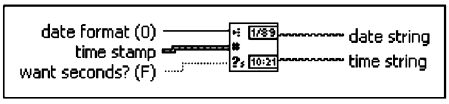
Get Date/Time String Function

Owning Palette: Timing VIs and Functions

Requires: Base Package

Converts a timestamp value or a numeric value to a date and time string in the time zone configured for the computer. The function interprets timestamp and numeric values as the time-zone-independent number of seconds that have elapsed since 12:00 a.m., Friday, January 1, 1904, Universal Time.

Example



Add to the block diagram Find on the palette

date format selects the appearance of the **date string**. Date formats vary with your system configuration. To use a different date format, use the Format Date/Time String function.

0	**Short**—1/21/94
1	**Long**—Friday, January 21, 1994
2	**Abbreviated**—Fri, Jan 21, 1994

time stamp can be a timestamp or a numeric. If numeric, this number is the time-zone-independent number of seconds that have elapsed since 12:00 a.m., Friday, January 1, 1904, Universal Time [01-01-1904 00:00:00]. The default is the current date and time. If **year** is before 1904, **time stamp** is negative.

want seconds? controls the display of seconds in the **time string**.

date string is the string returned by the function according to the specified **date format**.

time string returns the string formatted according to the configured time zone for your computer. **want seconds?** controls whether the string includes seconds.

Figure 1.61: Help file for the `Get Date/Time String` function

Data cannot be written to a file when you only know its relative path, the relative path must be converted to an absolute path to access the file. In the `Get Simulated Data` VI, the relative path can be converted to an absolute path using two LabVIEW functions: **`Application Directory`** and **`Build Path`**. The **`Application Directory`** function returns the absolute path to the directory of its calling VI: the help file is shown in Figure 1.62. The **`Build Path`** function accepts two inputs: an absolute base path and a second path relative to the base path. The output of the function is the absolute path to the location defined by the second path relative to the base path. The help file for this function is shown in Figure 1.63.

1.23 The `Append To Log File` Sub-VI

The **`Application Directory`** function and the **`Build Path`** function now can be used to develop the `Append To Log File` sub-VI with the icon shown in Figure 1.64. The block diagram is shown in Figure 1.65. The **`Application Directory`** function and the **`Build Path`** function are used to create an absolute path

Application Directory VI

Owning Palette: File Constants

Requires: Base Package

Returns the path to the directory containing the application.

If you call this VI from a stand-alone application, this VI returns the path to the folder containing the stand-alone application.

Note If you call the Application Directory VI from a shared library, the VI returns `<Not a Path>`.

If you call this VI from the development environment and the VI is loaded in a LabVIEW project file (`.lvproj`), this VI returns the path to the folder containing the project file. If the project is not saved, this VI returns `<Not a Path>`.

If the top-level VI is not loaded in a project, the VI returns the path to the directory containing the top level VI. If the VI is not saved to disk, this VI returns `<Not a Path>`.



Add to the block diagram Find on the palette

Application Directory is the path to the directory where this VI is running.

Figure 1.62: Help file for the `Application Directory` function

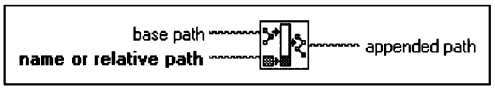
Build Path Function

Owning Palette: File I/O VIs and Functions

Requires: Base Package

Creates a new path by appending a name (or relative path) to an existing path.

Details



Add to the block diagram Find on the palette

base path specifies the path to which this function appends **name**. The default is an empty path. If **base path** is invalid, this function sets **appended path** to `<Not A Path>`.

name or relative path is the new path component appended to **base path**. If **name or relative path** is an empty string or an invalid path, this function sets **appended path** to `<Not A Path>`. If base path is an empty path then **name or relative path** must be an absolute path. This function then sets **appended path** to the absolute path in **name or relative path**.

appended path is the resulting path.

Build Path Details

Suppose the path to the VI containing this function is `c:\dir1\first.vi`. To use this function to build a path to another VI such as `c:\dir1\second.vi`, first wire the output of a Current VI's Path function to a Strip Path function. The Strip Path function returns a path of `c:\dir1`. You can wire this path to the Build Path function along with the name of the second VI.

Figure 1.63: Help file for the `Build Path` function

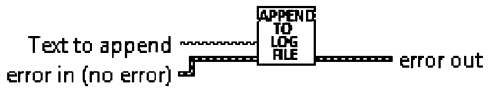


Figure 1.64: Icon of the `Append To Log File` sub-VI

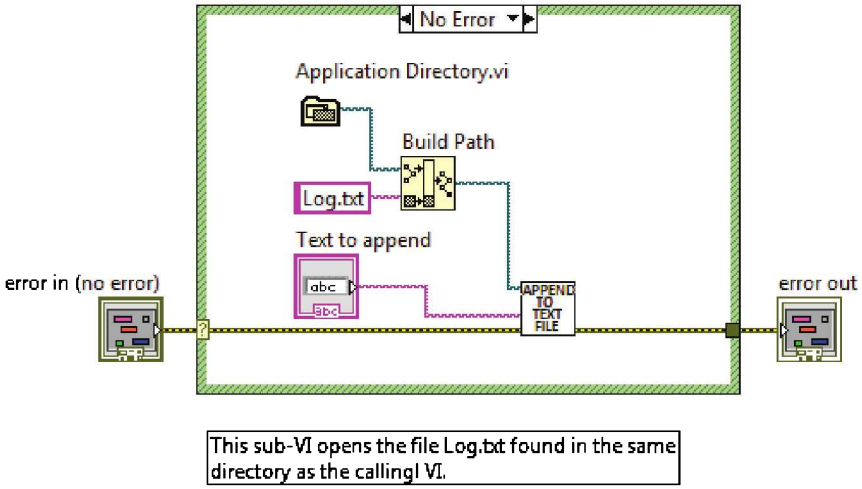


Figure 1.65: This sub-VI opens a log file in the same directory as the main application and appends data to it

to the log file in the application directory of the calling VI. This absolute path and the desired data string are then passed to the `Append To Text File` sub-VI shown in Figure 1.45. This sub-VI opens the file found at the absolute path and appends the input text to it on a new line followed by a carriage return.

1.24 Property Nodes and Invoke Nodes

The `Simulated Data Acquisition` specification requires that the front panel chart be cleared of old data when the VI is started. The VI also contains a STOP button, which should be initialized to False when the VI starts. These two tasks can be accomplished using property nodes and invoke nodes.

Front-panel objects have properties and methods. Properties are attributes that describe the object, while methods are things that the object can do. The properties of an object can be altered using property nodes. The methods of an object can be invoked using an invoke node. In the `Simulated Data Acquisition` VI, the *History* property node will be used to clear the front panel chart when the VI starts. The STOP button will be initialized using the *Reinitialize To Default* invoke node.

Property nodes and invoke nodes can be explicitly or implicitly linked to front panel objects. Implicitly linked nodes are tied to specific front panel objects and only

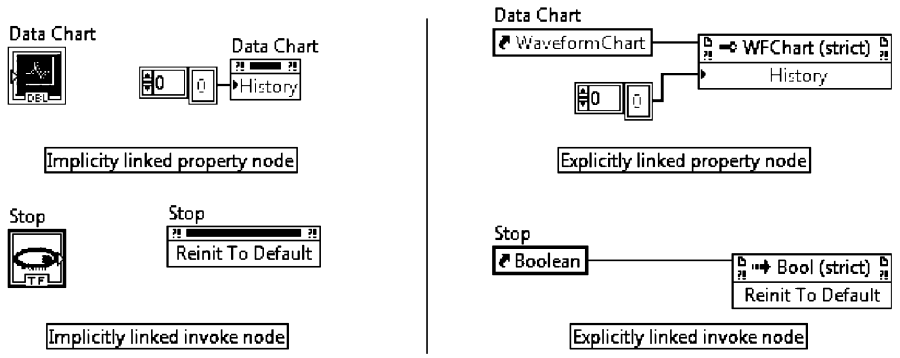


Figure 1.66: Property and invoke nodes used in the `Get Simulated Data` VI

can be used with those objects. Explicitly linked nodes are generic and must be linked to front panel objects by wiring control references to them. Property nodes can be configured to read or write a given property. Invoke nodes, being command-like, do not need to be configured as read or write.

Figure 1.66 shows the property and invoke nodes needed for the `Simulated Data Acquisition` VI. The left half of the figure shows implicitly linked nodes, and the right half of the figure shows explicitly linked nodes. In the `Simulated Data Acquisition` VI, we use the implicitly linked nodes—the explicitly linked nodes are shown for reference. The chart history can be cleared by wiring an array of zeros to the *History* property node of the chart. Clearing the chart history also clears the chart display. The STOP button can be initialized to its default state using the *Reinitialize To Default* invoke node. The default state of a Boolean control is False, so reinitializing the STOP button to its default state satisfies the requirements of the program specification.

1.25 Developing the `Simulated Data Acquisition` VI

We are now ready to develop the `Simulated Data Acquisition` VI. The block diagram of the VI is shown in Figure 1.67. When the VI begins, it clears the chart by wiring an array of zeros to the chart history using the *History* property node of the data chart. The VI also reinitializes the STOP button using the *Reinitialize To Default* invoke node of the STOP button. Notice the error wire that connects these two nodes to each other and to the main loop. The error wire enforces the dataflow for this sequence of operations. The property node executes first and then passes its error information to the invoke node. The invoke node then executes and passes its error information to the **`While`** loop. The error value initializes the error shift register of the loop and ensures that the loop does not start until both nodes have executed. The **`While`** loop has a second shift register to store the current value of the counter. The counter is used to keep track of the number of data points that have exceeded the

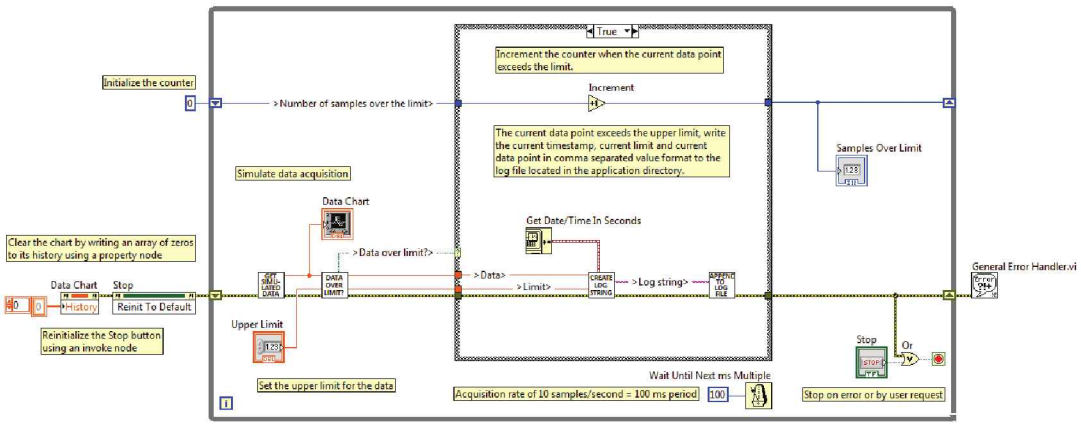


Figure 1.67: This VI writes the simulated data to a log file if it exceeds the user-specified limit

user-defined limit. The shift register for the counter is initialized to zero before the **`While`** loop begins.

The **`While`** loop starts after both shift registers have been initialized. The loop iteration rate is set by the **`Wait Until Next ms Multiple`** function, which is configured to iterate the loop every 100 milliseconds. Since the data will be sampled once per loop iteration, this is equivalent to a sampling rate of 10 times per second, as specified in the program specification.

At the beginning of each loop iteration, a new simulated data point is generated by the `Get Simulated Data` sub-VI shown in Figure 1.53. The new data point is wired to the chart on the front panel. The new data point is also passed to the `Data Over Limit` sub-VI shown in Figure 1.57. The user-defined limit is also passed to this sub-VI. The control containing the limit is placed inside the loop, so it is read each loop iteration. If the user changes the limit when the loop is running, the new value will passed to the `Data Over Limit` sub-VI.

The `Data Over Limit` sub-VI outputs a Boolean value that indicates whether the data exceeds the limit or not. If the data does exceed the limit, the current time stamp is generated using the **`Get Date/Time In Seconds`** function. This help file is shown in Figure 1.68. This time stamp is passed to the `Create Log String` sub-VI along with current data value and limit value. This sub-VI formats the comma-separated output string, as shown in Figure 1.59. The log string is then appended to the `Log.txt` file in the application directory using the `Append To Log File` sub-VI shown in Figure 1.65. The counter is incremented in parallel with these operations. If the simulated data does not exceed the limit, the case structure simply passes the error and counter information through, as shown in Figure 1.69.

After the case structure executes, the counter information is displayed, and the STOP button is read. If the STOP button has been pressed or if an error has occurred, the loop terminates; otherwise, the loop continues. The error data and the counter value are placed on their shift registers for the next loop iteration. When the loop finally terminates the error information is passed to the **`General Error Handler`** function.

Get Date/Time In Seconds Function

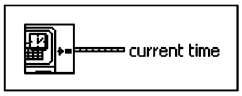
Owning Palette: Timing VIs and Functions

Requires: Base Package

Returns a timestamp of the current time. LabVIEW calculates this timestamp using the number of seconds elapsed since 12:00 a.m., Friday, January 1, 1904, Universal Time.

Use the To Double Precision Float function to convert the timestamp value to a lower precision, floating-point number.

Details



Add to the block diagram Find on the palette

current time returns a timestamp of the LabVIEW system time. LabVIEW calculates this timestamp using the number of seconds elapsed since 12:00 a.m., Friday, January 1, 1904, Universal Time [01-01-1904 00:00:00].

Get Date/Time In Seconds Details

Timer resolution is system dependent and might be less accurate than one millisecond, depending on your platform. When you perform timing, use the Tick Count (ms) function to improve resolution.

Figure 1.68: Help file for the `Get Date/Time In Seconds` function

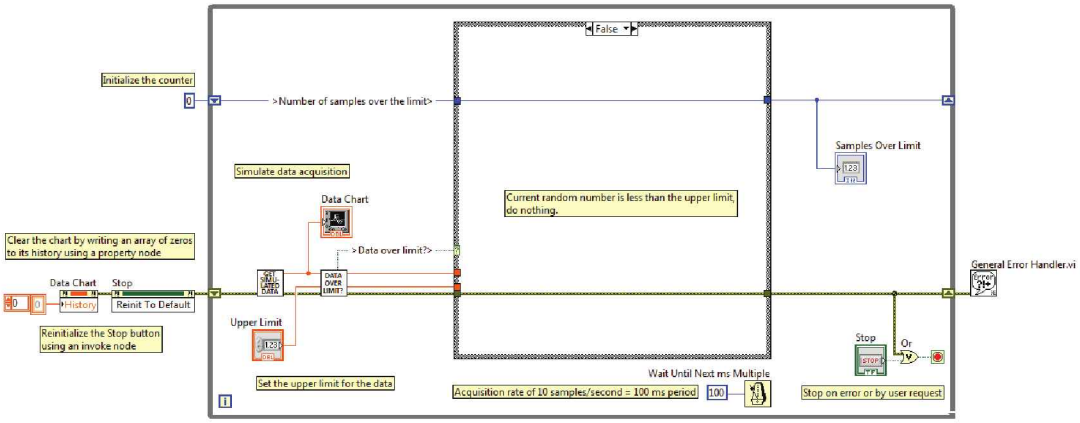


Figure 1.69: The log file is not accessed if the simulated data is below the specified limit

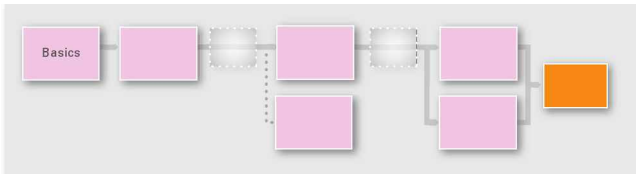
1.26 Elements of LabVIEW Style

Good programming style is important in any language. Good style improves the readability and maintainability of your code. Since LabVIEW is a graphical language, its style guidelines are especially important. In LabVIEW, spaghetti code really does look like spaghetti.

It is impossible to thoroughly cover the topic of good LabVIEW style in just one section of one chapter of one book. Luckily, the purpose of this book is not to explain good LabVIEW style, but to demonstrate it. All of the programs in this book have been constructed with simple style guidelines in mind. Hopefully, by reading this book, you will absorb these guidelines and apply them in your own work. If you do, you will find that good style is its own reward. Style guidelines are not arbitrary rules, they are based on the collected wisdom of the LabVIEW community. By adopting these style guidelines, you will be leveraging the experience of the community and applying it to your own work. Pitfalls that you may not have anticipated will be avoided, and you will benefit from advantages you may not have expected.

Here is a list of the style guidelines used in this book:

- Every program should occupy one screen (if possible) with no scrolling. If scrolling is absolutely necessary, construct your program so that scrolling is in one direction only.
- Keep wires as straight as possible.
- Align functions on the block diagram to improve readability.
- Make every effort to improve readability. For example, use **`Unbundle By Name`** instead of **`Unbundle`**.
- Every loop should have at least one descriptive free label.
- Every case of every case structure should have at least one descriptive free label.
- Sub-VIs should use the error-handling design pattern if possible.
- Every sub-VI block diagram should have at least one free label documenting the sub-VI function.
- All sub-VIs should use the $4 \times 2 \times 2 \times 4$ connector pane if possible.
- Connector panes should be wired with inputs on the left and outputs on the right.
- Error wires should use the lower corner terminals of the connector pane. References should use the upper corners of the connector pane.
- Every sub-VI should have a descriptive custom icon.
- Type definitions should be created for all custom clusters and all enumerated controls.
- The error chain should be continuous from the beginning of the VI to the end. If a function has error terminals, they should be wired. All errors wires should eventually merge with the main error wire.



2

Functional Specifications

Throughout the course of this book, we use two example codes to illustrate various LabVIEW design patterns. The first code is a simple but useful VI called the `Multitest` VI. The second code is a more complex code for controlling a car wash. The chapters in the book alternate between these two codes. The `Multitest` VI is used to introduce new concepts. The car wash controller is used to demonstrate how the new concepts are used in a more complex situation.

In this chapter, we present the functional specifications for these two codes. A functional specification is a formal document that describes the purpose and functioning of a computer program, its inputs and outputs, hardware specifications, etc. The writing of functional specifications is beyond the scope of this book, but an informal introduction to both programs is given here.

2.1 The `Multitest` VI

The `Multitest` VI is a framework for running two tests: *Test 1* and *Test 2*. The front panel of the VI is given by Figure 2.1. The front panel consists of three buttons: RUN TEST 1, RUN TEST 2, and STOP. *Test 1* is run when the user presses the RUN TEST 1 button. *Test 2* is run when the user presses the RUN TEST 2 button. The program ends when the user presses the STOP button.

The program waits for user input. The user should be able to run as many tests as desired in whatever order desired. The program makes no assumptions about test order or about the number of tests run. The tests themselves are simulated by displaying dialog windows to the user. When the TEST 1 button is pressed, a dialog window should appear that says "Test 1." Likewise, "Test 2" should be displayed when the RUN TEST 2 button is pressed. The dialog windows should have a "Done" button to allow the user to dismiss them. No dialog window should appear when the STOP button is pressed—the program should simply end. When the program starts, it should initialize all buttons on the front panel. The code should end gracefully if an error is detected, and it should report any errors to the user. The front panel should



Figure 2.1: Front panel of the `Multitest` VI

be responsive to user input to within 100 milliseconds. The program should not use excessive CPU time.

This `Multitest` VI is simple, but it is also useful. For example, imagine that you needed to write a program that performs data acquisition using a sensor. *Test 1* could contain code that calibrates the sensor and *Test 2* could contain the code that performs data acquisition using the sensor. Using the `Multitest` VI, the user could run multiple tests and have the option of recalibrating the sensor between tests as often as desired.

2.2 State Transition Diagram for the `Multitest` VI

A state transition diagram is a way of mapping out the functionality of a program at a higher level of abstraction than a flow chart. The various actions of a program are its "states." States describe the cohesive, independent actions that a program performs. In a state transition diagram, states are represented by ovals or rounded rectangles.

As a program executes, it transitions from one state to the next. These transitions can be caused by user input or by other forms of input such as sensor data, analysis of data, or a timed schedule. Anything that the computer can detect can be used as a trigger to transition from one state to another. In a state transition diagram, these transitions are designated by arrows.

State transition diagrams are not unique. A given functional specification could generate many valid state transition diagrams depending on the level of abstraction desired by the developer. One possible state transition diagram of the `Multitest` VI is shown in Figure 2.2. The filled circle at the top of the diagram is the starting point, and the filled circle within the open circle is the ending point of the state transition diagram.

The program starts with an *Initialize* state in which any start-up code is executed, such as initializing the front panel buttons. The code then transitions to a *Wait* state. The code continually checks to see if the front panel buttons have been pressed in this state. If the RUN TEST 1 button is pressed, the *Run Test 1* state executes and the "Test 1" dialog window is displayed. Likewise, if the RUN TEST 2 button is pressed, the *Run Test 2* state executes and the "Test 2" dialog window is displayed. Both of these states return to the *Wait* state when they have completed. When the user presses the STOP button (or if an error occurs), the state machine transitions to the *Shutdown* state, ending the program.

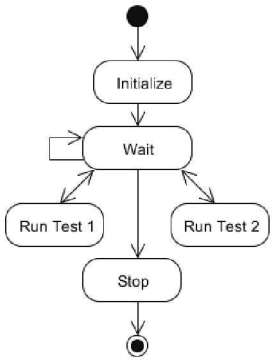


Figure 2.2: State transition diagram for the `Multitest` VI

2.3 The Car Wash Controller

The car wash controller has a significantly more complex specification than the `Multitest` VI. The functional specification for the car wash controller was developed by National Instruments as a test problem for the Certified LabVIEW Developer exam. Examinees were given four hours to produce a functional and well-documented code written in good LabVIEW style that satisfied the following specifications.

Task

Your company has received a contract to design a controller for a touchless car wash. A touchless car wash is one that relies on water spray only—there are no brushes. The specifications and rules are listed next. Your job is to design a controller using LabVIEW that satisfies all specifications.

Instructions

Using a front panel similar to the graphic in Figure 2.3, create a LabVIEW application that implements the actions of the procedure. For this application, the car wash switches are simulated by switches on the Front Panel. The car wash starts when a purchase switch has been selected. The execution of a cycle is denoted by illuminating the appropriate LED. For this application, each cycle's duration should be set to 5 seconds, unless otherwise stated in the operating rules that follow.

Purchase Options

- $5 Deluxe Wash
- $3 Economy Wash

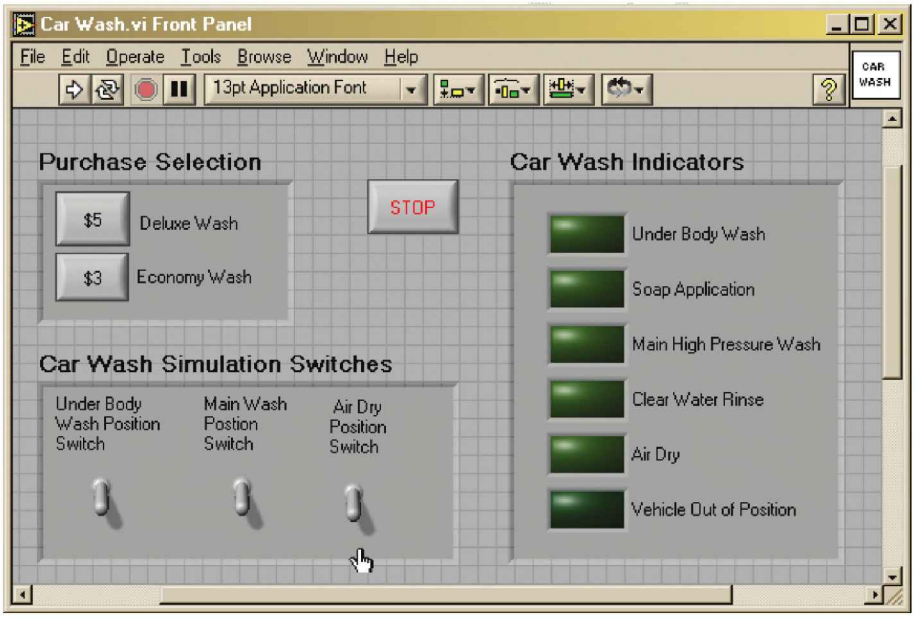


Figure 2.3: Car wash controller Front Panel

Cycles and Associated Timing

1. Underbody Wash—10 seconds
2. Soap Application—5 seconds
3. Main High-Pressure Wash—5 seconds
4. Clear Water rinse—5 seconds
5. Air Dry Cycle—10 seconds

Car Wash Operating Rules

1. Only one purchase selection can be made at a time. The car wash should not accept a second purchase selection while the car wash is in operation.
2. Not all cycles are performed for each wash. The more expensive wash performs more cycles. The list of cycles performed are
   - Deluxe Wash: 1–2–3–4–5
   - Economy Wash: 2–3–4
3. Each cycle is initiated by a switch as shown in Figure 2.4. If the vehicle rolls off of the switch, the wash immediately pauses and illuminates an indication to the driver to re-position the vehicle. The amount of time that expires while the car is out of position should not count against the wash time.

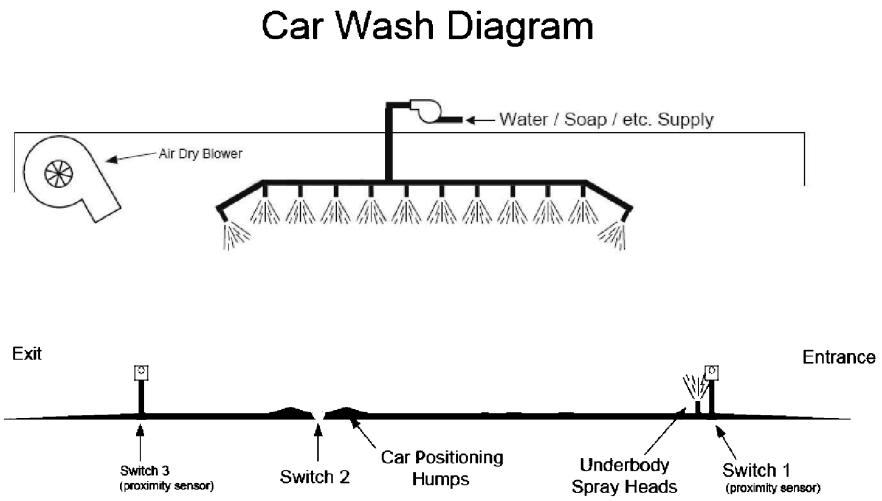


Figure 2.4: Car Wash controller schematic

4. Underbody Wash Cycle: The spray heads for this wash are located near the entrance of the car wash. The underbody spray heads are fixed in position and require the vehicle be slowly driven over them to wash the underbody of the vehicle. The Underbody Wash is activated under the following conditions:
   - The Deluxe Wash has been purchased.
   - The car wash is in the Underbody Wash cycle.
   - UNDERBODY WASH position switch is closed (proximity switch).

   This cycle of the wash should last for 10 seconds. Upon completion of this cycle, the controller should signal moving to the next cycle by activating the VEHICLE OUT OF POSITION LED.
5. Main Wash Cycle: The MAIN WASH position switch verifies the vehicle is in the correct location in order for the wash cycles (cycles 2, 3, and 4) to operate. Each cycle should last for 5 seconds. If the vehicle rolls off of the MAIN WASH position switch, the wash immediately pauses and illuminates an indication to the driver to re-position the vehicle. The amount of time that expires while the car is out of position should not count against the wash time. The wash resumes after the vehicle is properly positioned. Upon completion of this cycle, the controller should signal moving to the next cycle by activating the VEHICLE OUT OF POSITION LED.
6. Air Dry Cycle: The air drier is a set of fixed-position blowers located near the exit of the car wash. They require the vehicle be driven slowly out of the car wash through the air stream to dry. The Air Dry cycle activates on the following conditions:
   - The Deluxe Wash has been purchased.
   - The car wash has reached the Air Dry cycle.
   - AIR DRY position switch is closed (proximity switch).

If the vehicle rolls off of the AIR DRY position switch, the cycle immediately pauses and illuminates an indication to the driver to re-position the vehicle. The amount of time that expires while the car is out of position should not count against the air dry time. The cycle resumes after the vehicle is properly positioned. This cycle of the wash should last for 10 seconds. Upon completion of this cycle, the controller should allow the next vehicle in line to select another wash.

7. The car wash must respond to the STOP Boolean and Vehicle Position switches within 100 milliseconds. The STOP button aborts the operation of the VI.

2.4 State Transition Diagrams for the Car Wash Controller

In the discussion of the state transition diagram for the `Multitest` VI, it was mentioned that state transition diagrams are not unique. To demonstrate this, two state transition diagrams will be developed for the car wash controller. Both are valid, but they operate at different levels of abstraction.

The first diagram is shown in Figure 2.5. This diagram is a very literal representation of the specification. Since each state exists only once in the physical car wash, each state is listed only once in the diagram. The transition arrows indicate the paths

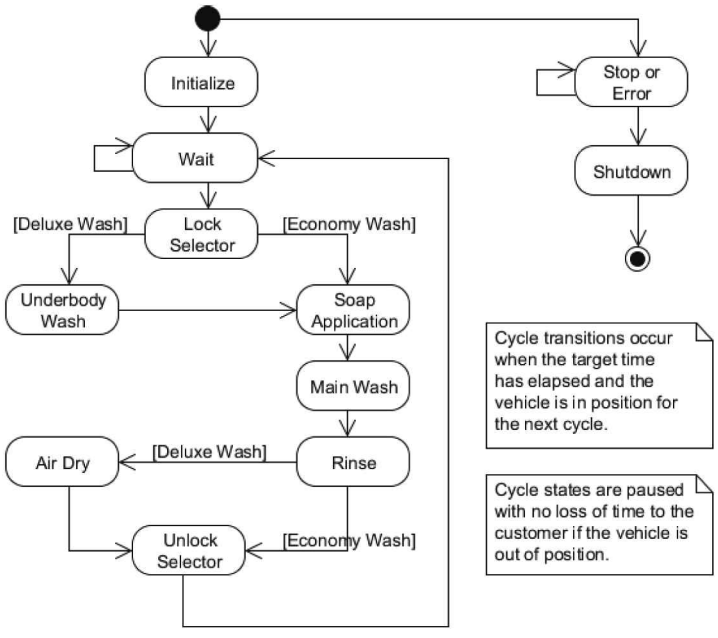


Figure 2.5: Car wash controller state diagram using a literal interpretation of the state sequence; since each state exists only once in the car wash, each state is represented once on the diagram

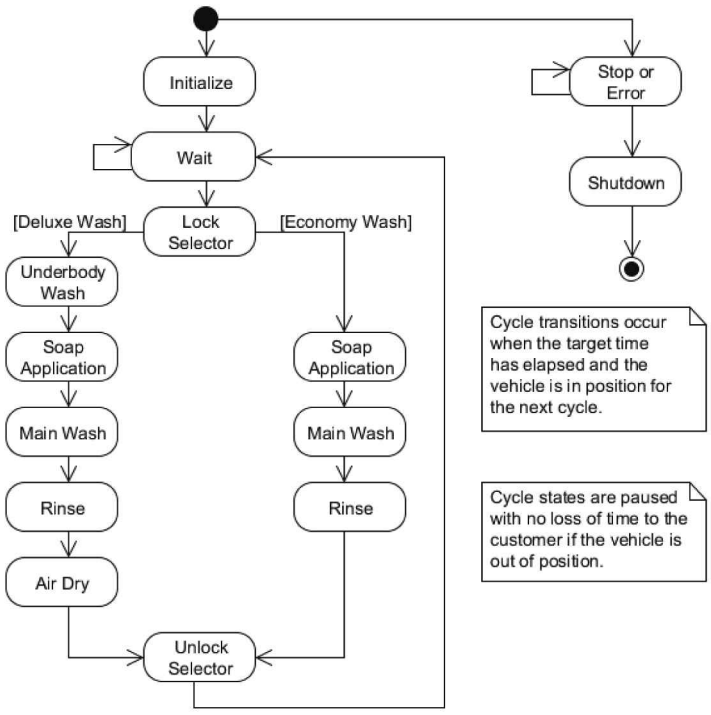
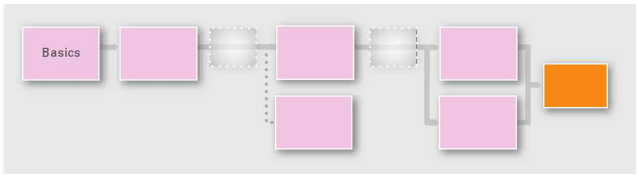


Figure 2.6: Car wash controller state diagram using states grouped by car wash type, where some states are repeated on the diagram

taken by the Deluxe and Economy Washes. This version of the diagram is the one that will be used by the classic state machine and the event-driven implementations of the car wash controller.

The second diagram is shown in Figure 2.6. This diagram takes a more abstract approach to the car wash controller. In this diagram, the states are grouped by car wash type. Since some states are common to both the Deluxe and the Economy Wash, some states appear twice in the diagram even though they physically appear only once in the car wash. This diagram also will be used by the car wash controllers that use queues in their implementations.

Notice that both of these state transition diagrams have two parallel branches. The branch on the left is used to show the main work of the code. The branch on the right indicates that the program must exit gracefully whenever an error is detected or the STOP button is pressed. This can happen at any time from any state, so a parallel path is used to indicate that these actions can interrupt the normal operation of the left branch and end the program.



2

Functional Specifications

Throughout the course of this book, we use two example codes to illustrate various LabVIEW design patterns. The first code is a simple but useful VI called the `Multitest` VI. The second code is a more complex code for controlling a car wash. The chapters in the book alternate between these two codes. The `Multitest` VI is used to introduce new concepts. The car wash controller is used to demonstrate how the new concepts are used in a more complex situation.

In this chapter, we present the functional specifications for these two codes. A functional specification is a formal document that describes the purpose and functioning of a computer program, its inputs and outputs, hardware specifications, etc. The writing of functional specifications is beyond the scope of this book, but an informal introduction to both programs is given here.

2.1 The `Multitest` VI

The `Multitest` VI is a framework for running two tests: *Test 1* and *Test 2*. The front panel of the VI is given by Figure 2.1. The front panel consists of three buttons: RUN TEST 1, RUN TEST 2, and STOP. *Test 1* is run when the user presses the RUN TEST 1 button. *Test 2* is run when the user presses the RUN TEST 2 button. The program ends when the user presses the STOP button.

The program waits for user input. The user should be able to run as many tests as desired in whatever order desired. The program makes no assumptions about test order or about the number of tests run. The tests themselves are simulated by displaying dialog windows to the user. When the TEST 1 button is pressed, a dialog window should appear that says "Test 1." Likewise, "Test 2" should be displayed when the RUN TEST 2 button is pressed. The dialog windows should have a "Done" button to allow the user to dismiss them. No dialog window should appear when the STOP button is pressed—the program should simply end. When the program starts, it should initialize all buttons on the front panel. The code should end gracefully if an error is detected, and it should report any errors to the user. The front panel should



Figure 2.1: Front panel of the `Multitest` VI

be responsive to user input to within 100 milliseconds. The program should not use excessive CPU time.

This `Multitest` VI is simple, but it is also useful. For example, imagine that you needed to write a program that performs data acquisition using a sensor. *Test 1* could contain code that calibrates the sensor and *Test 2* could contain the code that performs data acquisition using the sensor. Using the `Multitest` VI, the user could run multiple tests and have the option of recalibrating the sensor between tests as often as desired.

2.2 State Transition Diagram for the `Multitest` VI

A state transition diagram is a way of mapping out the functionality of a program at a higher level of abstraction than a flow chart. The various actions of a program are its "states." States describe the cohesive, independent actions that a program performs. In a state transition diagram, states are represented by ovals or rounded rectangles.

As a program executes, it transitions from one state to the next. These transitions can be caused by user input or by other forms of input such as sensor data, analysis of data, or a timed schedule. Anything that the computer can detect can be used as a trigger to transition from one state to another. In a state transition diagram, these transitions are designated by arrows.

State transition diagrams are not unique. A given functional specification could generate many valid state transition diagrams depending on the level of abstraction desired by the developer. One possible state transition diagram of the `Multitest` VI is shown in Figure 2.2. The filled circle at the top of the diagram is the starting point, and the filled circle within the open circle is the ending point of the state transition diagram.

The program starts with an *Initialize* state in which any start-up code is executed, such as initializing the front panel buttons. The code then transitions to a *Wait* state. The code continually checks to see if the front panel buttons have been pressed in this state. If the RUN TEST 1 button is pressed, the *Run Test 1* state executes and the "Test 1" dialog window is displayed. Likewise, if the RUN TEST 2 button is pressed, the *Run Test 2* state executes and the "Test 2" dialog window is displayed. Both of these states return to the *Wait* state when they have completed. When the user presses the STOP button (or if an error occurs), the state machine transitions to the *Shutdown* state, ending the program.

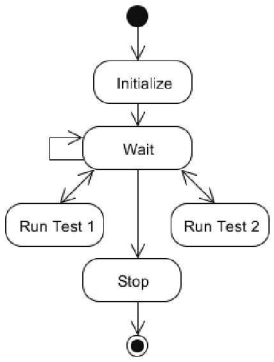


Figure 2.2: State transition diagram for the `Multitest` VI

2.3 The Car Wash Controller

The car wash controller has a significantly more complex specification than the `Multitest` VI. The functional specification for the car wash controller was developed by National Instruments as a test problem for the Certified LabVIEW Developer exam. Examinees were given four hours to produce a functional and well-documented code written in good LabVIEW style that satisfied the following specifications.

Task

Your company has received a contract to design a controller for a touchless car wash. A touchless car wash is one that relies on water spray only—there are no brushes. The specifications and rules are listed next. Your job is to design a controller using LabVIEW that satisfies all specifications.

Instructions

Using a front panel similar to the graphic in Figure 2.3, create a LabVIEW application that implements the actions of the procedure. For this application, the car wash switches are simulated by switches on the Front Panel. The car wash starts when a purchase switch has been selected. The execution of a cycle is denoted by illuminating the appropriate LED. For this application, each cycle's duration should be set to 5 seconds, unless otherwise stated in the operating rules that follow.

Purchase Options

- $5 Deluxe Wash
- $3 Economy Wash

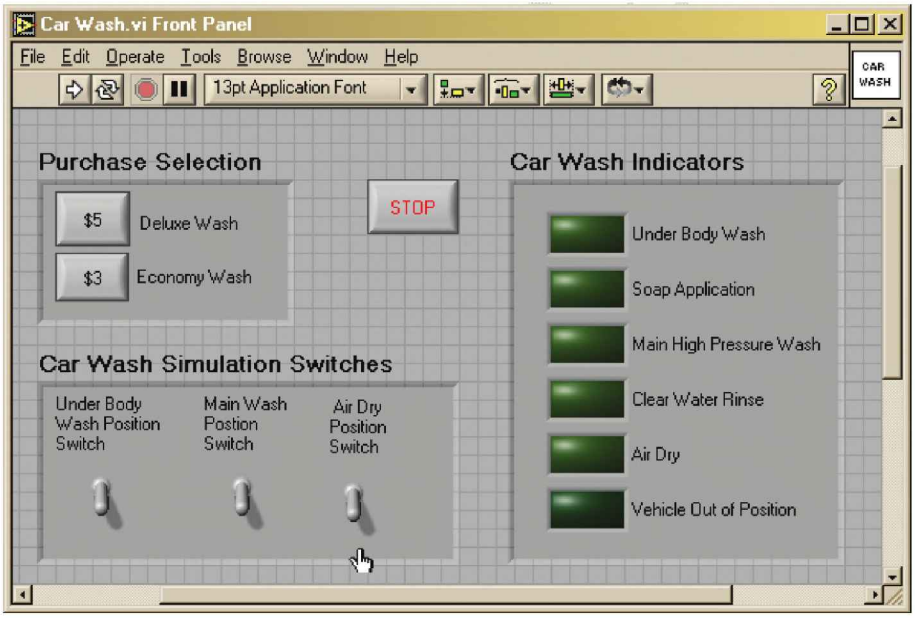


Figure 2.3: Car wash controller Front Panel

Cycles and Associated Timing

1. Underbody Wash—10 seconds
2. Soap Application—5 seconds
3. Main High-Pressure Wash—5 seconds
4. Clear Water rinse—5 seconds
5. Air Dry Cycle—10 seconds

Car Wash Operating Rules

1. Only one purchase selection can be made at a time. The car wash should not accept a second purchase selection while the car wash is in operation.
2. Not all cycles are performed for each wash. The more expensive wash performs more cycles. The list of cycles performed are
   - Deluxe Wash: 1–2–3–4–5
   - Economy Wash: 2–3–4
3. Each cycle is initiated by a switch as shown in Figure 2.4. If the vehicle rolls off of the switch, the wash immediately pauses and illuminates an indication to the driver to re-position the vehicle. The amount of time that expires while the car is out of position should not count against the wash time.

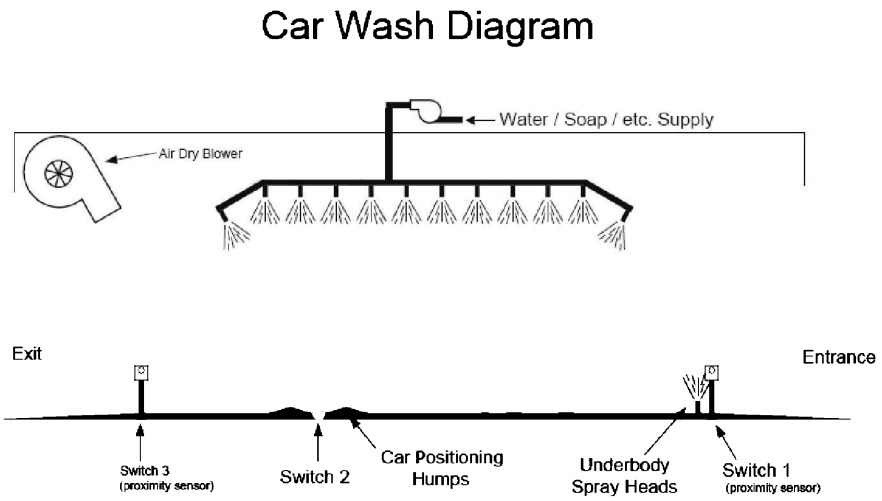


Figure 2.4: Car Wash controller schematic

4. Underbody Wash Cycle: The spray heads for this wash are located near the entrance of the car wash. The underbody spray heads are fixed in position and require the vehicle be slowly driven over them to wash the underbody of the vehicle. The Underbody Wash is activated under the following conditions:
   - The Deluxe Wash has been purchased.
   - The car wash is in the Underbody Wash cycle.
   - UNDERBODY WASH position switch is closed (proximity switch).

   This cycle of the wash should last for 10 seconds. Upon completion of this cycle, the controller should signal moving to the next cycle by activating the VEHICLE OUT OF POSITION LED.
5. Main Wash Cycle: The MAIN WASH position switch verifies the vehicle is in the correct location in order for the wash cycles (cycles 2, 3, and 4) to operate. Each cycle should last for 5 seconds. If the vehicle rolls off of the MAIN WASH position switch, the wash immediately pauses and illuminates an indication to the driver to re-position the vehicle. The amount of time that expires while the car is out of position should not count against the wash time. The wash resumes after the vehicle is properly positioned. Upon completion of this cycle, the controller should signal moving to the next cycle by activating the VEHICLE OUT OF POSITION LED.
6. Air Dry Cycle: The air drier is a set of fixed-position blowers located near the exit of the car wash. They require the vehicle be driven slowly out of the car wash through the air stream to dry. The Air Dry cycle activates on the following conditions:
   - The Deluxe Wash has been purchased.
   - The car wash has reached the Air Dry cycle.
   - AIR DRY position switch is closed (proximity switch).

If the vehicle rolls off of the AIR DRY position switch, the cycle immediately pauses and illuminates an indication to the driver to re-position the vehicle. The amount of time that expires while the car is out of position should not count against the air dry time. The cycle resumes after the vehicle is properly positioned. This cycle of the wash should last for 10 seconds. Upon completion of this cycle, the controller should allow the next vehicle in line to select another wash.

7. The car wash must respond to the STOP Boolean and Vehicle Position switches within 100 milliseconds. The STOP button aborts the operation of the VI.

2.4 State Transition Diagrams for the Car Wash Controller

In the discussion of the state transition diagram for the `Multitest` VI, it was mentioned that state transition diagrams are not unique. To demonstrate this, two state transition diagrams will be developed for the car wash controller. Both are valid, but they operate at different levels of abstraction.

The first diagram is shown in Figure 2.5. This diagram is a very literal representation of the specification. Since each state exists only once in the physical car wash, each state is listed only once in the diagram. The transition arrows indicate the paths

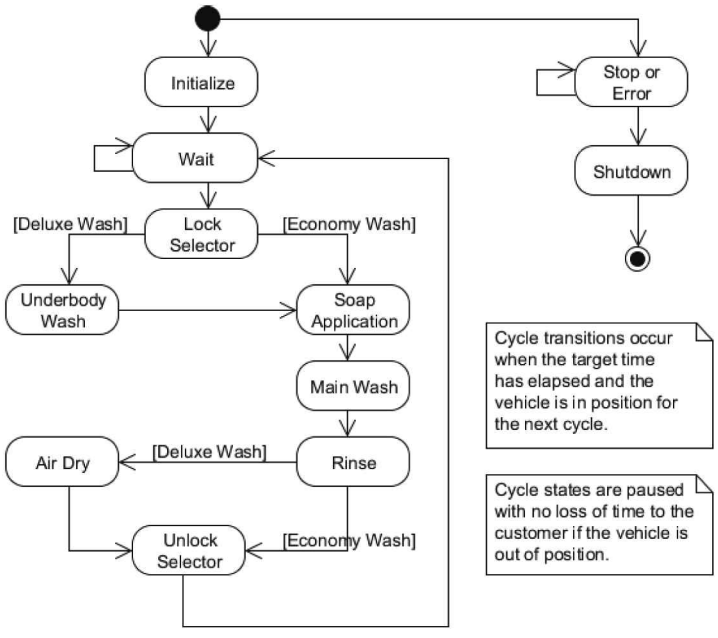


Figure 2.5: Car wash controller state diagram using a literal interpretation of the state sequence; since each state exists only once in the car wash, each state is represented once on the diagram

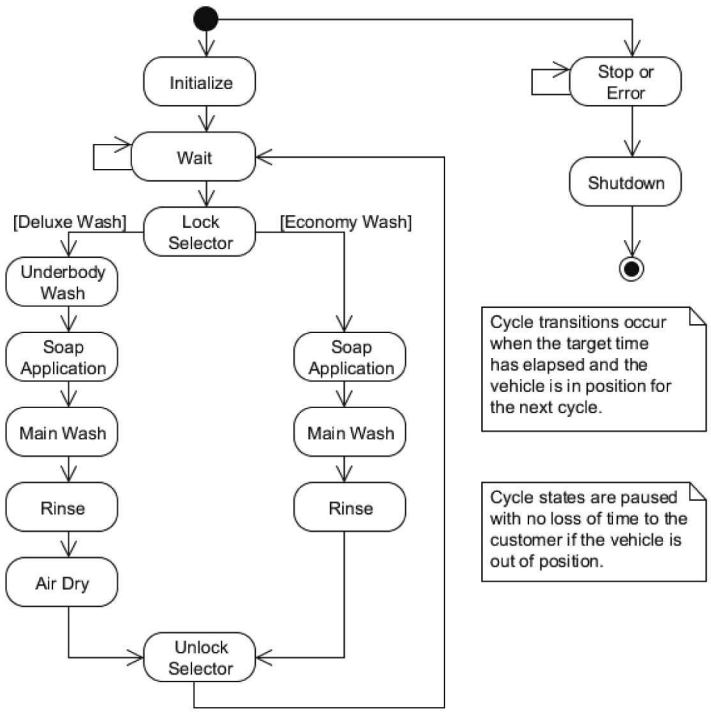
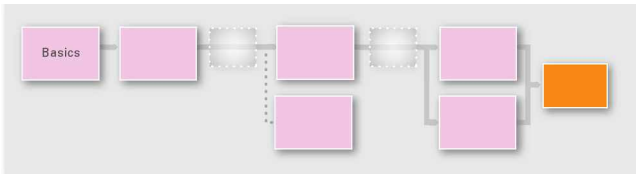


Figure 2.6: Car wash controller state diagram using states grouped by car wash type, where some states are repeated on the diagram

taken by the Deluxe and Economy Washes. This version of the diagram is the one that will be used by the classic state machine and the event-driven implementations of the car wash controller.

The second diagram is shown in Figure 2.6. This diagram takes a more abstract approach to the car wash controller. In this diagram, the states are grouped by car wash type. Since some states are common to both the Deluxe and the Economy Wash, some states appear twice in the diagram even though they physically appear only once in the car wash. This diagram also will be used by the car wash controllers that use queues in their implementations.

Notice that both of these state transition diagrams have two parallel branches. The branch on the left is used to show the main work of the code. The branch on the right indicates that the program must exit gracefully whenever an error is detected or the STOP button is pressed. This can happen at any time from any state, so a parallel path is used to indicate that these actions can interrupt the normal operation of the left branch and end the program.



3

Problem Set 1

3.1 Car Wash Controller Exercises

Exercise 1.1: Creating Type-Defined Clusters for the Car Wash Controller

Create the car wash controller front panel shown in Figure 2.3. The purchase selection buttons, the simulation switches, and the car wash indicators should all be placed in clusters. Align them as shown in the figure. Save them as type-defined controls.

The labels shown above the three simulation switches are quite long. It would be awkward to use LabVIEW labels for the switches. Do you see why? Captions are a better choice here. Take the text shown in the figure and use it as the captions for the three switches. For the labels of the switches use *Under Body Wash*, *Main Wash*, and *Air Dry*. ■

Exercise 1.2: Tip Strips and Descriptions

Create tip strips and descriptions for all of the controls that you created in the first exercise. Create tip strips and descriptions for the clusters containing the switches and indicators as well. Save them as strict type definitions to ensure that the documentation you just created will carry over to any instances of the controls. ■

3.2 Traffic Light Controller Exercises

Exercise 1.3: Traffic Light States

Read the functional specification for the traffic light controller in Appendix A. What states will the program need? Use the light pattern charts in the specification as a guide. Are there any needed states that are not related to the light pattern charts? Create a type-defined enumerated control containing the necessary states. ■

Exercise 1.4: Traffic Light State Transition Diagram

Draw a state transition diagram for the states you listed in the previous exercise. Remember that any state can transition to the *Shutdown* state if the STOP button is pressed. Use the state transition diagrams in this chapter as a guide. Create a type-defined enumerated constant containing these states. ■

Exercise 1.5: Exploiting the Symmetry of the Traffic Light Controller

The front panel of the traffic light controller is shown in Figure A.1. Each traffic light signal on the front panel is a cluster of three Boolean indicators: one red, one yellow and one green. Create a strict type definition of this cluster for later use. The left-turn lights are an identical cluster with the addition of leftward arrows. Create the arrows using the *Decorations* sub-palette of the *Controls* palette. Create a strict type definition of the the left-turn cluster as well.

The front panel of the traffic controller contains eight traffic light clusters, but due to the symmetry of the problem, there will only be four unique clusters displayed at any given time since the north and south signals are identical and the east and west signals are identical. Create a type-defined cluster of clusters that contains the four unique signal patterns: N-S, N-S LEFT, E-W, and E-W LEFT, as shown in Figure 3.1. This one cluster will contain all of the information needed to determine the status of every traffic light indicator on the front panel.

The traffic light controller must monitor four vehicle sensors. Rather than use four wires to pass on the data from these sensors, it will be convenient to store them in a cluster and pass all four values on one wire. This information will be used by the traffic light controller to decide whether the normal sequence of lights needs to be followed or if the left-turn sequence lights needs to be followed instead. The symmetry

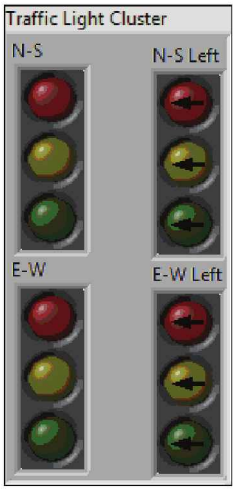


Figure 3.1: The type-defined traffic light cluster

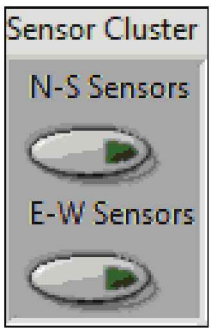


Figure 3.2: The type-defined vehicle sensor cluster

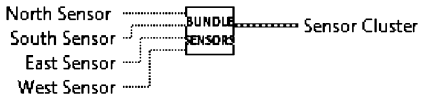


Figure 3.3: Icon for the `Bundle Sensors` sub-VI

of the problem can be used to reduce the amount of information stored in the cluster. Rather than store all four sensors, create a type-defined cluster with two Boolean controls: a N-S SENSORS control and an E-W SENSORS control, as shown in Figure 3.2. The N-S SENSORS Boolean is True if either the NORTH SENSOR or the SOUTH SENSOR is True. Likewise, the E-W SENSORS Boolean is True if either the EAST SENSOR or the WEST SENSOR is True.

Next create the `Bundle Sensors` sub-VI. This icon is shown in Figure 3.3. This sub-VI accepts the values of the four vehicle sensors and outputs the type-defined cluster shown in Figure 3.2. ■

Exercise 1.6: Create the `Next Traffic State` Sub-VI

When the traffic light controller is in a given state, it needs a way to determine which state, to go to next. To do this, create the `Next Traffic State` sub-VI with the icon shown in Figure 3.4. This sub-VI accepts the enumerated constant containing the current state developed in Exercise 1.4 as an input. It also accepts a DONE? Boolean and the sensor cluster developed in Exercise 1.5. The output of this sub-VI is a NEXT STATE enumerated constant indicating the next state. If the DONE? Boolean is False, the `Next State` should equal `Current State`. If the DONE? button is True, the sub-VI needs to use the current state and the values of the vehicle sensors to determine the next state.

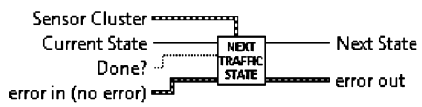


Figure 3.4: Icon for the `Next Traffic State` sub-VI ■

3.3 Security System Controller Exercises

Exercise 1.7: Security System States

Read the functional specification for the security system controller in Appendix B. What states will the program need? Create a type-defined enumerated control containing these states. ■

Exercise 1.8: Security System State Transition Diagram

Draw a state transition diagram for the security system states you listed in the previous exercise. Remember that any state can transition to the *Shutdown* state if the STOP button is pressed. Use the state transition diagrams in Chapter 2 as a guide. ■

Exercise 1.9: Creating a *Status* Enumerated Control

A security system zone can only be in one of four states. What are these states? Create a type-defined enumerated control containing these states. ■

Exercise 1.10: Create the `Initialize Status Array From Cluster` Sub-VI

It is often desirable to preallocate and initialize an array when a program starts. Preallocation doesn't usually affect the logic of a program, but it can increase performance speed and improve memory usage. In the security system controller, the array of zone status enumerated controls will always have the same number of elements as the switches clusters, since both are determined by the number of zones. The cluster size is fixed, so when the controller starts, LabVIEW knows the cluster size. The zone status array size is not known when the program starts unless it is preallocated.

Create the `Initialize Status Array From Cluster` sub-VI with the icon shown in Figure 3.5. This sub-VI accepts a switches cluster (or the local variable of a switches cluster) as an input and produces a zone status array with the same number of elements as an output. Initialize all of the elements of the output array to *Normal*. The sub-VI should not assume a particular number of zones. You might find the following functions helpful: **`Cluster To Array`** (see the help file in Figure 3.6), **`Array Size`** (see the help file in Figure 3.7), and **`Initialize Array`** (see the help file in Figure 3.8).

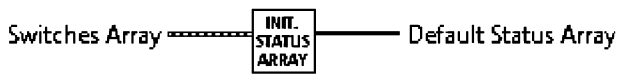


Figure 3.5: The icon for the `Initialize Status Array From Cluster` sub-VI

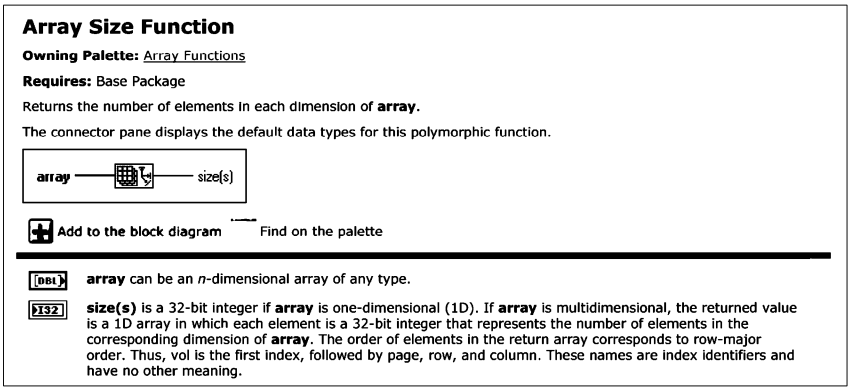


Figure 3.6: Help file for the `Cluster To Array` function

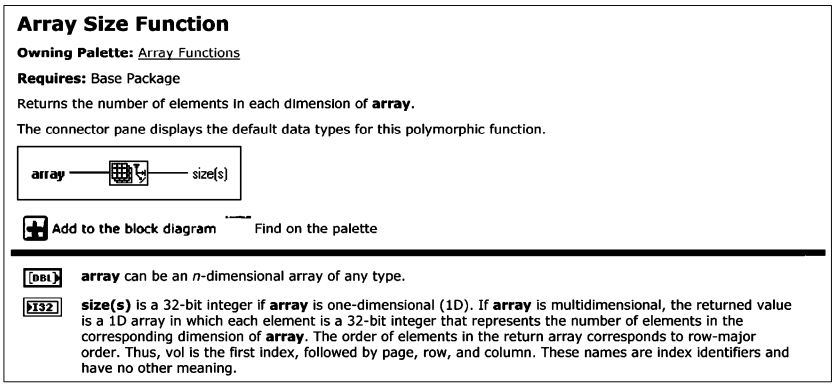


Figure 3.7: Help file for the `Array Size` function

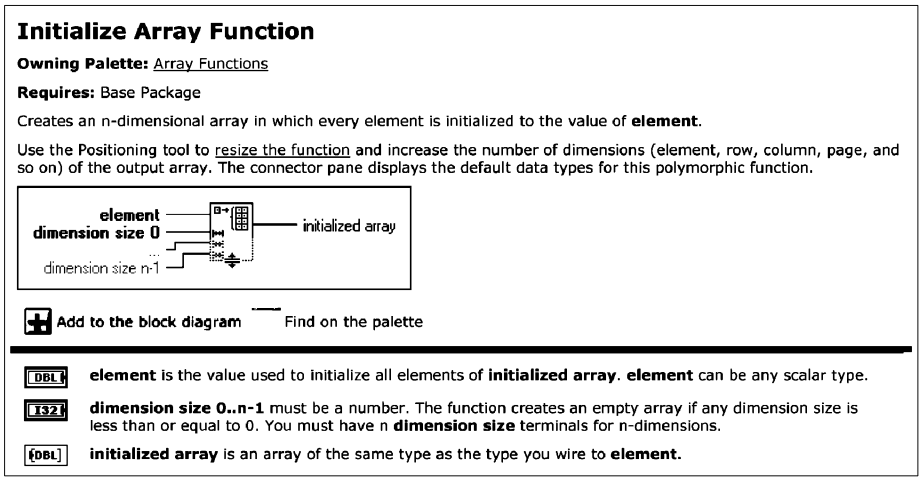


Figure 3.8: Help file for the `Initialize Array` function ■

Exercise 1.11: Create the `Reset Zone Array` Sub-VI

Create a `Reset Zone Array` sub-VI. The icon is shown in Figure 3.9. This sub-VI will be used to reset the zones of the security system to *Normal*. This sub-VI accepts an array of the zone status enumerated constants created in Exercise 1.9. The size of the array is not known until the program is run. This sub-VI should return an array of zone status enumerated constants all set to *Normal*. The output array should be the same size as the input array.

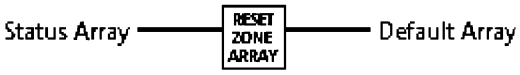


Figure 3.9: The icon for the `Reset Zone Array` sub-VI ■

Exercise 1.12: Create the `Get Zone Status` Sub-VI

Create the `Get Zone Status` sub-VI with the icon shown in Figure 3.10. This sub-VI accepts three Boolean inputs: *Tamper*, *Bypass*, and *Alarm*. The output of the sub-VI is a zone status enum that represents the status of the zone. Following the logic specified in the functional specification, this sub-VI should have the following behavior:

(a) If the tamper input is True, the output is *Tamper*.

(b) If the tamper input is False and the bypass input is True, the output is *Bypass*.

(c) If both the tamper and bypass inputs are False and the alarm input is True, the output is *Alarm*.

(d) If all three inputs are false, the output is *Normal*.

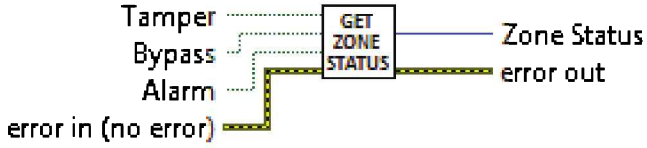


Figure 3.10: The icon for the `Get Zone Status` sub-VI ■

Exercise 1.13: Create the `Get Status Of All Zones` Sub-VI

Use the `Get Zone Status` sub-VI developed in Exercise 1.12 to create the `Get Status Of All Zones` sub-VI. The icon is shown in Figure 3.11. This sub-VI

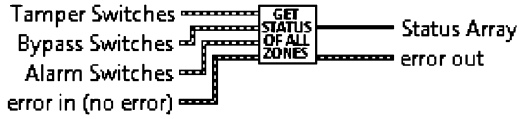


Figure 3.11: The icon for the `Get Status Of All Zones` sub-VI

accepts three clusters of switches (TAMPER, BYPASS, and ALARM) and calculates the status of each zone in the security system. The output of this sub-VI is an array of zone status enumerated constants with one enumerated constant for each zone. The sub-VI should not assume a particular number of zones, since the number of zones may change in the future. You may find the **`Cluster To Array`** function useful (see the help file in Figure 3.6). ■

Exercise 1.14: Writing to the Security System Log File

The security system functional specification states that when the status of a zone changes the new status needs to be written to a log file. The log file is an ASCII text file in a spreadsheet-readable format. For our purposes, we will use a four-column comma-separated-value (csv) format for the text file with fields for date, time (including seconds), zone number of the changed zone, and new status of the changed zone.

Create the `Write To Log File` sub-VI with the icon shown in Figure 3.12. This sub-VI accepts a timestamp, zone number, and new zone status enumerated constant as inputs. It formats them into the comma-separated string required by the functional specification and appends it to the end of the log file with a trailing carriage return. Use the `Append To Text File` sub-VI shown in Figure 1.45 to write the string to the log file. You might find the **`Get Date/Time String`** function and the **`Format Into String`** functions useful (see the help files in Figures 1.61 and 1.60).

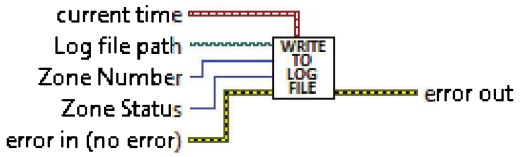


Figure 3.12: The icon for the `Write To Log File` sub-VI ■

3.4 Boiler Controller Exercises

Exercise 1.15: Boiler Controller States

Read the functional specification for the boiler controller in Appendix C. What states will the program need? Create a type-defined enumerated control containing these states. ■

Exercise 1.16: Boiler Controller State Transition Diagram

Draw a state transition diagram for the boiler controller states you listed in the previous exercise. Remember that any state can transition to the *Shutdown* state if the STOP button is pressed. Use the state transition diagrams in this chapter as a guide. ■

Exercise 1.17: Create Type-Defined Clusters for the Boiler Controller

Create the boiler controller front panel shown in Figure C.1. The boiler controls, boiler simulation controls, and boiler status indicators all should be placed in clusters. Align them as shown in the figure. Save them as type-defined controls. Also create a LIMITS cluster that contains the following numerical controls: FLAME SENSOR LOW, FUEL CONTROL HIGH, FUEL CONTROL LOW, PRE-PURGE TIME (S), and PURGE TIME (S). Save this cluster as a type-defined control. This cluster does not appear on the front panel of the boiler controller, but it will be used on the block diagram. ■

Exercise 1.18: Create the `Application Run Done` Sub-VI for the Boiler Controller

Create the `Application Run Done` sub-VI with the icon shown in Figure 3.13. This sub-VI accepts the BOILER CONTROLS cluster and the BOILER SIMULATION CONTROLS clusters as inputs and outputs a READY Boolean indicator. This sub-VI should check the BOILER RESET control and the RUN INTERLOCK control. If both of these controls are True, then READY should be True. Otherwise, READY should be False. Use the sub-VI with the error-handling design pattern.

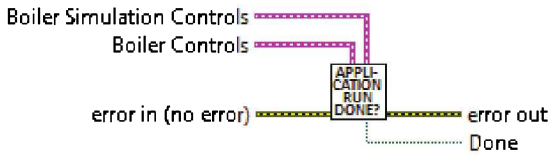


Figure 3.13: The icon of the `Application Run Done` VI ■

Exercise 1.19: Create the `Application Run Output` Sub-VI for the Boiler Controller

Create the `Application Run Output` sub-VI with the icon shown in Figure 3.14. This sub-VI accepts a BOILER STATUS cluster as an input. This sub-VI should output a BOILER STATUS cluster and a log string. This sub-VI should perform two tasks. First, it should bundle the string `Lockout` to the STATUS string of the input BOILER STATUS cluster. Second, it should output the log string `Boiler Initialized, 0`. Use the sub-VI with the error-handling design pattern.

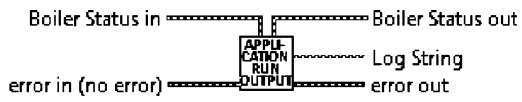


Figure 3.14: The icon of the `Application Run Output` VI ■

Exercise 1.20: Create the `Reset Lockout Done` Sub-VI for the Boiler Controller

Create the `Reset Lockout Done` sub-VI with the icon shown in Figure 3.15. This sub-VI accepts a BOILER CONTROLS cluster as an input and outputs a DONE Boolean indicator. This sub-VI should unbundle the START SEQUENCE control and assign its value to the READY indicator. Use the sub-VI with the error-handling design pattern.

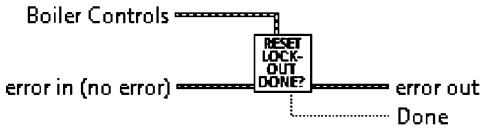


Figure 3.15: The icon of the `Reset Lockout Done` VI ■

Exercise 1.21: Create the `Reset Lockout Output` Sub-VI for the Boiler Controller

Create the `Reset Lockout Output` sub-VI whose icon is shown in Figure 3.16. This sub-VI accepts a BOILER STATUS cluster as an input and bundles the string `Ready` to the STATUS control. This sub-VI should also output `Boiler Ready, 0` as a log string. Use the sub-VI with error-handling design pattern.

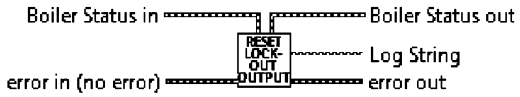


Figure 3.16: The icon of the `Reset Lockout Output` VI ■

Exercise 1.22: Creating a Rounding Sub-VI for the Boiler Controller

Two of the boiler states are timed, and the functional specification states that they will be displayed to the nearest tenth of a second. There are times when rounding errors may cause the state machine to make an extra iteration. For example, suppose that the time limit for a state is 10 seconds and the loop iterates every 100 milliseconds. Suppose further that the elapsed time is 9.99997 seconds. If your sub-VI requires that the elapsed time is greater than or equal to 10 seconds, this elapsed time would fail the test. This problem can be solved by rounding the elapsed time to the nearest 0.1 seconds. The rounded elapsed time would be 10.0 seconds, and the elapsed time would pass the test.

Create the `Round To Nearest Tenths Place` sub-VI with the icon shown in Figure 3.17. This sub-VI accepts a double-precision, floating-point number as an input and outputs the same number rounded to the nearest tenth. Use the following algorithm: multiply the number by ten, round the result to the nearest integer, and then divide rounded result by 10. You might find the **`Round To Nearest`** function useful (see the help file in Figure 3.18).

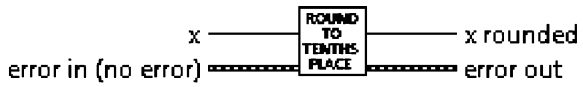
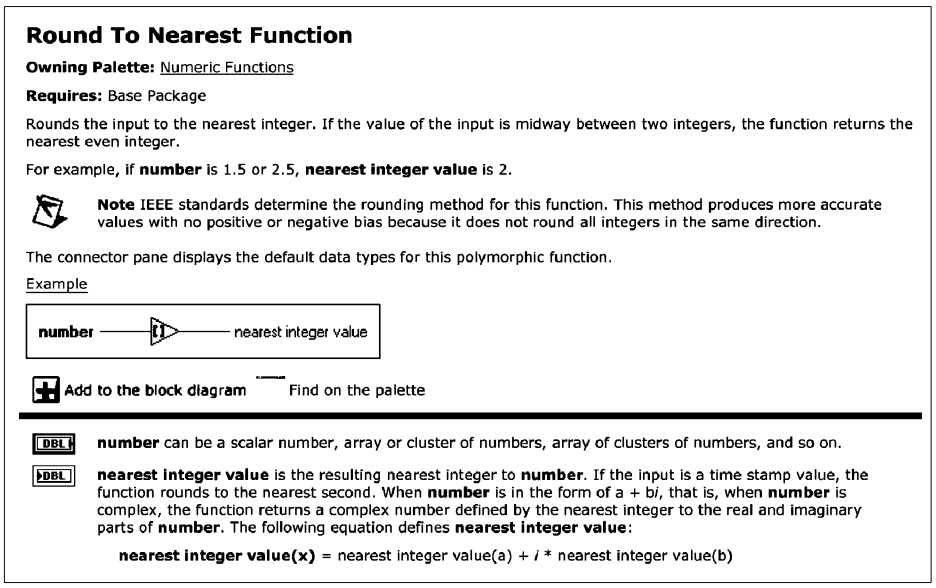


Figure 3.17: The icon of the `Round To Nearest Tenths Place` VI



Round To Nearest Function

Owning Palette: Numeric Functions

Requires: Base Package

Rounds the input to the nearest integer. If the value of the input is midway between two integers, the function returns the nearest even integer.

For example, if **number** is 1.5 or 2.5, **nearest integer value** is 2.

Note IEEE standards determine the rounding method for this function. This method produces more accurate values with no positive or negative bias because it does not round all integers in the same direction.

The connector pane displays the default data types for this polymorphic function.

Example

number — nearest integer value

Add to the block diagram Find on the palette

number can be a scalar number, array or cluster of numbers, array of clusters of numbers, and so on.

nearest integer value is the resulting nearest integer to **number**. If the input is a time stamp value, the function rounds to the nearest second. When **number** is in the form of a + b*i*, that is, when **number** is complex, the function returns a complex number defined by the nearest integer to the real and imaginary parts of **number**. The following equation defines **nearest integer value**:

nearest integer value(x) = nearest integer value(a) + *i* * nearest integer value(b)

Figure 3.18: Help file for the **`Round To Nearest`** function ■

Exercise 1.23: Create the `Pre-Purge Timer Done` Sub-VI for the Boiler Controller

Create the `Pre-Purge Timer Done` sub-VI where the icon is shown in Figure 3.19. This sub-VI accepts a LIMITS cluster as an input as well as an ELAPSED TIME numeric control. This sub-VI should have a Boolean indicator output called STOP. Round the elapsed time to the nearest tenths place using the `Round To Nearest Tenths Place` sub-VI you developed in Exercise 1.22. If the rounded elapsed time is greater than or equal to the value of the PRE-PURGE TIME (S) control of the LIMITS cluster developed in Exercise 1.17, the value of DONE should be True: otherwise, it should be False. You might find the **`Greater or Equal?`** function useful (see the help file in

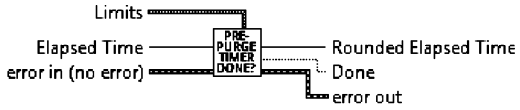


Figure 3.19: The icon of the `Pre-Purge Timer Done` VI

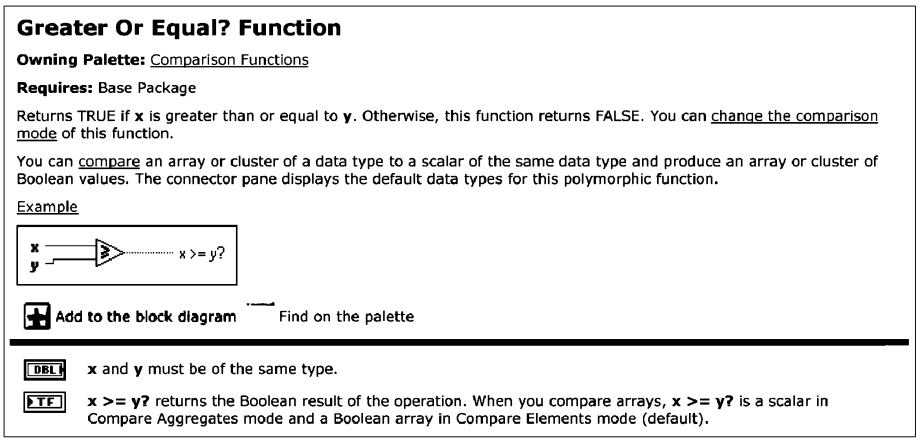


Figure 3.20: Help file for the `Greater or Equal?` function

Figure 3.20). Also, pass the value of the rounded elapsed time to a ROUNDED ELAPSED TIME output indicator. Use the sub-VI with the error-handling design pattern. ■

Exercise 1.24: Create the `Pre-Purge Timer Output` Sub-VI for the Boiler Controller

Create the `Pre-Purge Timer Output` sub-VI with the icon shown in Figure 3.21. This sub-VI accepts a BOILER STATUS cluster, an ELAPSED TIME numeric control, and a DONE? Boolean control as inputs. This sub-VI should assign the string `Pre-Purge` to the STATUS string of the BOILER STATUS cluster, the Boolean value True to the PRIMARY FAN Boolean of the BOILER STATUS cluster, and the value of the ELAPSED TIME input numeric control to the TIME COUNT (S) numeric control of the BOILER STATUS cluster. The new value of the BOILER STATUS cluster should be passed as an output of the sub-VI.

This sub-VI should also output a log string. The string should be of the format `Text, Number`. If the input DONE? Boolean is True, the value of `Text` should be `Pre-Purge Complete`. If the value is False, the value of `Text` should be `Start Pre-Purge`. In either case, the value of `Number` should be the value of the input ELAPSED TIME to the nearest tenth of a second. You might find the **`Format Into String`** function useful, (see the help file in Figure 1.60). Use the sub-VI with the error-handling design pattern.

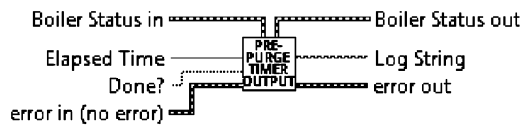


Figure 3.21: The icon of the `Pre-Purge Timer Output` VI ■

Exercise 1.25: Create the `Pre-Purge Done` Sub-VI for the Boiler Controller

Create the `Pre-Purge Done` sub-VI with the icon shown in Figure 3.22. This sub-VI accepts a BOILER CONTROLS cluster as an input. This sub-VI should assign the value of the PILOT BUTTON control of the BOILER CONTROLS cluster to a READY Boolean indicator output. Use the sub-VI with the error-handling design pattern.

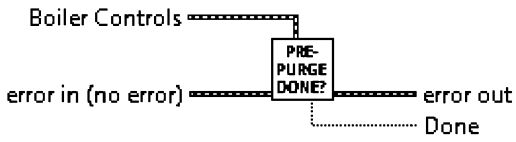


Figure 3.22: The icon of the `Pre-Purge Done` VI ■

Exercise 1.26: Create the `Pre-Purge Output` Sub-VI for the Boiler Controller

Create the `Pre-Purge Output` sub-VI with the icon shown in Figure 3.23. This sub-VI accepts a BOILER STATUS cluster as an input. This sub-VI should assign the string `Pre-Purge Complete` to the STATUS string of the BOILER STATUS cluster. Use the sub-VI with the error-handling design pattern.

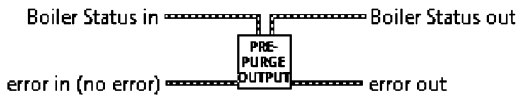


Figure 3.23: The icon of the `Pre-Purge Output` VI ■

Exercise 1.27: Create the `Ignition Done` Sub-VI for the Boiler Controller

Create the `Ignition Done` sub-VI with the icon shown in Figure 3.24. This sub-VI accepts a BOILER SIMULATION CONTROLS cluster and a LIMITS cluster as inputs. This sub-VI should check to see if the value of the FLAME SENSOR VALUE (%) control is greater than the value of the FLAME SENSOR LOW control found in the LIMITS cluster developed in Exercise 1.17. If so, the output READY Boolean should be True: otherwise, it should

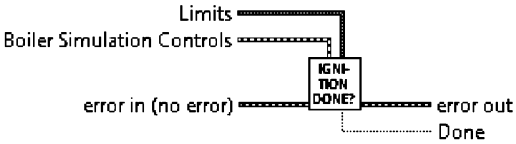


Figure 3.24: The icon of the `Ignition Done` VI

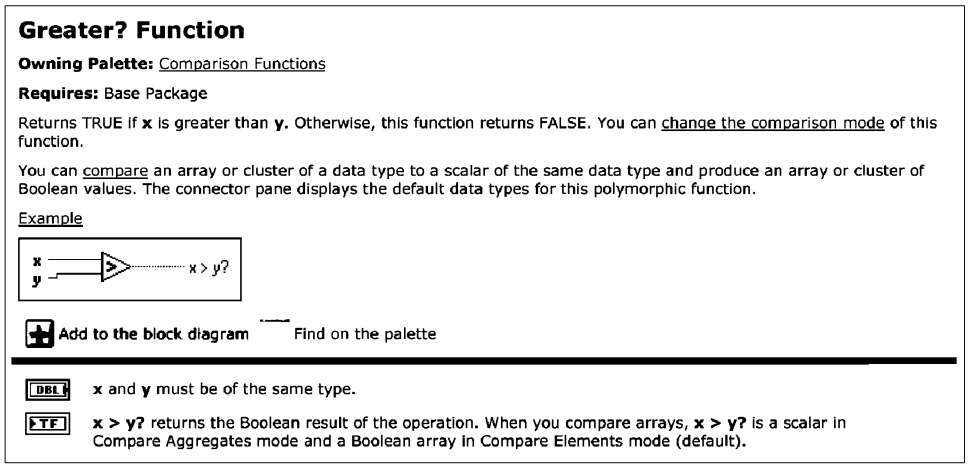


Figure 3.25: Help file for the `Greater?` function

be False. You might find the **`Greater?`** function useful (see help file in Figure 3.25). Use the sub-VI with the error-handling design pattern. ■

Exercise 1.28: Create the `Ignition Output` Sub-VI for the Boiler Controller

Create the `Ignition Output` sub-VI with the icon shown in Figure 3.26. This sub-VI accepts a BOILER STATUS cluster as an input. This sub-VI should assign the string `Pilot On` to the STATUS string of the BOILER STATUS cluster and a Boolean True to both the NATURAL GAS VALVE and PILOT INDICATOR Boolean indicators of the BOILER STATUS cluster. The revised BOILER STATUS cluster should be passed as an output. This sub-VI should also pass the log string `Pilot ON, True` as an output.

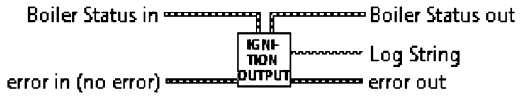


Figure 3.26: The icon of the `Ignition Output` VI ■

Exercise 1.29: Create the `Prove The Pilot Done` Sub-VI for the Boiler Controller

Create the `Prove The Pilot Done` sub-VI with the icon shown in Figure 3.27. This sub-VI accepts a BOILER CONTROLS cluster as an input. This sub-VI should assign the value of the FORCED DRAFT FAN control of the BOILER CONTROLS cluster to a READY Boolean indicator output. This sub-VI should also assign the value of the FLAME SENSOR VALUE (%) control to a FLAME SENSOR numeric output. Use the sub-VI with the error-handling design pattern.

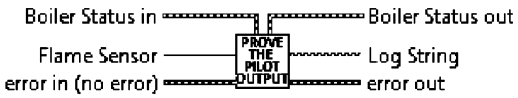


Figure 3.27: The icon of the `Prove The Pilot Done` VI ■

Exercise 1.30: Create the `Prove The Pilot Output` Sub-VI for the Boiler Controller

Create the `Prove The Pilot Output` sub-VI with the icon shown in Figure 3.28. This sub-VI accepts a BOILER STATUS cluster and a FLAME SENSOR numeric control as inputs. This sub-VI should assign the string `Boiler Ready` to the STATUS string of the BOILER STATUS cluster and a Boolean True to both the NATURAL GAS VALVE and PILOT INDICATOR Boolean indicators of the BOILER STATUS cluster. The revised BOILER STATUS cluster should be passed as an output. This sub-VI should also pass the log string `Pilot ON, True` as an output.

This sub-VI should also output a log string. The string should be of the format `Pilot Proved, Number`, where `Number` is the value of the FLAME SENSOR input to the nearest tenth of a percent. You might find the **`Format Into String`** function useful (see the help file in Figure 1.60). Use the sub-VI with the error-handling design pattern.

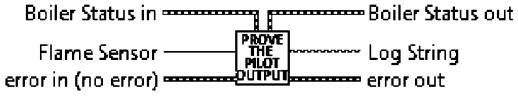


Figure 3.28: The icon of the `Prove The Pilot Output` VI ■

Exercise 1.31: Create the `Start Done` Sub-VI for the Boiler Controller

Create the `Start Done` sub-VI with the icon shown in Figure 3.29. This sub-VI accepts a LIMITS cluster and a BOILER CONTROLS cluster as inputs. This sub-VI should check to see if the FUEL CONTROL VALVE POSITION (% OPEN) control is within the limits of the FUEL CONTROL LOW and FUEL CONTROL HIGH controls found in the LIMIT cluster developed in Exercise 1.17. If it is within these limits, a Boolean READY output should be set to True: otherwise, it should be set to False. You might find the **`In Range and Coerce`** function helpful (see the help file in Figure 3.30). Use the sub-VI with the error-handling design pattern.

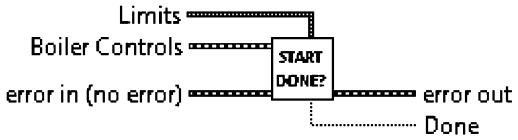


Figure 3.29: The icon of the `Start Done` VI

In Range and Coerce Function

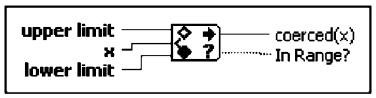
Owning Palette: Comparison Functions

Requires: Base Package

Determines whether **x** falls within a range specified by the **upper limit** and **lower limit** inputs and optionally coerces the value to fall within the range. The function performs the coercion only in Compare Elements mode. This function accepts time stamp values if all inputs are time stamp values. You can change the comparison mode of this function.

The connector pane displays the default data types for this polymorphic function.

Details Example



Add to the block diagram Find on the palette

upper limit, **x**, and **lower limit** should usually be of the same data type structure, either arrays or clusters, but they can have different numeric representations. For example, if you change one of the data types to an array, you must change the remaining data types to arrays to avoid broken wires. However, you can wire an array to **x** and two scalars to **upper limit** and **lower limit**. You also can wire double-precision, floating-point and integer data.

coerced(x) returns the coerced or unchanged value of **x**. If **x** is within the range set by the **upper limit** and **lower limit** inputs or if the function is in Compare Aggregates mode, the value is unchanged. If **x** is not in range and the function is in Compare Elements mode, the function converts the value to the same value as **upper limit** or **lower limit**. If **upper limit**, **x**, or **lower limit** is NaN, **coerced(x)** is NaN.

In Range? is a Boolean value in Compare Aggregates mode. In Compare Elements mode, the data type structure of **In Range?** matches the data type structure of **x**, with each scalar replaced by a Boolean value. If **upper limit**, **x**, or **lower limit** is NaN, **In Range?** is FALSE.

In Range and Coerce Details

This function compares the input data values according to the Boolean comparison rules. If the function is in Compare Aggregates mode, it returns the unchanged value of **x** in **coerced(x)** rather than a coerced value. The function considers each input array as a single aggregate object, similar to a cluster, where the first element is primary in the comparison. If **x** is greater than **upper limit** and the function is in Compare Elements mode (default), the function coerces **x** to the **upper limit** value. If **x** is less than **lower limit** and the function is in Compare Elements mode, the function coerces **x** to the lower limit value.

In the case of numeric values, the **upper limit**, **lower limit**, and **x** inputs do not have to be the same data type representation. If you wire a combination of signed and unsigned integers to the **upper limit**, **x**, and **lower limit** inputs of the In Range and Coerce function, the data type of the **coerced(x)** output must be large enough to represent the full range of possible input values. If the **coerced(x)** output is an unsigned integer, the signed input values must be positive because an unsigned integer cannot represent a negative number.

By default, In Range And Coerce includes the **lower limit** value in the range, as indicated by the solid black diamond at the **lower limit** input. By default, the **upper limit** value is not included in the range, as indicated by the empty diamond at the **upper limit** input.

To change whether the function includes the value wired to **upper limit** or **lower limit**, right-click the function and select **Include upper limit** or **Include lower limit** from the shortcut menu to include or remove the value in the range. These options only affect the result of **In Range?**. The function always coerces **x** to the exact values of **upper limit** or **lower limit**, regardless of whether you select **Include upper limit** or **Include lower limit** from the shortcut menu.

If the **lower limit** value is greater than the **upper limit** value, LabVIEW switches the values internally without displaying an error message. If the function is in Compare Elements mode, it still coerces **x** to either the **upper limit** or **lower limit** value. However, **In Range?** is always FALSE, even if **x** is between the lower and upper limits.

Figure 3.30: Help file for the `In Range and Coerce` function ■

Exercise 1.32: Create the `Start Output` Sub-VI for the Boiler Controller

Create the `Start Output` sub-VI with the icon shown in Figure 3.31. This sub-VI accepts a BOILER STATUS cluster as an input. This sub-VI should assign the string `Boiler Ready` to the STATUS string of the BOILER STATUS cluster and a Boolean True to the NATURAL GAS VALVE, PILOT INDICATOR, and FORCED DRAFT FAN Boolean indicators of the BOILER STATUS cluster. The revised BOILER STATUS cluster should be passed as an output. This sub-VI should also pass the log string `Forced Draft Fan ON, True` as an output. Use the sub-VI with the error-handling design pattern.

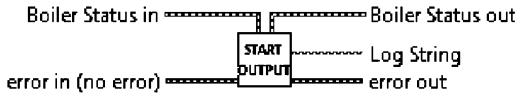


Figure 3.31: The icon of the `Start Output` VI ■

Exercise 1.33: Create the `Run Done` Sub-VI for the Boiler Controller

Create the `Run Done` sub-VI with the icon shown in Figure 3.32. This sub-VI accepts the LIMITS cluster developed in Exercise 1.17, a BOILER CONTROLS cluster, and a BOILER SIMULATION CONTROLS cluster as inputs. This sub-VI should output a STOP Boolean indicator. This indicator is true if any of the following conditions are true:

- The FUEL CONTROL VALVE POSITION (% OPEN) control is outside the limits of the FUEL CONTROL LOW and FUEL CONTROL HIGH controls in the LIMIT cluster.
- The SHUTDOWN button has been pressed (value is True).
- The RUN INTERLOCK button has been released (value is False).
- The FORCED DRAFT FAN button has been released (value is False).

If none of these conditions are met, the STOP Boolean indicator should be False. You might find the **`Compound Arithmetic`** function useful (see the help file in Figure 3.33). Use the sub-VI with the error-handling design pattern.

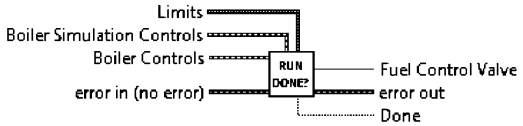


Figure 3.32: The icon of the `Run Done` VI

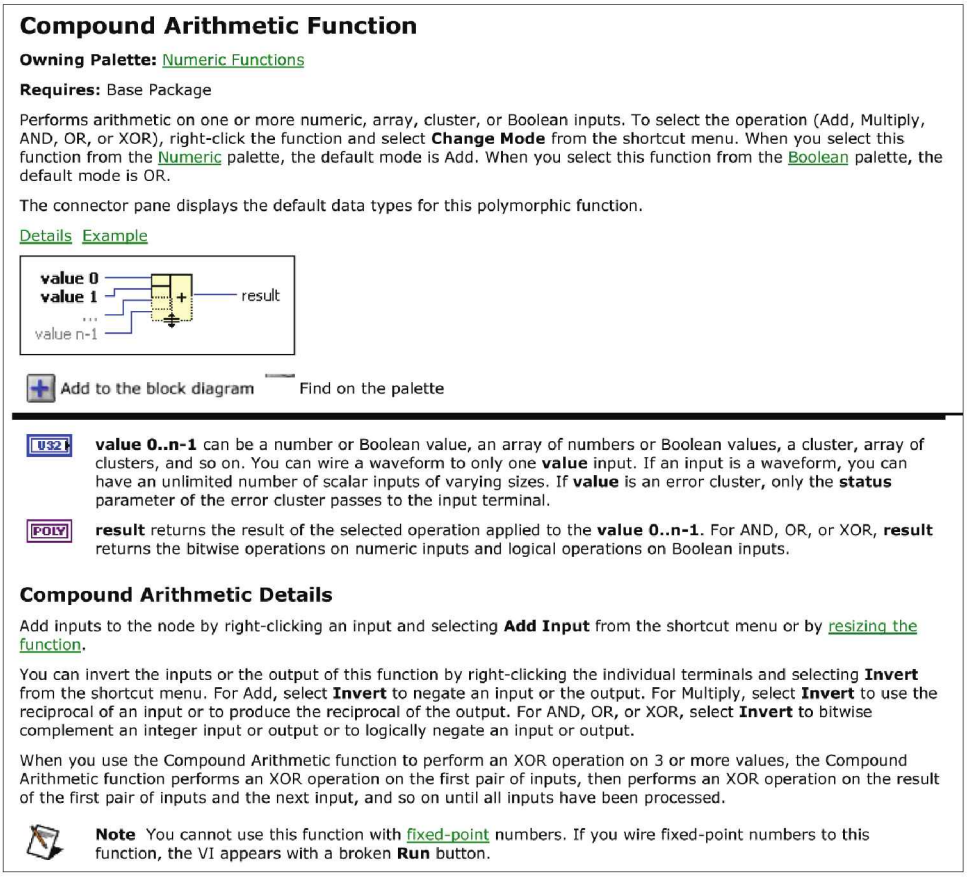


Figure 3.33: Help file for the `Compound Arithmetic` function ■

Exercise 1.34: Create the `Run Output` Sub-VI for the Boiler Controller

Create the `Run Output` sub-VI with the icon shown in Figure 3.34. This sub-VI accepts a BOILER STATUS cluster as an input. This sub-VI should assign the string `Boiler Running` to the STATUS string of the BOILER STATUS cluster and a Boolean True to the FUEL VALVE Boolean indicator of the BOILER STATUS cluster. The revised BOILER STATUS cluster should be passed as an output.

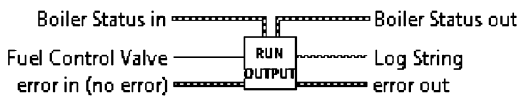


Figure 3.34: The icon of the `Run Output` VI

This sub-VI should also output a log string. The string should be of the format `Boiler Running, Number`, where `Number` is the value of the FUEL CONTROL VALVE input to the nearest tenth of a percent. You might find the **`Format Into String`** function useful (see the help file in Figure 1.60). Use the sub-VI with the error-handling design pattern. ■

Exercise 1.35: Create the `Purge Done` Sub-VI for the Boiler Controller

Create the `Purge Done` sub-VI with the icon shown in Figure 3.35. This sub-VI accepts the LIMITS cluster developed in Exercise 1.17 and an ELAPSED TIME numeric control as inputs. This sub-VI should round the elapsed time to the nearest tenths place, using the `Round To Nearest Tenths Place` sub-VI you developed in Exercise 1.22. Check to see if the rounded elapsed time is greater than or equal to the value of the PURGE TIME (S) control of the LIMITS cluster. The result of this comparison should be assigned to a STOP Boolean indicator. You might find the **`Greater Or Equal?`** function useful (see the help file in Figure 3.25). The rounded elapsed time should be passed as an output. Use the sub-VI with the error-handling design pattern.

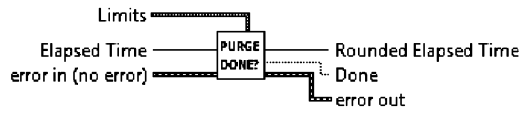


Figure 3.35: The icon of the `Purge Done` VI ■

Exercise 1.36: Create the `Purge Output` Sub-VI for the Boiler Controller

Create the `Purge Output` sub-VI with the icon shown in Figure 3.36. This sub-VI accepts a BOILER STATUS cluster, an ELAPSED TIME numeric control, and a DONE Boolean control as inputs. This sub-VI should assign the string `Purge` to the STATUS string of the BOILER STATUS cluster, a Boolean True to the PRIMARY FAN Boolean indicator of the BOILER STATUS cluster, and the value of the ELAPSED TIME numeric control to the TIME COUNT (S) control of the BOILER STATUS cluster. The revised BOILER STATUS cluster should be passed as an output.

This sub-VI should also output a log string. The string should be of the format `Text, Number`. If the input DONE? Boolean is True, the value of `Text` should be

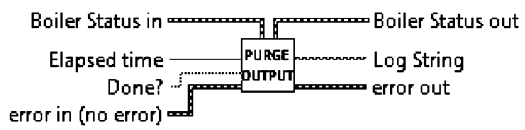


Figure 3.36: The icon of the `Purge Output` VI

`Shutdown Purge Complete`. If the value is False, the value of `Text` should be `Start Shutdown Purge`. In either case, the value of `Number` should be the value of the input ELAPSED TIME to the nearest tenth of a second. You might find the **`Format Into String`** function useful (see the help file in Figure 1.60). Use the sub-VI with the error-handling design pattern. ■

3.5 Car Wash 2 Controller Exercises

Exercise 1.37: Car Wash 2 States

Read the functional specification for the Car Wash 2 controller in Appendix D. What states will the program need? Create a type-defined enumerated control containing these states. ■

Exercise 1.38: Car Wash 2 State Transition Diagram

Draw a state transition diagram for the Car Wash 2 controller using the states you listed in the previous exercise. Remember that any state can transition to the *Shutdown* state if the STOP button is pressed. Use the state transition diagrams in Chapter 2 as a guide. Compare and contrast the Car Wash 2 controller with the original car wash controller. ■

Exercise 1.39: Create the `Car In Position` Sub-VI

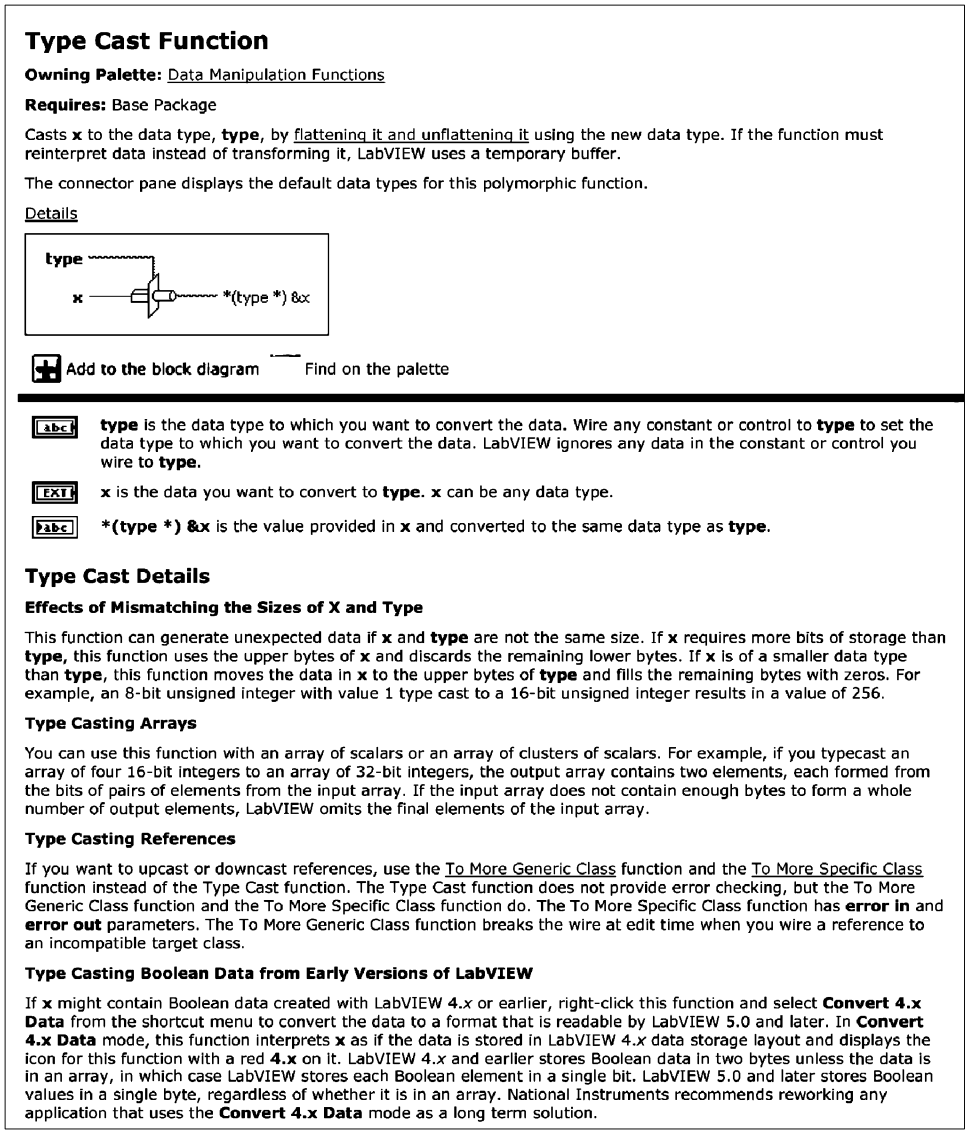
Create type-defined controls for the ENTRY CONSOLE, the CAR WASH DISPLAY, and the CAR POSITION CONTROL. Use captions for the front panel display and shorter labels for the block diagram. The WASH OPTIONS and the WASH INDICATORS also should be saved as type definitions.

The `Car Wash Display` is a slider, and it presents readability issues. The front panel appearance of the slider is very informative with text labels indicating to the user which slider position corresponds to which station. The block diagram appearance of the slider is not as informative, since a block diagram constant of the type-defined slider control is merely an integer.

The Car Wash 2 controller will need to check the slider to see if a vehicle is currently in position for a particular wash option. Improve the readability of this operation by creating a type-defined enumerated constant containing the five text labels of the slider: *Entry*, *Station 1*, *Station 2*, *Station 3*, and *Exit*. Then create the `Car In Position` sub-VI with the icon shown in Figure 3.37. This sub-VI takes the



Figure 3.37: The icon of the `Car In Position` VI



Type Cast Function

Owning Palette: Data Manipulation Functions

Requires: Base Package

Casts **x** to the data type, **type**, by flattening it and unflattening it using the new data type. If the function must reinterpret data instead of transforming it, LabVIEW uses a temporary buffer.

The connector pane displays the default data types for this polymorphic function.

Details

Add to the block diagram Find on the palette

type is the data type to which you want to convert the data. Wire any constant or control to **type** to set the data type to which you want to convert the data. LabVIEW ignores any data in the constant or control you wire to **type.**

x is the data you want to convert to **type**. **x** can be any data type.

***(type *) &x** is the value provided in **x** and converted to the same data type as **type**.

Type Cast Details

Effects of Mismatching the Sizes of X and Type

This function can generate unexpected data if **x** and **type** are not the same size. If **x** requires more bits of storage than **type**, this function uses the upper bytes of **x** and discards the remaining lower bytes. If **x** is of a smaller data type than **type**, this function moves the data in **x** to the upper bytes of **type** and fills the remaining bytes with zeros. For example, an 8-bit unsigned integer with value 1 type cast to a 16-bit unsigned integer results in a value of 256.

Type Casting Arrays

You can use this function with an array of scalars or an array of clusters of scalars. For example, if you typecast an array of four 16-bit integers to an array of 32-bit integers, the output array contains two elements, each formed from the bits of pairs of elements from the input array. If the input array does not contain enough bytes to form a whole number of output elements, LabVIEW omits the final elements of the input array.

Type Casting References

If you want to upcast or downcast references, use the To More Generic Class function and the To More Specific Class function instead of the Type Cast function. The Type Cast function does not provide error checking, but the To More Generic Class function and the To More Specific Class function do. The To More Specific Class function has **error in** and **error out** parameters. The To More Generic Class function breaks the wire at edit time when you wire a reference to an incompatible target class.

Type Casting Boolean Data from Early Versions of LabVIEW

If **x** might contain Boolean data created with LabVIEW 4.*x* or earlier, right-click this function and select **Convert 4.x Data** from the shortcut menu to convert the data to a format that is readable by LabVIEW 5.0 and later. In **Convert 4.x Data** mode, this function interprets **x** as if the data is stored in LabVIEW 4.*x* data storage layout and displays the icon for this function with a red **4.x** on it. LabVIEW 4.*x* and earlier stores Boolean data in two bytes unless the data is in an array, in which case LabVIEW stores each Boolean element in a single bit. LabVIEW 5.0 and later stores Boolean values in a single byte, regardless of whether it is in an array. National Instruments recommends reworking any application that uses the **Convert 4.x Data** mode as a long term solution.

Figure 3.38: Help file for the `Type Cast` function

slider as an input and compares it to a selected position indicated by the enumerated constant created in this exercise. The enumerated constant will show its text on the block diagram, improving readability. You might find the **`Type Cast`** function useful (see the help file in Figure 3.38). ■

Exercise 1.40: Create the `Build Option Array` Sub-VI

One of the challenges of the Car Wash 2 controller is that the sequence of states for a car wash can vary from one wash to another based on user input. The sequence of states is determined by the options selected by the user. This challenge will be addressed by the `Build Option Array` sub-VI with the icon shown in Figure 3.39. This sub-VI takes the WASH OPTIONS cluster as an input and produces an array of car wash states corresponding to the option selections as an output. If the *High Pressure Wash* option has not been selected by the user, it is added to the final array of states in its proper position.

Convert the cluster of wash options to a Boolean array, and use this array with a **`For`** loop conditional output tunnel to find the selected options. You might find the **`Cluster To Array`** function useful (see help file in Figure 3.6).

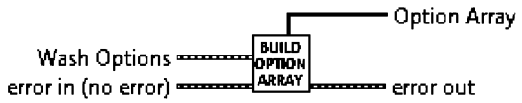


Figure 3.39: The icon of the `Build Option Array` VI ■

Exercise 1.41: Create the `Next Car Wash State` Sub-VI

The array of options generated by the `Build Option Array` developed in Exercise 1.40 will be used to determine the state transitions in the Car Wash 2 controller. This task of determining the next state will fall to the `Next Car Wash State` sub-VI with the icon shown in Figure 3.40. Create this sub-VI. This sub-VI accepts the array of options, the index of the current state in the array, and a DONE? Boolean as inputs. It also outputs the next index and the next state.

If the current state is not done, the current state and index should be passed through as outputs. If the current state is done, increment the index and select the next option state from the options array. If the current state is done and it is the last state in the option array, return the *Car Exit* state and the incremented index as outputs. You might find the **`Index Array`** function useful whose help file is shown in Figure 3.41.

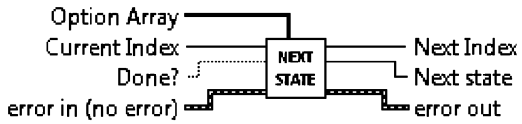


Figure 3.40: The icon of the `Next Car Wash State` VI

Index Array Function

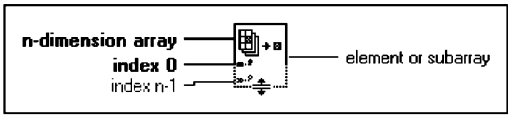
Owning Palette: Array Functions

Requires: Base Package

Returns the **element or subarray** of **n-dimension array** at **index**.

When you wire an array to this function, the function resizes automatically to display **index** inputs for each dimension in the array you wire to **n-dimension array**. You also can add additional **element or subarray** terminals by resizing the function. The connector pane displays the default data types for this polymorphic function.

Details



Add to the block diagram Find on the palette

n-dimension array can be an n-dimensional array of any type. If **n-dimension array** is an empty array, **element or subarray** returns the default value of the defined data type for the array.

index 0..n-1 must be numeric. The number of **index** inputs matches the number of dimensions in **n-dimension array**.

If the **index** is out of range (<0 or ≥N, where N is the size of **n-dimension array**), **element or subarray** returns the default value of the defined data type for the array.

Array functions in LabVIEW access array data in row-major order. In a 2D array, row is the first, major index. Column is the last, minor index. In bigger multidimensional arrays, column remains the last index and the more major indexes are added to the front. Thus, vol is the first index, followed by page, row, and column. These names are index identifiers and have no other meaning.

You can extract a subarray of the array by leaving one or more of the index terminals unwired. For example, extract column 1 of a 2D array by specifying `1` in the column index and leaving the row index unwired. If you index a 1D array and do not wire anything to the **index** input terminal, the Index Array function extracts the first element of the array. If you expand the node to show more than one **element or subarray** output, the function extracts the first number of elements equal to the number of **element or subarray** outputs. If you wire indexes only for some **element or subarray** outputs, the function extracts the index after the last wired index for the elements without wired indexes, and continues in order until the next **element or subarray** output with a wired index.

For example, if you wire a 2D array to the Index Array function, expand the node to show three outputs, wire `3` to the row index for the first output, and leave all other indexes unwired, Index Array outputs subarray rows `3`, `4`, and `5`. If you wire `3` to the column index, Index Array outputs subarray columns `3`, `4`, and `5`. If you wire `3` to both indexes, Index Array outputs the array elements at `[3,3]`, `[3,4]`, and `[3,5]`.

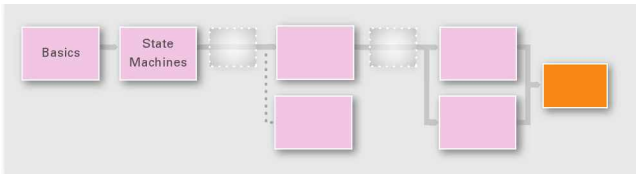
element or subarray has the same type as the elements of **n-dimension array**.

Index Array Details

You can disable indexing along a dimension by leaving the corresponding index input unwired, unless you are indexing a 1D array. By default, the first dimension has indexing enabled, with the others disabled. If disabled, the input terminal is an outlined black rectangle. If enabled, it is filled. You can wire a constant or control to the index inputs that you want enabled.

For example, if you want to index a row in a 2D array, the first **index** input is enabled and the second **index** input is disabled. If you want to index that same 2D array also by column, you can resize the function to show another set of input terminals. This next set of inputs has its own corresponding **subarray** output. By default, if you do not wire any **index** input terminals, the first **subarray** indexes row 0, the second **subarray** indexes row 1, and so on.

Figure 3.41: Help file for the `Index Array` function ■



4

Classic State Machines

This chapter introduces a robust design pattern called the state machine that can be used to tackle many common LabVIEW coding problems. The state machine translates state transition diagrams into LabVIEW code that is readable, scalable, and maintainable. State machines are robust, flexible, and compact. They provide storage for persistent data and handle errors gracefully and efficiently.

4.1 State Machine Elements

The state machine architecture organizes a VI into states and transitions. The states are the general functions and abilities of the VI—the things the VI can do. Transitions map out the ways that the VI can move from one state to another.

The states are represented by individual cases in a case structure. Each case contains all of the code needed to execute the functions of its state. The case structure is placed within a loop, driving the VI from one state to another. Each state decides which state to move to next. Typically, the states are listed in a type-defined enumerated constant, and this constant is stored in a shift register on the main loop. Each loop iteration, which is the value of the enumerated constant, is read from the shift register and fed to the selector terminal of the case structure to choose the next state. When a state is completed, it places the name of the next state on the shift register. Each state is responsible for telling the state machine where to go to next.

Each state must also decide if the main loop continues or terminates. This is typically done by creating a Boolean output tunnel on the main case structure and wiring it to the continuation terminal of the loop. Each state in the case structure wires a value to this tunnel to either stop the state machine or keep it running.

Let's take a look at the standard elements of a classic state machine in more detail. A basic schematic of the classic state machine is shown in Figure 4.1. In order to discuss the various elements of the state machine, we have to name them. Unfortunately, there are very few widely accepted terms in the LabVIEW community used to discuss state machine elements. This means that we have the freedom to invent our own names for use in this book. The terms "lobby" and "alley" were coined for this book. The term

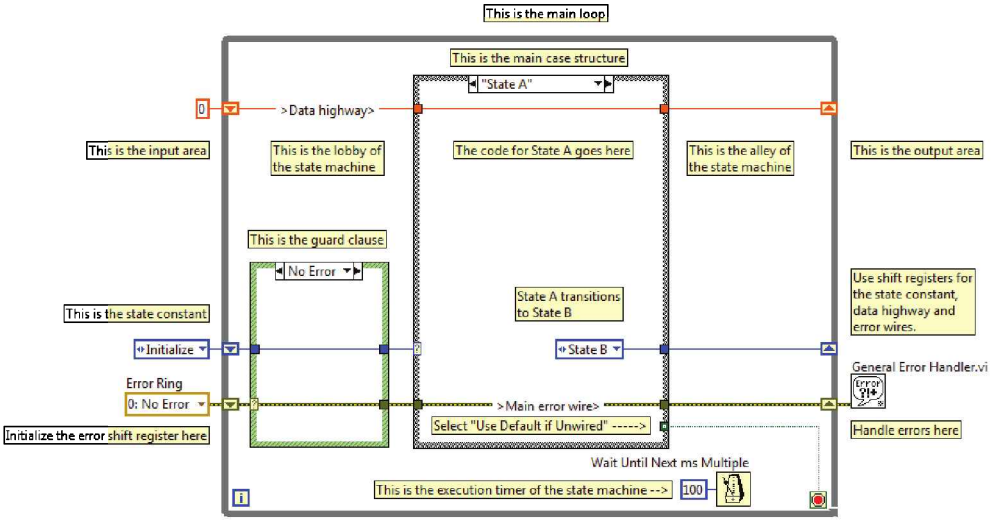


Figure 4.1: State machine terminology

"data highway" is a commonly accepted term in the LabVIEW community. The term "guard clause" is well-established in text-based programming, but it has not been widely used in the LabVIEW community.

Here is a glossary defining the various parts of a classic state machine:

Alley This is the portion of the state machine between the right edge of the case structure and the right edge of the main loop. The alley is often used for indicators and for logic connected to the stop terminal of the main loop.

Data Highway This is the group of wires that run from one end of the main loop to the other. The data highway connects pairs of shift registers and runs through every case in the main case structure. The data highway provides persistent storage in the state machine. This is a commonly used term in the LabVIEW community.

Error Shift Register This shift register is usually found near the bottom of the main loop. It passes warnings and errors from one iteration of the loop to the next.

Execution Timer This provides timing to regulate the loop iteration rate. It does not provide overall timing for the main application.

Guard Clause The guard clause sends the state machine into shutdown if an error is detected or if other user-defined conditions occur, such as hitting a stop button. If there is no reason to shut down the state machine, the next state is passed to the selector of the the main case structure.

Input Area This area contains inputs to the main loop of the state machine. It is commonly used for initializing shift registers, reinitializing front panel controls, and queue creation.

Lobby This is the portion of the state machine between the left edge of the main loop and the left edge of the case structure. The lobby is commonly used for error

checking, for polling a stop button, as a location for inputs that need to be monitored every loop iteration, or for inputs needed by multiple states.

Main Case Structure The case structure is the heart of the state machine. The selector of the case structure accepts an input that designates the next state, typically an enumerated constant. It contains the code for each state of the state machine. Each state is represented by a case in the case structure.

Main Error Wire The main error wire is the error-handling backbone of the state machine. This wire runs from one end of the main loop to the other. The error wire connects a pair of shift registers, and it runs through both the guard clause and the main case structure. The main error wire should be sent to an error-handling routine after the state machine finishes.

Main Loop This is the `While` loop that drives the state machine. The `While` loop contains shift registers for the data highway, the state constant, and the main error wire. The conditional terminal is wired to an output of the main case structure. Each case must send either a True or False value to the conditional terminal.

Output Area This area contains the outputs of the main loop of the state machine. It is commonly used for closing and destroying references and for error handling.

State Constant This is the constant that determines the next state to be executed. It is typically a type definition enumerated constant or a string constant. The state constant is placed on a shift register to pass it on to the next iteration of the main loop.

State Shift Register This shift register stores the value of the state constant. It allows each state to determine which state will be executed next.

4.2 State Machine Structure

The area to the left of the main loop is the input area of the state machine. Code in the input section that is wired to the main loop executes before the loop starts running. Wires enter the loop through tunnels or shift registers. If a wire enters the loop at a non-indexing tunnel, the value on the wire at the beginning of loop is constant throughout the life of the loop. If a wire terminates at a shift register, the value on the wire initializes the shift register. The input area is a natural place to put code that initializes shift registers, sets front panel objects to known starting states, or creates queues (covered in Chapter 13). It is also a natural place to put items that are constant for the duration of the loop.

The main loop is the engine that drives the state machine from one state to the next. The main loop has shift registers devoted to storing important data known as the data highway. It also has a shift register dedicated to transmitting error information and the value of the next state, usually in the form of an enumerated constant that lists the states of the state machine.

In Figure 4.1, the state shift register is initialized with the *Initialize* state. This ensures that the *Initialize* state will be the first state executed. The data highway and error shift registers are also initialized. Depending on the nature of the information

on the data highway, it may or may not be necessary to initialize the data highway shift registers. The error shift register, though, always should be initialized in a top-level VI. Uninitialized shift registers retain their data from one run of the VI to the next as long as LabVIEW remains open on the host computer. This may be desirable in an action engine sub-VI, as we will see in Chapter 5, but it is usually undesirable for a top-level VI. If you do not initialize the error shift register when a top-level VI starts, an error from the last run may be read from the unitialized shift register and enter the VI during the first loop iteration. Initializing the error shift register prevents this from happening.

The most common way of initializing the error shift register is to use the error wire emanating from start-up code placed in the input area of the loop. This technique uses the dataflow of the error wire to ensure that the start-up code executes before the loop begins to run. Figure 4.1 has no start-up code with error wires, so it uses an **`Error Ring`** constant to transmit a clean error cluster to the shift register. The help file for the **`Error Ring`** constant is shown in Figure 4.2.

When the main loop iterates, the state and error information are passed to the guard clause. The guard clause checks for errors and shuts down the state machine

Defining a Custom Error Code from the Error Ring

Creating a custom error code is useful if you want to define a single error code or overwrite a single built-in error code. If you have multiple custom errors you want to use in the Error Ring, use the Error Code Editor dialog box. Complete the following steps to create a custom error from the Error Ring for a one-time use.

1. Add the Error Ring to the block diagram.

   Add Find
2. Click the drop-down arrow to display the Select Error dialog box.
3. In the **Error Code Range** drop-down listbox, select **Custom Error Code**.
4. Specify an error code in the **Error Code** field.
   - To define a custom error code, enter an error code from the following ranges of error codes specifically reserved for custom error messages: -8999 through -8000, 5000 through 9999, and 500,000 through 599,999.
   - To overwrite an existing error code message, enter the existing error code. Creating a custom error message for an already-defined error code is useful when you want to return the error code for a common error case but you want to provide specific information about this particular instance of the error. For example, you can use error code 7, "File not found." and replace that with a description about the specific file that was not found.
5. Enter a description in the **Error Description (Optional)** field. The description field accepts the same % codes as the Format Into String function. For example, for the custom error message "At time %T, the file %s was not found", the configured Error Ring adjusts to include inputs for a timestamp and a string, respectively, as displayed in the following figure.

   time
   file
   7: At time %T, the file %s was not found.
6. Use the radio buttons to configure the Error Ring to return either an error or a warning.
7. Remove the checkmark from the **Include Call Chain** checkbox if you do not want to include the call chain information in the error cluster. If you include the call chain, the error description displays the chain of callers from the current VI to the top-level VI, which is useful when determining where in the VI hierarchy the error occurred. **(Real-Time Module)** Exclude the call chain to reduce jitter in real-time applications.
8. Click **OK**.

After you configure the Error Ring, you can adjust the configuration without returning to the dialog box.

- Click the two icons on the Error Ring to toggle the error/warning setting and the include/exclude call chain setting, respectively.
- Right-click the Error Ring and select **Generate Error**, **Generate Warning**, **Include Call Chain**, or **Exclude Call Chain** from the shortcut menu to affect the relevant configuration.
- Right-click the Error Ring and select **Visible Items»Error Explanation Text** from the shortcut menu to show or hide the error description.

Figure 4.2: Help file for the `Error Ring` constant

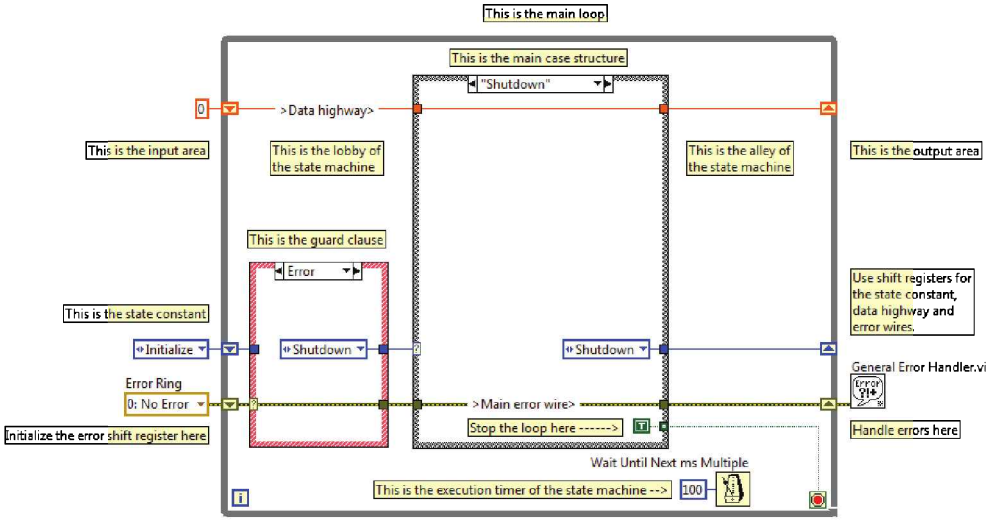


Figure 4.3: Illustration of how to use a *Shutdown* state to stop a state machine

if necessary. The guard clause in Figure 4.1 checks the status of the main error wire. If the error wire is clear, the guard clause passes the state information to the selector terminal of the main case structure. This can be seen in the *No Error* case of the guard clause shown in Figure 4.1.

If an error is detected by the guard clause, the *Shutdown* state is sent to the selector of the main loop. This is illustrated in the *Error* case of the guard clause shown in Figure 4.3. The guard clause is executed every loop iteration, since it is in the lobby of the state machine. This allows the the state machine to detect and respond to errors one loop iteration after the error occurs.

The guard clause is in the lobby of the state machine. The lobby contains code that executes every loop iteration before the main case structure executes. This makes it a good location for code that must be updated frequently to provide inputs for the main case structure. Be careful, though, code in the lobby executes before the main case structure only if dataflow demands it.

The main case structure is the heart of the state machine. Each state is represented by a case in the case structure. The code necessary to execute the state is placed in the case representing that state. Each case of the case structure also determines which state to execute next and places the appropriate value of the state constant on the state shift register.

Each state must also decide whether to continue or terminate the main loop. The *Shutdown* state, shown in Figure 4.3, wires a Boolean True to the conditional terminal of the main loop using an output tunnel in the main case structure. Every case of the case structure must wire a value to this output tunnel. One way to do this would be to wire a Boolean False constant to this tunnel in every case other than the *Shutdown* state. A simpler method is to enable the *Use Default If Unwired* option for the output tunnel, since the default value for a Boolean is False.

Wiring Tip: Be careful with the *Use Default If Unwired* option. The visual cue on the output tunnel is easy to miss and might lead to confusion, degrading readability. Stopping a loop is one of the acceptable uses of this option. When a reader flips through the cases of the case structure, the Boolean True in the *Shutdown* state jumps out because it is the only value explicitly wired to the tunnel. This increases the readability of the code and is a common enough use that the danger of confusion is minimal.

Code in the alley of the state machine executes when the code in the main case structure is finished. The code in the alley is executed every loop iteration, so it is a good location to put indicators that must be frequently updated from data generated by multiple states. Code in the alley of the state machine executes after the main case structure only if dataflow demands it.

With proper dataflow dependence, code in the lobby executes before the main case structure, and code in the alley executes after the main case structure. Code in the lobby often creates input tunnels to the main case structure, while code in the alley often flows from output tunnels from the main case structure. Input tunnels to a case structure can be used or ignored by the individual cases in the structure, but every case must output values to every output tunnel in a case structure.

Sometimes this can lead to difficulties. The *Shutdown* case shown in Figure 4.3 sends a Boolean True to the stop terminal of the of the main loop. Even though there will be no more state transitions, *something* must be wired to the enumerated constant output tunnel or the VI will not run. The *Shutdown* state was chosen because it is the least confusing choice in this situation.

State machines often require loop timing control. One choice for controlling the loop speed is the **`Wait Until ms Multiple`** function, which was discussed in Chapter 1. This function provides a variable waiting time, ensuring that the loop period is equal to the wired value in milliseconds (if possible). If the duration of the code within the loop exceeds the time target, the period will be set to the nearest multiple of the time target. This function also "puts LabVIEW to sleep" during the waiting period. The CPU is released by LabVIEW to perform other tasks during the wait period.

When the main loop has ended, the code in the output section of the state machine is executed. This is a natural place to put error-handling code. It is also a good place to close references, release resources, and do whatever else is needed to shut down the VI gracefully.

State machines often require storage for important data. This storage is provided by the data highway. The data highway is a wire (or group of wires) that connects pairs of shift registers at the top of the main loop. The shift registers ensure that data on the data highway is stored from one loop iteration to the next. The data highway runs through the main case structure, making it available to every state in the state machine.

4.3 Placing Elements in the State Machine

Once you have the main elements of the state machine in place, how do you decide where to place the controls, indicators, and constants needed by the state machine? Often the best location for a particular item will be obvious, but sometimes there may be more than one reasonable choice. Here are some guidelines to consider while making the decision of where to place controls, indicators, and constants on the block diagram.

Two questions should be considered when deciding where to place an item in a state machine. How many states need to access the item? Does the item need to be stored? The answers to these two questions will help. Some general guidelines are shown in Figure 4.4. The input area of the loop executes before the main loop starts, so it is a good place to put start-up code and initialization data.

The lobby and alley of the state machine are executed every iteration of the main loop and can be accessed by every state in the state machine. The lobby is a good place for controls that change frequently and are needed by multiple states. The alley is a good place for indicators that need to be updated by multiple states. If a control or indicator is only used by a single state, place it within that state.

If data is needed by multiple states and must be stored, consider placing it on the data highway. The data highway runs through the case structure and can be accessed by all states. The shift registers of the data highway provide storage for the data from one loop iteration to the next.

The output area of the state machine only executes after the state machine finishes. This makes it an ideal place for clean-up code, error handling, or controls and indicators that deal with data output by the state machine.

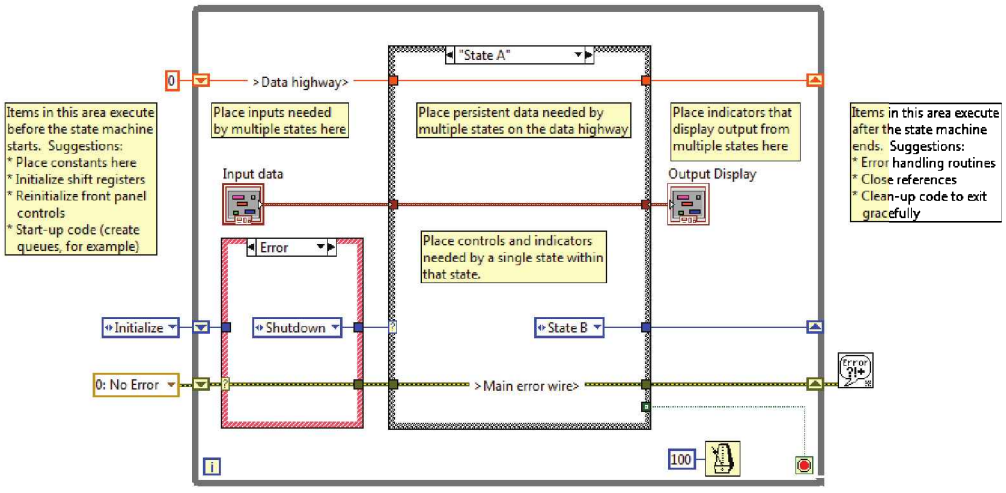


Figure 4.4: Rules of thumb for placement of controls, indicators, constants, and data in a state machine

4.4 Building a State Machine From Scratch: The `Multitest` VI

Now it's time to build our first complete state machine. Take a moment to review the specification for the `Multitest` VI given in Chapter 2 and the state transition diagram shown in Figure 2.2. This diagram will be the blueprint for the `Multitest` state machine.

Defining the States and the State Enumerated Constant

The first step in building a state machine is to identify the states, all of the possible transitions, and the conditions or triggers that cause the transitions. The states can be read directly from the state transition diagram in Figure 2.2: *Initialize*, *Wait*, *Run Test 1*, *Run Test 2*, and *Shutdown*. Using the methods described in Chapter 1, create a type-defined enumerated constant listing the states. It should look similar to the one shown in Figure 4.5. Note that the pull-down menu below the menu bar shows *Type Def.*, indicating that the control is being saved as a type definition.

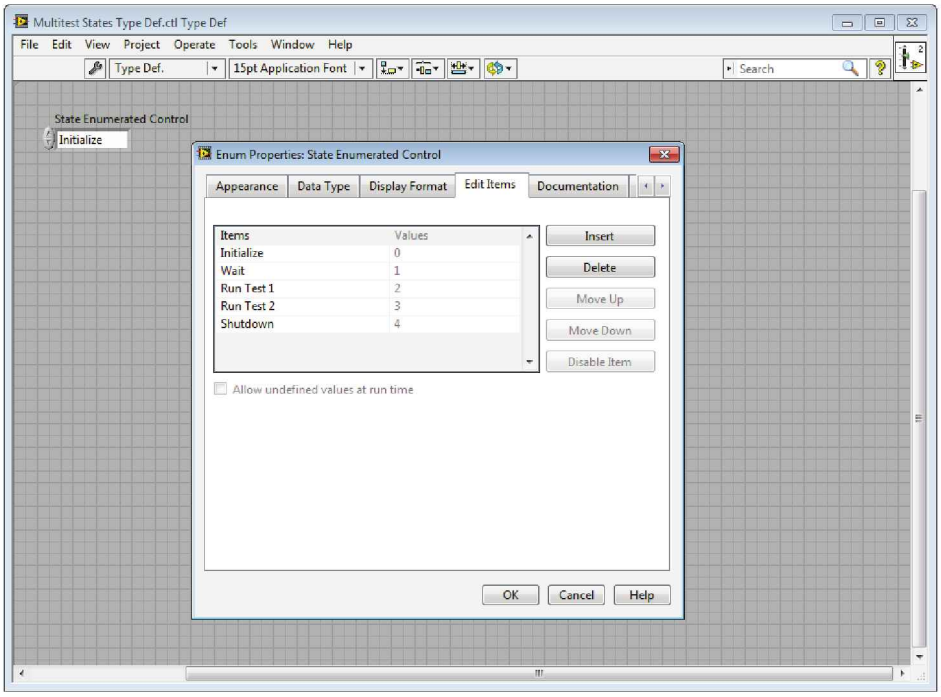


Figure 4.5: The type definition of the `Multitest` state enumerated control

Wiring Tip: Always save enumerated state controls as type definitions. Enumerated state constants will be used many times throughout a given state machine, and by saving them as type definitions, you help protect your code from changes during development. If states need to be added or deleted, you will only need to change the type definition file, and all of the state constants in the program will be automatically updated.

Build the Front Panel

Build the front panel of the VI as shown in Figure 2.1. In the programs in this book, this step is trivial since screenshots of the front panels are provided in the functional specifications. In real-life projects, this step may require significant design work to build a satisfactory and professional user interface.

Lay Out the Skeleton of the State Machine

Once the front panel has been built, lay out the skeleton of the state machine. Place the main loop, the main case structure, and the guard clause on the block diagram. Make the main loop large enough to fill most of the screen, but leave enough room at the left and right edges for initialization code and clean-up code. The goal is for the entire state machine to fit on the screen without scrolling.

Style Guide: Always strive to have the state machine fit entirely on one computer screen. This makes the code much more readable. If that is not possible, arrange the code so that the user only needs to scroll in one direction: left/right or up/down.

Place the **`Wait Until Next ms Multiple`** function in the lower portion of the main loop to control the execution timing. Right-click on the *millisecond multiple* input and select *Create... Constant*. Set the constant to `100`. At this point, the block diagram should look something like Figure 4.6.

Create and Initialize the Shift Registers

The `Multitest` VI will not need a data highway, so no shift registers are required at the top of the loop. There will be a shift register for the state constant located in the middle of the loop, and a shift register for the error cluster located near the bottom of the loop.

Drop a copy of the enumerated state control on the block diagram to the left of the loop. When controls are dropped directly onto the block diagram instead of the front panel, they are automatically converted to constants. Wire the state constant to the left edge of the main loop, creating an input tunnel. Right-click the tunnel, and select *Replace with Shift Register* to convert it to a shift register. Set the state constant to the *Initialize* state.

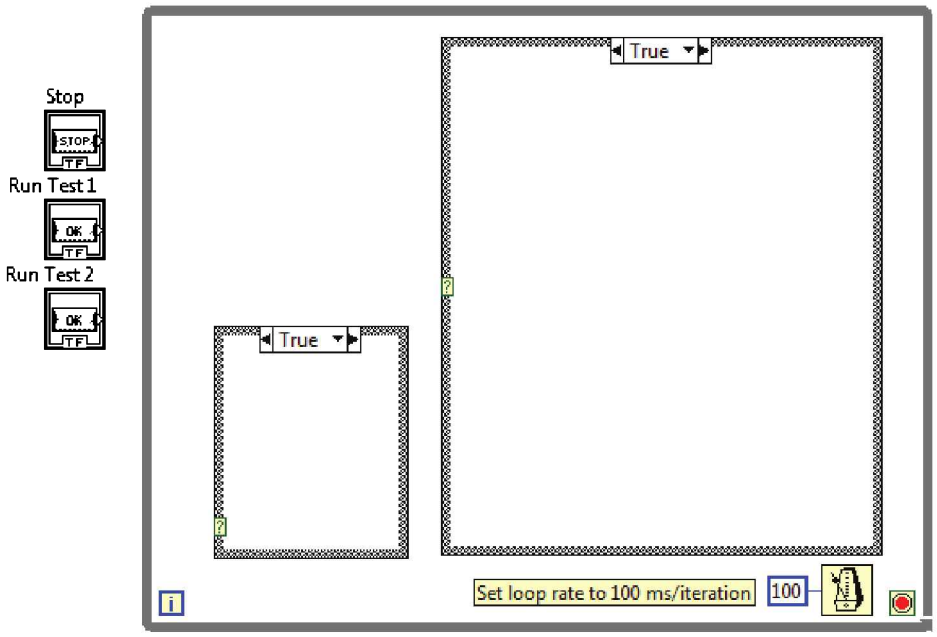


Figure 4.6: Laying out the skeleton of the `Multitest` VI

To create the error shift register, create a *Reinitialize To Default* invoke node for the STOP button as described in Chapter 1. Move the invoke node to the left of the main loop and wire it to the the left edge of the loop, creating an input tunnel. Repeat the procedure for converting the tunnel to a shift register. The block diagram should now look like Figure 4.7.

Wiring the *Reinitialize To Default* invoke node of the STOP button to the main-loop error shift register accomplishes two important goals. First, it ensures that the STOP button is set to False before the main loop begins. Second, it initializes the error shift register and overwrites any error information from previous runs of the VI.

Wire the Guard Clause and the Main Case Structure

Wire the error shift register on the left edge of the main loop to the selector of the guard clause. The edges of the structure should change color: green for the *No Error* case and red for the *Error* case. Pass the error wire through the guard clause to create an output tunnel, wiring the error output tunnel in both cases. Wire the state shift register to the left edge of the guard clause as well, creating another input tunnel. In the *No Error* case, wire this tunnel directly to the other side of the guard clause, creating an output tunnel. In the *Error* case, right-click on the output tunnel and select *Create...Constant* to automatically create a state constant. Select the *Shutdown* state. Take the opportunity to document both cases of the guard clause with free labels.

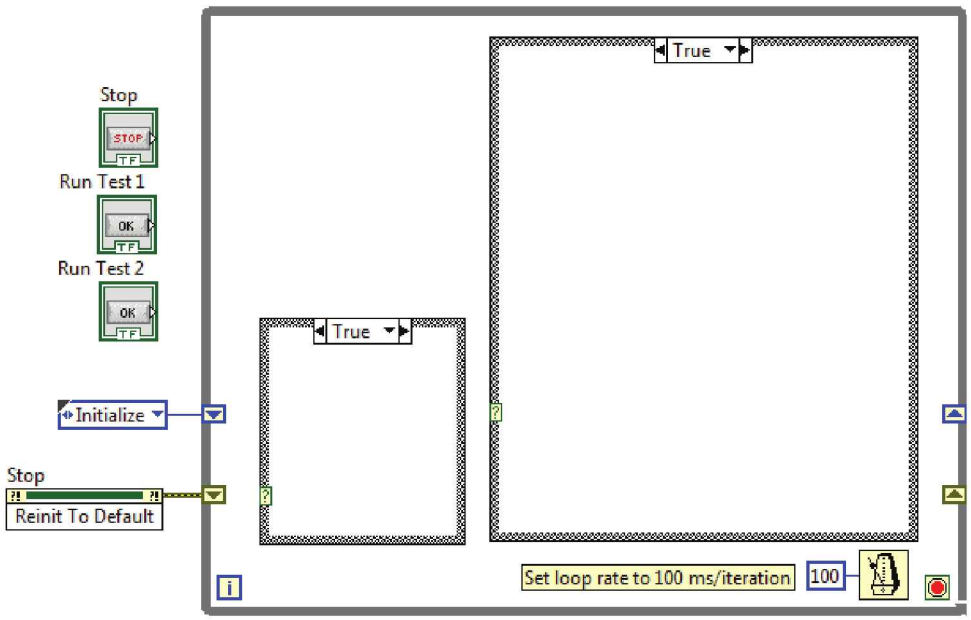


Figure 4.7: Creating and initializing the shift registers for the `Multitest` VI

Style Guide: It is good LabVIEW practice to use a free label in each case of every case structure in a VI.

Now run the state wire from the output of the guard clause to the selector of the main case structure. Right-click on the case menu at the top of the case structure and select *Add Case for Every Value*. This will automatically create a case for every state in the enumerated constant.

The `Multitest` VI will be polling the STOP, RUN TEST 1, and RUN TEST 2 buttons in the *Wait* state, so place them there for now.

Build the *Shutdown* State

Select the *Shutdown* state and drop a Boolean True constant in the lower-right corner. Wire the constant to the conditional terminal of the main loop. Right-click on the Boolean output tunnel of the case structure and select *Use Default if Unwired*. This will cause the other states to send a Boolean False (the default value of a Boolean) to the loop conditional terminal.

Drop another state constant in the *Shutdown* state and set it to *Shutdown*. Wire it to the edge of the case structure, creating an output tunnel. Then wire the tunnel to the state shift register.

Wire the error line across the main case structure to the output tunnel and then to the right-hand shift register. Drop a **`General Error Handler`** function outside

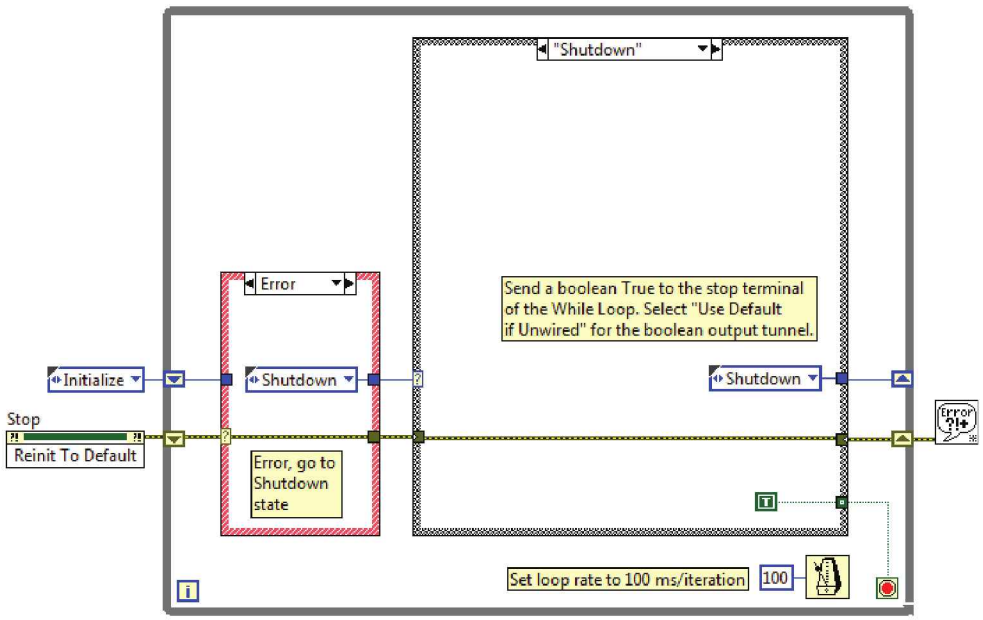


Figure 4.8: The *Shutdown* state of the `Multitest` state machine

the loop and wire it to the right-hand shift register. Your code should now look that shown in Figure 4.8.

Build the *Initialize* State

The RUN TEST 1 and RUN TEST 2 buttons will be reinitialized in the *Initialize* state. To do this, create *Reinitialize To Default* invoke nodes and place them in the *Initialize* state. Wire their error input and output terminals to the error tunnels of the main case structure, as shown in Figure 4.9. Wire a state enumerated constant to the state output tunnel and set it to the *Wait* state.

Look at the case selector and check to see if the *Initialize* case is the default case. If it is not, right-click on the selector and select *Make This The Default Case*.

Build the *Run Test 1* and *Run Test 2* States

The *Run Test 1* and *Run Test 2* states simulate tests by displaying a dialog window with the text `Run Test 1` and `Run Test 2`. This is done using the LabVIEW **`One Button Dialog`** function (see the help file in Figure 4.10). The completed *Run Test 1* state is shown in Figure 4.11, and the completed *Run Test 2* state is shown in Figure 4.12. Both of these states transition back to the *Wait* state when the user closes the dialog window.

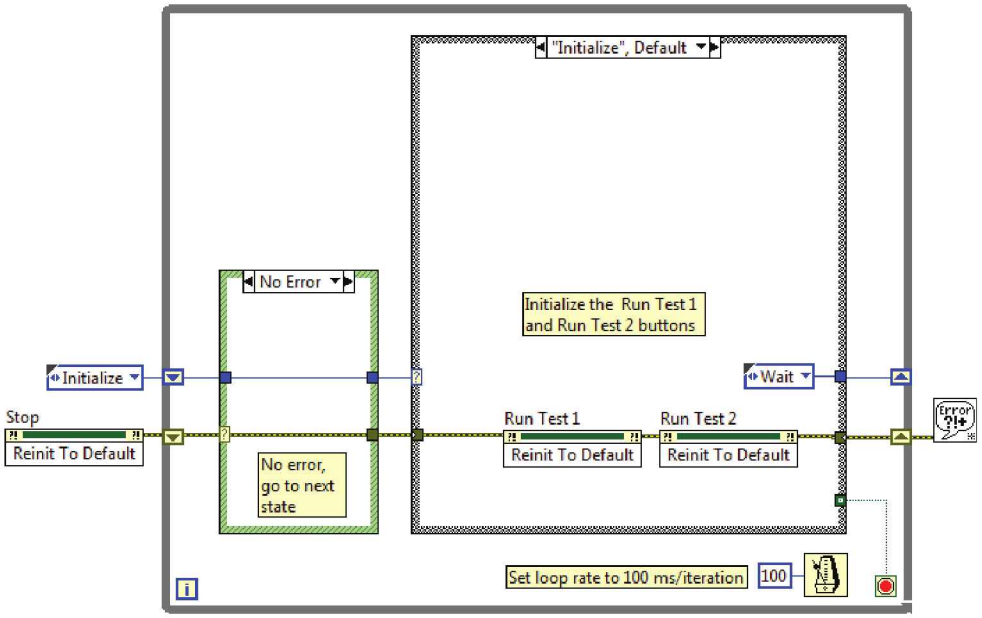


Figure 4.9: The *Initialize* state of the `Multitest` state machine

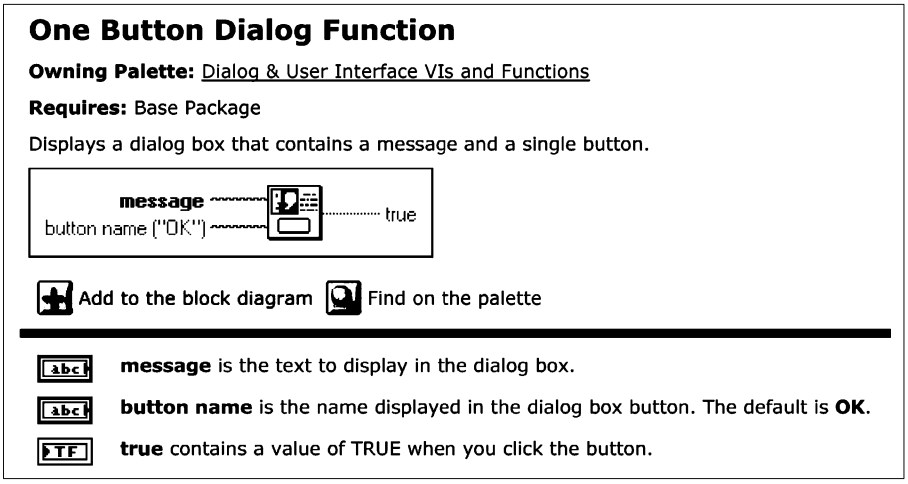


Figure 4.10: Help file for the `One Button Dialog` function

Build the *Wait* State

The *Wait* state is the most awkward and complicated state in the `Multitest` VI. The function of the *Wait* state is a demanding one: It must poll all front panel objects and perform whatever calculations are necessary to determined the next state transition.

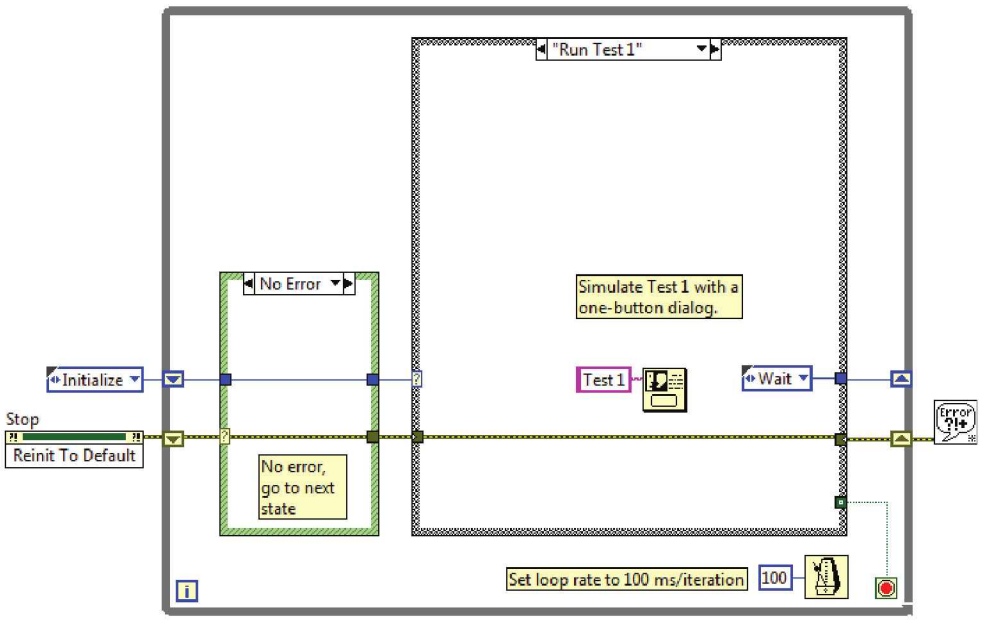


Figure 4.11: The *Run Test 1* state of the `Multitest` state machine, where the test is simulated by a one-button dialog and after the completion of the test the state machine returns to the *Wait* state

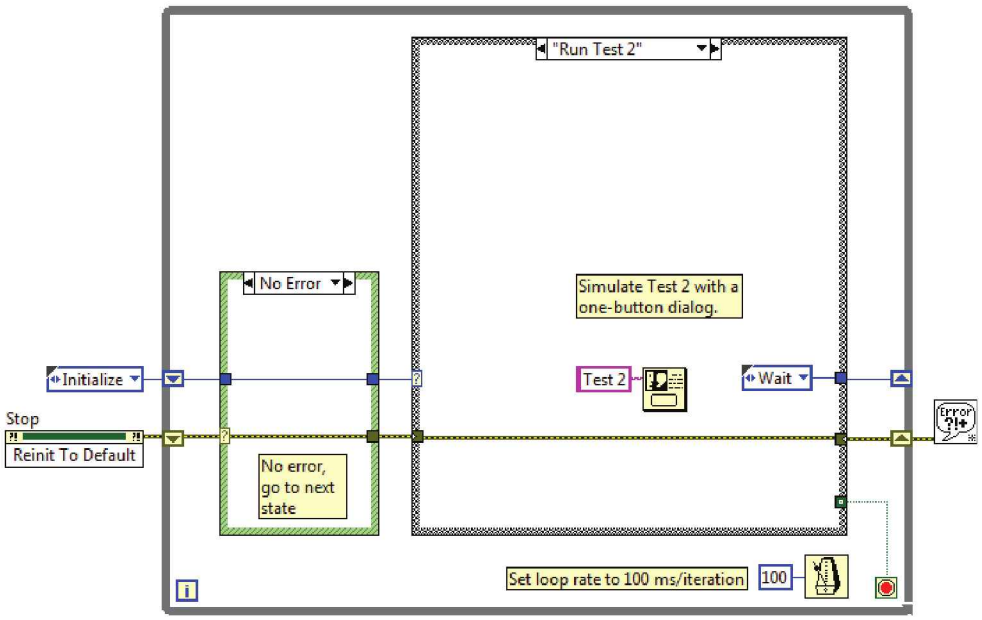


Figure 4.12: The *Run Test 2* state of the `Multitest` state machine, where the test is simulated by a one-button dialog and after the completion of the test the state machine returns to the *Wait* state

The `Multitest` VI has relatively few front panel objects, so this task isn't very difficult. But as the number of front panel objects increases, polling all of the objects becomes more unmanageable. Fortunately, LabVIEW provides event structures to perform this task cleanly and easily. This topic will be addressed in Chapter 10.

In the `Multitest` VI, the front panel objects are all Booleans. This makes monitoring the front panel easier, since Booleans only have two values: True and False. One way to monitor a set of Boolean controls is to use a conditional ladder. A conditional ladder is a series of conditional statements that tests each control in turn. A conditional ladder would read something like "If STOP was pressed choose the *Shutdown* state, else if the RUN TEST 1 button was pressed choose the *Run Test 1* state, else if the RUN TEST 2 button was pressed choose the *Run Test 2* state, else go back to the *Wait* state."

A conditional ladder can be implemented in LabVIEW using the **`Select`** function with the help file shown in Figure 4.13. The **`Select`** function accepts three inputs: a Boolean selector input, a True input, and a False input. The data types of the True input and the False input must match. If the Boolean selector input is True, the **`Select`** function passes the value of the True input. If the Boolean selector input is False, the **`Select`** function passes the value of the False input. The **`Select`** function is demonstrated in Figure 4.14. This VI outputs x if $x \geq y$ and outputs y otherwise.

`Select` functions can be connected in a cascade to form a conditional ladder, as shown in Figure 4.15. By following the dataflow, you can confirm that this cascade of **`Select`** VIs behaves in the same way as the conditional ladder described previously.

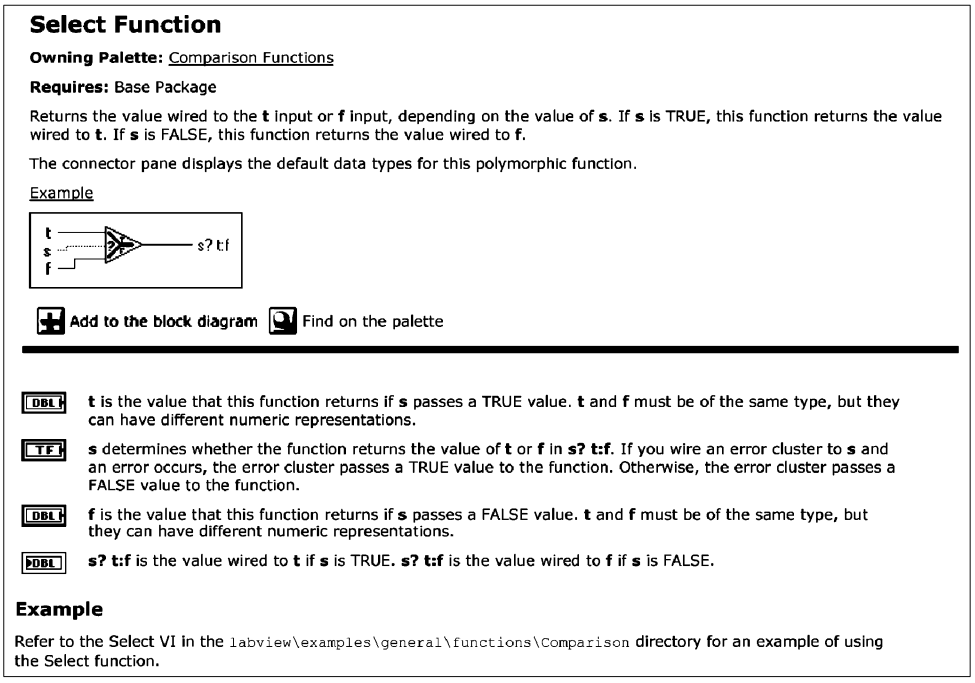


Figure 4.13: Help file for the `Select` function

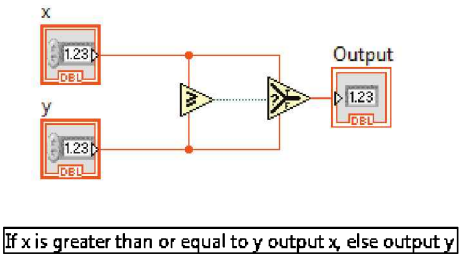


Figure 4.14: Using the `Select` function

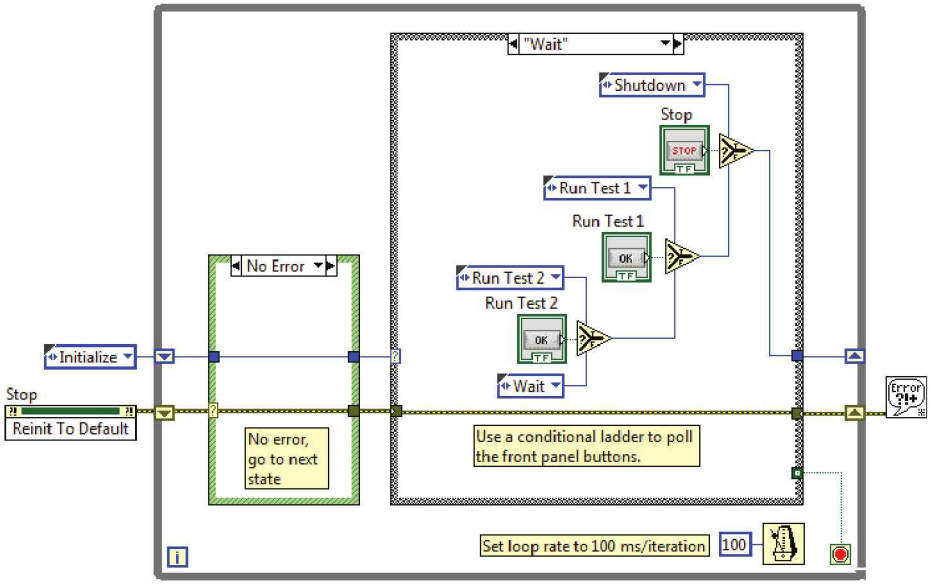
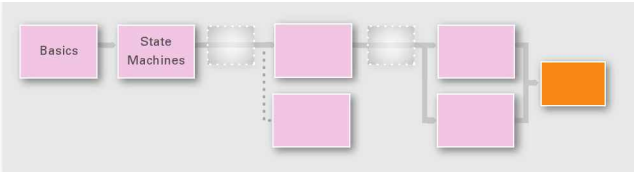


Figure 4.15: The *Wait* state of the `Multitest` state machine, where the *Wait* state uses a selector ladder to poll the front panel controls and to decide which state to go to next

If the STOP button is True, the top **`Select`** passes the *Shutdown* state to the shift register. If the STOP button is False, the result of the middle **`Select`** is passed on. The middle **`Select`** checks the status of the RUN TEST 1 button. If it has been pressed, the *Run Test 1* state is passed on to the shift register. The bottom **`Select`** function passes on the *Run Test 2* state if the RUN TEST 2 button has been pressed. Finally, the bottom **`Select`** passes on the *Wait* state if none of the buttons have been pressed.

This method is effective, but it becomes awkward for large number of Boolean controls and even more awkward if non-Boolean controls need to be monitored. Another drawback is that all of the front panel objects need to be polled every loop iteration. This is not an efficient use of computer resources.



5

Action Engines

An action engine is a sub-VI that stores data from one call to the next. It is a commonly used design pattern in the LabVIEW community, and it is frequently used in conjunction with the state machine design pattern. For example, the car wash controller described in Chapter 2 requires a timer. The timer keeps track of the total time that a vehicle has spent at a given station, pausing when the vehicle is out of position. The timer could be developed as a sub-VI, but the sub-VI would have to store the total elapsed time from one call to the next. The timer is a good candidate for the action engine pattern.

5.1 Anatomy of an Action Engine

Action engines are structurally similar to state machines—just on a smaller scale. Action engines, also known as functional global variables, have been in use since LabVIEW 2.0. LabVIEW 2.0 did not have a native global variable construct, and action engines were a way around this limitation. An action engine uses shift registers as storage elements and has the following distinguishing features:

- The sub-VI has a loop with *uninitialized* shift registers.
- The loop is hardwired to run for a single iteration.

The action engine architecture is illustrated in Figure 5.1. The action engine works because LabVIEW sets aside persistent storage for uninitialized shift registers. Every time the sub-VI in Figure 5.1 is called, the last value stored in that shift register is passed into the loop. The code inside the loop processes the value from the shift register, places a new value on the shift register, and then exits after one iteration. The single-iteration **`While`** loop is created simply as a place to hang a shift register in order to take advantage of the fact that uninitialized shift registers in sub-VIs store their values from one call to the next.

An equivalent (although rarely seen) implementation is shown in Figure 5.2. This is functionally equivalent to the template shown in Figure 5.1, but it is more

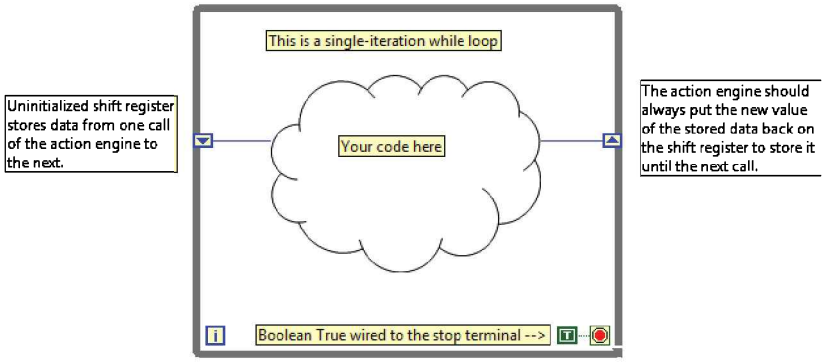


Figure 5.1: Action engine template

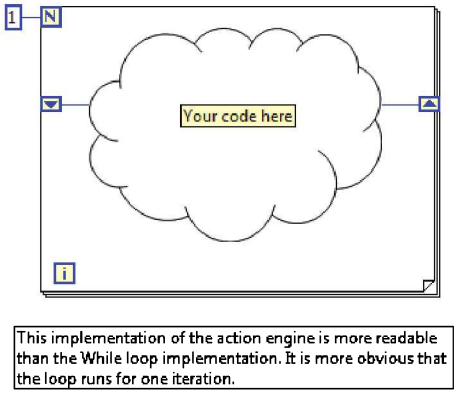


Figure 5.2: `For` Loop version of the action engine template

straightforward. One of the subtle features of Figure 5.1 that the Boolean True is hard-wired to the stop terminal of the **`While`** loop. This might be overlooked by a casual reader. It also violates the spirit of **`While`** loops, since **`While`** loops are not usually run for a fixed number of iterations.

The **`For`** loop implementation in Figure 5.2 addresses this. The `1` wired to the *N* terminal makes it obvious that this is a single-iteration loop. The **`For`** loop implementation is also a better choice, because **`For`** loops are supposed to be run for a fixed number of iterations. Single iteration loops don't violate the spirit of **`For`** loops. They do violate the spirit of **`While`** loops.

5.2 Feedback Node Action Engines

Feedback nodes also can be used to provide storage in a sub-VI. Feedback nodes can be used in a loop in place of shift registers to store data from one loop iteration to the next, as shown in Figure 5.3. Data from the previous loop iteration is stored

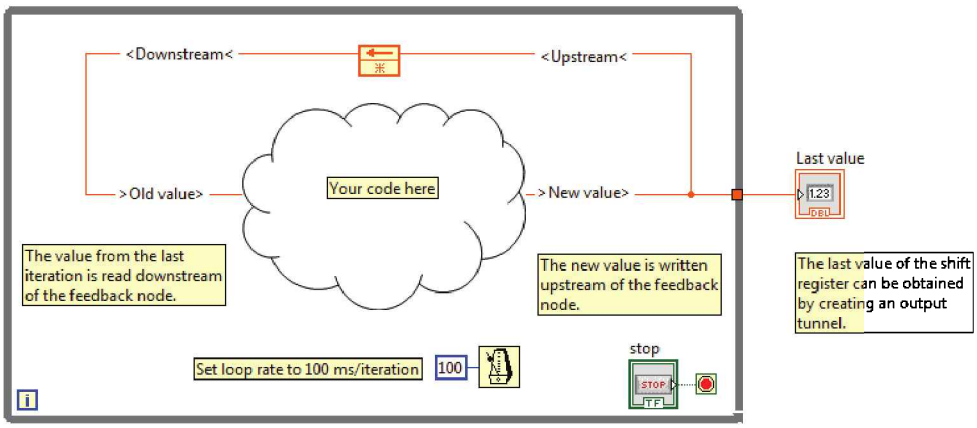


Figure 5.3: An example of how to use a feedback node, where the old value is read from the wire leaving the feedback node and the new value is written to the wire leading to the feedback node

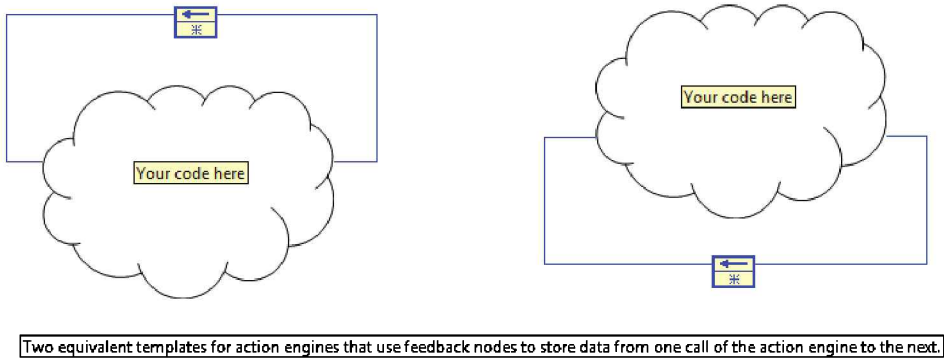


Figure 5.4: Feedback node implementations of the action engine

in the feedback node and is delivered to the next loop iteration. Feedback nodes are initialized by writing a value to the initializer terminal beneath the arrow on the feedback node icon. An unitialized feedback node in a sub-VI stores its value from one call of the sub-VI to the next, just like shift registers. Unlike shift registers, feedback nodes also can be used without a loop. A stand-alone feedback node without a loop stores information from one evaluation of the feedback node to the next without the need for a single-iteration loop structure. This is exactly the functionality needed by an action engine.

The basic template of a feedback-node action engine is shown in Figure 5.4. The feedback node points in the direction of the dataflow. The value that is fed into the back of the arrow is stored until the next time that the feedback node is evaluated. When the feedback node is evaluated, the stored value is fed from the node in the direction of the arrow on the feedback node.

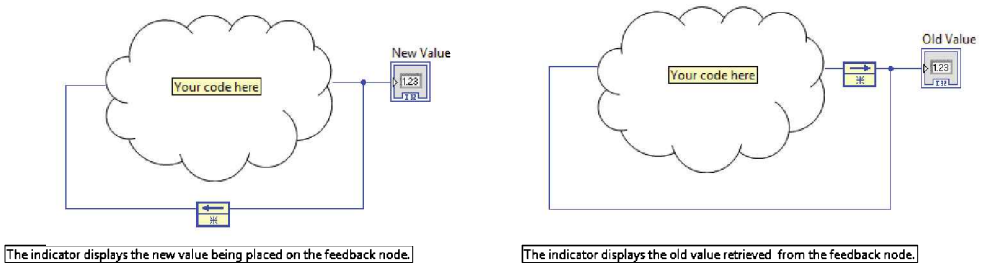


Figure 5.5: Changing the direction of the feedback node can make it easier to display the old value of the feedback node

The feedback node also has an initialization terminal below the arrow. If a value is wired to this terminal, that value is passed to the feedback-node output terminal the first time the node is evaluated. If no value is wired to the initialization terminal, the default of the data type wired through the node is passed on the first evaluation of the node.

The direction of the arrow can be flipped so that it points from left to right instead of from right to left. This can be handy in some situations. Figure 5.5 shows an example of a flipped feedback node. In the diagram on the left, the feedback node is downstream of the indicator. The value displayed by the indicator is the "new" value of the feedback node, which is the value being fed to the feedback node for storage. In the diagram on the right, the feedback node is upstream of the indicator. The value displayed by the indicator is the "old" value of the feedback node, which is the value that was stored from the previous evaluation of the feedback node.

Wiring Tip: To flip the direction of a feedback node, right-click on the node and select *Change Direction*.

An action engine that uses a feedback node has the following advantages:

- The feedback node is specifically designed to provide the functionality needed by the action engine.
- The feedback node provides a more readable block diagram.
- The feedback node is easier to create than a single iteration loop with shift registers.
- The feedback node has less overhead than a single-iteration loop with shift registers.

The feedback-node action engine has two disadvantages. First, it is backwards compatible only to LabVIEW 8.5. Prior to LabVIEW 8.5, feedback nodes only could be used inside loops. Second, feedback-node action engines have not been widely

used by the LabVIEW programming community. Because of this, it is advantageous to be familiar with both the **`While`** loop and feedback node implementations of action engines.

5.3 Example: Creating a Counter

Action engines are often used as counters. A counter must store the current value of the count from one call to the next, so an action engine is a natural choice for implementing a counter. Counters typically have at least three functions: *Reset*, *Get Value*, and *Increment*. These functions correspond to setting the counter to zero, retrieving the current value of the counter, and incrementing the current value of the counter. A counter may also have a *Decrement* function to allow decrementing the current value of the counter.

These functions easily can be implemented in an action engine. A feedback node implementation is shown in Figures 5.6 and 5.7. The *Reset* function sets the stored counter value to zero. The *Get Value* function reads the stored counter value, displays it, and places it back into storage.

The *Increment* and *Decrement* functions read the stored counter value, operate on it, and then place it back into storage. Many action engines follow this pattern: read the stored data, operate on it, and store the new values of the data. **`While`** and **`For`** loop versions of the counter are shown in Figure 5.8.

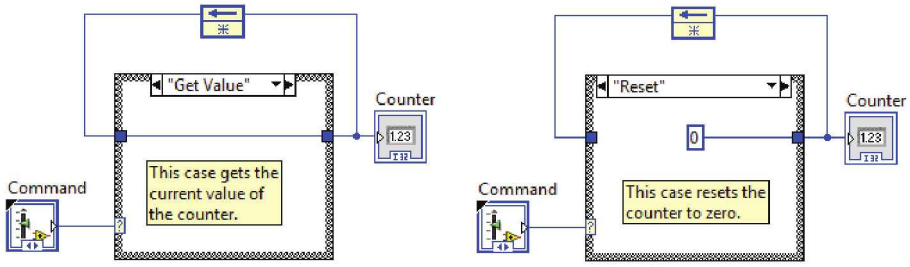


Figure 5.6: The *Get Value* and *Reset* functions of a counter action engine

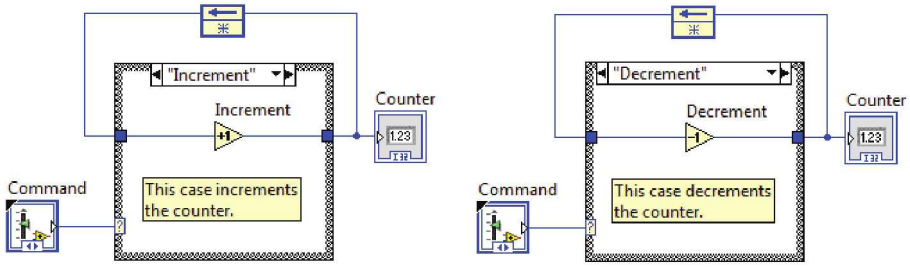


Figure 5.7: The *Increment* and *Decrement* functions of a counter action engine

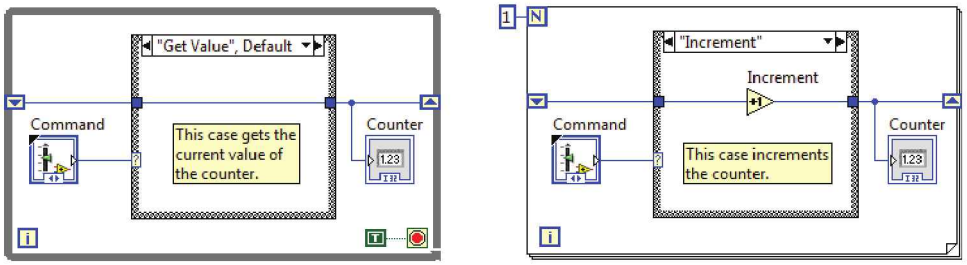


Figure 5.8: Single-iteration loop implementations of the counter action engine

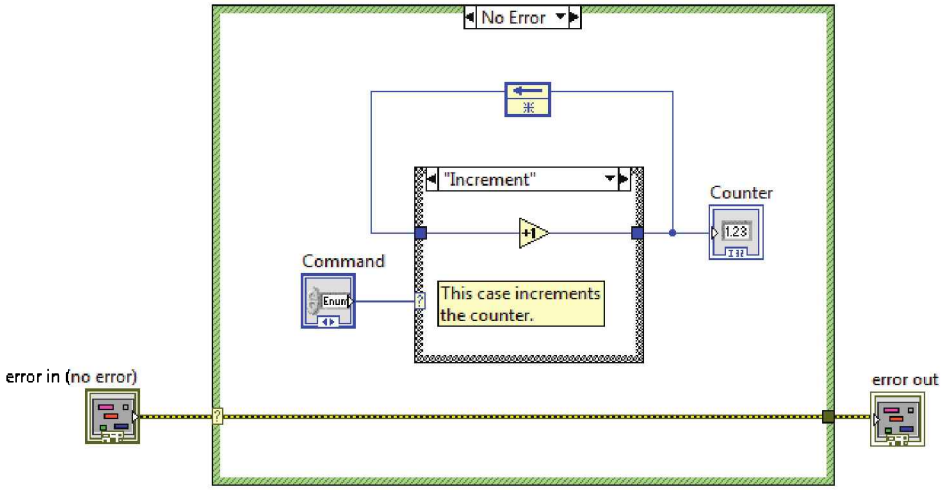


Figure 5.9: The block diagram of the fully developed counter action engine, including error handling

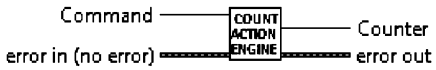


Figure 5.10: The icon of the fully developed counter action engine including error handling

A fully developed version of the counter including error handling is shown in Figure 5.9. The icon for this action engine is shown in Figure 5.10.

5.4 The State Variable Method

Action engines have two distinguishing characteristics: they store data persistently and they provide methods to operate on the stored data. In the examples presented so far, these two characteristics have been intertwined in the structure of the

action engine. But what if the methods of the action engine could be separated from the storage mechanism? This would improve code maintainability, allowing the data storage mechanism to be changed without impacting the methods for manipulating the data.

One way to separate the storage from the methods is to use a state variable approach. State variables are variables that describe the history of a system. In an action engine, state variables are the pieces of data that need to be stored from one call to the next.

In the state variable approach, the methods of the action engine are captured in a kernel sub-VI. The kernel has no storage of its own. The action engine passes the state variables to the kernel as inputs. The kernel then executes and produces new values of the state variables as outputs. The state variables are then stored by the action engine until the next call. The procedure is given here.

1. Define the set of functions that will be performed by the action engine. These are the methods of the action engine.
2. Design an algorithm for each method. Identify the pieces of data that need to be stored persistently. These are the state variables of the action engine.
3. Create the kernel of the action engine. The kernel is a sub-VI consisting of a case structure with an enumerated control input. The enumerated control lists the methods of the action engine. Each method is handled by a separate case in the case structure. Create input and output terminals for the state variables.
4. Create a new sub-VI that provides a means of persistent storage for the state variables and passes them through the kernel sub-VI. This new sub-VI is the action engine.

As an example of this process, let's create a simple stopwatch action engine. The stopwatch has two methods: *Reset* and *Elapsed Time*. Resetting the stopwatch sets the elapsed time to zero. The *Elapsed Time* function returns the elapsed time since the stopwatch was last reset. The stopwatch will have an enumerated control input to select the function, and it will have a single output to display the elapsed time.

The algorithm for the *Elapsed Time* method of the stopwatch goes like this:

1. Get the current time stamp.
2. Subtract the old time stamp from the current time stamp to get the elapsed time since the last call of the stopwatch.
3. Add the elapsed time since the last call of the stopwatch to the total elapsed time since the stopwatch was last reset.
4. Overwrite the old time stamp with the new time stamp.
5. Overwrite the old elapsed time with the new elapsed time.

This algorithm assumes that two pieces of information are being stored: the old time stamp and the elapsed time since the stopwatch was last reset. These will be the state variables of the stopwatch action engine.

The algorithm for the *Reset* function is much simpler:

1. Get the current time stamp.
2. Overwrite the old time stamp with the new time stamp.
3. Set the elapsed time to zero.

These algorithms can be implemented in a `Stopwatch Kernel` sub-VI. The icon for the sub-VI is shown in Figure 5.11. The *Elapsed Time* method is shown in Figure 5.12, and the *Reset* method is shown in Figure 5.13.

In this sub-VI, the two state variables have been bundled into a cluster. The elapsed time and time stamp are passed into the kernel, modified, and then passed



Figure 5.11: Icon for the `Stopwatch Kernel` sub-VI

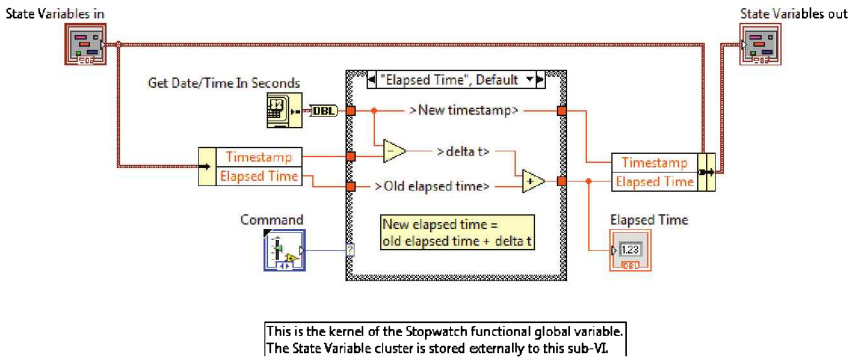


Figure 5.12: The *Elapsed Time* case of the `Stopwatch Kernel` sub-VI

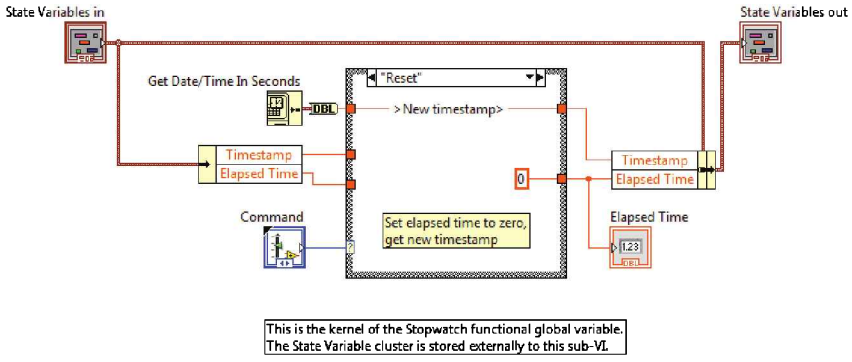


Figure 5.13: The *Reset* case of the `Stopwatch Kernel` sub-VI

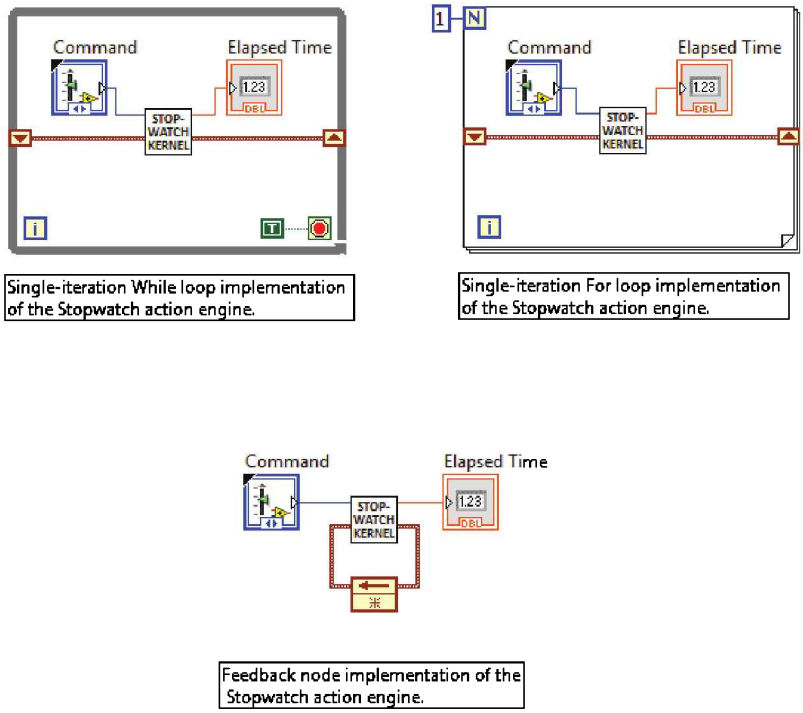
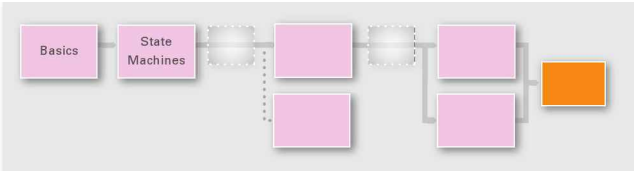


Figure 5.14: A comparison of various ways to provide persistent storage for the `Stopwatch` action engine

out of the kernel. The kernel provides no storage and is independent of the storage method.

Figure 5.14 shows three ways to provide persistent storage for the stopwatch kernel. The first two methods use uninitialized shift registers in single-iteration loops. The third method uses a feedback node. All three cases use the same kernel—only the storage method changes. The storage method is totally uncoupled from the functionality. This makes the action engine easier to read and maintain. If the storage method needs to change in the future, the kernel will be unaffected.

Of the three methods, the feedback node is the easiest to read and understand. The reader does not have to know arcane details about uninitialized shift registers and single-iteration loops to understand that data is being stored from one call of the sub-VI to the next. The feedback node is also the easiest to implement.



6

The Car Wash Timer

In this chapter, we design an action engine timer for the car wash controller. Over the course of the book, the timer form changes to suit different design patterns, but the changes primarily are for the way the timer stores data from one call to the next. The kernel of the timer won't change, because every version of the timer must satisfy the same functional specifications.

The state variable approach will be used to design the car wash timer:

1. Define the set of functions that will be performed by the action engine. These are the methods of the action engine.
2. Design an algorithm for each method. Identify the pieces of data that need to be stored persistently. These are the state variables of the action engine.
3. Create the kernel of the action engine. The kernel is a sub-VI consisting of a case structure with an enumerated control input. The enumerated control lists the methods of the action engine. Each method is handled by a separate case in the case structure. Create input and output terminals for the state variables.
4. Create a new sub-VI that provides a means of persistent storage for the state variables and passes them through the kernel sub-VI. This new sub-VI is the action engine.

This procedure has quite a few layers of abstraction built into it and may seem like more work than is necessary. However, it is these layers of abstraction that make the resulting action engine maintainable and reusable. At the highest level of abstraction, the user sees an action engine on the block diagram of the main VI. Opening the action engine reveals a kernel sub-VI attached to a method of storage. The logic is contained in the kernel and is totally divorced from the method of storage. Within the kernel sub-VI is a set sub-VIs, each one encoding the logic for one method of the action engine. These individual methods can be tested separately and reused in other versions of the action engine.

These levels of abstraction may or may not be necessary for a given project. Small, informal codes used primarily by the developer may not need a process this formal.

But larger projects, especially projects maintained by programmers other than the original developer or developers, will benefit from this approach.

6.1 The Functions of the Car Wash Timer

The requirements for the car wash timer can be extracted from the functional specification of the car wash controller found in Chapter 2:

- The timer needs to keep track of the amount of time that a car is in position at a given station within the car wash.
- When an operation begins at a station, the elapsed time needs to be reset to zero.
- The timer must signal if the time target has been met.
- The timer must accept a Boolean input that simulates a position switch. If the car moves out of position, the timer should pause, and the VEHICLE OUT OF POSITION LED should be lit. Time spent while out of position should not count against the elapsed time.
- If the car is in position and the timer is running, the LED lamp specifying the current operation should be lit.
- When the timer expires, the VEHICLE OUT OF POSITION LED should be lit until the car moves into position for the next operation.

This list of requirements suggests that the car wash timer needs two distinct actions. The primary action is called *Timecheck*. The *Timecheck* role is to take a time target and a proximity switch as inputs and keep track of the elapsed time. The second action is called *Reset*. This will reset the elapsed time to zero and do whatever else is necessary to initialize the timer at the start of a wash cycle.

6.2 Inputs of the Car Wash Timer

Each cycle of the car wash has a time target associated with it. For instance, the *Main Wash* cycle has a 5 second duration in both the Deluxe and Economy Washes. Each cycle also monitors one position switch to see if the vehicle is in position or not. If a vehicle moves off of the switch during a cycle, the timer is paused. When the vehicle moves back into position, the timer resumes. Finally, when the time target is reached, the timer needs to be reset in preparation for timing the next cycle of the car wash.

This suggests that the car wash timer needs three inputs: the target time in seconds, a proximity switch, and a command input to select the *Timecheck* or *Reset* actions. A double-precision numeric control can be used for the time target, and a Boolean control can be used for the proximity switch. An enumerated control can be used to select the desired function of the timer.

6.3 Outputs of the Car Wash Timer

There are two obvious outputs for the car wash timer, since there are two indicators that need to be lit. If the vehicle is out of position or if the timer has expired, the VEHICLE OUT OF POSITION LED indicator needs to be lit. If the vehicle is in position and the target time has not been exceeded, the corresponding cycle indicator needs to be lit.

The timer should have two Boolean outputs corresponding to these indicators. A Running? Boolean output could be used for the cycle indicator. This Boolean would be True when the vehicle is in position and the timer is running. It would be False when the vehicle is out of position or the timer has expired.

A Paused? Boolean output could be used to light the VEHICLE OUT OF POSITION LED lamp. This Boolean would be True if the vehicle were out of position or the timer expired. It would be False when the vehicle was in position and the timer was running.

There is third Boolean output that would be useful. The timer needs to signal when a cycle is done. A cycle is done when the target time has been met or exceeded. A Done? Boolean would be True when the target time had been met or exceeded and would be False otherwise.

These three Booleans are related to each other. The cycle is running if the vehicle is not out of position and the cycle is not done. Another way of saying this is that the the cycle is running if neither the vehicle is out of position nor the cycle is done:

Running? = NOT(Done?) AND NOT(Out of Position?) = NOR(Done?, Out of Position?)

Since these three Boolean outputs are related, it is a good idea to bundle them into a single output cluster. This allows all three values to be passed on a single wire. The **`Unbundle By Name`** function can be used to access the individual Boolean outputs on the block diagram. This displays the individual Boolean labels and makes the block diagram more readable. In this chapter, the output cluster will be called the *Status* cluster.

6.4 Icon and Front Panel of the Car Wash Timer

Now that the inputs and outputs have been specified, the icon and front panel can designed. The icon for the car wash timer used in this book is shown in Figure 6.1. The front panel of the timer is shown in Figure 6.2.

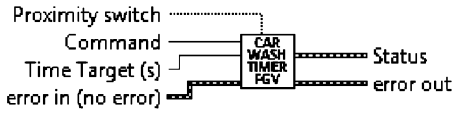


Figure 6.1: Icon for the car wash timer action engine

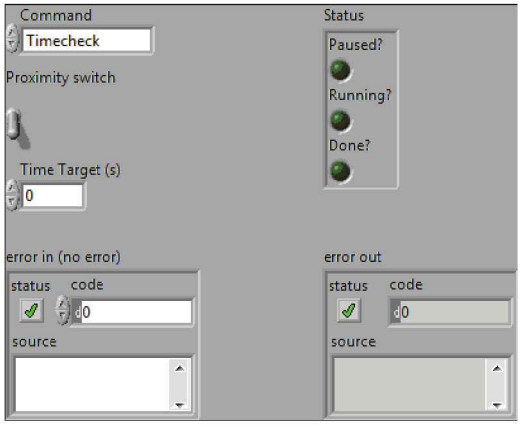


Figure 6.2: Front panel for the car wash timer action engine

Style Guide: Sub-VI front panels are not seen by the user, so don't spend too much time designing them. It is good practice to place the inputs on the left and the outputs on the right in the approximate order that they are wired on the connector pane. Align the edges to make the columns neat and readable.

6.5 State Variables of the Car Wash Timer

The timer needs to keep track of the elapsed time since the timer was last reset, so the total elapsed time will need to be a state variable of the car wash timer. The timer updates the total elapsed time by calculating the time since it was last called and adding it to the previous elapsed time. This means that the time of the last call also needs to be stored as a state variable.

6.6 The *Timecheck* Action of the Car Wash Timer

The basic algorithm for *Timecheck* is:

1. Get the current timestamp.
2. Calculate Δt by subtracting the old timestamp from the current timestamp.
3. If the proximity switch is True (vehicle is in position), add Δt to the elapsed time.
4. If the proximity switch is False (vehicle is out of position), add zero to the elapsed time.
5. Check to see if the target time has been reached or exceeded. If so, set the Done? Boolean to True.
6. Set the Out Of Position? Boolean to the inverse of the proximity switch value.
7. Set the Running? Boolean to NOR(Done?,Out of Position?).

8. Overwrite the old timestamp with the current timestamp.
9. Overwrite the old elapsed time with the new elapsed time.

This algorithm can be implemented in a `Time and Proximity Check` sub-VI. The icon of this sub-VI is shown in Figure 6.3. The block diagram is shown in Figure 6.4. Note that the two state variables (the last time stamp and the elapsed time) have been wired as inputs and outputs of the VI. This sub-VI has no storage: it merely manipulates the state variables and then passes them through. The icon has been arranged so that the state variables pass through the icon at the same positions on the left and right edges to facilitate wiring.

The current timestamp is generated by the **`Get Date/Time In Seconds`** function, whose help file is shown in Figure 1.68. Notice that the **`To Double Precision Float`** function is used to convert the timestamp to a double-precision value before calculating Δt. This was done to avoid coercion dots at the subtraction function. LabVIEW can subtract timestamps but only if they are converted from the timestamp data type to a numeric data type.

Notice too that the **`Bundle By Name`** function wired to the *Status* cluster needs a cluster constant wired to its center input to set the data type of the cluster. This was accomplished by creating a type definition of the *Status* cluster and then dropping the

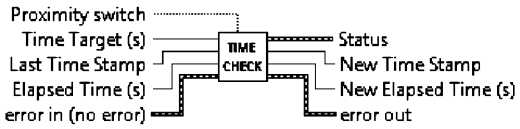


Figure 6.3: Icon of the `Time and Proximity Check` sub-VI

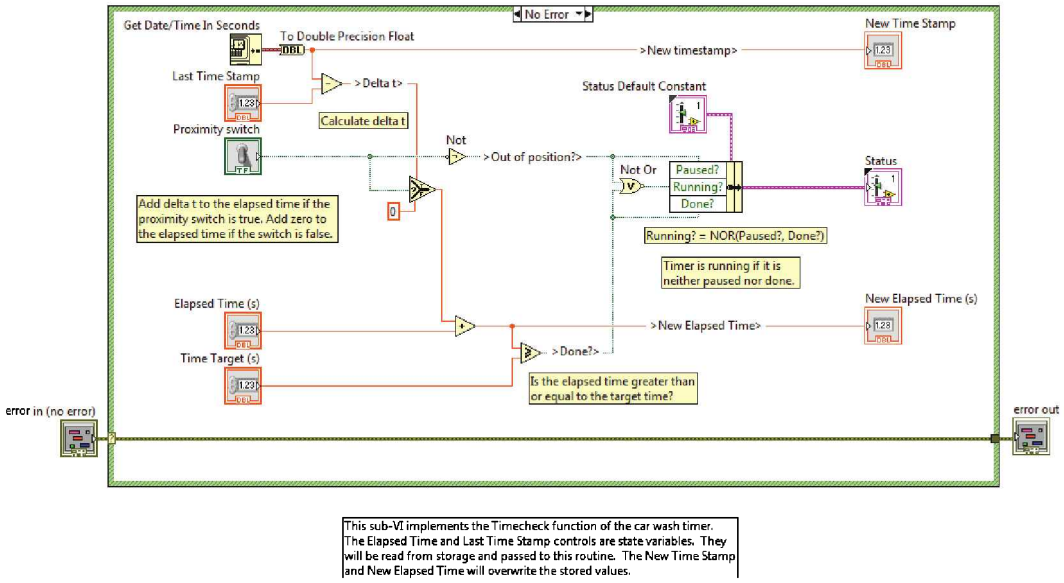


Figure 6.4: Implementation of the `Time and Proximity Check` algorithm

control on the block diagram. LabVIEW automatically converts controls to constants when they are dropped on the block diagram of a VI.

6.7 The *Reset Timer* Function of the Car Wash Timer

The *Reset Timer* algorithm is much simpler than the *Time and Proximity Check* algorithm:

1. Get the current timestamp.
2. Overwrite the old time stamp with the new timestamp.
3. Overwrite the elapsed time with a zero value.

Note that the old timestamp needs to be updated even when the timer is reset to zero. This is to ensure that Δt is calculated correctly the next time the timer is called.

The icon of the `Reset Timer Method` sub-VI is shown in Figure 6.5. The block diagram is shown in Figure 6.6. The state variables are not stored in this sub-VI: they are wired as outputs only. The `Reset Timer Method` VI does not need to read the state variables, so they are not wired as inputs—only as outputs.

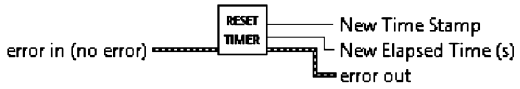


Figure 6.5: Icon of the `Reset Timer Method` sub-VI

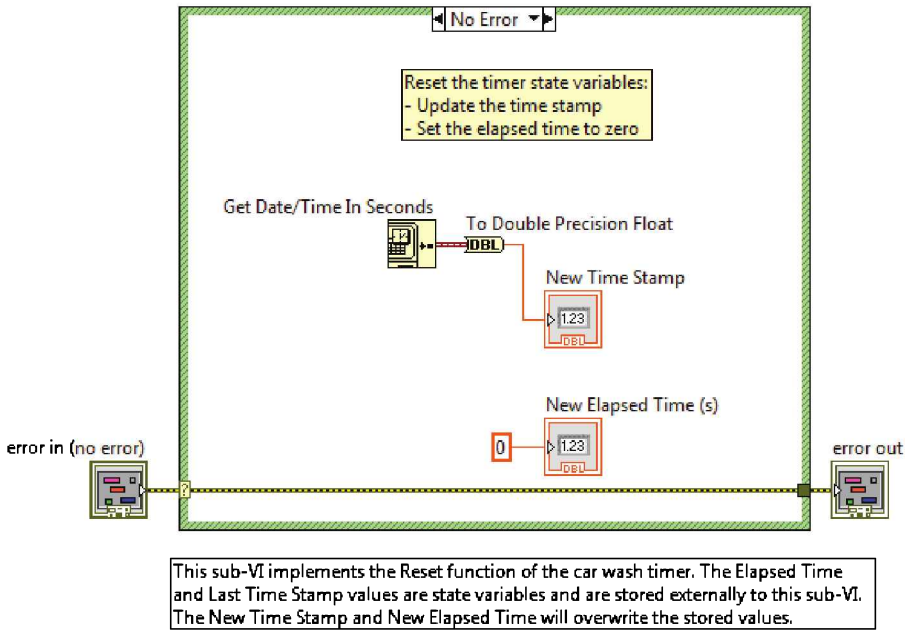


Figure 6.6: Implementation of the `Reset Timer Method` algorithm

6.8 The Kernel of the Car Wash Timer

Now that the individual functions of the car wash timer have been encapsulated as sub-VIs, it is time to create the kernel of the car wash timer. The kernel is simply a case structure with an enumerated control input that will collect the timer methods into one sub-VI.

The kernel will not have any storage itself, but it will serve as the interface between the chosen storage method and the method sub-VIs. Since it is more convenient to provide storage for one variable rather than multiple variables, the kernel will bundle the state variables into a single cluster for storage.

The icon of the `Car Wash Timer Kernel` sub-VI is shown in Figure 6.7. Notice that the state variables have been bundled into a *Timer Data* cluster and that the cluster passes through the icon. The icon also has a *Command* input. This input is an enumerated control with two items: *Timecheck* and *Reset Timer*.

The *Timecheck* case of the timer kernel is shown in Figure 6.8. The state variables are unbundled from the *Timer Data* cluster and are passed to the `Timecheck` sub-VI. The new values of the state variables are then re-bundled into the *Timer Data* cluster, overwriting the old values.

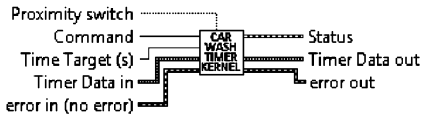


Figure 6.7: Icon of the `Car Wash Timer Kernel` sub-VI

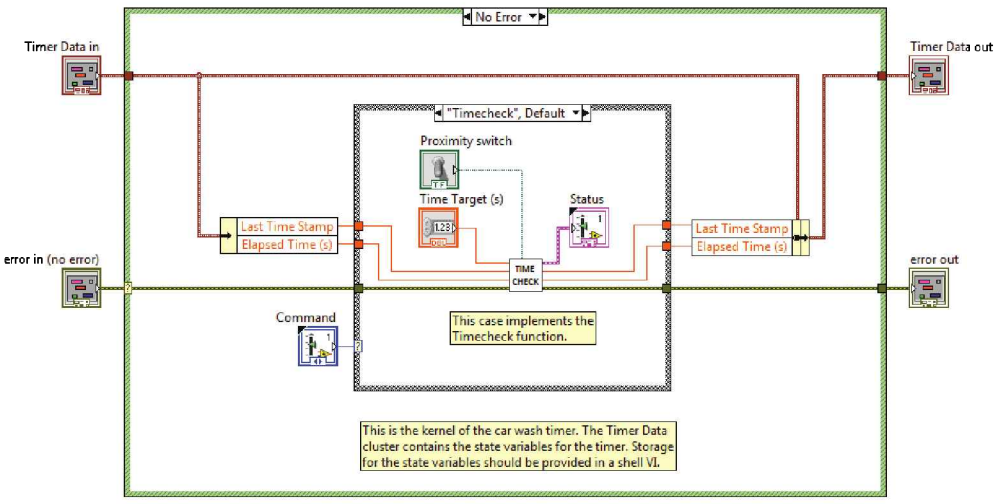


Figure 6.8: The *Timecheck* case of the `Car Wash Timer Kernel` sub-VI

The *Reset Timer* case of the timer kernel is shown in Figure 6.9. The input state variables are not needed in this function. New values of the state variables are generated by the `Reset Timer` sub-VI, and these values are re-bundled into the *Timer Data* cluster, overwriting the old values.

The state variable clusters are placed outside the main case structure to ensure preservation of the state data even when an error is present at the error input terminals of the kernel. If an error is detected in the kernel VI, the state variable data is passed through untouched, as shown in Figure 6.10.

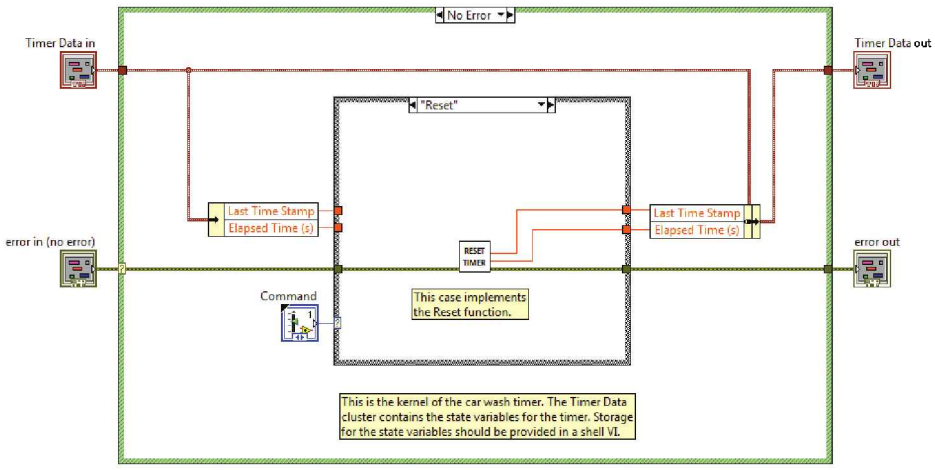


Figure 6.9: The *Reset* case of the `Car Wash Timer Kernel` sub-VI

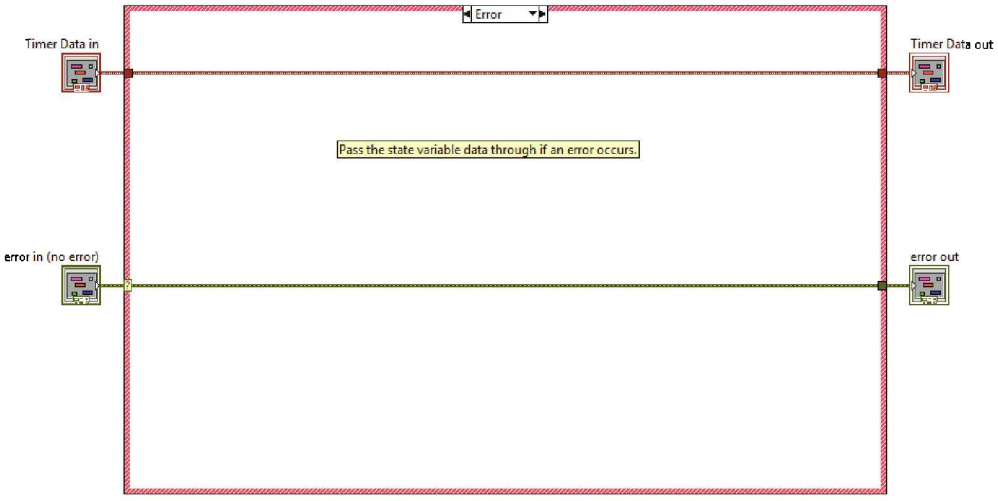


Figure 6.10: The `Car Wash Timer Kernel` sub-VI passes the state variable data through if an error is detected

6.9 Providing Storage for the Car Wash Timer

Now that the timer kernel has been created, we can choose a method for storing the state variables. The purpose of creating a kernel sub-VI is to divorce the logic of an action engine from the storage method. Now that we have such a kernel, we are free to choose whatever method of persistent storage we wish without impacting the logic of the action engine.

Figure 6.11 shows three methods of providing persistent storage that all use the same kernel: a single-iteration **`While`** loop with an uninitialized shift register, a single-iteration **`For`** loop with an uninitialized shift register, and a feedback-node implementation.

If backward compatibility with older versions of LabVIEW is a concern, the **`While`** loop and **`For`** loop implementations are the best choice, since they are compatible all the way back to LabVIEW 2.0. The feedback-node implementation would

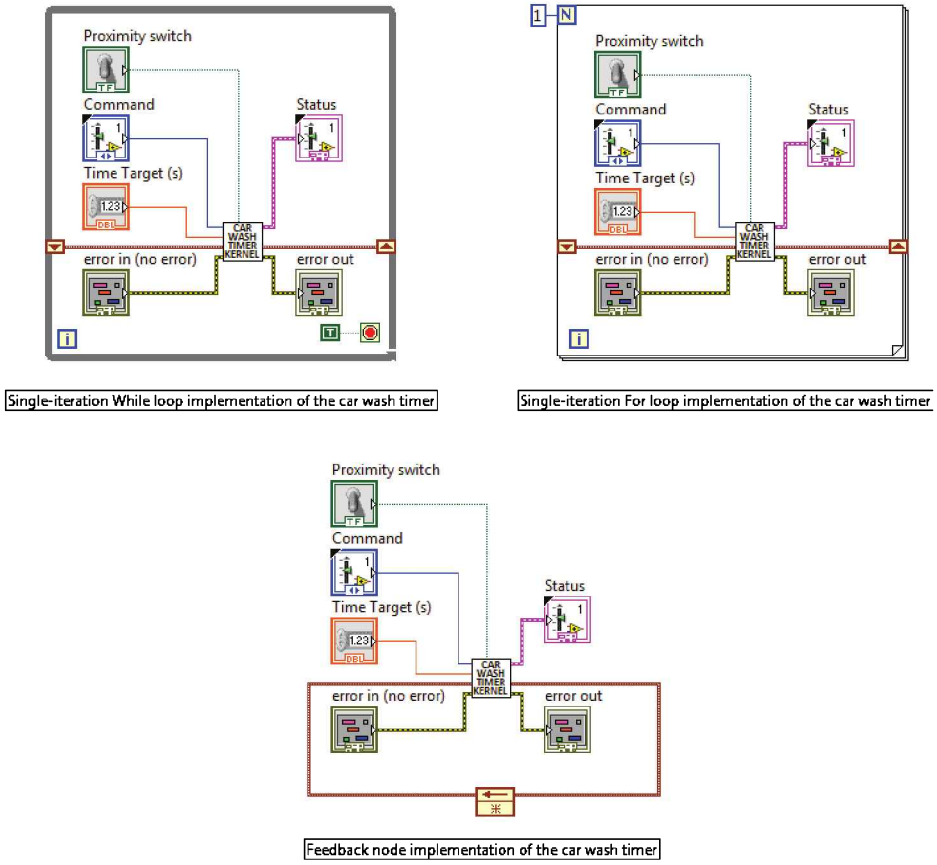


Figure 6.11: There are many ways to provide storage for the car wash timer state variables

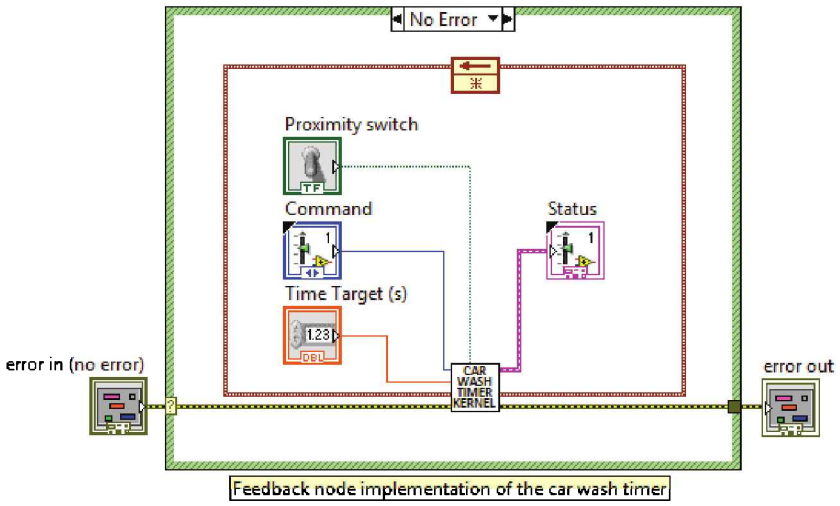
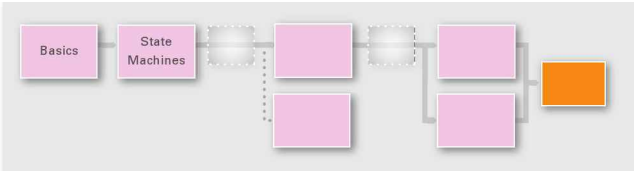


Figure 6.12: The block diagram of the `Car Wash Timer` sub-VI

be the worst choice for backward compatibility, since feedback nodes did not acquire the ability to be used outside of a loop until LabVIEW 8.5. If style compatibility with existing code is an issue, then the **`While`** loop implementation would be the best choice. This is the most recognized form for action engines in the LabVIEW community.

All other things being equal, the author's preference is for the feedback-node implementation. It is the cleanest and most readable of the four implementations. The block diagram of a timer using a feedback node and the standard error-handling template is shown in Figure 6.12.



7

Problem Set 2

7.1 Car Wash 2 Controller Exercises

The car wash timer action engine used by the original car wash easily can be modified for use by the second car wash controller. The main difference between the two car wash timers is that the second car wash controller displays the elapsed time, while the original car wash controller does not. The timer for the original car wash controller already calculates the elapsed time, so no new method sub-VIs need to be developed. We simply need to provide an output for the elapsed time in the action engine kernel and expose that output in the action engine itself.

Exercise 2.1: Creating the Kernel for the Car Wash 2 Timer

Create the `Car Wash 2 Timer Kernel` sub-VI with the icon shown in Figure 7.1. Model the sub-VI on the `Car Wash Timer Kernel` sub-VI shown in Figure 6.8, and add an output for the elapsed time.

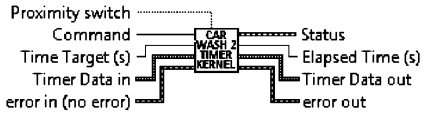


Figure 7.1: The icon for the `Car Wash 2 Timer Kernel` sub-VI ■

Exercise 2.2: Creating the Car Wash 2 Timer

Create the `Car Wash 2 Timer` action engine sub-VI with the icon shown in Figure 7.2. This sub-VI provides persistent storage for the state variables of the `Car Wash 2 Timer Kernel` sub-VI developed in Exercise 2.1.

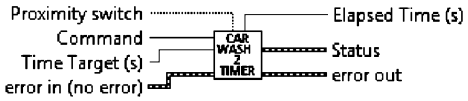


Figure 7.2: The icon for the `Car Wash 2 Timer` sub-VI ■

7.2 Boiler Controller Exercises

Exercise 2.3: Monitoring a String

Create the `String Changed` action engine with the icon shown in Figure 7.3. This action engine accepts a string input, stores it, and outputs a Boolean that indicates if the string has changed since the last time the action engine was called. The Boolean output should be True if the string has changed. Use a feedback node to store the string. Make the action engine as simple as possible. There is no need to include error inputs and outputs. Leave the initialization terminal of the feedback node unwired. When this sub-VI is executed the first time, what string inputs will cause the Boolean output to be True? What string inputs will cause it to be False? This action engine will be used in the `Boiler Controller` program.

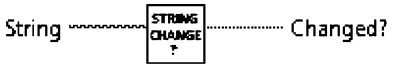


Figure 7.3: The icon of the `String Changed` VI ■

Exercise 2.4: Create a `Write To Boiler Log File` Sub-VI

Create the `Write To Boiler Log File` sub-VI with the icon shown in Figure 7.4. This boiler controller accepts an event string and a path control as inputs. If the string has changed since the last time the sub-VI was called, write the string to the log file at the given path. Add a timestamp to the beginning of the string before writing it to the file. See the functional specification of the boiler controller for details on the format of the log string. You might find the **`Format Into String`** function useful (see the help file in Figure 1.60). Use the sub-VI with the error-handling design pattern.

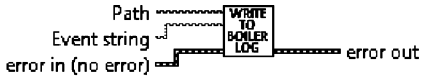


Figure 7.4: The icon of the `Write To Boiler Log File` VI ■

Exercise 2.5: Stopwatch Timer Kernel

Create the `Stopwatch Timer Kernel` sub-VI developed in Figures 5.11, 5.12, and 5.13, but add error input and output terminals and use the sub-VI with the error-handling design pattern. The icon of the new timer is shown in Figure 7.5.



Figure 7.5: The icon of the `Stopwatch Timer Kernel` sub-VI ■

Exercise 2.6: Stopwatch Timer Action Engine

Use the timer kernel developed in Exercise 2.5 to create a `Stopwatch Timer` action engine with the icon shown in Figure 7.6. Use a feedback node to provide persistent storage for the state variables. This timer will be used in the boiler controller.



Figure 7.6: Icon of the `Stopwatch Timer` action engine ■

7.3 Traffic Light Controller Exercises

The traffic light controller runs continuously without interruption. There are two possible lighting sequences: normal and left-turn. The user can influence which sequence comes next by interacting with the vehicle position sensors, but the user cannot interrupt or change the timing of the sequences. There is no delay in the timing when the lights change—as soon as one set of lights are finished the next set starts. Since the timing for each sequence is uninterrupted, an automatic reset function for the timer makes sense.

It will be convenient to create a timer that automatically resets itself when the given time target has been met. In this series of exercises, you will be asked to create an `Auto Reset Timer` action engine using the state variable method. This timer will have two input commands: one to manually reset the timer and the other to calculate the elapsed time since the last reset. The manual reset command will be used when the traffic light controller starts to ensure that the first traffic light state is timed correctly.

In the following exercises, you will first build the method sub-VIs to handle the commands of the `Auto Reset Timer`. You will then combine these methods into a kernel sub-VI that handles the state variables. Finally, you will build the action engine itself, which provides storage for the state variables.

Exercise 2.7: Create the `Auto Reset Timer` Method Sub-VIs

Create the `Auto Reset Time Check Method` with the icon shown in Figure 7.7. This sub-VI will handle the *Elapsed Time* command of the auto-reset timer. Use the following algorithm for the *Elapsed Time* function:

(a) Get the current timestamp.

(b) Calculate Δt by subtracting the old timestamp from the current timestamp.

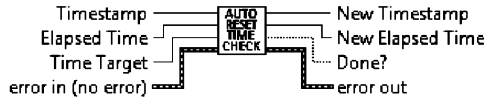


Figure 7.7: Icon for the `Auto Reset Time Check Method` VI

(c) Add Δt to the old elapsed time to get the new elapsed time.

(d) Overwrite the old timestamp with the current timestamp.

(e) If the new elapsed time is greater than or equal to the time target, overwrite the elapsed time with a value of zero. Otherwise, overwrite the old elapsed time with the new elapsed time.

The *Reset Timer* method of the auto-reset timer will use the `Reset Timer Method` sub-VI developed in Figure 6.6. ■

Exercise 2.8: Create the `Auto Reset Timer Kernel` Sub-VI

Create the `Auto Reset Timer Kernel` sub-VI with the icon shown in Figure 7.8. This sub-VI accepts an input state variable cluster containing the double-precision TIMESTAMP and ELAPSED TIME controls used and returned by the method sub-VIs developed in Exercise 2.7. It also accepts the enumerated control COMMAND input. This sub-VI unbundles the state variable data and passes it to the appropriate method sub-VI. It then collects the method sub-VI outputs and bundles them back into the output state variable cluster.

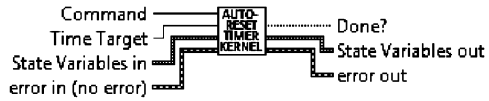


Figure 7.8: Icon for the `Auto Reset Timer Kernel` VI ■

Exercise 2.9: Create the `Auto Reset Timer` Action Engine

Create the `Auto Reset Timer` action engine with the icon shown in Figure 7.9. This sub-VI accepts a time target as an input and has a Boolean output that indicates if the time target has been met. The timer should have two input commands: one to reset the timer and the other to calculate the elapsed time since the last reset. The timer should also automatically reset itself when it determines that the target time has been met or exceeded. The sub-VI should accomplish this by providing persistent storage

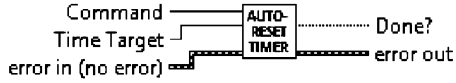


Figure 7.9: Icon for the `Auto Reset Timer` VI

of the state variables needed by the `Auto Reset Timer Kernel` sub-VI developed in Exercise 2.8.

This timer will be used in the *Traffic Light* program. Review the *Traffic Light* functional specification in Appendix A. Do you see why the automatic reset function of the timer will be useful in this application? If the timer is designed to reset automatically, why include a command for forcing a reset of the timer? Why would an automatic reset function *not* be useful in the car wash timer? ■

Exercise 2.10: Other Uses for the Auto-Reset Timer

Would an auto-reset feature be useful in the car wash timer? Why or why not? Would it be useful for the boiler controller? Why or why not? ■

7.4 Security System Controller Exercises

The following exercises relate to updating the log file in the security system controller. The security system reacts whenever a switch is pressed, but sometimes the overall system status remains unchanged. For example, if a zone is in bypass mode and the alarm switch is pressed for that zone, the zone remains in bypass mode. The log file would not be updated in this case.

The system status in the security system controller will be represented by an array of enumerated constants. In the following exercises you will develop sub-VIs that will determine if this array has changed. If it has changed, the sub-VIs will determine which zones have changed and the new status of the changed zones. You will also develop a sub-VI that will write this information to the log file.

Exercise 2.11: Create the `Diff Status Arrays` Sub-VI

Create the `Diff Status Arrays` sub-VI with the icon shown in Figure 7.10. This sub-VI performs a differencing operation on two arrays of zone status enumerated constants. This sub-VI has three outputs: a CHANGED? Boolean that indicates if the two arrays are different, a CHANGED ZONE NUMBER ARRAY that contains the zone numbers of the zones that have changed status, and a CHANGED STATUS ARRAY that contains the new statuses of the changed zones. This sub-VI should not make any assumptions about the size of the status arrays.

This sub-VI is much like the kernel of an action engine that stores the status array from one call to the next. The OLD STATUS ARRAY input accepts the stored value of the status array. The sub-VI operates on this data and outputs the NEW STATUS ARRAY for external storage, overwriting the value stored previously.

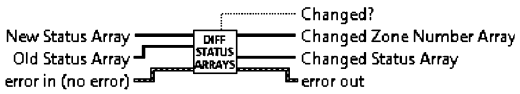
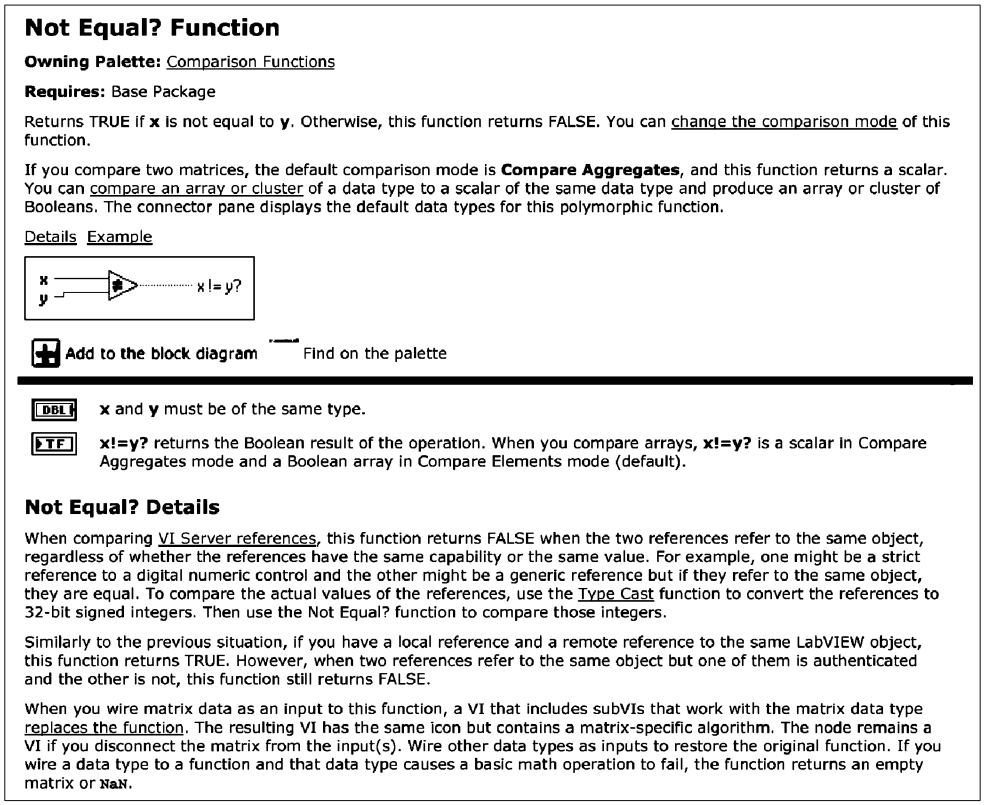


Figure 7.10: The icon for the `Diff Status Arrays` sub-VI



Not Equal? Function

Owning Palette: Comparison Functions

Requires: Base Package

Returns TRUE if **x** is not equal to **y**. Otherwise, this function returns FALSE. You can change the comparison mode of this function.

If you compare two matrices, the default comparison mode is **Compare Aggregates**, and this function returns a scalar. You can compare an array or cluster of a data type to a scalar of the same data type and produce an array or cluster of Booleans. The connector pane displays the default data types for this polymorphic function.

Details Example

x
y
x != y?

Add to the block diagram Find on the palette

DBL **x** and **y** must be of the same type.

TF **x!=y?** returns the Boolean result of the operation. When you compare arrays, **x!=y?** is a scalar in Compare Aggregates mode and a Boolean array in Compare Elements mode (default).

Not Equal? Details

When comparing VI Server references, this function returns FALSE when the two references refer to the same object, regardless of whether the references have the same capability or the same value. For example, one might be a strict reference to a digital numeric control and the other might be a generic reference but if they refer to the same object, they are equal. To compare the actual values of the references, use the Type Cast function to convert the references to 32-bit signed integers. Then use the Not Equal? function to compare those integers.

Similarly to the previous situation, if you have a local reference and a remote reference to the same LabVIEW object, this function returns TRUE. However, when two references refer to the same object but one of them is authenticated and the other is not, this function still returns FALSE.

When you wire matrix data as an input to this function, a VI that includes subVIs that work with the matrix data type replaces the function. The resulting VI has the same icon but contains a matrix-specific algorithm. The node remains a VI if you disconnect the matrix from the input(s). Wire other data types as inputs to restore the original function. If you wire a data type to a function and that data type causes a basic math operation to fail, the function returns an empty matrix or NaN.

Figure 7.11: Help file for the `Not Equal` function

You might find the **`Not Equal?`** function useful (see the help file in Figure 7.11). When this function operates on arrays, you can choose the operating mode of the function by right-clicking the function and selecting *Comparison Mode... Compare Elements* or *Comparison Mode... Compare Aggregates*. When the function compares aggregates, it outputs a single Boolean indicating whether or not the two arrays are different. When the function compares elements, it produces an array of Booleans that indicate whether or not each pair of corresponding elements in the input arrays are different. Use this array in conjunction with a **`For`** loop conditional output tunnel to find the elements of the status array that have changed. ■

Exercise 2.12: Creating the `Update Log` Sub-VI

In this exercise, you will create the `Update Log` sub-VI with the icon shown in Figure 7.12. This sub-VI will update the security system log file if necessary. To do this, it accepts both the old and new zone status arrays and compares them. If any of the zones have changed status, this sub-VI must extract the zone numbers and new

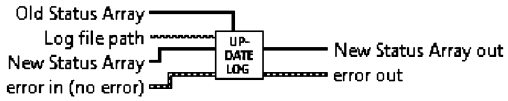
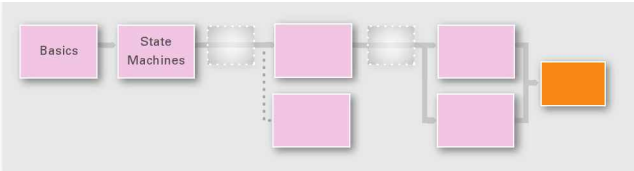


Figure 7.12: The icon for the `Update Log` sub-VI

statuses of the changed zones. It must then format this information into log strings—one log string for each changed zone. The log strings all should be given the same timestamp, but the zone numbers and statuses will be unique for each log string. The log strings then should be written to the log file in zone number order. Each log string should appear on its own line in the log file. If none of the zones have changed status, the sub-VI should exit without accessing the log file. The new zone status array should be passed as an output. It will be stored externally to this sub-VI and will be passed back again as the old status array the next time this sub-VI is called. In this way, the sub-VI acts like the kernel of an action engine that acts on an externally stored status array state variable.

This sub-VI should be written without assuming any particular number of zones. The sub-VI should also assume that more than one zone may have changed status. You might find the `Diff Status Arrays` sub-VI useful—as developed in Exercise 2.11. The `Write To Log File` developed in Exercise 1.14 also might be useful to you. ■



8

The Classic State Machine Car Wash Controller

We now apply the classic state machine and the action engine design patterns to the car cash controller. In most of the chapters in this book, the implementations of the car wash controller is presented fully developed. But in this chapter, we take the time to address the design decisions involved in creating a state machine from scratch.

There are two general approaches to designing a VI: the top-down approach and the bottom-up approach. The top-down approach starts from the big picture and works its way down to the details. The bottom-up approach starts from the details and works up to the big picture. One benefit of working with design patterns such as state machines is that it allows you to move between the two approaches easily.

The top-down approach is usually the best place to start. This approach gives you an opportunity to create a plan for a code that will meet the requirements. But in most projects, there are also lower-level technical obstacles that must be overcome if the code is to succeed, such as interfacing with hardware, communicating with other instruments, or meeting aggressive timing challenges. Waiting until the end of the development process to address these obstacles is not a good idea.

Design patterns give you a way of addressing the initial top-down design quickly and effectively. Choosing a design pattern that is well-matched to the problem at hand is a good start for the top-down design, because allows you to work on the lower-level obstacles next to prove that the problem can be solved. Once the lower-level challenges have been met, you can move back to the top-down approach and develop the rest of your code. This back-and-forth approach between top-down and bottom-up designs is known as the sandwich approach.

We've already made a good start on the car wash controller using the sandwich approach. Starting with top-down design, we have the the state transition diagram shown in Figure 2.5 and the classic state machine design pattern described in Chapter 4. Continuing with the bottom-up approach, we have the car wash timer action engine developed in Chapter 6. Switching back to the top-down approach,

we can now focus on the details of developing the classic state machine implementation of the car wash controller.

8.1 Determine the States

The first step in designing a state machine is to draw the state transition diagram, such as the one shown in Figure 2.5. One of the advantages of doing this first is that it creates a list of the needed states. The states for the car wash controller are *Initialize*, *Wait*, *Lock Selector*, *Underbody Wash*, *Soap Application*, *Main Wash*, *Rinse*, *Air Dry*, *Unlock Selector*, and *Shutdown*.

At this point in the design process, it is useful to review the list of states. Are all of the states needed? Are some of the states extraneous? Could some of the states be combined? Do some of the states need to be broken down into other states?

Two design principles that are very helpful in answering these questions are high cohesion and loose coupling. High cohesion means that each state should have a very focused task or goal, and all of the code in the state should be focused on accomplishing that task or attaining that goal. A good test for this is to write a mission statement for each state. If you can't write the mission statement of a state in a simple sentence, the state most likely lacks cohesion and should be redesigned. If the sentence is a compound sentence ("This state does this and that and this other thing and then does something else...") it is also a sign that the state has poor cohesion and should probably be divided into a set of more focused states. For example, suppose you had a state that configured an instrument, took data, analyzed it and wrote it to a file. This is too much for one state, and your code would benefit by breaking it into separate states for each of those tasks.

Loose coupling means that the tasks performed by the states are relatively self-contained and the states aren't bound tightly to each other. Look for states that only perform part of a task or states that only occur when they are preceded or followed by other specific states. For example, suppose you needed to write data to a file and you had separate states for opening the file, writing to the file, and closing the file. These tasks are very focused, but if they always happen in that same sequence and never on their own, they are too coupled. You should consider combining them into a single *Write File* state.

The states in the car wash controller list are easy to justify in terms of mission, cohesion, and coupling. The car wash states (*Soap Application*, *Main Wash*, etc.) are focused on individual car wash cycles and are independent of each other. They have high cohesion and loose coupling. The *Initialize* and *Shutdown* states perform well-defined tasks that do not overlap with the other states. The *Wait* state is necessary, because the car wash controller will spend much of its time waiting for a user to make a purchase.

The last two states on the list are the *Lock Selector* and *Unlock Selector* states. It could be argued that these two states are too "small." The only purpose of these states is to lock and unlock the purchase selector on the front panel. These functions could probably be absorbed by the other states. It is usually good programming practice to minimize the number of states in a state machine, so is it justifiable to keep them on our list as separate states?

This is a judgment call. An important consideration is that the functional specification explicitly requires that the purchases selector be locked during the car wash operation. It is important for a software developer to be able to prove that all requirements in a functional specification have been covered. Dedicating separate states to the *Lock Selector* and *Unlock Selector* functions makes it is easier to prove that the requirement of locking the purchase selector has been addressed.

8.2 Plan the State Transitions

Now that we have the list of states, we need to consider the transitions between the states. In the classic state machine, each state must set the transition to the next state. Looking back at the state diagram in Figure 2.5, we see that most of the states have only one possible transition. These transitions will be easy to code. However, three of the states have more than one possible transition: the *Wait* state, the *Lock Selector* state, and *Rinse* state.

The *Wait* state will loop back on itself until the user selects a wash type. When a selection is made, the *Wait* state always transitions to the *Lock Selector* state, no matter which type of wash was selected. The *Lock Selector* and *Rinse* states are more interesting, because these states need to know which type of wash was selected by the user before they can decide which state to move on to next.

This means we need to store the selected wash type. Following the advice given in Figure 4.4, a good way to do this is by placing it on the data highway. The purchase selector will be monitored when the state machine is in the *Wait* state. When the user selects a wash, the value of the DELUXE WASH button will be placed on the data highway for use by the *Lock Selector* and *Rinse* states.

The state transition diagram in Figure 2.5 also shows that the STOP button must be continuously monitored and the state machine should shut down if an error is detected. Both of these requirements can be met if we read the STOP button and check the error wire at the beginning of each loop iteration.

8.3 Create the State Constant

Now that the states have been defined and the transitions have been planned, it is time to start laying out the state machine. First create a type-definition enumerated control listing these states following the directions given in Chapter 1. The resulting enumerated control should look like the one shown in Figure 1.9.

Wiring Tip: If you ever need to edit a type-defined control or constant, either open and edit the type definition file directly or right-click on an instance of the control or constant in your VI and select *Open Type Def.*

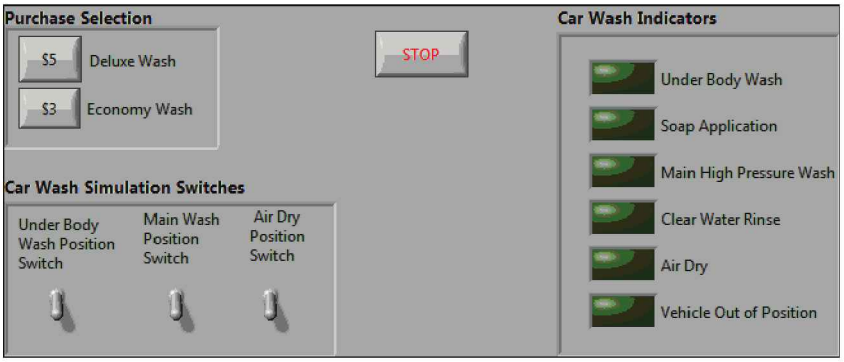


Figure 8.1: The front panel of the car wash controller showing the default values of the car wash indicators

8.4 Build the Front Panel

This step is straightforward, since a template for the front panel of the car wash controller was provided with the functional specification. Notice the visual cues on the purchase selector, proximity switches, and indicators. The front panel template suggests that these three items should all be clusters. This is a good design choice, since it is easier to work with clusters of related items rather than each individual item separately. The descriptive names on the switches are quite long—follow the advice given in Exercise 1.1, use captions for the descriptive text, and keep the labels short and descriptive. The captions will be displayed on the front panel only; the labels will be displayed on the block diagram.

The purchase selector, switch, and indicator clusters are fundamental to the operation of the state machine. They also are likely to be used by sub-VIs. This suggests that the clusters should be saved as type definitions. When the front panel is completed, it should look like the one in Figure 8.1.

Wiring Tip: Custom clusters are just as unique and valuable as custom enums. Make it a habit to save custom clusters as type definitions.

8.5 Add the Main Loop and Case Structure

The next step is to add the main loop. Follow the same basic steps that were described for the `Multitest` VI in Chapter 4. Start with the main loop and the main case structure. Size the loop to take up most of the screen area, but leave room outside both the left and right edges for start-up and clean-up code. The main case structure

should take up most of the loop area, but leave room for the guard clause case structure to the left of the main case structure.

Drop a state constant outside the loop, and create a shift register for it (as described in Chapter 1). Run the state wire through the *False* case of the guard clause, and connect it to the selector of the case structure. Create a case for every value of the enumerated constant.

Create a *Reinitialize to Default* invoke node for the STOP button, and place it in the input area of the loop. Use the invoke node to create an error line shift register on the main loop. Add a **`General Error Handler`** function to the output area of the state machine, and wire it to the output error shift register. Place the **`Wait Until Next ms Multiple`** function in the lower-right hand corner, and set it to iterate every 100 ms. The layout should now look something like Figure 8.2.

Where should the terminals for the front panel controls and indicators be placed? Following the design guidelines in Chapter 4, consider the scope and volatility of each item. The purchase selector will be polled often to see if a selection has been made. But the selector will only be polled in the *Wait* state, which is similar to the way the front panel controls were polled the `Multitest` VI in Chapter 4. This means the scope of the purchase selector is restricted to the *Wait* state only. The purchase selector cluster will not be stored, so it is a volatile item. If an item is volatile and local in scope, place it in the state where it is needed. Put the purchase selector in the *Wait* state.

Remember that the value of the DELUXE WASH button needs to be placed on the data highway so that it can be used by other states. Use the **`Unbundle By Name`** function to unbundle the value of the DELUXE WASH Boolean from the purchase selector, and wire it to the edge of the **`While`** loop. Convert the output tunnel into a shift register, and wire the left-hand shift register back to the case structure as an input. Create an appropriate label for the portion of the wire in the lobby to increase the readability of the code. The result should look something like Figure 8.3.

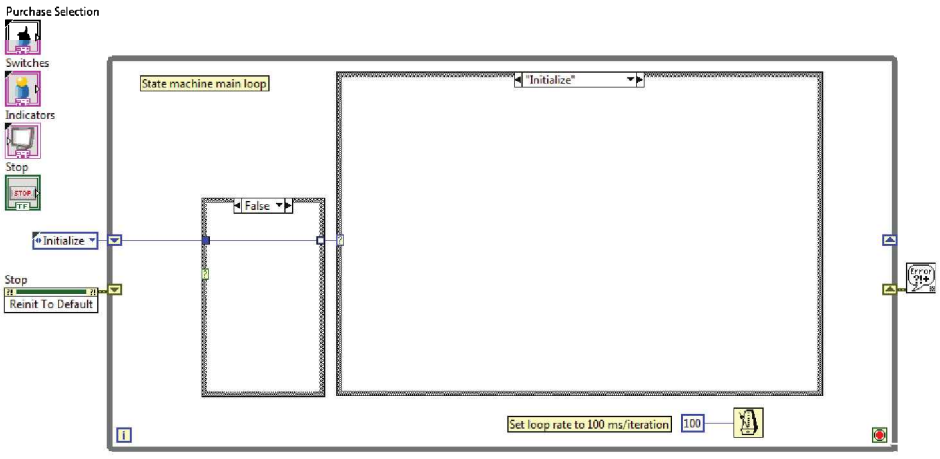


Figure 8.2: Laying out the elements of the classic state machine car wash controller

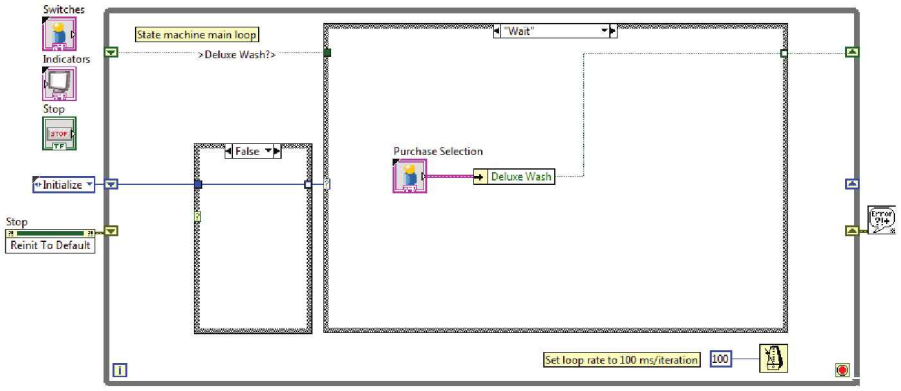


Figure 8.3: Creating the data highway for the classic state machine implementation of the car wash controller

Style Guide: It is good LabVIEW style to label wires on the data highway. It is customary to label wires using the format >*Wire Label*>, where the "arrowheads" point in the direction of the data flow. Prior to LabVIEW2010, free labels had to be used to label wires. Starting with LabVIEW2010, wires have labels of their own—just like controls and indicators. Right-click on the wire and select *Visible Items... Label*.

Now consider the proximity switches. The proximity switch cluster will change value often, since cars will be moving in and out of position at the various stations in the car wash. The switch cluster does not need to be stored, only the current value is needed. The switches are needed by multiple states. Following the advice given in Figure 4.4, place the switches in the lobby of the state machine, and wire the cluster as an input to the main case structure.

The car wash indicators are similar to the switches. Each car wash state has the potential to change the values of the indicators during each loop iteration. Following the advice given in Figure 4.4, the indicators should be placed in the alley of the state machine.

As a general rule of thumb, items in the alley of a state machine require more consideration than items in the lobby. This is because items in the lobby are inputs to the main case structure and items in the alley are outputs. Inputs to a case structure can be ignored by individual cases, but every case in the case structure MUST provide an output to every output tunnel. This means that indicators in the alley of a state machine place an obligation on every case in the case structure.

We've already noted that the car wash states (*Underbody Wash*, *Main Wash*, *Rinse*, etc.) will directly manipulate the indicators to show which cycle is active and if the

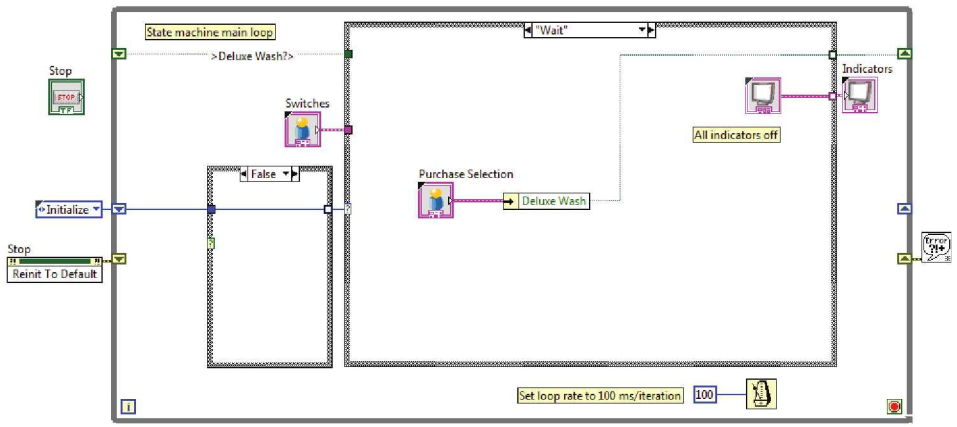


Figure 8.4: An inefficient way to set the default value of the car wash indicators in the non-wash states

car is in position. But what about the other states? What will the indicators display when the state machine is in the *Initialize*, *Wait*, *Lock Selector*, and *Unlock Selector* states? The functional specification is unclear on this topic, but a reasonable choice would be to leave all indicators unlit during these states. This will be the default value of the indicator cluster.

How will we ensure that this default value is displayed in all non-wash states? One way would be to place a cluster constant in each non-wash state and wire it to the car wash indicators, as shown in Figure 8.4. The cluster constant is viewed as an icon to improve code readability. But placing the cluster constant in the case structure causes an inefficient duplication of code and a waste of block diagram space, since every case that needs to access the cluster will have to duplicate it. This also degrades readability, because it is not obvious that the indicator default display is the same for all states until you examine each state.

Consider the location of the indicator cluster default value. The default value never changes, so it is a constant. The constant is needed by all non-wash states. This suggests that the best place to put the constant is in the input area of the state machine, as shown in Figure 8.5. Notice that in Figure 8.5 the wire proceeding from the constant has been labeled. This improves the readability of the block diagram. In this implementation, it is obvious that the non-wash states will display the default value of the indicator cluster.

Style Guide: It is good LabVIEW style to label long wires such as data highway wires or constants originating from outside the main loop. Use the same "arrowhead" format as data highway wires to indicate the direction of data flow.

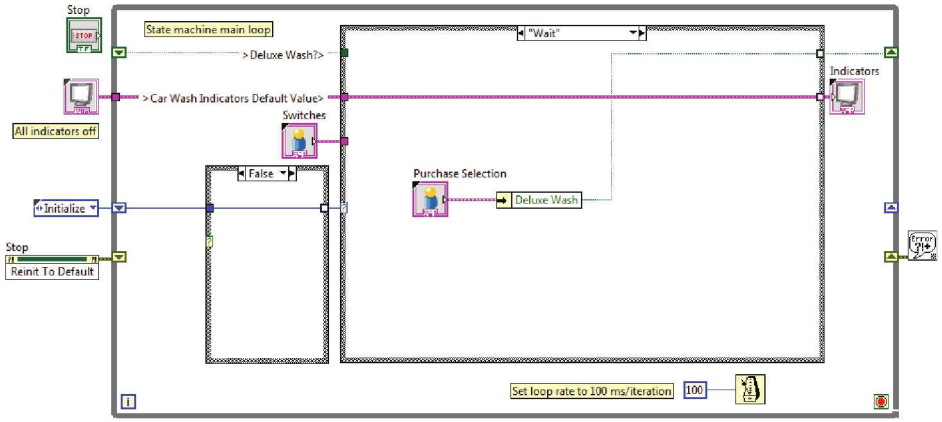


Figure 8.5: A better way to set the default value of the car wash indicators in the non-wash states, where the default is constant and is needed by multiple states, so place it in the input area of the state machine

Wiring Tip: Make sure that the constant cluster used for the default value of the indicators is also tied to the type definition file of the cluster. Create the constant by right-clicking on the indicator cluster and selecting *Create. . . Constant*, or simply drop the type-defined cluster control on the block diagram. The control will be changed automatically into a constant. If you don't create the cluster constant from the type definition, you will see a coercion dot where the cluster constant wire connects with the loop input tunnel. Make it a habit to always avoid coercion dots if possible.

8.6 Add an Error or Stop Check

We still need to place the STOP button on the block diagram. In the `Multitest` VI, we placed the STOP button in the *Wait* state, since it needed to be polled while the state machine was waiting. But the functional specification for the car wash controller calls for a different behavior.

In the car wash controller, the STOP button needs to be polled while the controller is waiting. The STOP button also needs to be polled while the controller is running a selected wash cycle since the user may stop the state machine at any time. The best place for the STOP button is the lobby of the state machine where it can be checked every loop iteration in conjunction with the guard clause.

In the `Multitest` VI, the guard clause was a case structure whose selector was directly wired to the main error line of the state machine. In the car wash controller,

we still need to check for errors, but we also need to check the STOP button. One way to do this is with a `Stop or Error` sub-VI. The purpose of this sub-VI is to poll the STOP button and to check the status of the error line. If the STOP button has been pushed or if an error is present, the `Stop or Error` sub-VI will output a Boolean True; otherwise, it will output False. This Boolean then can be used by the guard clause case structure to decide whether to pass the next state or to force a transition to the *Shutdown* state.

The icon for the `Stop or Error` sub-VI is shown in Figure 8.6. The block diagram is shown in Figure 8.7. Notice that the error wire runs directly to the **`Or`** function. This is a LabVIEW graphical convention. Whenever an error wire runs directly to a function that accepts a Boolean input, it means that the status Boolean of the error cluster is being fed to the function with the Boolean input.

The *Shutdown?* Boolean output of the `Stop or Error` sub-VI is wired to the selector of the guard clause of the car wash controller. If an error is present or the STOP button has been pressed, the *Shutdown?* output will be True, and the guard clause sends the state machine to the *Shutdown* state as shown in Figure 8.8. If there is no error and the STOP button has not been pressed, the *Shutdown?* output will be False, and the guard clause passes the next state to the selector of the main case structure. This is shown in Figure 8.9.

Notice that in Figures 8.8 and 8.9 that some finishing touches on the state machine skeleton have been added in the *Shutdown* state. The main error line has been extended through the case structure to the opposite shift register. This creates the output tunnel in the case structure for the error line. A state constant also has been added to the *Shutdown* state and has been wired to the state shift register, creating an output tunnel in the case structure for the state constant. Finally, a Boolean True has been added to the *Shutdown* state and has been wired to the loop conditional terminal. The *Use Default if Unwired* option has been selected for the Boolean tunnel.



Figure 8.6: Icon of the `Stop or Error` sub-VI

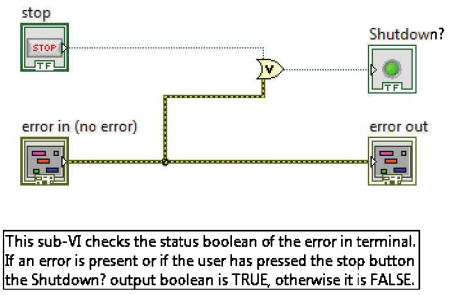


Figure 8.7: Block diagram of the `Stop or Error` sub-VI

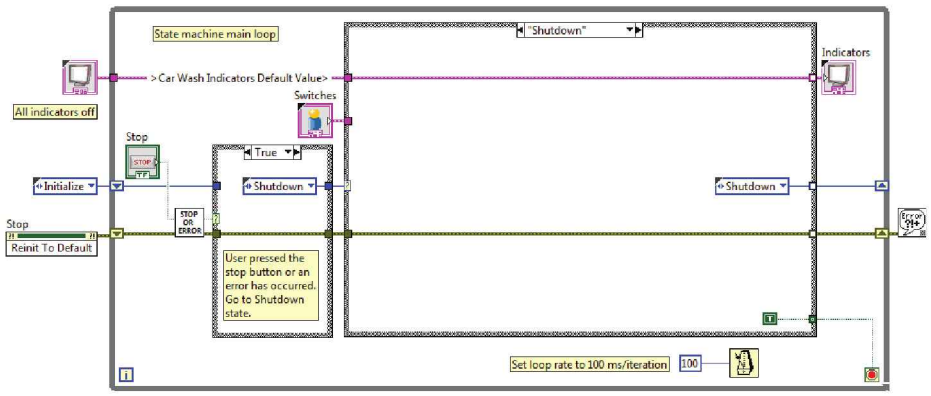


Figure 8.8: The guard clause sends the state machine to the *Shutdown* state if an error is detected or if the STOP button has been pressed

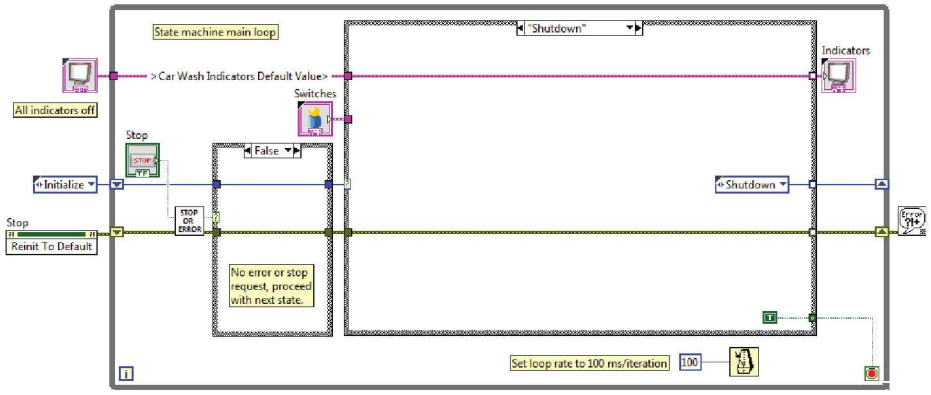


Figure 8.9: The guard clause passes the next state to the main case structure if no error is present and the STOP button has not been pressed

This completes the basic layout of the classic state machine. Here is a checklist of the steps that have been taken:

- The states have been defined and saved in a type-defined enumerated control.
- The main loop is in place with shift registers for the state constant, the main error wire, and the data highway. The shift registers have been initialized as needed.
- The loop execution timing has been implemented.
- The guard case structure is in place with all of its inputs and outputs defined. Outputs have been wired in all cases.
- The main case structure is in place with all input and output tunnels created.
- The loop termination has been wired in the *Shutdown* state.

- All front panel controls and indicators have been placed in appropriate locations.
- Important constants (the car wash indicator defaults in this case) have been created and placed in appropriate locations.
- Error handling has been implemented outside of the main loop.

8.7 Develop Individual States

Now that we've laid out the basic structure of the state machine, we can develop the individual states of the classic state machine implementation of the car wash controller. To make it easier to follow the flow of the code, we develop the states in the order that they would normally occur in the state machine.

8.8 Develop the *Initialize* State

The *Initialize* state of the car wash controller is shown in Figure 8.10. This state resets all of the front panel controls and then sends the state machine to the *Unlock Selector* state by placing the *Unlock Selector* state on the state constant shift register.

It is good programming practice to reset all front panel controls to their default state in the *Initialize* state. Even if you reset all front panel controls before a program shuts down, you should still initialize them at the start of the program. If a program is aborted using the Abort Execution button on the front panel toolbar or if the user changes the state of the front panel controls while the VI is not running, the front panel controls will be in an unknown state at run time.

There are a number of ways to reinitialize a control in LabVIEW. Reinitializing a control is simple in concept, but it should be handled carefully, because it reverses

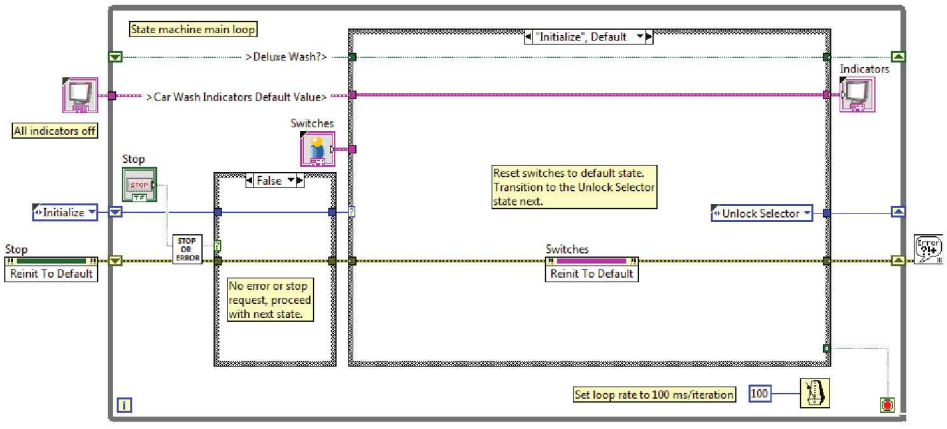


Figure 8.10: The *Initialize* state of the classic state machine car wash controller

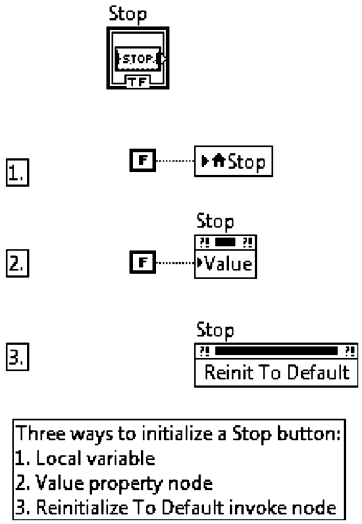


Figure 8.11: Three ways to reinitialize a control: local variables, property nodes, and invoke nodes

the normal dataflow convention. Typically, data flows *from* a control, but when you reinitialize a control, data flows *to* a control. Three commonly used methods of reinitializing a control in LabVIEW are illustrated in Figure 8.11:

- Create a local variable of the control and write to the local variable.
- Create a *Value* property node for the control and write to the property node.
- Create a *Reinitialize to Default* invoke node for the control and execute it.

Local variables are created from existing controls and indicators on the block diagram. They can be used to read from or write to these existing controls and indicators. An example of their appearance is shown in Figure 8.11. Local variables take on the color of the data type of their parent control, and the thickness of the border indicates whether they are configured to write to or read from the parent control or indicator. A thick border indicates that the local variable will read from the parent object, acting like a control. A thin border indicates that the local variable will write to the parent object, acting like an indicator. Arrowheads in the local variable also indicate whether it is configured to act as a control or an indicator.

Local variables allow you to write to a control. Thus, they can be used to initialize the value of a control. Local variables usually should be avoided as a matter of good LabVIEW style, but this is one occasion where the use of local variables is acceptable. There are a few drawbacks to consider, though. Local variables don't have error terminals and so are more difficult to use when you need to enforce execution order through the use of data flow. For example, in the layout of the state machine

shown in Figure 8.9, we have the STOP button reinitialized in the input area of the state machine using an invoke node. Notice that the error wire from the invoke node is connected to the error shift register. This does two things: It initializes the shift register, and it ensures that the STOP button is reinitialized before the loop starts. Placing a naked local variable of the STOP button in the input area would not ensure that the button would be reinitialized before the loop is executed, because the local variable would operate in parallel with the loop. There would be no dataflow relation that would force the local variable to execute first.

Another issue with local variables is that Boolean controls with a latch mechanical action are incompatible with local variables. If you create a local variable of a latch mechanical action Boolean, the run arrow of the VI will break.

The *Value* property node option avoids some of the difficulties associated with local variables. Property nodes have error terminals, so it is easy to enforce data flow when using property nodes. Boolean controls with latch mechanical action still cause problems, though. The *Value* property node can be used on a latched Boolean control, but the data type of the node will be a variant rather than a Boolean. If you wire a Boolean constant to such a property node, a coercion dot will be placed at the input terminal of the node.

The *Reinitialize to Default* invoke node approach avoids these difficulties. Invoke nodes have error terminals, so they are suitable for enforcing data flow and can be used on latched Boolean controls. The *Reinitialize to Default* invoke node is also compact, since it has no inputs to wire. The drawback to the invoke node approach is that it only can be used to restore controls to their default value. If you need to write a non-default value to a control, this approach will not work.

Style Guide: Be careful if you find yourself trying to reverse the normal data flow by writing to a control or reading from an indicator. This could be a warning sign that you are not using "data flow thinking."

Initializing indicators is a much simpler matter. To initialize an indicator, you simply need to write the desired value to the indicator. This follows the normal direction of data flow. In Figure 8.10, the car wash indicators are initialized by wiring them directly to the cluster constant of default indicator values in the input section of the state machine.

This leaves the purchase selector. Every other front panel control and indicator has been initialized by this point in the program. The purchase selector offers an extra challenge, though, since the functional requirements specify that the purchase selector be locked during an active car wash. Initializing the purchase selector involves two steps: unlocking the selector and setting its initial value. These two steps need to be taken whenever a car wash cycle ends too, so the *Unlock Selector* state needs to be developed to handle these two steps. We can use this state to initialize the purchase selector by simply transitioning from the *Initialize* state directly to the *Unlock Selector* state.

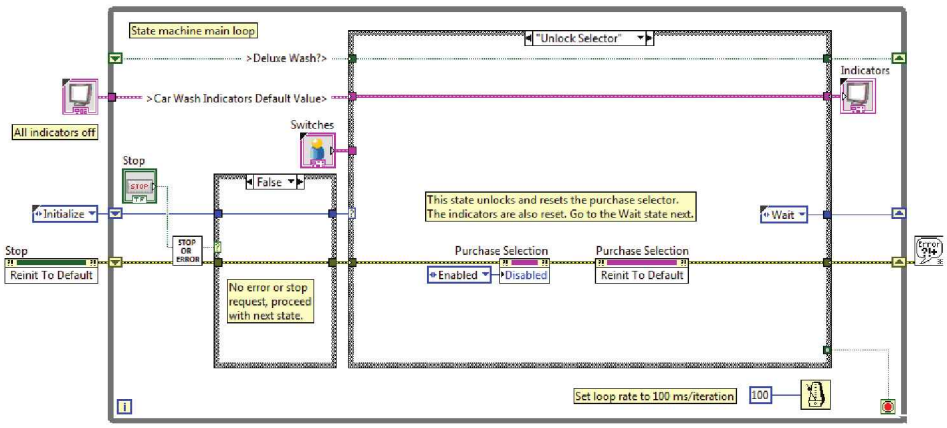


Figure 8.12: The *Unlock Selector* state of the classic state machine car wash controller

8.9 Develop the *Unlock Selector* State

The *Unlock Selector* state is shown in Figure 8.12. This state uses the *Disable* property node to unlock the selector. The *Disable* property node has three possible inputs: *Enabled*, *Disabled*, and *Disabled and Grayed Out*. When a control is enabled, it is fully visible and accepts user input. When it is disabled, it is fully visible but does not accept user input. When it is disabled and grayed out, it is only partially visible and does not accept user input. As we will see, the *Lock Selector* state uses the *Disabled and Grayed Out* input of the *Disable* property node to lock the selector. This was chosen because it satisfies the functional specification and gives a visual cue to the user that the control is inoperative. To unlock the selector, the *Unlock Selector* state selects the *Enabled* input of the *Disable* property node. After the selector is unlocked, it is reinitialized using the *Reinitialize to Default* invoke node. The *Wait* state is then placed on the shift register to establish the transition to the next state.

8.10 Develop the *Wait* State

The *Wait* state of the car wash controller is shown in Figure 8.13. The *Wait* state of a classic state machine is typically awkward to implement. The classic state machine must poll all relevant front panel objects to capture user input, and this can be difficult if the number of controls is large. Polling is also undesirable because it is an inefficient use of computer resources. Even if the user is idle, the program is not. If there is no user input, the *Wait* state places the *Wait* state back on the state shift register, cycling over and over again until the user finally makes a selection.

Luckily, the car wash controller only needs to poll the purchase selector in the *Wait* state (the STOP button is polled in the lobby of the state machine). A **`Select`** function is used to determine which state will be put on the shift register. If either the

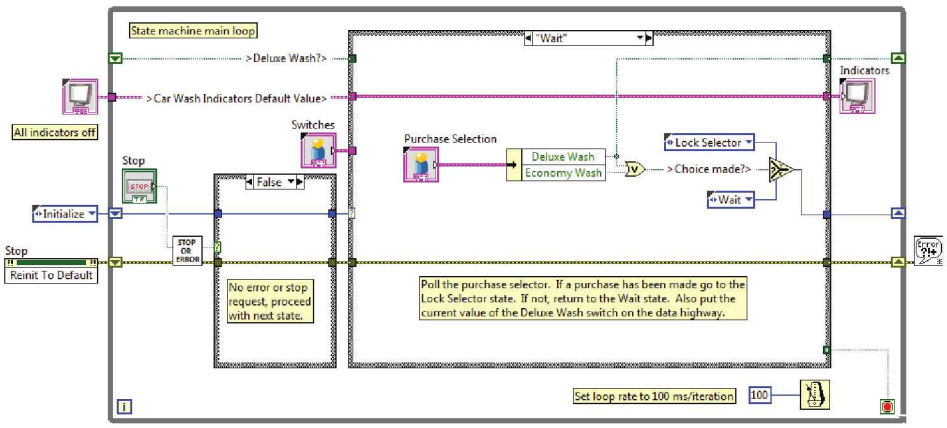


Figure 8.13: The *Wait* state of the classic state machine car wash controller

DELUXE WASH or the ECONOMY WASH controls are True, the state machine transitions to the *Lock Selector* state. Otherwise, the state machine goes back to the *Wait* state.

The *Wait* state also places the value of the DELUXE WASH control on the data highway. This information is used by other states to determine their state transitions.

8.11 Develop the *Lock Selector* State

After a Deluxe or Economy wash has been selected, the first state to execute is the *Lock Selector* state, shown in Figure 8.14. This state uses the *Disable* property node to disable and grey out the purchase selector. This satisfies the functional requirement that the purchase selector not accept user input while a car wash is active.

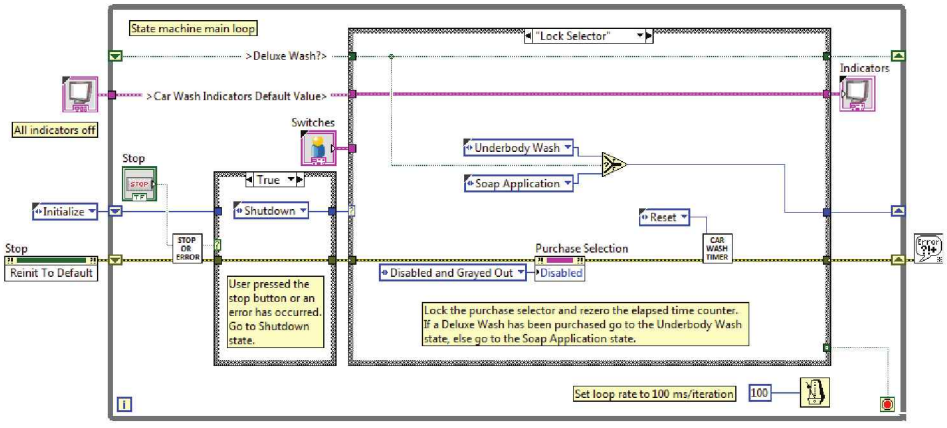


Figure 8.14: The *Lock Selector* state of the classic state machine car wash controller

An interesting feature of the *Lock Selector* state is that it needs to know which wash type was selected in order to determine which state to transition to next. If the Deluxe wash was selected, the next state is the *Underbody Wash* state. If the Economy wash was selected, the next state is the *Soap Application* state. A **`Select`** function is used to implement the decision. The value of the DELUXE WASH Boolean is pulled from the data highway and is used as an input to the **`Select`** function. This state also needs to reset the car wash proximity timer to zero. Since all of the car wash states share the same action engine timing sub-VI, the timer must be reset to zero when transitioning from one car wash cycle state to the next.

8.12 Develop the Car Wash States

Now we move on to the main portion of the car wash controller, which is the car wash states themselves. These states all have the same basic structure. They must perform the following tasks:

1. Select the correct proximity switch and pass it to the timing sub-VI.
2. Execute the timing sub-VI to determine if the target time has elapsed.
3. Light the proper indicators to indicate if the cycle is running or if the vehicle is out of position.
4. Determine which state to transition to next.

The *Underbody Wash* state is shown in Figure 8.15. The heart of this state is the car wash timer, described in detail in Chapter 6. The car wash timer keeps track of the elapsed time in a given car wash cycle, pausing the timer when the vehicle moves off of the relevant proximity switch.

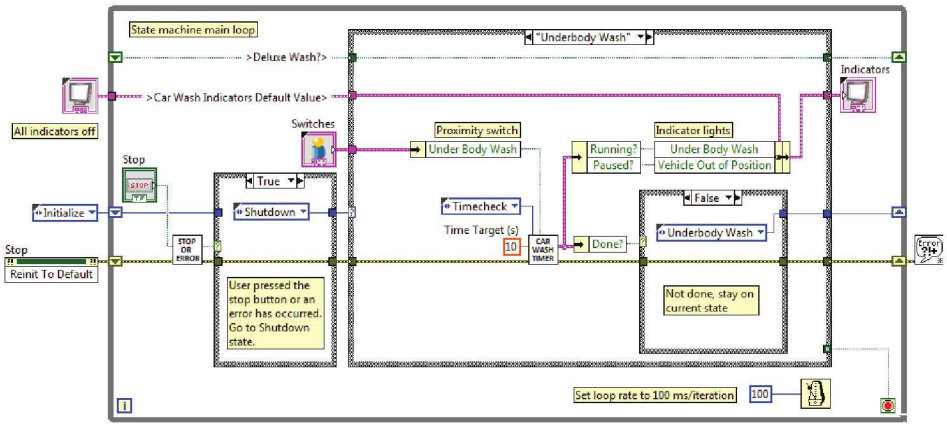


Figure 8.15: The *Underbody Wash* state of the classic state machine car wash controller with the state not done

The car wash timer requires two inputs: a target time and a proximity switch. The *Underbody Wash* state hardwires a target time to the car wash timer. It also uses the **Unbundle By Name** function to extract the relevant proximity switch from the simulation switch cluster and passes the value of that switch to the car wash timer.

Style Guide: Always use the **Unbundle By Name** and **Bundle By Name** functions instead of **Unbundle** and **Bundle**. The "By Name" versions of these functions are much more readable, and they give you the ability to access the specific cluster elements that you are interested in instead of the entire cluster.

The car wash timer has a single output, which is a status cluster containing the *Running?*, *Paused?* and *Done?* Booleans. Once again, the **Unbundle By Name** function is used to access these Booleans. Two separate **Unbundle By Name** functions are used. The *Running?* and *Paused?* Booleans are both used to set the car wash indicators, so they are grouped together for readability. The *Done?* Boolean is used as an input to the case structure that decides the next state. Since this Boolean is used in a different portion of the block diagram, a second **Unbundle By Name** function is used to minimize the distance between the **Unbundle By Name** function and the case structure.

The **Bundle By Name** function is used to set the proper indicators. The *Running?* Boolean output is assigned to the UNDERBODY WASH indicator on the front panel, and the *Paused?* output is assigned to the VEHICLE OUT OF POSITION indicator on the front panel.

The indicator default cluster is used as the central input to the **Bundle By Name** function. Only the bundled elements of the cluster are overwritten. This ensures that the any indicator not affected by the *Underbody Wash* state remains at its default value.

The *Done?* Boolean output is sent to the selector of a case structure. If the cycle is not done, the *Underbody Wash* state is once again placed on the shift register. If the *Underbody Wash* state is done, the *Soap Application* state is put on the shift register and the timer is reset, as shown in Figure 8.16. The *Underbody Wash* cycle is executed again and again until the target time has elapsed, and then the state machine will move on to the *Soap Application* state with the timer reset to zero.

Notice how simple this state is. The main action of this state is to simply feed the correct inputs to the car wash timer and send the outputs of the timer to the correct indicators. This shows the power of a well-designed sub-VI. The action engine allows the reader of the top-level VI to focus on the dataflow of the main program and ignore the details of the timer implementation.

The *Soap Application*, *Main Wash*, and *Air Dry* states have the same structure as the *Underbody Wash* state and are shown in Figures 8.17, 8.18, and 8.19, respectively. The only differences are in which proximity switch is fed to the car wash timer, which cycle indicator is connected to the *Running?* output of the car wash timer, and which states are placed on the state constant shift register. In each of these figures, the True case of the *Done?* case structure is shown to illustrate the transition to the next car

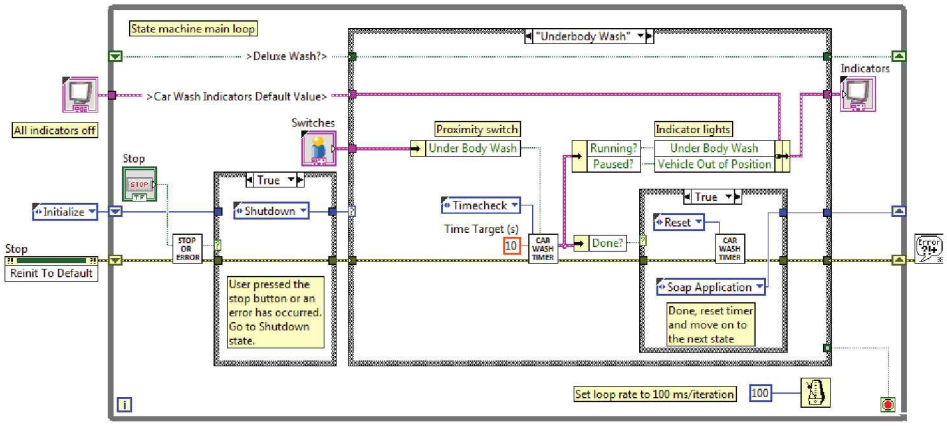


Figure 8.16: *Underbody Wash* state of the classic state machine car wash controller, state done

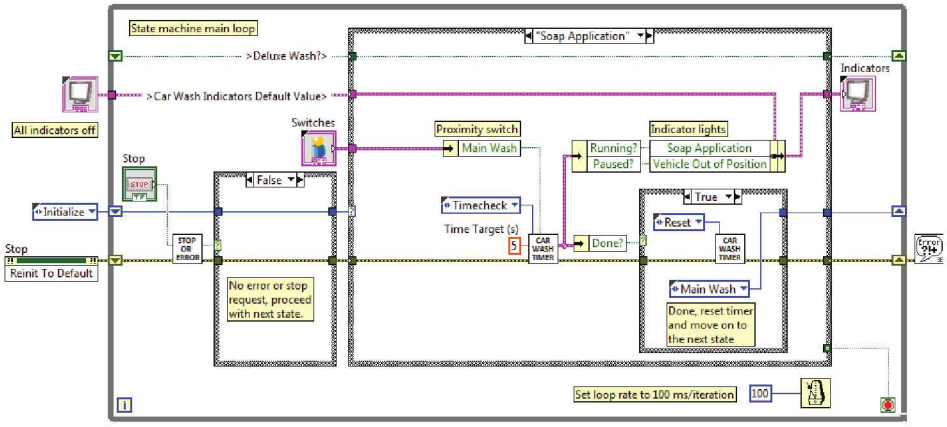


Figure 8.17: The *Soap Application* state of the classic state machine car wash controller

wash cycle. The False case of the *Done?* case structure simply places the current state back on the state shift register.

The *Rinse* state shown in Figure 8.20 needs to know whether a Deluxe or Economy Wash cycle has been selected in order to determine the next state transition when the *Rinse* cycle is completed. Once again, a **`Select`** function is used to make the decision. If a Deluxe Wash has been selected, the *Rinse* cycle transitions to the *Air Dry* state; otherwise, it transitions to the *Unlock Selector* state.

The *Unlock Selector* state shown in Figure 8.12 unlocks the purchase selector, transitions back to the *Wait* state, and the process begins again. The state machine continues to cycle until the guard clause detects an error or the STOP button is pressed, sending the state machine to the *Shutdown* state.

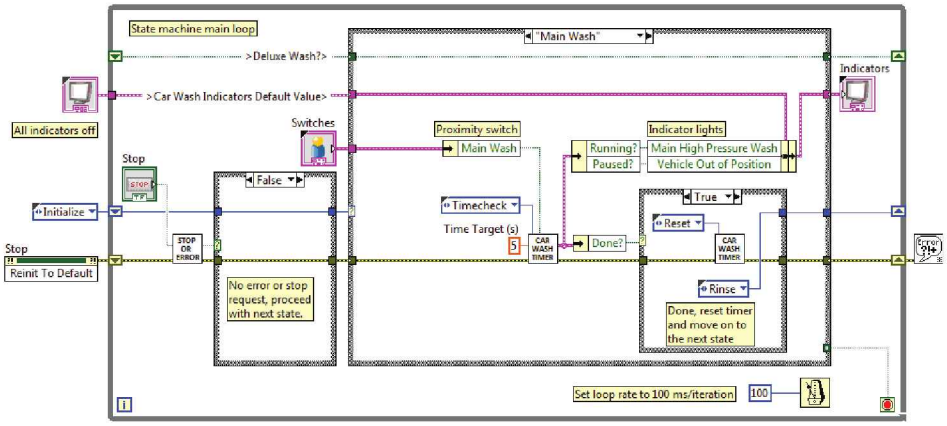


Figure 8.18: The *Main Wash* state of the classic state machine car wash controller

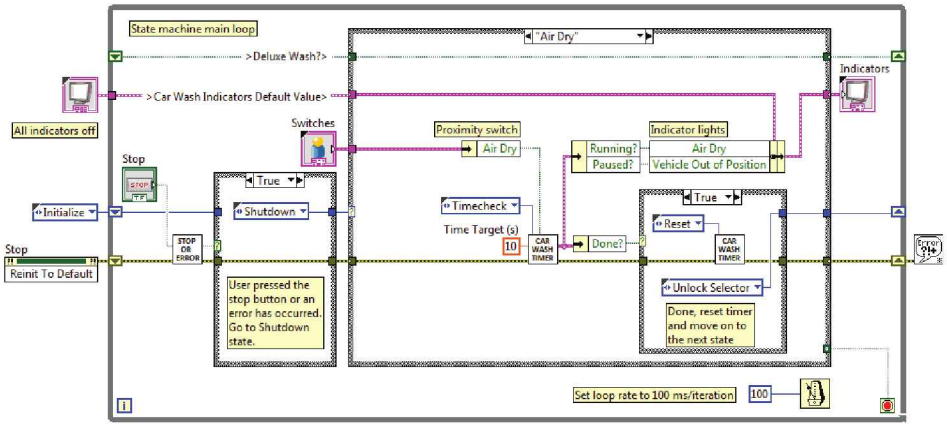


Figure 8.19: The *Air Dry* state of the classic state machine car wash controller

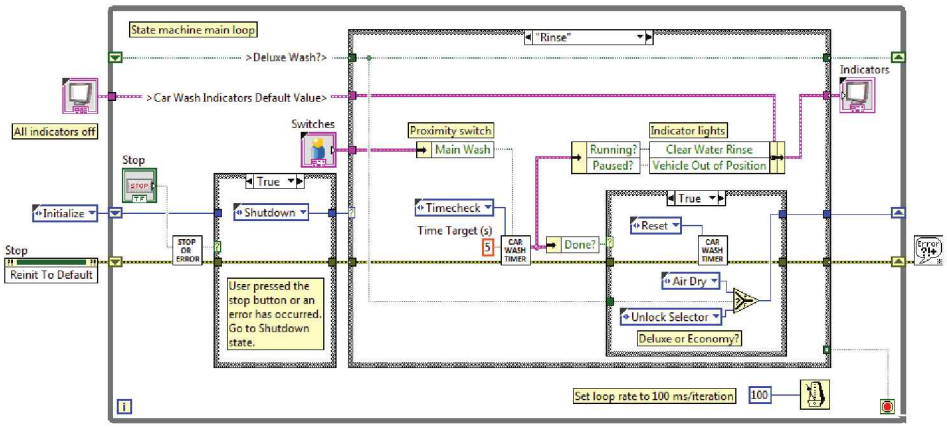


Figure 8.20: The *Rinse* state of the classic state machine car wash controller

8.13 Develop the *Shutdown* State

Chapter 4 discussed some common pitfalls in creating a shutdown state. This state can be triggered by an error, so it must be constructed with the assumption that an error may be present on the main error wire.

Figure 8.21 shows an implementation of the *Shutdown* state for the car wash controller. This state does a number of things:

- The purchase selector is enabled using the *Disable* property node.
- The purchase selector is reinitialized using the *Reinitialize to Default* invoke node.
- The proximity switches are reinitialized using the *Reinitialize to Default* invoke node.
- The indicators are set to their default value.
- A Boolean True constant is sent to the loop conditional terminal to terminate the main loop.

The property node and invoke nodes are on error wires parallel with the main error wire. This was done because property nodes and invoke nodes will not execute if an error is present at the error-input terminals of the nodes. Placing them on parallel error wires ensures that they will execute even if the *Shutdown* state is triggered by an error. In order to keep the error chain tight, the two error wires are merged into the main error wire before exiting the *Shutdown* state. The main error wire of the state machine is connected to the top terminal of the **`Merge Errors`** function to give it priority over the other error wires.

Notice that the *Shutdown* state has been wired to the state constant shift register in the *Shutdown* state itself in Figure 8.21. This second *Shutdown* state will not be executed, since the loop will terminate before it can be acted upon. But remember, the state constant output tunnel must be wired in every case of the case structure. Something must be wired to the state output tunnel, even in the *Shutdown* state. Any state could have been selected, but using the *Shutdown* state is common practice.

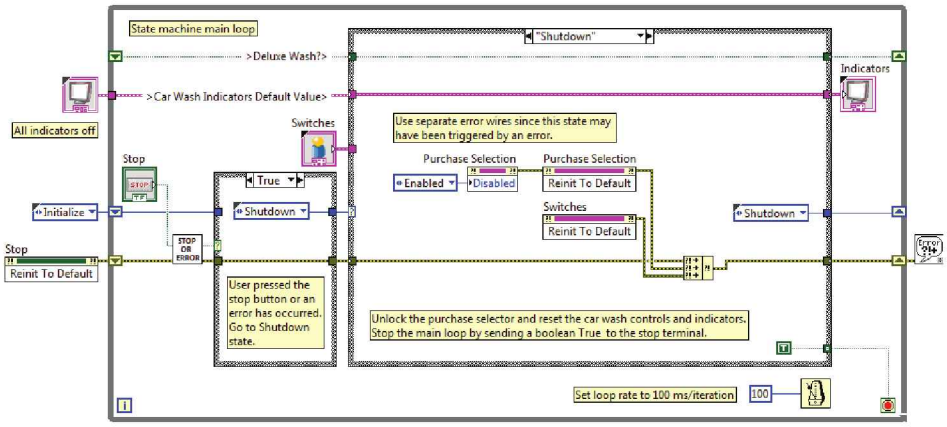
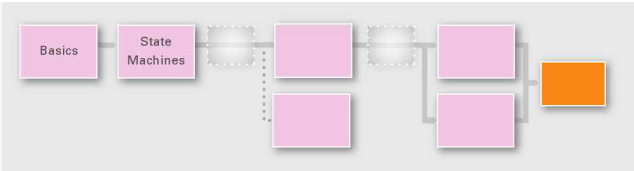


Figure 8.21: The *Shutdown* state of the classic state machine car wash controller





Problem Set 3

9.1 Traffic Light Controller Exercises

Exercise 3.1: Traffic Light Classic State Machine

Reread the functional specification of the traffic light controller given in Appendix A. Design and implement a classic state machine VI that satisfies the requirements of the functional specification. Review the discussion about state machine structure from Chapter 4. Where should the vehicle sensors be placed? Where should the traffic light indicators be placed? Where should the STOP button be placed? Does the traffic light controller need data stored persistently on a data highway?

Use the all-inclusive cluster of traffic light clusters developed in Exercise 1.5 and the **`Bundle By Name`** function to light the correct traffic lights each loop iteration. Use the `Bundle Sensors` sub-VI developed in Exercise 1.5 to streamline the data transfer from the vehicle sensors to the various traffic light states. Use the `Auto Reset Timer` developed in Exercise 2.9 to time the individual traffic light states. Use the `Next Traffic State` sub-VI developed in Exercise 1.6 to determine the state transitions. ■

9.2 Security System Controller Exercises

The requirements found in the security system functional specification are all straightforward and easily can be met with a classic state machine architecture. There is one requirement, though, which is a bit tricky. The functional specification calls for the zone status to be indicated by color-changing Boolean indicators. As we will see, this can be done in LabVIEW, but it requires detailed knowledge of property nodes and explicit control references. These topics are not directly related to constructing a classic state machine, so we take the following approach. First, develop a solution for a simplified form of the functional specification that does not involve color-changing

Boolean controls. You then test the scalability of your solution to ensure that it meets the functional specification. You then are given sample code for manipulating the color of Boolean indicators using explicit references and property nodes and be construct a test program to demonstrate your mastery of the concepts involved. Finally, you add the color-changing Boolean indicators to your simplified solution to meet the requirements of the original functional specification. You then test the scalability of your revised solution.

Exercise 3.2: Simplified Security System Classic State Machine

Reread the functional specification of the security controller given in Appendix B. Design and implement a classic state machine that satisfies the requirements of the functional specification with the following exception. Instead of indicating the zone statuses with color-changing Booleans, display the zone statuses as an array of enumerated indicators, as seen in Figure 9.1. The enumerated indicators shall display one of four values corresponding to the zone status: *Normal*, *Alarm*, *Bypass*, or *Tamper*.

All other requirements of the functional specification remain in force. The specification is a bit vague when describing the specifications of the log file. For the purposes of these exercises, assume that the log file does not need a header. The log file is formatted as a comma-separated value (csv) ASCII text file. If no log file exists when the program starts, it should be created. If a log file exists when the program starts, this file should be used without overwriting the existing data.

Preallocate the zone status array using the `Initialize Status Array From Cluster` sub-VI developed in Exercise 1.10, wiring a local variable of one of the switches clusters to the cluster input. Keep scalability in mind throughout this exercise. The program should not assume a particular number of zones. Ideally, you should be able to increase the size of the switches clusters and the zone indicator array without having to modify the block diagram of the code.

Where should the TAMPER, BYPASS, and ALARM switch cluster be placed? Where should the zone indicator array be placed? Where should the STOP button be placed? Does the security system controller need data stored persistently on a data highway?

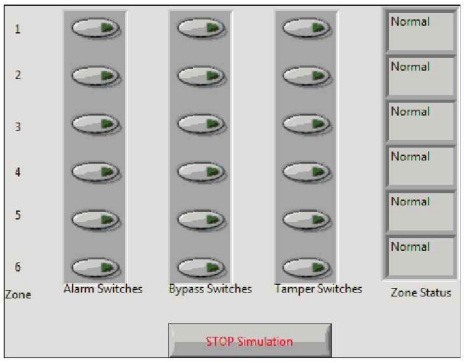


Figure 9.1: The front panel of the simplified security system functional specification

You may find the results of previous exercises helpful: the `Reset Zone Array` sub-VI developed in Exercise 1.11, the `Get Status Of All Zones` sub-VI developed in Exercise 1.13, and the `Update Log` sub-VI developed in Exercise 2.12. ■

Exercise 3.3: Scalability of the Simplified Security System

Test the scalability of the solution developed in Exercise 3.2. Open the type definition of the switches cluster, and add a seventh zone. Save and close the type-definition file. Add a seventh displayed element to the zone status indicator array as well. Is the *Run* arrow of your VI still unbroken? If it is, run the code and confirm that it works as expected. Change the status of some of the zones, including the new seventh zone. Examine the log file and find the entry or entries generated by changing the status of the seventh zone. Is the log file entry correct? Once you have demonstrated the scalability of your code, think about the structure of the main application and the sub-VIs. What were the primary features of these vis that ensured the scalability of the main application? ■

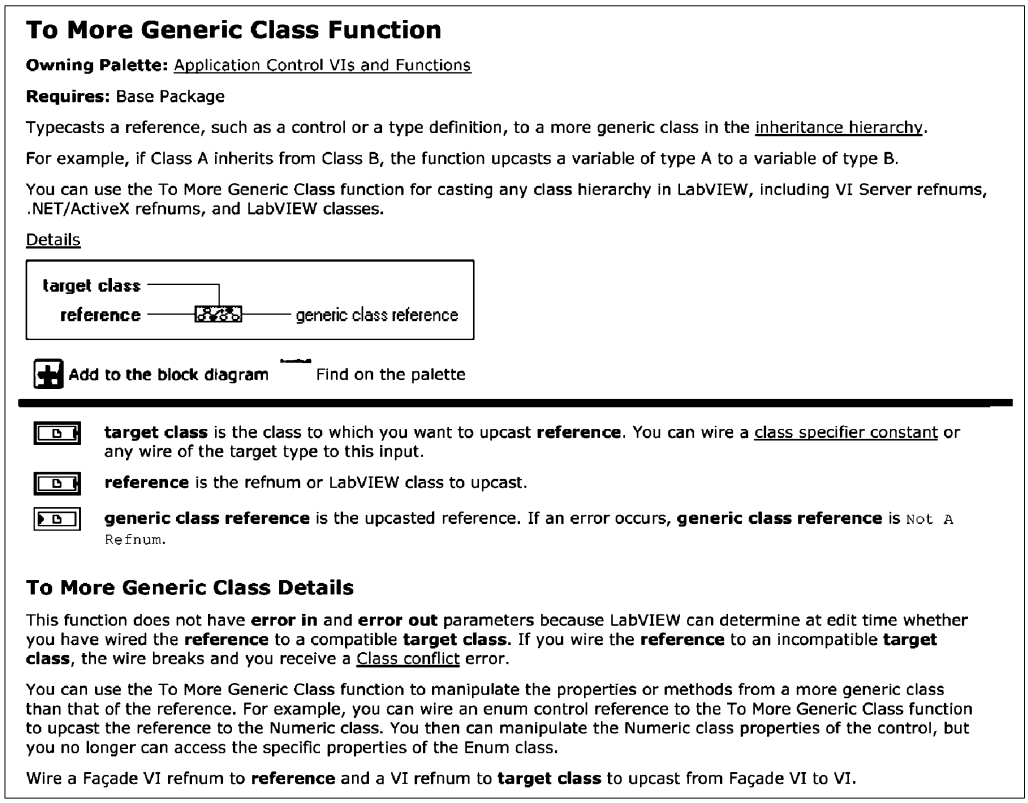
Exercise 3.4: On the Use of Local Variables

The style guidelines listed at the end of Chapter 1 suggest we should minimize the use of local variables. In Exercise 3.2, you were asked to use a local variable of the switches cluster to preallocate the zone status array. Is this an acceptable use of local variables? Why or why not? ■

As discussed in Chapter 1, explicitly linked references can be used to manipulate front panel objects from a sub-VI. Control references have class hierarchy, and the class of the reference determines its data type. For example, a reference to a Boolean control has the same data type as a reference to any other Boolean control, but it has a different data type than a reference to a numerical control. They both belong to the family of references to controls.

References of controls can be used to access properties and methods that are applicable to all controls. For example, any control can be made invisible or can be reinitialized to default. More specific references can be used to access more specific properties and methods. For example, the Boolean text of a Boolean control only can be accessed with a Boolean control reference. A specific reference can be used to access the more general properties higher up in the family tree by casting the reference to a more general class using the **`To More Generic Class`** (see the help file in Figure 9.2). The generic reference can be used to access the more specific properties further down the family tree by using the **`To More Specific Class`** (see the help file in Figure 9.3).

It is generally good programming practice to cast a reference as high up the family tree as possible, making it as general as possible. This practice has two advantages. First, a general reference is more likely to be accepted by a sub-VI with a control refnum than a more specific reference. Second, references to controls with different data types can be grouped into an array if the array element is a general reference higher up the



To More Generic Class Function

Owning Palette: Application Control VIs and Functions

Requires: Base Package

Typecasts a reference, such as a control or a type definition, to a more generic class in the inheritance hierarchy.

For example, if Class A inherits from Class B, the function upcasts a variable of type A to a variable of type B.

You can use the To More Generic Class function for casting any class hierarchy in LabVIEW, including VI Server refnums, .NET/ActiveX refnums, and LabVIEW classes.

Details

target class
reference
generic class reference

Add to the block diagram Find on the palette

target class is the class to which you want to upcast **reference**. You can wire a class specifier constant or any wire of the target type to this input.

reference is the refnum or LabVIEW class to upcast.

generic class reference is the upcasted reference. If an error occurs, **generic class reference** is `Not A Refnum`.

To More Generic Class Details

This function does not have **error in** and **error out** parameters because LabVIEW can determine at edit time whether you have wired the **reference** to a compatible **target class**. If you wire the **reference** to an incompatible **target class**, the wire breaks and you receive a Class conflict error.

You can use the To More Generic Class function to manipulate the properties or methods from a more generic class than that of the reference. For example, you can wire an enum control reference to the To More Generic Class function to upcast the reference to the Numeric class. You then can manipulate the Numeric class properties of the control, but you no longer can access the specific properties of the Enum class.

Wire a Façade VI refnum to **reference** and a VI refnum to **target class** to upcast from Façade VI to VI.

Figure 9.2: Help file for the `To More Generic Class` function

family tree that is shared by all of the controls. To demonstrate this, examine the `Make Controls Invisible` sub-VI with the icon shown in Figure 9.4 and block diagram shown in Figure 9.5. This program accepts an array of generic control refnums as an input as well as a VISIBLE Boolean control. The array of references are passed to the auto-indexing input of a **`For`** loop, where they are then passed to an explicitly linked property node configured to write the *Visible* property. When this sub-VI runs, it will make the referenced controls on the front panel of the calling VI visible or invisible, depending on the value of the Boolean input. The control reference on the front panel of this sub-VI is simply an array control refnums. This is constructed by first dropping an empty array shell on the front panel and then placing a control refnum from the *Refnum* sub-palette within the shell. This array will accept references from any control, although a coercion dot will appear if references from specific controls further down the family tree are wired to its terminal.

It is generally considered good programming style to avoid coercion dots whenever possible. This can be done by casting the specific reference of a control further up the family tree to become that of a generic control using the **`To More Generic Class`** function. This is demonstrated by the example program with the front panel

To More Specific Class Function

Owning Palette: Application Control VIs and Functions

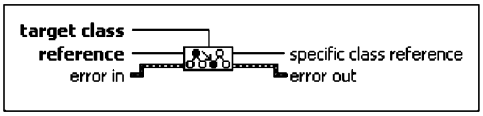
Requires: Base Package

Typecasts a reference, such as a control or a type definition, to a more specific class in the inheritance hierarchy.

For example, if Class A inherits from Class B, a variable of type B can hold a value of type A. You can use the function to downcast from type B to type A. If the typecast is invalid, this function returns an error at run time.

You can use the To More Specific Class function for casting any class hierarchy in LabVIEW, including VI Server refnums, .NET/ActiveX refnums, and LabVIEW classes.

Details Example



Add to the block diagram Find on the palette

target class is the class to which you want to downcast **reference**. You can wire a class specifier constant or any wire of the target type to this input.

reference is the refnum or LabVIEW class to downcast.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

specific class reference is the downcast reference. If the **target class** is incompatible with the **reference**, the function returns an error and **specific class reference** is `Not A Refnum`.

error out contains error information. This output provides standard error out functionality.

To More Specific Class Details

You can use the To More Specific Class function to manipulate the properties or methods from a more specific class than that of the reference. For example, if you build an array of references to boolean, numeric, and string controls, LabVIEW casts these references to the more generic Control class because an array can contain only one data type, and each of these controls is a member of the Control class. If you then want to manipulate the properties for only the boolean controls in the array, you can create a reference to the array and use the To More Specific Class function to downcast the array reference to the Boolean class.

Wire a VI refnum to **reference** and a Façade VI refnum to **target class** to downcast from VI to Façade VI.

You also can use data value references to downcast classes. The **To More Specific Class** function waits to execute until the reference is available for downcast. To downcast a class, wire the data value reference of the target class to **target class**.

Figure 9.3: Help file for the `To More Specific Class` function

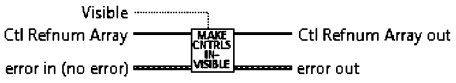


Figure 9.4: The icon of the `Make Controls Invisible` sub-VI

shown in Figure 9.6 and block diagram shown in Figure 9.7. The front panel contains five controls and indicators. When the Boolean switch is toggled from *Visible* to *Invisible* and back again, the other four controls and indicators vanish and reappear. The block diagram shows how this is done. Control references to the four disappearing controls

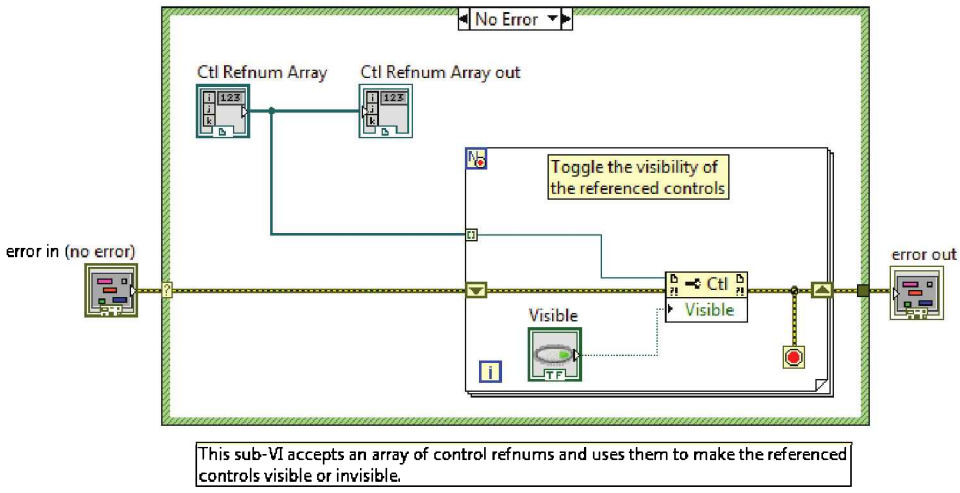


Figure 9.5: The block diagram of the `Make Controls Invisible` sub-VI

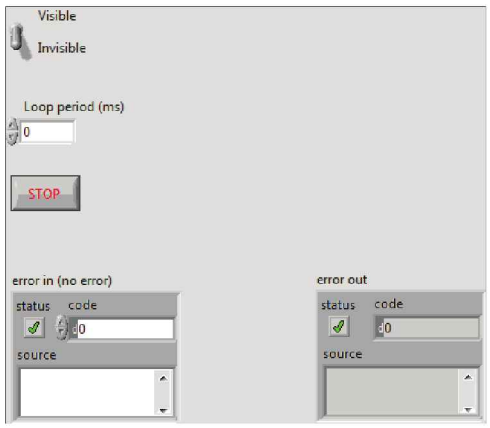


Figure 9.6: The front panel of the `Control Reference Example` VI

and indicators are created and cast to generic control references using the `To More Generic Class` function. The class specifier constant shown on the block diagram are found on the *Programming...Application Control* sub-palette. Click on the constant to navigate through the class hierarchy, and select the desired class. For a generic control, the path is *Generic...GObject...Control...Control*. The resulting generic control references are built into an array using the **`Build Array`** function with the help file shown in Figure 9.8. Since the references are all generic control references, there are no coercion dots at the terminals of the **`Build Array`** function. The array is then passed

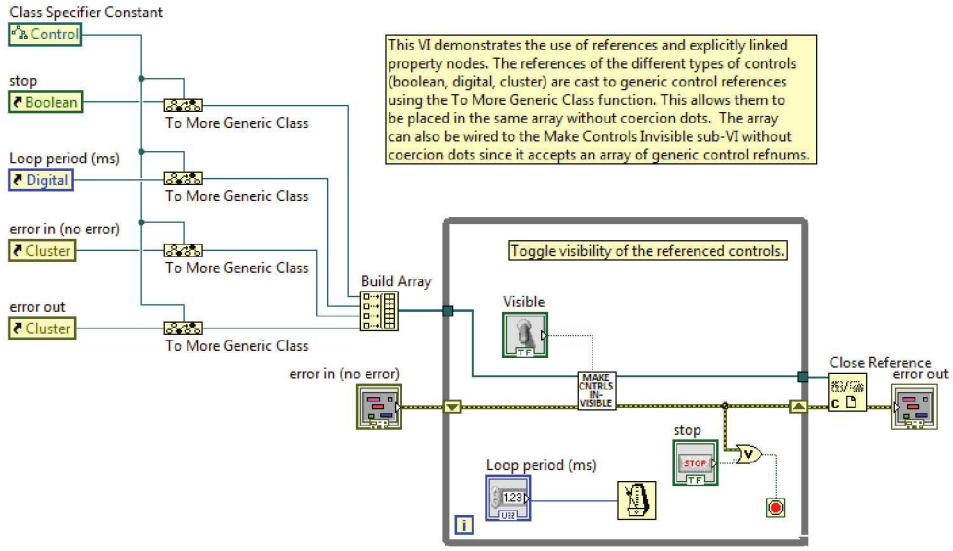


Figure 9.7: The block diagram of the `Control Reference Example` VI

to `Make Controls Invisible` sub-VI to toggle the visibility of the referenced controls. This sub-VI expects an array of generic control references, so again there is no coercion dot at the input terminal. When the program ends, the array of references is passed to the **`Close Reference`** function. The help file in Figure 9.9 reveals two interesting features of this function. First, it will accept either a single reference or a one-dimensional array of references as an input. Second, the function will operate even if an error is present at the error input terminal. Both of these facts are taken advantage of in Figure 9.7.

Exercise 3.5: Preallocating the Zone Status Array using a Cluster Reference

Revise the `Initialize Status Array From Cluster` developed in Exercise 1.10 to accept a reference to a cluster instead of the cluster itself. The icon of the revised `Initialize Status Array From Cluster Reference` sub-VI is shown in Figure 9.10. Use the cluster reference to generate an array of *Normal* zone status enumerated constants with the same number of elements as the referenced cluster. Revise your simplified security system to use a cluster reference to preallocate the zone status array instead of a local variable. The sub-VI should close the cluster reference before it exits.

To do this, take advantage of the *Controls[]* property of cluster references. The output of this property node is an array of control refnums to the controls in the referenced cluster. The size of this array is equal to the number of controls in the cluster. In order to access this property node though, the control refnum in the sub-VI must be cast to that of a cluster control. To do this, right-click on the

Build Array Function

Owning Palette: Array Functions

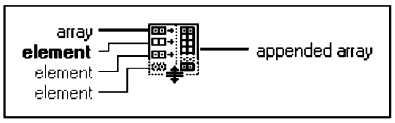
Requires: Base Package

Concatenates multiple arrays or appends elements to an n-dimensional array.

You also can use the Replace Array Subset function to modify an existing array.

The connector pane displays the default data types for this polymorphic function.

Details Example



Add to the block diagram Find on the palette

array or **element** can be any n-dimensional array or scalar element. All inputs must be either elements and 1D arrays or n-dimensional and (n-1)-dimensional arrays. All inputs must have the same base type.

appended array is the resulting array.

Build Array Details

When you place the function on the block diagram, it has only one input available. Add inputs to the node by right-clicking an input and selecting **Add Input** from the shortcut menu or by resizing the node.

If you wire control references of different classes to this function, this function typecasts the references to a more generic class that is the lowest common class in the inheritance hierarchy. This function then returns **appended array** with that class.

The Build Array function operates in one of two modes depending on whether you select **Concatenate Inputs** from the shortcut menu. If you select **Concatenate Inputs**, the function appends all inputs in order, forming an output array of the same dimensionality as the array input wired.

If you do not select **Concatenate Inputs**, the function builds an output array of one dimension higher than the dimension of the inputs. For example, if you wire 1D array input to a Build Array function, the output is a 2D array, even if the 1D array is empty. The inputs must all be the same dimensionality. The function appends each input in order, forming a subarray, element, row, or page, of the output array. Inputs are padded, as necessary, to match the size of the longest input.

For example, if you wire two 1D arrays, {1, 2} and {3, 4, 5} to Build Array, and select **Concatenate Inputs** from the shortcut menu, the output is the 1D array {1, 2, 3, 4, 5}. If you wire the same two arrays to Build Array, and do not select **Concatenate Inputs** from the shortcut menu, the output is a 2D array containing {{1, 2, 0}, {3, 4, 5}}, where the first input is padded to match the length of the second input.

If the inputs are all arrays of the same dimensionality, right-click the function to remove the checkmark next to or select the **Concatenate Inputs** shortcut menu item. If the inputs do not have the same dimensionality, **Concatenate Inputs** is automatically selected and cannot be deselected. If all the inputs are scalar elements, **Concatenate Inputs** is automatically deselected and cannot be selected, and the output is a 1D array containing the elements in order.

When you select **Concatenate Inputs** from the shortcut menu, the glyphs in the Build Array icon change to distinguish between the two input types. Inputs with the same dimensionality as the output show an array glyph, while inputs with a dimensionality that is one less than the output show an element glyph.

Figure 9.8: Help file for the `Build Array` function

control refnum, select *Select VI Server Class*, and navigate through the class hierarchy: *Generic... GObject... Control... Cluster... Cluster*. Close all references before exiting the sub-VI. Since the control refnum in the sub-VI will expect a generic cluster reference, cast the switches cluster reference to a generic cluster in the main application before wiring it to your revised preallocation sub-VI.

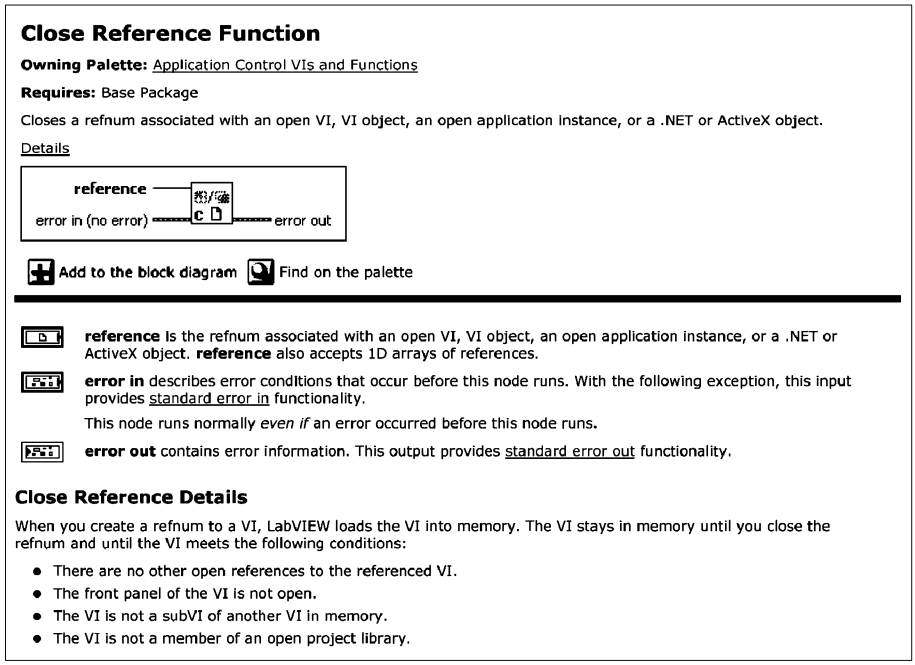


Figure 9.9: Help file for the `Close Reference` function



Figure 9.10: The icon for the `Initialize Status Array From Cluster Reference` sub-VI

Test the scalability of your revised security system controller as you did in Exercise 3.3. What features of your `Initialize Status Array From Cluster Reference` sub-VI maintained the scalabilty of the main application? ■

The security system controller functional specification in Appendix B requires that the color of front panel Boolean indicators be changed to represent the current zone status. Green is normal, red is an alarm state, blue represents a bypass state, and orange represents a tamper state. It is desirable to assign the color-changing task to a `Set Zone Colors` sub-VI to improve the readability of the main application. This can be done using explicitly linked property nodes and a reference to the ZONE STATUS INDICATORS cluster seen in Figure B.1. Since this task requires detailed knowledge of the properties of Boolean controls, the color-changing sub-VI is developed for you here, and you will implement it in the security system in the exercises.

It is important to realize that the zone status indicators in the main application are not actually being used as Boolean indicators. They will not be used to display

True or False values. The idea here is to simply change the color of the indicator in its default (False) state to indicate the zone status. The values of the Boolean indicators are always False, so they will remain unwired on the block diagram of the main applicaiton. However, even though they are unwired, their color can be changed through the use of references and explicitly linked property nodes.

The task of changing the color of the Boolean indicators in the ZONE STATUS INDICATORS cluster is broken down into two sub-tasks. First, a `Get Lamp Refnums` sub-VI is developed that accepts a generic cluster reference to the ZONE STATUS INDICATORS cluster and produces an array of Boolean control refnums referencing the Boolean zone status indicators on the front panel of the main application. Second, the `Set Zone Colors` sub-VI is developed that accepts this cluster of Boolean control refnums and a zone status array as inputs and changes the color of the referenced indicators accordingly.

Exercise 3.6: Generating an Array of Boolean Control References from a Cluster Reference

Develop the `Get Lamp Refnums` sub-VI with the icon shown in Figure 9.11. This sub-VI accepts a generic cluster reference to the ZONE STATUS INDICATORS cluster and produces an array of references to the individual Boolean indicators within the referenced cluster. Use the *Controls[]* property node to obtain an array of generic control references to the individual Boolean indicators. Then cast those references to the Boolean control class using the **`To More Specific Class`** function with the help file shown in Figure 9.3. This last step is important, because a generic control reference cannot be used to access the property node needed to change the color of a Boolean indicator. A Boolean control reference is needed for that. Make sure to close the generic cluster reference before exiting the sub-VI.



Figure 9.11: The icon of the `Get Lamp Refnums` sub-VI ■

Exercise 3.7: Preallocating the Zone Status Array Using an Array of Control References

Revise the `Initialize Status Array From Cluster` developed in Exercise 1.10 to accept as an input the array of Boolean control references generated by the `Get Lamp Refnums` developed in Exercise 3.6. The sub-VI should use this array to generate a preallocated default array of zone status enumerated constants that is the same size as the input array. The icon of the revised `Initialize Status Array From Boolean Reference Array` sub-VI is shown in Figure 9.12.



Figure 9.12: The icon for the `Initialize Status Array From Boolean Reference Array` sub-VI ■



Figure 9.13: The icon of the `Set Zone Colors` sub-VI

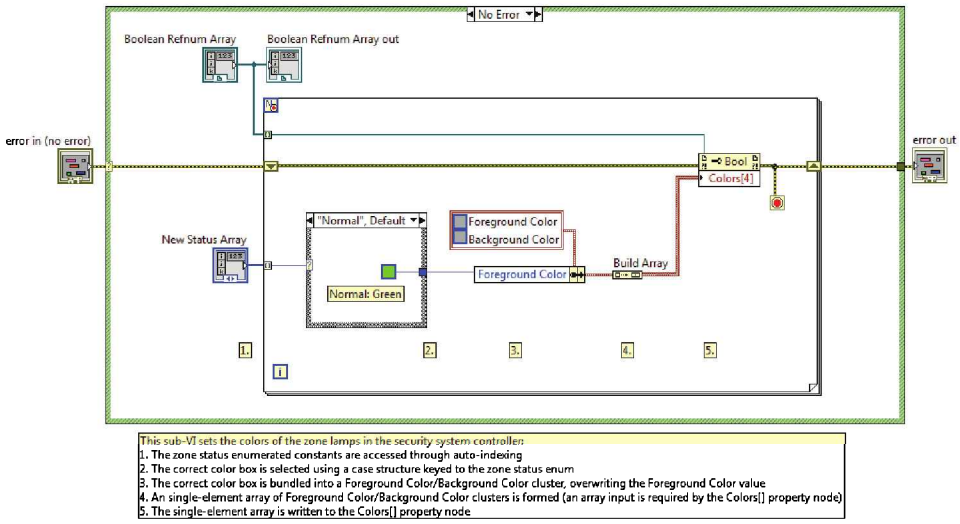


Figure 9.14: The block diagram of the `Set Zone Colors` sub-VI

The output of the `Get Lamp Refnums` sub-VI developed in Exercise 3.6 will now be passed to the `Set Zone Colors` sub-VI with the icon shown in Figure 9.13 and block diagram shown in Figure 9.14. This sub-VI accepts two inputs: the array of Boolean control references and a corresponding array of zone status enumerated constants. This sub-VI reads each zone status and changes the color of the referenced indicator appropriately.

The color change is affected through the use of the *Colors[4]* property node. The help file is shown in Figure 9.15. This property node allows the foreground and background colors of the Boolean control to be changed for four different states of the control: *False*, *True*, *True to False*, and *False to True*. The foreground and background colors for each state are represented by a cluster of two colorboxes. A colorbox is a clickable LabVIEW constant that represents a specific color. Clicking the colorbox brings up a color picker navigation window that allows the user to specify the exact color desired. The *Colors[4]* property node expects an input array of four foreground/background colorbox clusters, but if a single-element array is wired to it, the property node uses the single element for all four states of the Boolean control. For the purposes of the security system controller, we are interested in setting the foreground color of the single-element colorbox cluster array.

Each element of the zone status array is an enumerated constant with the values *Normal*, *Alarm*, *Bypass*, and *Tamper*. The zone status array is wired to an auto-indexing input tunnel of a **`For`** loop. The appropriate colorbox for each individual zone status

Colors [4] Property

Short Name: Colors[4]

Requires: Base Package

Class: Boolean Properties

To Use: Create a property.

Array of up to 4 (Foreground Color, Background Color) pairs, where Foreground Color is the foreground color of the Boolean control and Background Color is the background color of the Boolean control. Color pairs include False, True, True to False, and False to True.

You can set the color of the front panel object by wiring a hexadecimal number with the form RRGGBB or by wiring the color box constant to the property.

This property is similar to the **Colors** option on the Appearance page of the **Boolean Properties** dialog box.

If the Boolean control has four (Foreground Color, Background Color) pairs for the four states, this property always returns an array of four (Foreground Color, Background Color) pairs when reading. When writing, you can pass in an array of length 1, 2 or 4. If you pass in only one element, this property uses that element value for all four states. If you pass in two elements, this property uses the first element value for the False and True to False states. Similarly, this property uses the second element for the True and False to True states.

If the Boolean control has only one (Foreground Color, Background Color) pair for all four states, this property returns an array of one (Foreground Color, Background Color) pair when reading, and uses the first array element when writing. You cannot change a Boolean control with one (Foreground Color, Background Color) pair to a Boolean control with multiple (Foreground Color, Background Color) pairs.

This property only uses the True to False and False to True states when you set the Boolean control to latch or switch when released. You can get and set these (Foreground Color, Background Color) pairs even if you do not have the control set to latch or switch when released; they just are not used.

Because this property does not change the data of the Boolean control, you can set it at any time.

You can use the this property for Boolean controls on the **Classic**, **Express**, **Modern**, and **Silver** palettes. You cannot use this property for Boolean controls on the **System** palette.

Figure 9.15: Help file for the `Colors[4]` property node

is selected by wiring the zone status enumerated constant to the selector of a case structure. The colorbox is then bundled into a foreground/background colorbox cluster as the foreground color. The foreground/background colorbox cluster is then used as the basis of a single-element array using the **`Build Array`** function (see help file in Figure 9.8). The resulting array is then written to the explicitly linked *Colors[4]* property node, changing the color of the referenced Boolean indicator on the front panel of the main application.

Exercise 3.8: Security System with Color-Changing Indicators

Implement the full version of the security system controller that includes the use of color-changing Boolean indicators to signal the current zone statuses. Replace the array of enumerated indicators you used in Exercise 3.5 with a cluster of Boolean indicators, as seen in Figure B.1. Use the `Get Lamp Refnums` sub-VI developed in Exercise 3.6 to generate the lamp references and preallocate the zone status array with the `Initialize Status Array From Boolean Reference Array` sub-VI developed in Exercise 3.7. Change the color of the Boolean indicator lamps with the `Set Zone Colors` sub-VI shown in Figure 9.14. Make sure to close the Boolean references when the main application exits. ■

Exercise 3.9: Testing the Scalability of the Security System Controller

Test the scalability of the full version of the security system controller as you did in Exercise 3.3. Does the program run as expected when you add an extra zone to the front panel? Are the log file entries correct? Do you need to make any changes to the block diagram when the number of zones is increased?

Review the three sub-VIs you added to the full version of the security system controller: the `Get Lamp Refnums` sub-VI developed in Exercise 3.6, the `Initialize Status Array From Boolean Reference Array` sub-VI developed in Exercise 3.7, and the `Set Zone Colors` sub-VI shown in Figure 9.14. What features of these sub-VIs either degraded or preserved the scalability of the security system controller? ■

Exercise 3.10: Polymorphic Sub-VIs

Imagine that you needed to maintain all three of the security system controllers developed in this problem set in Exercises 3.2, 3.5, and 3.8. These three versions of the security system controller each preallocated the zone status array with unique sub-VIs developed previously in Exercises 1.10, 3.5, and 3.7. Although they are all unique, these three preallocation sub-VIs are very similar. They all produce the same output—they only vary in the type of input they accept.

In a case like this, the three individual sub-VIs can be replaced by a single polymorphic VI. A polymorphic VI is one that can operate on many different types of inputs or produce multiple types of outputs. Sometimes, polymorphic Vis are configured manually (such as the data acquisition functions found in Chapter 15), or they may automatically configure themselves depending on the data type of the wires wired to their input terminals.

We have seen many polymorphic native LabVIEW functions so far. The **`Not Equal?`** function with the help file shown in Figure 7.11 can accept either scalars or arrays as inputs. The inputs can be of many data types: numeric, Booleans, strings, etc. The output of this function can be either a single Boolean scalar in the *Compare Aggregates* mode or an array in the *Compare Elements* mode. The **`Not Equal?`** function has both polymorphic inputs and polymorphic outputs.

In this exercise, you create a polymorphic `Initialize Status Array` that accepts either a cluster, a reference to a cluster, or an array of Boolean control references as inputs. In each case, it outputs a preallocated default zone status array that is the same length as the input cluster or array. To create a polymorphic VI, select *File. . . New* and *Polymorphic VI* from the menu window. A configuration window will open, as shown in Figure 9.16. This window allows you to add the individual VIs that handle the different input data types. You can create an icon for the polymorphic VI that appears on the block diagram instead of the individual VI icons. You may opt to have the polymorphic VI adapt automatically to the input type. A selector can be made to appear beneath the VI that allows the user to manually choose the instance VI that is used. The instance can be manually chosen from the right-click menu of the polymorphic VI. The names appearing on the selector and on the right-click menu can be specified in the configuration window.

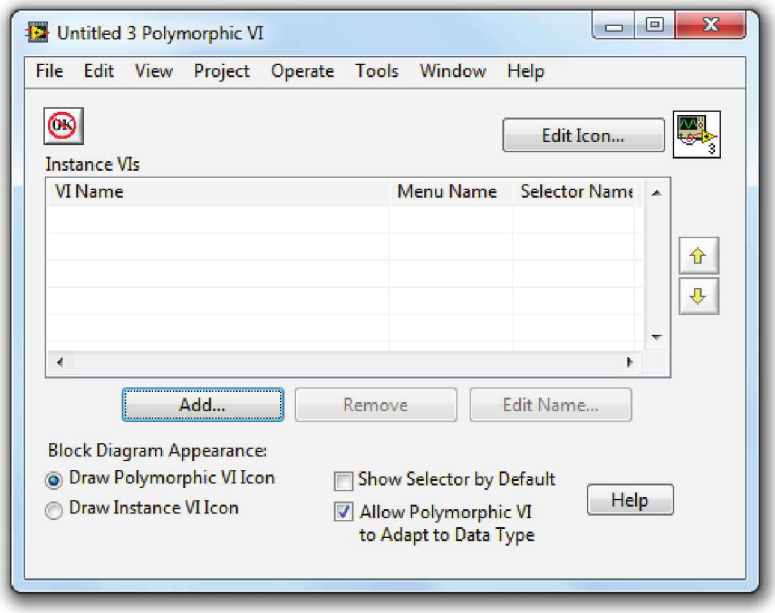


Figure 9.16: Configuration window for a new polymorphic VI

The instance VIs in the polymorphic VI must all have the same connector pane, and these connector panes all must be compatible. This means that, if one instance VI wires a particular connector pane terminal as an input, all instance VIs must either wire that same terminal as an input or must leave it unwired. It cannot be wired as an output by any of the other instance VIs. Of course, the data types wired to that terminal do not need to be the same, since that would defeat the purpose of the polymorphic VI. Likewise, if any instance VI wires a particular terminal of the connector pane as an output, all instance VIs must either wire that terminal as an output or leave it unwired. Polymorphic VIs may not be used as instance VIs in other polymorphic VIs.

Confirm that the connector panes of the `Initialize Status Array From Cluster` sub-VI, the `Initialize Status Array From Cluster Reference` sub-VI, and the `Initialize Status Array From Boolean Reference Array` sub-VI are compatible with each other. Create the polymorphic `Initialize Status Array` VI. Create an icon for the polymorphic VI and configure the polymorphic VI to display this icon instead of the instance VI icons. Configure the polymorphic VI to automatically adapt to the input type. Create names for the right-click menu and selector. Confirm that the polymorphic VI works by deleting the instance VIs from the block diagrams of the three security system controllers. Add the polymorphic VI to all three. Wire up the inputs to the polymorphic VI to confirm that it automatically adapts to the input type. Run the security system implementations to confirm that they still function correctly. ■

9.3 Boiler Controller Exercises

Exercise 3.11: Boiler Controller Classic State Machine

Reread the functional specification of the security controller given in Appendix C. Design and implement a classic state machine VI that satisfies the requirements of the functional specification. Where should the BOILER SIMULATION control and the BOILER CONTROL clusters be placed? Where should the BOILER STATUS indicator cluster be placed? Where should the STOP button be placed? Does the boiler controller need data stored persistently on a data highway? ■

9.4 Car Wash 2 Exercises

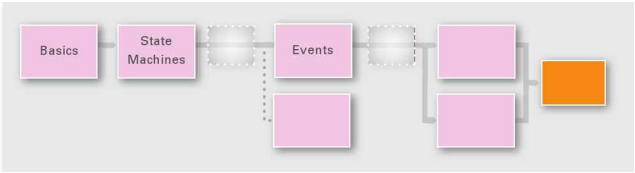
Exercise 3.12: Car Wash 2 Classic State Machine

Reread the functional specification of the Car Wash 2 controller given in Appendix D. Design and implement a classic state machine VI that satisfies the requirements of the functional specification. Use the state transition diagram you created in Exercise 1.38 as a guide. How is this car wash controller similar to the one developed in Chapter 8? How is it different?

Consider the placement of the controls and indicators. Where should the CAR POSITION CONTROL be placed? Where should the CAR WASH DISPLAY indicator cluster be placed? Where should the STOP button be placed?

Next, consider the data highway. One major difference between the two car wash controllers is that the second controller does not know ahead of time what sequence of states will make up the next car wash. Use the `Build Option Array` developed in Exercise 1.40 to create an array of option states chosen by the user. This array needs to be stored persistently, so it should be placed on the data highway. What other information needs to be stored persistently to allow this array to be used effectively?

Use the `Car Wash 2 Timer` you developed in Exercise 2.2 to time the individual car wash states. Use the `Car In Position` sub-VI you developed in Exercise 1.39 to determine if the vehicle is at the correct station to perform a given wash option. Use the `Next Car Wash State` you developed in Exercise 1.41 to determine the state transitions if needed. ■



10 Event-Driven State Machines

State machines can be used in many different applications. Imagine, for example, a code that needs to acquire data, analyze the data, and store the data and results. A state machine can be written with a set of states that always executes in order: *Initialize*→ *Acquire Data*→ *Analyze Data*→ *Save Data*. Such a program has minimal user input and always executes in the same order. The resulting VI is called a procedure-driven state machine.

State machines can be used for procedure-driven codes, but they really shine in event-driven codes. An event-driven code is a code where the order of execution depends on external factors known as *events*. Events are typically triggered from one of these sources:

User Events The user can trigger events by interacting with front panel controls.

Time Events Events can be triggered by timers.

Data Events Events can be triggered by analysis of incoming or existing data.

The classic car wash controller described in Chapter 8 is an event-driven state machine. It uses all three of the types of events:

User Events The user triggers events by manipulating the purchase selector, simulation switches, and the STOP button.

Time Events Transitions from one car wash cycle to another are controlled by the car wash timer.

Data Events If an error is detected, the state machine immediately transitions to the *Shutdown* state.

State machines are an excellent choice for event-driven problems, because the transitions from one state to another are flexible. Each state can decide where to go next based on user input, timing considerations, and data analysis. But the classic state machine architecture discussed in Chapter 4 can be awkward for event-driven scenarios. Event-driven scenarios require continuous monitoring of the front panel

to capture the desired events. The classic state machine performs this monitoring by polling the front panel controls every loop iteration.

Polling a large number of front panel controls is both awkward and an inefficient use of resources. Luckily, the classic state machine implementation of the car wash controller only needed to poll two controls: the purchase selector and the STOP button. Polling would be much more awkward for larger numbers of controls, and coding the *Wait* state would be more difficult.

Event structures are a much better way to monitor front panel controls. Event-driven state machines can be significantly improved by using event structures in their wait states rather than by polling front panel objects.

10.1 Event Structures

Event structures can respond to many types of events. Mouse movements, dragging, key presses, shortcut menu activations, and value changes all can be handled by the event structure. Some events, called *notify events*, only can be handled after they have had an effect on the front panel. Other events, called *filter events*, can be intercepted before they have an effect on the front panel. For example, if the user right-clicks on a front panel object to activate a shortcut menu, a filter event can be used to intercept the right-click event and alter the menu programatically before it is shown to the user.

Filter events are beyond the scope of this book. The most common use of event structures in this book is to respond to value changes of front panel controls. A value change event is fired whenever the user changes the value of a front panel object, and event structures can be configured to respond to these events.

Event-handling code is segregated into individual cases that respond to designated events, making event structures compact and readable. Event structures also "put LabVIEW to sleep" while the structure waits for an event to occur. Similar to the way that the **`Wait Until Next ms Multiple`** function frees the CPU while waiting, an event structure also reduces CPU usage while waiting.

10.2 Anatomy of an Event Structure

Figure 10.1 shows the anatomy of a fully loaded event structure with all of its options visible. Figure 10.2 shows the stripped-down event structure that is typical of the programs in this book.

An event structure is very similar to a case structure. It has a bounding rectangle, a menu selector, and can have input and output terminals. Input terminals can be ignored by individual cases, but output terminals must be wired in all cases. Output tunnels in event structures are automatically configured as *Use Default if Unwired*, but this behavior can be changed by right-clicking on the structure and deselecting this option.

At the top-left corner of the event structure is the *Timeout* terminal. The default value of this terminal is `-1`, which signals the event structure to wait indefinitely for

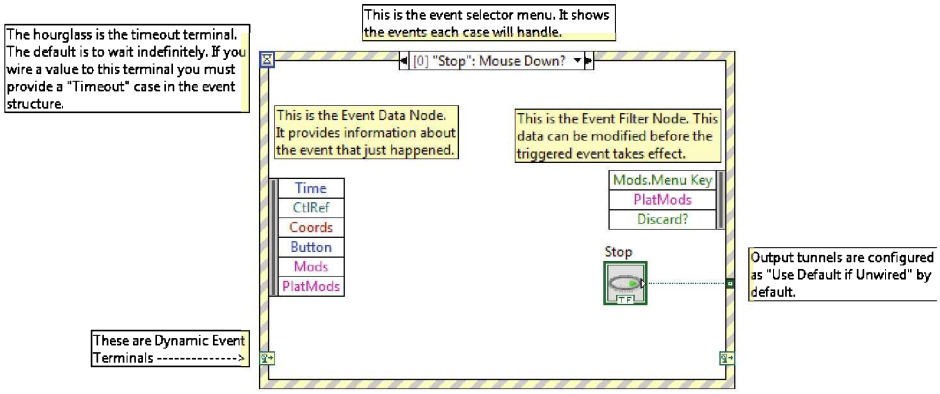


Figure 10.1: Anatomy of a fully-loaded event structure with the typical event structure configuration used in this book on the right

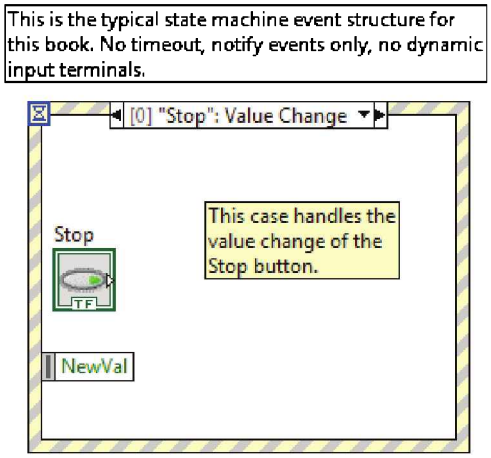


Figure 10.2: The typical event structure configuration used in this book

an event to occur. If a value is wired to this terminal, the event structure will wait a number of milliseconds equal to the wired value before timing out. A *Timeout* case must be provided if the *Timeout* terminal is wired, and this case will execute when the structure times out.

Two types of events can be handled: *notify* and *filter*. A *notify* event allows the event structure to respond to an event after the effect of the event has taken place. A *filter* event allows the event structure to intercept the event after it is triggered but before the effect has taken place. The names of filter events end with a question mark.

For example, suppose you wish to respond to the user clicking the mouse. If you program the event structure to respond to the *Mouse Down* notify event, the mouse click will take effect before the event structure executes. If you program the event structure to respond to the *Mouse Down?* filter event, the event structure will execute

before the mouse click takes effect. You could discard the click, change the effect of the click based on what state the program was in, or take other actions.

The *Event Data Node* is found inside the event structure along the left edge. This is a cluster-like node that can be expanded or contracted by clicking it and dragging. It contains data about the event that can be used inside the event case if desired. This node is available for both notify events and filter events. On the right-hand edge of the structure is the *Event Filter Node*. This is an output node that contains data that can be modified before the event takes effect. It is only present in cases that handle filter events. We only will be dealing with notify events, so the *Event Filter Node* will not be present in the event structures shown in this book.

An event structure also can be configured to handle dynamic events. These are events that can be fired programatically without user input. For instance, a dynamic event can be fired when data exceeds an imposed limit or when an error is detected. This topic will be covered in Chapter 27.

Throughout most of this book, we will be using the stripped-down version of the event structure shown in Figure 10.1. We will not be using filter events, so we won't have access to the *Event Filter Node*. We will not be using the data available in the *Event Data Node*, so this node usually will be collapsed to a single entry and left unwired. Dynamic events will not be used until Chapter 27, so the dynamic event terminals will remain hidden. Since the event structures primarily will be used to detect changes in the front panel controls during the *Wait* state, the only event type that we consider here are value changes of front panel objects.

10.3 Configuring an Event Structure

To create an event structure, simply drop it onto the block diagram and drag it to the desired size. The event structure is created with a single event case handling the *Timeout* event. The event structures in this book are not configured to time out, so this case can be deleted or edited to handle a different event.

Lets take a look at how to configure an event structure using the `Multitest` VI developed in Chapter 4. The configuration window for handling the value change of the STOP button is shown in Figure 10.3. This window has a number of panels. The *Event Sources* panel allows you to choose the source of the event. Events can come from a number of sources at the application level, VI level, or front panel controls level. Once a source has been selected, the types of events that can be monitored from that source are shown in the *Events* panel. To configure the event case to handle the value change of the STOP button, simply select the *Stop* control from the *Controls* section of the *Event Sources* panel and then select the *Value Change* event from the *Events* panel, as shown in Figure 10.3.

The configuration window allows an individual event case to respond to more than one event. Additional events can be added to an existing event case by clicking the ADD EVENT button in the lower-left corner of the configuration window. In this book, we do not use this ability and only configure event cases to respond to single events.

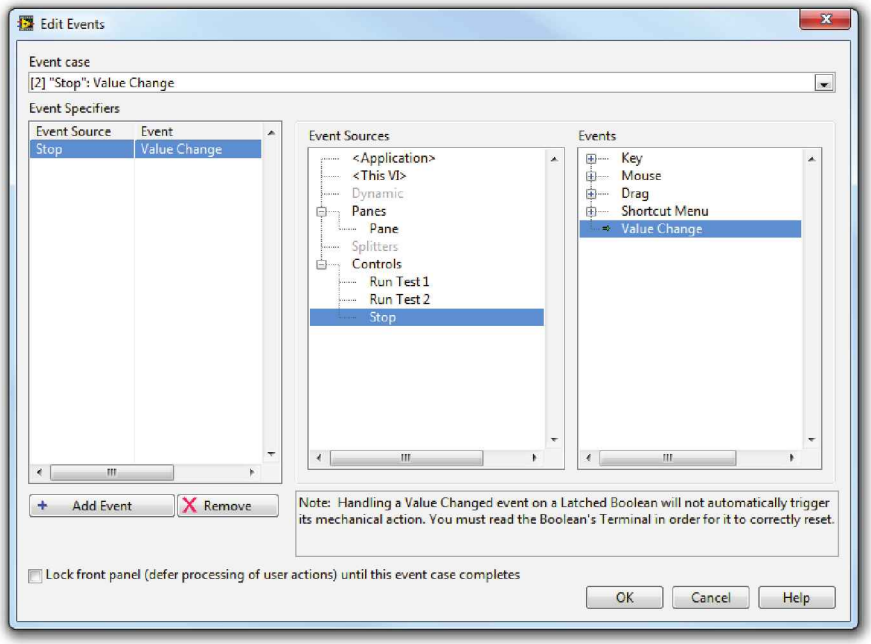


Figure 10.3: The configuration window of the STOP value change event case

10.4 Event Structure Quirks and Caveats

Event structures are powerful, but they can be confusing and possibly lead to unexpected behavior in programs that use them. Event structures are powerful, because LabVIEW is continuously monitoring and processing event-related tasks in the background when a VI is running. To use event structures effectively, you need to be aware of what LabVIEW is doing behind the scenes.

The Event Queue

Event structures are serviced by an event queue. This queue is maintained by LabVIEW and is not accessible to the developer. When an event that an event structure is configured to handle occurs, the event is put on the event queue. *This happens while the VI is running, even if the event structure is not actively waiting for an event to occur.*

When the data flow of a program allows an event structure to execute, the event structure pulls the first event from the event queue and responds to it. If the event queue is empty, the event structure waits until an event that it has been programmed to respond to occurs. This can lead to unexpected behavior, because the event structure can respond to events that occur before the event structure executes.

Loops

An event structure is like a case structure, where the entire structure terminates once a single event case has executed. Event structures are typically used to monitor events

continuously, so they are usually found inside of loops. The loop will sleep while the event structure is waiting for an event, so it usually is best to only have one event structure in a loop. Multiple event structures in a loop could cause the loop to sleep longer than intended or could cause the program to hang.

Value Changes of Latched Booleans

Value changes of latched Boolean controls can also cause problems. A latched Boolean control does not reset to its default value until LabVIEW "reads" it. This can cause difficulties when using a latched Boolean control to stop a loop containing an event structure.

Figure 10.4 shows a poor method for shutting down a loop with an event structure. The block diagram has a **`While`** loop containing an event structure with a STOP button outside of the event structure wired directly to the continuation terminal of the loop. The intent is that the loop will stop when the STOP button is pressed, but

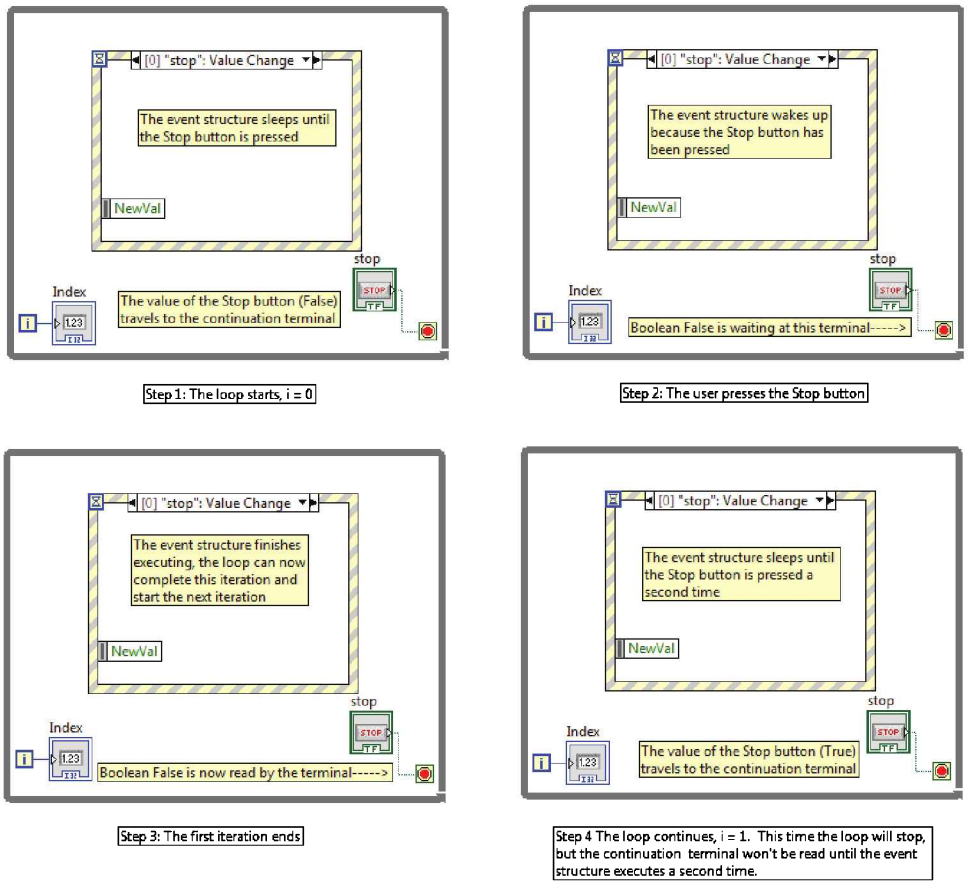


Figure 10.4: Dataflow sequence showing the wrong way to shut down a **`While`** loop containing an event structure

this is not what happens. When the user presses the STOP button, the button stays down but doesn't pop back up, and the program continues to run. If the user presses the button a second time, it will pop back up and the program stops.

Why does this happen? The key lies in the dataflow of the program. The STOP button is wired directly to the stop terminal of the loop, and there is no input/output relationship between the button and the event structure. The STOP button and the event structure are in parallel with each other.

When the program is executed, two things happen at once during the first iteration of the loop: the value of the STOP button (False) travels to the stop terminal of the loop and the event structure goes to sleep waiting for a value change of the STOP button. Even though the loop is sleeping, a Boolean False is now waiting at the conditional terminal.

The loop sleeps until the STOP button is pressed. When the STOP button is pressed, the event structure executes and the False at the loop conditional terminal is evaluated. The loop continues for another iteration, and the value of the STOP button—now True—travels to the stop terminal of the loop. But the conditional terminal of the loop won't be evaluated until the event structure wakes up again and executes. The event structure will sleep until the STOP button is pressed again. When the user presses the stop button a second time, its value changes and the event structure executes. The True value at the stop terminal of the loop is now read and the loop terminates.

A much better way to stop a loop with an event structure is shown in Figure 10.5. The STOP button is placed within the event structure case that handles the value change of the STOP button. When the STOP button is pressed:

- The event case handling the value change of the STOP button executes.
- The STOP button is read and its mechanical action resets it to its default value.
- The value of the STOP button (True) is sent to the loop conditional terminal.
- The loop terminates after the first iteration (i = 0).

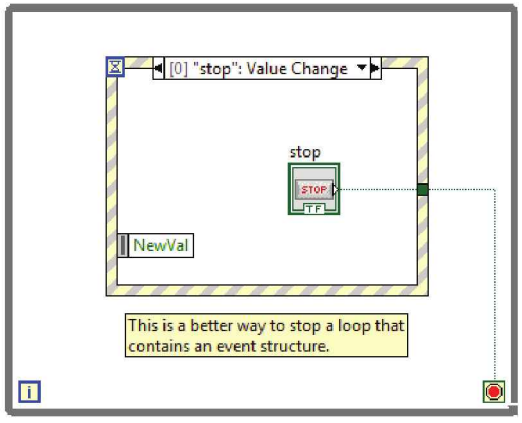


Figure 10.5: Block diagram showing the proper way to shut down a `While` loop containing an event structure

If the event structure is part of a state machine with a *Shutdown* state, the event structure case handling the value change of the STOP button should send the state machine to the *Shutdown* state rather than shutting down the state machine directly. This allows the state machine to shut down gracefully when the STOP button is pressed by the user.

Wiring Tip: As a rule of thumb, a control should be placed within the event case that handles its value change unless it is needed elsewhere in the program.

Locking the Front Panel

At the very bottom of the event structure configuration window, there is a checkbox with the message: "Lock front panel (defer processing of user actions) until this event case completes" (see Figure 10.3). If this box is checked, the front panel is unresponsive until the code within that event case completes. If the user clicks on a button, moves a slide, or turns a knob, nothing happens on the screen while the event case code is processing. However, LabVIEW still registers these events and puts them on the event queue. When the event case that has locked the front panel completes, all of the front panel events are processed in the order that they occur. Suddenly, the button will press, the slide will move, and the knob will turn.

This can be confusing to the user. Even worse, remember that LabVIEW puts events on the event queue even if the event structure isn't actively waiting. If an event case locks the front panel, the front panel is locked even if the event case isn't actively waiting for an event to occur or after the event structure has completed executing. In extreme cases, this can can cause the program to hang.

There are situations where it is necessary to lock the front panel while an event structure is executing. For example, event cases that handle filter events must have the front panel locked. But situations like this are beyond the scope of this book. None of the event cases shown in this book lock the front panel. In general, it is a good rule of thumb to avoid locking the front panel unless the functional specifications require it. Be careful, though, since the option to lock the front panel is checked by default in event cases that handle notify events.

Parallel loop architectures (discussed in Chapter 27) are more complex than single loop architectures, but they make event structures easier in many ways. In a parallel loop architecture, the event structure is usually sits in a dedicated loop and is always "on"—always waiting to respond to events. In a single loop architecture, the event structure is only "on" when the state machine is in the *Wait* state. So be careful when locking the front panel in a single loop architecture. If the user triggers an event that locks the front panel when the event structure is not actively waiting for an event, the front panel could be frozen for quite some time or the program may be prevented from shutting down without being aborted. Try to construct your program in such a way that the user can only trigger events while the event structure is actively waiting for them. For example, use property nodes to disable and gray out front panel controls when the event structure that handles them is not active.

10.5 Event Structure Guidelines

Many issues described previously can be avoided if you heed the following suggestions.

- Avoid using an event structure outside a loop.
- Avoid placing two event structures in one loop.
- Use caution when you configure one event case to handle multiple events.
- Remember to read the terminal of a latched Boolean control in its *Value Change* event case.
- Don't lock the front panel during in an event case unless absolutely necessary.

A good rule of thumb is to try to put as little code as possible in each event case. Use the event case to transition to the state where the work is done rather than do the work in the event case itself. This improves code readability and keeps the event structure nimble, reacting to user input and then passing control to states designed to handle that input as quickly as possible.

10.6 Example: Building an Event-Driven State Machine

To demonstrate the use of event structures, let's upgrade the `Multitest` VI developed in Chapter 4 to an event-driven state machine. Recall that, in the classic state machine implementation of the `Multitest` VI, the *Wait* state consisted of a ladder of select functions that polled the front panel controls to determine the next state transition (see Figure 4.15).

The classic state machine implementation of the *Wait* state is awkward. It keeps the state machine cycling even when there is no user input. It is also difficult to maintain and scale if more front panel controls are added in the future. An event structure would be a much better solution for the *Wait* state.

Upgrading the classic state machine to an event-driven state machine is quite simple. The only change that needs to be made is to revise the *Wait* state, replacing the selector ladder with an event structure as shown in Figure 10.6.

The event structure will have three event cases handling the value changes of the RUN TEST 1 button, the RUN TEST 2 button, and the STOP button. Figure 10.6 shows the case that handles the value change of the RUN TEST 1 button. When the RUN TEST 1 button is pressed, the *Run Test 1* state is placed on the state shift register. Notice that the terminal for the RUN TEST 1 button has been placed in the event case that handles its value change. This ensures that the terminal is read by LabVIEW when its value changes, allowing the button to reset to its latched default value. The configuration window that created this case is identical to the one shown in Figure 10.3, except the event case handles the value change of the RUN TEST 1 button instead of the STOP button.

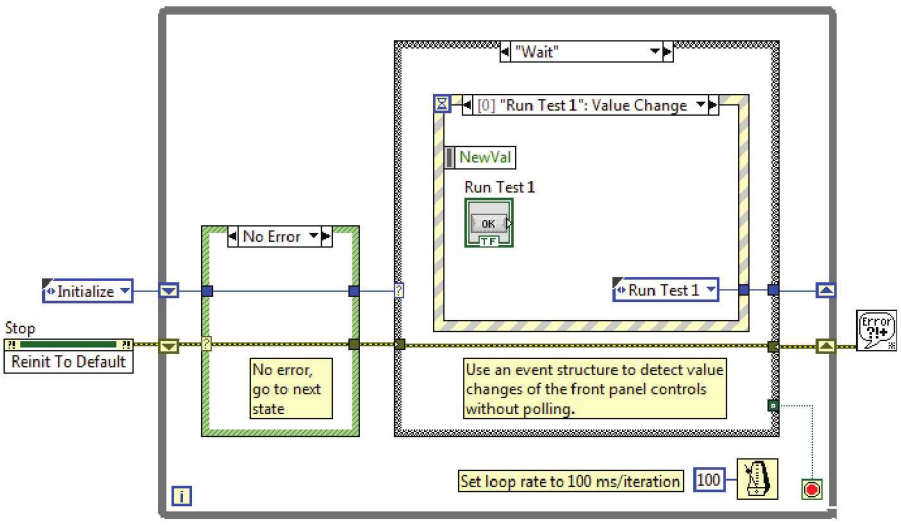


Figure 10.6: Replacing the *Wait* state of the `Multitest` classic state machine with an event structure, where this shows the RUN TEST 1 value change state

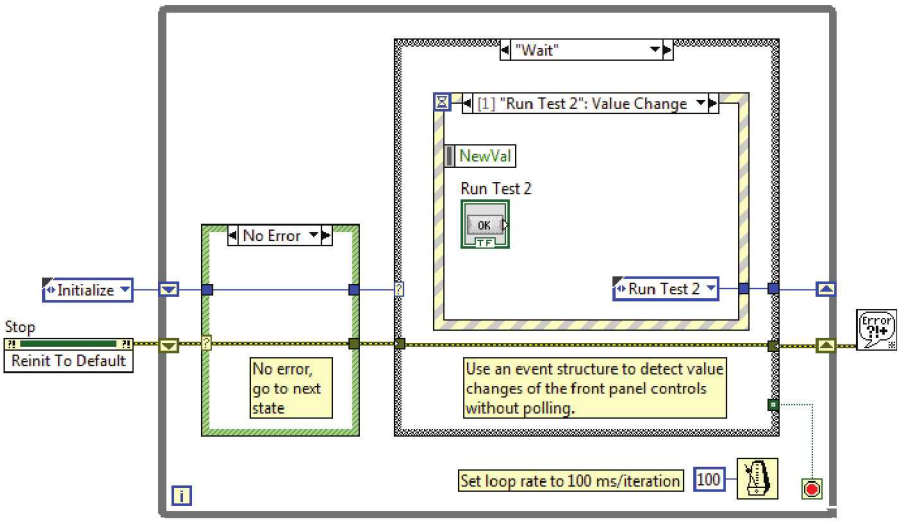


Figure 10.7: The event case handling the value change of the RUN TEST 2 button

The event case for the value change of the RUN TEST 2 button is very similar and is shown in Figure 10.7. This event case puts the *Run Test 2* state on the state shift register when the RUN TEST 2 button is pressed. The event case handling the value change of the STOP button is shown in Figure 10.8.

The rest of the state machine remains unchanged from the one developed in Chapter 4. By replacing the **`Select`** ladder with an event structure, we have upgraded

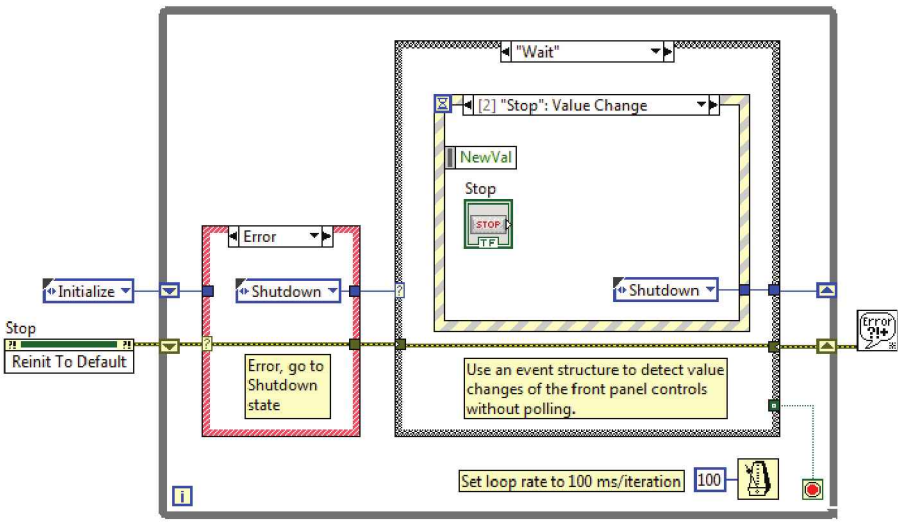
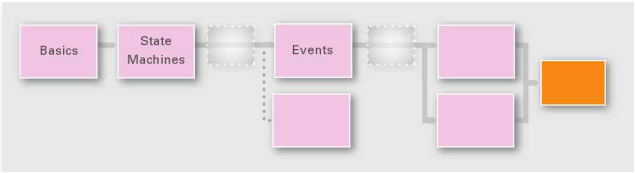


Figure 10.8: The event case handling the value change of the STOP button

the classic state machine to an event-driven state machine. This makes the state machine more readable, scalable, and maintainable. The upgrade also improves its performance, allowing the state machine to "sleep" while waiting for user input.



11

The Event-Driven Car Wash Controller

We can now upgrade the classic state machine implementation of the car wash controller to an event-driven car wash controller. This upgrade only affects the *Wait* state—all other states of the classic state machine car wash controller presented in Chapter 8 remain the same.

A drawback of the classic state machine is the awkwardness and inefficiency of the *Wait* state due to polling front panel controls. These problems can be solved by replacing the code in the *Wait* state of the classic state machine car wash controller with an event structure. The event structure provides two major benefits. First, it is easily extensible. New front panel controls can be accommodated by adding new cases to the event structure. Second, it is efficient. The event structure "goes to sleep" when there are no events to process.

11.1 Adding the Event Structure to the *Wait* State

Figure 11.1 shows how an event structure can be added to the *Wait* state of the car wash controller. The event case shown detects a value change of the purchase selector. The purchase selector is a cluster of two Boolean controls: the DELUXE WASH button and the ECONOMY WASH button. If either one of these buttons is pressed, the value of the purchase selector cluster changes. The value of the DELUXE WASH control is then placed on the data highway, and the *Lock Selector* state is placed on the state shift register of the main loop.

The configuration window for this event case is shown in Figure 11.2. Notice that the value change of the <*All Elements*> entry of the *Purchase Selection* cluster has been selected. This means that the event case will execute if any element of the purchase selector changes value.

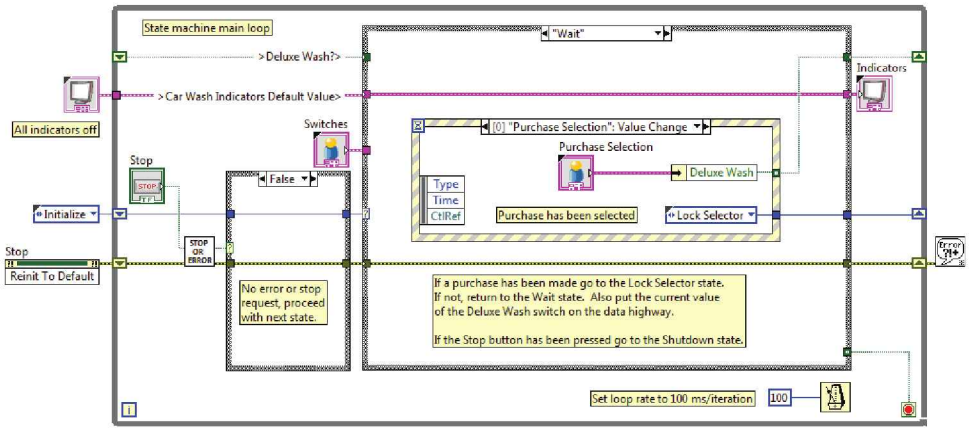


Figure 11.1: The event case handling the value change of the PURCHASE SELECTION control

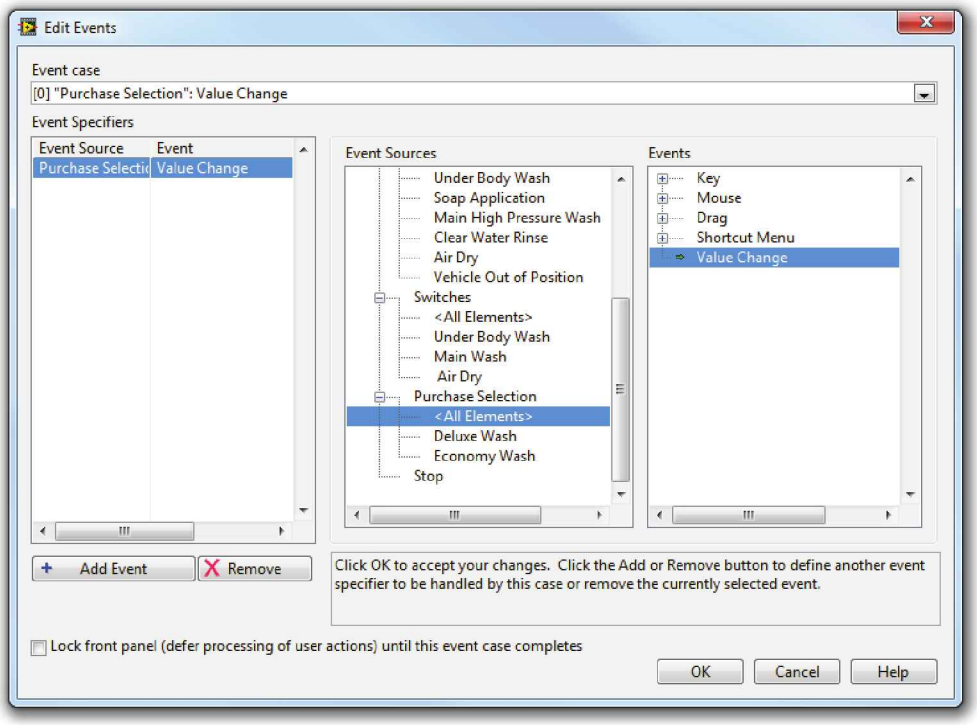


Figure 11.2: The configuration window for the event case handling the value change of the PURCHASE SELECTION control

11.2 The STOP Button

The STOP button needs special consideration in an event-driven state machine. In the classic state machine implementation of the car wash controller developed in Chapter 8, the STOP button is checked every iteration by the guard clause. In fact, checking the STOP button is the first thing that is done each iteration of the state machine.

The classic state machine runs continuously, polling all front panel controls every 100 milliseconds. This is inefficient, but it does guarantee that the state machine will respond to a stop request within the 100 millisecond limit mandated by the functional specification.

The event-driven state machine is more efficient, putting the main loop to sleep in the *Wait* state when no front panel activity has been detected. But the STOP button is no longer being polled when the event structure is sleeping. In order to respond to a stop request while in the *Wait* state, we need to configure an event case to monitor the value change of the STOP button, as shown in Figure 11.3. The corresponding event configuration window is shown in Figure 11.4.

The STOP value change event case places the *Shutdown* state on the state shift register if the STOP button is pressed. This ensures that the state machine will wake up and shut down if the STOP button is pressed. It is usually recommended that a control should be placed within the event case that handles its value change, but in this case this is not desirable, since the STOP button needs to be polled in the lobby of the state machine. We need to monitor the STOP button in two different places. We need a case in the event structure to respond to the STOP button while the state machine is waiting, and we need to poll the STOP button every loop iteration when the state machine is not waiting. This duplication of effort is due to the fact that in a

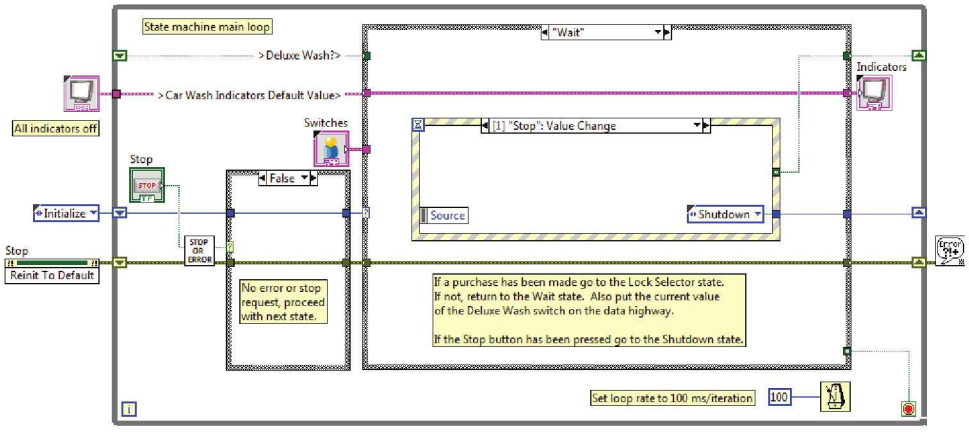


Figure 11.3: The event case handling the value change of the STOP button

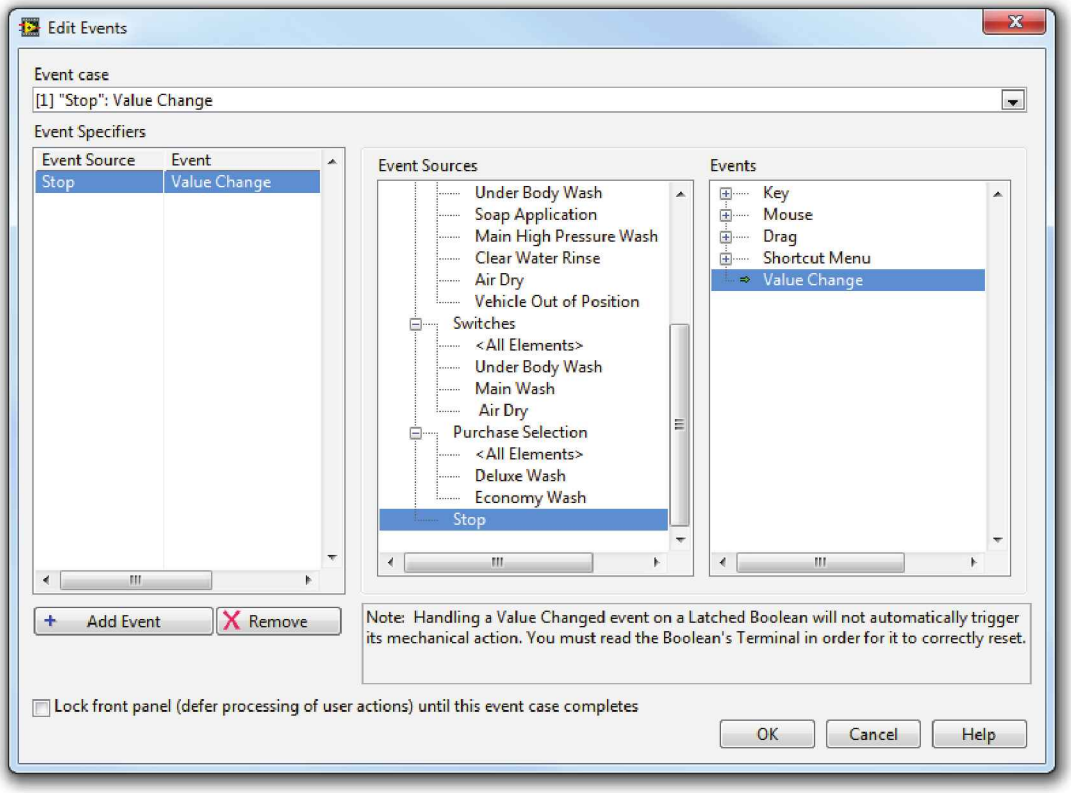
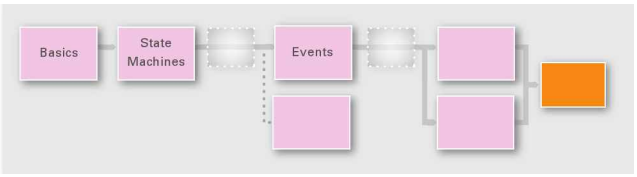


Figure 11.4: The configuration window for the event case handling the value change of the STOP button

single-loop state machine the event structure is only active when the program is in the *Wait* state.



12

Problem Set 4

12.1 Traffic Light Controller Exercises

Exercise 4.1: Event-Driven Traffic Light Controller

Is the *Traffic Light* program a good candidate for an event structure? Why or why not? If it is a good candidate for an event-driven architecture, upgrade the classic state machine implementation developed in Exercise 3.1. ■

12.2 Security System Controller Exercises

Exercise 4.2: Event-Driven Security System Controller

The *Security System* program is an excellent candidate for an event structure. Do you see why? Upgrade the classic state machine implementation developed in Exercise 3.8 by adding an event structure to the *Wait* state. Consider the location of the STOP button. Does it still need to be in the lobby of the state machine? What are the consequences and benefits of moving it into the event structure? ■

12.3 Boiler Controller Exercises

Exercise 4.3: Event-Driven Boiler Controller

Is the *Boiler Controller* program a good candidate for an event structure? Why or why not? If it is a good candidate for an event-driven architecture, upgrade the classic state machine implementation developed in Exercise 3.11. ■

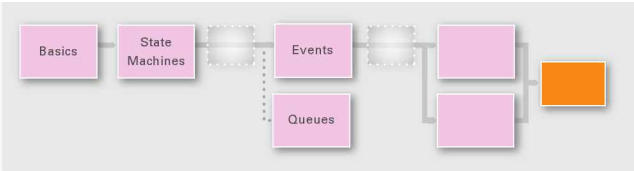
12.4 Car Wash 2 Exercises

Exercise 4.4: Event-Driven Car Wash 2 Controller

Like the original car wash controller, the Car Wash 2 controller is a good candidate for an event structure. Add an event structure to the *Wait* state to avoid polling the START button, but do not change anything else in the code. The Car Wash 2 controller also waits during the *Car Enter* and *Car Exit* states. Is it a good idea to add event structures to these states also? Why or why not? ■

Exercise 4.5: Revising the Event-Driven Car Wash 2 Controller

In Exercise 4.4, you were asked to add an event structure to the *Wait* state of the Car Wash 2 controller to avoid polling the START button. The implementation in the solution of this exercise left the ENTRY CONSOLE control in the event structure and placed its value on the data highway. But keeping the control in the event structure is not necessary, because the event structure can monitor the START button without the control being in the structure itself. Revise the event-driven Car Wash 2 controller developed in Exercise 4.4. Remove the ENTRY CONTROL from the event structure and from the data highway. Place the control in the state where it is needed instead. Compare the two event-driven Car Wash 2 controllers. What are the advantages and disadvantages of each implementation? ■



13 Queued State Machines

The event-driven state machine is an improvement over the classic state machine. The event structure eliminated polling the user interface, and it made the car wash controller easier to read, scale, and maintain. But the event structure did nothing to change the architecture of the classic state machine.

The hallmark of the classic state machine is that each state must determine the transition to the next state. The classic state machine can only think one move ahead. This is a serious limitation. Some states in a state machine are bound to be branch points, which are states that have more than one possible transition. In a classic state machine, these branch point states need to have logic built in to them to determine which state to transition to next.

The car wash controller developed in Chapter 8 is a good example. The *Lock Selector* and the *Rinse* states are branch points. This can be seen graphically in the state diagram shown in Figure 2.5. The transitions from these two states depend on whether the Deluxe Wash or the Economy Wash is chosen. This means that a Boolean needs to be added to the data highway to store the type of wash selected. These states also need logic to interpret this Boolean and select the next state.

This technique solves the problem, but at a cost. The *Lock Selector* and the *Rinse* states are coupled to the wash type. These states need to know which wash type is chosen before they can perform their function. This degrades the maintainability and scalability of the code.

But how can this be avoided? How can the car wash states be decoupled from the type of car wash? What we need is a way to store a list of states. We need a way to add states to the list as a group when a wash type is chosen and we need a way to take them off the list one by one in the correct order as the state machine runs through the chosen wash. The list of states will drive the state machine transitions. Individual states need not decide which state to transition to next and thus not need to know which type of wash had been selected. The state transition diagram for this sort of state machine looks like the one shown in Figure 2.6.

13.1 The Power of Queues

LabVIEW provides a group of functions that can be used to manage ordered groups of states called the queue functions. A queue is an ordered buffer used to store information for later use—first in, first out (FIFO)—like a line of people in a store. People enter the line at the end and exit the line at the front. Similarly, elements are added to the end of a queue and are removed from the front.

Another way to think of a queue is as a special type of array. Like arrays, every element in a queue has the same data type and is ordered in a fixed sequence. Unlike an array, though, queue elements only can be added at the ends the queue. Elements only can be taken off the queue from the front. Arrays are random access, where any element can be accessed by specifying an index. Queues are sequentially accessed, and elements must be read in the order that they are queued.

13.2 Queue Functions

The queue functions are found in the in the *Programming...Synchronization...Queue Operations* sub-palette. The subset of functions used in this book are described in Table 13.1.

A queue is created and configured with the **`Obtain Queue`** function (see the help file in Figure 13.1). The data type of the elements on the queue is determined by

Table 13.1: Useful queue functions

Obtain Queue	This function creates a reference to a new queue or obtains a reference to an existing named queue. This function also determines the data type of the queue. Optionally, a name can be given to the queue and a maximum size may be set. For more details see Figure 13.1.
Enqueue Element	This function places an element at the back of a queue. Queues are FIFO (first in, first out), so this is the function typically used to place an element on the queue. For more details see Figure 13.2.
Enqueue Element At Opposite End	This function places an item at the front of the queue. This makes the element first in line to be dequeued. For more details see Figure 13.3.
Dequeue Element	This function removes the element from the front of the queue and returns it. This function waits forever for an element to be available unless a timeout value has been wired to the timeout terminal. Attempting to dequeue an element from a non-existent queue produces an error. For more details see Figure 13.4.
Release Queue	This function releases the queue reference, destroying the queue. For more details see Figure 13.5.

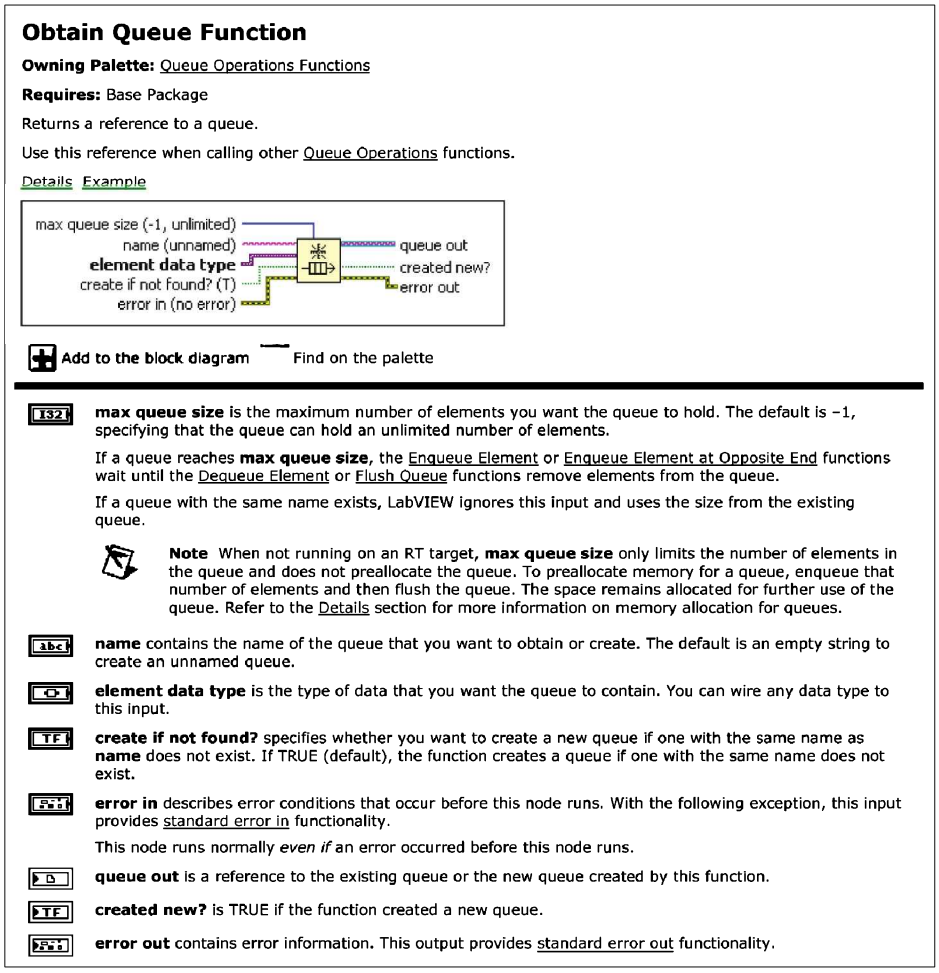


Figure 13.1: Help file for the `Obtain Queue` function

the data type of the constant or control wired to the element type input. The value that is wired to the element data type input is *not* placed on the queue. It is only used to determine what kind of element will be permitted to be placed on the queue.

The size of the queue is unlimited unless a specific queue size is wired to the max queue size input. If a maximum size is specified, the queue will never exceed this size. If an attempt is made to add an element to a queue that is already full, the function attempting to add the element will wait until space is made available by elements exiting the queue.

A queue can be given a name when it is created. Queues are generally accessed through the queue reference wire. The reference to the queue travels along the wire like any other piece of data. But if a queue has a name, a reference to the queue can be generated anywhere in the program by wiring the name to another **`Obtain`**

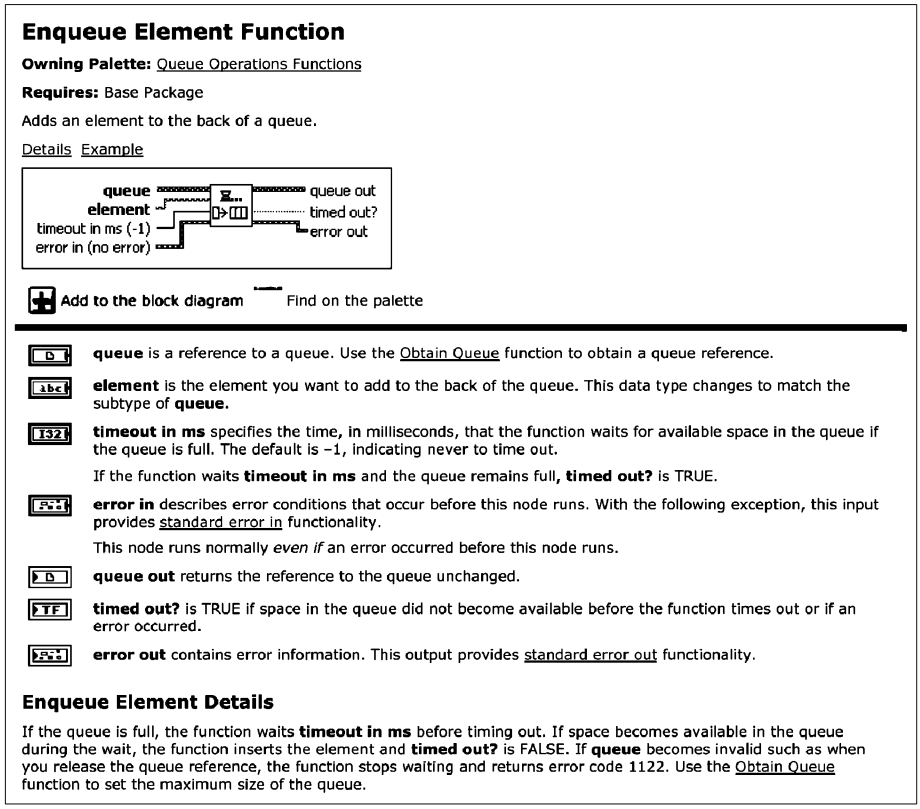


Figure 13.2: Help file for the `Enqueue Element` function

`Queue` function. If a queue with that name already exists, the reference generated by the function will be to the existing queue. Different VIs running under the same application of LabVIEW can access the same queue if the queue is named. This is a powerful feature of queues, but it is beyond the scope of this book. The queues used in the examples in this book are not named.

Once a queue has been created, it is populated with the **`Enqueue Element`** function (see the help file in Figure 13.2). The **`Enqueue Element`** function places an element on the back of the queue. The **`Enqueue Element at Opposite End`** function (see the help file in Figure 13.3) places an element at the front of the queue. Elements are dequeued from the front of the queue, so enqueuing an element at the front puts it next in line for dequeuing.

The **`Dequeue Element`** function removes the element at the front of the queue and returns it as an output (see the help file in Figure 13.4). If the queue is empty, this function will wait forever until an element becomes available unless a timeout is specified. If this function tries to dequeue an element from a non-existent queue, an error is generated.

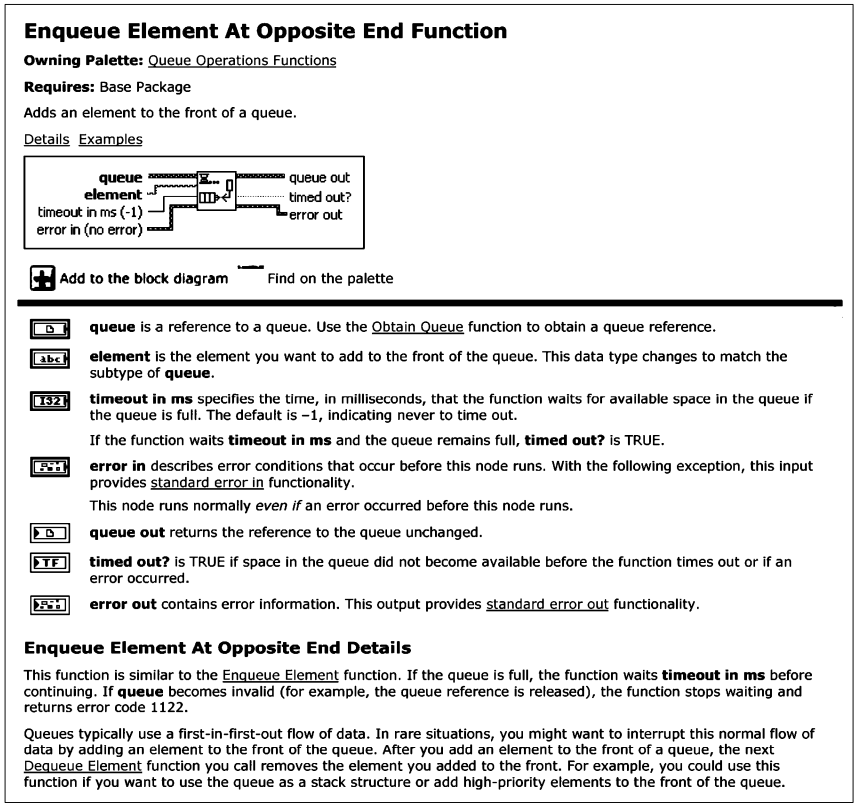


Figure 13.3: Help file for the `Enqueue Element At Opposite End` function

The **`Release Queue`** function destroys the queue by releasing the queue reference (see the help file in Figure 13.5). It is good programming practice to "free-up" system resources by releasing references when they are no longer needed. The **`Release Queue`** function operates normally—even if an error is present at the *error in* terminal. This means that if the **`Release Queue`** function is placed in the *Shutdown* state of a state machine it can safely be placed on the main error wire. The queue reference will be released even if an error is present on the main error wire.

13.3 By-Value versus By-Reference

LabVIEW is a by-value dataflow language. This means that data values travel along the wires connecting functions. Splitting a by-value wire creates copies of the data. These copies are independent of each other, and each can be modified without affecting each other.

Queues are passed by-reference rather than by-value. The actual queue does not travel on a wire, only a reference to the queue. If a queue wire is split, both wires

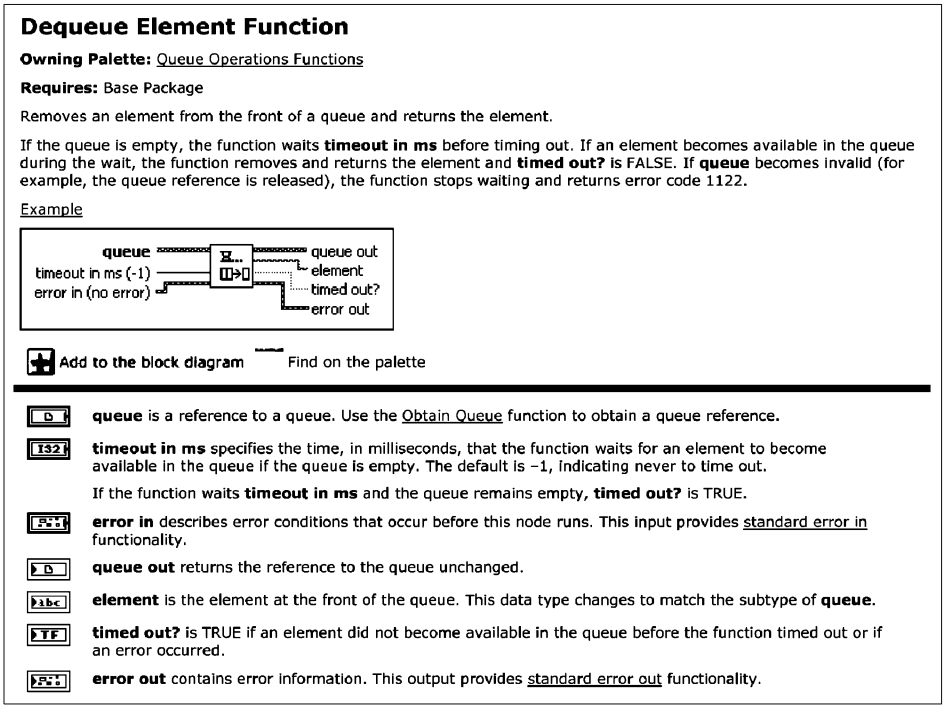


Figure 13.4: Help file for the `Dequeue Element` function

carry references to the same queue. Changes made to the queue on one wire are seen by the other wire. Because of this, queues need to be used carefully. By-reference wires have power and convenience, but they can be more vulnerable to problems than by-value wires.

Here's an example. Imagine that you are making a purchase using cash. When you pay for a purchase with cash, you are making a by-value transaction. Your wealth is decreased at the time you make the purchase. If you don't have enough money to make the purchase, you can't complete the transaction. This is inconvenient, but cash transactions prevent you from spending money that you do not have.

Now imagine that you make the purchase using a check. Now you are making a by-reference transaction. Instead of transferring money to the cashier directly, you are providing a reference to your bank account with a promise to pay a specified amount. This is convenient, but it has the potential for problems. You can make a promise with a check that your account cannot keep, resulting in bounced checks and overdrafts.

Queues are relatively well-behaved in single-loop state machines, but they can cause trouble if operations on the same queue are performed in parallel with each other. There are two common scenarios to watch out for. The first scenario occurs when there are multiple enqueue functions putting elements on the same queue at the same time. This can lead to the situation shown in Figure 13.6. Three enqueuing

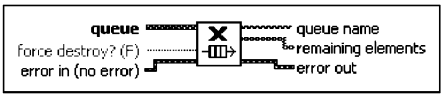
Release Queue Function

Owning Palette: Queue Operations Functions

Requires: Base Package

Releases a reference to a queue.

Details Example



Add to the block diagram Find on the palette

queue is a reference to a queue. Use the Obtain Queue function to obtain a queue reference.

force destroy? indicates whether to destroy the queue. If FALSE (default) and you want to destroy the queue, either call the Release Queue function a number of times equal to the number of times you obtained a reference to the queue or stop all VIs using the queue reference. If TRUE, the function destroys the queue and you do not have to call the function multiple times or stop all VIs using the queue reference. Destroying the queue invalidates all references to the queue.

error in describes error conditions that occur before this node runs. With the following exception, this input provides standard error in functionality.

This node runs normally *even if* an error occurred before this node runs.

queue name is the name of the queue.

remaining elements is the array of elements that were in the queue before the function released the queue. The first element in the array is the element from the front of the queue and the last element in the array is the element from the back of the queue.

This data type changes to match the subtype of **queue**.

error out contains error information. This output provides standard error out functionality.

Release Queue Details

You can use the Obtain Queue function to obtain a reference to the same queue with the same name multiple times. To destroy a queue, call the Release Queue function a number of times equal to the number of times you obtained a reference to the queue or stop all VIs using the queue reference.

If **force destroy?** is TRUE, this function releases all references to the queue and destroys the queue.

Any Enqueue Element, Enqueue Element at Opposite End, Preview Queue Element, and Dequeue Element functions waiting on the queue time out and return error code 1122.

Figure 13.5: Help file for the `Release Queue` function

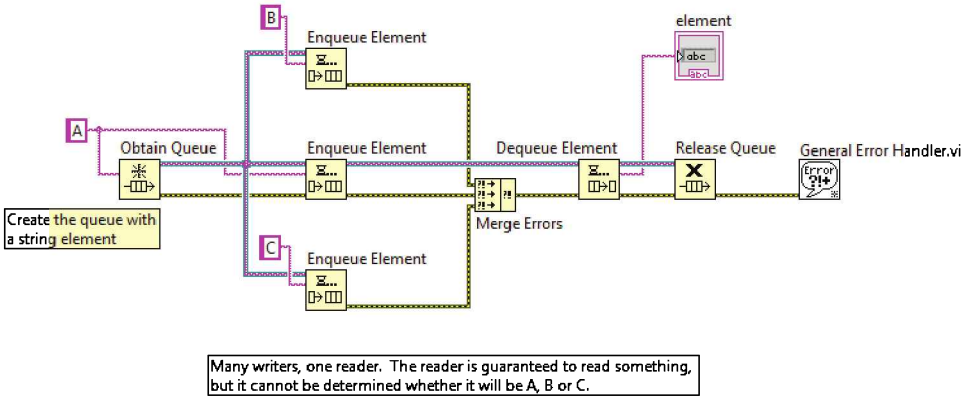


Figure 13.6: Which element will the reader dequeue?

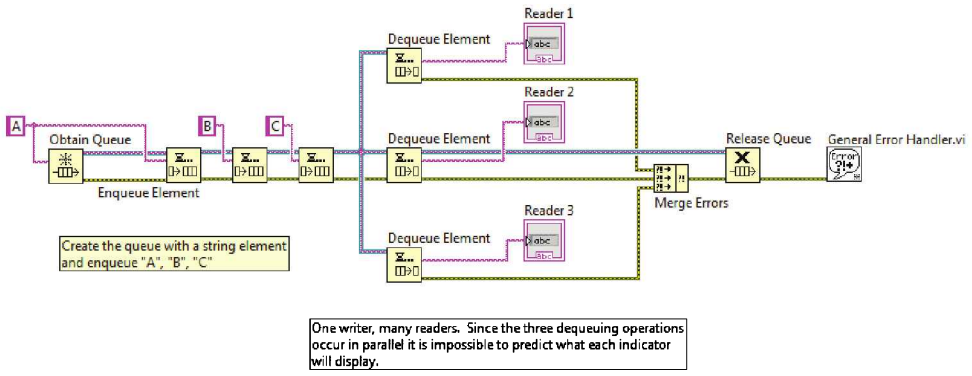


Figure 13.7: What will be displayed by each indicator?

operations occur in parallel upstream from a dequeuing operation. Dataflow does not determine the order in which the three enqueuing operations accessed the queue, so it is impossible to predict which element will be dequeued when the code executes.

Figure 13.7 shows the opposite problem. In this situation, the strings `A`, `B` and `C` are enqueued upstream of three parallel dequeuing operations. Since the three dequeuing operations have no data dependence it is impossible to predict what values will be displayed by the three indicators.

Both of these examples suffer from race conditions. A race condition is a programming situation in which the outcome of a program is affected by the order in which parallel operations execute. Race conditions will be discussed more thoroughly in Chapter 28. To reduce the chance of race conditions, pay careful attention to the sections of code that write to and read from queues. Try to keep the functions that enqueue elements confined to one section of the code, typically the *Wait* state in a state machine. Try to also confine the dequeuing operations to one section of the code, typically the guard clause of the state machine. In this way, you will avoid the problems associated with many readers and writers.

13.4 Building a Queued Event-Driven State Machine

Let's demonstrate the use of queues by upgrading the `Multitest` VI that was first developed as a classic state machine in Chapter 4 and then upgraded with an event structure in Chapter 10. In both of these implementations, each state needed to determine the transition to the next state.

By adding a queue to this state machine, we can streamline it significantly. We can remove the code in each state that specifies the transition to the next state. Instead, a sequence of states is placed on the queue in the *Wait* state whenever a

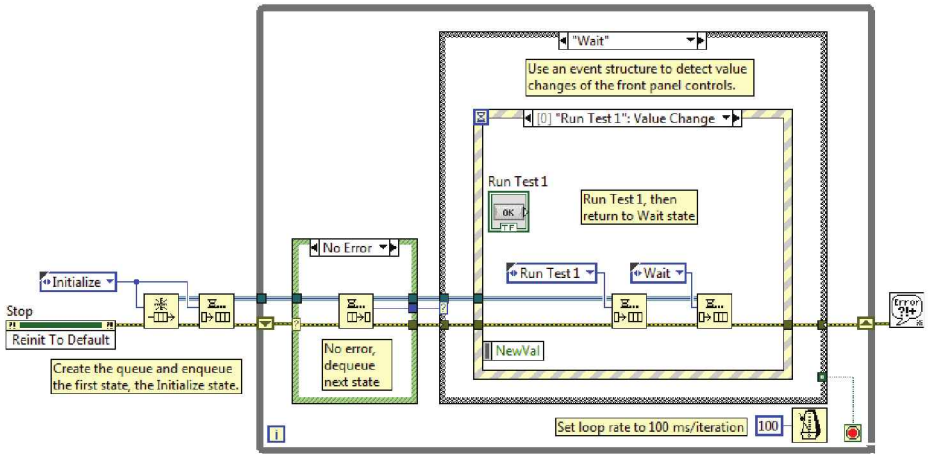


Figure 13.8: This figure shows the event case handling the value change of the RUN TEST 1 button; notice that a series of states are queued, showcasing the advantage of using a queue

button is pressed on the front panel. This is illustrated in Figure 13.8, which shows the event case that handles the value change of the RUN TEST 1 button. The *Run Test 1* state is enqueued first and is followed by the *Wait* state. These states will be dequeued in the same order. First the *Run Test 1* state will execute and then the *Wait* state.

This means that the *Run Test 1* state no longer needs to dictate the transition back to the *Wait* state as it did in the classic state machine. It may seem like a small change, but it is very significant. In the queued state machine, the individual states have no knowledge of either the states that came before them or the states that will come after them. The states are now uncoupled from each other—like beads on a string that can be arranged and rearranged in any order that the developer desires. This decrease in coupling is a huge step forward in maintainability and scalability. States can be added, deleted, or modified with less chance of disturbing the code in other states.

Let's step through the queued implementation of the `Multitest` VI starting with the *Initialize* state shown in Figure 13.9. The queue is created in the input area of the state machine. Queue creation is an example of start-up code, which is usually placed in the input area of the state machine. The type-defined enumerated constant that contains the list of states is used to define the data type of the queue element in the **`Obtain Queue`** function. This allows the queue to store state transitions. The list of states in the enumerated constant is the same used in Chapters 4 and 10.

The *Initialize* state is then placed on the queue using the **`Enqueue Element`** function. This ensures that the first state executed will be the *Initialize* state. The *Initialize* constant is also used to set the data type of the queue in the **`Obtain Queue`** function and as the first element of the queue. Remember, the constant used to create

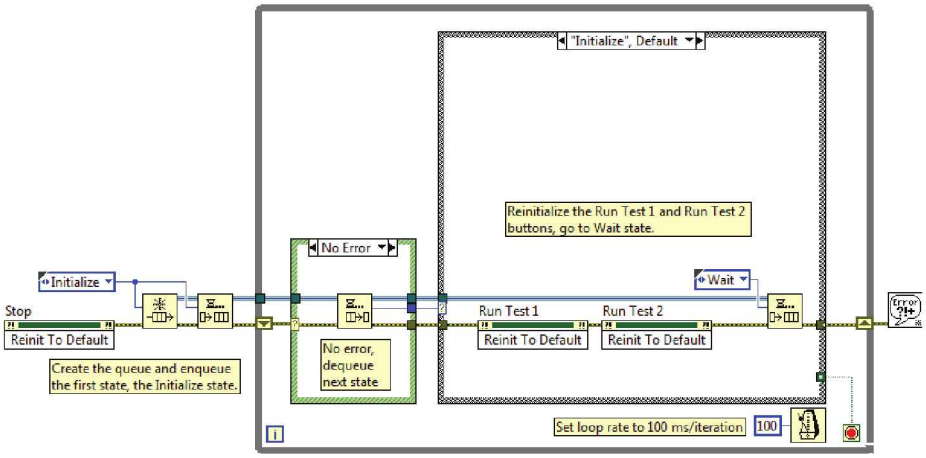


Figure 13.9: The *Initialize* state of the `Multitest` VI

the queue is not placed on the queue. It is only used to determine the data type of the queue.

The queue reference wire and the error wire then enter the main loop. The error output of the **`Enqueue Element`** function is used to initialize the error shift register of the main loop. Now take a closer look at the point where the queue reference wire enters the main loop. Notice that the wire enters the loop through a simple tunnel instead of a shift register. Why doesn't the queue reference wire require a shift register?

Remember that the actual queue does not travel along the wire. If the actual queue did travel on the wire in a by-value fashion, the queue would need a shift register on the main loop. Every time a state was added or removed from the queue, the queue would change, and the changed queue would need to be passed from one loop iteration to the next via a shift register. But queues do not travel along wires in a by-value fashion. They travel by-reference instead. The actual queue is not on the wire, only a reference to the queue. The reference to the queue remains the same, no matter how the queue changes as states are added or removed. The queue reference is a constant, and constants do not need to be placed on shift registers. Constants should be wired into loops through standard input tunnels.

When the queue has been created and initialized, the main loop begins to execute. First the guard clause checks to see if an error is present. If the error line is clear, the next state of the state machine is dequeued, as seen in Figure 13.9. At the beginning of the program, the only state on the queue is *Initialize*, so the state machine will transition to the *Initialize* state where the RUN TEST 1 and RUN TEST 2 buttons are initialized. The *Initialize* state then enqueues the *Wait* state. This *Initialize* state dictates the transition to the next state—just as it did in the classic state machine. Another way to do this would be to place two **`Enqueue Element`** functions outside the main loop of the state machine. Both the *Initialize* and the *Wait* states could be enqueued before the main loop starts. Both methods are functionally equivalent.

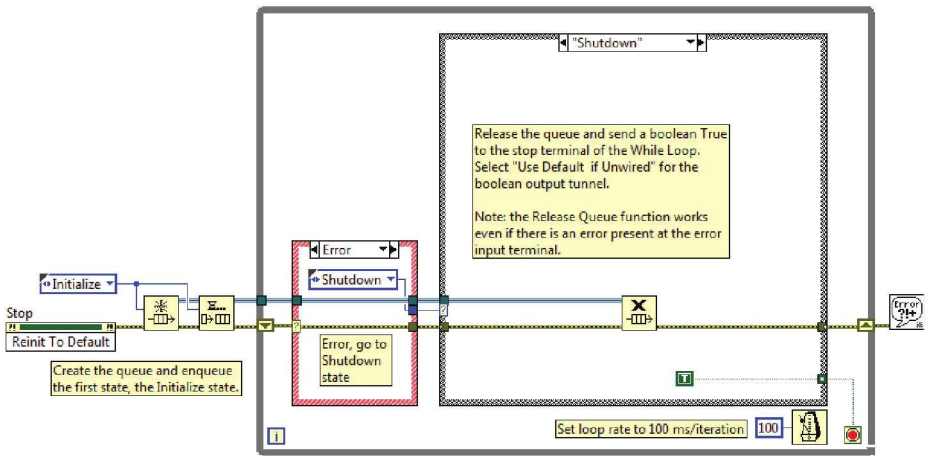


Figure 13.10: The *Shutdown* state of the `Multitest` VI

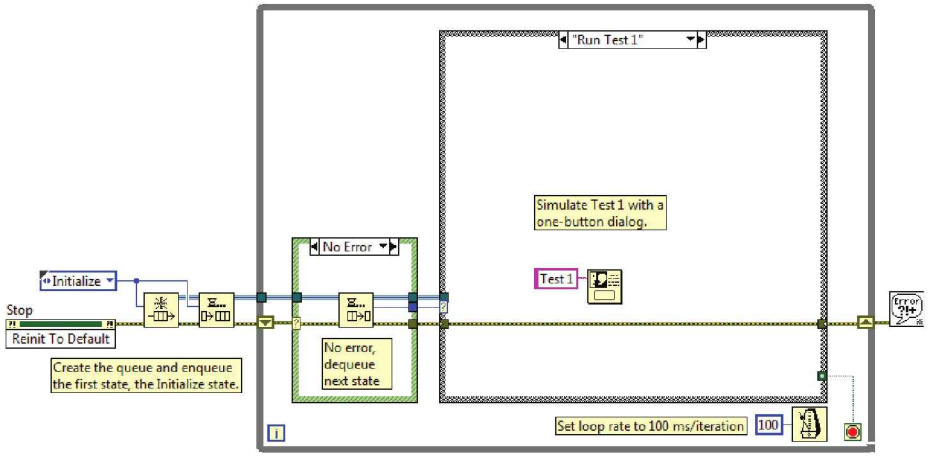


Figure 13.11: This figure shows the *Run Test 1* state; notice that there is no code in this state for determining what state to go to next—this is another benefit of using queues

If the guard clause detects an error, it ignores the elements on the queue and forces a transition to the *Shutdown* state, as shown in Figure 13.10. The *Shutdown* state sends a Boolean True to the loop stop terminal, stopping the loop. The *Shutdown* state also releases the queue. The *Release Queue* function can be placed safely on the main error wire, since it will execute even if an error is present at its error-input terminal.

The heart of the queued state machine is in the *Wait* state. The event case handling the value change of the RUN TEST 1 button was shown in Figure 13.8. In this event case, two states are enqueued: *Run Test 1* and *Wait*. The *Run Test 1* state executes first, as seen in Figure 13.11. The state machine then returns to the *Wait* state. Notice that the

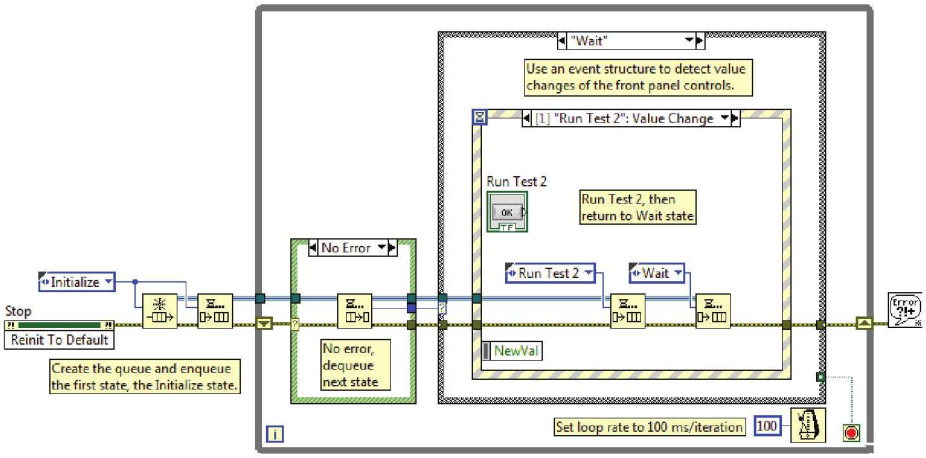


Figure 13.12: This figure shows the event case handling the value change of the RUN TEST 2 button

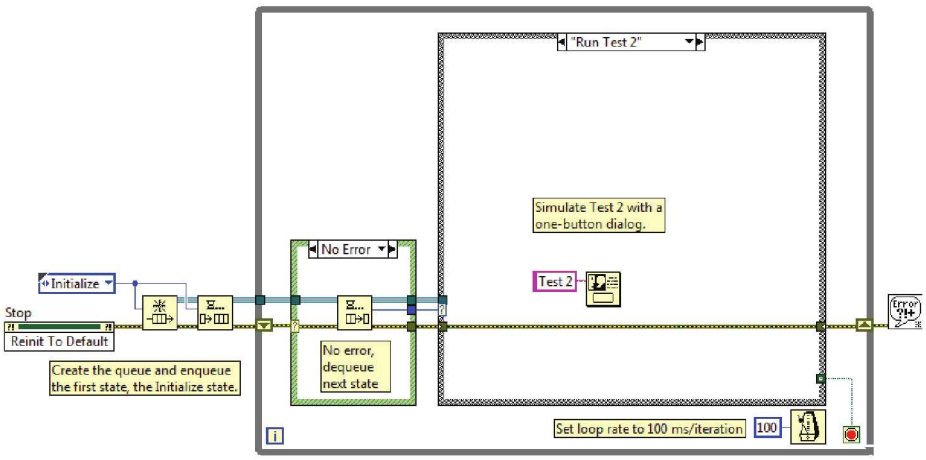


Figure 13.13: The *Run Test 2* state of the `Multitest` VI

Run Test 1 state does not have any code specifying the next state transition. The return to the *Wait* state is dictated by the queue, not by the *Run Test 1* state.

Similarly, the event case handling the value change of the RUN TEST 2 button enqueues the *Run Test 2* state and the *Wait* state, as shown in Figure 13.12. The *Run Test 2* state is shown in Figure 13.13.

Finally, the event case handling the value change of the STOP button is shown in Figure 13.14. The **`Enqueue Element At Opposite End`** function is used to place the *Shutdown* state at the front of the queue for immediate execution. This is a good programming practice in queued state machines, since it makes the code immediately

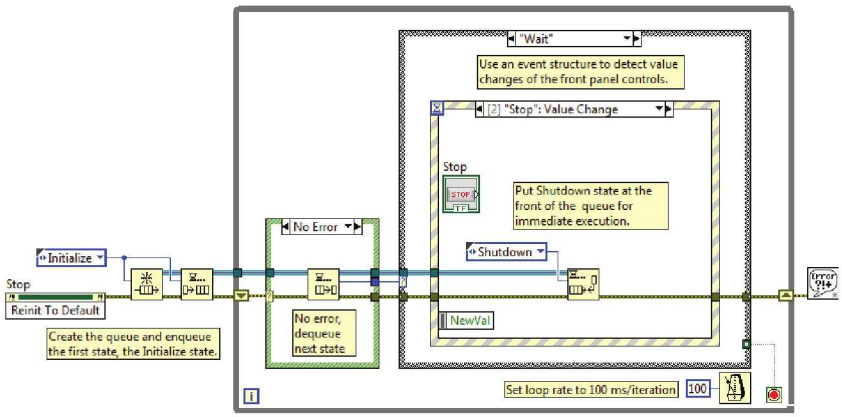
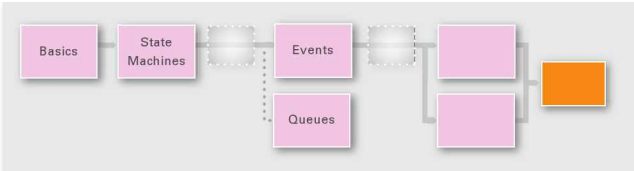


Figure 13.14: This figure shows the event case handling the value change of the STOP button

responsive to a shutdown request. The STOP button has been placed in the event case to ensure that it is read when the case is executed.



14 The Queued Car Wash Controller

Upgrading the event-driven car wash controller to a queued car wash controller represents a major shift in the conception of the program. In Chapter 2, two state transition diagrams were presented for the car wash controller. Figure 2.5 represents the classic state machine approach. Each state leads to the next, and some of the transitions depend on which type of wash was selected.

Figure 2.6 shows the states grouped by wash type. The states are treated as sequences rather than individually. This diagram represents the approach taken by a queued car wash controller. In this implementation of the car wash controller, a pre-determined sequence of events takes place when the user selects a wash type. Once this sequence has been initiated, there is no need to refer to the wash type to determine the state transitions.

This approach is easily implemented using queues. The state machine architecture is still very similar to the event-driven classic state machine developed in Chapters 4 and 10. However, the addition of queues impacts every state, so we will examine this new version of the car wash controller in detail.

Figure 14.1 shows the *Initialize* state of the queued implementation of the car wash controller. The queue is created in the input area of the state machine. The type-defined enumerated constant containing the list of states is used to determine the data type of the queue elements in the **`Obtain Queue`** function. This enumerated constant is based on the same type definition used in Chapters 4 and 10. The *Initialize* state is then enqueued as the first state.

The error wire from the **`Enqueue Element`** function in the input area is used to initialize the error shift register. As discussed in Chapter 13, the queue reference wire enters the main loop through a tunnel instead of a shift register, because the queue reference value is a constant. Inside the main loop, the first function to execute is the `Stop or Error` VI. Recall that the purpose of this VI is to check the STOP button and the status of the error wire. If the STOP button has not been pressed and there is

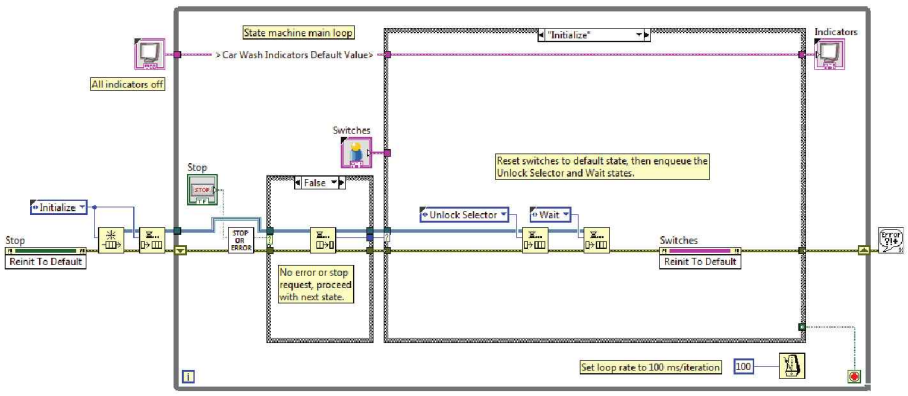


Figure 14.1: Overall structure of a queued event-driven state machine implementation of the car wash controller; notice that the queue reference wire needs to be routed around the `Stop or Error` sub-VI

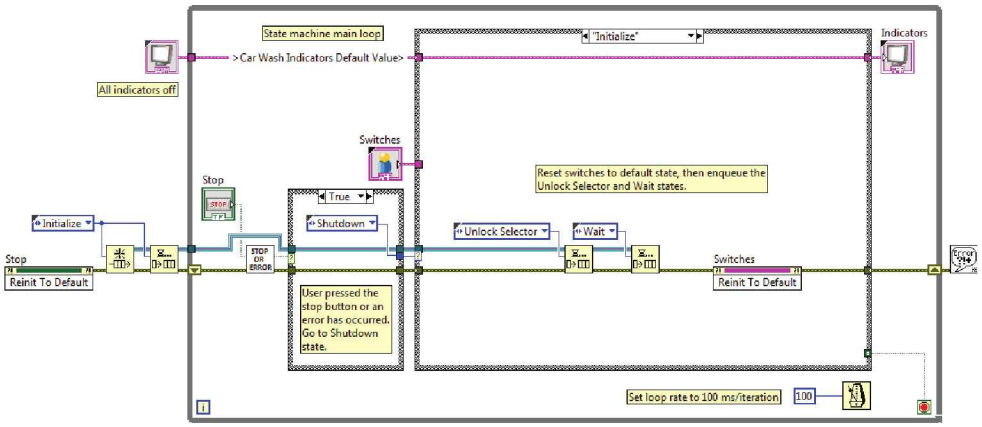


Figure 14.2: The guard clause bypasses the queue and transitions directly to the *Shutdown* state if the user presses the STOP button or if an error is present

no error, the False case of the guard clause is executed, and the next state is dequeued. If an error is present or the STOP button has been pressed, the guard clause bypasses the queue and transitions directly to the *Shutdown* state, as shown in Figure 14.2.

The queue reference wire in these figures is obstructed by the `Stop or Error` sub-VI. The queue reference wire is routed around the sub-VI, but this produces kinks in the reference wire. It is good LabVIEW practice to minimize the number of wire bends on the block diagram to improve readability. This is especially true for error wires and reference wires. These wires often tie together groups of functions with a common purpose, such as file I/O or queue manipulation. Straight references wires keep these functions visually aligned, improving readability.

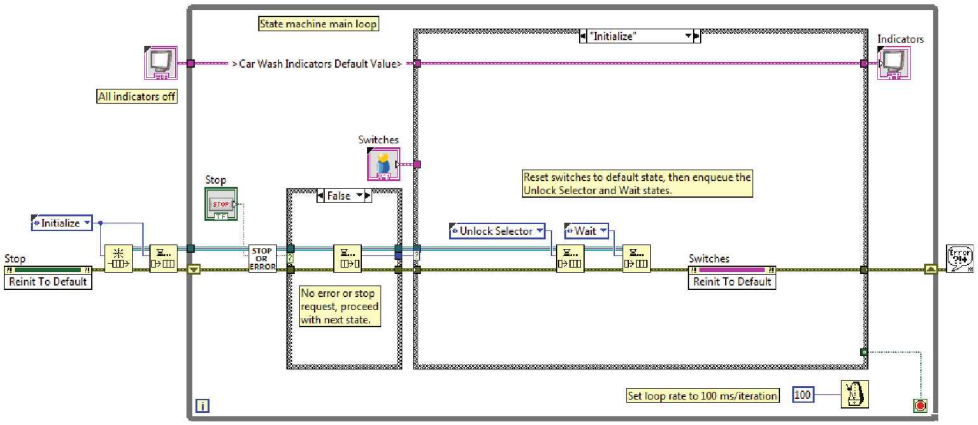


Figure 14.3: Adding queue reference input and output terminals to the `Stop or Error` sub-VI keeps the queue reference wire straight but degrades the cohesion and reusability of the sub-VI

One way to keep the reference wire straight is to add input and output queue reference terminals to the upper-left and -right corners of the `Stop or Error` connector pane, as shown in Figure 14.3. This keeps the queue reference wire straight, but it is undesirable for two reasons. First, the queue reference has nothing to do with the functionality of the `Stop or Error` sub-VI. Adding unneeded inputs to a sub-VI degrades its cohesion. Second, the queue reference input and output are now specific to the data type of the enumerated constant used by the car wash controller. That means that the `Stop or Error` sub-VI could not be used in other programs unless those programs used the exact same queue element as the car wash controller. This degrades code reusability.

14.1 Building a Non-Square Icon

One way to keep the queue reference wire straight and still maintain cohesion and code reusability is to use a non-square icon for the `Stop or Error` sub-VI. If the sub-VI had a shorter icon like the one shown in Figure 14.4, the queue reference wire could pass above the sub-VI without kinks.

To change the icon of the `Stop or Error` sub-VI, right-click on the icon and bring up the icon editor shown in Figure 14.5. The icon editor also can be brought up by double-clicking on the icon you wish to edit. After clearing the old icon layers with *Edit...Clear All*, draw a rectangle that only encloses the desired area using the drawing tools at the right side of the editor. The icon terminals can be made visible



Figure 14.4: Icon for the non-square `Stop or Error` sub-VI

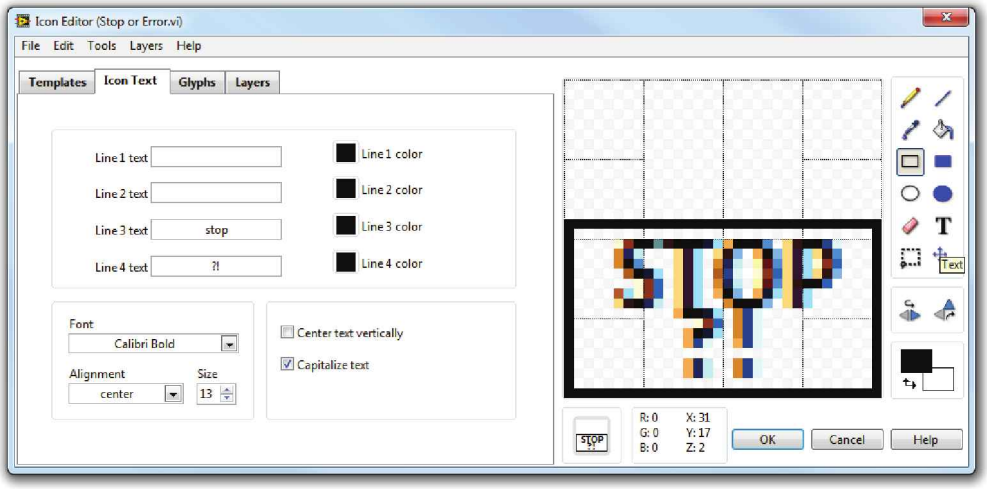


Figure 14.5: Building a non-square icon

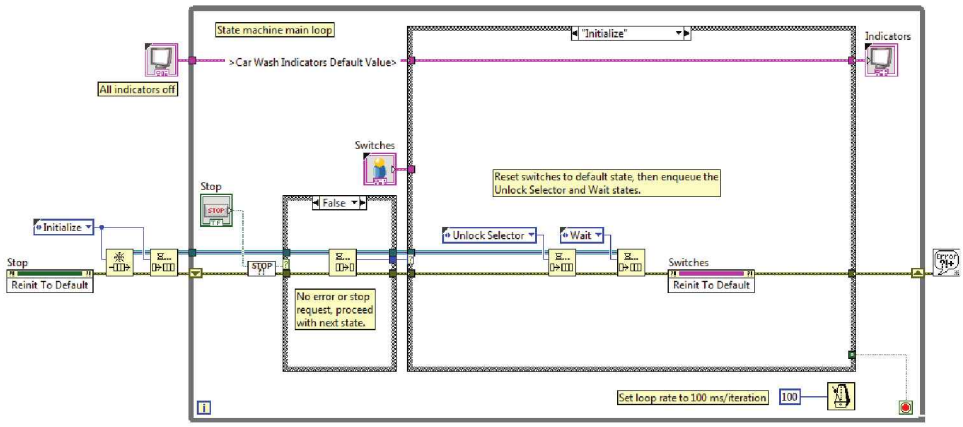


Figure 14.6: The *Initialize* state of the queued car wash controller using the non-square `Stop or Error` icon

in the editor by selecting *Edit...Show Terminals*. Figure 14.5 also shows how the icon text can be inserted using the *Icon Text* menu of the icon editor. Make sure that the *Center Text Vertically* option is unchecked, or the text will spill outside the bounding rectangle of a short icon. When the icon is finished, wire the connector pane as usual using only the terminals in the bounding box of the icon.

The impact of the new version of the `Stop or Error` sub-VI can be seen in Figure 14.6. The queue reference wire is now straight and passes above the `Stop or Error` sub-VI. Notice how the queue reference wire runs like a visual backbone from

left to right through the middle of the block diagram, increasing the readability of the program.

This may seem like a small thing, but good readability is made up of many small touches such as this one. Good style is important in any programming language, but it is especially important in LabVIEW because of its graphical nature. A good LabVIEW programmer should always take steps to make a VI as clean, clear, and readable as possible. In this case, the `Stop or Error` sub-VI has a good chance of being reused in other state machine codes, so any effort made now to improve readability will likely benefit future projects.

14.2 Examining the Queued Car Wash Controller

Now that we've straightened the reference wire, let's examine the *Initialize* state itself. This state initializes the proximity switches just as it did in the classic state machine implementation. This state also dictates transitions to the *Unlock Selector* state followed by the *Wait* state.

The *Unlock Selector* state is shown in Figure 14.7. The purchase selector is initialized and enabled in this state. No code is needed to dictate the transition to the next state, since the next state is already waiting on the queue.

The *Initialize* and *Unlock Selector* states illustrate the utility of queues. Multiple states can be enqueued at one time, and individual states no longer need to dictate transitions to the next state.

Queues and Event Structures

The real power of queues is found in the *Wait* state. This state uses an event structure to monitor the user interface described in Chapter 11, but the code in the various

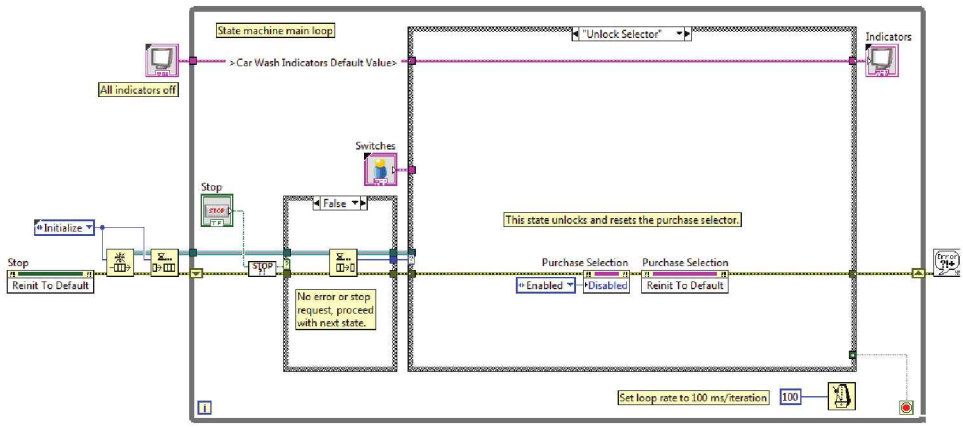


Figure 14.7: The *Unlock Selector* state of the queued car wash controller

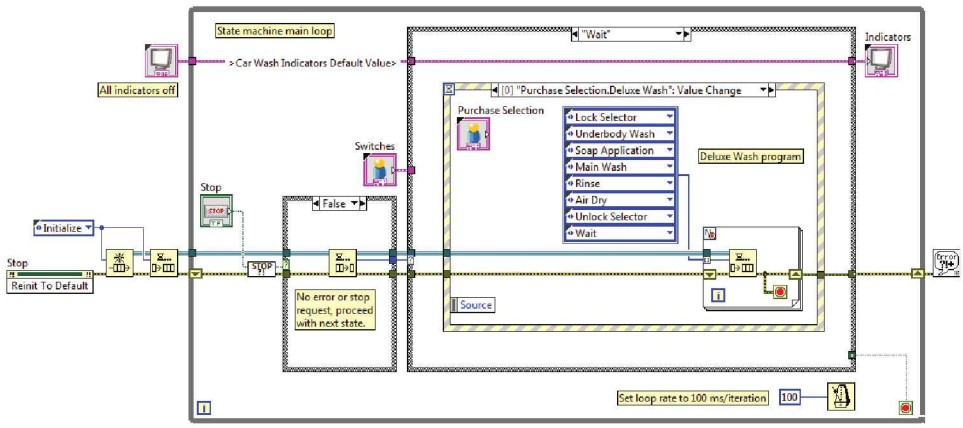


Figure 14.8: The event case handling the value change of the DELUXE WASH button, where the entire sequence of wash cycles for the Deluxe Wash are loaded onto the queue in this event case

event cases now uses queues to specify sequences of states instead of transitions to individual states.

The event case handling the value change of the DELUXE WASH button is shown in Figure 14.8. In this event case, the entire sequence of states for a Deluxe wash is placed on the queue. The states are first placed in an array, and the array is then fed into a **`For`** loop. Nothing has been wired to the *N* terminal of the loop to specify the number of times the loop is to be run—auto-indexing is used to determine the number of loop iterations. The loop will run a number of iterations equal to the length of the array, passing the elements of the array into the loop one element per loop iteration. The individual states in the array will be placed on the queue in order from the first array element to the last. The eight states needed for a Deluxe wash are enqueued using one **`Enqueue Element`** function in a **`For`** loop. This is a more efficient use of screen space than wiring eight **`Enqueue Element`** functions in series.

> **Style Guide:** Try to make your programs fit on one screen (if possible) to reduce the need for navigation and improve readability. If you can't fit the program in one screen, arrange your code so that scrolling is only needed in one direction.

The **`For`** loop also has a conditional terminal, which is an option discussed in Chapter 1. If a conditional terminal is used, the **`For`** loop executes specified number of iterations unless it is aborted by the stop terminal. In Figure 14.8, the error wire from the **`Enqueue Element`** function is wired directly to the stop terminal. The status Boolean of the error wire is is checked each loop iteration to determine whether the loop should continue or not.

Notice too that the error wire in the **`For`** loop has been placed on a shift register. This is good programming practice, because it ensures that any warnings generated

in the loop are trapped and passed on. If an error is generated inside the loop, the conditional terminal causes the loop to abort, but warnings do not cause the loop to abort. Warnings have error codes, but the status Boolean in the error cluster remains False. The error shift register is necessary to trap any warnings that are generated and to pass them on once the loop has ended.

The queue reference wire enters the **For** loop only once and is not put on a shift register. This is because queues are passed by reference instead of by value. Even though the elements of the queue change throughout the course of the loop, the reference to the queue does not change. Therefore it doesn't need to be placed on a shift register.

Handling the Purchase Selector and the Stop button

Now let's examine the placement of the purchase selector. The terminal for the purchase selector cluster has been placed in the DELUXE WASH value change case even though it is not wired to anything. Why was this done? Recall from Chapter 10 that it is good programming practice to put a control within the event case that handles its value change if possible. This ensures that LabVIEW reads the control when it is pressed and that the mechanical action of the control executes as expected.

The individual DELUXE WASH and ECONOMY WASH buttons have mechanical actions of *Switch When Pressed*. Since the controls are switch-action oriented rather than latch-action, LabVIEW does not need to read the buttons to ensure proper execution of their mechanical actions. This wasn't important in the classic and event-driven implementations of the car wash controller, but it is important here. The reason for this is that, in the classic and event-driven implementations, a single event case was configured to wait for the value change of the entire purchase selection cluster—not the individual controls in the cluster. The mechanical action of the Boolean controls didn't matter.

In the queued car wash controller, the value changes of the DELUXE WASH button and of the ECONOMY WASH button are handled in different event cases. Look back at Figure 11.1. Notice that the event case is looking for a value change of the entire cluster, considering <*All Elements*> of the cluster. Compare this to Figure 14.9, where the event case is configured to handle the value change of just the DELUXE WASH button. Instead of looking for the value change of the cluster as a whole, we are now looking for the value change of one element of the cluster. This is necessary because we need to enqueue different sets of states depending on which wash was selected.

So how does this relate to the placement of the purchase selector and the mechanical action of the wash buttons? In the queued implementation of the car wash controller, we have separate event cases for each button in the purchase selector, but there is only one terminal for the purchase selector itself. We can place the selector in either the DELUXE WASH value change case or the ECONOMY WASH value change case, but not in both of them. If the buttons in the purchase selector have a latching action, the behavior of the buttons will be different from each other. The button that has the purchase selector in its event case will be read when the button is pressed, but other button will not be read when it is pressed. One way around this difficulty is to use switching action rather than latching action on the buttons. Since the program already

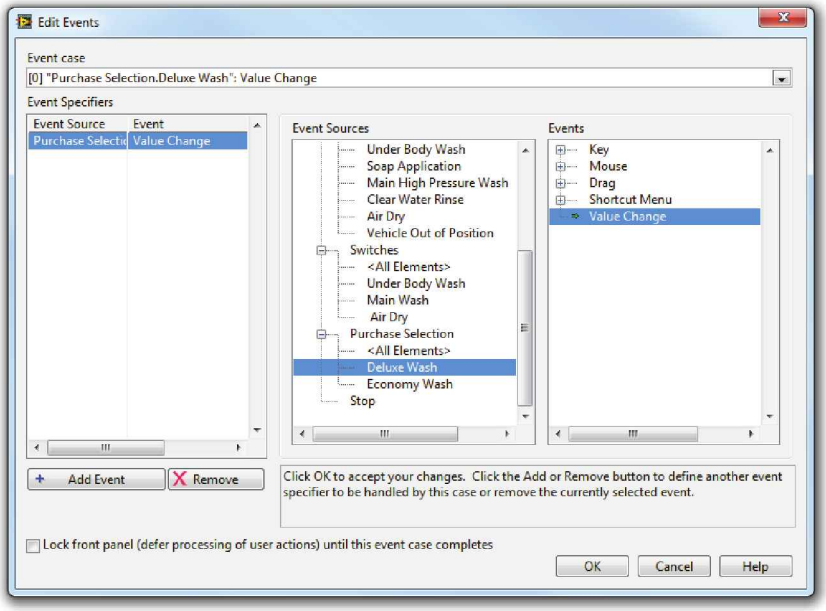


Figure 14.9: The event configuration window for the value change of the DELUXE WASH button

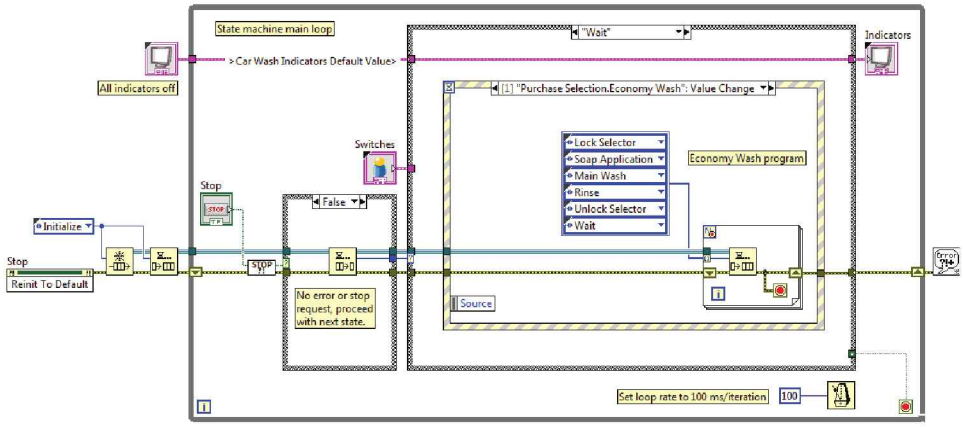


Figure 14.10: The event case handling the value change of the ECONOMY WASH button, where the entire sequence of wash cycles for the Economy wash are loaded onto the queue

reinitializes the purchase selector when the selector is unlocked at the end of the wash, the buttons will be returned to their default positions when a wash is completed.

The event case handling the value change of the ECONOMY WASH button is shown in Figure 14.10. This event case is very similar to the event case handling the value change of the DELUXE WASH button. The only difference is in which states are queued.

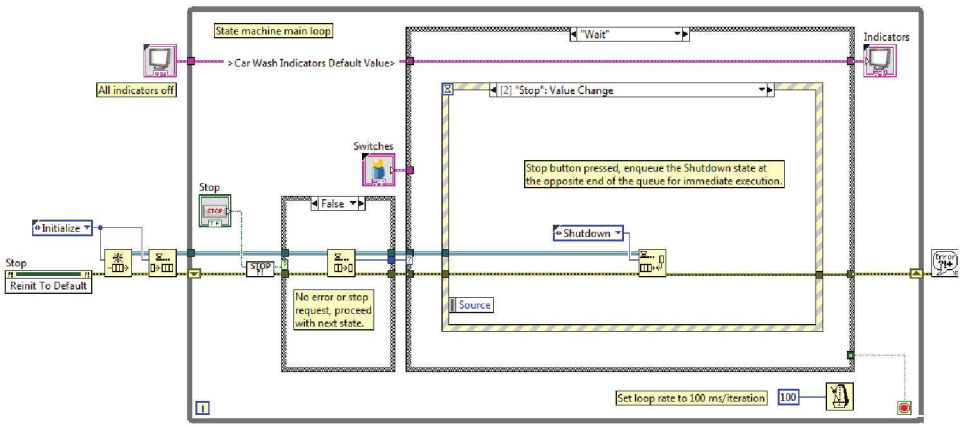


Figure 14.11: The event case handling the value change of the STOP button

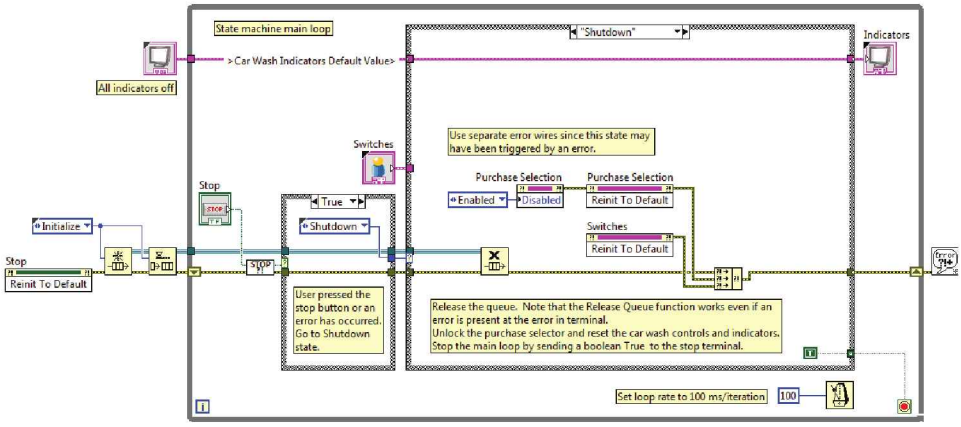


Figure 14.12: The *Shutdown* state of the queued car wash controller

The event case handling the value change of the STOP button is shown in Figure 14.11. This state enqueues the *Shutdown* state if the STOP button is pressed. The **`Enqueue Element At Opposite End`** function is used to place the *Shutdown* state at the front of the queue for immediate execution.

The *Shutdown* state is shown in Figure 14.12. The *Shutdown* state stops the main loop and reinitializes the front panel controls. This state also releases the queue reference. The queue will not be needed after the VI shuts down, and it is good programming practice to release system resources when a VI exits.

The **`Release Queue`** function is placed on the main error wire, but the property nodes are placed on parallel error wires. This is because the **`Release Queue`** function operates normally if there is an error present at its *error in* terminal, but property nodes will not execute if an error is present. This parallel wiring arrangement ensures that

all of the functions in the *Shutdown* state execute even when an error is present on the main error wire. The parallel error wires are then merged back into the main error wire using the **`Merge Errors`** function.

14.3 The Car Wash States

When a wash type has been selected, the first state executed is the *Lock Selector* state shown in Figure 14.13. In the classic and event-driven implementations, the *Lock Selector* state needs to know which wash type is selected in order to choose the correct state transition. This is no longer the case. The *Lock Selector* simply executes and depends on the queue for determining the next state transition. The DELUXE WASH

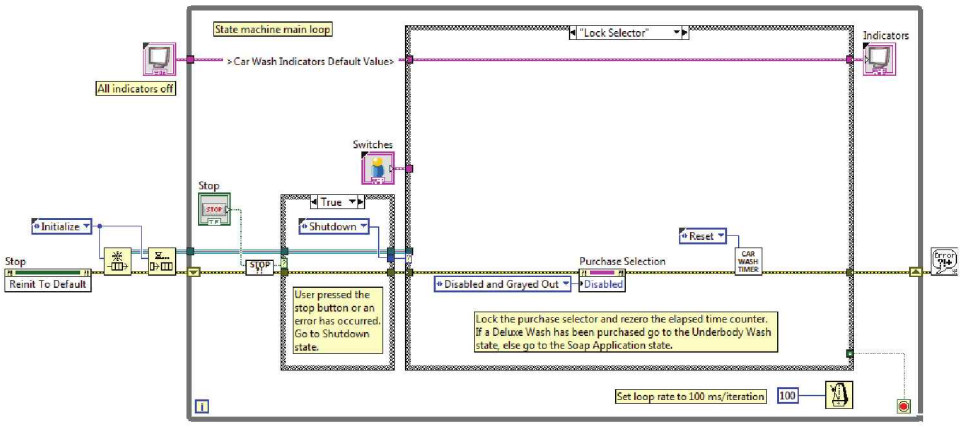


Figure 14.13: The *Lock Selector* state of the queued car wash controller

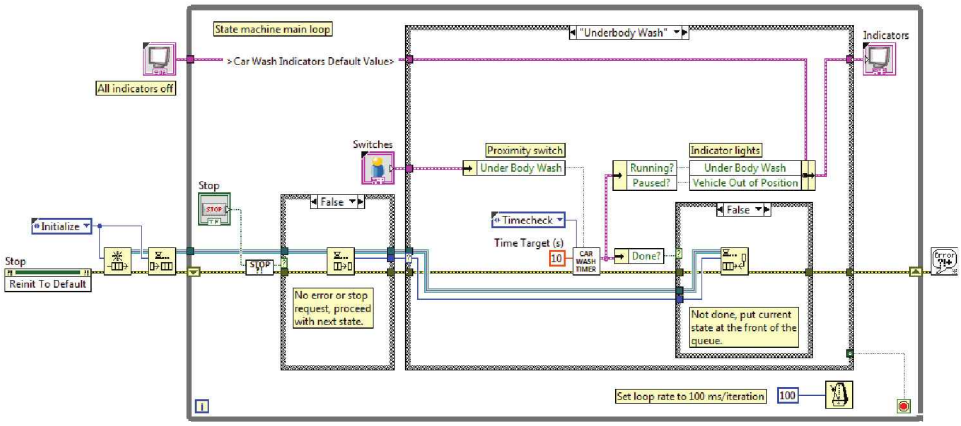


Figure 14.14: The *Underbody Wash* state of the queued car wash controller, where if the cycle is not done, the *Underbody Wash* state is placed at the front of the queue for immediate execution

Boolean is removed from the data highway, since it is no longer needed. The car wash states are now totally decoupled from the type of wash selected.

The car wash cycle states are also affected by the addition of a queue to the car wash controller. In the classic state machine and event-driven state machine car wash controllers, each car wash state needs to know which state to go to next. They have to decide whether to return to the current state or to move on to the next state, depending on whether the target time has elapsed. If it is time to move on to the next state, each car wash cycle state needs to know which cycle came next. Sometimes this required knowledge of which car wash type is chosen.

In the queued implementation of the car wash controller, the cycle states don't need to know the next state or the wash type, but they still need to decide whether or not to repeat the current state. Figure 14.14 shows the *Underbody Wash* state. The car wash timer action engine is used to determine if the cycle is finished or not.

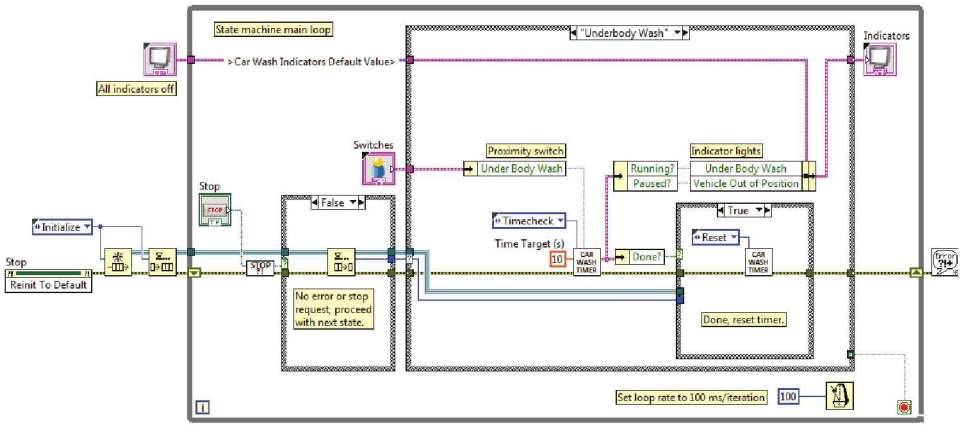


Figure 14.15: The *Underbody Wash* state of the queued car wash controller, where if the cycle is done, the timer is rezeroed and no operations are performed on the queue

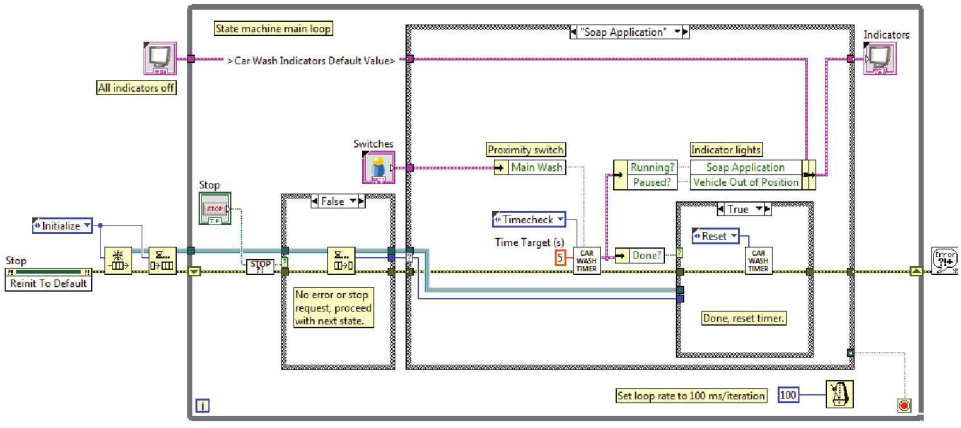


Figure 14.16: The *Soap Application* state of the queued car wash controller

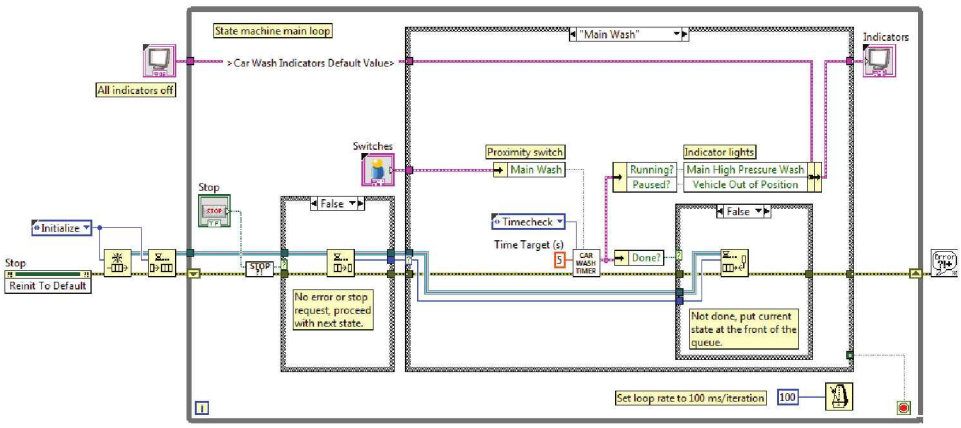


Figure 14.17: The *Main Wash* state of the queued car wash controller

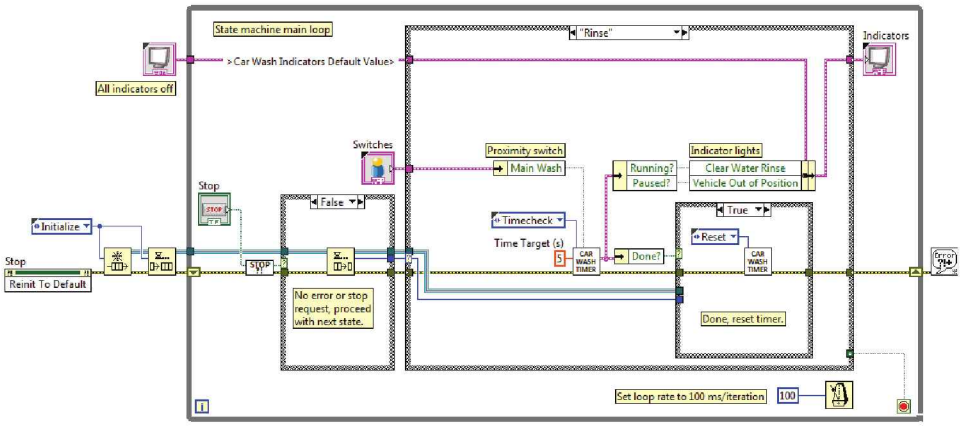


Figure 14.18: The *Rinse* state of the queued car wash controller, where if the cycle is done, the timer is rezeroed and the car wash controller moves to the next wash state

If the cycle is not finished, the current state is placed back on the front of the queue using the **`Enqueue At Opposite End`** function. If the current cycle is finished, the timer is reset, but no operations are performed on the queue, as shown in Figure 14.15. The next state of the wash sequence is then dequeued when the loop iterates again.

The other wash cycle states are all similar; only the unbundled proximity switches, value of the time targets, and bundled cycle indicators change. All of the wash states now have identical structures. All of the wash states have become independent of each other, and all are independent of the wash cycle chosen. Each wash cycle state exists on its own—unaware that other cycle states even exist. The *Soap Application*

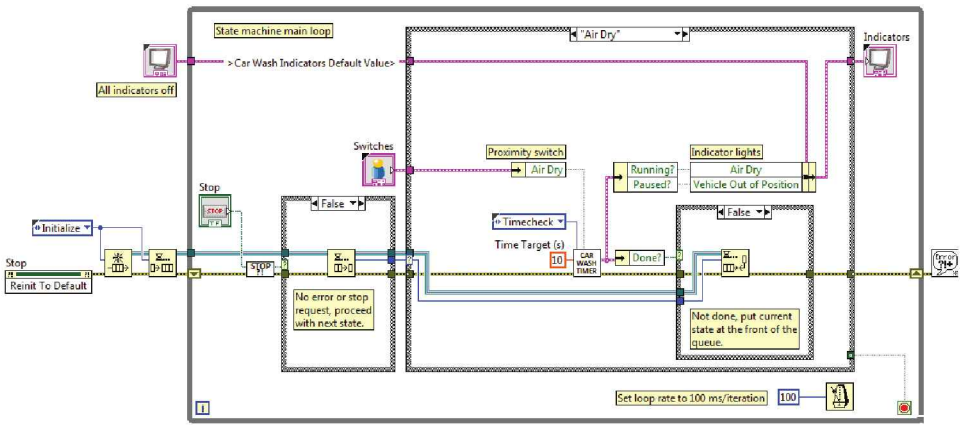
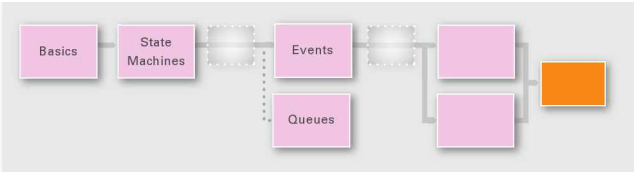


Figure 14.19: The *Air Dry* state of the queued car wash controller

state is shown in Figure 14.16, the *Main Wash* cycle in Figure 14.17, the *Rinse* cycle in Figure 14.18, and the *Air Dry* cycle in Figure 14.19.



15

Data Acquisition with the Multitest VI

This book is focused on LabVIEW as a programming language rather than as a data acquisition tool. But since LabVIEW is extensively used for data acquisition, it hardly seems fair to ignore the subject entirely. This chapter demonstrates how to write a data acquisition VI using the `Multitest` VI as a template. The example VI allows a user to null, calibrate, and read a strain gage connected to a quarter-bridge Wheatstone bridge.

15.1 Strain Gages and Bridge Circuits

Strain is a measure of change of shape and of deformation from a given reference state. Strain is usually expressed as the ratio of a change in length to a reference length. For example, Figure 15.1 shows a bar under an axial load. Suppose that the original length of the bar is l_0, and the bar stretches to a length $l_0 + \Delta l$ when a load is applied. The strain ε is equal to $\Delta l / l_0$. Strain is dimensionless, since it is a ratio of lengths.

Strain gages are essentially resistors bonded to a surface where strain is to be measured. When the surface deforms due to strain, the strain gage also deforms, changing its resistance. The ratio of the change in resistance to the original resistance ($\Delta R / R_0$) is proportional to the strain. This factor of proportionality is known as the gage factor. If the gage factor is known and the change in resistance of the strain gage can be measured, the strain at the surface beneath the strain gage can be calculated.

Strain gages are usually used to measure small strains (often smaller than 0.1%). This means that the change in resistance of a strain gage is also small. Small changes in resistance are difficult to measure directly, so a special circuit known as a bridge circuit is used to convert small changes in resistance to small changes in voltage which

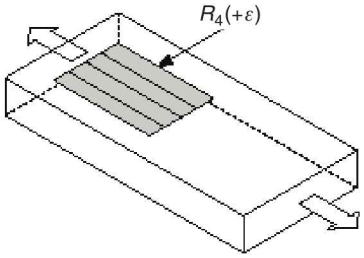


Figure 15.1: Measuring axial strain with a single strain gage

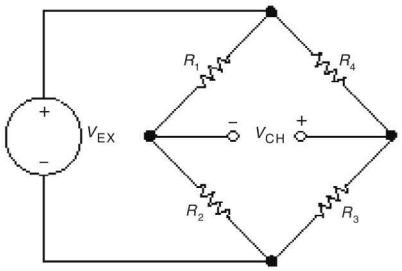


Figure 15.2: Bridge circuit for measuring strain

are more easily measured. A typical bridge circuit, known as a Wheatstone bridge, is shown in Figure 15.2.

Notice in Figure 15.1 that the strain gage on the bar is designated as R_4 and that one of the legs of the bridge circuit in Figure 15.2 is also designated as R_4. The strain gage is wired into the bridge circuit in the R_4 location. The other three resistors are either fixed resistors of known values or variable resistors (potentiometers). This is called a quarter-bridge configuration, since the strain gage makes up one quarter of the bridge configuration. The value of V_{CH} is the bridge output voltage, and V_{EX} is the excitation voltage. If the bridge circuit is correctly nulled and calibrated, the ratio of V_{CH}/V_{EX} will be proportional to the ratio of $(\Delta R_4/R_4)$ and thus proportional to the strain.

The strain reading is accurate only if the bridge circuit is properly nulled and calibrated. Typically, at least one of the non-gage resistors in the bridge is a potentiometer whose resistance can be varied over a range of values. When no strain is applied to the gage, the bridge output should be zero. Nulling the bridge means varying the potentiometer resistance when the gage is unstrained until the bridge output is zero.

Calibrating the bridge consists of placing a shunt resistor of known value in parallel with the gage resistor. This changes the resistance of the the leg of the bridge containing the strain gage and simulates imposing a known strain on the test specimen. The bridge output is then read and the strain calculated. The calculated strain is compared to the known value of the simulated strain to calibrate the bridge output.

15.2 Data Acquisition in LabVIEW

Data acquisition (DAQ) in LabVIEW is performed using the DAQmx functions found in the *Measurement I/O* sub-palette of the *Functions* palette. The DAQmx functions are very similar to the file I/O functions. The file I/O functions operate on a file reference. When a file is opened, a file reference is created, and that reference must be passed as an input to other LabVIEW functions that read from and write to the file. Similarly, the DAQmx functions operate on a DAQ task reference. To perform a data acquisition task in LabVIEW, a DAQ task reference first must be created. Channels are then added to the task to represent the different measurements that need to be made. The strain measurement described in this chapter is represented by a strain channel added to the DAQ task. Timing and triggering data also can be added to a task. A single task can have many channels, as long as the channels all share the same timing and triggering requirements as well as the same directionality (input or output). Once the task has been defined, the task reference can be passed to other DAQmx functions that perform other functions, such as nulling, shunt calibration, and data acquisition.

15.3 Functional Specification

Devices that acquire data from strain gages often include internal signal conditioning circuitry that provides the Wheatstone bridge circuit. These devices also may have internal circuitry for nulling and calibrating the bridge that can be enabled by software commands. In this chapter, we assume that the user wishes to measure the output of a strain gage in a quarter-bridge configuration using data acquisition hardware with software-enabled bridge nulling and calibration capabilities.

The software should allow the user to configure the measurement by entering data into a cluster on the front panel. The user should be able to null the bridge or calibrate the bridge at any time. When the user wishes to read the gage, a burst of readings is made, and the average result is displayed on the front panel. The number of readings in the burst and the sampling rate is set by the user in the configuration cluster. The user is able to change any of the values in the configuration cluster while the program is running. The configuration cluster contains a significant amount of user-entered information, so it should never be reinitialized to default by the program. The front panel of the VI looks something like the one shown in Figure 15.3.

15.4 State Transition Diagram

One possible state transition diagram for this functional specification is shown in Figure 15.4. When the program starts, it initializes the front panel, and then builds the DAQ task to prepare for taking the data. The program will then wait. The user can then choose to null the bridge, perform shunt calibration, or read the strain gage.

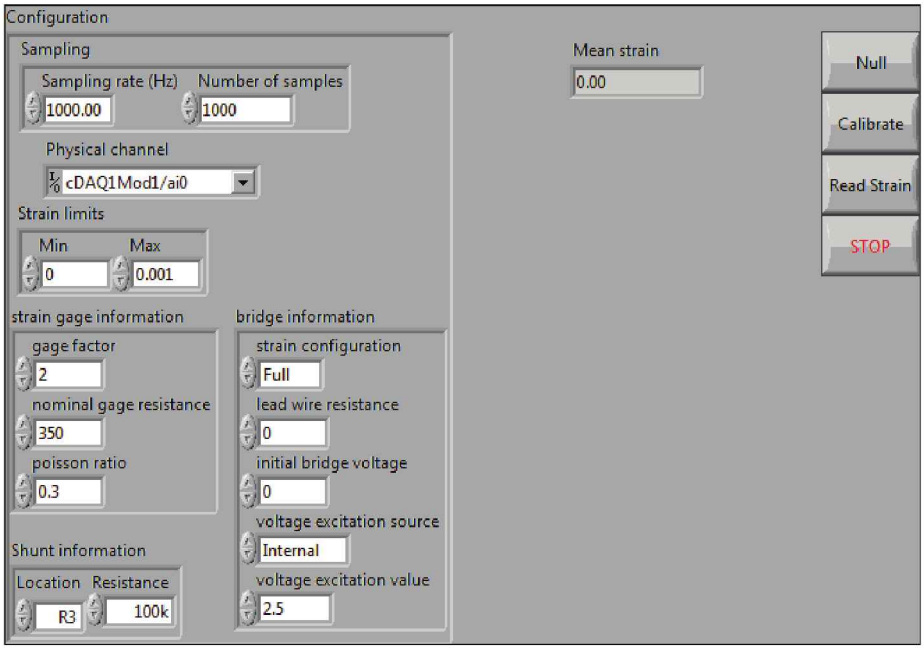


Figure 15.3: Front panel of the `Multitest` data acquisition VI

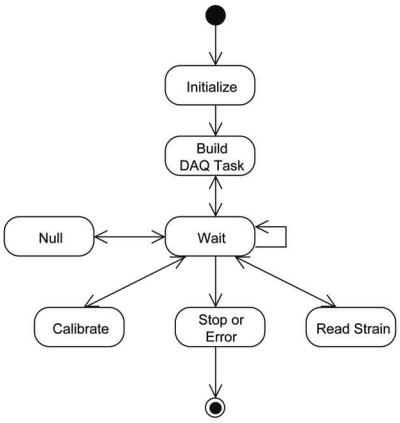


Figure 15.4: State diagram for the `Multitest` data acquisition VI

The state machine returns to the *Wait* state after each of these actions. The program ends if the user presses the STOP button or if an error occurs.

Notice that the program can also revisit the *Build DAQ Task* state from the *Wait* state. The functional specification states that the user is able to change the configuration parameters while the program is running. The DAQ task needs to be rebuilt if the parameters change, so the state transition from the *Wait* state to the *Build DAQ Task*

state and back again is necessary to fulfill the requirements of the functional specification. This transition occurs whenever the configuration cluster changes.

Notice the similarity between this state transition diagram and the one for the `Multitest` VI shown in Figure 2.2. The strain measurement VI is essentially the same as the `Multitest` VI with three tests: *Null*, *Calibrate*, and *Read Strain*. This gives us a plan of attack. The strain measurement VI is based on the `Multitest` framework. We use the queued-string version of the `Multitest` VI developed in Chapter 21 as the basis for the strain measurement VI.

15.5 Building the DAQ Task

Before developing the main VI, we need to look at the specific tasks involving the strain gage. In order to null the bridge, perform shunt calibration, or read the strain, we first need to build a DAQ task. Building a DAQ task requires three steps:

1. Create a task reference.
2. Add channels for the various measurements.
3. Configure the timing of the task.

The DAQ task is created using the **`DAQmx Create Task`** function. The icon and connector pane of this native LabVIEW function is shown in Figure 15.5. The strain channel is added to the DAQ task using the native **`DAQmx Create Channel`** function shown in Figure 15.6. This function configures the physical channel of the data acquisition system, specifies the details of the strain gage used, and also specifies the configuration of the Wheatstone bridge. The timing information is then added to

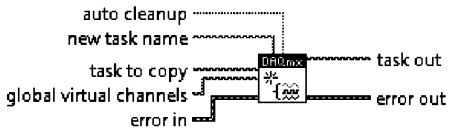


Figure 15.5: Icon of the **`DAQmx Create Task`** VI

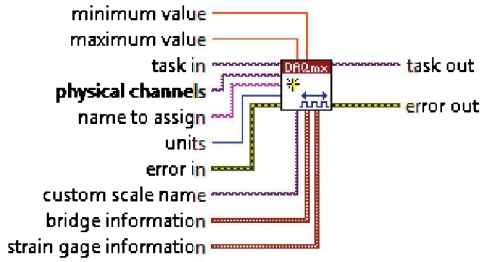


Figure 15.6: Icon of the **`DAQmx Create Channel`** VI

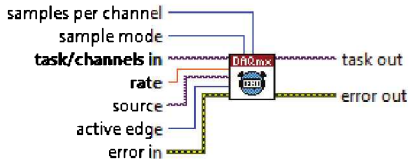


Figure 15.7: Icon of the `DAQmx Timing` VI



Figure 15.8: Icon of the `DAQmx Clear Task` VI

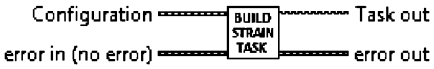


Figure 15.9: Icon of the `Build Strain Task` sub-VI

the task using the native **`DAQmx Timing`** function shown in Figure 15.7. This function specifies the number of points to be read in each burst of data and the sampling rate of the burst. When it is no longer needed, an existing DAQ task is cleared by the native **`DAQmx Clear Task`** function shown in Figure 15.8.

Wiring Tip: If the DAQmx functions do not appear in your function palette, download the DAQmx drivers from `www.ni.com`.

These three DAQmx functions are used to build the DAQ task in the `Build Strain Task` sub-VI. The icon for this sub-VI is shown in Figure 15.9, and the block diagram is shown in Figure 15.10. This sub-VI accepts the type-defined configuration cluster shown in Figure 15.3 as an input. This configuration cluster is created by combining all of the data needed to configure the DAQ task into one cluster, allowing all of the configuration data to be passed on a single wire. The data is organized into sub-clusters to group related items together: strain gage data, bridge configuration data, etc. The `Build Strain Task` unbundles the data as necessary and uses it to perform the three tasks listed: create a new task, configure a strain channel, and add timing information.

The icons of the **`DAQmx Create Channel`** VI and the **`DAQmx Timing`** VI look slightly different in Figure 15.10 than they do in Figures 15.6 and 15.7. These VIs are polymorphic; the versions used in the VI must be chosen when they are dropped on the block diagram. The version of the **`DAQmx Create Channel`** shown in Figure 15.10 is configured to perform an analog input reading from a strain gage. The version of the **`DAQmx Timing`** VI shown in Figure 15.10 configures the sample clock.

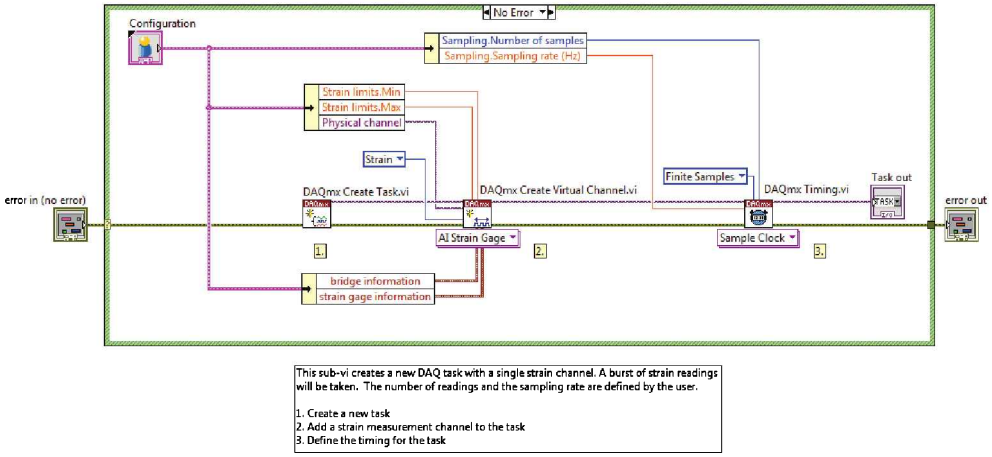


Figure 15.10: Block diagram of the `Build Strain Task` sub-VI

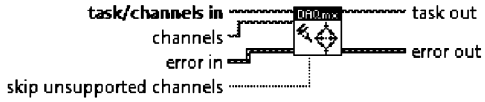


Figure 15.11: Icon for the **`DAQmx Perform Bridge Offset Nulling Calibration`** function

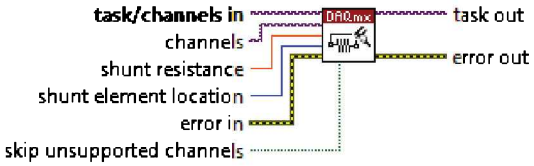


Figure 15.12: Help file for the **`DAQmx Perform Shunt Calibration (Strain)`** function

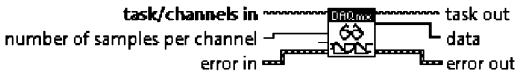


Figure 15.13: Help file for the **`DAQmx Read`** function

The output of this sub-VI is a DAQ task that is used to null the bridge, perform shunt calibration, and read the strain gage. The bridge is nulled using the native **`DAQmx Perform Bridge Offset Nulling Calibration`** function shown in Figure 15.11. The calibration is performed by the native **`DAQmx Perform Shunt Calibration (Strain)`** VI shown in Figure 15.12. The burst of strain gage readings is performed by the native **`DAQmx Read`** VI shown in Figure 15.13.

15.6 Developing the `Multitest` Data Acquisition VI

We now adapt the `Multitest` VI to the strain measurement task at hand. The front panel has been developed in Figure 15.3, and the states and transitions are shown in Figure 15.4. The groundwork for configuring the data acquisition has been laid in Figure 15.10.

The next task is to consider the placement of the various front panel controls and indicators on the block diagram. The NULL, CALIBRATE, READ STRAIN, and STOP buttons are all placed in the event cases that handle their value changes. But where do we place the configuration cluster? Chapter 1 included a discussion of this very question. The answer depends on the requirements laid out in the functional specification. The specification states that the user be able to change the configuration data while the program is running. This means that the configuration cluster needs to be placed inside the main loop. The configuration cluster is needed by both the *Calibrate* and *Read Strain* states, so it should be placed in the lobby of the state machine.

We can now develop the individual states of the `Multitest` VI. The overall structure of the state machine and the *Initialize* state are shown in Figure 15.14. The queue is created in the input area of the main loop. The queue element is an enumerated constant containing the states of the DAQ state machine: *Initialize*, *Build DAQ Task*, *Wait*, *Null*, *Calibrate*, *Read Strain*, and *Shutdown*. The *Initialize* state is enqueued in the input area of the loop.

Inside the main loop, the guard clause checks to see if the error line is clear. If there is no error, the next state is dequeued. If an error is detected, the queue is bypassed, and the state machine is sent to the *Shutdown* state, as shown in Figure 15.15. The *Shutdown* state releases the queue, clears the DAQ task, and stops the main loop.

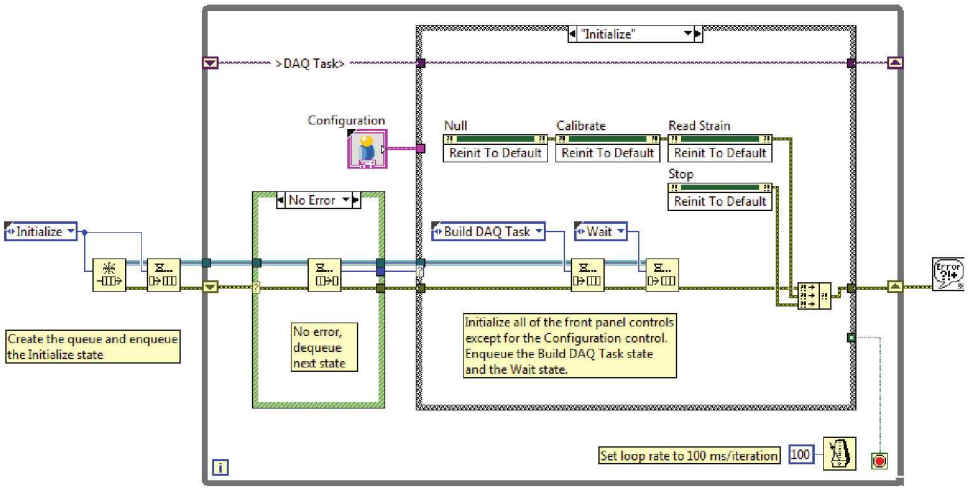


Figure 15.14: The *Initialize* state of the `Multitest` data acquisition VI

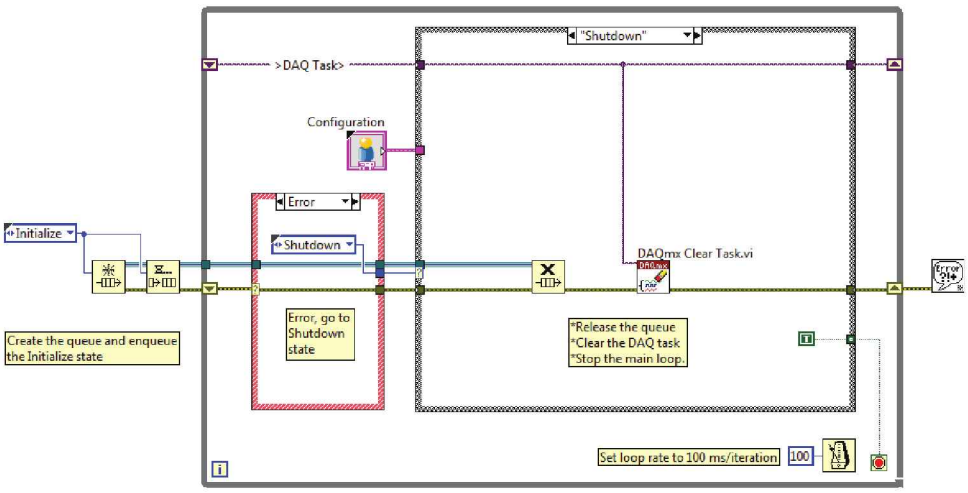


Figure 15.15: The *Shutdown* state of the `Multitest` data acquisition VI

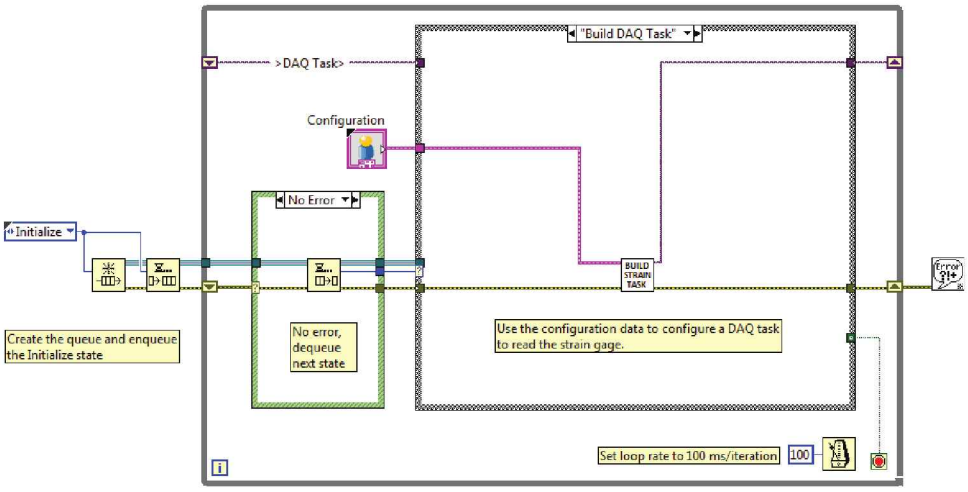


Figure 15.16: The *Build DAQ Task* state of the `Multitest` data acquisition VI

The *Initialize* state shown in Figure 15.14 reinitializes the front panel controls to their default positions—except for the configuration cluster. The *Initialize* state also enqueues the *Build DAQ Task* state and the *Wait* state (in that order).

After the *Initialize* state executes, the DAQ task is built in the *Build DAQ Task* state shown in Figure 15.16. This state feeds the configuration cluster to the `Build Strain Task` sub-VI developed previously and places the resulting DAQ task on the data highway for use by the other states.

The state machine then transitions to the *Wait* state and waits for user input. The event case handling the value change of the STOP button is shown in Figure 15.17.

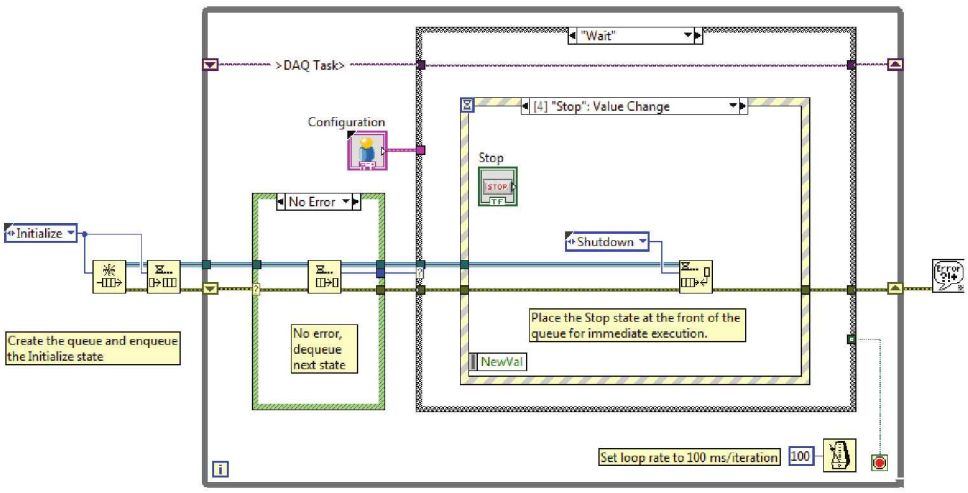


Figure 15.17: The event case handling the value change of the STOP button of the `Multitest` data acquisition VI

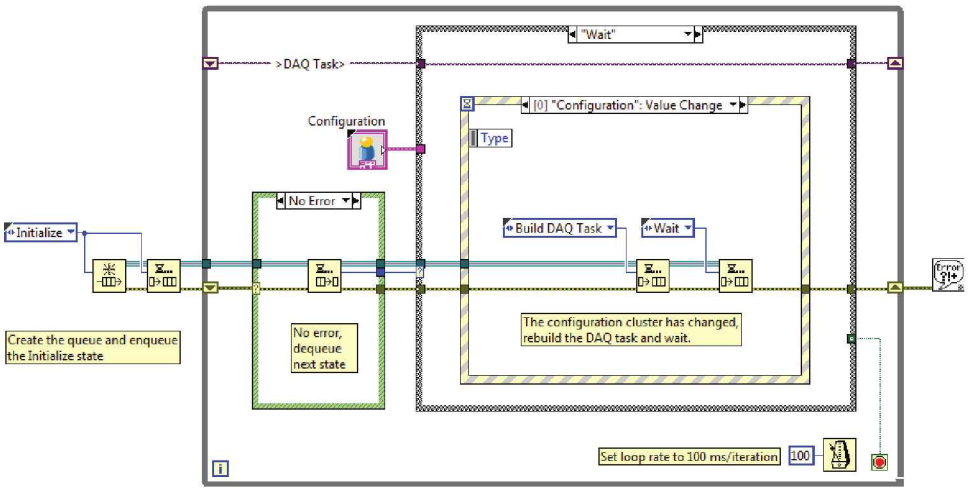


Figure 15.18: The event case handling the value change of the CONFIGURATION cluster of the `Multitest` data acquisition VI

The *Shutdown* state is placed at the front of the queue for immediate execution using the **`Enqueue Element At Opposite End`** function. The *Shutdown* state itself is shown in Figure 15.15.

If the user changes the configuration cluster, the event case shown in Figure 15.18 executes. This event case rebuilds the DAQ task and then returns the state machine to the *Wait* state.

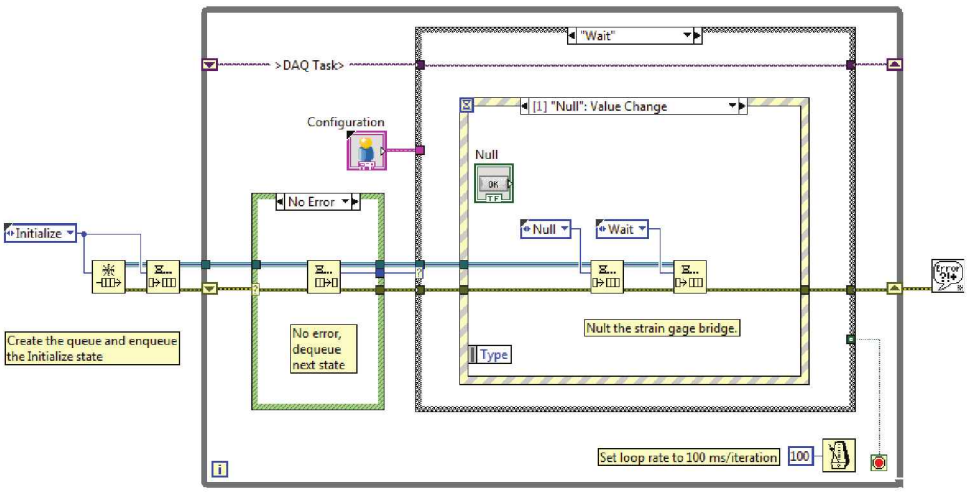


Figure 15.19: The event case handling the value change of the NULL button of the `Multitest` data acquisition VI

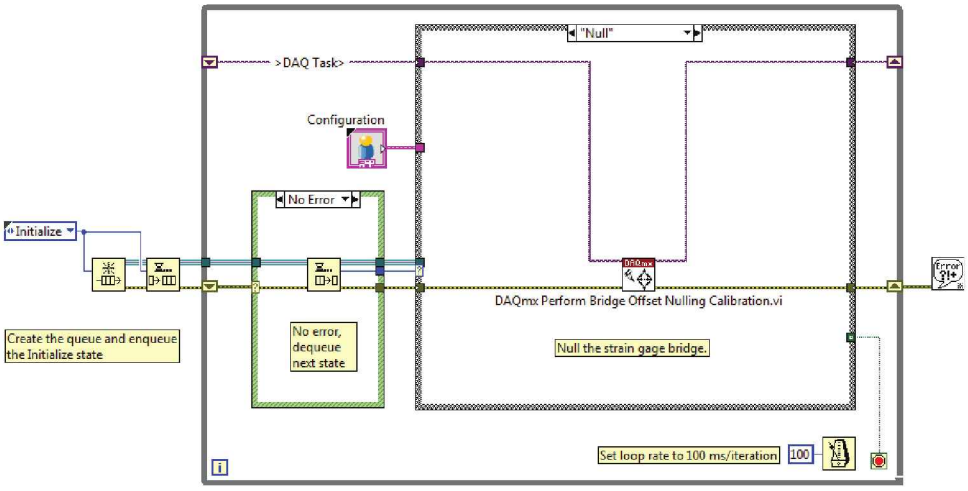


Figure 15.20: The *Null* state of the `Multitest` data acquisition VI

The event case handling the value change of the NULL button is shown in Figure 15.19. This state enqueues the *Null* state shown in Figure 15.20 and then returns to the *Wait* state. The *Null* state uses the DAQ task reference on the data highway to null the strain gage bridge using the **`DAQmx Perform Bridge Offset Nulling Calibration`** function shown in Figure 15.11.

The event case handling the value change of the CALIBRATE button is shown in Figure 15.21. This state enqueues the *Calibrate* state shown in Figure 15.22 and then

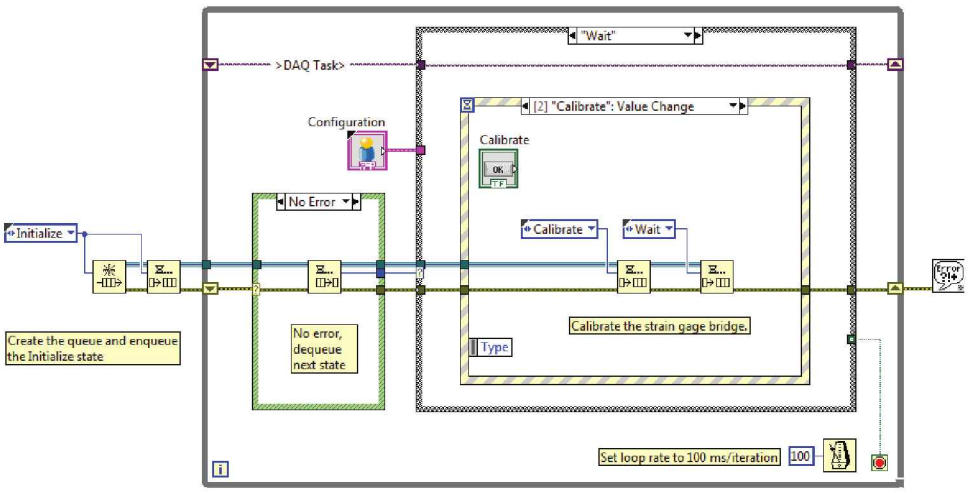


Figure 15.21: The event case handling the value change of the CALIBRATE button of the `Multitest` data acquisition VI

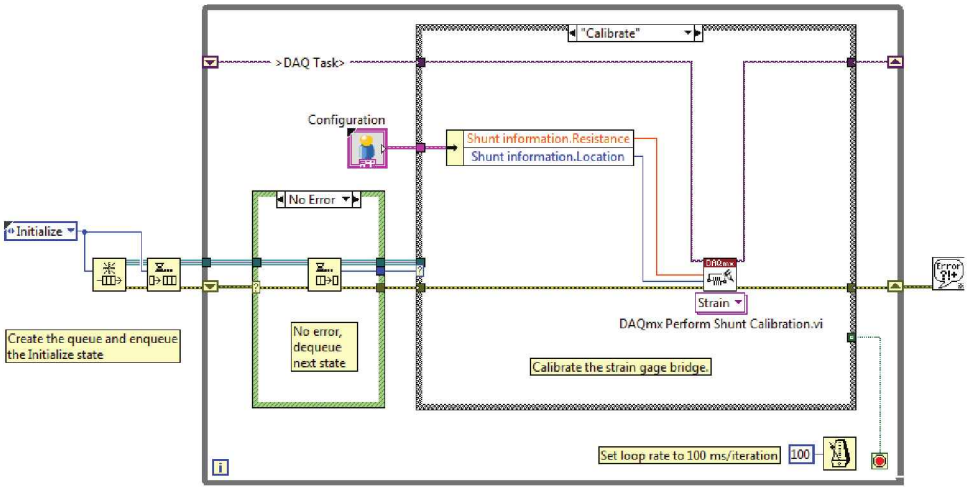


Figure 15.22: The *Calibrate* state of the `Multitest` data acquisition VI

returns to the *Wait* state. The *Calibrate* state uses the DAQ task reference on the data highway and the configuration cluster to perform shunt calibration on the strain gage bridge using the **`DAQmx Perform Shunt Calibration (Strain)`** VI shown in Figure 15.12. This is a polymorphic VI; the version shown in Figure 15.22 is configured to calibrate strain gages.

The event case handling the value change of the READ STRAIN button is shown in Figure 15.23. This state enqueues the *Read Strain* state shown in Figure 15.24 and

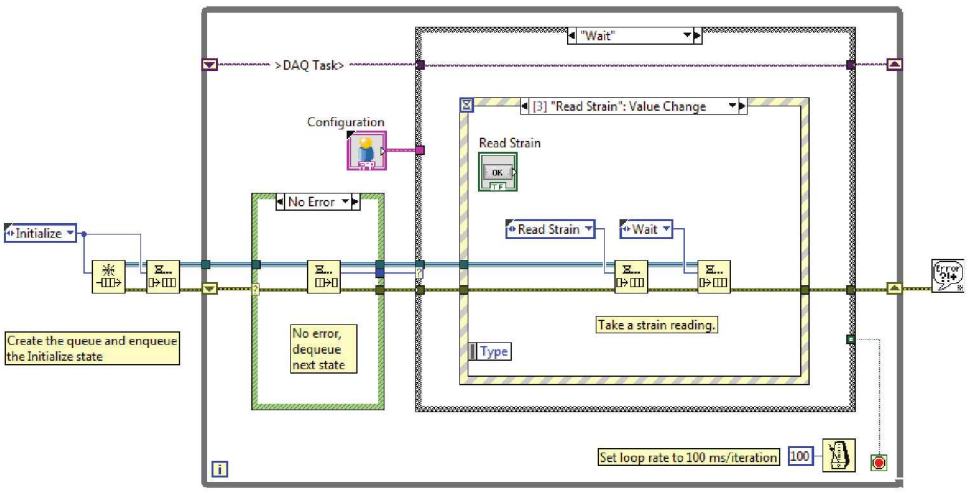


Figure 15.23: The event case handling the value change of the READ STRAIN button of the `Multitest` data acquisition VI

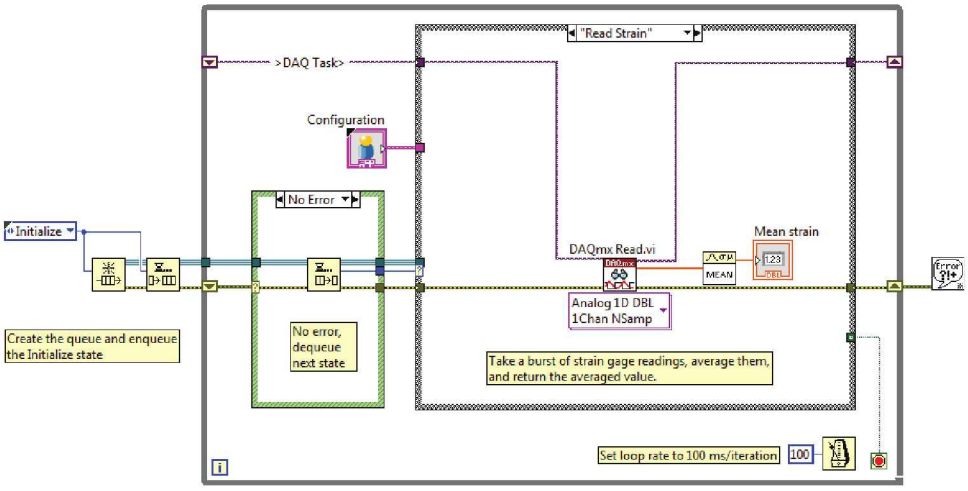


Figure 15.24: The *Read Strain* state of the `Multitest` data acquisition VI

then returns to the *Wait* state. The *Read Strain* state uses the DAQ task reference on the data highway and the configuration cluster to acquire a burst of readings from the strain gage using the **`DAQmx Read`** VI shown in Figure 15.13. This VI is polymorphic; the version shown in Figure 15.24 is configured to read multiple samples from a single DAQ channel. The output data is in double-precision format (DBL). The averaging of the strain data is performed by the **`Mean`** function. This help file is shown in Figure 15.25.

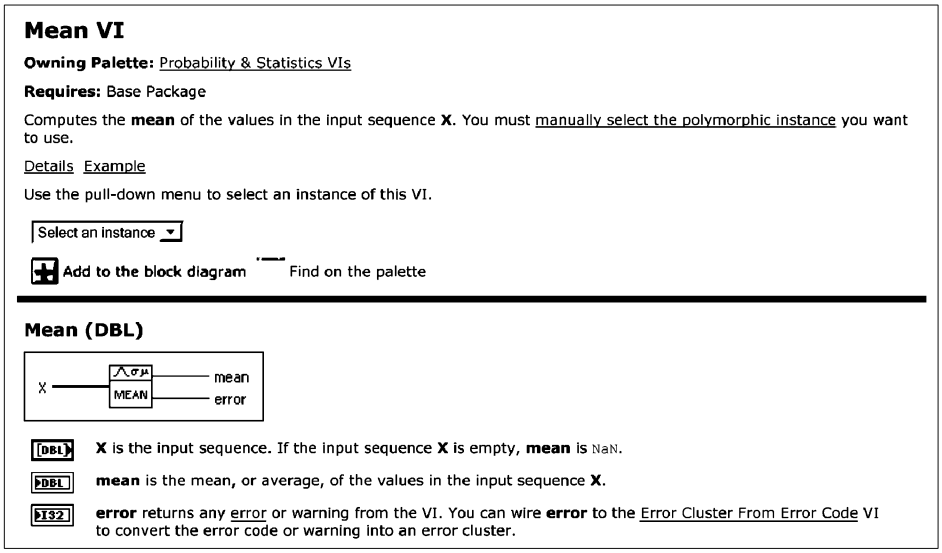
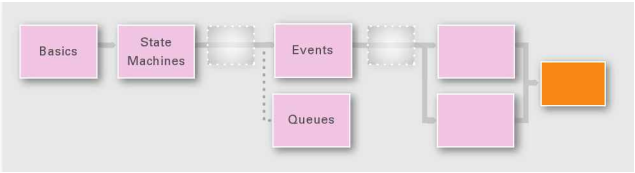


Figure 15.25: Help file for the `Mean` function

The strain measurement VI is now complete. This VI introduced many new functions and LabVIEW concepts related to data acquisition, but the structure of this VI is nearly identical to the versions of the `Multitest` VI developed in Chapter 13.



16 Problem Set 5

16.1 Traffic Light Controller Exercises

Exercise 5.1: Queued *Traffic Light Controller* Program

Would the traffic light controller be a good candidate for queues? Why or why not? ■

16.2 Security System Controller Exercises

Exercise 5.2: Queued Security System

Revise the security system controller to use queues. In the version developed in Exercises 3.8, and 4.2, the log was updated in the same state that detected the value change of the front-panel switches. In this exercise, create a new state called *Update Log* to perform this task. When a switch value change is detected, place the *Update Log* state on the queue. The *Update Log* state will then use the queue to return to the monitoring state.

What are the advantages of creating an *Update Log* state? What are the disadvantages? Consider the structural impact of this decision on the state machine architecture. In your previous implementations of the security system controller, the `Set Status Of All Zones` sub-VI and the `Update Log` sub-VIs were in the same state. This meant that the `Update Log` sub-VI had easy access to both the old zone status array on the data highway and the new zone status array output from the `Set Status Of All Zones` sub-VI. If you move the `Update Log` sub-VI to a different state, it will still be able to access the data highway, but it will no longer have direct access to the new zone status array.

One possible solution to this problem is to place the new status array on the data highway along with the old zone status array. The `Update Log` sub-VI will be able to access both the old and new zone states. Implement this solution in your queued version of the security system controller. ■

Exercise 5.3: Create an `Update Log Action Engine` Sub-VI

In Exercise 5.2, you were asked to store the new zone status array on the data highway so that it could be accessed by the `Update Log` sub-VI. In this exercise, you explore another solution to this problem. Create the `Update Log` action engine with the icon shown in Figure 16.1. This action engine persistently stores the old zone status state. This action engine has two commands: *Set Status* and *Update Log*. The *Set Status* command initializes or resets the stored zone status array. The *Update Log* command causes the action engine to compare the old and new zone status arrays and update the log if any zone has changed status.

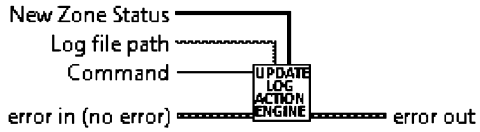


Figure 16.1: The icon of the `Update Log Action Engine` action engine ■

Exercise 5.4: Queued Security System with Action Engine

Implement the action engine you developed in Exercise 5.3 in a revised version of the security system controller you developed in Exercise 5.2. What are the advantages and disadvantages of using an action engine instead of the data highway to store the old zone status array? ■

16.3 Boiler Controller Exercises

Exercise 5.5: Queued *Boiler Controller* Program

The *Boiler Controller* program has many states that are performed in a fixed sequence. This makes it a good candidate for queues. Upgrade the event-driven *Boiler Controller* program to use queues. Put the entire sequence of states on the queue in the *Initialize* state. What are the advantages to using a queue in the boiler controller? Are they worth the extra effort in this case? ■

16.4 Car Wash 2 Exercises

In the following exercises, you will revise the event-driven Car Wash 2 controller developed in Exercise 4.5 to incorporate a queue. This is a major improvement of the Car Wash 2 controller, because it will simplify the way that the car wash option state transitions are handled. In previous implementations of the Car Wash 2 controller, an options array was created and stored on the data highway. An array index variable was also stored on the data highway. Both the array and the array index were used in conjunction with the `Next Car Wash State` sub-VI to determine the next state

transition. The `Next Car Wash State` sub-VI needed to check whether or not the options array was exhausted before determining the next state.

A queue can be used to eliminate most of these features. The option array and index will no longer be needed and will be removed from the data highway. Since the state transitions will be handled by the queue, the `Next Car Wash State` sub-VI will no longer needed. These changes will improve the readability and maintainability of the code.

Exercise 5.6: Create the `Build Wash Array` Sub-VI

Create a new state enumerated constant for the queued Car Wash 2 controller. All of the states are the same as before with the exception of the *Build Option Array* state. Rename this state as the *Queue Wash Sequence* state.

Create the `Build Wash Array` sub-VI with the icon shown in Figure 16.2. This sub-VI is very similar to the `Build Option Array` sub-VI developed in Exercise 1.40. The sub-VI accepts a cluster of wash option Booleans as an input, and it outputs a complete array of car wash states from vehicle entry to vehicle exit. The output array has the form: *Car Entry*, *Wash Option 1*, . . . , *Wash Option n*, and *Car Exit*. The state enumerated constant used in the array is the new version with the *Queue Wash Sequence* state.



Figure 16.2: The icon of the `Build Wash Array` sub-VI ■

Exercise 5.7: Queued Car Wash 2 Controller

Revise the event-driven Car Wash 2 controller developed in Exercise 4.5 to incorporate a queue. Use the new enumerated state constant created in Exercise 5.6. Use the `Build Wash Array` sub-VI also developed in Exercise 5.6 in the *Queue Wash Sequence* state to create the wash sequence of states. Enqueue the states using a **`For`** loop with an auto-indexed input tunnel. ■

16.5 DAQ State Machine Exercises

Exercise 5.8: The `Clear Error Code` Sub-VI

Develop the `Clear Error Code` sub-VI with the icon shown in Figure 16.3. This sub-VI accepts an error cluster and an error code as inputs. The error code input

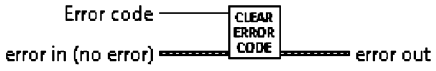


Figure 16.3: The icon of the `Clear Error Code` VI

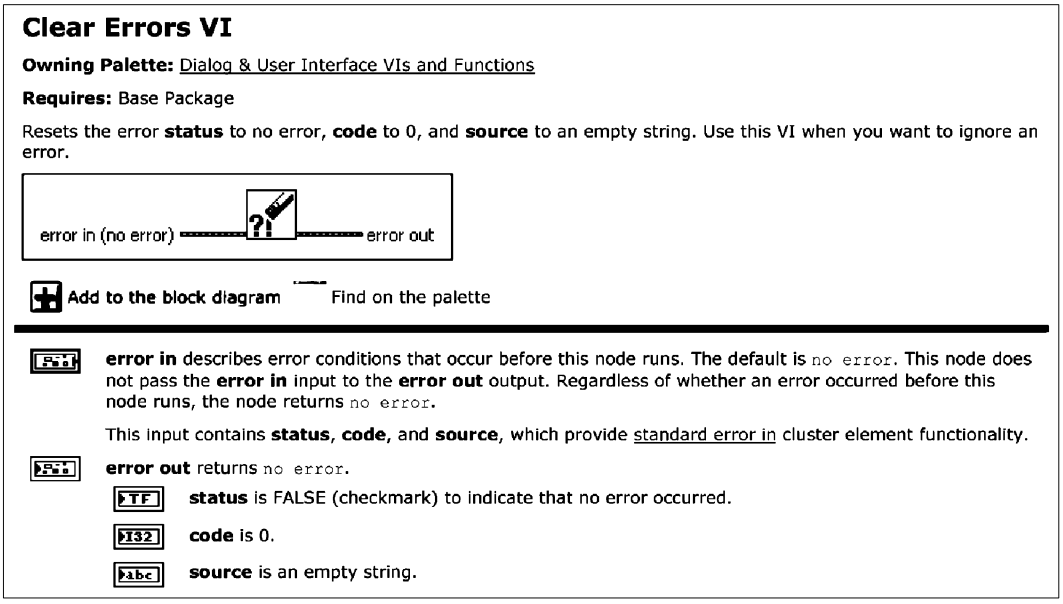


Figure 16.4: Help file for the `Clear Errors` function

should be an `I32` numeric control. This sub-VI will check to see if an error exists. If an error does exist and the error code matches the input error code, this sub-VI will clear the error. Otherwise, the error will be passed to an error cluster output. You might find the **`Clear Errors`** function useful (see the help file in Figure 16.4). ■

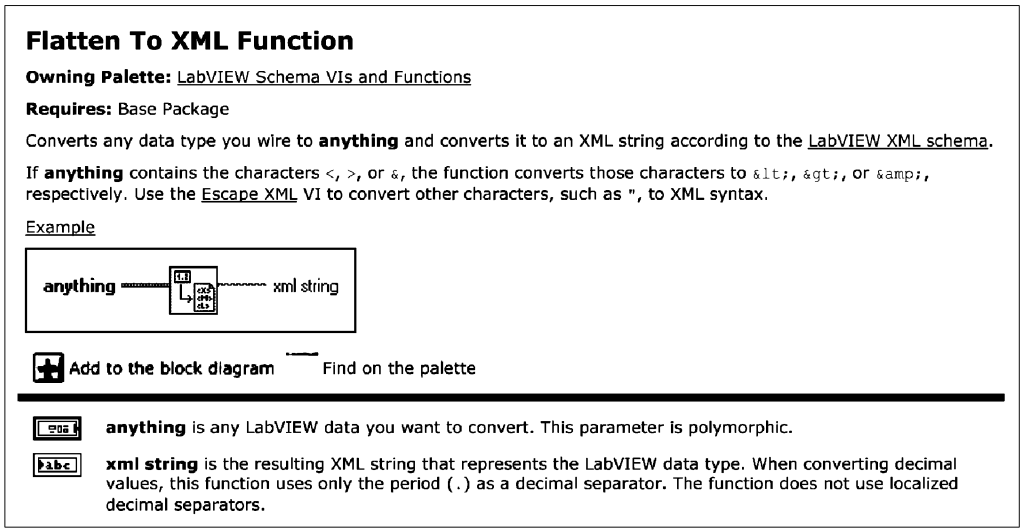


Figure 16.5: Help file for the `Flatten To XML` function

Exercise 5.9: Read/Write Configuration Cluster to an XML File

Modify the strain-gage program developed in Chapter 15 to allow the user to write the contents of the configuration cluster to a human-readable text file. The user should also be able to read a previously generated file and write the contents to the configuration cluster. An easy way to do this is to flatten the configuration cluster to an XML string using the **`Flatten To XML`** function (see the help file in Figure 16.5). The flattened string then can be written to a text file using the **`Write to Text File`** function (see the help file in Figure 1.48).

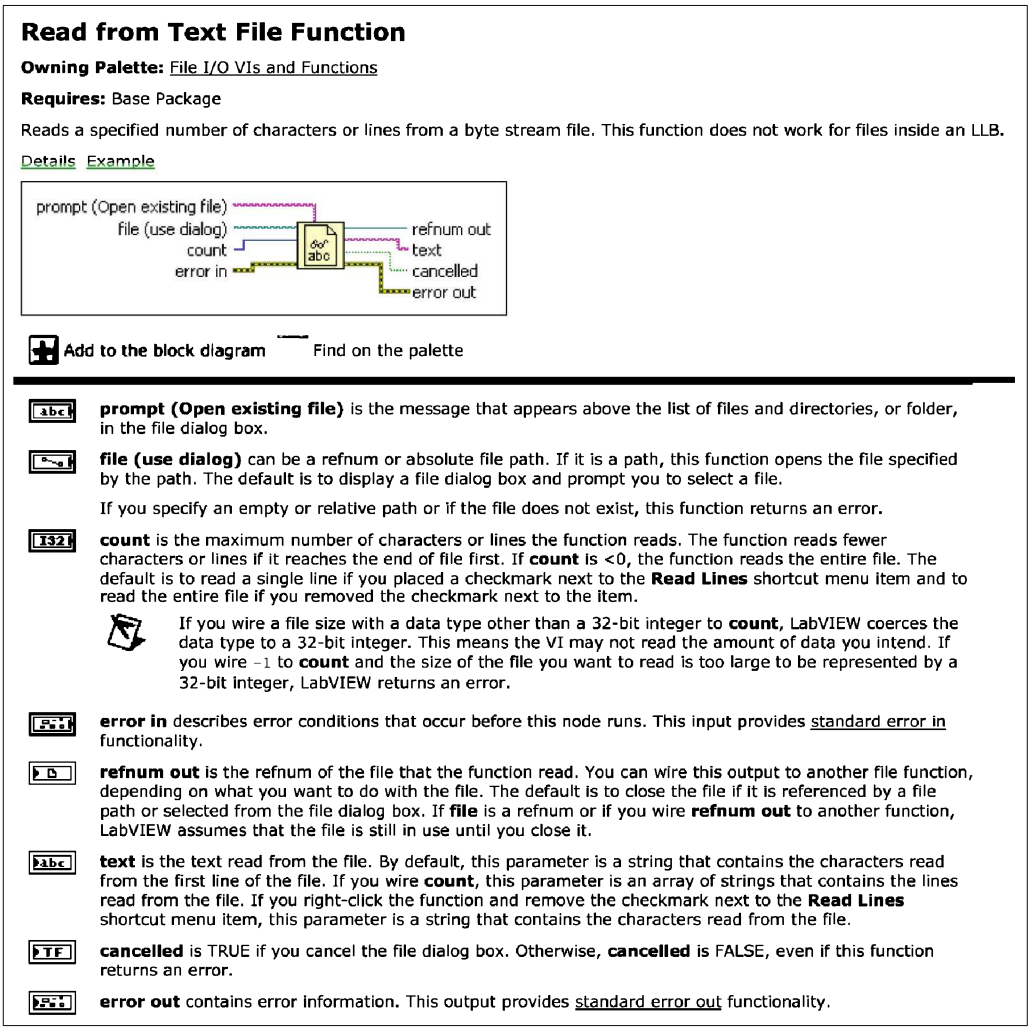


Figure 16.6: Help file for the `Read from Text File` function

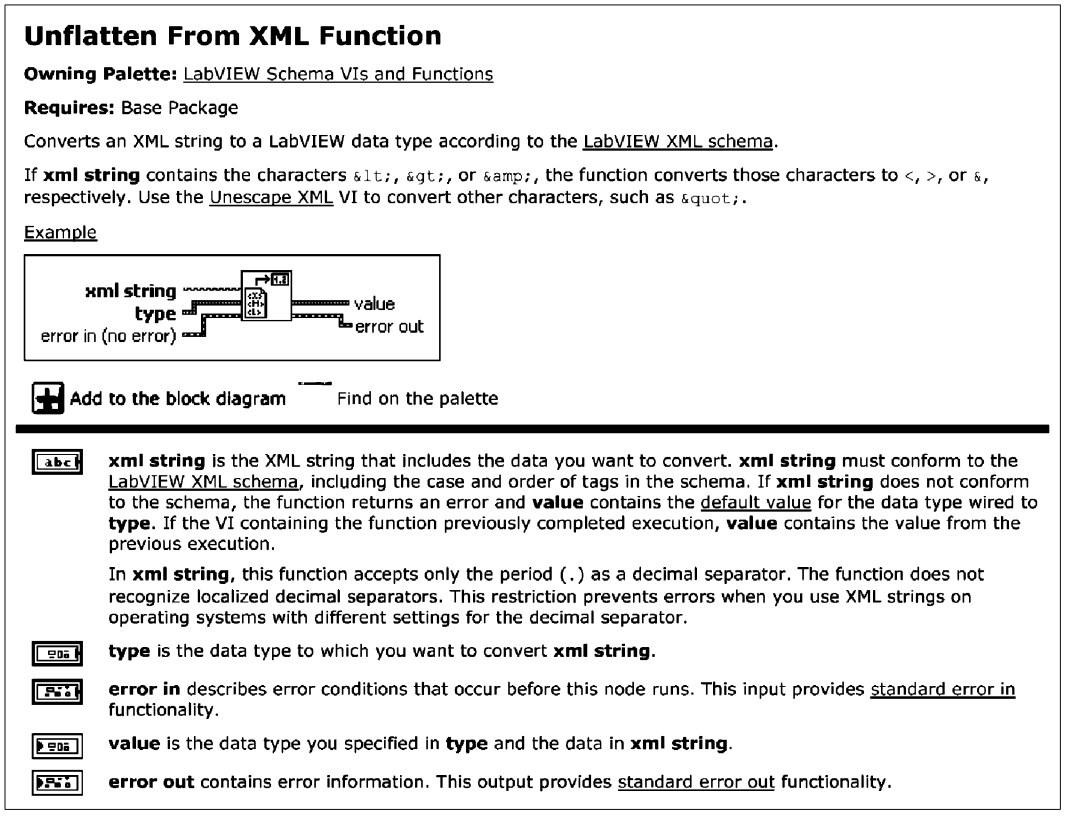


Figure 16.7: Help file for the `Unflatten from XML` function

The XML-formatted configuration data can be read from an existing text file using the **`Read from Text File`** function (see the help file in Figure 16.6). The XML data can be converted back to a cluster using the **`Unflatten from XML`** function (see the help file in Figure 16.7).

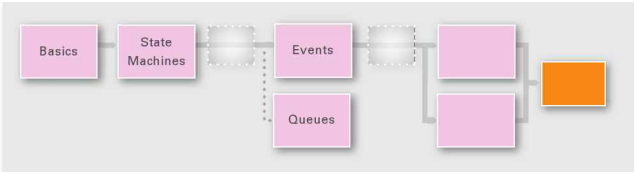
Add two new buttons to the front panel to allow the user to read or write the configuration cluster data to a text file. The program will bring up a file dialog window to allow the user to navigate to the desired file location. If the user cancels the dialog window, error `43` will be generated. Use the `Clear Error Code` sub-VI you developed in Exercise 5.8 to clear the error if it occurs to prevent the state machine from shutting down. ■

Exercise 5.10: Write the Strain Data to an Output File

Modify the strain-gage program developed in Exercise 5.9 to save the strain data to an output file. Place a SAVE DATA button and an output file path control on the front panel of the VI. When the SAVE DATA button is pressed, the VI will append a

comma-separated string to the end of the file designated by the output file control. The specification of the string is as follows:

- The string will have the following format: date, time, sampling rate in Hz, number of samples averaged, and strain.
- Each string should appear on its own line in the data file.
- The strain should be recorded in scientific notation.
- The date and time recorded in the string should be the date and time that the data was taken, not the date and time that the data was recorded to the file.
- The sampling rate and number of samples averaged should be the values that were in use when the data was taken, not the values stored in the configuration cluster when the data was recorded to the file. ■



17 Preparing for the CLD Exam

The Certified LabVIEW Developer (CLD) examination is a four-hour programming exam. The examinee is given a functional specification describing the requirements of a desired LabVIEW program. The examinee is placed in a room with a computer with a standard installation of LabVIEW. The computer has no internet connection. Externally developed code and third-party tools are not allowed to be used in the exam.

The goal of the exam is simple. Starting with nothing but the standard LabVIEW installation, the functional specification, and four hours on the clock, develop a fully functional, well documented application with good programming style that satisfies the requirements in the functional specification.

The test is graded by National Instruments and points are awarded in three categories:

1. Programming style: 15 points
2. Functionality: 15 points
3. Documentation: 10 points

A perfect score is 40 points. The passing grade is 75% or 30 points.

The biggest complaint about the test is that the four hour time limit is too short. Granted, the four hour limit is very tight. Granted, in a real job, you would be given more than four hours to produce a similar code. So what is the purpose of the time limit? One possible reason is that it tends to separate the applicants into two groups: those that have truly internalized LabVIEW "thinking" and style and those that have not.

Let's take a closer look at the point distribution. Only 15 points out of 40 are dedicated to functionality. That's only 37.5% of the total, which is a little more than a third of the total points. About two-thirds of the points are awarded for style and documentation, and style and documentation are not really affected by the four-hour time limit.

Documentation points are the easiest ones to earn. They don't require specialized LabVIEW knowledge, you just have to know what is expected and do it. Documentation requires some time, but not much. You shouldn't spend more than a half an hour

of the four-hour time limit on documentation. Documentation is worth 10 points, so that is a very good return on a small investment of time.

That leaves you about three and half hours to earn a possible 30 points in style and functionality. The style points are key. It is very hard to fake good style. You either have it or you don't. Style is learned through experience. It is gained by following good examples and putting them into practice in your own work. If you haven't seen good examples of LabVIEW code and you haven't incorporated them into your own work, you probably don't have good LabVIEW style, and it will very hard to pass the exam.

But if you have read this book, done the exercises, and applied the lessons to your own code, you probably do have good LabVIEW style. Good programming style has two huge benefits in terms of the CLD exam. First, good style promotes functionality. If you practice the style recommendations and the general layouts of the design patterns in this book, it will be easy to produce functional code that will satisfy the CLD graders. Even better, once good style is internalized, it becomes a self-reinforcing habit. If you have good style going in to the exam, you should expect to get a good style score with very little expended time.

This is the crux of the matter. If you have already internalized good LabVIEW practices and style guidelines and you take care to follow the documentation suggestions in this chapter, you will likely score the majority of the 25 style and documentation points with less than an hour's worth of effort during the test. That leaves you three hours to focus on functionality. If you can then score at least 10 out of 15 points on functionality, you have an excellent chance of passing the test.

This chapter will provide you with tips on how to prepare for and take the CLD exam. The author wishes to point out that these suggestions are his own opinions and do not reflect any official policy, guidelines or position of National Instruments.

17.1 Documentation

The documentation points are yours to lose. You should go in to the exam assuming that you will get full credit in this category. When you are taking the exam, you should be disciplined and document as you go along. Don't succumb to the temptation to jump right into the programming and save the documentation for the last few minutes of the exam.

There are three types of documentation that you should pay attention to during the test:

VI Documentation The main VI and all sub-VIs should be documented in the *VI Properties... Documentation* window.

Front Panel Documentation All front panel controls and indicators should be documented with descriptions and tip strips.

Block Diagram Documentation Block diagrams should have free label documenting function. Try to place a label in every loop and in every case of every case or event structure. Sub-VIs should have a free label documenting their function. Important wires, such as those in the data highway, should also be labeled. Label block diagram constants.

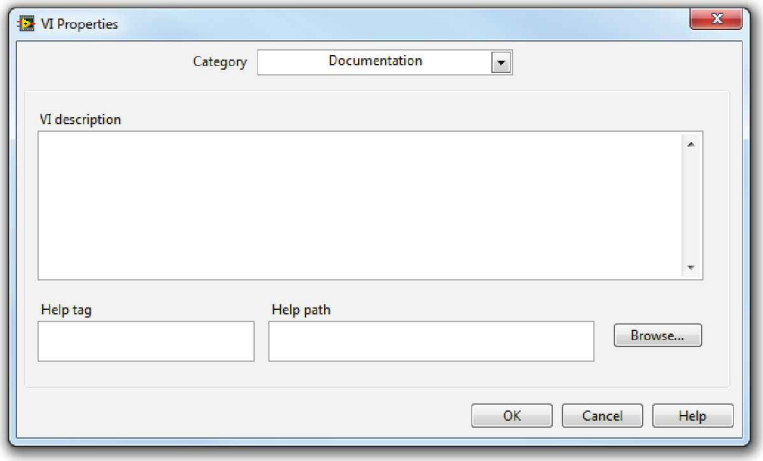


Figure 17.1: Documenting a VI using a VI description

Every time you create a new VI, including the main VI, immediately document it by right-clicking on the icon and selecting *VI Properties*. Select the *Documentation* category and fill in the *VI Description* field, as shown in Figure 17.1. The text entered in this window will be displayed if the user hovers the mouse over the VI icon while the context help window is open. To save time during the test, select the text and copy it (`Control-C`). Then create a free label on the block diagram and paste the text into the label (`Control-V`). This documents the code in two different places with the same description.

Front panel objects need to be documented with descriptions and tip strips. The description text shows up in the context help window when the user hovers the mouse over the object. Tip strips show up in small pop-up windows when the user hovers the mouse over the object.

To create descriptions and tips, right-click on a front panel object and select `Description and Tip...`. This brings up a window with text fields for the description and tip, as shown in Figure 17.2. Again, feel free to cut and paste between the two fields to save time during the test.

If a front panel object is a cluster, there are two levels of descriptions and tips. The cluster itself can have a description and tip, and each individual element of the cluster can have a description and tip. During the exam, you should document both the overall cluster and the individual elements with descriptions and tip strips.

Sub-VI front panel objects also should be documented with descriptions and tip strips. To save time during the exam, use strict type definitions for all front panel controls and indicators and document the type-definition file with a description and tip strip. Then when you use these type-defined controls and indicators in your sub-VIs, they will automatically carry over the descriptions and tip strips from the type-definition file.

Block diagrams should be documented with free labels. Sub-VIs should have a label describing their functionality. Complicated sequences of code should have labels

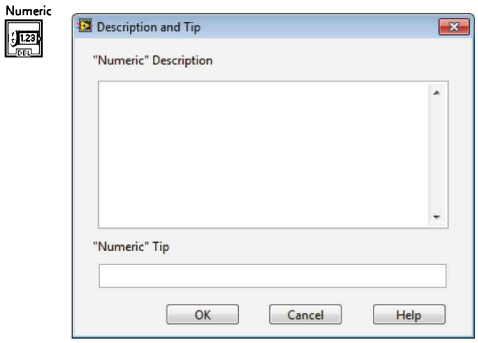


Figure 17.2: Documenting front panel objects with descriptions and tip strips

describing what the code is doing. As a rule of thumb, try to include a label in each loop and in each case of every case or event structure. In the main case structure of a state machine, each state should have a label describing what the state does.

Certain block diagram items should also be labeled. Long wires should be labeled using the `>>Label Text>>` convention. All wires on the data highway also should be labeled in this fashion. Block diagram constants also should be labeled.

Block diagram labels also can be used to communicate with the exam grader. If the functional specification is not completely clear, list your assumptions in the appropriate places in your code using free labels. If you are running out of time and your code is not fully functional, use free labels to describe what you would do to finish if you had more time.

The key to getting full credit in the documentation category is to do the documentation as you go. Don't wait until the end of the test to document your code. When you create a new VI, fill in the VI documentation. When you create a type definition or drop a new front panel control or indicator, fill in the description and tip strip. When you create a loop, put in a label. When you create a case structure, document each case as it is created.

17.2 Style

Good style is easier to demonstrate than describe, especially in a graphical language like LabVIEW. You learn it by seeing it and by practicing it yourself. The hard part is finding programs written with good style to study and emulate, especially if you are a lone-wolf LabVIEW programmer.

This was the motivation for writing this book. This book provides you with LabVIEW codes written with good style in the hope that you will emulate them. So if you want to score well in the style category of the CLD exam, try to make your code look like the codes in this book.

Chapter 1 closed with a list of style guidelines followed in this book. The list is repeated here, but modified slightly to focus more tightly on code written for the

CLD exam. Try to absorb these points in your own programming style before you take the test. If you have to think about them during the test, you will waste valuable time.

- The program should occupy one screen with no scrolling.
- Avoid sequence structures, use state machines instead.
- Minimize the use of local variables. Eliminate them if possible.
- Do not use global variables.
- Keep wires as straight as possible.
- Align functions on the block diagram to improve readability.
- Make every effort to improve readability. For example, use **`Unbundle By Name`** instead of **`Unbundle`**.
- Every loop should have at least one free label.
- Every case of every case or event structure should have at least one free label.
- Sub-VIs should use the error-handling template if possible.
- Every sub-VI block diagram should have at least one free label documenting the sub-VI function.
- All sub-VIs should use the $4 \times 2 \times 2 \times 4$ connector pane if possible.
- Connector panes should be wired with inputs on the left and outputs on the right.
- Error wires should use the lower corner terminals of the connector pane. References should use the upper corners of the connector pane.
- Every sub-VI should have a descriptive custom icon.
- The error chain should be tight. If a function has error terminals, they should be wired. All errors wires should eventually merge with the main error wire.
- Create type definitions for clusters of front panel controls or indicators.

17.3 Functionality

Functionality is key, but you don't need a fully functional VI to pass the CLD exam. In fact, if you score full marks for documentation and style you only need 5 points out of 15 in the functionality category to pass the test. That being said, of course you would like to turn in a fully functional program. Here are some suggestions that will help you do that in the four-hour time frame of the test.

This book contains the functional specifications of six past CLD exams. This should give you a good idea of the scope of the exam. Be warned, though, that the traffic light and security system exams are older than the others and are likely to be simpler than the tests offered today.

All of these exams can done using the single loop classic state machine architecture described in Chapter 4. Know this design pattern well. Do not take the exam if this design pattern is not in your muscle memory. You should be able to set up the basic structure of a state machine in minutes and with little thought.

You also should be familiar with event structures and queues. These also should be in your muscle memory before you take the test. Queues and event structures may or may not be helpful in a given test, but you should definitely have these tools at your disposal.

Another area to review is the placement of constants, controls, and indicators in a state machine. Review Chapter 4 and pay special attention to the discussion of how persistence and access affect where an item should be placed in the state machine. Ask yourself the following questions about each constant, control, and indicator:

- Should this item go in an individual state?
- Should it be in the lobby or the alley of the state machine?
- Is this element volatile, changing every loop iteration?
- Is this item needed by more than one state?
- Do this item need to be stored from one loop iteration to the next?
- Should this item be on a data highway? Do I really need a data highway?

Of the six previous CLD exams provided in this book, four require a timer of some sort. Timers are typically implemented as action engines. The basic action-engine template also should be in your muscle memory before taking the test. Different storage formats were presented in this book (single loop, **`For`** and **`While`** loops, and feedback nodes). Choose your favorite and practice it before taking the test. Practice by building a basic timer action engine with *Elapsed Time* and *Reset* functionality until you can build it with little thought. This will save time during the test. Keep in mind that timers need to be reset. If you are testing your code and the timing is not working, make sure that the timer is reset before you first call it and that it is reset before each new timed task.

Error handling is an easy thing to overlook during a test. Make sure that your VI has a tight error chain. Wire every error terminal of every function that you use. Merge the errors where necessary. Put critical functions on parallel error chains and then merge them back into the main line. Build your sub-VIs with error inputs and outputs, and use the error-handling template described in Chapter 1 whenever possible.

Organization is also important. Create a LabVIEW Project for your exam entry. Organize the project to make the file structure clear.

17.4 Preparing for the CLD Exam

Remember that the CLD is a performance exam. It doesn't test what you know; it tests what you can do. So you can't really study for it in the traditional sense. You need to practice for it instead.

This book provides you with more than enough material to study. Know the material about classic state machines, event-driven state machines, queued state machines, and action engines.

Practice by building the codes shown in this book. Build the various versions of the `Multitest` VI as a start. The ultimate test, of course, is to time yourself and

attempt to work through the sample CLD exams presented here. Then compare your solutions with the ones given in this book. When you are confident that you can do the sample exams in the four-hour time limit, you are ready to take the CLD exam.

17.5 Time Management

Time management is key to passing the exam. Here are some general guidelines to consider on how best to spend your time during the CLD exam.

1. The first hour or hour and a half should be spent in planning and preparation. Resist the urge to start programming immediately. Read the functional specification thoroughly. Make sure that you understand what the program needs to do. Are there any fuzzy areas in the specification? If so, reread the specification more carefully. If it is still unclear, make any needed assumptions and document them.
2. Decide which design pattern would be best suited for solving the problem. Ask yourself the following questions.
   - Does the state machine need to wait? Would an event structure be helpful?
   - Would it be helpful to be able to queue a series of states instead of just one at a time? Are queues worth the extra effort?
   - Does any data need to be stored from one loop iteration to the next? Do I need a data highway?
   - Where are the best places to put the various controls and indicators?
   - Would an action engine be helpful? If so, what data needs to be stored and what functions should it perform?
   - What states are needed? What are the state transitions?
3. Take the time to draw a state transition diagram, it will help you visualize the state transitions and may lead you to revise your list of needed states.
4. Once the planning is done, start the preparation phase:
   - Create the LabVIEW project.
   - Open the main VI and save it. Save your work often.
   - Fill in the VI documentation.
   - Create the front panel. Make it look like the example in the specification. Make sure that data types are correct. Make sure that the data representations (DBL, I32, etc.) are correct.
   - Create type definitions for important clusters of controls and indicators and for any enumerated controls. Fill in the description and tip strips in the type definition files.
   - Create the basic structure of your chosen state machine in the block diagram. Add free labels as you do so. Put the controls and indicators in their approximate locations.
5. At this point you are ready to do the real work. Plan on spending two hours or so following your plan and completing the state machine. Document as you go.

Decide ahead of time whether you are a bottom-up or a top-down programmer. If you prefer bottom-up, spend time developing any needed sub-VIs or action engines and then place them in the appropriate places in your state machine.

If you are a top-down programmer, create stubs for the sub-VIs and action engines. A stub is a sub-VI with a custom icon and a wired connector pane and front panel—but with no working code on the block diagram. Stubs are useful because you can drop them into the main program and wire them properly. When the main VI is at or near completion, develop the block diagrams of the stubs without changing the connector pane.

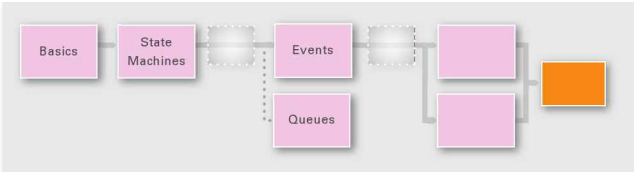
If you find yourself approaching the last half hour and the code is not fully functional, stop and document with free labels what you would do if you had more time. If you have stubs that have not been developed fully, place labels on their block diagrams describing how you have developed them if you had more time.

6. The last half hour should be dedicated to a final review. Remember, documentation and style points are yours to lose! Review every loop and every case of every case structure. Is everything documented? Are all front panel controls and indicators documented with descriptions and tip strips? Are all sub-VIs documented with VI documentation and free labels? Do all sub-VIs have custom icons? Is everything laid out cleanly and in good style? Is there a label in every case of every case and event structure?

   After you have confirmed that you have done all you could to fully earn the documentation and style points, return to functionality if you have time. Finish what you can, test what you have finished, and document the rest with what you would have done if you had more time.

17.6 Practicing for the Exam

The key to passing the CLD exam is to internalize good LabVIEW style, and the way to internalize good style in any programming language is to practice, practice, practice. The best way to do this is to work out the sample practice exams included in this book. Two of the published sample exams have not been used in the exercises presented thus far: the sprinkler controller exam and the ATM controller exam. This was done to give you the opportunity to test yourself with these exams. The best way to do this is to time yourself and see if you can complete each test in four hours. If you get stuck or you are not preparing for the CLD exam, you can work through these exams in a guided fashion by completing the exercises in Chapters 18 and 19. Chapter 18 covers the sprinkler controller exam, and Chapter 19 covers the ATM controller exam. You can also practice for the exam by working through the other exams presented in this book and imposing the four hour time limit.



18

Problem Set 6

This problem set will guide you through a solution of the sprinkler controller sample Certified LabVIEW Developer exam. If you are preparing for the CLD exam, try to complete the sprinkler controller in four hours without reading the exercises in this problem set.

Exercise 6.1: Sprinkler Controller State Transition Diagram

Read the functional specification for the sprinkler controller in Chapter E. What states will the program need? How do they transition from one to another? Draw a state transition diagram for a sprinkler controller. Pay attention to how the STOP and SETUP buttons are handled in the diagram. Add notes to the diagram to explain any actions that are difficult to capture in the diagram itself. Use the diagram to create a list of states for the sprinkler controller. ■

Exercise 6.2: Select an Architecture

Select a state machine architecture for the sprinkler controller. Would an event structure be useful? Would queues be useful? After you have selected an architecture, consider the placement of the controls and indicators. Where should the CONTROLS cluster, the ZONE SETUP ARRAY control, and the SIMULATION CONTROLS control be placed? Are these controls needed by multiple states? Do they change often? Do their values need to be stored? Consider the indicators. Where should the ZONE INDICATORS cluster and the ZONE STATUS cluster be placed? Do they need to be updated by multiple states? Do they need to be updated frequently? Consider persistent storage. Does the sprinkler controller need a data highway? If so, what should be stored on the data highway?

Consider the guard clause. Can it be used to monitor STOP and SETUP buttons? What states should the guard clause have? ■

Exercise 6.3: Sprinkler Controller Type Definitions

The sprinkler controller uses several type definitions. First, create an enumerated constant containing the list of states generated in Exercise 6.1. If you have decided to use the guard clause to monitor the front panel activity, create an enumerated constant of states present in the guard clause.

The front panel has many important clusters that should be saved as type definitions. Create type-defined control files for CONTROLS cluster, the SIMULATION CONTROLS cluster, and the ZONE INDICATORS cluster. Consider also the ZONE SETUP ARRAY control. The ZONE SELECTOR combo box used in the array should be saved as a type definition. There is no need to make a type definition of the entire array, but the array element consisting of the ZONE SELECTOR combo box and the TIMING numerical control also should be saved as a type definition. ■

Exercise 6.4: Create a `Setup Stop or Error` Sub-VI for the Guard Clause

Create the `Setup Stop or Error` sub-VI with the icon shown in Figure 18.1. The STATUS output of this sub-VI is an enumerated constant wired to the selector terminal of the guard clause. This sub-VI will monitor the STOP button and the SETUP button, as well as check for errors. The guard clause case structure has three states: *Stop or Error*, *Setup*, and *Continue*. If the STOP button is pressed or if an error is detected, the guard clause will execute the *Stop or Error* state. If the SETUP button is pressed, the guard clause will execute the *Setup* state. If none of these events have occurred, the guard clause will execute the *Continue* state. This sub-VI should operate normally even if an error is present at the error input terminal.

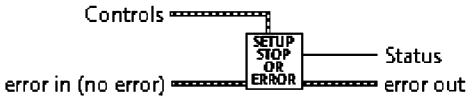


Figure 18.1: The icon of the `Setup Stop or Error` VI ■

Exercise 6.5: Create a Sub-VI to Read the Zone Array File

The initial values of the ZONE ARRAY control are stored in a comma-separated value (csv) text file. Create the `Initialize Zone Array` sub-VI with the icon shown in Figure 18.2. This sub-VI reads the text file at the given path, converts the data from strings to the needed data type, and writes the data to the ZONE ARRAY control.



Figure 18.2: The icon of the `Initialize Zone Array` VI

This sub-VI should be constructed without making assumptions about the number of zones contained in the input file.

You might find the **`Read From Spreadsheet File`** function useful for reading the comma-separated value text file (see the help file in Figure 18.3). Once the values have been read from the file, they need to be converted into the correct data types. You might find the **`Fract/Exp String To Number`** function useful (see the help file in Figure 18.4). You might also find the **`Type Cast`** function helpful (see the help file in Figure 3.38).

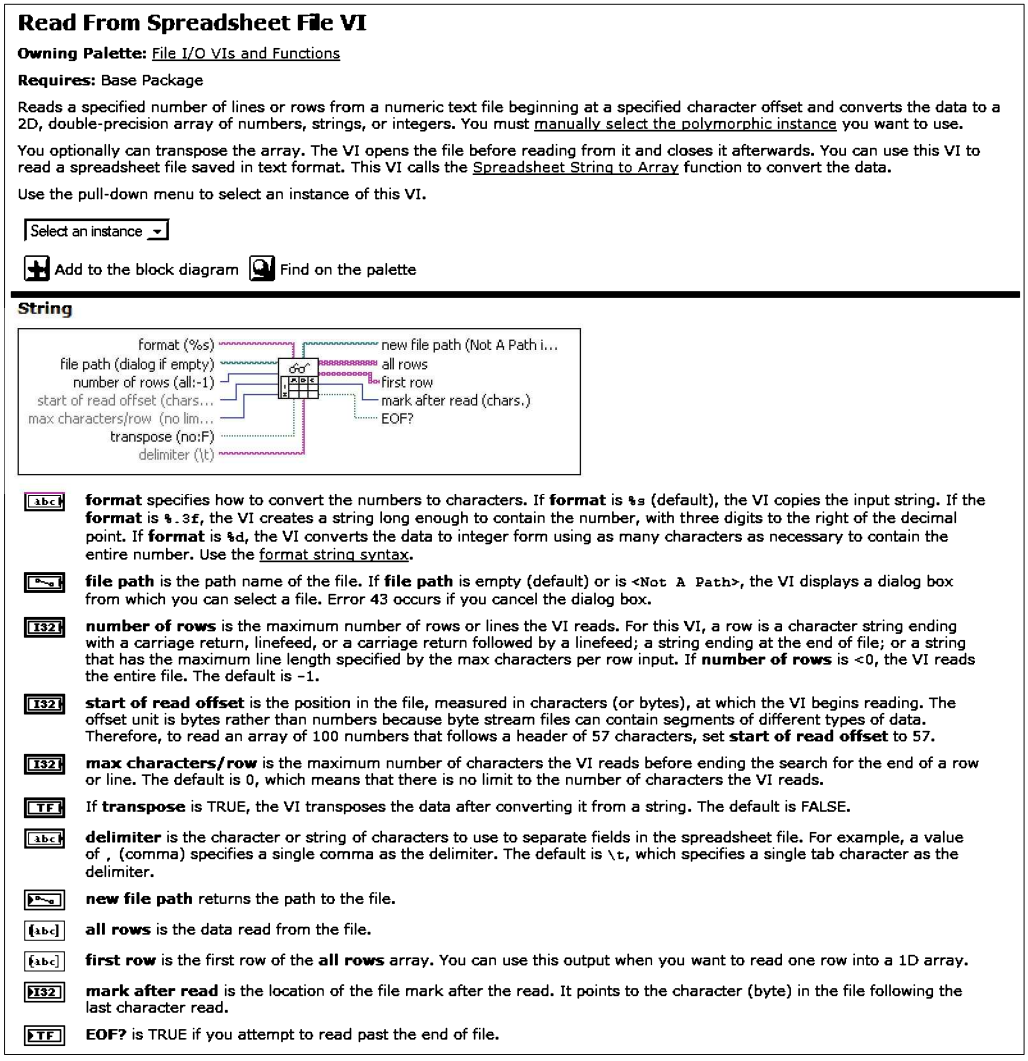


Figure 18.3: Help file for the `Read From Spreadsheet File` function

Fract/Exp String To Number Function

Owning Palette: String/Number Conversion Functions

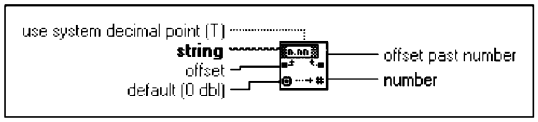
Requires: Base Package

Interprets the characters 0 through 9, plus, minus, e, E, and the decimal point (usually period) in **string** starting at **offset** as a floating-point number in engineering notation, exponential, or fractional format and returns it in **number**.

The connector pane displays the default data types for this polymorphic function.

If you want the function to return a 64-bit integer output, you must wire a 64-bit integer to the **default** input.

Example



Add to the block diagram Find on the palette

use system decimal point defines the decimal separator. If TRUE (default), the decimal separator uses the localized decimal separator. If FALSE, the decimal separator is a period.

string can be a string, a cluster of strings, an array of strings, or an array of clusters of strings. If **string** contains the characters `Inf` or `NaN`, this function returns the LabVIEW values `Inf` and `NaN`, respectively.

offset is the starting position and must be numeric. The offset of the first character in **string** is 0. If unwired or if less than 0, the default is 0.

default is any object of numeric representation and specifies the numeric representation for **number**. The default is an extended-precision, floating-point value of 0.

offset past number is the index in **string** of the first character following the number. **offset past number** reflects the value from the last string if you input an array of strings.

number can be a number, a cluster, an array of numbers, or an array of clusters, depending on the structure of **string**. The following table shows how the values of **string**, **offset**, and **default** affect **number**.

string	offset	default	offset past number	number	Comments
-4.7e-3x	0	0	7	-0.0047	x is not allowed, so conversion stops there.
+5.3.2	0	0	4	5.3	Second decimal point not allowed, so conversion stops there.

If you scan a string that does not fit into the numeric data type you specify, this function returns the largest number that fits into that data type.

Figure 18.4: Help file for the `Fract/Exp String To Number` function ■

Exercise 6.6: Create the `Next Sprinkler State` Sub-VI

One of the challenges in the sprinkler controller is determining the state transition from one state to the next. The sequence of zones is specified in the ZONE SETUP ARRAY, but the normal progression of zones can be interrupted at any time by rain or low water pressure. Some state transitions depend on whether the sprinkler is in single or continuous mode. The value of the START button can also influence the determination of the next state. The next state can also depend on the controller's current state.

Create the `Next Sprinkler State` sub-VI with the icon shown in Figure 18.5. This sub-VI accepts the SIMULATION CONTROLS, the current state, and an ENDED? Boolean as inputs, and it returns the next state as an output. The ENDED? Boolean is True if the controller is in single-program mode and the current program has ended and is False otherwise.

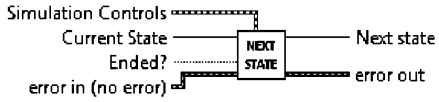


Figure 18.5: The icon of the `Next Sprinkler State` VI ■

In the next few exercises, you develop a series of sub-VIs that are used to service a sprinkler zone. These sub-VIs are collected into a single `Service Zone` sub-VI for use by the *Service Zone* state. The `Service Zone` sub-VI is responsible for running the zone time check, for creating the output cluster for display by the main application, and for incrementing the index of the ZONE SETUP ARRAY. The readability of the main application is increased by putting all of these related functions into a single cohesive sub-VI.

Exercise 6.7: Create a Sprinkler Timer

The sprinkler states are timed, so a timer needs to be developed. Consider the timers developed for the other projects in this book. The sprinkler timer is much like the traffic light timer. The sprinkler timer runs continuously when the zones are being serviced. The timer never needs to be paused, and there is no break between zones. As soon as one zone ends, the next zone begins. This suggests that an auto-reset timer (such as the one developed for the traffic light controller) could be used for the sprinkler controller. The only change that needs to be made to the traffic light timer is the addition of an elapsed time output.

Create the `Auto Reset Timer With Elapsed Time` action engine with the icon shown in Figure 18.6. Use the auto-reset timer developed in Exercise 2.9, but add an output for the elapsed time.

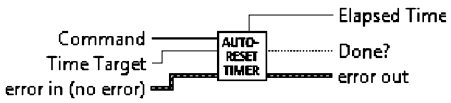


Figure 18.6: The icon of the `Auto Reset Timer With Elapsed Time` action engine ■

Exercise 6.8: Create a Mapping Sub-VI for the `Zone Indicators` Cluster

The sprinkler controller uses a ZONE SETUP ARRAY to manage the zone state transitions. One way to do this is to have a generic *Service Zone* state that services a given zone in

the array. Since this state can service any zone, a mapping sub-VI needs to be created to format the zone output based on the value of the ZONE SELECTOR. Create the `Light Zone Lamp` sub-VI with the icon shown in Figure 18.7 to perform this function. This sub-VI takes a TIMING input, a ZONE SELECTOR input, and a RUNNING? Boolean input and produces a correctly formatted ZONE INDICATORS cluster output.

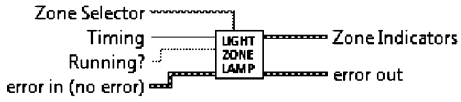


Figure 18.7: The icon of the `Light Zone Lamp` VI ■

Exercise 6.9: Create a Sub-VI to Increment the Zone Array Index

The sprinkler controller is similar to the Car Wash 2 controller. Both controllers use arrays to determine some of the state transitions, and both controllers need sub-VIs to handle the arrays. Both controllers also need to store an index value on the data highway to keep track of the current position in the array. Create an `Increment Index` sub-VI with the icon shown in Figure 18.8. This sub-VI increments the index if the current sprinkler zone is done. If the index exceeds the ZONE SETUP ARRAY size, the index is set to zero. The `Increment Index` sub-VI also outputs a SINGLE MODE DONE? Boolean indicator. If the new index exceeds the ZONE SETUP ARRAY size and the RUN SELECTOR switch is in single-program mode, this indicator is True; otherwise, it is False. You might find the **`Quotient & Remainder`** function useful (see the help file in Figure 18.9).

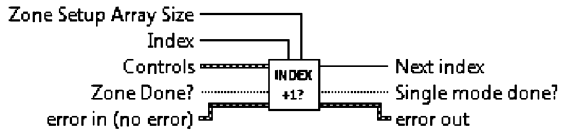


Figure 18.8: The icon of the `Increment Index` VI ■

Exercise 6.10: Create the `Service Zone` Sub-VI

We can now create the `Service Zone` sub-VI with the icon shown in Figure 18.10. This sub-VI selects the current zone from the ZONE SETUP ARRAY and performs the necessary time check using the sprinkler timer developed in Exercise 6.7. The output cluster for display by the main application is generated using the `Light Zone Lamp` sub-VI developed in Exercise 6.8. Remember to round the elapsed time to the nearest second as required by the functional specification. The index is then incremented (if needed) using the `Increment Index` sub-VI developed in Exercise 6.9.

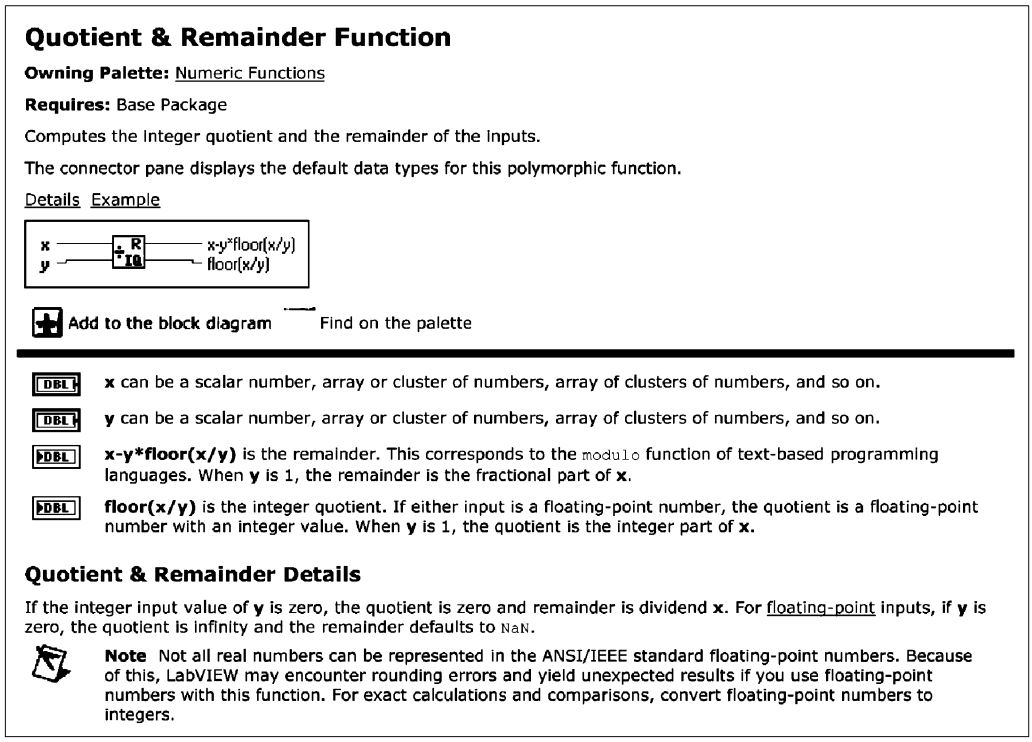


Figure 18.9: Help file for the `Quotient & Remainder` function

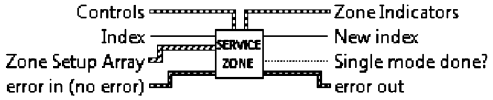


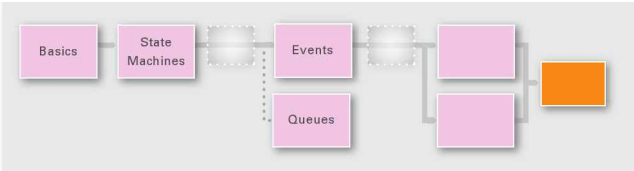
Figure 18.10: The icon of the `Service Zone` VI ■

Exercise 6.11: Create the Sprinkler Controller Main Application

Create the sprinkler controller main application using the state transition diagram, type definitions, and sub-VIs developed in the previous exercises of this problem set. ■

Exercise 6.12: Queues?

In Exercise 5.7, you were asked to convert the array-based version of the Car Wash 2 controller to a queue-based version. Using a queue simplified the main application by removing the array and index from the data highway and eliminating the need for the `Next Car Wash State` sub-VI. Would that approach be useful for the sprinkler controller? Why or why not? ■



19

Problem Set 7

This problem set guides you through a solution of the ATM controller sample Certified LabVIEW Developer exam. If you are preparing for the CLD exam, try to complete the ATM controller in four hours without reading the exercises in this problem set.

The ATM controller is a challenging exercise for a number of reasons. One of the challenges involves the personalization of the ATM messages. We need to develop a scheme for personalizing the messages using information available in the user's account. To do this, we develop a key-value parsing scheme. Message templates contain keys that act as placeholders for account information. The messages are parsed to identify the keys and replace them with the relevant account information.

Another challenge involves the ENTER button. The ENTER button is modal; the response to its value change depends on what mode it is in. There are four modes to consider:

Verify Account If the user presses ENTER after keying in an account number, the controller must verify the account.

Deposit If the user presses ENTER after keying in a deposit amount, the controller must make a deposit.

Withdraw If the user presses ENTER after keying in a withdrawal request, the controller must verify that the account has sufficient funds and then make the withdrawal.

Idle If the user presses ENTER at any other time, the controller must ignore the button press and take no action.

Read the functional specification in Appendix F carefully and keeping these challenges in mind. Then proceed with the following exercises.

Exercise 7.1: ATM Controller State Transition Diagram

Read the functional specification for the ATM controller in Appendix F. What states will the program need? How do they transition from one to another? Draw a state transition diagram for ATM controller. Pay attention to how the *Wait* state is handled

before and after a card is inserted. Add notes to the diagram to explain any actions that are difficult to capture in the diagram itself. Use the diagram to create a list of states for the ATM controller. ■

Exercise 7.2: Select an Architecture

Select a state machine architecture for the ATM controller. Would an event structure be useful? Would queues be useful? After you have selected an architecture consider the placement of the controls and indicators. Where should the CARD SIMULATOR button, the USER INPUT field, and the ENTER button be placed? Where should the left and right menu buttons be placed? Are these controls needed by multiple states? Do they change often? Do their values need to be stored?

Consider the indicators. Where should the ATM MESSAGE indicator and the left and right menu indicators be placed? Do they need to be updated by multiple states? Do they need to be updated frequently?

Consider persistent storage. Does the ATM controller need a data highway? If so, what should be stored on the data highway?

Consider the guard clause. Does it need to monitor anything other than the error status of the main error wire? ■

Exercise 7.3: ATM Controller Type Definitions

The ATM controller uses several type definitions. First, create an enumerated constant containing the list of states generated in Exercise 7.1. If you have decided to use the guard clause to monitor front panel activity, create an enumerated constant of states present in the guard clause.

The front panel has important clusters that should be saved as type definitions. Create a type-defined control containing three ATM buttons. This control is used for both the left and right ATM buttons. Create a type-defined control for containing three string indicators. This control is used for both the left and right side menu displays.

The ATM controller must personalize many different message types for each account holder, such as the *Welcome* message and the *Session Terminate* message. Create an enumerated constant that contains the names of all messages delivered by the ATM controller.

The ATM controller must deal with accounts. Accounts contain all necessary information about the ATM user. Account information is stored in a text file. Since account information is important and widely used in the controller, create a type-defined cluster that contains string controls for the data stored in the account: *First Name*, *Last Name*, *Balance*, and *Amount*. The *Amount* field is not stored in the text file. It is used to temporarily store deposit and withdrawal information.

The ENTER button of the ATM controller is modal; its function depends on the operation that is taking place when it is pressed. The ENTER button has four modes: *Idle*, *Deposit*, *Withdraw*, and *Verify Account*. Create a type-defined enumerated constant containing these modes. ■

One of the challenges of the functional specification is parsing the ATM messages. The ATM messages need to be personalized with the following information: first name

of the account holder, last name of the account holder, the current account balance, and the amount of the current transaction. This can be done using a key-value system. The keys will be pre-defined placeholders in the ATM message templates that will be replaced by the corresponding values from the account and transaction history. The resulting message will be displayed on the ATM.

To do this, we need to define a convention for identifying the keys in a given message. We use the convention where keys are enclosed by square brackets "[]". For example, the template of the main menu message would read: "Welcome to Acme Bank [First Name] [Last Name]. Please select transaction by using the buttons." The balance inquiry message template would read "Your Balance Is: $[Balance]." In these examples, the bracketed key values would be replaced by the proper account information before the message was displayed. There are four keys in the set of ATM messages: [First Name], [Last Name], [Balance], and [Amount].

Exercise 7.4: Create the `Get Message Templates` Sub-VI

Create the `Get Message Templates` sub-VI with the icon shown in Figure 19.1. The message templates are retrieved by this sub-VI. This sub-VI accepts as an input the enumerated constant listing the message types created in Exercise 7.3. The output of the sub-VI is the specified message template containing keys in square brackets.



Figure 19.1: The icon of the `Get Message Templates` VI ■

Exercise 7.5: Create the `Replace Keys With Values` Sub-VI

Create the `Replace Keys With Values` sub-VI with the icon shown in Figure 19.2. This sub-VI accepts a message template with keys contained in square brackets as an input. It also accepts an array of bracketed keys and a corresponding array of values as inputs. This sub-VI will iterate through the array of keys, searching the message template for all instances of each key and replacing them with the corresponding value. You might find the **`Search and Replace String`** function useful (see the help file in Figure 19.3).

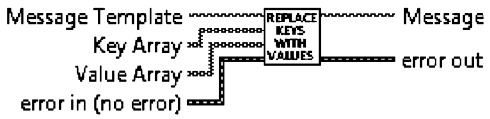


Figure 19.2: The icon of the `Replace Keys With Values` VI

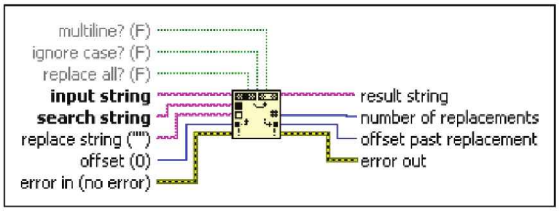
Search and Replace String Function

Owning Palette: String Functions

Requires: Base Package

Replaces one or all instances of a substring with another substring. To include the **multiline?** Boolean input, right-click the function and select **Regular Expression**.

Details Example



Add to the block diagram Find on the palette

multiline? sets whether or not to treat the text in **input string** as a multiple-line string. This affects how the ^ and $ characters handle matches. When FALSE (the default), entering "^" matches the beginning of the line in **input string** only and entering "$" matches the end of **input string** only. When TRUE, "^" matches the beginning of any line in **input string** and "$" matches the end of any line in **input string**.

ignore case? specifies whether the string search is case sensitive. If FALSE (default), the string search is case sensitive.

replace all? specifies whether the function should replace all occurrences of the **search string** or only the first occurrence. If FALSE (default), the function replaces only the first occurrence.

input string is the input string the function searches.

search string is the string you want to search for and replace. If **search string** is an empty string and **replace all?** is FALSE, the function inserts **replace string** at the beginning of **input string**. If **search string** is an empty string and **replace all?** is TRUE, the function inserts **replace string** at the beginning, in between each character, and at the end of **input string**.

replace string replaces the string in **search string**. The default is an empty string.

offset determines the number of characters into **input string** at which the function starts searching for **search string**.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

result string contains the **input string** with one or all occurrences of **search string** replaced with **replace string**. If **replace string** is empty, **result string** contains **input string** with **search string** deleted.

number of replacements returns the number of times LabVIEW replaced **search string**.

offset past replacement indicates the offset in **result string** of the character immediately following where the last match and replacement occurred. If **replace all?** is FALSE, the next match, if any, is after this point.

If the function does not find **search string**, **offset past replacement** is -1.

error out contains error information. This output provides standard error out functionality.

Figure 19.3: Help file for the `Search and Replace String` VI ■

Exercise 7.6: Create the `Get Message` Sub-VI

Create the `Get Message` sub-VI with the icon shown in Figure 19.4. This sub-VI accepts an account cluster and a message type as inputs and it produces a personalized message as an output. This sub-VI will call the `Get Message Templates` sub-VI

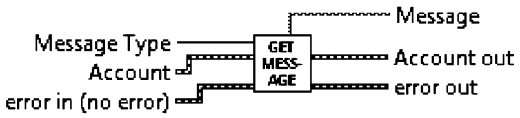


Figure 19.4: The icon of the `Get Message` VI

developed in Exercise 7.4 to get the correct message template. It will then construct the key and value arrays and pass them to the `Replace Keys With Values` sub-VI developed in Exercise 7.5 to generate the personalized message. ■

Most of the activity of the ATM controller involves user accounts. The accounts are stored in a comma-separated value (CSV) text file. Each account is given a separate row in the file. Each account consists of a five digit account number, first name, last name, and current balance (in that order). When the balance is changed, the file needs to be updated. We operate under the assumption that the record does not need to be locked during the operation of the program. We assume that the account only can be accessed from one ATM machine at a time. This is a significant assumption, but designing a security system for locking one record of a text file that is a shared resource for a network of ATMs is not in the scope of the functional specification.

The account information is represented in the controller as a type-defined cluster created in Exercise 7.3. The cluster contains five string controls: *Account Number*, *First Name*, *Last Name*, *Balance*, and *Transaction*. The first four of these items are recorded in the accounts file. The *Transaction* field is used to store pending deposit and withdrawal amounts. In the following exercises, you will develop the sub-VIs needed to manage the account file. You will also develop sub-VIs to manipulate the the account information through deposits and withdrawals.

Exercise 7.7: Create the `Get Account` Sub-VI

Create `Get Account` sub-VI with the icon shown in Figure 19.5. This sub-VI accepts an account number string and the account file path as inputs. It returns the user account cluster, its position in the file, and a Boolean indicating if an account was found as outputs.

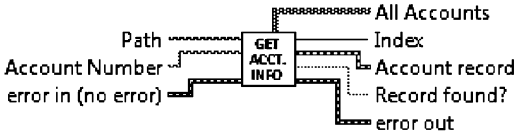


Figure 19.5: The icon of the `Get Account` VI ■

Exercise 7.8: Create the `Verify Account` Sub-VI

Create the `Verify Account` sub-VI with the icon shown in Figure 19.6. This sub-VI accepts a path to the account file and an account number string as inputs. It searches

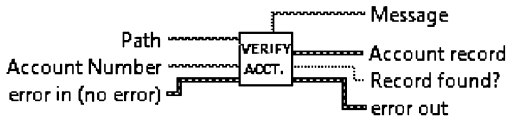


Figure 19.6: The icon of the `Verify Account` VI

the account file to see if the input account number exists within the file and outputs a RECORD FOUND? Boolean indicating the result of the search. This sub-VI also generates the appropriate ATM message informing the user whether or not the account was found. The account record is also produced as an output. ■

Exercise 7.9: Create the `Update Account File` Sub-VI

Create the `Update Account File` sub-VI with the icon shown in Figure 19.7. This sub-VI takes an account cluster and the account file path as inputs. It opens the account file, finds the record corresponding to the account number contained in the input account cluster, and updates the record using the information found in the input account cluster. Use the `Get Account` sub-VI developed in Exercise 7.7 to read the account file and generate an array of all account information, replace the old account information with the new account information, and then overwrite the accounts file with the updated array. You might find the **`Replace Array Subset`** function useful (see the help file in Figure 19.8), as well as the **`Write To Spreadsheet File`** function (see the help file in Figure 19.9).



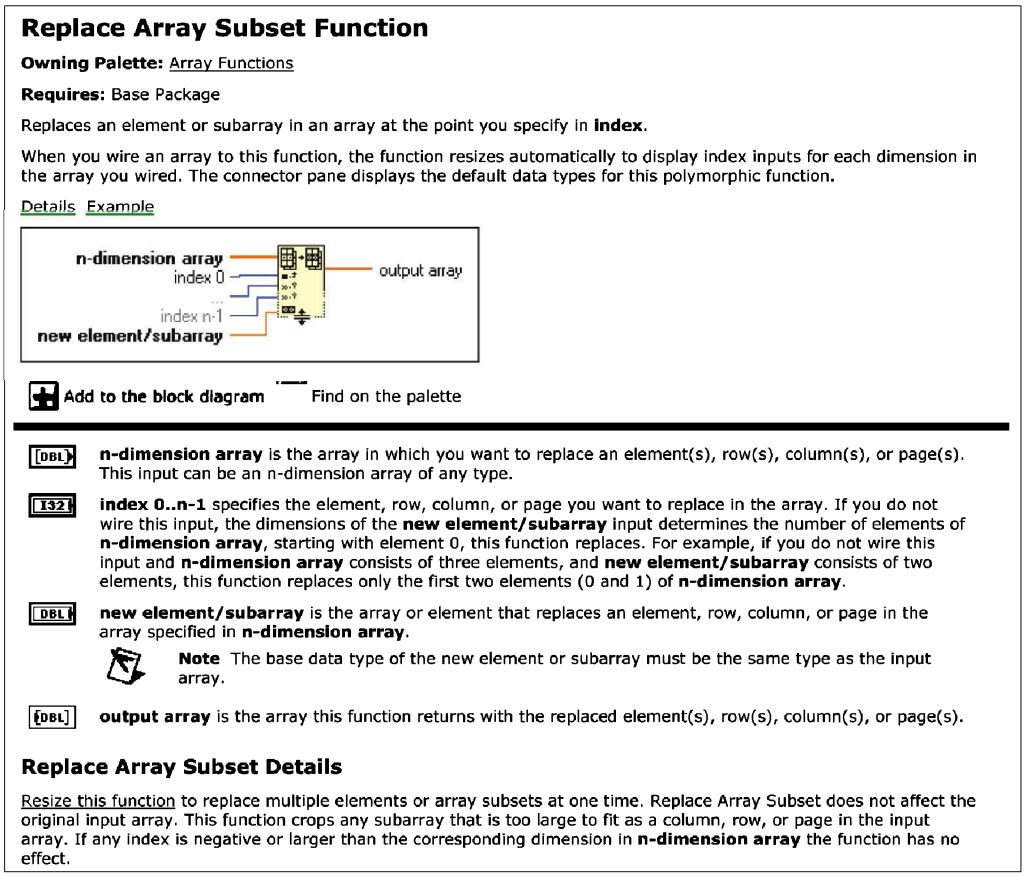
Figure 19.7: The icon of the `Update Account File` VI ■

Exercise 7.10: Create the `Deposit` Sub-VI

Create the `Deposit` sub-VI with the icon shown in Figure 19.10. This sub-VI accepts an account cluster and a deposit amount string as inputs. The deposit amount is added to the balance of the account and updates the balance of the account cluster. This sub-VI should also store the deposit amount in the *Transaction* field of the account cluster. ■

Exercise 7.11: Create the `Withdrawal` Sub-VI

Create the `Deposit` sub-VI with the icon shown in Figure 19.11. This sub-VI accepts an account cluster and a withdrawal amount string as inputs. This sub-VI subtracts the withdrawal amount from the balance of the account and updates the account cluster if there are sufficient funds in the account. The withdrawal amount also should be written to the *Transaction* field of the account cluster. If the balance is not large enough to permit the withdrawal, the account information is passed through unchanged. A SUFFICIENT FUNDS? Boolean is also output indicating if the withdrawal was successful or not. ■



Replace Array Subset Function

Owning Palette: Array Functions

Requires: Base Package

Replaces an element or subarray in an array at the point you specify in **index**.

When you wire an array to this function, the function resizes automatically to display index inputs for each dimension in the array you wired. The connector pane displays the default data types for this polymorphic function.

Details Example

Add to the block diagram Find on the palette

n-dimension array is the array in which you want to replace an element(s), row(s), column(s), or page(s). This input can be an n-dimension array of any type.

index 0..n-1 specifies the element, row, column, or page you want to replace in the array. If you do not wire this input, the dimensions of the **new element/subarray** input determines the number of elements of **n-dimension array**, starting with element 0, this function replaces. For example, if you do not wire this input and **n-dimension array** consists of three elements, and **new element/subarray** consists of two elements, this function replaces only the first two elements (0 and 1) of **n-dimension array**.

new element/subarray is the array or element that replaces an element, row, column, or page in the array specified in **n-dimension array**.

Note The base data type of the new element or subarray must be the same type as the input array.

output array is the array this function returns with the replaced element(s), row(s), column(s), or page(s).

Replace Array Subset Details

Resize this function to replace multiple elements or array subsets at one time. Replace Array Subset does not affect the original input array. This function crops any subarray that is too large to fit as a column, row, or page in the input array. If any index is negative or larger than the corresponding dimension in **n-dimension array** the function has no effect.

Figure 19.8: Help file for the `Replace Array Subset` VI

Exercise 7.12: Create the `Enter Button Next State` Sub-VI

Create the `Enter Button Next State` sub-VI with the icon shown in Figure 19.12. This sub-VI handles the modal nature of the ENTER button. It accepts the enumerated constant containing the modes of the ENTER button as an input, and it outputs the next state following the value change of the enter button. If the current mode of the ENTER button is *Verify Account*, then a value change of the ENTER button signifies that the user entered an incorrect account number and is trying again. The next state is *Verify Account*. If the current mode of the ENTER button is *Deposit*, a value change of the ENTER button signifies that the user is entering the deposit amount, and the next state is *Complete Deposit*. If the current mode of the ENTER button is *Withdraw*, then a value change of the ENTER button signifies that the user is entering the withdrawal amount, and the next state is *Complete Withdrawal*. If the current mode of the ENTER button is *Idle*, then a value change of the ENTER button should be ignored and the next state is *Wait*. ■

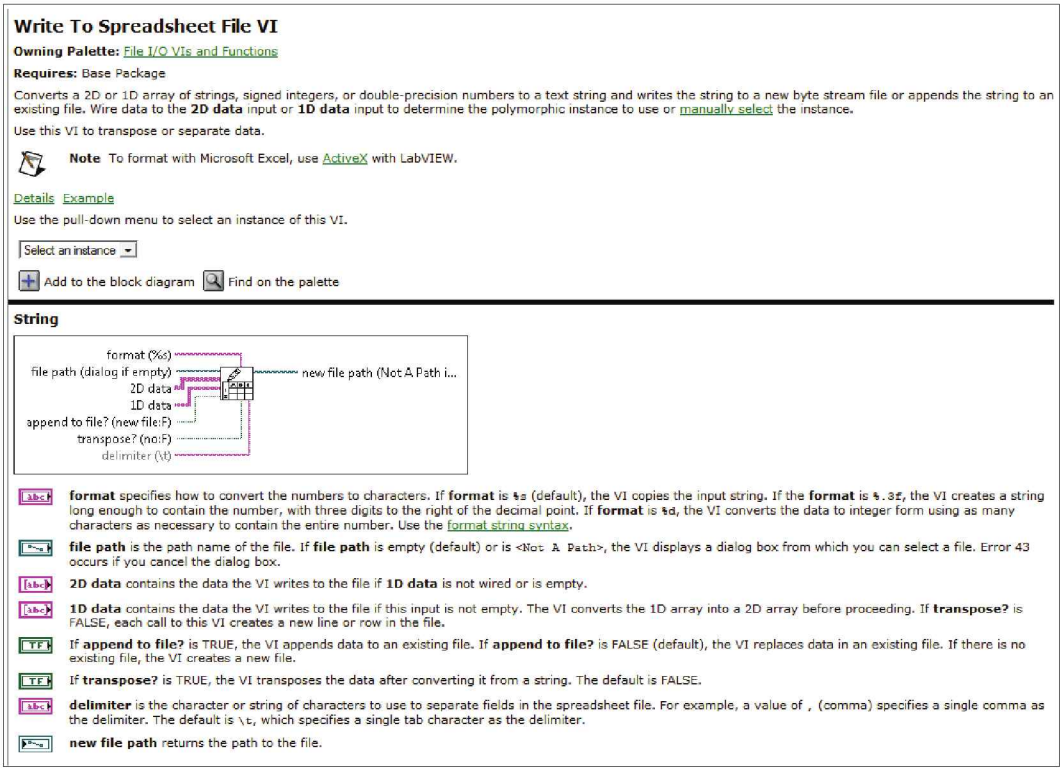


Figure 19.9: Help file for the `Write To Spreadsheet File` VI

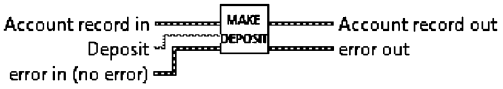


Figure 19.10: The icon of the `Deposit` VI

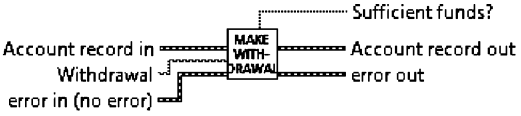


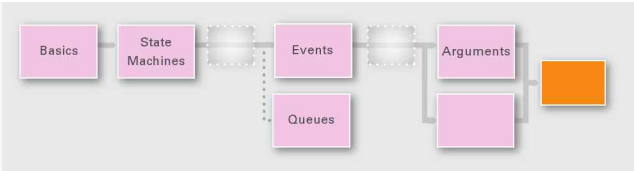
Figure 19.11: The icon of the `Withdrawal` VI



Figure 19.12: The icon of the `Enter Button Next State` VI

Exercise 7.13: Create the ATM Controller

Now that all of the sub-VIs have been created for handling message personalization, account manipulation, and the modal ENTER button, create the event-driven classic state machine implementation of the ATM controller. Place the ATM side menus on the data highway to make the application more scalable in the future. ■



20

State Machines with Arguments

Queued, event-driven state machines are powerful, flexible, and efficient and are applicable to a wide range of problems. In this chapter, we look at an upgrade to this design pattern that is not as widely applicable but is powerful when it can be used—the state machine with arguments.

States in a state machine are very much like sub-VIs. The code within a state is self-contained—just like a sub-VI. Dequeuing a state and executing it is much like calling a sub-VI. A state can be repeated over and over again—just like a sub-VI can be called repeatedly.

Information can be passed to a state—just like inputs or arguments—can be passed to a sub-VI. For example, in the car wash controller, the proximity switch cluster is passed as an input to the car wash cycle states. The main error wire and the queue reference wire also enter each state as inputs.

But what if we could pass other kinds of information to states as arguments? In the car wash controller, the five car wash cycle states are nearly identical (*Underbody Wash*, *Soap Application*, *Main Wash*, *Rinse*, and *Air Dry*). The only differences in these states are which proximity switch is monitored, what time target is used, and which indicators are lit. Imagine if we could pass that information to a generic *Cycle* state as arguments. Then we could replace five specific car wash cycle states with a single generic *Cycle* state that accepts arguments. This would be a big improvement in readability, maintainability, and scalability.

In this chapter, we examine different schemes for passing information to states as arguments using the data highway, queued clusters, and queued strings. We practice on the `Multitest` VI and then develop a full-blown car wash controller with arguments in Chapter 21.

20.1 The Data Highway

State machines use the data highway to pass information from one iteration of the main loop to the next, but it also can be used to pass information from one state to another, since the data highway can be accessed by all of the states in a state machine.

As an example of this, let's revise the queued `Multitest` VI developed in Chapter 13. Recall that in this implementation of the `Multitest` VI, there are separate states for running Test 1 and Test 2. The only difference between these states is the value of the string displayed by the one-button dialog. We can replace the *Run Test 1* and *Run Test 2* states with a single generic *Run Test* state if we use the data highway to store the value of the dialog string.

Figure 20.1 shows the new generic *Run Test* state of the revised `Multitest` VI. Notice the addition of a data highway to store the value of the *Test ID* string. The value of the *Test ID* string is retrieved from the data highway and is displayed to the user via the **`One Button Dialog`** function.

Style Guide: Notice that the data highway wire has a label. Always label wires on the data highway to improve readability of your code.

Figure 20.2 shows the event case handling the value change of the RUN TEST 1 button of the revised `Multitest` VI. The generic *Run Test* state is enqueued instead of the *Run Test 1* state used in Chapter 13. The `Run Test 1` string is placed on the data highway for use by the *Run Test* state.

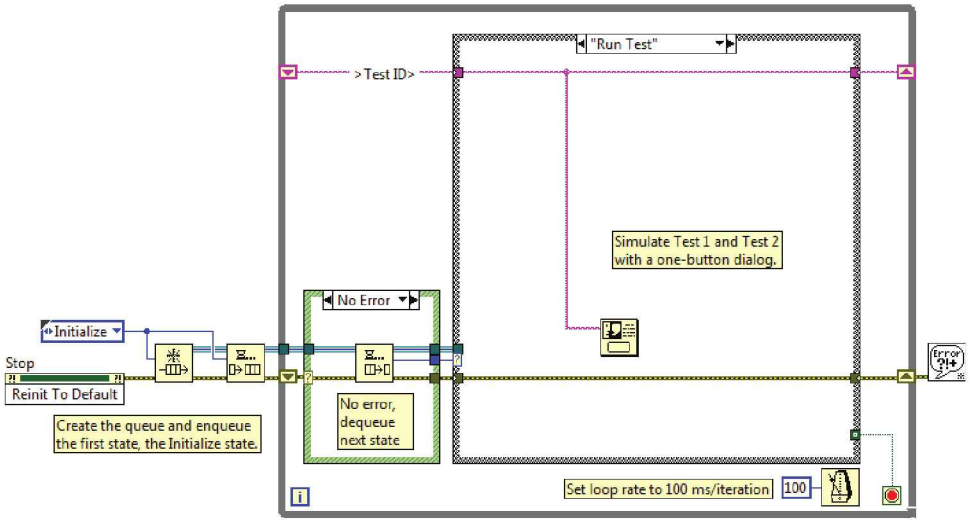


Figure 20.1: The *Run Test* state of the revised `Multitest` VI

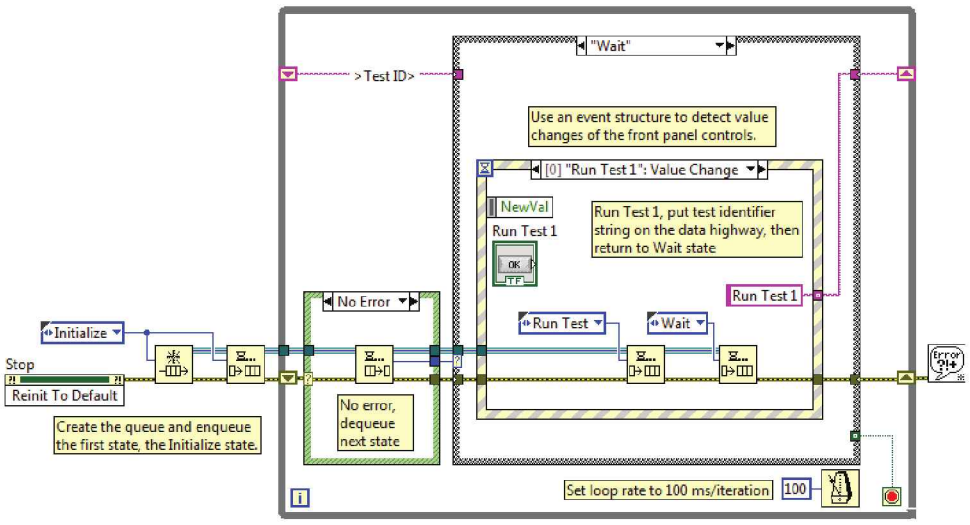


Figure 20.2: The event case handling the value change of the RUN TEST 1 button

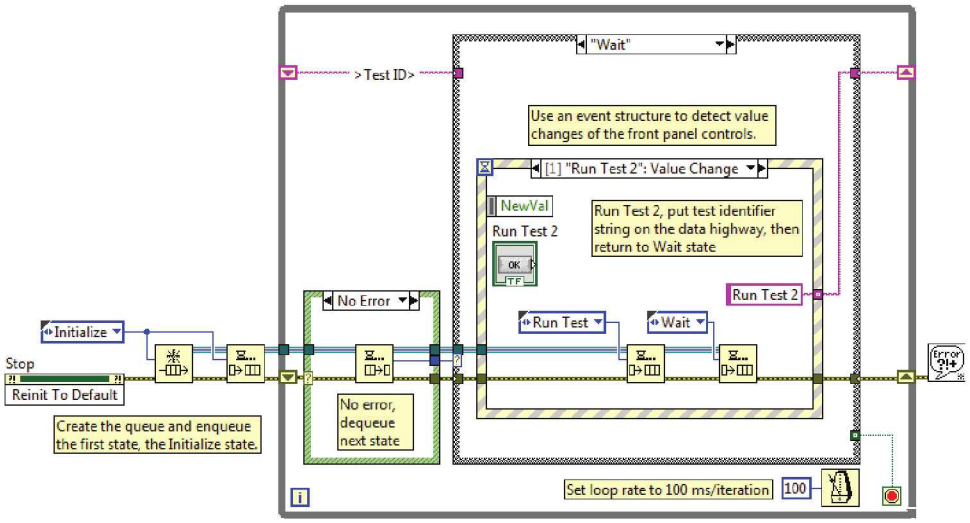


Figure 20.3: The event case handling the value change of the RUN TEST 2 button

Figure 20.3 shows the corresponding event case for the value change of the RUN TEST 2 button. This case enqueues the same *Run Test* state and places the `Run Test 2` string on the data highway for use by the *Run Test* state. The data highway acts as short-term storage for the *Test ID* string. The event cases handling the value changes of the RUN TEST 1 and RUN TEST 2 buttons use the data highway to pass the *Test ID* string as an argument to the generic *Run Test* state.

Wiring Tip: In this implementation of the `Multitest` VI, we removed the *Run Test 1* and *Run Test 2* states and replaced them with the *Run Test* state. Changing the states means changing the enumerated constant listing the states. This demonstrates why the enumerated constant should be saved as a type-definition. Changing the states in the type definition file automatically changes the states in all of the enumerated constants used in the `Multitest` VI without breaking any wires. If the enumerated constants were not saved as type definitions, the wires would break as soon as one of the constants was changed. Each constant would then have to be changed manually.

20.2 Using Arguments to Pass Information

The previous data highway approach worked because only one argument needed to be passed to one state at a time. But what if multiple arguments needed to be passed? What if a series of states required arguments? What if different states required arguments with various data types or various numbers of arguments?

The data highway would be awkward in such a scenario. The number of wires would multiply, or arrays or clusters of data would need to placed on the data highway with each state accessing a different element. This would degrade the readability, scalability, and maintainability of the code. A better method needs to be used that targets each state with the individual arguments needed by that state. We need a way to "bundle together" the state name and its arguments into a single data package that can be processed by a state machine. This idea of bundling gives a clue as how to proceed with adding arguments to a LabVIEW state machine.

20.3 Passing Arguments Using Clusters

In Chapters 13 and 14, we used a queue to store state information. That queue element was a type-defined enumerated constant containing the names of the possible states. But queue elements are not limited to enumerated constants. Queue elements can be used to carry both the name of the next state and the arguments used by that state.

One way to do this is to use a cluster as the queue element. The cluster bundles the state enumerated constant with the needed arguments. When the cluster is dequeued, the enumerated constant can be unbundled to determine the next state, and the arguments can be unbundled and passed as inputs to the main case structure of the state machine.

Let's take the `Multitest` VI as an example. Once again, the goal is to consolidate the *Run Test 1* state and the *Run Test 2* state into a generic *Run Test* state that accepts a string as an argument. The queue element will be a cluster of the enumerated constant containing the states and the string argument, as shown in Figure 20.4.



Figure 20.4: Cluster of state and argument used in the cluster-based queue of the `Multitest` VI

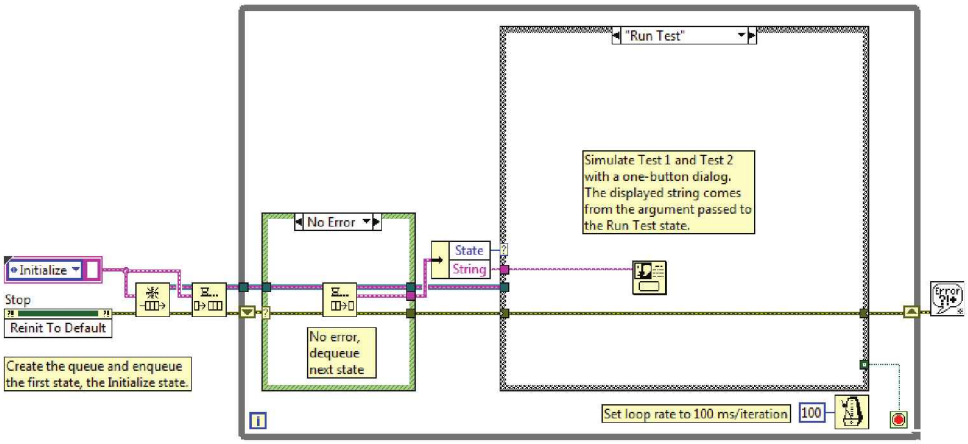


Figure 20.5: The *Run Test* state of the state machine with arguments implementation of the `Multitest` VI

Wiring Tip: It is usually a good idea to create type definitions of important clusters in a VI. In this case, the cluster that forms the basis of the queue that drives the `Multitest` state machine certainly qualifies as an important cluster. It should be saved as a type definition.

The implementation of the argument-based `Multitest` VI is straightforward. The overall structure of the cluster-based state machine with arguments is shown in Figure 20.5. The queue is created outside the loop (as in Chapter 14), but this time the cluster in Figure 20.4 is used as the queue data type.

After the queue has been created, the first element is enqueued: a cluster of the *Initialize* state and a blank string constant. This illustrates a drawback of the cluster-based queue. The *Initialize* state doesn't need a string argument, but it must be included anyway. The queue element has the data type of a cluster consisting of an enumerated constant and a string constant, and every element in the queue must have the same data type. The string constant must be included even when it is not needed.

The queue reference is then passed into the main loop. The guard clause checks to see if an error is present. If the guard clause detects no error, the next element is dequeued. The **`Unbundle By Name`** function is then used between the guard clause and the main case structure to unbundle the state information and the string argument. The state information is wired to the case structure selector to determine the next state.

The string argument is passed into the case structure as an input. The *Run Test* state shown in Figure 20.5 accepts the string as an argument and displays it to the user.

If the guard clause detects an error, the queue is bypassed, and the state machine transitions to the *Shutdown* state, as shown in Figure 20.6. The *Shutdown* state does not need an argument, so the string input tunnel is left unwired. Even though the *Shutdown* state needs no arguments, the cluster in the guard clause must still include the string constant.

The event cases handling the value change of the RUN TEST 1 or the RUN TEST 2 buttons are shown in Figures 20.7 and 20.8. Each of these states combines the *Run Test* state with its argument into a single cluster and then places that cluster on the queue.

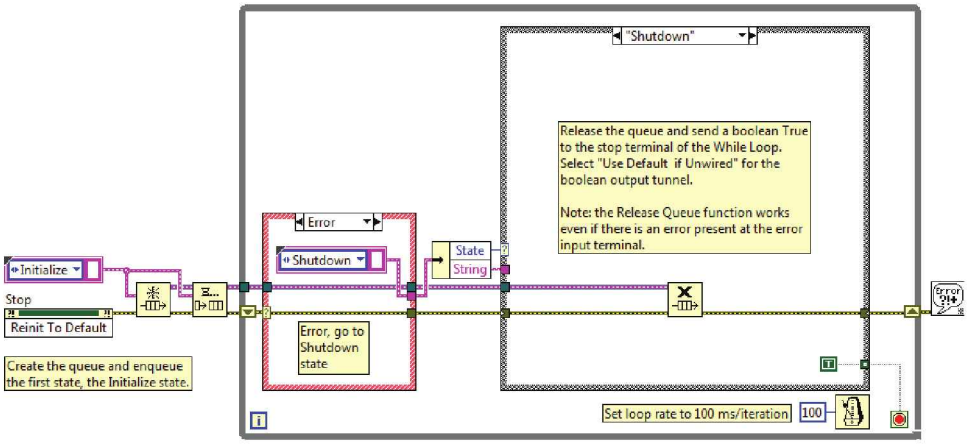


Figure 20.6: The *Shutdown* state of the state machine with arguments implementation of the `Multitest` VI

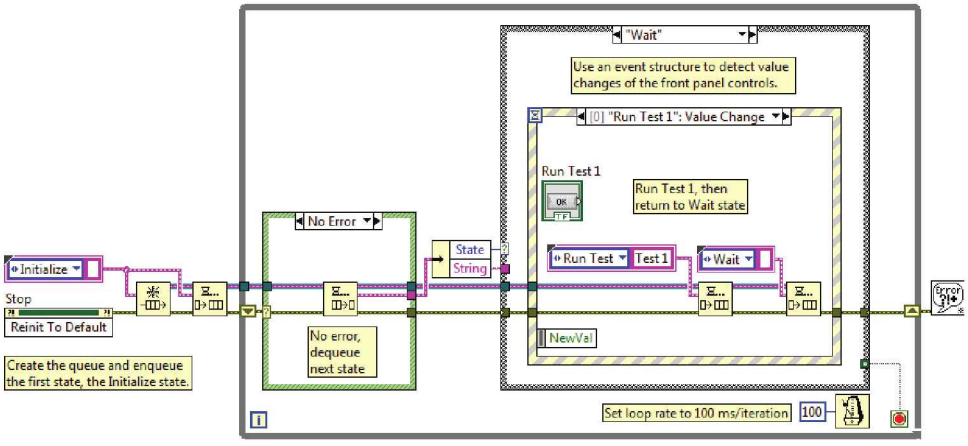


Figure 20.7: The event case handling the value change of the RUN TEST 1 button of the state machine with arguments implementation of the `Multitest` VI

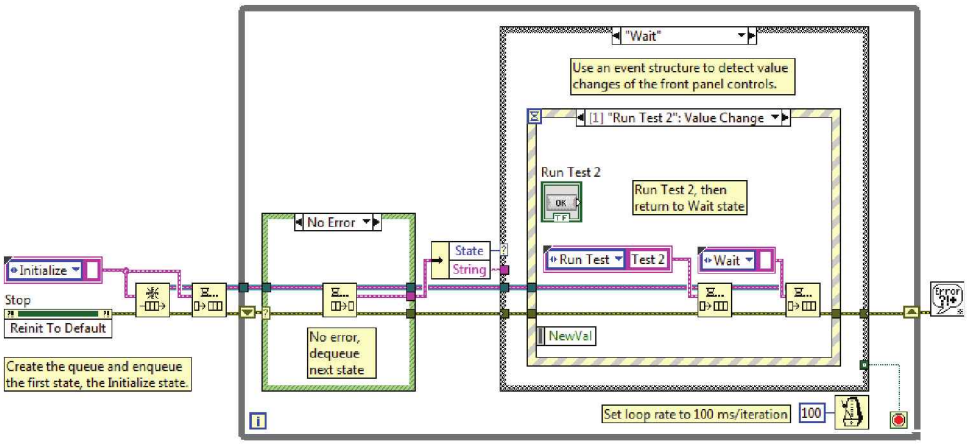


Figure 20.8: The event case handling the value change of the RUN TEST 2 button of the state machine with arguments implementation of the `Multitest` VI

20.4 Passing Arguments Using Strings

Clusters are an easy way to pass arguments to states, but they are limited by the fact that clusters have fixed data types. This means that the arguments always have to be included even if they aren't needed for a particular state. When clusters with arguments are used to create a queue, the arguments are never optional. This problem gets worse as the number of arguments increases. Clusters with large numbers of dummy arguments take up valuable screen real estate.

Another issue arises if different states need different types of arguments. Since the data type of the cluster is fixed, all of the required argument data types would need to be included in the queued cluster. For instance, if one state needs a Boolean argument and another state needs a numeric argument, the queued cluster would need to include both a Boolean and a numeric constant as well as the state enumerated constant. One state would use the Boolean argument and ignore the numeric argument, and the other state would use the numeric argument and ignore the Boolean argument.

One way around these difficulties is to use strings as the base element of the queue. Strings also have a fixed data type, but strings are more malleable than clusters. LabVIEW has many native functions for manipulating, splitting, searching, and concatenating strings. A clever LabVIEW developer can use these functions to create and parse strings that contain both state information and arguments in any format the developer desires.

20.5 Designing Parsing Sub-VIs

As an example, let's adopt a simple programming convention for separating state information from arguments in a string using a delimiter. If we use a colon as a

delimiter, we could construct a string with state information and an argument using the format `State:Argument`. The string consists of two pieces of information separated by a colon with the state first and the argument following the state.

A parsing routine could be written that accepts a string with this format as an input. The routine splits the state from the argument, outputting them as separate strings. If no delimiter is found in the input string (and thus no argument), a null string is output as the argument string.

This scheme can be implemented using the native **`Match Pattern`** function whose help file is shown in Figure 20.9. The **`Match Pattern`** function takes a string and a regular expression as inputs. The outputs include the portion of the string before the regular expression, the matching string, and the portion after the regular expression.

Regular expressions are dense, compact, and very powerful statements used for parsing text. They are beyond the scope of this book, but for our purposes, a regular expression can be thought of as a pattern string to be matched in a larger string.

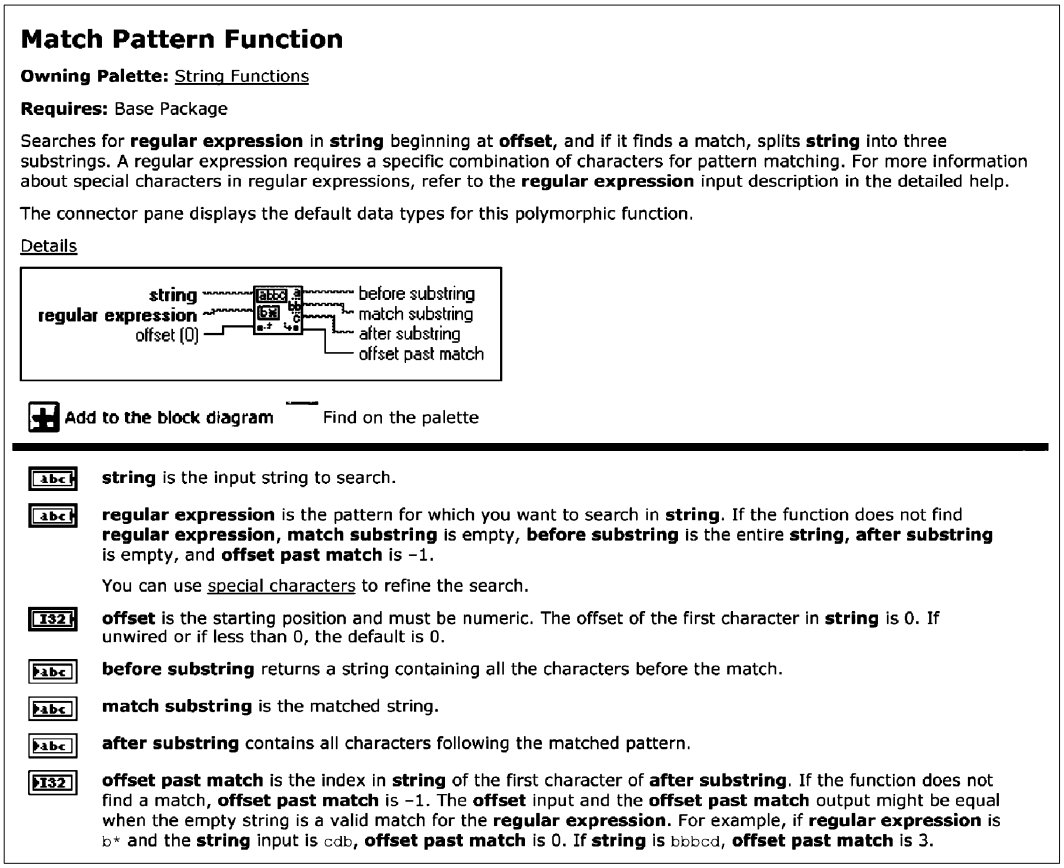


Figure 20.9: Help file for the `Match Pattern` function

To search for a colon, the regular expression is simply a colon, ":". The "before substring" output of the **Match Pattern** VI corresponds to the state, and the "after substring" output corresponds to the argument. Note that the **Match Pattern** function only finds the first instance of the search string. This will be an important consideration in the next chapter when we extend our parsing routine to handle strings with multiple arguments.

A parsing sub-VI, `Parse State and Arguments`, can be developed using the **Match Pattern** function. The icon for this sub-VI is shown in Figure 20.10. The VI takes a string as an input, splits the string at the first colon found, and returns the portions of the string before and after the colon. This is illustrated by the front panel of the sub-VI, shown in Figure 20.11. The block diagram to accomplish this task is shown in Figure 20.12. As seen in the figures, this sub-VI is simply a wrapper of the **Match Pattern** function designating a colon as the delimiter.

What happens if the `Parse State and Arguments` VI is given a string without a colon as an input? If the **Match Pattern** function cannot find the pattern string, then the "before substring" will equal the input string and the "after substring" will be the empty string. This is desired behavior for our parsing sub-VI. If the input string has no arguments, such as `Initialize`, the state output will be `Initialize` and the argument output will be the empty string.



Figure 20.10: The icon of the `Parse State and Arguments` sub-VI



Figure 20.11: The front panel of the `Parse State and Arguments` sub-VI

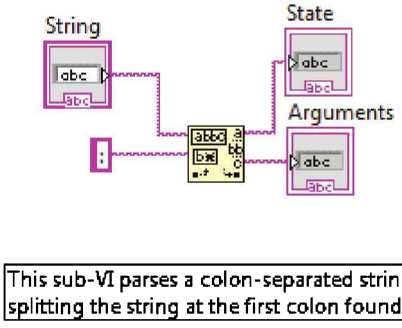


Figure 20.12: The block diagram of the `Parse State and Arguments` sub-VI

20.6 The String-Based `Multitest` VI

The parsing scheme described previously now can be applied to the `Multitest` VI. Let's look at the overall structure of the `Multitest` VI with a string-based queue by examining Figure 20.13. This figure shows the block diagram of the `Multitest` VI in the *Initialize* state.

As usual, the queue is created outside the main loop of the state machine. The `Initialize` string is used to specify the data type of the queue element and is also enqueued as the first element. Notice that there are no arguments attached to this string. Contrast this with the cluster-based queue element used to set the data type of the queue in Figure 20.5. Even though the *Initialize* state needs no arguments, the cluster in the input section of the loop must include a dummy argument for the test string. In the string-based scheme, the arguments are optional.

The guard clause then checks to see if an error has occurred. When no error is present, the next state is dequeued, parsed, and sent to the main case structure of the state machine. When an error is present, the queue is bypassed and the *Shutdown* state is sent to the main case structure of the state machine, as seen in Figure 20.14. The `Stop` string in the guard clause needs no arguments.

After the guard clause, we find the `Parse State and Arguments` sub-VI. This replaces the **`Unbundle By Name`** function used in the cluster-based version of the state machine with arguments. This sub-VI parses the input string and separates it into a state string and an argument string. The state string is wired to the selector terminal of the case structure, and the argument string is passed to the case structure as an input.

The *Initialize* case shown in Figure 20.13 resets the front panel controls and queues up the *Wait* state. The *Wait* state needs no argument, so none is provided. The *Wait* state contains the event structure that waits for user input, such as pressing

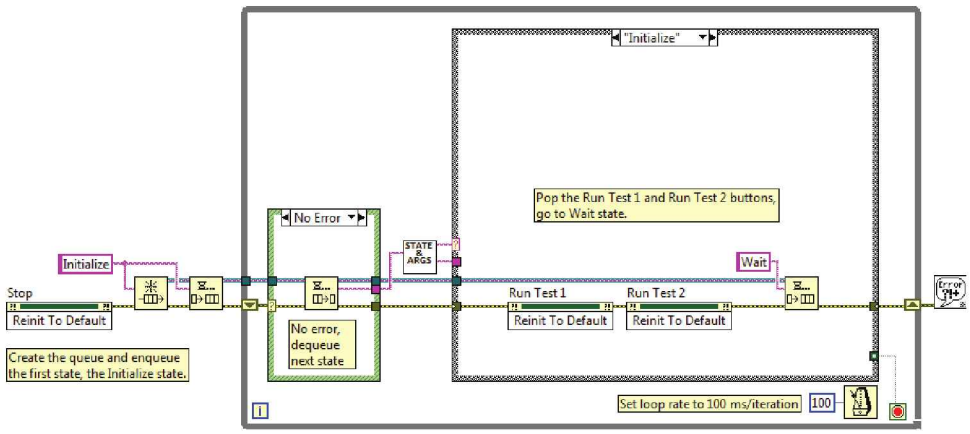


Figure 20.13: Overall structure of a queued string state machine implementation of the `Multitest` VI example where the *No Error* case of the guard clause is shown

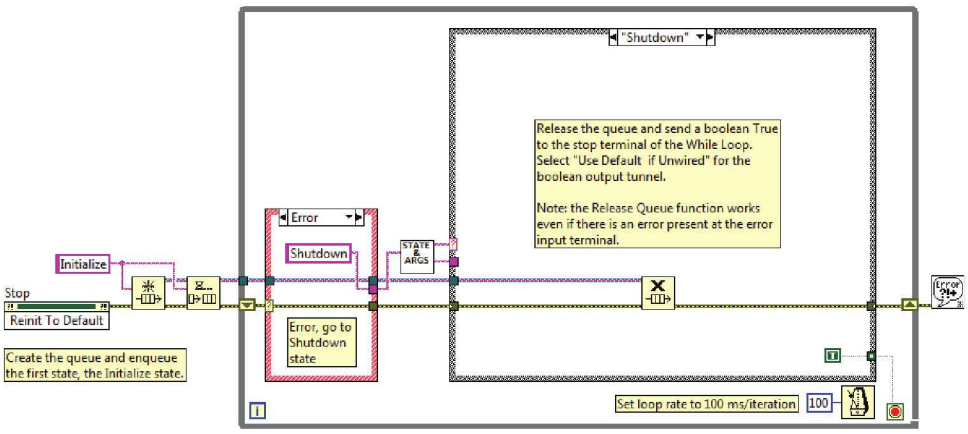


Figure 20.14: The *Shutdown* state of the queued string `Multitest` VI, where the *Error* case of the guard clause is shown

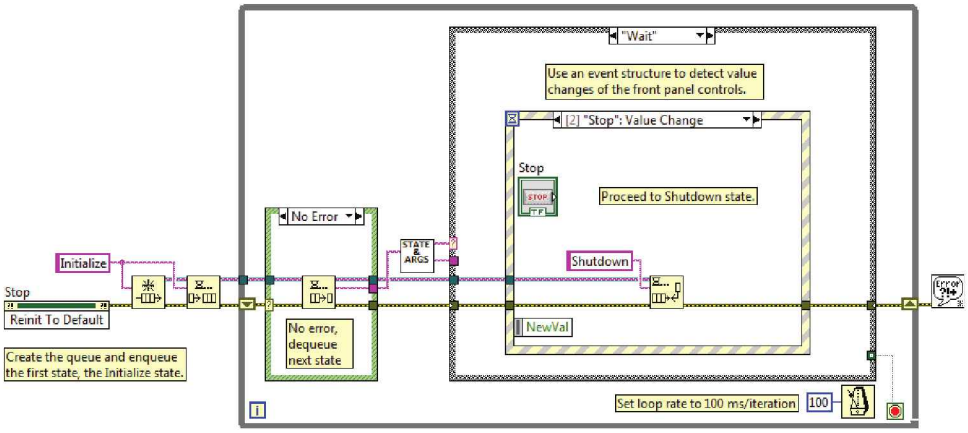


Figure 20.15: The event case handling the value change of the STOP button of the queued string `Multitest` VI

the STOP button. The event case handling the value change of the STOP button is shown in Figure 20.15. The *Shutdown* case of the state machine was shown in Figure 20.14.

The event case handling the value change of the RUN TEST 1 button is shown in Figure 20.16. Two strings are enqueued: `Run Test:Test 1` and `Wait`. Compare this with the cluster-based implementation in Figure 20.7. Notice how the string-based version is more readable. The block diagram has a "lighter" feel to it, and the elements are simple strings—not clusters. The border of the string elements are thinner, and there is less clutter within the elements. Elements that don't need arguments don't

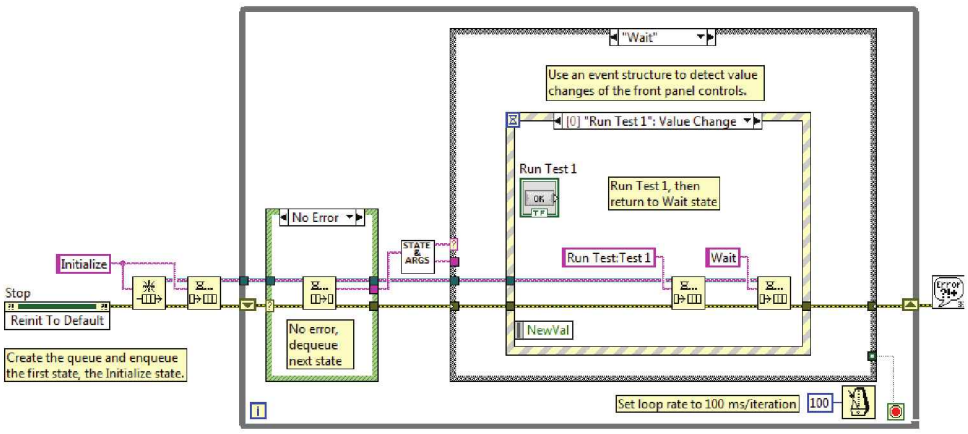


Figure 20.16: The event case handling the value change of the RUN TEST 1 button of the queued string `Multitest` VI example

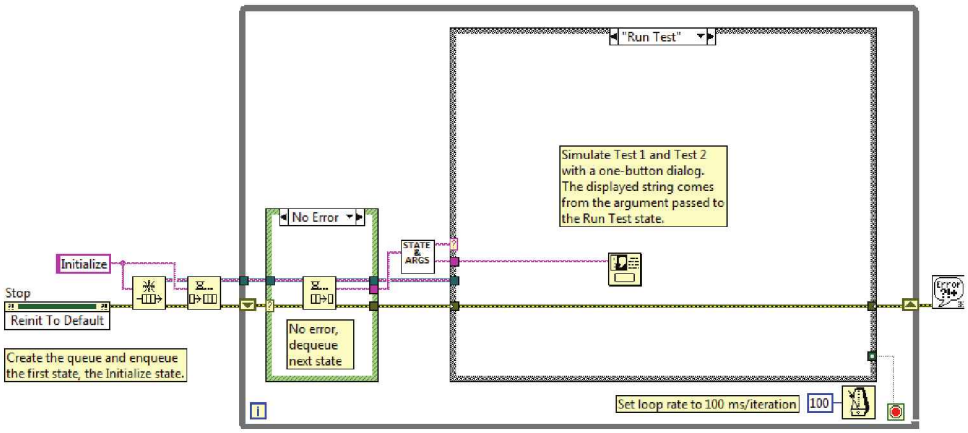


Figure 20.17: The *Run Test* state of the queued string `Multitest` VI

have arguments. All of these factors increase the readability of the code. This increase in readability will be even more evident when we apply a string-based queue to the car wash controller in the next chapter.

After the `Run Test:Test 1` and `Wait` strings have been enqueued, the state machine moves on to the next iteration. When the `Run Test:Test 1` string is dequeued and parsed, the `Run Test` string is passed to the selector of the case structure, and the `Test 1` string is passed to the input tunnel of the case structure as an argument. The *Run Test* state is shown in Figure 20.17.

The event case handling the value change of the RUN TEST 2 button, shown in Figure 20.18, is much the same as the case handling the RUN TEST 1 button. The only

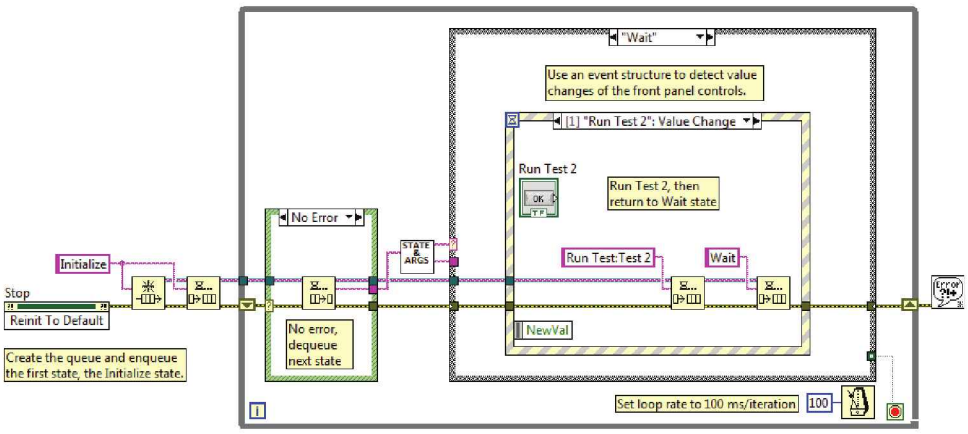


Figure 20.18: The event case handling the value change of the RUN TEST 2 button of the queued string `Multitest` VI

difference is that the argument of the of the *Run Test* state is `Run 2` instead of `Run 1`. A single *Run Test* state can be used for running either test.

20.7 Disadvantages of Using Strings

String-based queues have two disadvantages when compared to queues using clusters and enumerated constants. The major disadvantage is that you have to parse the strings yourself. The cluster-based state machine used the **`Unbundle By Name`** function to separate the state and arguments. To use strings, you must first decide on a parsing convention to separate the states and arguments. Then you must build the support Vis to perform the parsing.

A second disadvantage is that case structures in LabVIEW are not optimized to work with strings as they are with enumerated constants. When an enumerated constant is wired to a case structure, you can right-click on the case selector and choose *Add Case for Every Value* to automatically populate the case structure with every value in the enumerated constant. This is a good defensive programming technique. When an enumerated constant is used with a case structure that has been populated in this way, there is very little danger of an unknown state being sent to the selector terminal of the case structure.

This works because LabVIEW knows all of the possible values of an enumerated constant before the code runs. This is not true for strings. An enumerated constant has a fixed range of values, while a string does not. LabVIEW can't guess ahead of time what strings will be sent to a case structure, so it can't assign a case for every value. This means that there is a real danger of unknown strings being sent to the case structure (usually as a result of the LabVIEW developer misspelling one of the states).

To address this problem, a string-based state machine must always have a default state designated to trap unknown states. One way to do this is to create a default case

for the empty string. The default case should contain code to handle the unknown string, perhaps by generating a custom error.

Custom errors are useful tools, allowing the developer to create warnings or errors for situations specific to a particular VI. LabVIEW reserves the ranges of error codes −8999 through −8000, 5000 through 9999, and 500,000 through 599,999 for custom use. Custom errors can be generated using the **`Error Cluster From Error Code`** function. The help file is shown in Figure 20.19. This function accepts an error code and an error message string as inputs and creates the custom error cluster. The **`Error Cluster From Error Code`** function is useful because the error message strings can be programatically determined at run-time.

An example of this technique can be found in Figure 20.20. The default case shown here creates a custom error message when an unknown state is requested. The **`Concatenate Strings`** function is used to incorporate the unknown state string into the error message. When this custom error is generated, the state machine will respond by shutting down gracefully and notifying the user that an unknown state has been requested.

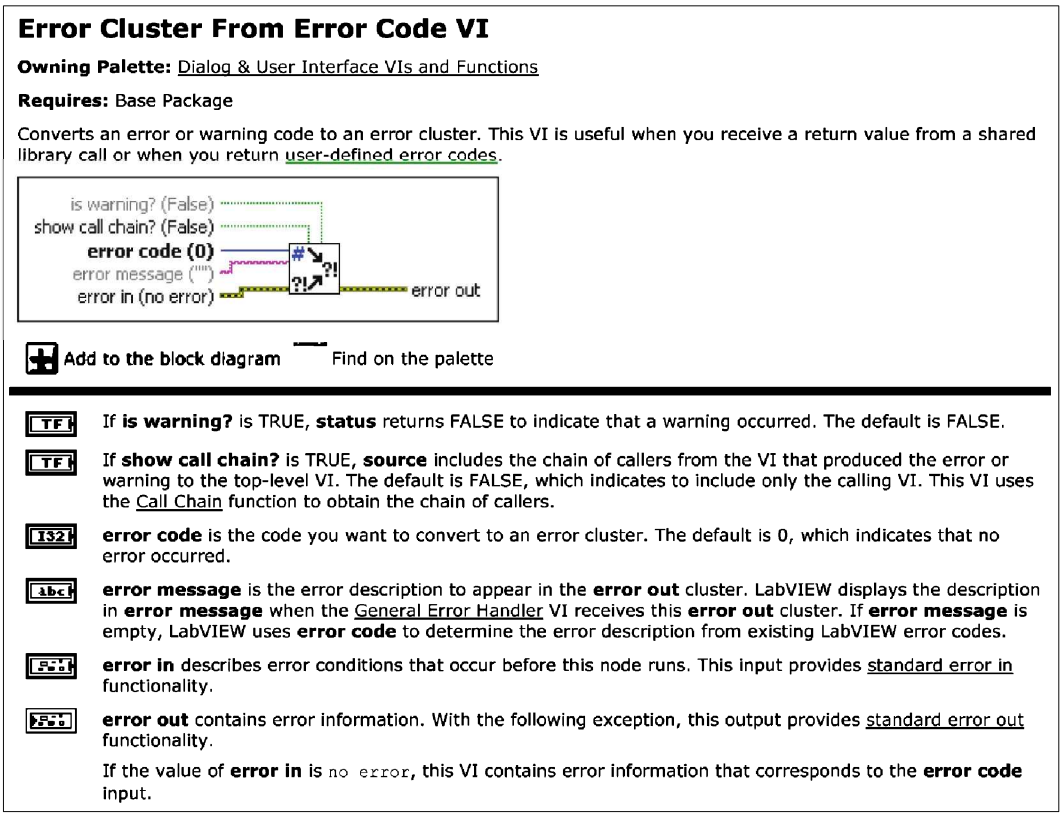


Figure 20.19: Help file for the `Error Cluster From Error Code` function

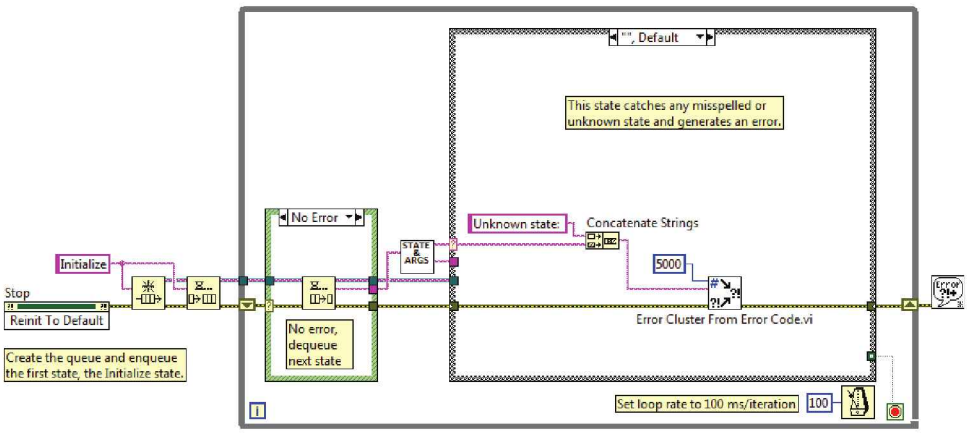


Figure 20.20: Default case of the queued string `Multitest` VI

20.8 Improving the String-Based State Machine

The `Multitest` VI that we just developed is functional, but not easily scalable. Queueing individual strings is awkward and consumes too much space on the block diagram. Look back at Figure 20.16. Imagine queueing all of the states needed to run the Deluxe Wash with each string wired to a separate **`Enqueue Element`** function. It would stretch the block diagram beyond the width of the screen and would be difficult to read.

In Chapter 14, we used an array of states and a **`For`** loop to enqueue a group of states (see Figure 14.8). This saved space and improved readability. We can do the same here for the string-based `Multitest` VI, as demonstrated in Figure 20.21.

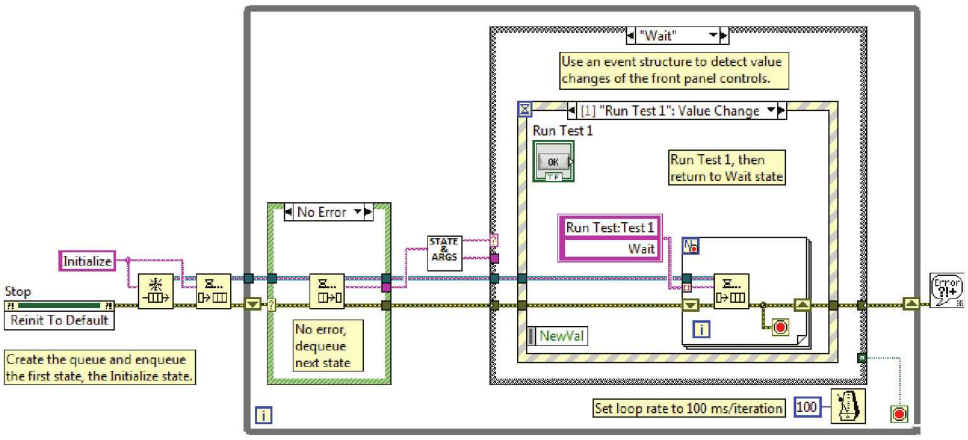


Figure 20.21: Implementing an array and `For` loop to queue states in the `Multitest` VI

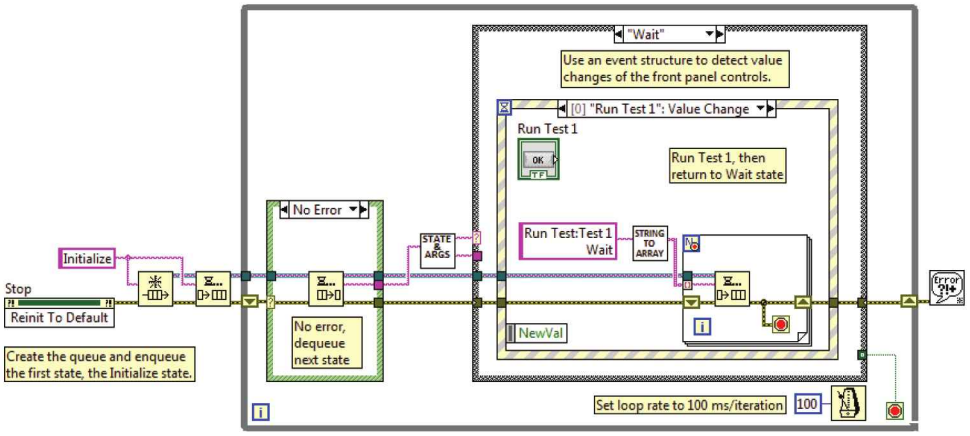


Figure 20.22: Replacing the array of string constants with a single string using the `String to Array` VI



Figure 20.23: Icon for the `String to Array` sub-VI

Now that we are working with strings, we can improve upon this technique. Instead of using an array of strings in the event structure, it is much easier to use a single multiline string that could be converted into an array. This idea is illustrated in Figure 20.22. In this version of the `Multitest` VI, we adopted a second parsing convention. Multiple state and argument strings are now represented by a single string with each state and argument string separated by a linefeed. Notice how this improves the readability of the code. It also improves maintainability, because it is much easier to edit a single multiline string than it is to edit an array of strings.

The conversion of a multiline string to a string array can be accomplished with the `String to Array` sub-VI with the icon shown in Figure 20.23. The sub-VI takes a multiline string as an input, splits it into its individual lines, and returns an array of strings made up of these individual lines. This process is illustrated by the inputs and outputs of the front panel shown in Figure 20.24.

The `String to Array` sub-VI makes use of the native **`Spreadsheet String To Array`** function. This help file is found in Figure 20.25. This function takes a string that represents tabulated data as an input. The data type of the input data is specified by a format input, and the delimiter is also specified as an input. The output of this function is an array of the tabulated data values in their correct order and in the specified data format. The `String to Array` sub-VI uses this function to split a multiline string into an array of individual strings by specifying the linefeed character as a delimiter, as shown in Figure 20.26.

We can take a further step to improve the readability of the string-based state machine with arguments. Compare Figures 14.8 and 20.22. The **`For`** loops in these

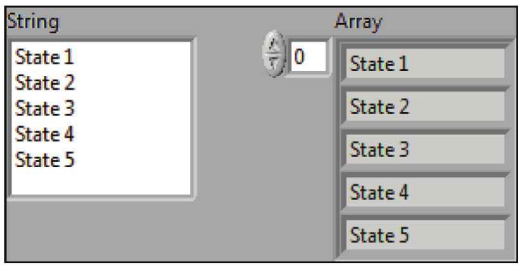


Figure 20.24: Front panel of the `String to Array` sub-VI

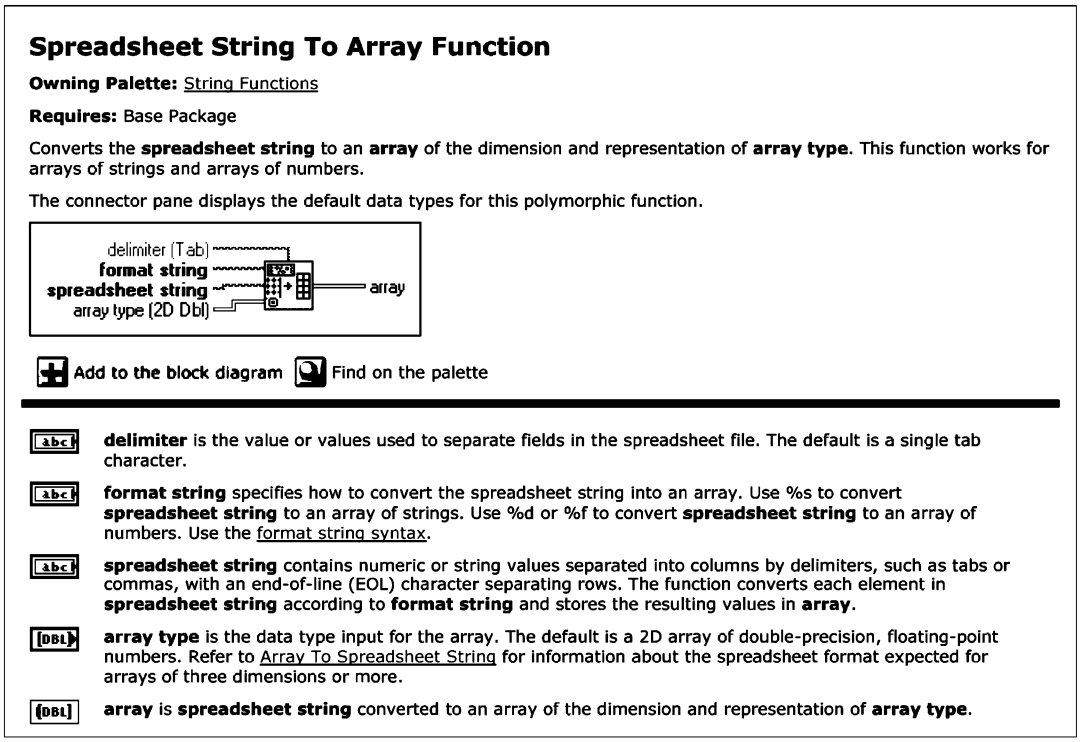


Figure 20.25: Help file for the `Spreadsheet String To Array` function

two figures look nearly identical. The only difference between them is in the data type of the queue reference input and array input tunnels. One of the benefits of string-based queues is that all string-based queues and string-based arrays have the same data type—no matter what parsing convention is used. This is not true for clusters. Clusters with different elements have different data types.

This has a big impact on code reusability. If we develop a useful sub-VI that operates on a string-based queue, we can use it on *any* string-based queue. For example, Figure 20.22 can be simplified by replacing the `String To Array` sub-VI and the

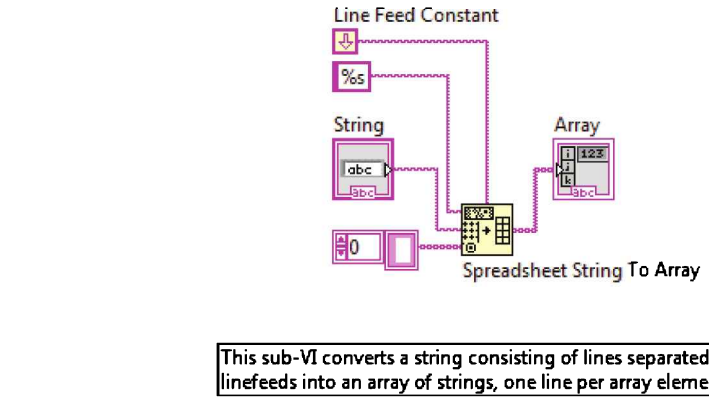


Figure 20.26: Block diagram of the `String to Array` sub-VI

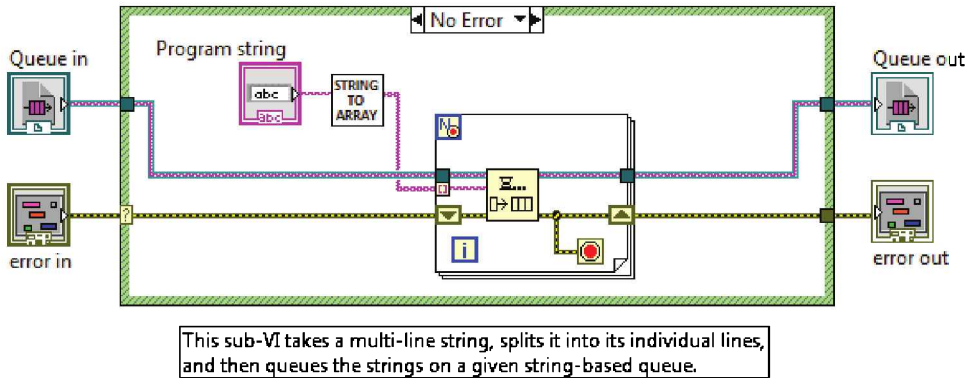


Figure 20.27: Block diagram for the `Split And Queue String` sub-VI



Figure 20.28: Icon for the `Split And Queue String` sub-VI

For loop with a reusable `Split And Queue String` sub-VI. The block diagram for this sub-VI is shown in Figure 20.27.

The icon of the `Split and Queue String` sub-VI is shown in Figure 20.28, and its front panel is shown in Figure 20.29. Notice that an output terminal for the queue reference has been included in this sub-VI. This terminal is not needed by the `Multitest` VI, but it is a good programming practice to include reference output terminals for sub-VIs with reference input terminals. This increases the reusability of the sub-VI and allows the sub-VI to be chained with other VIs that use the queue reference if the need arises.

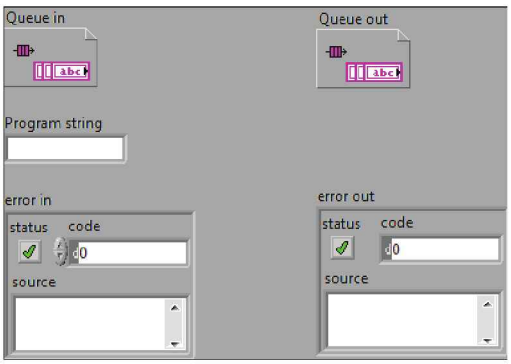


Figure 20.29: Front panel for the `Split And Queue String` sub-VI

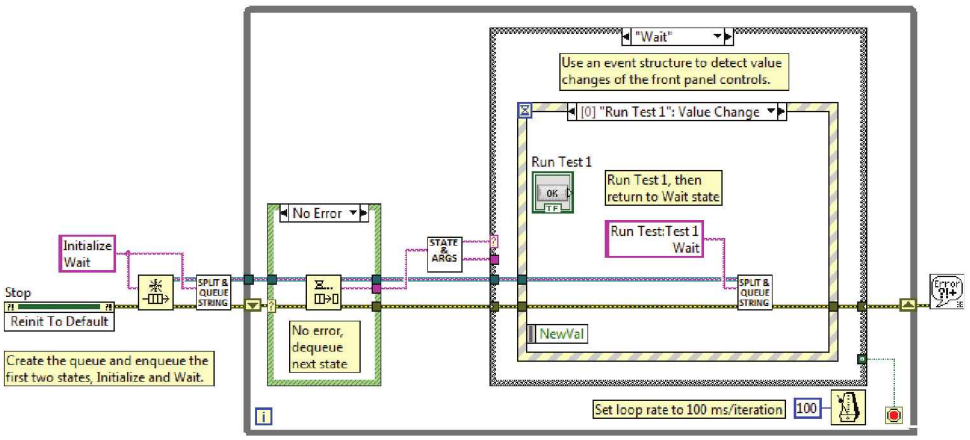


Figure 20.30: Block diagram of the `Multitest` VI using the `Split and Queue String` sub-VI

Wiring Tip: The queue reference control and indicator seen in Figure 20.29 are not found in the the standard LabVIEW palette of controls. To create the queue reference control, right-click on the queue reference input of one of the the native LabVIEW queue functions and select *Create. . . Control*. This will create a string-based queue reference control on the front panel. You can then copy the control and change it into an indicator by right-clicking and selecting *Change to Indicator*. Alternatively, you can right-click on the queue reference output of a queue function on the block diagram and select *Create. . . Indicator*.

The `Split and Queue String` sub-VI can now be used to simplify the `Multitest` VI. The event case handling the value change of the RUN TEST 1 button is shown in Figure 20.30. The multiline string within the event case is much easier to

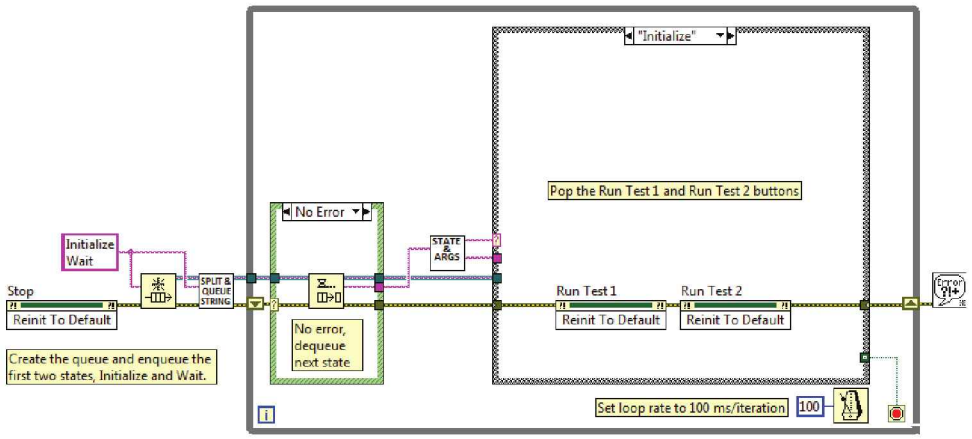
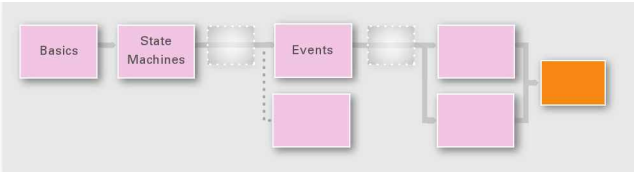


Figure 20.31: *Initialize* state of the `Multitest` VI using the `Split and Queue String` sub-VI

read and edit than an array of string constants. The `Split and Queue String` sub-VI suggests that the contents of the string constant are placed on the queue, hiding the low-level details of how the string is split and enqueued and how the errors are trapped. This increases the readability of the top-level VI. The `Split and Queue String` sub-VI can be opened and examined if the reader is interested in the details of how this is accomplished.

Notice too that the `Split and Queue String` sub-VI has been used in the input area of the loop. In Figure 20.13, the *Initialize* state is enqueued outside the main loop, and the *Wait* state is enqueued in the *Initialize* state. This has been done to save space in the input area of the loop, allowing the loop to be made larger and still fit on one screen. In Figure 20.30, the `Split and Queue String` sub-VI is used outside of the main loop instead of the native **`Enqueue Element`** function, allowing both the *Initialize* and the *Wait* state to be enqueued without taking up extra space. This increases the readability of the `Multitest` VI. The first two states of the `Multitest` VI are always *Initialize* and *Wait*. Placing these two states in the input area of the loop makes this fact obvious to the reader.

The new version of the *Initialize* state is shown in Figure 20.31. Compare this figure with Figure 20.13. The *Initialize* state no longer needs to queue the *Wait* state, since it is now queued in the input area of the loop.



11

The Event-Driven Car Wash Controller

We can now upgrade the classic state machine implementation of the car wash controller to an event-driven car wash controller. This upgrade only affects the *Wait* state—all other states of the classic state machine car wash controller presented in Chapter 8 remain the same.

A drawback of the classic state machine is the awkwardness and inefficiency of the *Wait* state due to polling front panel controls. These problems can be solved by replacing the code in the *Wait* state of the classic state machine car wash controller with an event structure. The event structure provides two major benefits. First, it is easily extensible. New front panel controls can be accommodated by adding new cases to the event structure. Second, it is efficient. The event structure "goes to sleep" when there are no events to process.

11.1 Adding the Event Structure to the *Wait* State

Figure 11.1 shows how an event structure can be added to the *Wait* state of the car wash controller. The event case shown detects a value change of the purchase selector. The purchase selector is a cluster of two Boolean controls: the DELUXE WASH button and the ECONOMY WASH button. If either one of these buttons is pressed, the value of the purchase selector cluster changes. The value of the DELUXE WASH control is then placed on the data highway, and the *Lock Selector* state is placed on the state shift register of the main loop.

The configuration window for this event case is shown in Figure 11.2. Notice that the value change of the <*All Elements*> entry of the *Purchase Selection* cluster has been selected. This means that the event case will execute if any element of the purchase selector changes value.

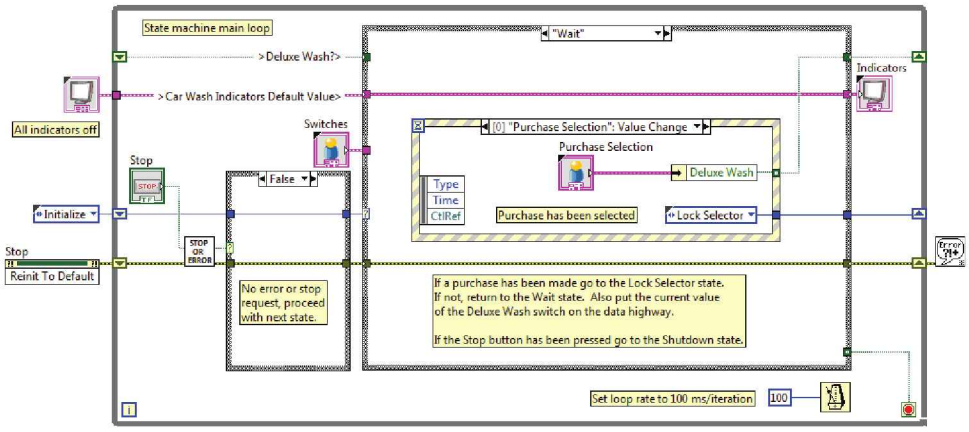


Figure 11.1: The event case handling the value change of the PURCHASE SELECTION control

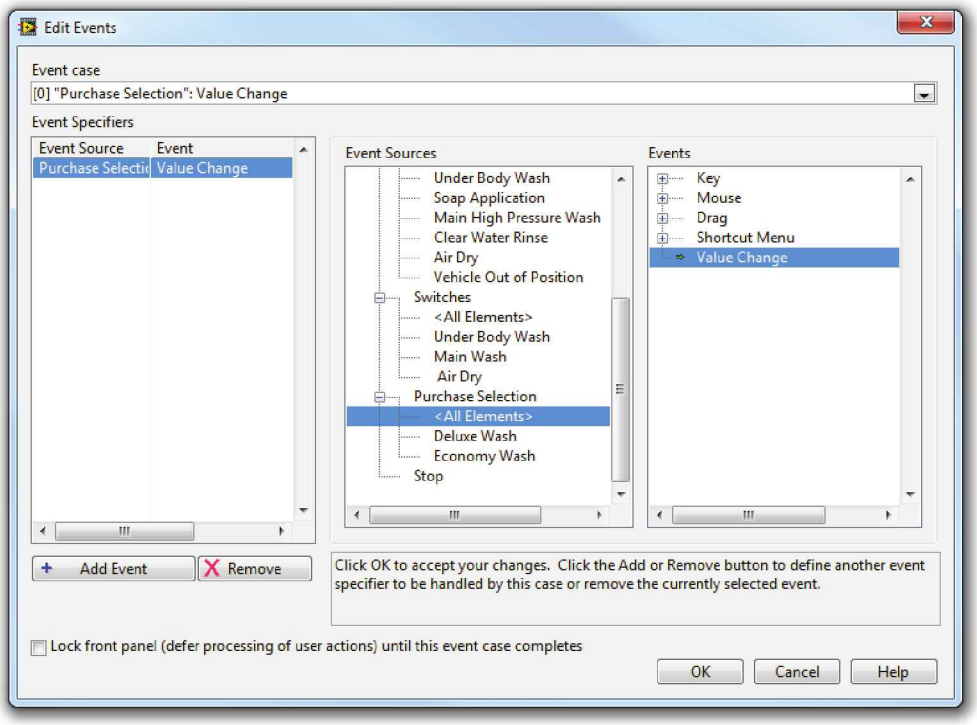


Figure 11.2: The configuration window for the event case handling the value change of the PURCHASE SELECTION control

11.2 The STOP Button

The STOP button needs special consideration in an event-driven state machine. In the classic state machine implementation of the car wash controller developed in Chapter 8, the STOP button is checked every iteration by the guard clause. In fact, checking the STOP button is the first thing that is done each iteration of the state machine.

The classic state machine runs continuously, polling all front panel controls every 100 milliseconds. This is inefficient, but it does guarantee that the state machine will respond to a stop request within the 100 millisecond limit mandated by the functional specification.

The event-driven state machine is more efficient, putting the main loop to sleep in the *Wait* state when no front panel activity has been detected. But the STOP button is no longer being polled when the event structure is sleeping. In order to respond to a stop request while in the *Wait* state, we need to configure an event case to monitor the value change of the STOP button, as shown in Figure 11.3. The corresponding event configuration window is shown in Figure 11.4.

The STOP value change event case places the *Shutdown* state on the state shift register if the STOP button is pressed. This ensures that the state machine will wake up and shut down if the STOP button is pressed. It is usually recommended that a control should be placed within the event case that handles its value change, but in this case this is not desirable, since the STOP button needs to be polled in the lobby of the state machine. We need to monitor the STOP button in two different places. We need a case in the event structure to respond to the STOP button while the state machine is waiting, and we need to poll the STOP button every loop iteration when the state machine is not waiting. This duplication of effort is due to the fact that in a

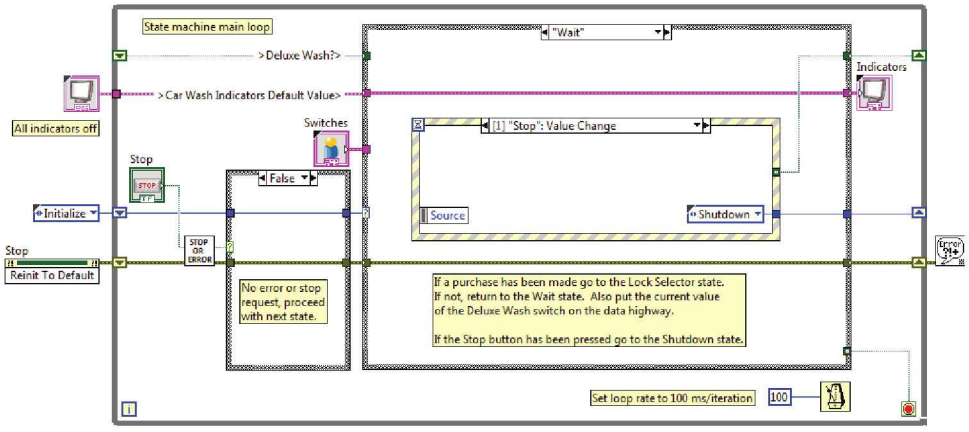


Figure 11.3: The event case handling the value change of the STOP button

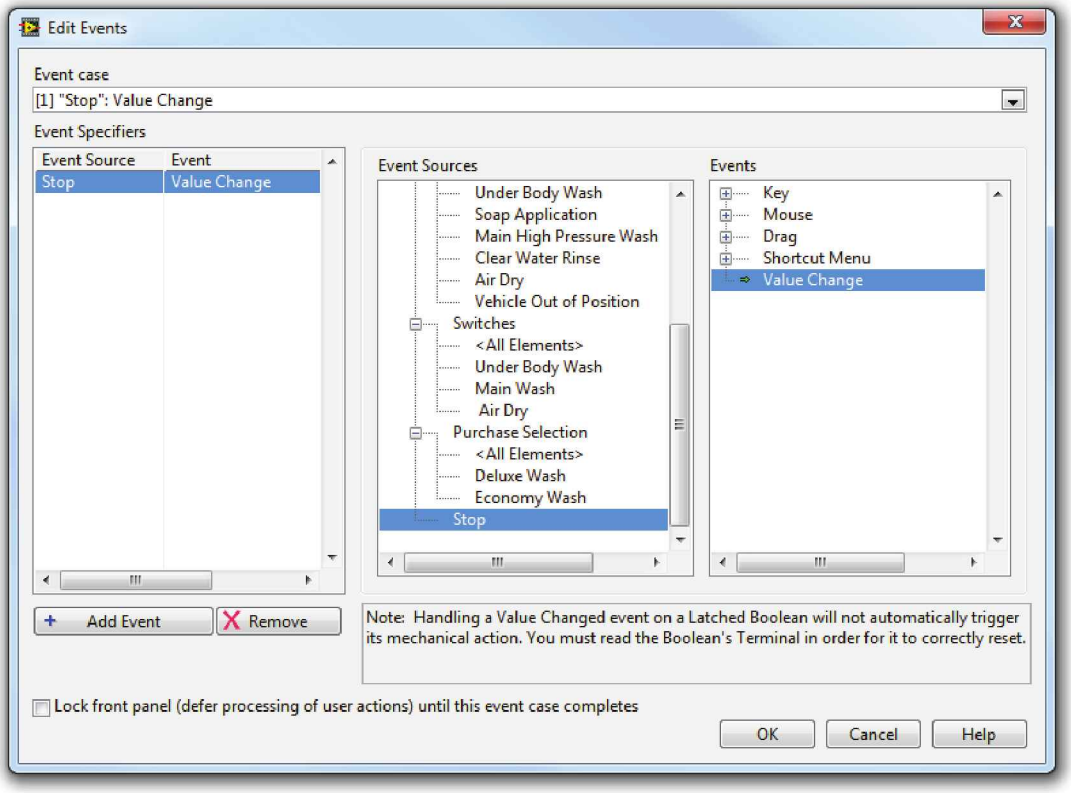
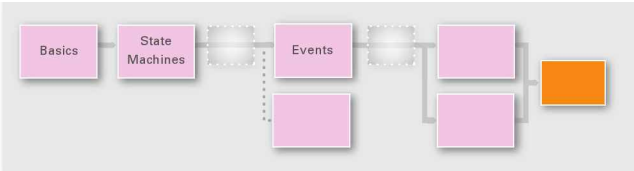


Figure 11.4: The configuration window for the event case handling the value change of the STOP button

single-loop state machine the event structure is only active when the program is in the *Wait* state.



12

Problem Set 4

12.1 Traffic Light Controller Exercises

Exercise 4.1: Event-Driven Traffic Light Controller

Is the *Traffic Light* program a good candidate for an event structure? Why or why not? If it is a good candidate for an event-driven architecture, upgrade the classic state machine implementation developed in Exercise 3.1. ■

12.2 Security System Controller Exercises

Exercise 4.2: Event-Driven Security System Controller

The *Security System* program is an excellent candidate for an event structure. Do you see why? Upgrade the classic state machine implementation developed in Exercise 3.8 by adding an event structure to the *Wait* state. Consider the location of the STOP button. Does it still need to be in the lobby of the state machine? What are the consequences and benefits of moving it into the event structure? ■

12.3 Boiler Controller Exercises

Exercise 4.3: Event-Driven Boiler Controller

Is the *Boiler Controller* program a good candidate for an event structure? Why or why not? If it is a good candidate for an event-driven architecture, upgrade the classic state machine implementation developed in Exercise 3.11. ■

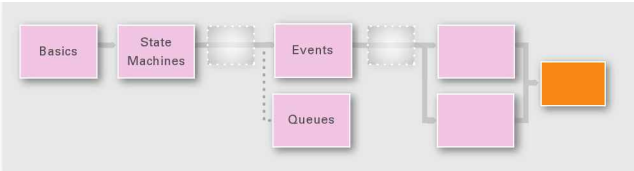
12.4 Car Wash 2 Exercises

Exercise 4.4: Event-Driven Car Wash 2 Controller

Like the original car wash controller, the Car Wash 2 controller is a good candidate for an event structure. Add an event structure to the *Wait* state to avoid polling the START button, but do not change anything else in the code. The Car Wash 2 controller also waits during the *Car Enter* and *Car Exit* states. Is it a good idea to add event structures to these states also? Why or why not? ■

Exercise 4.5: Revising the Event-Driven Car Wash 2 Controller

In Exercise 4.4, you were asked to add an event structure to the *Wait* state of the Car Wash 2 controller to avoid polling the START button. The implementation in the solution of this exercise left the ENTRY CONSOLE control in the event structure and placed its value on the data highway. But keeping the control in the event structure is not necessary, because the event structure can monitor the START button without the control being in the structure itself. Revise the event-driven Car Wash 2 controller developed in Exercise 4.4. Remove the ENTRY CONTROL from the event structure and from the data highway. Place the control in the state where it is needed instead. Compare the two event-driven Car Wash 2 controllers. What are the advantages and disadvantages of each implementation? ■



13

Queued State Machines

The event-driven state machine is an improvement over the classic state machine. The event structure eliminated polling the user interface, and it made the car wash controller easier to read, scale, and maintain. But the event structure did nothing to change the architecture of the classic state machine.

The hallmark of the classic state machine is that each state must determine the transition to the next state. The classic state machine can only think one move ahead. This is a serious limitation. Some states in a state machine are bound to be branch points, which are states that have more than one possible transition. In a classic state machine, these branch point states need to have logic built in to them to determine which state to transition to next.

The car wash controller developed in Chapter 8 is a good example. The *Lock Selector* and the *Rinse* states are branch points. This can be seen graphically in the state diagram shown in Figure 2.5. The transitions from these two states depend on whether the Deluxe Wash or the Economy Wash is chosen. This means that a Boolean needs to be added to the data highway to store the type of wash selected. These states also need logic to interpret this Boolean and select the next state.

This technique solves the problem, but at a cost. The *Lock Selector* and the *Rinse* states are coupled to the wash type. These states need to know which wash type is chosen before they can perform their function. This degrades the maintainability and scalability of the code.

But how can this be avoided? How can the car wash states be decoupled from the type of car wash? What we need is a way to store a list of states. We need a way to add states to the list as a group when a wash type is chosen and we need a way to take them off the list one by one in the correct order as the state machine runs through the chosen wash. The list of states will drive the state machine transitions. Individual states need not decide which state to transition to next and thus not need to know which type of wash had been selected. The state transition diagram for this sort of state machine looks like the one shown in Figure 2.6.

13.1 The Power of Queues

LabVIEW provides a group of functions that can be used to manage ordered groups of states called the queue functions. A queue is an ordered buffer used to store information for later use—first in, first out (FIFO)—like a line of people in a store. People enter the line at the end and exit the line at the front. Similarly, elements are added to the end of a queue and are removed from the front.

Another way to think of a queue is as a special type of array. Like arrays, every element in a queue has the same data type and is ordered in a fixed sequence. Unlike an array, though, queue elements only can be added at the ends the queue. Elements only can be taken off the queue from the front. Arrays are random access, where any element can be accessed by specifying an index. Queues are sequentially accessed, and elements must be read in the order that they are queued.

13.2 Queue Functions

The queue functions are found in the in the *Programming...Synchronization...Queue Operations* sub-palette. The subset of functions used in this book are described in Table 13.1.

A queue is created and configured with the **`Obtain Queue`** function (see the help file in Figure 13.1). The data type of the elements on the queue is determined by

Table 13.1: Useful queue functions

Obtain Queue	This function creates a reference to a new queue or obtains a reference to an existing named queue. This function also determines the data type of the queue. Optionally, a name can be given to the queue and a maximum size may be set. For more details see Figure 13.1.
Enqueue Element	This function places an element at the back of a queue. Queues are FIFO (first in, first out), so this is the function typically used to place an element on the queue. For more details see Figure 13.2.
Enqueue Element At Opposite End	This function places an item at the front of the queue. This makes the element first in line to be dequeued. For more details see Figure 13.3.
Dequeue Element	This function removes the element from the front of the queue and returns it. This function waits forever for an element to be available unless a timeout value has been wired to the timeout terminal. Attempting to dequeue an element from a non-existent queue produces an error. For more details see Figure 13.4.
Release Queue	This function releases the queue reference, destroying the queue. For more details see Figure 13.5.

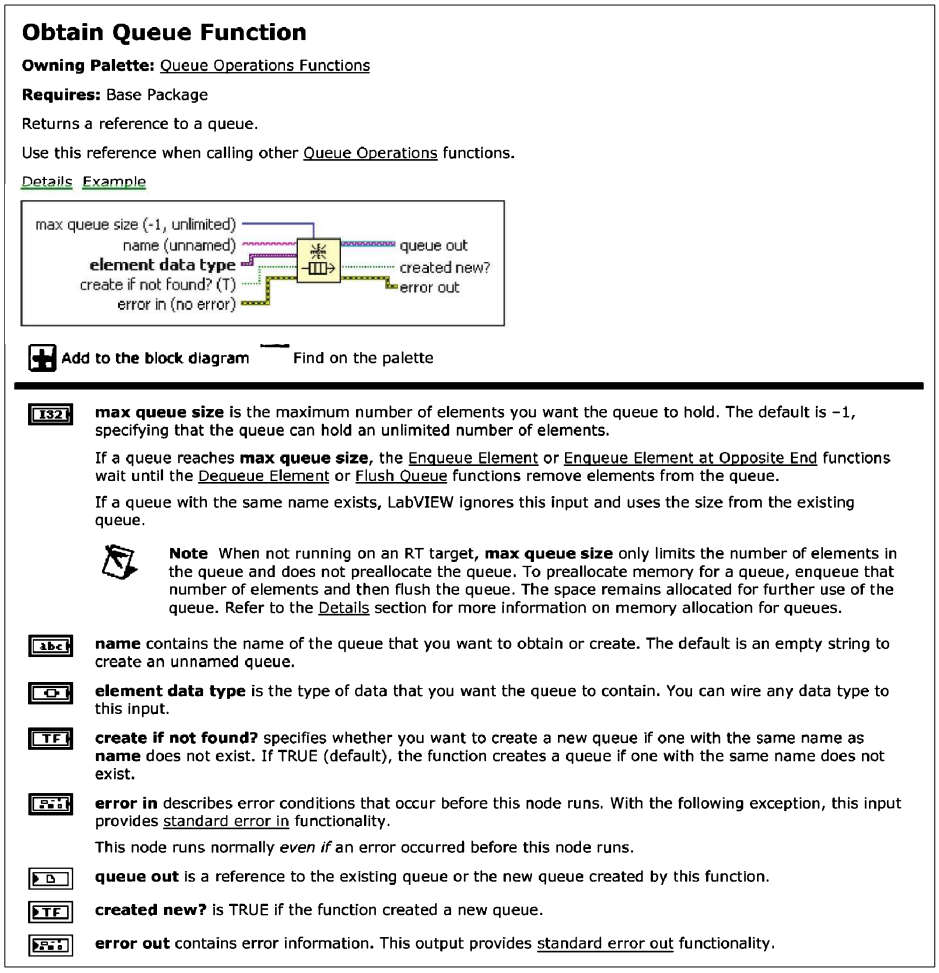


Figure 13.1: Help file for the `Obtain Queue` function

the data type of the constant or control wired to the element type input. The value that is wired to the element data type input is *not* placed on the queue. It is only used to determine what kind of element will be permitted to be placed on the queue.

The size of the queue is unlimited unless a specific queue size is wired to the max queue size input. If a maximum size is specified, the queue will never exceed this size. If an attempt is made to add an element to a queue that is already full, the function attempting to add the element will wait until space is made available by elements exiting the queue.

A queue can be given a name when it is created. Queues are generally accessed through the queue reference wire. The reference to the queue travels along the wire like any other piece of data. But if a queue has a name, a reference to the queue can be generated anywhere in the program by wiring the name to another **`Obtain`**

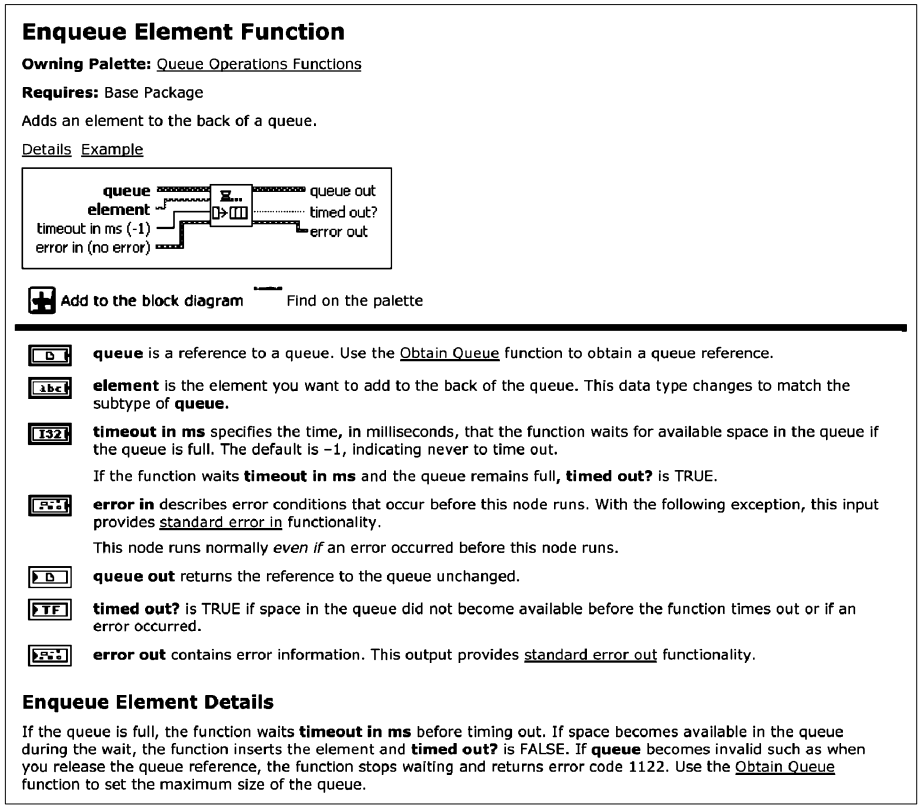


Figure 13.2: Help file for the `Enqueue Element` function

`Queue` function. If a queue with that name already exists, the reference generated by the function will be to the existing queue. Different VIs running under the same application of LabVIEW can access the same queue if the queue is named. This is a powerful feature of queues, but it is beyond the scope of this book. The queues used in the examples in this book are not named.

Once a queue has been created, it is populated with the **`Enqueue Element`** function (see the help file in Figure 13.2). The **`Enqueue Element`** function places an element on the back of the queue. The **`Enqueue Element at Opposite End`** function (see the help file in Figure 13.3) places an element at the front of the queue. Elements are dequeued from the front of the queue, so enqueuing an element at the front puts it next in line for dequeuing.

The **`Dequeue Element`** function removes the element at the front of the queue and returns it as an output (see the help file in Figure 13.4). If the queue is empty, this function will wait forever until an element becomes available unless a timeout is specified. If this function tries to dequeue an element from a non-existent queue, an error is generated.

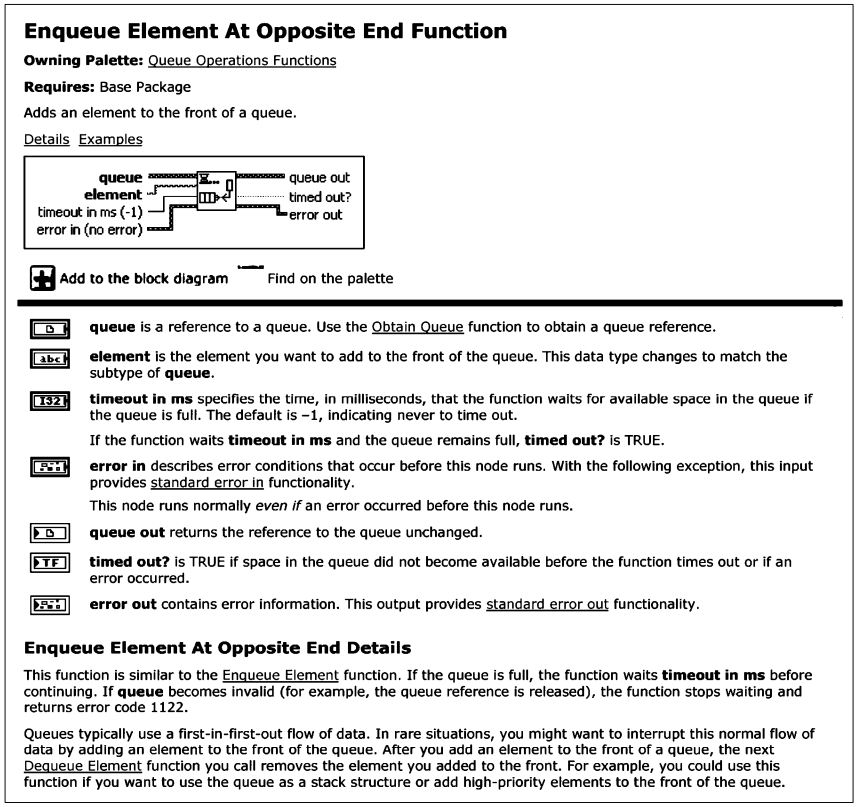


Figure 13.3: Help file for the `Enqueue Element At Opposite End` function

The **`Release Queue`** function destroys the queue by releasing the queue reference (see the help file in Figure 13.5). It is good programming practice to "free-up" system resources by releasing references when they are no longer needed. The **`Release Queue`** function operates normally—even if an error is present at the *error in* terminal. This means that if the **`Release Queue`** function is placed in the *Shutdown* state of a state machine it can safely be placed on the main error wire. The queue reference will be released even if an error is present on the main error wire.

13.3 By-Value versus By-Reference

LabVIEW is a by-value dataflow language. This means that data values travel along the wires connecting functions. Splitting a by-value wire creates copies of the data. These copies are independent of each other, and each can be modified without affecting each other.

Queues are passed by-reference rather than by-value. The actual queue does not travel on a wire, only a reference to the queue. If a queue wire is split, both wires

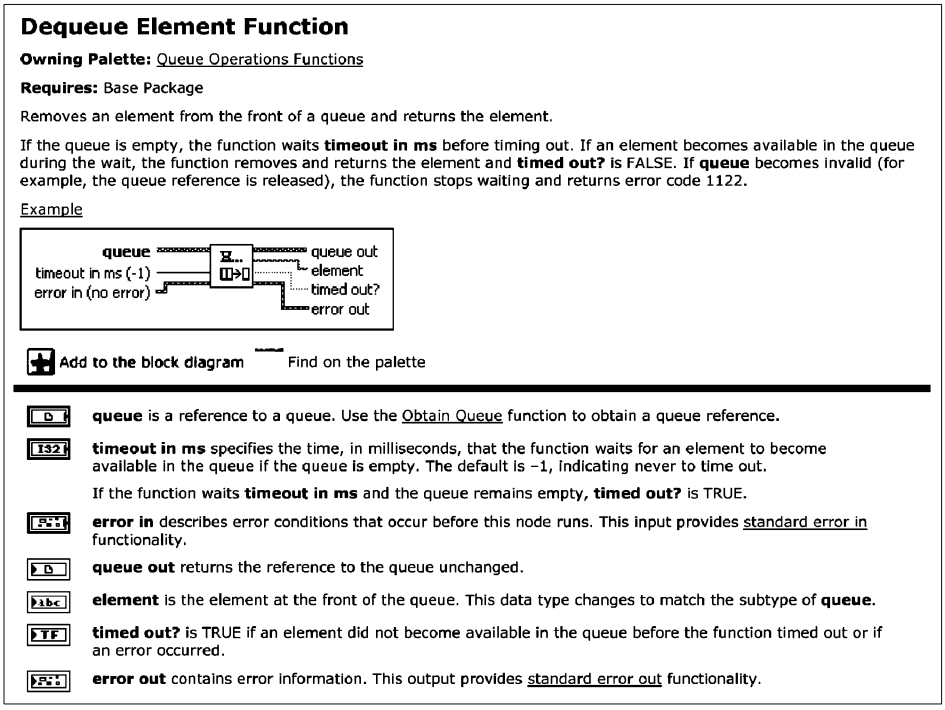


Figure 13.4: Help file for the `Dequeue Element` function

carry references to the same queue. Changes made to the queue on one wire are seen by the other wire. Because of this, queues need to be used carefully. By-reference wires have power and convenience, but they can be more vulnerable to problems than by-value wires.

Here's an example. Imagine that you are making a purchase using cash. When you pay for a purchase with cash, you are making a by-value transaction. Your wealth is decreased at the time you make the purchase. If you don't have enough money to make the purchase, you can't complete the transaction. This is inconvenient, but cash transactions prevent you from spending money that you do not have.

Now imagine that you make the purchase using a check. Now you are making a by-reference transaction. Instead of transferring money to the cashier directly, you are providing a reference to your bank account with a promise to pay a specified amount. This is convenient, but it has the potential for problems. You can make a promise with a check that your account cannot keep, resulting in bounced checks and overdrafts.

Queues are relatively well-behaved in single-loop state machines, but they can cause trouble if operations on the same queue are performed in parallel with each other. There are two common scenarios to watch out for. The first scenario occurs when there are multiple enqueue functions putting elements on the same queue at the same time. This can lead to the situation shown in Figure 13.6. Three enqueuing

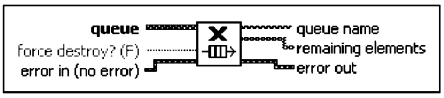
Release Queue Function

Owning Palette: Queue Operations Functions

Requires: Base Package

Releases a reference to a queue.

Details Example



Add to the block diagram Find on the palette

queue is a reference to a queue. Use the Obtain Queue function to obtain a queue reference.

force destroy? indicates whether to destroy the queue. If FALSE (default) and you want to destroy the queue, either call the Release Queue function a number of times equal to the number of times you obtained a reference to the queue or stop all VIs using the queue reference. If TRUE, the function destroys the queue and you do not have to call the function multiple times or stop all VIs using the queue reference. Destroying the queue invalidates all references to the queue.

error in describes error conditions that occur before this node runs. With the following exception, this input provides standard error in functionality.

This node runs normally *even if* an error occurred before this node runs.

queue name is the name of the queue.

remaining elements is the array of elements that were in the queue before the function released the queue. The first element in the array is the element from the front of the queue and the last element in the array is the element from the back of the queue.

This data type changes to match the subtype of **queue**.

error out contains error information. This output provides standard error out functionality.

Release Queue Details

You can use the Obtain Queue function to obtain a reference to the same queue with the same name multiple times. To destroy a queue, call the Release Queue function a number of times equal to the number of times you obtained a reference to the queue or stop all VIs using the queue reference.

If **force destroy?** is TRUE, this function releases all references to the queue and destroys the queue.

Any Enqueue Element, Enqueue Element at Opposite End, Preview Queue Element, and Dequeue Element functions waiting on the queue time out and return error code 1122.

Figure 13.5: Help file for the `Release Queue` function

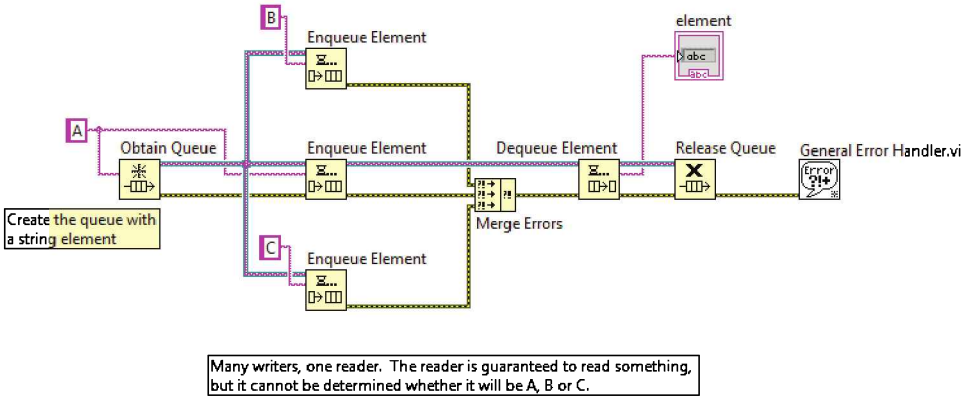


Figure 13.6: Which element will the reader dequeue?

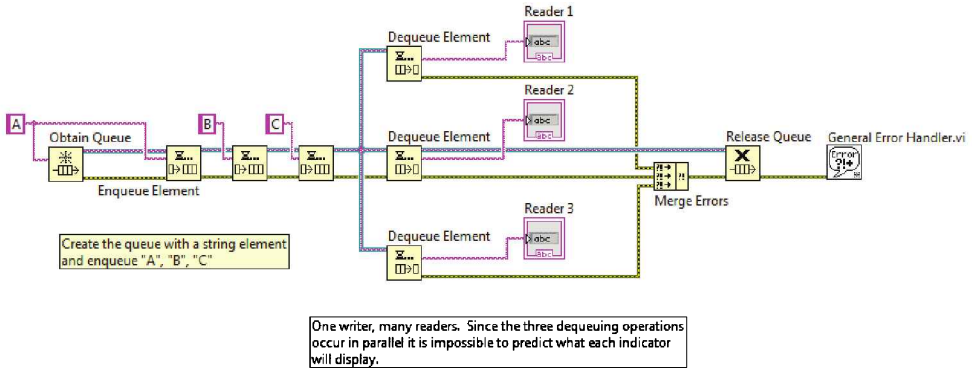


Figure 13.7: What will be displayed by each indicator?

operations occur in parallel upstream from a dequeuing operation. Dataflow does not determine the order in which the three enqueuing operations accessed the queue, so it is impossible to predict which element will be dequeued when the code executes.

Figure 13.7 shows the opposite problem. In this situation, the strings `A`, `B` and `C` are enqueued upstream of three parallel dequeuing operations. Since the three dequeuing operations have no data dependence it is impossible to predict what values will be displayed by the three indicators.

Both of these examples suffer from race conditions. A race condition is a programming situation in which the outcome of a program is affected by the order in which parallel operations execute. Race conditions will be discussed more thoroughly in Chapter 28. To reduce the chance of race conditions, pay careful attention to the sections of code that write to and read from queues. Try to keep the functions that enqueue elements confined to one section of the code, typically the *Wait* state in a state machine. Try to also confine the dequeuing operations to one section of the code, typically the guard clause of the state machine. In this way, you will avoid the problems associated with many readers and writers.

13.4 Building a Queued Event-Driven State Machine

Let's demonstrate the use of queues by upgrading the `Multitest` VI that was first developed as a classic state machine in Chapter 4 and then upgraded with an event structure in Chapter 10. In both of these implementations, each state needed to determine the transition to the next state.

By adding a queue to this state machine, we can streamline it significantly. We can remove the code in each state that specifies the transition to the next state. Instead, a sequence of states is placed on the queue in the *Wait* state whenever a

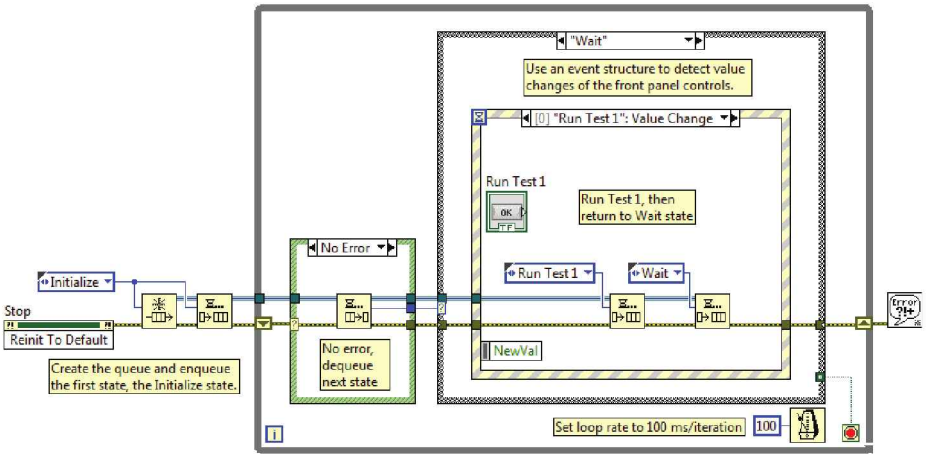


Figure 13.8: This figure shows the event case handling the value change of the RUN TEST 1 button; notice that a series of states are queued, showcasing the advantage of using a queue

button is pressed on the front panel. This is illustrated in Figure 13.8, which shows the event case that handles the value change of the RUN TEST 1 button. The *Run Test 1* state is enqueued first and is followed by the *Wait* state. These states will be dequeued in the same order. First the *Run Test 1* state will execute and then the *Wait* state.

This means that the *Run Test 1* state no longer needs to dictate the transition back to the *Wait* state as it did in the classic state machine. It may seem like a small change, but it is very significant. In the queued state machine, the individual states have no knowledge of either the states that came before them or the states that will come after them. The states are now uncoupled from each other—like beads on a string that can be arranged and rearranged in any order that the developer desires. This decrease in coupling is a huge step forward in maintainability and scalability. States can be added, deleted, or modified with less chance of disturbing the code in other states.

Let's step through the queued implementation of the `Multitest` VI starting with the *Initialize* state shown in Figure 13.9. The queue is created in the input area of the state machine. Queue creation is an example of start-up code, which is usually placed in the input area of the state machine. The type-defined enumerated constant that contains the list of states is used to define the data type of the queue element in the **`Obtain Queue`** function. This allows the queue to store state transitions. The list of states in the enumerated constant is the same used in Chapters 4 and 10.

The *Initialize* state is then placed on the queue using the **`Enqueue Element`** function. This ensures that the first state executed will be the *Initialize* state. The *Initialize* constant is also used to set the data type of the queue in the **`Obtain Queue`** function and as the first element of the queue. Remember, the constant used to create

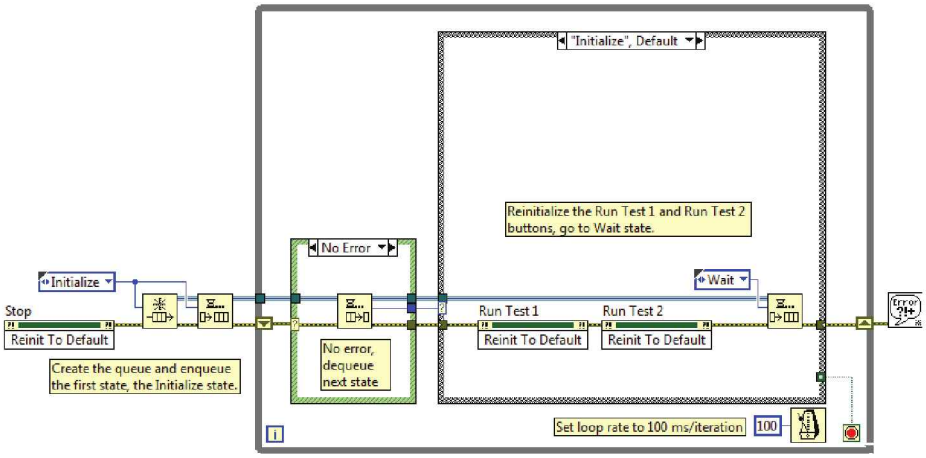


Figure 13.9: The *Initialize* state of the `Multitest` VI

the queue is not placed on the queue. It is only used to determine the data type of the queue.

The queue reference wire and the error wire then enter the main loop. The error output of the **`Enqueue Element`** function is used to initialize the error shift register of the main loop. Now take a closer look at the point where the queue reference wire enters the main loop. Notice that the wire enters the loop through a simple tunnel instead of a shift register. Why doesn't the queue reference wire require a shift register?

Remember that the actual queue does not travel along the wire. If the actual queue did travel on the wire in a by-value fashion, the queue would need a shift register on the main loop. Every time a state was added or removed from the queue, the queue would change, and the changed queue would need to be passed from one loop iteration to the next via a shift register. But queues do not travel along wires in a by-value fashion. They travel by-reference instead. The actual queue is not on the wire, only a reference to the queue. The reference to the queue remains the same, no matter how the queue changes as states are added or removed. The queue reference is a constant, and constants do not need to be placed on shift registers. Constants should be wired into loops through standard input tunnels.

When the queue has been created and initialized, the main loop begins to execute. First the guard clause checks to see if an error is present. If the error line is clear, the next state of the state machine is dequeued, as seen in Figure 13.9. At the beginning of the program, the only state on the queue is *Initialize*, so the state machine will transition to the *Initialize* state where the RUN TEST 1 and RUN TEST 2 buttons are initialized. The *Initialize* state then enqueues the *Wait* state. This *Initialize* state dictates the transition to the next state—just as it did in the classic state machine. Another way to do this would be to place two **`Enqueue Element`** functions outside the main loop of the state machine. Both the *Initialize* and the *Wait* states could be enqueued before the main loop starts. Both methods are functionally equivalent.

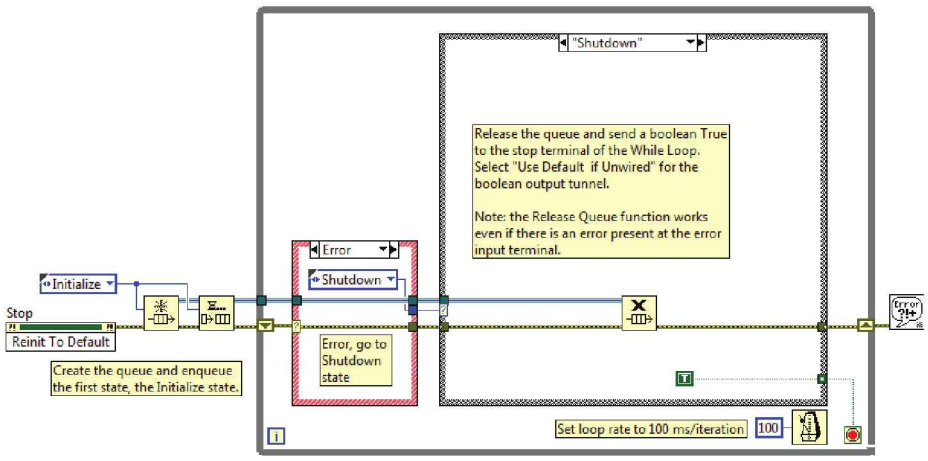


Figure 13.10: The *Shutdown* state of the `Multitest` VI

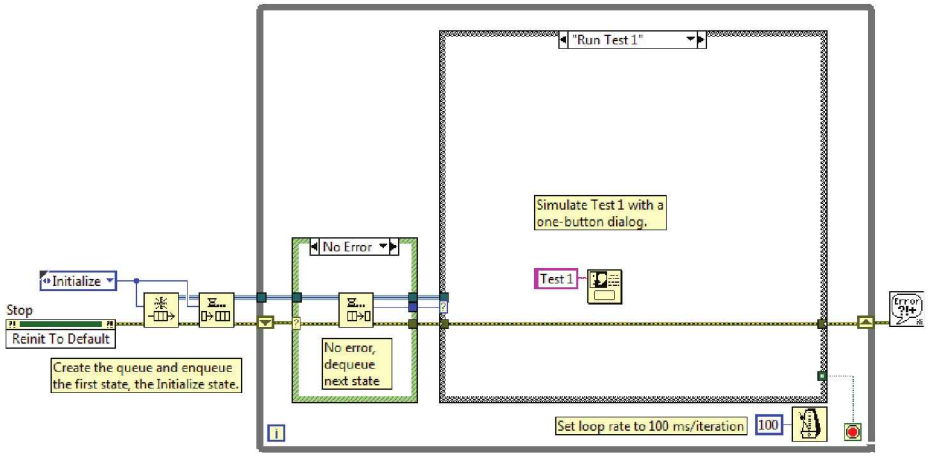


Figure 13.11: This figure shows the *Run Test 1* state; notice that there is no code in this state for determining what state to go to next—this is another benefit of using queues

If the guard clause detects an error, it ignores the elements on the queue and forces a transition to the *Shutdown* state, as shown in Figure 13.10. The *Shutdown* state sends a Boolean True to the loop stop terminal, stopping the loop. The *Shutdown* state also releases the queue. The *Release Queue* function can be placed safely on the main error wire, since it will execute even if an error is present at its error-input terminal.

The heart of the queued state machine is in the *Wait* state. The event case handling the value change of the RUN TEST 1 button was shown in Figure 13.8. In this event case, two states are enqueued: *Run Test 1* and *Wait*. The *Run Test 1* state executes first, as seen in Figure 13.11. The state machine then returns to the *Wait* state. Notice that the

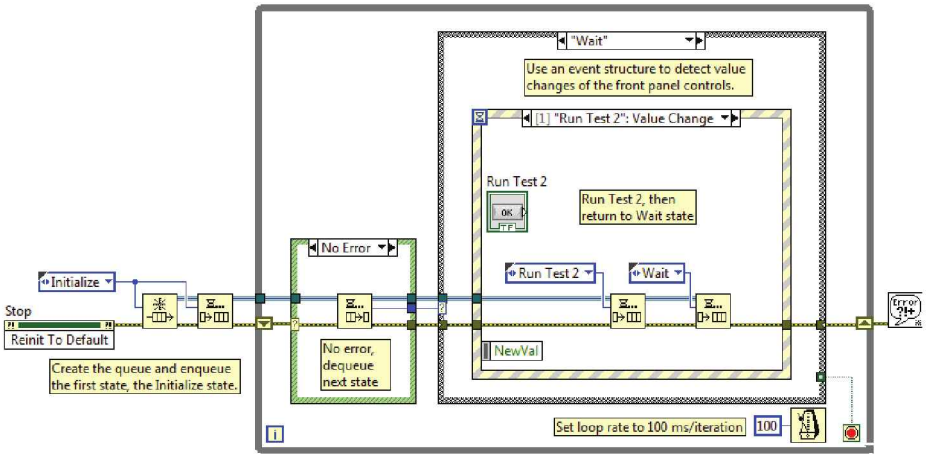


Figure 13.12: This figure shows the event case handling the value change of the RUN TEST 2 button

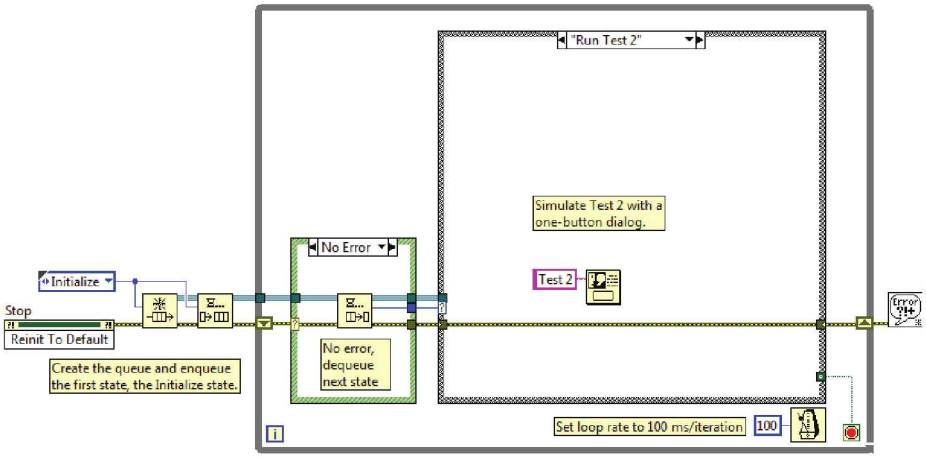


Figure 13.13: The *Run Test 2* state of the `Multitest` VI

Run Test 1 state does not have any code specifying the next state transition. The return to the *Wait* state is dictated by the queue, not by the *Run Test 1* state.

Similarly, the event case handling the value change of the RUN TEST 2 button enqueues the *Run Test 2* state and the *Wait* state, as shown in Figure 13.12. The *Run Test 2* state is shown in Figure 13.13.

Finally, the event case handling the value change of the STOP button is shown in Figure 13.14. The **`Enqueue Element At Opposite End`** function is used to place the *Shutdown* state at the front of the queue for immediate execution. This is a good programming practice in queued state machines, since it makes the code immediately

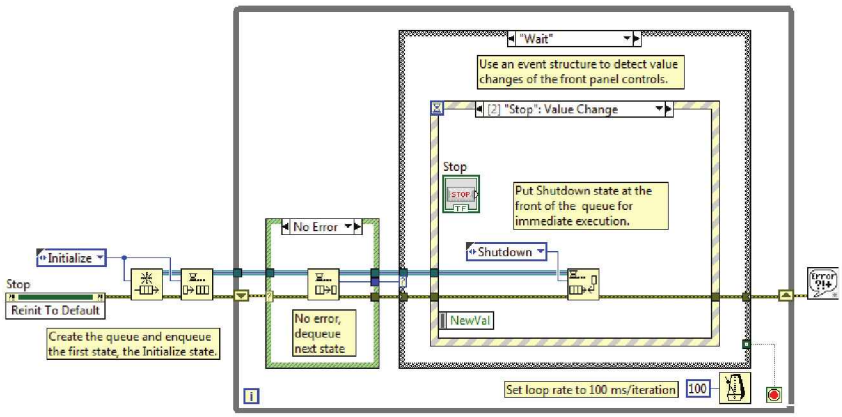
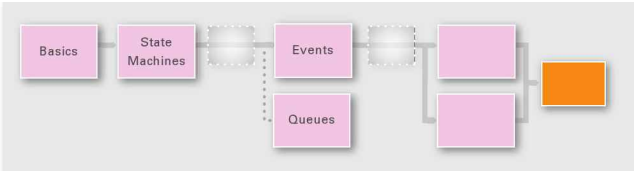


Figure 13.14: This figure shows the event case handling the value change of the STOP button

responsive to a shutdown request. The STOP button has been placed in the event case to ensure that it is read when the case is executed.



14 The Queued Car Wash Controller

Upgrading the event-driven car wash controller to a queued car wash controller represents a major shift in the conception of the program. In Chapter 2, two state transition diagrams were presented for the car wash controller. Figure 2.5 represents the classic state machine approach. Each state leads to the next, and some of the transitions depend on which type of wash was selected.

Figure 2.6 shows the states grouped by wash type. The states are treated as sequences rather than individually. This diagram represents the approach taken by a queued car wash controller. In this implementation of the car wash controller, a pre-determined sequence of events takes place when the user selects a wash type. Once this sequence has been initiated, there is no need to refer to the wash type to determine the state transitions.

This approach is easily implemented using queues. The state machine architecture is still very similar to the event-driven classic state machine developed in Chapters 4 and 10. However, the addition of queues impacts every state, so we will examine this new version of the car wash controller in detail.

Figure 14.1 shows the *Initialize* state of the queued implementation of the car wash controller. The queue is created in the input area of the state machine. The type-defined enumerated constant containing the list of states is used to determine the data type of the queue elements in the **`Obtain Queue`** function. This enumerated constant is based on the same type definition used in Chapters 4 and 10. The *Initialize* state is then enqueued as the first state.

The error wire from the **`Enqueue Element`** function in the input area is used to initialize the error shift register. As discussed in Chapter 13, the queue reference wire enters the main loop through a tunnel instead of a shift register, because the queue reference value is a constant. Inside the main loop, the first function to execute is the `Stop or Error` VI. Recall that the purpose of this VI is to check the STOP button and the status of the error wire. If the STOP button has not been pressed and there is

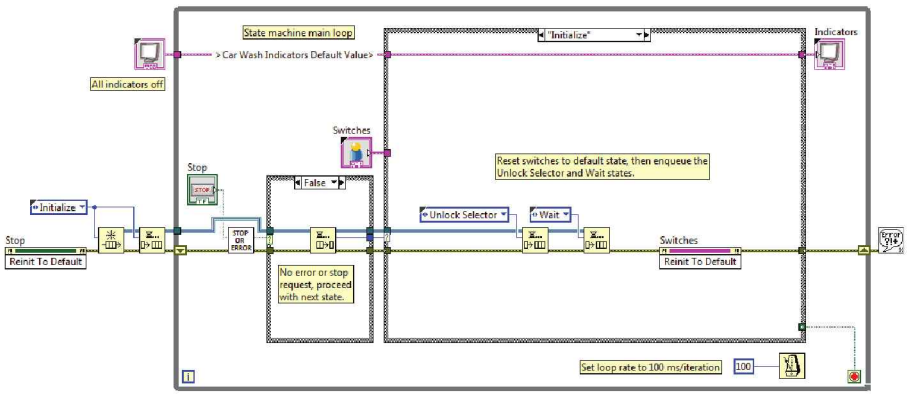


Figure 14.1: Overall structure of a queued event-driven state machine implementation of the car wash controller; notice that the queue reference wire needs to be routed around the `Stop or Error` sub-VI

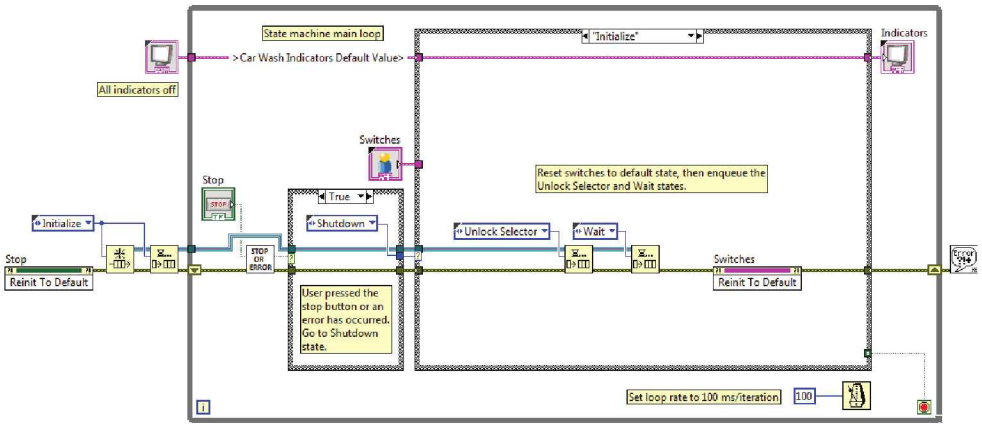


Figure 14.2: The guard clause bypasses the queue and transitions directly to the *Shutdown* state if the user presses the STOP button or if an error is present

no error, the False case of the guard clause is executed, and the next state is dequeued. If an error is present or the STOP button has been pressed, the guard clause bypasses the queue and transitions directly to the *Shutdown* state, as shown in Figure 14.2.

The queue reference wire in these figures is obstructed by the `Stop or Error` sub-VI. The queue reference wire is routed around the sub-VI, but this produces kinks in the reference wire. It is good LabVIEW practice to minimize the number of wire bends on the block diagram to improve readability. This is especially true for error wires and reference wires. These wires often tie together groups of functions with a common purpose, such as file I/O or queue manipulation. Straight references wires keep these functions visually aligned, improving readability.

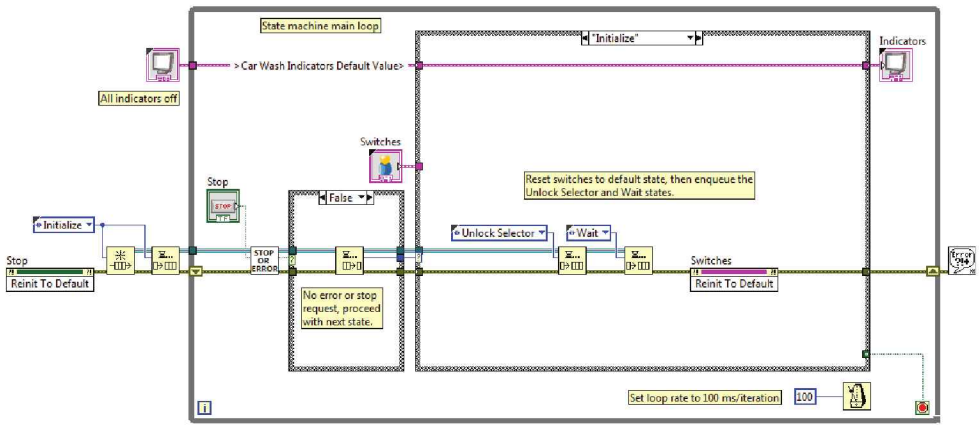


Figure 14.3: Adding queue reference input and output terminals to the `Stop or Error` sub-VI keeps the queue reference wire straight but degrades the cohesion and reusability of the sub-VI

One way to keep the reference wire straight is to add input and output queue reference terminals to the upper-left and -right corners of the `Stop or Error` connector pane, as shown in Figure 14.3. This keeps the queue reference wire straight, but it is undesirable for two reasons. First, the queue reference has nothing to do with the functionality of the `Stop or Error` sub-VI. Adding unneeded inputs to a sub-VI degrades its cohesion. Second, the queue reference input and output are now specific to the data type of the enumerated constant used by the car wash controller. That means that the `Stop or Error` sub-VI could not be used in other programs unless those programs used the exact same queue element as the car wash controller. This degrades code reusability.

14.1 Building a Non-Square Icon

One way to keep the queue reference wire straight and still maintain cohesion and code reusability is to use a non-square icon for the `Stop or Error` sub-VI. If the sub-VI had a shorter icon like the one shown in Figure 14.4, the queue reference wire could pass above the sub-VI without kinks.

To change the icon of the `Stop or Error` sub-VI, right-click on the icon and bring up the icon editor shown in Figure 14.5. The icon editor also can be brought up by double-clicking on the icon you wish to edit. After clearing the old icon layers with *Edit... Clear All*, draw a rectangle that only encloses the desired area using the drawing tools at the right side of the editor. The icon terminals can be made visible



Figure 14.4: Icon for the non-square `Stop or Error` sub-VI

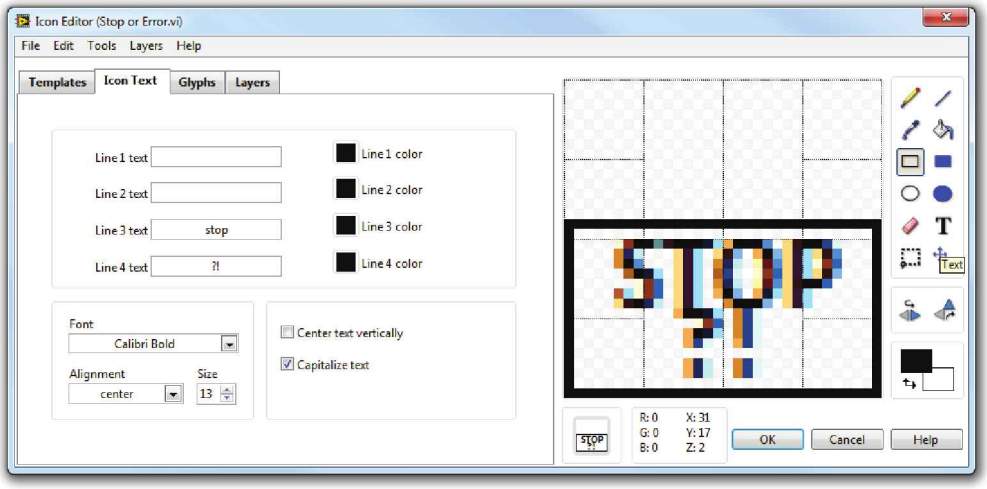


Figure 14.5: Building a non-square icon

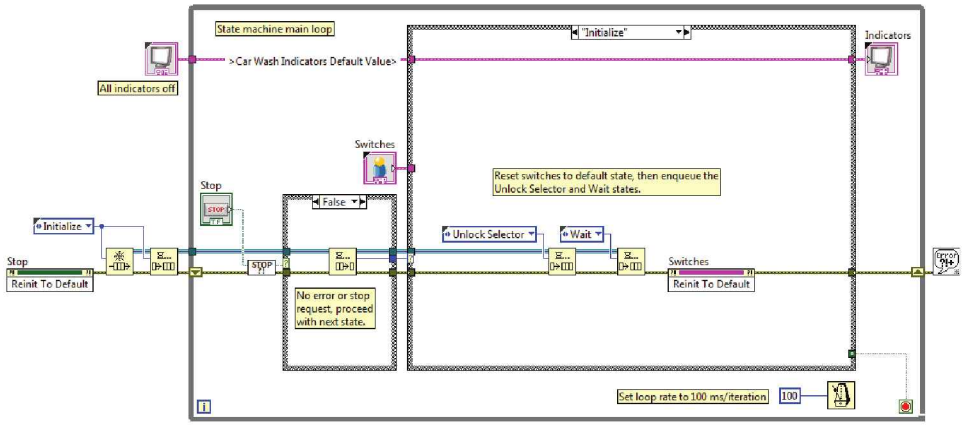


Figure 14.6: The *Initialize* state of the queued car wash controller using the non-square `Stop or Error` icon

in the editor by selecting *Edit...Show Terminals*. Figure 14.5 also shows how the icon text can be inserted using the *Icon Text* menu of the icon editor. Make sure that the *Center Text Vertically* option is unchecked, or the text will spill outside the bounding rectangle of a short icon. When the icon is finished, wire the connector pane as usual using only the terminals in the bounding box of the icon.

The impact of the new version of the `Stop or Error` sub-VI can be seen in Figure 14.6. The queue reference wire is now straight and passes above the `Stop or Error` sub-VI. Notice how the queue reference wire runs like a visual backbone from

left to right through the middle of the block diagram, increasing the readability of the program.

This may seem like a small thing, but good readability is made up of many small touches such as this one. Good style is important in any programming language, but it is especially important in LabVIEW because of its graphical nature. A good LabVIEW programmer should always take steps to make a VI as clean, clear, and readable as possible. In this case, the `Stop or Error` sub-VI has a good chance of being reused in other state machine codes, so any effort made now to improve readability will likely benefit future projects.

14.2 Examining the Queued Car Wash Controller

Now that we've straightened the reference wire, let's examine the *Initialize* state itself. This state initializes the proximity switches just as it did in the classic state machine implementation. This state also dictates transitions to the *Unlock Selector* state followed by the *Wait* state.

The *Unlock Selector* state is shown in Figure 14.7. The purchase selector is initialized and enabled in this state. No code is needed to dictate the transition to the next state, since the next state is already waiting on the queue.

The *Initialize* and *Unlock Selector* states illustrate the utility of queues. Multiple states can be enqueued at one time, and individual states no longer need to dictate transitions to the next state.

Queues and Event Structures

The real power of queues is found in the *Wait* state. This state uses an event structure to monitor the user interface described in Chapter 11, but the code in the various

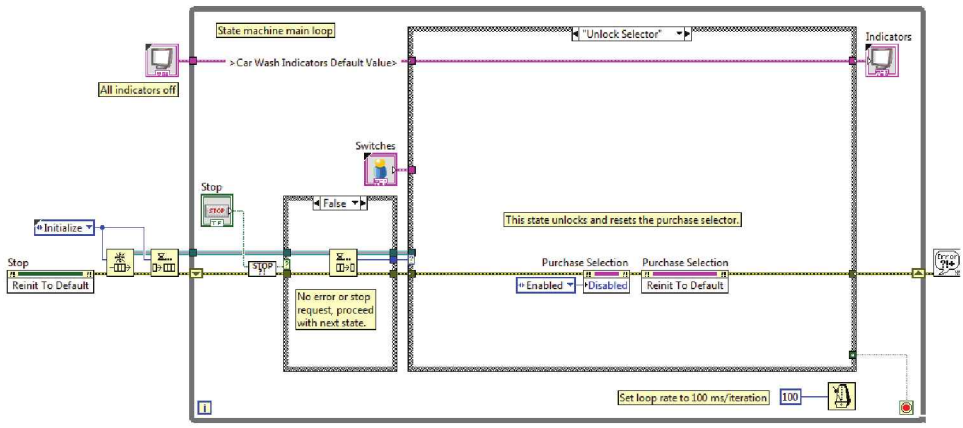


Figure 14.7: The *Unlock Selector* state of the queued car wash controller

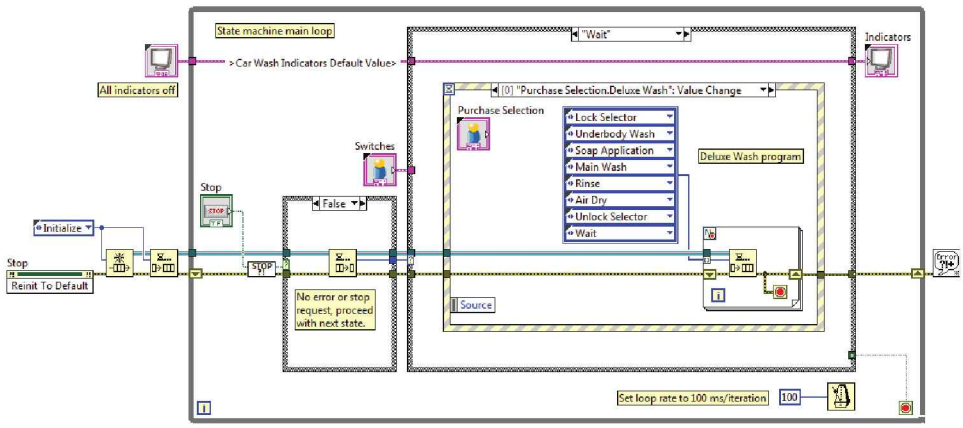


Figure 14.8: The event case handling the value change of the DELUXE WASH button, where the entire sequence of wash cycles for the Deluxe Wash are loaded onto the queue in this event case

event cases now uses queues to specify sequences of states instead of transitions to individual states.

The event case handling the value change of the DELUXE WASH button is shown in Figure 14.8. In this event case, the entire sequence of states for a Deluxe wash is placed on the queue. The states are first placed in an array, and the array is then fed into a **For** loop. Nothing has been wired to the *N* terminal of the loop to specify the number of times the loop is to be run—auto-indexing is used to determine the number of loop iterations. The loop will run a number of iterations equal to the length of the array, passing the elements of the array into the loop one element per loop iteration. The individual states in the array will be placed on the queue in order from the first array element to the last. The eight states needed for a Deluxe wash are enqueued using one **Enqueue Element** function in a **For** loop. This is a more efficient use of screen space than wiring eight **Enqueue Element** functions in series.

> **Style Guide:** Try to make your programs fit on one screen (if possible) to reduce the need for navigation and improve readability. If you can't fit the program in one screen, arrange your code so that scrolling is only needed in one direction.

The **For** loop also has a conditional terminal, which is an option discussed in Chapter 1. If a conditional terminal is used, the **For** loop executes specified number of iterations unless it is aborted by the stop terminal. In Figure 14.8, the error wire from the **Enqueue Element** function is wired directly to the stop terminal. The status Boolean of the error wire is is checked each loop iteration to determine whether the loop should continue or not.

Notice too that the error wire in the **For** loop has been placed on a shift register. This is good programming practice, because it ensures that any warnings generated

in the loop are trapped and passed on. If an error is generated inside the loop, the conditional terminal causes the loop to abort, but warnings do not cause the loop to abort. Warnings have error codes, but the status Boolean in the error cluster remains False. The error shift register is necessary to trap any warnings that are generated and to pass them on once the loop has ended.

The queue reference wire enters the **For** loop only once and is not put on a shift register. This is because queues are passed by reference instead of by value. Even though the elements of the queue change throughout the course of the loop, the reference to the queue does not change. Therefore it doesn't need to be placed on a shift register.

Handling the Purchase Selector and the Stop button

Now let's examine the placement of the purchase selector. The terminal for the purchase selector cluster has been placed in the DELUXE WASH value change case even though it is not wired to anything. Why was this done? Recall from Chapter 10 that it is good programming practice to put a control within the event case that handles its value change if possible. This ensures that LabVIEW reads the control when it is pressed and that the mechanical action of the control executes as expected.

The individual DELUXE WASH and ECONOMY WASH buttons have mechanical actions of *Switch When Pressed*. Since the controls are switch-action oriented rather than latch-action, LabVIEW does not need to read the buttons to ensure proper execution of their mechanical actions. This wasn't important in the classic and event-driven implementations of the car wash controller, but it is important here. The reason for this is that, in the classic and event-driven implementations, a single event case was configured to wait for the value change of the entire purchase selection cluster—not the individual controls in the cluster. The mechanical action of the Boolean controls didn't matter.

In the queued car wash controller, the value changes of the DELUXE WASH button and of the ECONOMY WASH button are handled in different event cases. Look back at Figure 11.1. Notice that the event case is looking for a value change of the entire cluster, considering <*All Elements*> of the cluster. Compare this to Figure 14.9, where the event case is configured to handle the value change of just the DELUXE WASH button. Instead of looking for the value change of the cluster as a whole, we are now looking for the value change of one element of the cluster. This is necessary because we need to enqueue different sets of states depending on which wash was selected.

So how does this relate to the placement of the purchase selector and the mechanical action of the wash buttons? In the queued implementation of the car wash controller, we have separate event cases for each button in the purchase selector, but there is only one terminal for the purchase selector itself. We can place the selector in either the DELUXE WASH value change case or the ECONOMY WASH value change case, but not in both of them. If the buttons in the purchase selector have a latching action, the behavior of the buttons will be different from each other. The button that has the purchase selector in its event case will be read when the button is pressed, but other button will not be read when it is pressed. One way around this difficulty is to use switching action rather than latching action on the buttons. Since the program already

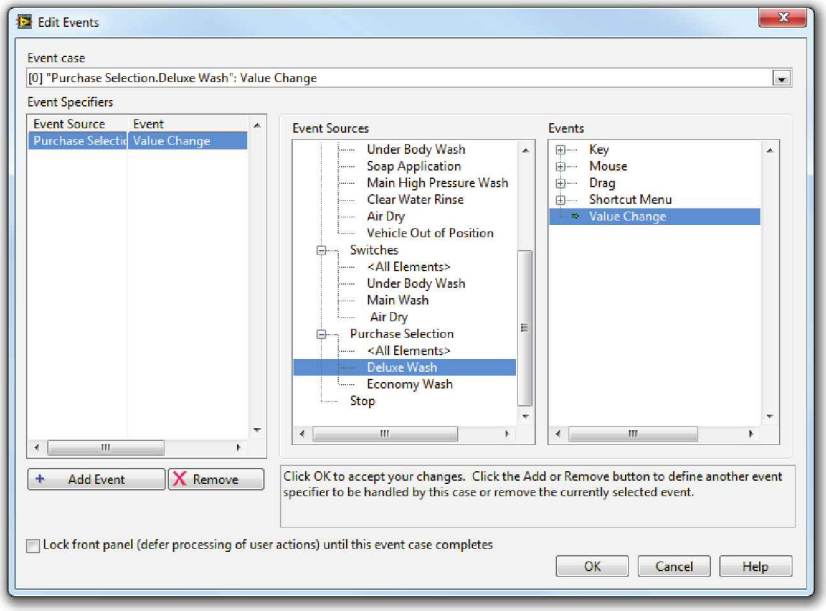


Figure 14.9: The event configuration window for the value change of the DELUXE WASH button

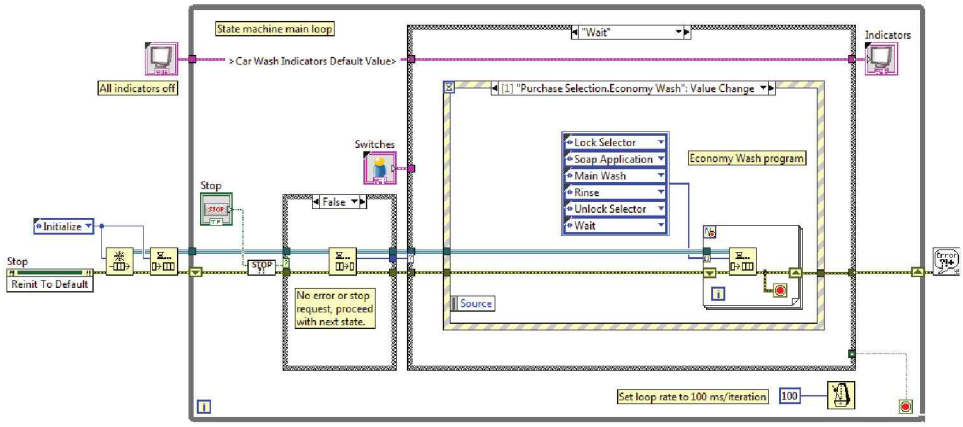


Figure 14.10: The event case handling the value change of the ECONOMY WASH button, where the entire sequence of wash cycles for the Economy wash are loaded onto the queue

reinitializes the purchase selector when the selector is unlocked at the end of the wash, the buttons will be returned to their default positions when a wash is completed.

The event case handling the value change of the ECONOMY WASH button is shown in Figure 14.10. This event case is very similar to the event case handling the value change of the DELUXE WASH button. The only difference is in which states are queued.

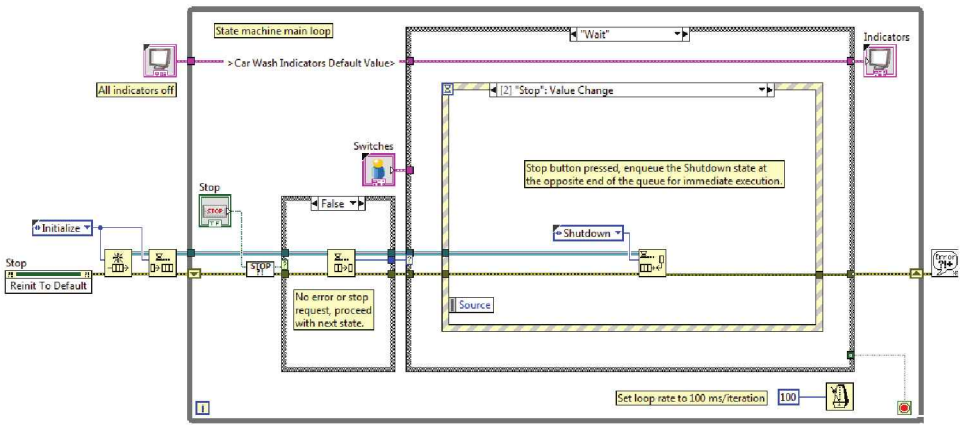


Figure 14.11: The event case handling the value change of the STOP button

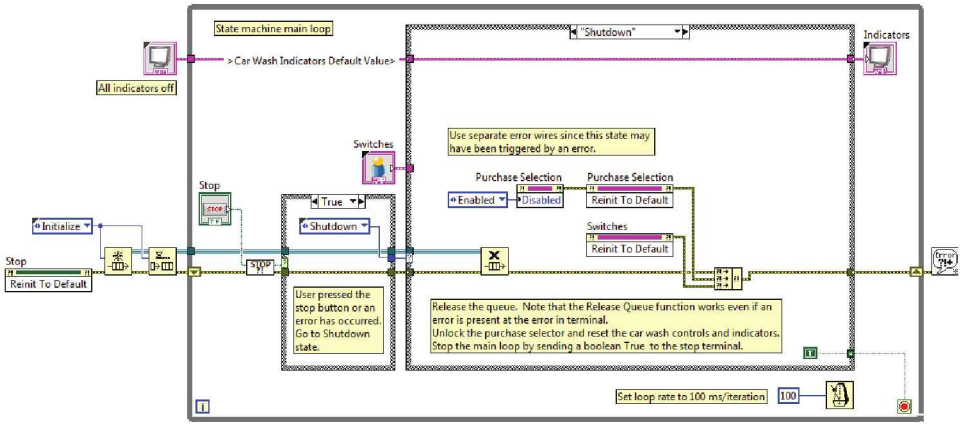


Figure 14.12: The *Shutdown* state of the queued car wash controller

The event case handling the value change of the STOP button is shown in Figure 14.11. This state enqueues the *Shutdown* state if the STOP button is pressed. The **`Enqueue Element At Opposite End`** function is used to place the *Shutdown* state at the front of the queue for immediate execution.

The *Shutdown* state is shown in Figure 14.12. The *Shutdown* state stops the main loop and reinitializes the front panel controls. This state also releases the queue reference. The queue will not be needed after the VI shuts down, and it is good programming practice to release system resources when a VI exits.

The **`Release Queue`** function is placed on the main error wire, but the property nodes are placed on parallel error wires. This is because the **`Release Queue`** function operates normally if there is an error present at its *error in* terminal, but property nodes will not execute if an error is present. This parallel wiring arrangement ensures that

all of the functions in the *Shutdown* state execute even when an error is present on the main error wire. The parallel error wires are then merged back into the main error wire using the **`Merge Errors`** function.

14.3 The Car Wash States

When a wash type has been selected, the first state executed is the *Lock Selector* state shown in Figure 14.13. In the classic and event-driven implementations, the *Lock Selector* state needs to know which wash type is selected in order to choose the correct state transition. This is no longer the case. The *Lock Selector* simply executes and depends on the queue for determining the next state transition. The DELUXE WASH

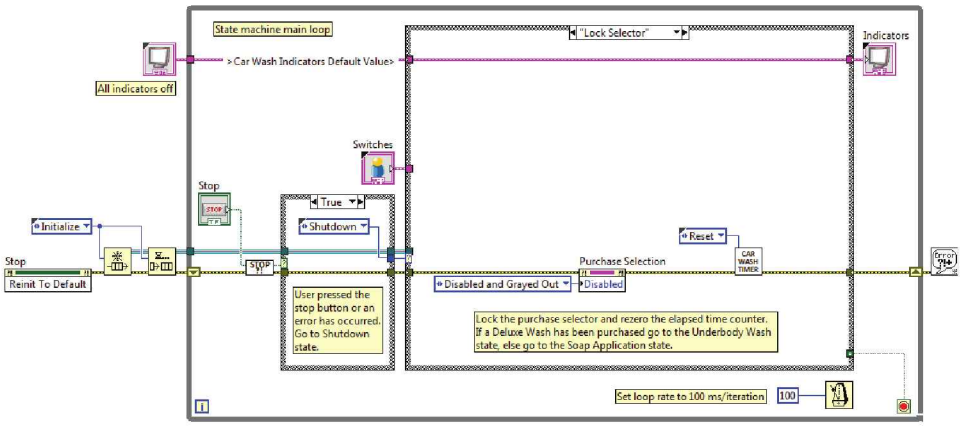


Figure 14.13: The *Lock Selector* state of the queued car wash controller

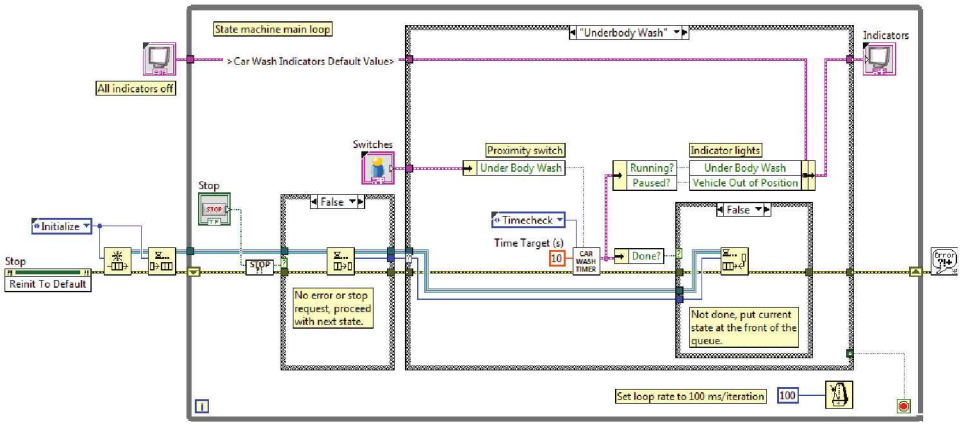


Figure 14.14: The *Underbody Wash* state of the queued car wash controller, where if the cycle is not done, the *Underbody Wash* state is placed at the front of the queue for immediate execution

Boolean is removed from the data highway, since it is no longer needed. The car wash states are now totally decoupled from the type of wash selected.

The car wash cycle states are also affected by the addition of a queue to the car wash controller. In the classic state machine and event-driven state machine car wash controllers, each car wash state needs to know which state to go to next. They have to decide whether to return to the current state or to move on to the next state, depending on whether the target time has elapsed. If it is time to move on to the next state, each car wash cycle state needs to know which cycle came next. Sometimes this required knowledge of which car wash type is chosen.

In the queued implementation of the car wash controller, the cycle states don't need to know the next state or the wash type, but they still need to decide whether or not to repeat the current state. Figure 14.14 shows the *Underbody Wash* state. The car wash timer action engine is used to determine if the cycle is finished or not.

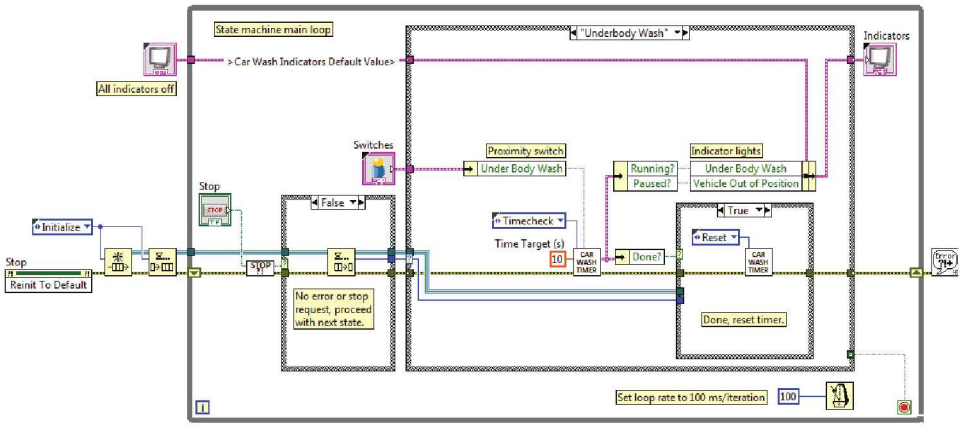


Figure 14.15: The *Underbody Wash* state of the queued car wash controller, where if the cycle is done, the timer is rezeroed and no operations are performed on the queue

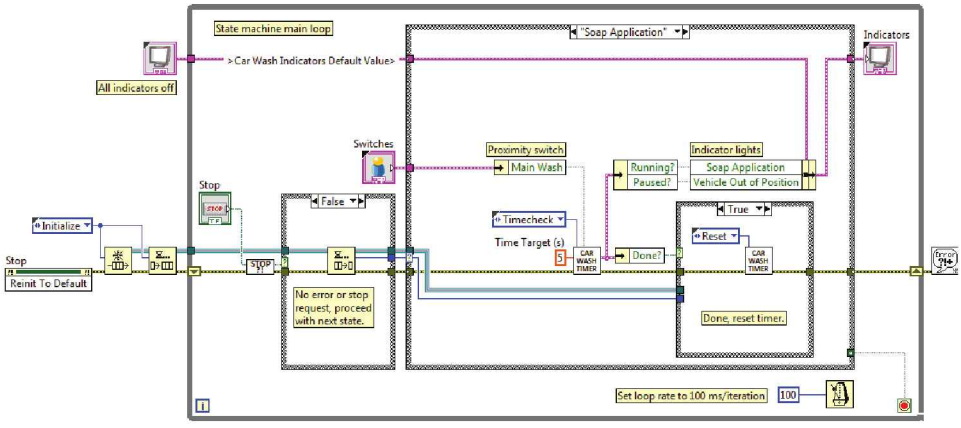


Figure 14.16: The *Soap Application* state of the queued car wash controller

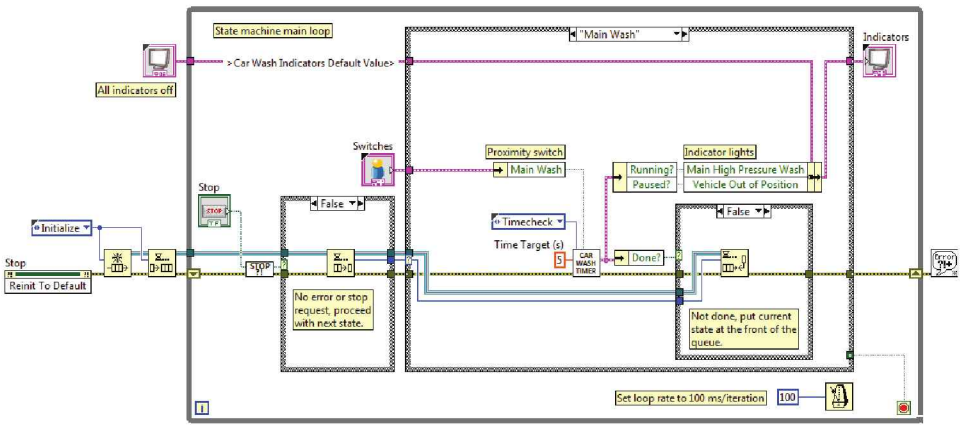


Figure 14.17: The *Main Wash* state of the queued car wash controller

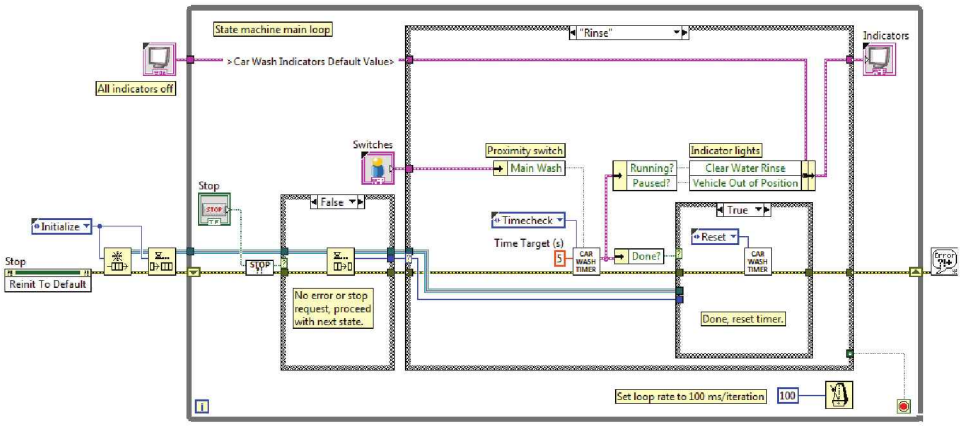


Figure 14.18: The *Rinse* state of the queued car wash controller, where if the cycle is done, the timer is rezeroed and the car wash controller moves to the next wash state

If the cycle is not finished, the current state is placed back on the front of the queue using the **`Enqueue At Opposite End`** function. If the current cycle is finished, the timer is reset, but no operations are performed on the queue, as shown in Figure 14.15. The next state of the wash sequence is then dequeued when the loop iterates again.

The other wash cycle states are all similar; only the unbundled proximity switches, value of the time targets, and bundled cycle indicators change. All of the wash states now have identical structures. All of the wash states have become independent of each other, and all are independent of the wash cycle chosen. Each wash cycle state exists on its own—unaware that other cycle states even exist. The *Soap Application*

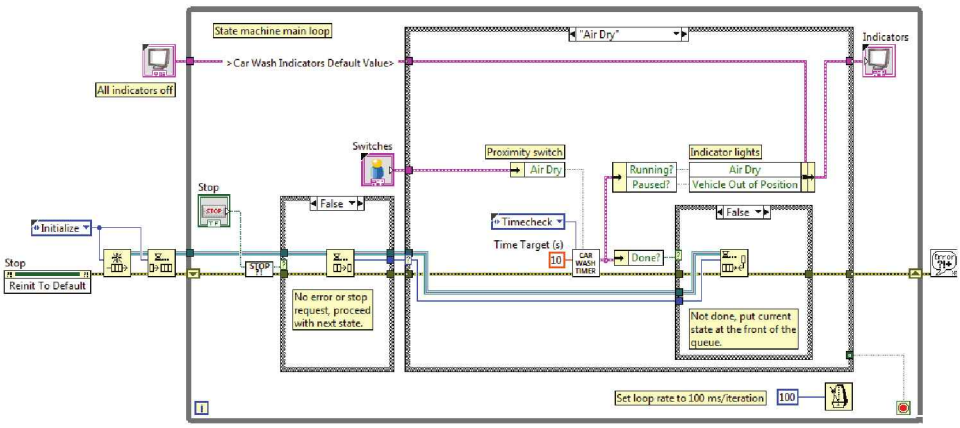
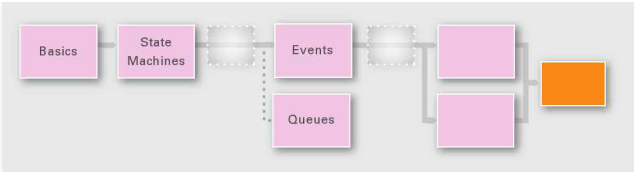


Figure 14.19: The *Air Dry* state of the queued car wash controller

state is shown in Figure 14.16, the *Main Wash* cycle in Figure 14.17, the *Rinse* cycle in Figure 14.18, and the *Air Dry* cycle in Figure 14.19.



15

Data Acquisition with the Multitest VI

This book is focused on LabVIEW as a programming language rather than as a data acquisition tool. But since LabVIEW is extensively used for data acquisition, it hardly seems fair to ignore the subject entirely. This chapter demonstrates how to write a data acquisition VI using the `Multitest` VI as a template. The example VI allows a user to null, calibrate, and read a strain gage connected to a quarter-bridge Wheatstone bridge.

15.1 Strain Gages and Bridge Circuits

Strain is a measure of change of shape and of deformation from a given reference state. Strain is usually expressed as the ratio of a change in length to a reference length. For example, Figure 15.1 shows a bar under an axial load. Suppose that the original length of the bar is l_0, and the bar stretches to a length $l_0 + \Delta l$ when a load is applied. The strain ε is equal to $\Delta l / l_0$. Strain is dimensionless, since it is a ratio of lengths.

Strain gages are essentially resistors bonded to a surface where strain is to be measured. When the surface deforms due to strain, the strain gage also deforms, changing its resistance. The ratio of the change in resistance to the original resistance ($\Delta R / R_0$) is proportional to the strain. This factor of proportionality is known as the gage factor. If the gage factor is known and the change in resistance of the strain gage can be measured, the strain at the surface beneath the strain gage can be calculated.

Strain gages are usually used to measure small strains (often smaller than 0.1%). This means that the change in resistance of a strain gage is also small. Small changes in resistance are difficult to measure directly, so a special circuit known as a bridge circuit is used to convert small changes in resistance to small changes in voltage which

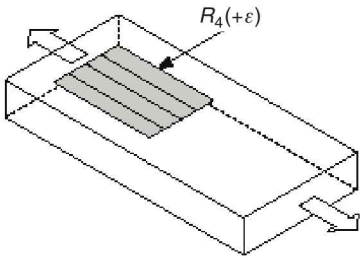


Figure 15.1: Measuring axial strain with a single strain gage

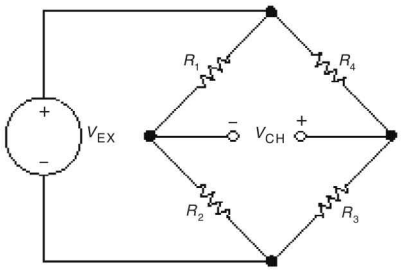


Figure 15.2: Bridge circuit for measuring strain

are more easily measured. A typical bridge circuit, known as a Wheatstone bridge, is shown in Figure 15.2.

Notice in Figure 15.1 that the strain gage on the bar is designated as R_4 and that one of the legs of the bridge circuit in Figure 15.2 is also designated as R_4. The strain gage is wired into the bridge circuit in the R_4 location. The other three resistors are either fixed resistors of known values or variable resistors (potentiometers). This is called a quarter-bridge configuration, since the strain gage makes up one quarter of the bridge configuration. The value of V_{CH} is the bridge output voltage, and V_{EX} is the excitation voltage. If the bridge circuit is correctly nulled and calibrated, the ratio of V_{CH}/V_{EX} will be proportional to the ratio of $(\Delta R_4/R_4)$ and thus proportional to the strain.

The strain reading is accurate only if the bridge circuit is properly nulled and calibrated. Typically, at least one of the non-gage resistors in the bridge is a potentiometer whose resistance can be varied over a range of values. When no strain is applied to the gage, the bridge output should be zero. Nulling the bridge means varying the potentiometer resistance when the gage is unstrained until the bridge output is zero.

Calibrating the bridge consists of placing a shunt resistor of known value in parallel with the gage resistor. This changes the resistance of the the leg of the bridge containing the strain gage and simulates imposing a known strain on the test specimen. The bridge output is then read and the strain calculated. The calculated strain is compared to the known value of the simulated strain to calibrate the bridge output.

15.2 Data Acquisition in LabVIEW

Data acquisition (DAQ) in LabVIEW is performed using the DAQmx functions found in the *Measurement I/O* sub-palette of the *Functions* palette. The DAQmx functions are very similar to the file I/O functions. The file I/O functions operate on a file reference. When a file is opened, a file reference is created, and that reference must be passed as an input to other LabVIEW functions that read from and write to the file. Similarly, the DAQmx functions operate on a DAQ task reference. To perform a data acquisition task in LabVIEW, a DAQ task reference first must be created. Channels are then added to the task to represent the different measurements that need to be made. The strain measurement described in this chapter is represented by a strain channel added to the DAQ task. Timing and triggering data also can be added to a task. A single task can have many channels, as long as the channels all share the same timing and triggering requirements as well as the same directionality (input or output). Once the task has been defined, the task reference can be passed to other DAQmx functions that perform other functions, such as nulling, shunt calibration, and data acquisition.

15.3 Functional Specification

Devices that acquire data from strain gages often include internal signal conditioning circuitry that provides the Wheatstone bridge circuit. These devices also may have internal circuitry for nulling and calibrating the bridge that can be enabled by software commands. In this chapter, we assume that the user wishes to measure the output of a strain gage in a quarter-bridge configuration using data acquisition hardware with software-enabled bridge nulling and calibration capabilities.

The software should allow the user to configure the measurement by entering data into a cluster on the front panel. The user should be able to null the bridge or calibrate the bridge at any time. When the user wishes to read the gage, a burst of readings is made, and the average result is displayed on the front panel. The number of readings in the burst and the sampling rate is set by the user in the configuration cluster. The user is able to change any of the values in the configuration cluster while the program is running. The configuration cluster contains a significant amount of user-entered information, so it should never be reinitialized to default by the program. The front panel of the VI looks something like the one shown in Figure 15.3.

15.4 State Transition Diagram

One possible state transition diagram for this functional specification is shown in Figure 15.4. When the program starts, it initializes the front panel, and then builds the DAQ task to prepare for taking the data. The program will then wait. The user can then choose to null the bridge, perform shunt calibration, or read the strain gage.

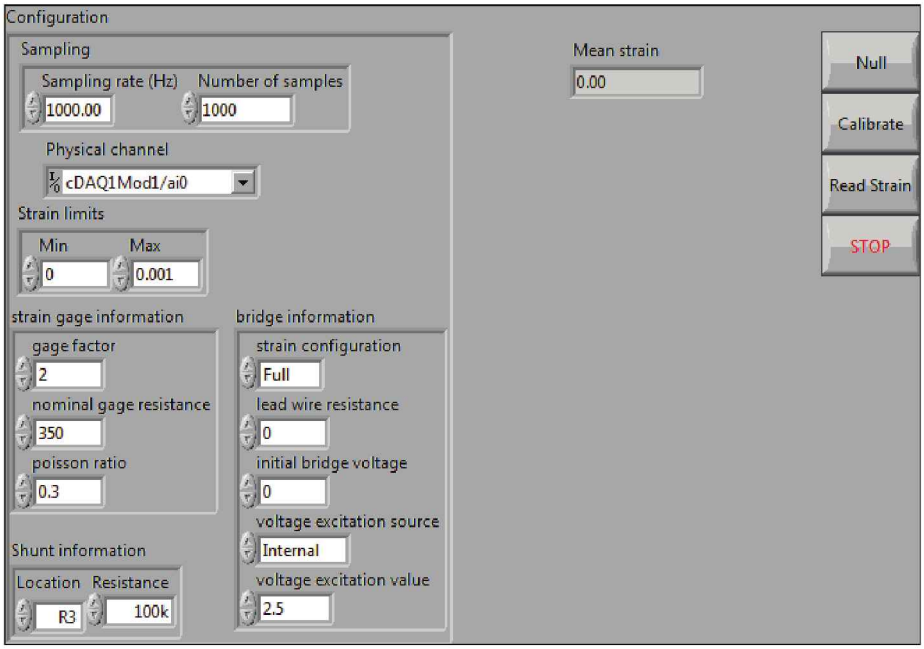


Figure 15.3: Front panel of the `Multitest` data acquisition VI

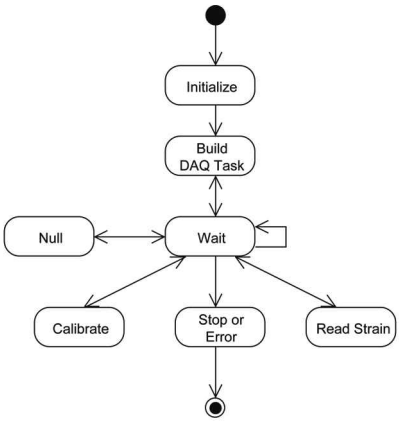


Figure 15.4: State diagram for the `Multitest` data acquisition VI

The state machine returns to the *Wait* state after each of these actions. The program ends if the user presses the STOP button or if an error occurs.

Notice that the program can also revisit the *Build DAQ Task* state from the *Wait* state. The functional specification states that the user is able to change the configuration parameters while the program is running. The DAQ task needs to be rebuilt if the parameters change, so the state transition from the *Wait* state to the *Build DAQ Task*

state and back again is necessary to fulfill the requirements of the functional specification. This transition occurs whenever the configuration cluster changes.

Notice the similarity between this state transition diagram and the one for the `Multitest` VI shown in Figure 2.2. The strain measurement VI is essentially the same as the `Multitest` VI with three tests: *Null*, *Calibrate*, and *Read Strain*. This gives us a plan of attack. The strain measurement VI is based on the `Multitest` framework. We use the queued-string version of the `Multitest` VI developed in Chapter 21 as the basis for the strain measurement VI.

15.5 Building the DAQ Task

Before developing the main VI, we need to look at the specific tasks involving the strain gage. In order to null the bridge, perform shunt calibration, or read the strain, we first need to build a DAQ task. Building a DAQ task requires three steps:

1. Create a task reference.
2. Add channels for the various measurements.
3. Configure the timing of the task.

The DAQ task is created using the **`DAQmx Create Task`** function. The icon and connector pane of this native LabVIEW function is shown in Figure 15.5. The strain channel is added to the DAQ task using the native **`DAQmx Create Channel`** function shown in Figure 15.6. This function configures the physical channel of the data acquisition system, specifies the details of the strain gage used, and also specifies the configuration of the Wheatstone bridge. The timing information is then added to

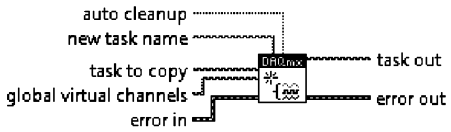


Figure 15.5: Icon of the **`DAQmx Create Task`** VI

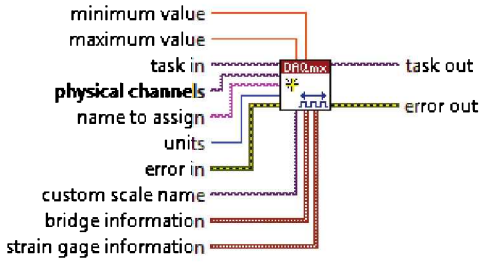


Figure 15.6: Icon of the **`DAQmx Create Channel`** VI

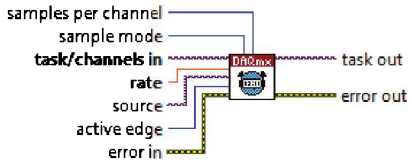


Figure 15.7: Icon of the `DAQmx Timing` VI



Figure 15.8: Icon of the `DAQmx Clear Task` VI

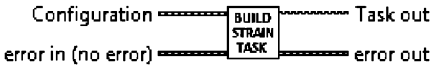


Figure 15.9: Icon of the `Build Strain Task` sub-VI

the task using the native **`DAQmx Timing`** function shown in Figure 15.7. This function specifies the number of points to be read in each burst of data and the sampling rate of the burst. When it is no longer needed, an existing DAQ task is cleared by the native **`DAQmx Clear Task`** function shown in Figure 15.8.

Wiring Tip: If the DAQmx functions do not appear in your function palette, download the DAQmx drivers from `www.ni.com`.

These three DAQmx functions are used to build the DAQ task in the `Build Strain Task` sub-VI. The icon for this sub-VI is shown in Figure 15.9, and the block diagram is shown in Figure 15.10. This sub-VI accepts the type-defined configuration cluster shown in Figure 15.3 as an input. This configuration cluster is created by combining all of the data needed to configure the DAQ task into one cluster, allowing all of the configuration data to be passed on a single wire. The data is organized into sub-clusters to group related items together: strain gage data, bridge configuration data, etc. The `Build Strain Task` unbundles the data as necessary and uses it to perform the three tasks listed: create a new task, configure a strain channel, and add timing information.

The icons of the **`DAQmx Create Channel`** VI and the **`DAQmx Timing`** VI look slightly different in Figure 15.10 than they do in Figures 15.6 and 15.7. These VIs are polymorphic; the versions used in the VI must be chosen when they are dropped on the block diagram. The version of the **`DAQmx Create Channel`** shown in Figure 15.10 is configured to perform an analog input reading from a strain gage. The version of the **`DAQmx Timing`** VI shown in Figure 15.10 configures the sample clock.

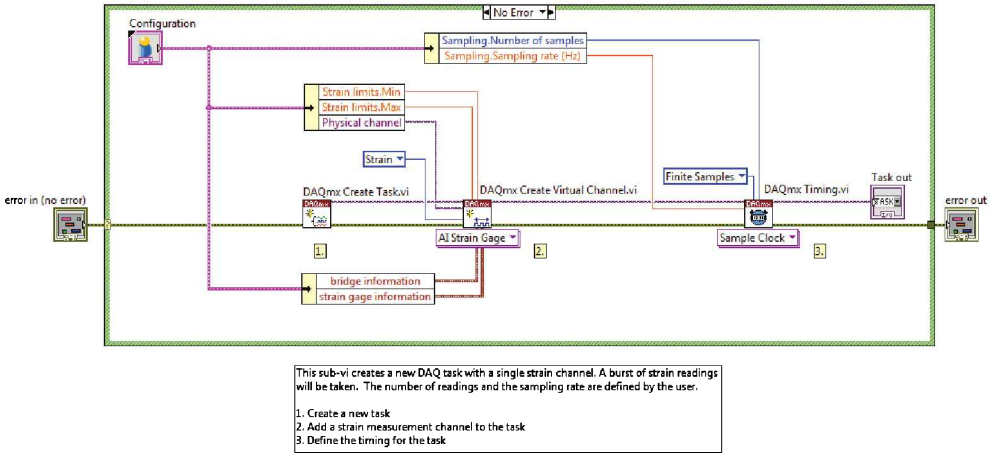


Figure 15.10: Block diagram of the `Build Strain Task` sub-VI

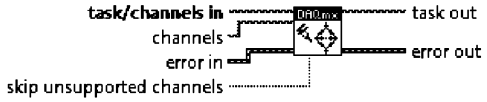


Figure 15.11: Icon for the **`DAQmx Perform Bridge Offset Nulling Calibration`** function

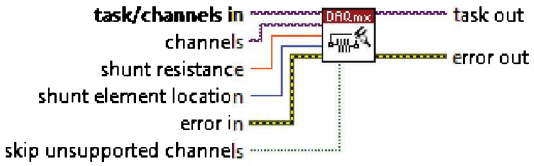


Figure 15.12: Help file for the **`DAQmx Perform Shunt Calibration (Strain)`** function

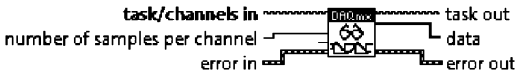


Figure 15.13: Help file for the **`DAQmx Read`** function

The output of this sub-VI is a DAQ task that is used to null the bridge, perform shunt calibration, and read the strain gage. The bridge is nulled using the native **`DAQmx Perform Bridge Offset Nulling Calibration`** function shown in Figure 15.11. The calibration is performed by the native **`DAQmx Perform Shunt Calibration (Strain)`** VI shown in Figure 15.12. The burst of strain gage readings is performed by the native **`DAQmx Read`** VI shown in Figure 15.13.

15.6 Developing the `Multitest` Data Acquisition VI

We now adapt the `Multitest` VI to the strain measurement task at hand. The front panel has been developed in Figure 15.3, and the states and transitions are shown in Figure 15.4. The groundwork for configuring the data acquisition has been laid in Figure 15.10.

The next task is to consider the placement of the various front panel controls and indicators on the block diagram. The NULL, CALIBRATE, READ STRAIN, and STOP buttons are all placed in the event cases that handle their value changes. But where do we place the configuration cluster? Chapter 1 included a discussion of this very question. The answer depends on the requirements laid out in the functional specification. The specification states that the user be able to change the configuration data while the program is running. This means that the configuration cluster needs to be placed inside the main loop. The configuration cluster is needed by both the *Calibrate* and *Read Strain* states, so it should be placed in the lobby of the state machine.

We can now develop the individual states of the `Multitest` VI. The overall structure of the state machine and the *Initialize* state are shown in Figure 15.14. The queue is created in the input area of the main loop. The queue element is an enumerated constant containing the states of the DAQ state machine: *Initialize*, *Build DAQ Task*, *Wait*, *Null*, *Calibrate*, *Read Strain*, and *Shutdown*. The *Initialize* state is enqueued in the input area of the loop.

Inside the main loop, the guard clause checks to see if the error line is clear. If there is no error, the next state is dequeued. If an error is detected, the queue is bypassed, and the state machine is sent to the *Shutdown* state, as shown in Figure 15.15. The *Shutdown* state releases the queue, clears the DAQ task, and stops the main loop.

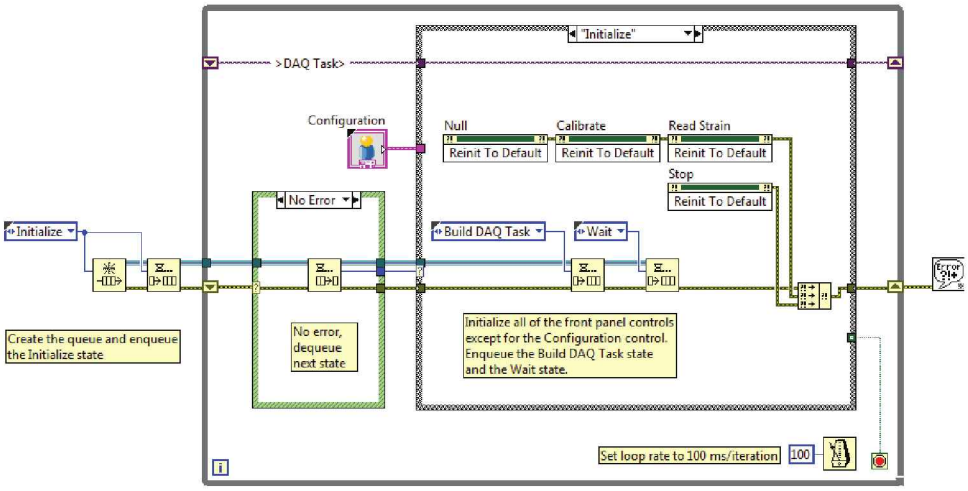


Figure 15.14: The *Initialize* state of the `Multitest` data acquisition VI

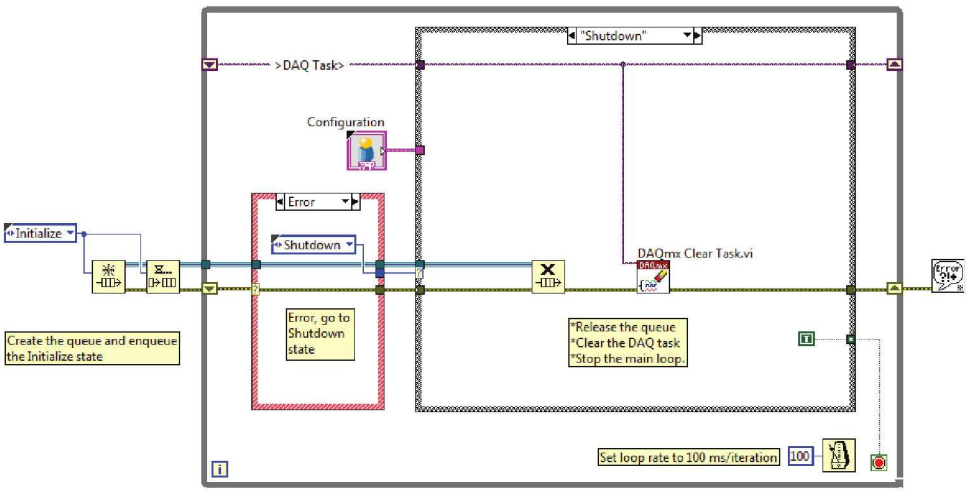


Figure 15.15: The *Shutdown* state of the `Multitest` data acquisition VI

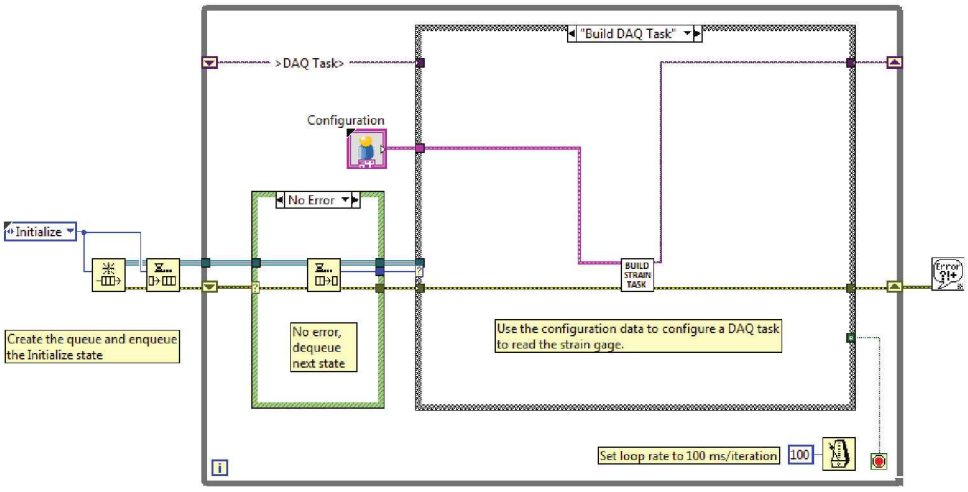


Figure 15.16: The *Build DAQ Task* state of the `Multitest` data acquisition VI

The *Initialize* state shown in Figure 15.14 reinitializes the front panel controls to their default positions—except for the configuration cluster. The *Initialize* state also enqueues the *Build DAQ Task* state and the *Wait* state (in that order).

After the *Initialize* state executes, the DAQ task is built in the *Build DAQ Task* state shown in Figure 15.16. This state feeds the configuration cluster to the `Build Strain Task` sub-VI developed previously and places the resulting DAQ task on the data highway for use by the other states.

The state machine then transitions to the *Wait* state and waits for user input. The event case handling the value change of the STOP button is shown in Figure 15.17.

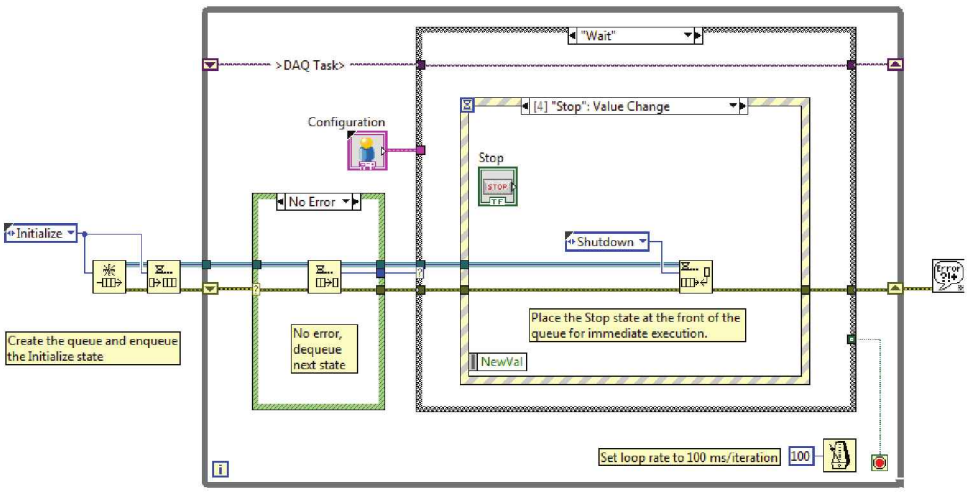


Figure 15.17: The event case handling the value change of the STOP button of the `Multitest` data acquisition VI

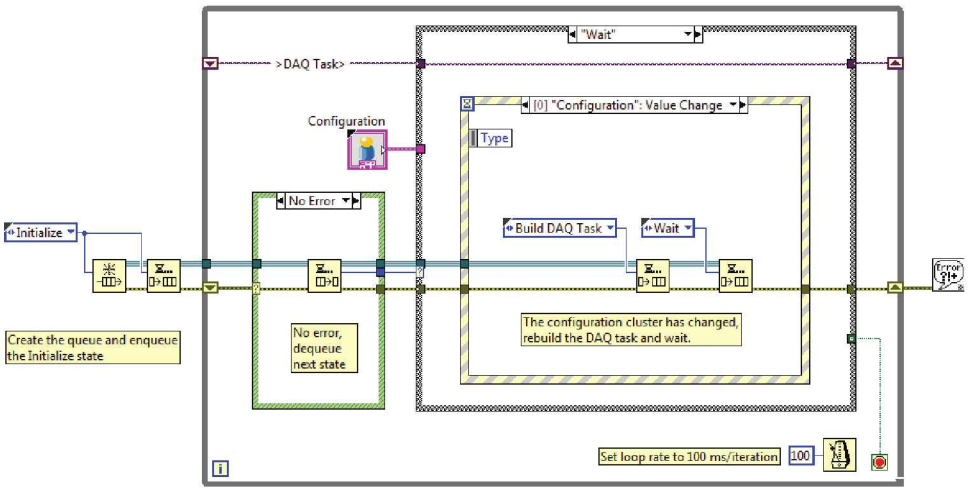


Figure 15.18: The event case handling the value change of the CONFIGURATION cluster of the `Multitest` data acquisition VI

The *Shutdown* state is placed at the front of the queue for immediate execution using the **`Enqueue Element At Opposite End`** function. The *Shutdown* state itself is shown in Figure 15.15.

If the user changes the configuration cluster, the event case shown in Figure 15.18 executes. This event case rebuilds the DAQ task and then returns the state machine to the *Wait* state.

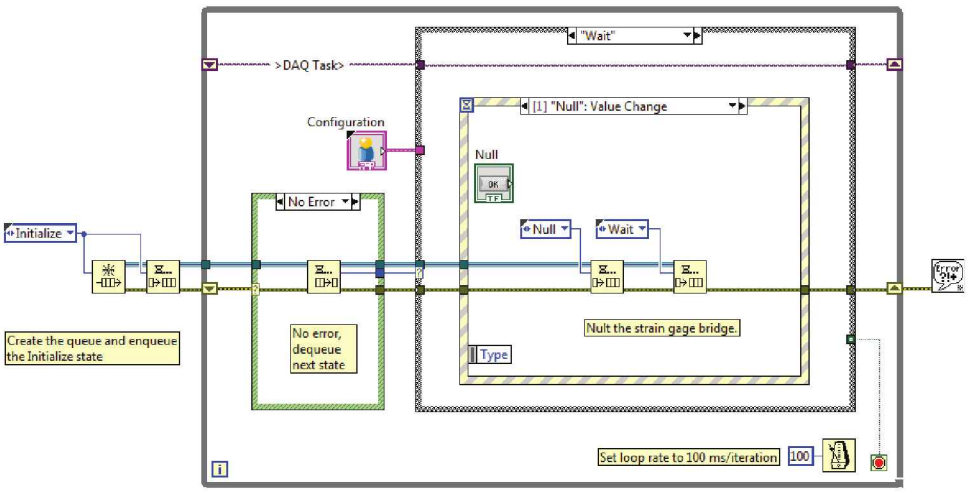


Figure 15.19: The event case handling the value change of the NULL button of the `Multitest` data acquisition VI

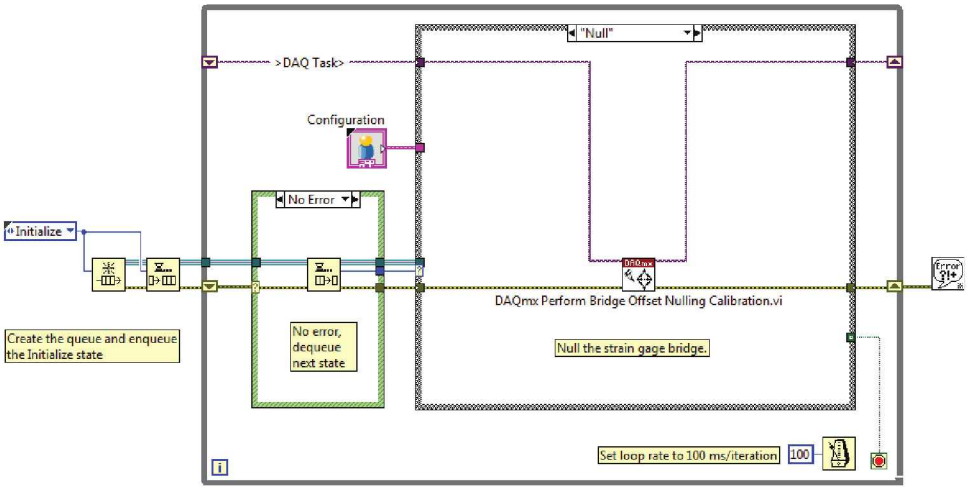


Figure 15.20: The *Null* state of the `Multitest` data acquisition VI

The event case handling the value change of the NULL button is shown in Figure 15.19. This state enqueues the *Null* state shown in Figure 15.20 and then returns to the *Wait* state. The *Null* state uses the DAQ task reference on the data highway to null the strain gage bridge using the **`DAQmx Perform Bridge Offset Nulling Calibration`** function shown in Figure 15.11.

The event case handling the value change of the CALIBRATE button is shown in Figure 15.21. This state enqueues the *Calibrate* state shown in Figure 15.22 and then

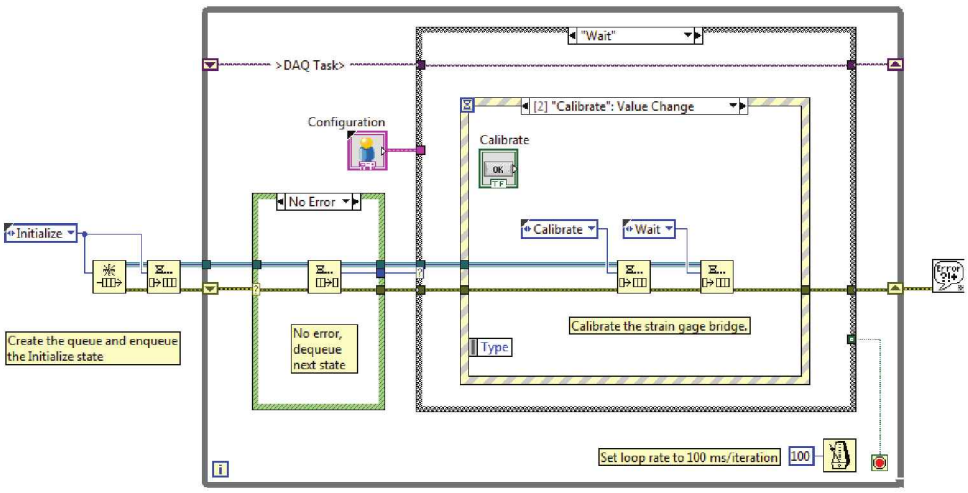


Figure 15.21: The event case handling the value change of the CALIBRATE button of the `Multitest` data acquisition VI

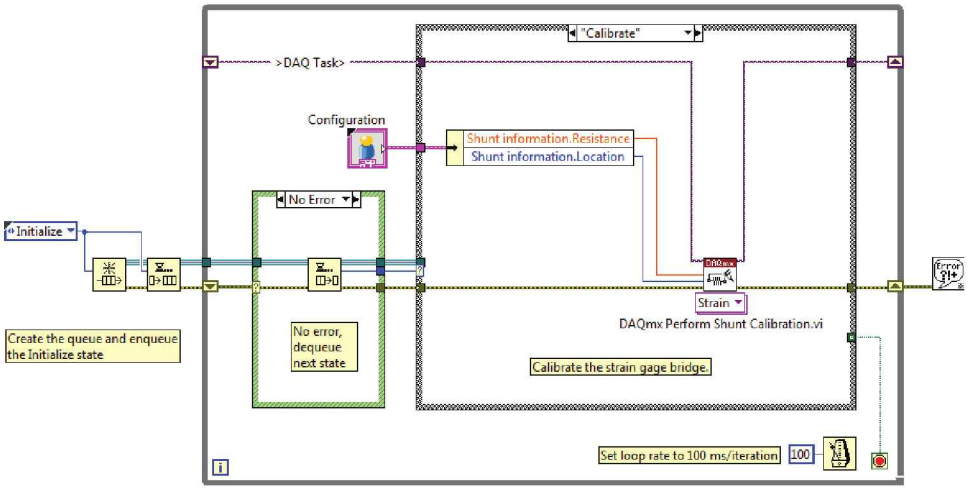


Figure 15.22: The *Calibrate* state of the `Multitest` data acquisition VI

returns to the *Wait* state. The *Calibrate* state uses the DAQ task reference on the data highway and the configuration cluster to perform shunt calibration on the strain gage bridge using the **`DAQmx Perform Shunt Calibration (Strain)`** VI shown in Figure 15.12. This is a polymorphic VI; the version shown in Figure 15.22 is configured to calibrate strain gages.

The event case handling the value change of the READ STRAIN button is shown in Figure 15.23. This state enqueues the *Read Strain* state shown in Figure 15.24 and

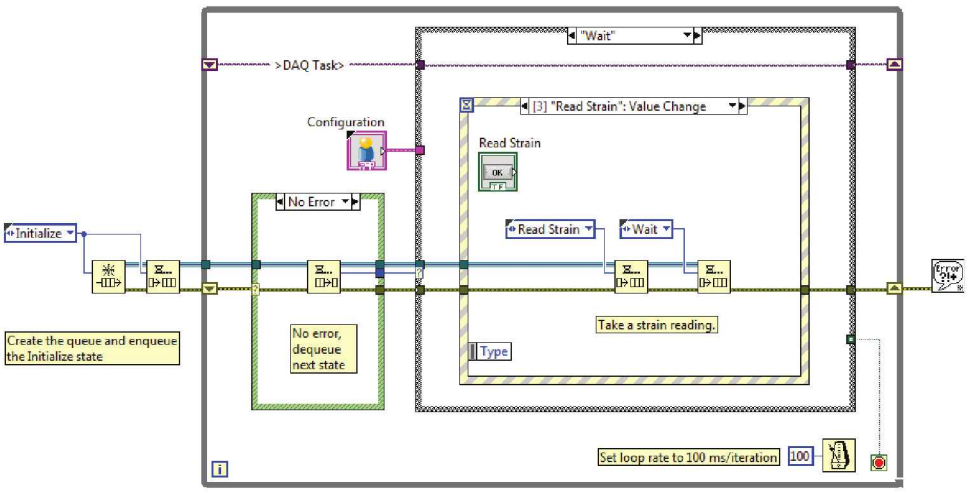


Figure 15.23: The event case handling the value change of the READ STRAIN button of the `Multitest` data acquisition VI

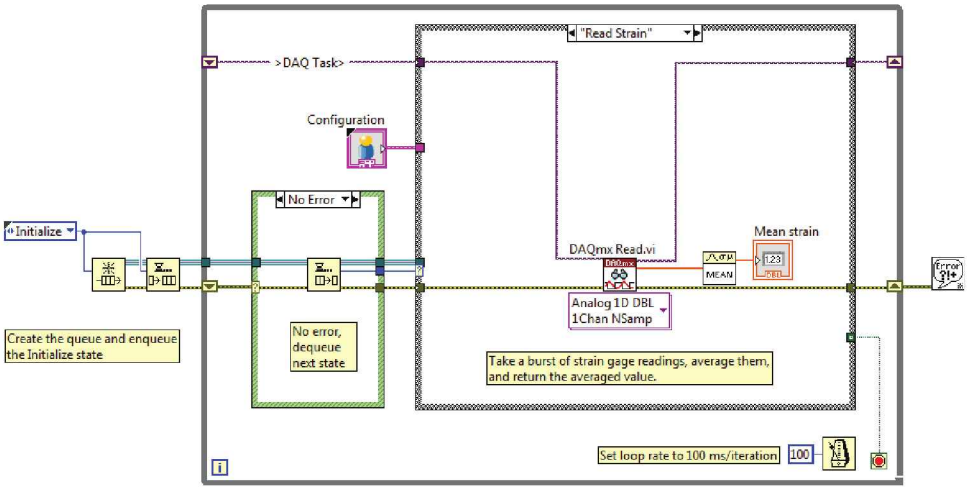


Figure 15.24: The *Read Strain* state of the `Multitest` data acquisition VI

then returns to the *Wait* state. The *Read Strain* state uses the DAQ task reference on the data highway and the configuration cluster to acquire a burst of readings from the strain gage using the **`DAQmx Read`** VI shown in Figure 15.13. This VI is polymorphic; the version shown in Figure 15.24 is configured to read multiple samples from a single DAQ channel. The output data is in double-precision format (DBL). The averaging of the strain data is performed by the **`Mean`** function. This help file is shown in Figure 15.25.

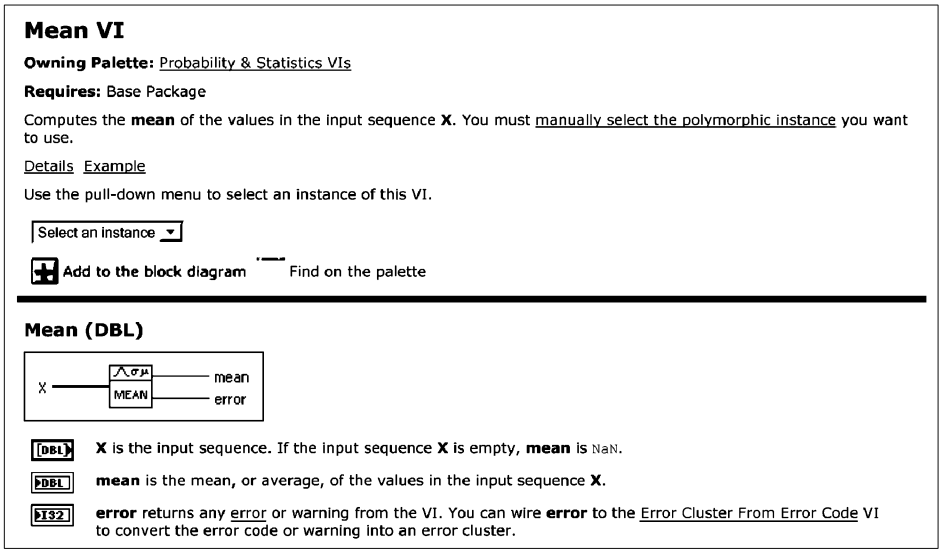
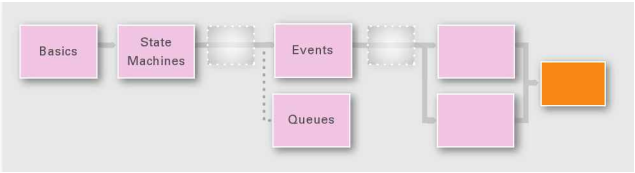


Figure 15.25: Help file for the `Mean` function

The strain measurement VI is now complete. This VI introduced many new functions and LabVIEW concepts related to data acquisition, but the structure of this VI is nearly identical to the versions of the `Multitest` VI developed in Chapter 13.



16

Problem Set 5

16.1 Traffic Light Controller Exercises

Exercise 5.1: Queued *Traffic Light Controller* Program

Would the traffic light controller be a good candidate for queues? Why or why not? ■

16.2 Security System Controller Exercises

Exercise 5.2: Queued Security System

Revise the security system controller to use queues. In the version developed in Exercises 3.8, and 4.2, the log was updated in the same state that detected the value change of the front-panel switches. In this exercise, create a new state called *Update Log* to perform this task. When a switch value change is detected, place the *Update Log* state on the queue. The *Update Log* state will then use the queue to return to the monitoring state.

What are the advantages of creating an *Update Log* state? What are the disadvantages? Consider the structural impact of this decision on the state machine architecture. In your previous implementations of the security system controller, the `Set Status Of All Zones` sub-VI and the `Update Log` sub-VIs were in the same state. This meant that the `Update Log` sub-VI had easy access to both the old zone status array on the data highway and the new zone status array output from the `Set Status Of All Zones` sub-VI. If you move the `Update Log` sub-VI to a different state, it will still be able to access the data highway, but it will no longer have direct access to the new zone status array.

One possible solution to this problem is to place the new status array on the data highway along with the old zone status array. The `Update Log` sub-VI will be able to access both the old and new zone states. Implement this solution in your queued version of the security system controller. ■

Exercise 5.3: Create an `Update Log Action Engine` Sub-VI

In Exercise 5.2, you were asked to store the new zone status array on the data highway so that it could be accessed by the `Update Log` sub-VI. In this exercise, you explore another solution to this problem. Create the `Update Log` action engine with the icon shown in Figure 16.1. This action engine persistently stores the old zone status state. This action engine has two commands: *Set Status* and *Update Log*. The *Set Status* command initializes or resets the stored zone status array. The *Update Log* command causes the action engine to compare the old and new zone status arrays and update the log if any zone has changed status.

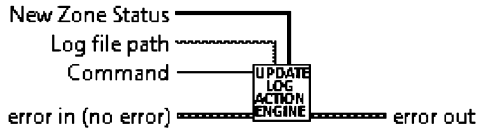


Figure 16.1: The icon of the `Update Log Action Engine` action engine ■

Exercise 5.4: Queued Security System with Action Engine

Implement the action engine you developed in Exercise 5.3 in a revised version of the security system controller you developed in Exercise 5.2. What are the advantages and disadvantages of using an action engine instead of the data highway to store the old zone status array? ■

16.3 Boiler Controller Exercises

Exercise 5.5: Queued *Boiler Controller* Program

The *Boiler Controller* program has many states that are performed in a fixed sequence. This makes it a good candidate for queues. Upgrade the event-driven *Boiler Controller* program to use queues. Put the entire sequence of states on the queue in the *Initialize* state. What are the advantages to using a queue in the boiler controller? Are they worth the extra effort in this case? ■

16.4 Car Wash 2 Exercises

In the following exercises, you will revise the event-driven Car Wash 2 controller developed in Exercise 4.5 to incorporate a queue. This is a major improvement of the Car Wash 2 controller, because it will simplify the way that the car wash option state transitions are handled. In previous implementations of the Car Wash 2 controller, an options array was created and stored on the data highway. An array index variable was also stored on the data highway. Both the array and the array index were used in conjunction with the `Next Car Wash State` sub-VI to determine the next state

transition. The `Next Car Wash State` sub-VI needed to check whether or not the options array was exhausted before determining the next state.

A queue can be used to eliminate most of these features. The option array and index will no longer be needed and will be removed from the data highway. Since the state transitions will be handled by the queue, the `Next Car Wash State` sub-VI will no longer needed. These changes will improve the readability and maintainability of the code.

Exercise 5.6: Create the `Build Wash Array` Sub-VI

Create a new state enumerated constant for the queued Car Wash 2 controller. All of the states are the same as before with the exception of the *Build Option Array* state. Rename this state as the *Queue Wash Sequence* state.

Create the `Build Wash Array` sub-VI with the icon shown in Figure 16.2. This sub-VI is very similar to the `Build Option Array` sub-VI developed in Exercise 1.40. The sub-VI accepts a cluster of wash option Booleans as an input, and it outputs a complete array of car wash states from vehicle entry to vehicle exit. The output array has the form: *Car Entry*, *Wash Option 1*, . . . , *Wash Option n*, and *Car Exit*. The state enumerated constant used in the array is the new version with the *Queue Wash Sequence* state.



Figure 16.2: The icon of the `Build Wash Array` sub-VI ■

Exercise 5.7: Queued Car Wash 2 Controller

Revise the event-driven Car Wash 2 controller developed in Exercise 4.5 to incorporate a queue. Use the new enumerated state constant created in Exercise 5.6. Use the `Build Wash Array` sub-VI also developed in Exercise 5.6 in the *Queue Wash Sequence* state to create the wash sequence of states. Enqueue the states using a **`For`** loop with an auto-indexed input tunnel. ■

16.5 DAQ State Machine Exercises

Exercise 5.8: The `Clear Error Code` Sub-VI

Develop the `Clear Error Code` sub-VI with the icon shown in Figure 16.3. This sub-VI accepts an error cluster and an error code as inputs. The error code input

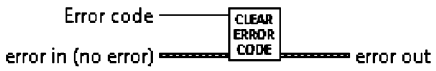


Figure 16.3: The icon of the `Clear Error Code` VI

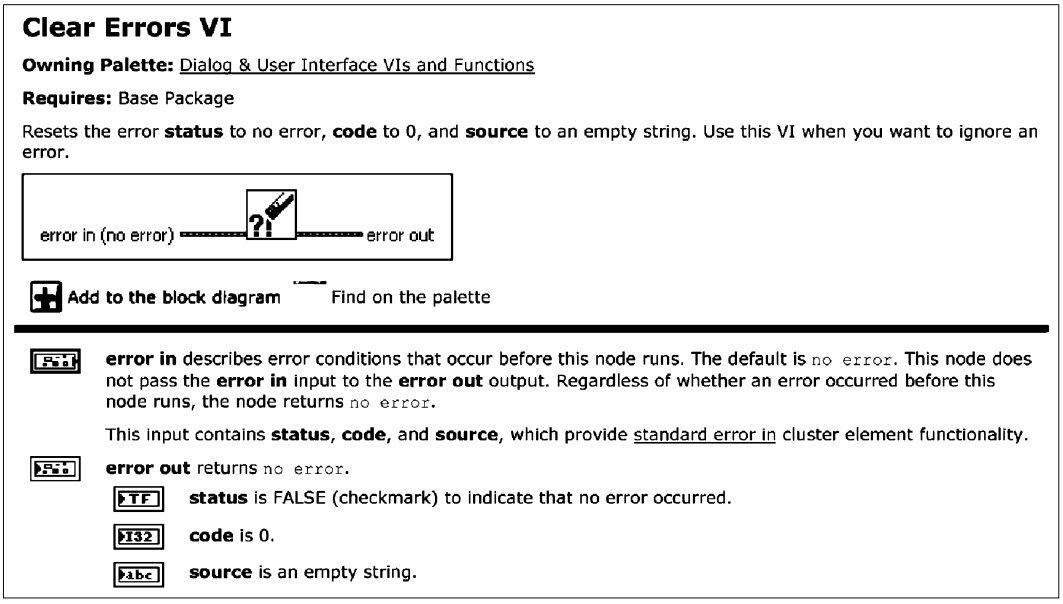


Figure 16.4: Help file for the `Clear Errors` function

should be an `I32` numeric control. This sub-VI will check to see if an error exists. If an error does exist and the error code matches the input error code, this sub-VI will clear the error. Otherwise, the error will be passed to an error cluster output. You might find the **`Clear Errors`** function useful (see the help file in Figure 16.4). ■

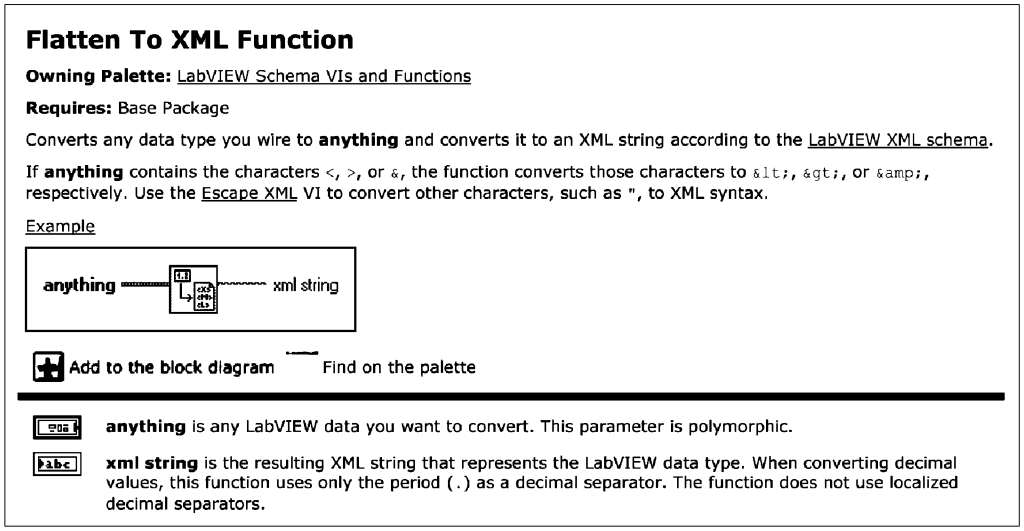


Figure 16.5: Help file for the `Flatten To XML` function

Exercise 5.9: Read/Write Configuration Cluster to an XML File

Modify the strain-gage program developed in Chapter 15 to allow the user to write the contents of the configuration cluster to a human-readable text file. The user should also be able to read a previously generated file and write the contents to the configuration cluster. An easy way to do this is to flatten the configuration cluster to an XML string using the **`Flatten To XML`** function (see the help file in Figure 16.5). The flattened string then can be written to a text file using the **`Write to Text File`** function (see the help file in Figure 1.48).

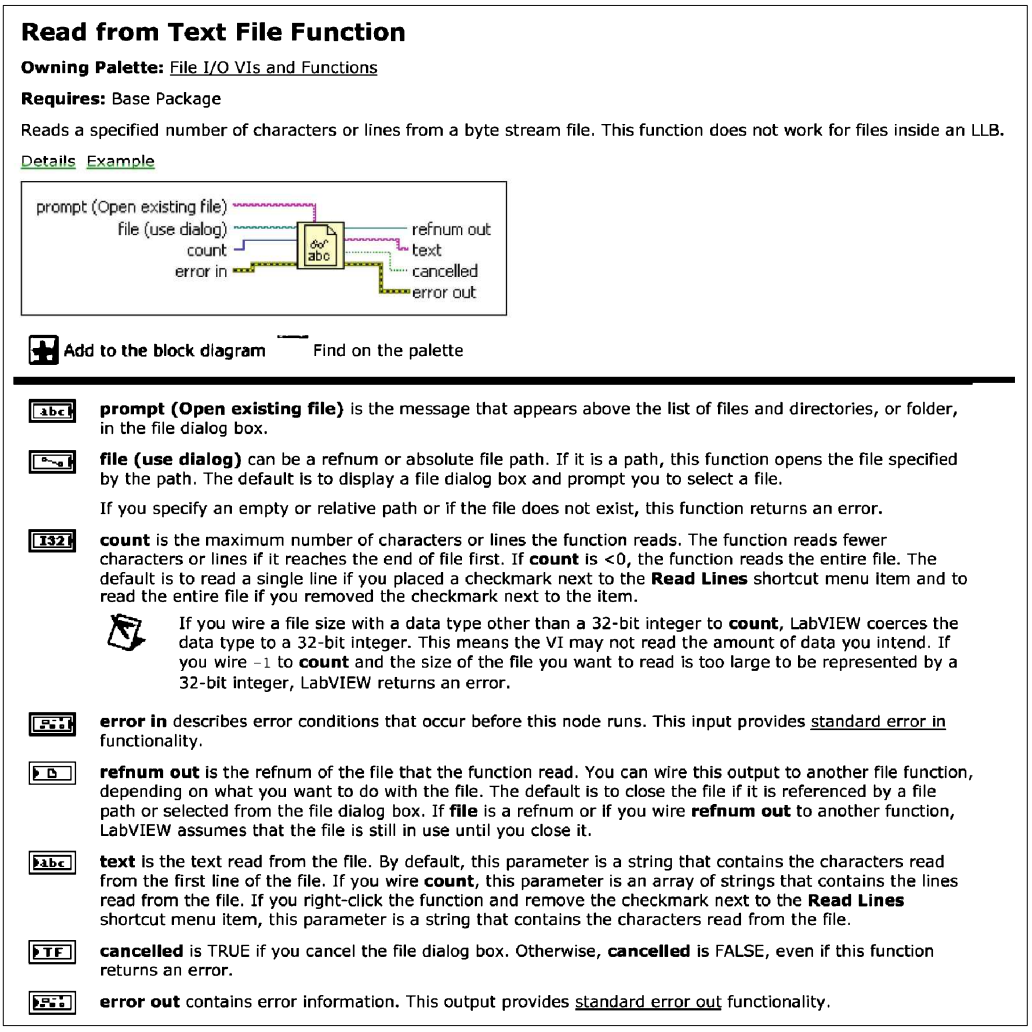


Figure 16.6: Help file for the `Read from Text File` function

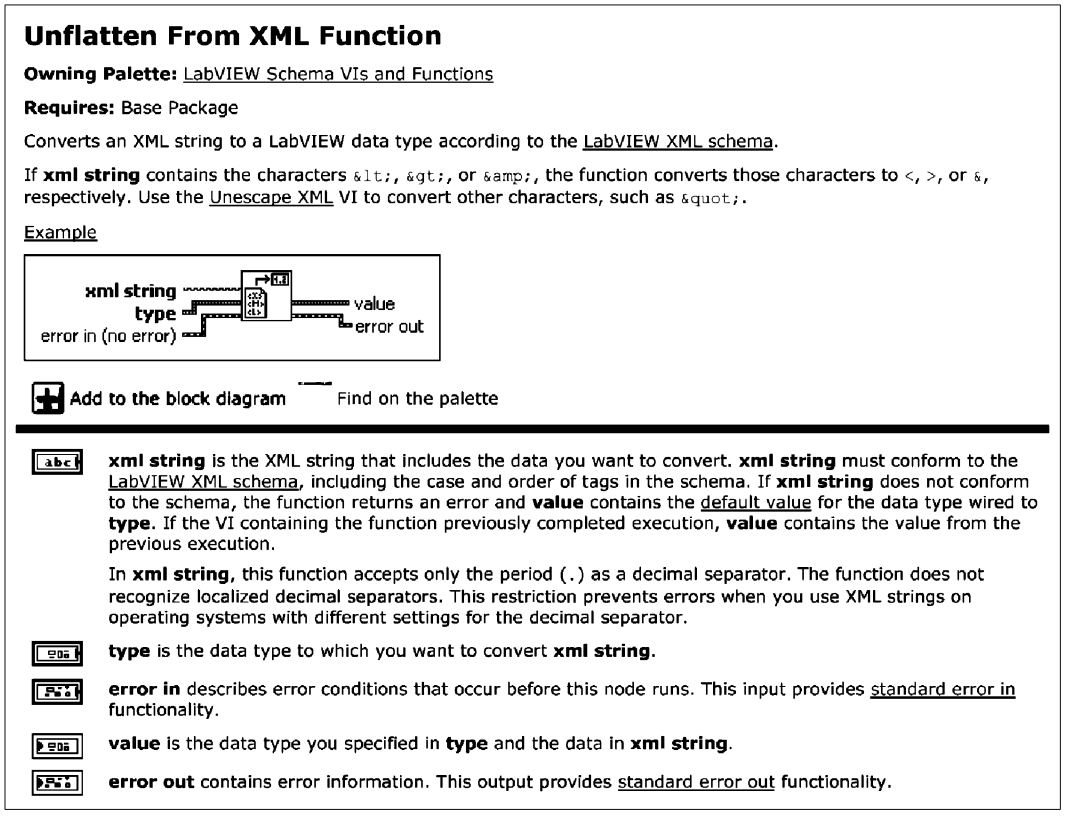


Figure 16.7: Help file for the `Unflatten from XML` function

The XML-formatted configuration data can be read from an existing text file using the **`Read from Text File`** function (see the help file in Figure 16.6). The XML data can be converted back to a cluster using the **`Unflatten from XML`** function (see the help file in Figure 16.7).

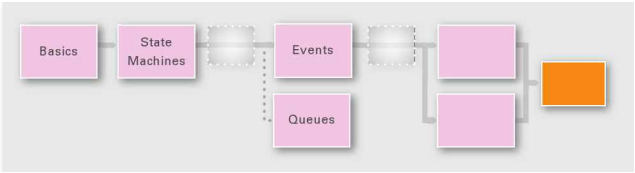
Add two new buttons to the front panel to allow the user to read or write the configuration cluster data to a text file. The program will bring up a file dialog window to allow the user to navigate to the desired file location. If the user cancels the dialog window, error `43` will be generated. Use the `Clear Error Code` sub-VI you developed in Exercise 5.8 to clear the error if it occurs to prevent the state machine from shutting down. ■

Exercise 5.10: Write the Strain Data to an Output File

Modify the strain-gage program developed in Exercise 5.9 to save the strain data to an output file. Place a SAVE DATA button and an output file path control on the front panel of the VI. When the SAVE DATA button is pressed, the VI will append a

comma-separated string to the end of the file designated by the output file control. The specification of the string is as follows:

- The string will have the following format: date, time, sampling rate in Hz, number of samples averaged, and strain.
- Each string should appear on its own line in the data file.
- The strain should be recorded in scientific notation.
- The date and time recorded in the string should be the date and time that the data was taken, not the date and time that the data was recorded to the file.
- The sampling rate and number of samples averaged should be the values that were in use when the data was taken, not the values stored in the configuration cluster when the data was recorded to the file. ■



17 Preparing for the CLD Exam

The Certified LabVIEW Developer (CLD) examination is a four-hour programming exam. The examinee is given a functional specification describing the requirements of a desired LabVIEW program. The examinee is placed in a room with a computer with a standard installation of LabVIEW. The computer has no internet connection. Externally developed code and third-party tools are not allowed to be used in the exam.

The goal of the exam is simple. Starting with nothing but the standard LabVIEW installation, the functional specification, and four hours on the clock, develop a fully functional, well documented application with good programming style that satisfies the requirements in the functional specification.

The test is graded by National Instruments and points are awarded in three categories:

1. Programming style: 15 points
2. Functionality: 15 points
3. Documentation: 10 points

A perfect score is 40 points. The passing grade is 75% or 30 points.

The biggest complaint about the test is that the four hour time limit is too short. Granted, the four hour limit is very tight. Granted, in a real job, you would be given more than four hours to produce a similar code. So what is the purpose of the time limit? One possible reason is that it tends to separate the applicants into two groups: those that have truly internalized LabVIEW "thinking" and style and those that have not.

Let's take a closer look at the point distribution. Only 15 points out of 40 are dedicated to functionality. That's only 37.5% of the total, which is a little more than a third of the total points. About two-thirds of the points are awarded for style and documentation, and style and documentation are not really affected by the four-hour time limit.

Documentation points are the easiest ones to earn. They don't require specialized LabVIEW knowledge, you just have to know what is expected and do it. Documentation requires some time, but not much. You shouldn't spend more than a half an hour

of the four-hour time limit on documentation. Documentation is worth 10 points, so that is a very good return on a small investment of time.

That leaves you about three and half hours to earn a possible 30 points in style and functionality. The style points are key. It is very hard to fake good style. You either have it or you don't. Style is learned through experience. It is gained by following good examples and putting them into practice in your own work. If you haven't seen good examples of LabVIEW code and you haven't incorporated them into your own work, you probably don't have good LabVIEW style, and it will very hard to pass the exam.

But if you have read this book, done the exercises, and applied the lessons to your own code, you probably do have good LabVIEW style. Good programming style has two huge benefits in terms of the CLD exam. First, good style promotes functionality. If you practice the style recommendations and the general layouts of the design patterns in this book, it will be easy to produce functional code that will satisfy the CLD graders. Even better, once good style is internalized, it becomes a self-reinforcing habit. If you have good style going in to the exam, you should expect to get a good style score with very little expended time.

This is the crux of the matter. If you have already internalized good LabVIEW practices and style guidelines and you take care to follow the documentation suggestions in this chapter, you will likely score the majority of the 25 style and documentation points with less than an hour's worth of effort during the test. That leaves you three hours to focus on functionality. If you can then score at least 10 out of 15 points on functionality, you have an excellent chance of passing the test.

This chapter will provide you with tips on how to prepare for and take the CLD exam. The author wishes to point out that these suggestions are his own opinions and do not reflect any official policy, guidelines or position of National Instruments.

17.1 Documentation

The documentation points are yours to lose. You should go in to the exam assuming that you will get full credit in this category. When you are taking the exam, you should be disciplined and document as you go along. Don't succumb to the temptation to jump right into the programming and save the documentation for the last few minutes of the exam.

There are three types of documentation that you should pay attention to during the test:

VI Documentation The main VI and all sub-VIs should be documented in the *VI Properties... Documentation* window.

Front Panel Documentation All front panel controls and indicators should be documented with descriptions and tip strips.

Block Diagram Documentation Block diagrams should have free label documenting function. Try to place a label in every loop and in every case of every case or event structure. Sub-VIs should have a free label documenting their function. Important wires, such as those in the data highway, should also be labeled. Label block diagram constants.

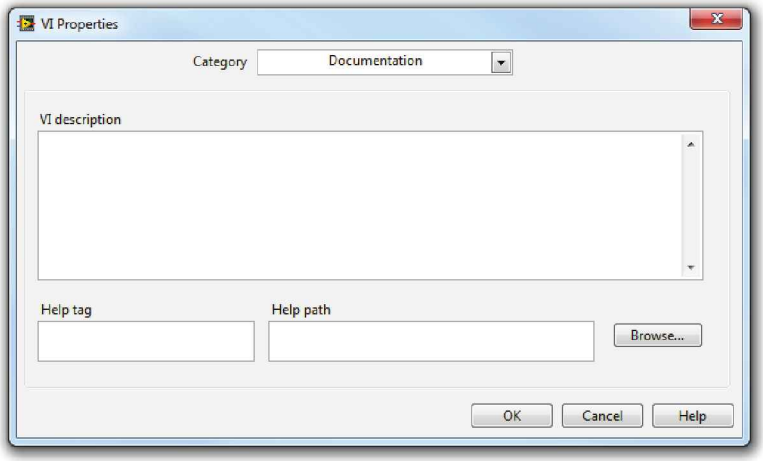


Figure 17.1: Documenting a VI using a VI description

Every time you create a new VI, including the main VI, immediately document it by right-clicking on the icon and selecting *VI Properties*. Select the *Documentation* category and fill in the *VI Description* field, as shown in Figure 17.1. The text entered in this window will be displayed if the user hovers the mouse over the VI icon while the context help window is open. To save time during the test, select the text and copy it (`Control-C`). Then create a free label on the block diagram and paste the text into the label (`Control-V`). This documents the code in two different places with the same description.

Front panel objects need to be documented with descriptions and tip strips. The description text shows up in the context help window when the user hovers the mouse over the object. Tip strips show up in small pop-up windows when the user hovers the mouse over the object.

To create descriptions and tips, right-click on a front panel object and select `Description and Tip...`. This brings up a window with text fields for the description and tip, as shown in Figure 17.2. Again, feel free to cut and paste between the two fields to save time during the test.

If a front panel object is a cluster, there are two levels of descriptions and tips. The cluster itself can have a description and tip, and each individual element of the cluster can have a description and tip. During the exam, you should document both the overall cluster and the individual elements with descriptions and tip strips.

Sub-VI front panel objects also should be documented with descriptions and tip strips. To save time during the exam, use strict type definitions for all front panel controls and indicators and document the type-definition file with a description and tip strip. Then when you use these type-defined controls and indicators in your sub-VIs, they will automatically carry over the descriptions and tip strips from the type-definition file.

Block diagrams should be documented with free labels. Sub-VIs should have a label describing their functionality. Complicated sequences of code should have labels

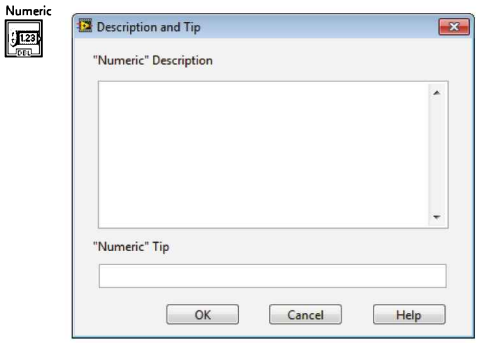


Figure 17.2: Documenting front panel objects with descriptions and tip strips

describing what the code is doing. As a rule of thumb, try to include a label in each loop and in each case of every case or event structure. In the main case structure of a state machine, each state should have a label describing what the state does.

Certain block diagram items should also be labeled. Long wires should be labeled using the `>>Label Text>>` convention. All wires on the data highway also should be labeled in this fashion. Block diagram constants also should be labeled.

Block diagram labels also can be used to communicate with the exam grader. If the functional specification is not completely clear, list your assumptions in the appropriate places in your code using free labels. If you are running out of time and your code is not fully functional, use free labels to describe what you would do to finish if you had more time.

The key to getting full credit in the documentation category is to do the documentation as you go. Don't wait until the end of the test to document your code. When you create a new VI, fill in the VI documentation. When you create a type definition or drop a new front panel control or indicator, fill in the description and tip strip. When you create a loop, put in a label. When you create a case structure, document each case as it is created.

17.2 Style

Good style is easier to demonstrate than describe, especially in a graphical language like LabVIEW. You learn it by seeing it and by practicing it yourself. The hard part is finding programs written with good style to study and emulate, especially if you are a lone-wolf LabVIEW programmer.

This was the motivation for writing this book. This book provides you with LabVIEW codes written with good style in the hope that you will emulate them. So if you want to score well in the style category of the CLD exam, try to make your code look like the codes in this book.

Chapter 1 closed with a list of style guidelines followed in this book. The list is repeated here, but modified slightly to focus more tightly on code written for the

CLD exam. Try to absorb these points in your own programming style before you take the test. If you have to think about them during the test, you will waste valuable time.

- The program should occupy one screen with no scrolling.
- Avoid sequence structures, use state machines instead.
- Minimize the use of local variables. Eliminate them if possible.
- Do not use global variables.
- Keep wires as straight as possible.
- Align functions on the block diagram to improve readability.
- Make every effort to improve readability. For example, use **`Unbundle By Name`** instead of **`Unbundle`**.
- Every loop should have at least one free label.
- Every case of every case or event structure should have at least one free label.
- Sub-VIs should use the error-handling template if possible.
- Every sub-VI block diagram should have at least one free label documenting the sub-VI function.
- All sub-VIs should use the $4 \times 2 \times 2 \times 4$ connector pane if possible.
- Connector panes should be wired with inputs on the left and outputs on the right.
- Error wires should use the lower corner terminals of the connector pane. References should use the upper corners of the connector pane.
- Every sub-VI should have a descriptive custom icon.
- The error chain should be tight. If a function has error terminals, they should be wired. All errors wires should eventually merge with the main error wire.
- Create type definitions for clusters of front panel controls or indicators.

17.3 Functionality

Functionality is key, but you don't need a fully functional VI to pass the CLD exam. In fact, if you score full marks for documentation and style you only need 5 points out of 15 in the functionality category to pass the test. That being said, of course you would like to turn in a fully functional program. Here are some suggestions that will help you do that in the four-hour time frame of the test.

This book contains the functional specifications of six past CLD exams. This should give you a good idea of the scope of the exam. Be warned, though, that the traffic light and security system exams are older than the others and are likely to be simpler than the tests offered today.

All of these exams can done using the single loop classic state machine architecture described in Chapter 4. Know this design pattern well. Do not take the exam if this design pattern is not in your muscle memory. You should be able to set up the basic structure of a state machine in minutes and with little thought.

You also should be familiar with event structures and queues. These also should be in your muscle memory before you take the test. Queues and event structures may or may not be helpful in a given test, but you should definitely have these tools at your disposal.

Another area to review is the placement of constants, controls, and indicators in a state machine. Review Chapter 4 and pay special attention to the discussion of how persistence and access affect where an item should be placed in the state machine. Ask yourself the following questions about each constant, control, and indicator:

- Should this item go in an individual state?
- Should it be in the lobby or the alley of the state machine?
- Is this element volatile, changing every loop iteration?
- Is this item needed by more than one state?
- Do this item need to be stored from one loop iteration to the next?
- Should this item be on a data highway? Do I really need a data highway?

Of the six previous CLD exams provided in this book, four require a timer of some sort. Timers are typically implemented as action engines. The basic action-engine template also should be in your muscle memory before taking the test. Different storage formats were presented in this book (single loop, **`For`** and **`While`** loops, and feedback nodes). Choose your favorite and practice it before taking the test. Practice by building a basic timer action engine with *Elapsed Time* and *Reset* functionality until you can build it with little thought. This will save time during the test. Keep in mind that timers need to be reset. If you are testing your code and the timing is not working, make sure that the timer is reset before you first call it and that it is reset before each new timed task.

Error handling is an easy thing to overlook during a test. Make sure that your VI has a tight error chain. Wire every error terminal of every function that you use. Merge the errors where necessary. Put critical functions on parallel error chains and then merge them back into the main line. Build your sub-VIs with error inputs and outputs, and use the error-handling template described in Chapter 1 whenever possible.

Organization is also important. Create a LabVIEW Project for your exam entry. Organize the project to make the file structure clear.

17.4 Preparing for the CLD Exam

Remember that the CLD is a performance exam. It doesn't test what you know; it tests what you can do. So you can't really study for it in the traditional sense. You need to practice for it instead.

This book provides you with more than enough material to study. Know the material about classic state machines, event-driven state machines, queued state machines, and action engines.

Practice by building the codes shown in this book. Build the various versions of the `Multitest` VI as a start. The ultimate test, of course, is to time yourself and

attempt to work through the sample CLD exams presented here. Then compare your solutions with the ones given in this book. When you are confident that you can do the sample exams in the four-hour time limit, you are ready to take the CLD exam.

17.5 Time Management

Time management is key to passing the exam. Here are some general guidelines to consider on how best to spend your time during the CLD exam.

1. The first hour or hour and a half should be spent in planning and preparation. Resist the urge to start programming immediately. Read the functional specification thoroughly. Make sure that you understand what the program needs to do. Are there any fuzzy areas in the specification? If so, reread the specification more carefully. If it is still unclear, make any needed assumptions and document them.
2. Decide which design pattern would be best suited for solving the problem. Ask yourself the following questions.
   - Does the state machine need to wait? Would an event structure be helpful?
   - Would it be helpful to be able to queue a series of states instead of just one at a time? Are queues worth the extra effort?
   - Does any data need to be stored from one loop iteration to the next? Do I need a data highway?
   - Where are the best places to put the various controls and indicators?
   - Would an action engine be helpful? If so, what data needs to be stored and what functions should it perform?
   - What states are needed? What are the state transitions?
3. Take the time to draw a state transition diagram, it will help you visualize the state transitions and may lead you to revise your list of needed states.
4. Once the planning is done, start the preparation phase:
   - Create the LabVIEW project.
   - Open the main VI and save it. Save your work often.
   - Fill in the VI documentation.
   - Create the front panel. Make it look like the example in the specification. Make sure that data types are correct. Make sure that the data representations (DBL, I32, etc.) are correct.
   - Create type definitions for important clusters of controls and indicators and for any enumerated controls. Fill in the description and tip strips in the type definition files.
   - Create the basic structure of your chosen state machine in the block diagram. Add free labels as you do so. Put the controls and indicators in their approximate locations.
5. At this point you are ready to do the real work. Plan on spending two hours or so following your plan and completing the state machine. Document as you go.

Decide ahead of time whether you are a bottom-up or a top-down programmer. If you prefer bottom-up, spend time developing any needed sub-VIs or action engines and then place them in the appropriate places in your state machine.

If you are a top-down programmer, create stubs for the sub-VIs and action engines. A stub is a sub-VI with a custom icon and a wired connector pane and front panel—but with no working code on the block diagram. Stubs are useful because you can drop them into the main program and wire them properly. When the main VI is at or near completion, develop the block diagrams of the stubs without changing the connector pane.

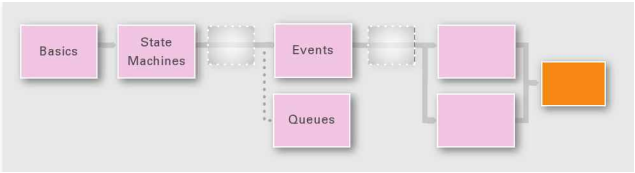
If you find yourself approaching the last half hour and the code is not fully functional, stop and document with free labels what you would do if you had more time. If you have stubs that have not been developed fully, place labels on their block diagrams describing how you have developed them if you had more time.

6. The last half hour should be dedicated to a final review. Remember, documentation and style points are yours to lose! Review every loop and every case of every case structure. Is everything documented? Are all front panel controls and indicators documented with descriptions and tip strips? Are all sub-VIs documented with VI documentation and free labels? Do all sub-VIs have custom icons? Is everything laid out cleanly and in good style? Is there a label in every case of every case and event structure?

   After you have confirmed that you have done all you could to fully earn the documentation and style points, return to functionality if you have time. Finish what you can, test what you have finished, and document the rest with what you would have done if you had more time.

17.6 Practicing for the Exam

The key to passing the CLD exam is to internalize good LabVIEW style, and the way to internalize good style in any programming language is to practice, practice, practice. The best way to do this is to work out the sample practice exams included in this book. Two of the published sample exams have not been used in the exercises presented thus far: the sprinkler controller exam and the ATM controller exam. This was done to give you the opportunity to test yourself with these exams. The best way to do this is to time yourself and see if you can complete each test in four hours. If you get stuck or you are not preparing for the CLD exam, you can work through these exams in a guided fashion by completing the exercises in Chapters 18 and 19. Chapter 18 covers the sprinkler controller exam, and Chapter 19 covers the ATM controller exam. You can also practice for the exam by working through the other exams presented in this book and imposing the four hour time limit.



18

Problem Set 6

This problem set will guide you through a solution of the sprinkler controller sample Certified LabVIEW Developer exam. If you are preparing for the CLD exam, try to complete the sprinkler controller in four hours without reading the exercises in this problem set.

Exercise 6.1: Sprinkler Controller State Transition Diagram

Read the functional specification for the sprinkler controller in Chapter E. What states will the program need? How do they transition from one to another? Draw a state transition diagram for a sprinkler controller. Pay attention to how the STOP and SETUP buttons are handled in the diagram. Add notes to the diagram to explain any actions that are difficult to capture in the diagram itself. Use the diagram to create a list of states for the sprinkler controller. ■

Exercise 6.2: Select an Architecture

Select a state machine architecture for the sprinkler controller. Would an event structure be useful? Would queues be useful? After you have selected an architecture, consider the placement of the controls and indicators. Where should the CONTROLS cluster, the ZONE SETUP ARRAY control, and the SIMULATION CONTROLS control be placed? Are these controls needed by multiple states? Do they change often? Do their values need to be stored? Consider the indicators. Where should the ZONE INDICATORS cluster and the ZONE STATUS cluster be placed? Do they need to be updated by multiple states? Do they need to be updated frequently? Consider persistent storage. Does the sprinkler controller need a data highway? If so, what should be stored on the data highway?

Consider the guard clause. Can it be used to monitor STOP and SETUP buttons? What states should the guard clause have? ■

Exercise 6.3: Sprinkler Controller Type Definitions

The sprinkler controller uses several type definitions. First, create an enumerated constant containing the list of states generated in Exercise 6.1. If you have decided to use the guard clause to monitor the front panel activity, create an enumerated constant of states present in the guard clause.

The front panel has many important clusters that should be saved as type definitions. Create type-defined control files for CONTROLS cluster, the SIMULATION CONTROLS cluster, and the ZONE INDICATORS cluster. Consider also the ZONE SETUP ARRAY control. The ZONE SELECTOR combo box used in the array should be saved as a type definition. There is no need to make a type definition of the entire array, but the array element consisting of the ZONE SELECTOR combo box and the TIMING numerical control also should be saved as a type definition. ■

Exercise 6.4: Create a `Setup Stop or Error` Sub-VI for the Guard Clause

Create the `Setup Stop or Error` sub-VI with the icon shown in Figure 18.1. The STATUS output of this sub-VI is an enumerated constant wired to the selector terminal of the guard clause. This sub-VI will monitor the STOP button and the SETUP button, as well as check for errors. The guard clause case structure has three states: *Stop or Error*, *Setup*, and *Continue*. If the STOP button is pressed or if an error is detected, the guard clause will execute the *Stop or Error* state. If the SETUP button is pressed, the guard clause will execute the *Setup* state. If none of these events have occurred, the guard clause will execute the *Continue* state. This sub-VI should operate normally even if an error is present at the error input terminal.

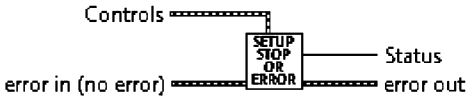


Figure 18.1: The icon of the `Setup Stop or Error` VI ■

Exercise 6.5: Create a Sub-VI to Read the Zone Array File

The initial values of the ZONE ARRAY control are stored in a comma-separated value (csv) text file. Create the `Initialize Zone Array` sub-VI with the icon shown in Figure 18.2. This sub-VI reads the text file at the given path, converts the data from strings to the needed data type, and writes the data to the ZONE ARRAY control.



Figure 18.2: The icon of the `Initialize Zone Array` VI

This sub-VI should be constructed without making assumptions about the number of zones contained in the input file.

You might find the **`Read From Spreadsheet File`** function useful for reading the comma-separated value text file (see the help file in Figure 18.3). Once the values have been read from the file, they need to be converted into the correct data types. You might find the **`Fract/Exp String To Number`** function useful (see the help file in Figure 18.4). You might also find the **`Type Cast`** function helpful (see the help file in Figure 3.38).

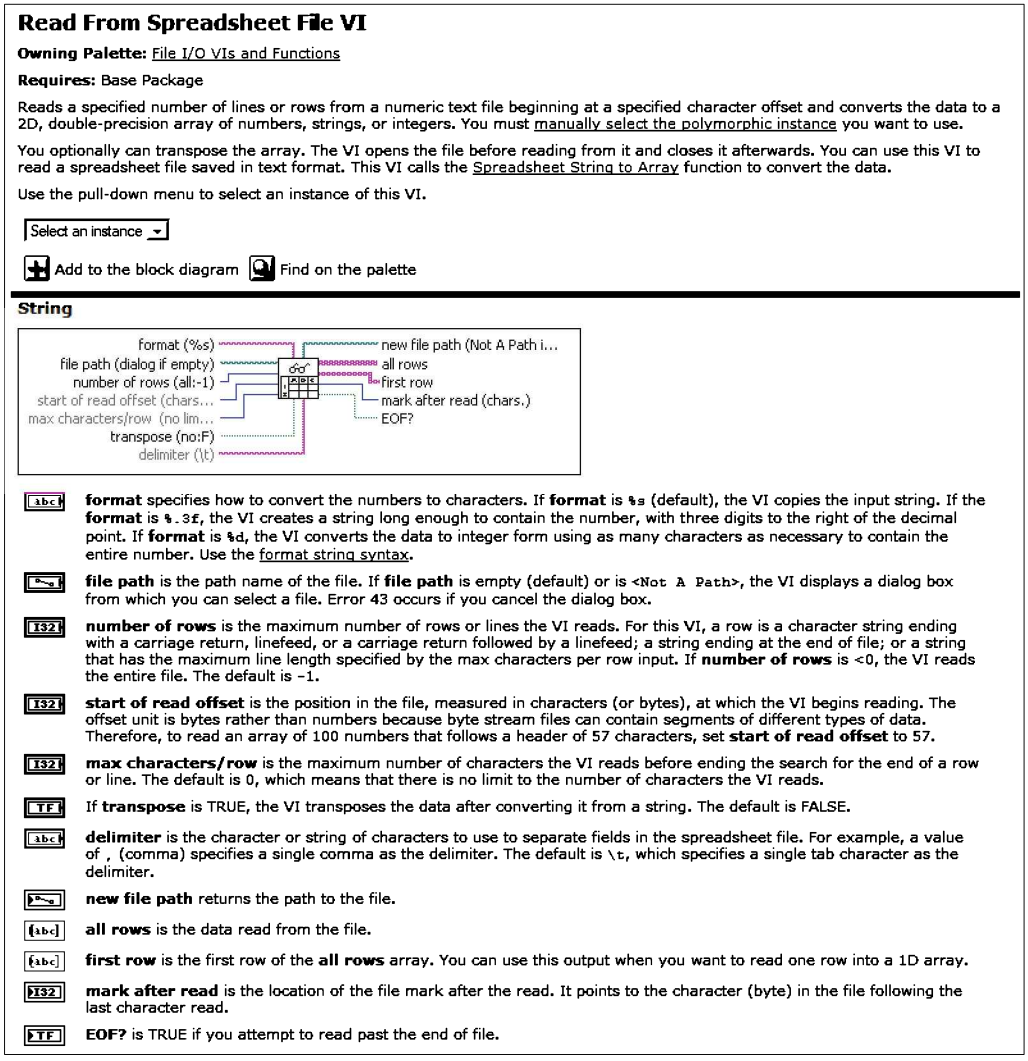


Figure 18.3: Help file for the `Read From Spreadsheet File` function

Fract/Exp String To Number Function

Owning Palette: String/Number Conversion Functions

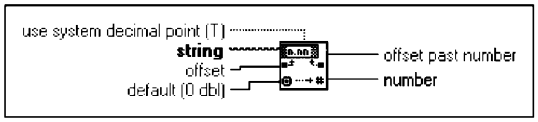
Requires: Base Package

Interprets the characters 0 through 9, plus, minus, e, E, and the decimal point (usually period) in **string** starting at **offset** as a floating-point number in engineering notation, exponential, or fractional format and returns it in **number**.

The connector pane displays the default data types for this polymorphic function.

If you want the function to return a 64-bit integer output, you must wire a 64-bit integer to the **default** input.

Example



Add to the block diagram Find on the palette

use system decimal point defines the decimal separator. If TRUE (default), the decimal separator uses the localized decimal separator. If FALSE, the decimal separator is a period.

string can be a string, a cluster of strings, an array of strings, or an array of clusters of strings. If **string** contains the characters `Inf` or `NaN`, this function returns the LabVIEW values `Inf` and `NaN`, respectively.

offset is the starting position and must be numeric. The offset of the first character in **string** is 0. If unwired or if less than 0, the default is 0.

default is any object of numeric representation and specifies the numeric representation for **number**. The default is an extended-precision, floating-point value of 0.

offset past number is the index in **string** of the first character following the number. **offset past number** reflects the value from the last string if you input an array of strings.

number can be a number, a cluster, an array of numbers, or an array of clusters, depending on the structure of **string**. The following table shows how the values of **string**, **offset**, and **default** affect **number**.

string	offset	default	offset past number	number	Comments
-4.7e-3x	0	0	7	-0.0047	x is not allowed, so conversion stops there.
+5.3.2	0	0	4	5.3	Second decimal point not allowed, so conversion stops there.

If you scan a string that does not fit into the numeric data type you specify, this function returns the largest number that fits into that data type.

Figure 18.4: Help file for the `Fract/Exp String To Number` function ■

Exercise 6.6: Create the `Next Sprinkler State` Sub-VI

One of the challenges in the sprinkler controller is determining the state transition from one state to the next. The sequence of zones is specified in the ZONE SETUP ARRAY, but the normal progression of zones can be interrupted at any time by rain or low water pressure. Some state transitions depend on whether the sprinkler is in single or continuous mode. The value of the START button can also influence the determination of the next state. The next state can also depend on the controller's current state.

Create the `Next Sprinkler State` sub-VI with the icon shown in Figure 18.5. This sub-VI accepts the SIMULATION CONTROLS, the current state, and an ENDED? Boolean as inputs, and it returns the next state as an output. The ENDED? Boolean is True if the controller is in single-program mode and the current program has ended and is False otherwise.

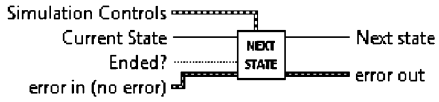


Figure 18.5: The icon of the `Next Sprinkler State` VI ■

In the next few exercises, you develop a series of sub-VIs that are used to service a sprinkler zone. These sub-VIs are collected into a single `Service Zone` sub-VI for use by the *Service Zone* state. The `Service Zone` sub-VI is responsible for running the zone time check, for creating the output cluster for display by the main application, and for incrementing the index of the ZONE SETUP ARRAY. The readability of the main application is increased by putting all of these related functions into a single cohesive sub-VI.

Exercise 6.7: Create a Sprinkler Timer

The sprinkler states are timed, so a timer needs to be developed. Consider the timers developed for the other projects in this book. The sprinkler timer is much like the traffic light timer. The sprinkler timer runs continuously when the zones are being serviced. The timer never needs to be paused, and there is no break between zones. As soon as one zone ends, the next zone begins. This suggests that an auto-reset timer (such as the one developed for the traffic light controller) could be used for the sprinkler controller. The only change that needs to be made to the traffic light timer is the addition of an elapsed time output.

Create the `Auto Reset Timer With Elapsed Time` action engine with the icon shown in Figure 18.6. Use the auto-reset timer developed in Exercise 2.9, but add an output for the elapsed time.

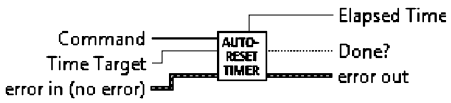


Figure 18.6: The icon of the `Auto Reset Timer With Elapsed Time` action engine ■

Exercise 6.8: Create a Mapping Sub-VI for the `Zone Indicators` Cluster

The sprinkler controller uses a ZONE SETUP ARRAY to manage the zone state transitions. One way to do this is to have a generic *Service Zone* state that services a given zone in

the array. Since this state can service any zone, a mapping sub-VI needs to be created to format the zone output based on the value of the ZONE SELECTOR. Create the `Light Zone Lamp` sub-VI with the icon shown in Figure 18.7 to perform this function. This sub-VI takes a TIMING input, a ZONE SELECTOR input, and a RUNNING? Boolean input and produces a correctly formatted ZONE INDICATORS cluster output.

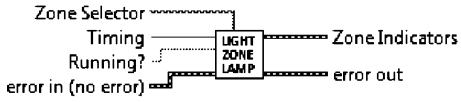


Figure 18.7: The icon of the `Light Zone Lamp` VI ■

Exercise 6.9: Create a Sub-VI to Increment the Zone Array Index

The sprinkler controller is similar to the Car Wash 2 controller. Both controllers use arrays to determine some of the state transitions, and both controllers need sub-VIs to handle the arrays. Both controllers also need to store an index value on the data highway to keep track of the current position in the array. Create an `Increment Index` sub-VI with the icon shown in Figure 18.8. This sub-VI increments the index if the current sprinkler zone is done. If the index exceeds the ZONE SETUP ARRAY size, the index is set to zero. The `Increment Index` sub-VI also outputs a SINGLE MODE DONE? Boolean indicator. If the new index exceeds the ZONE SETUP ARRAY size and the RUN SELECTOR switch is in single-program mode, this indicator is True; otherwise, it is False. You might find the **`Quotient & Remainder`** function useful (see the help file in Figure 18.9).

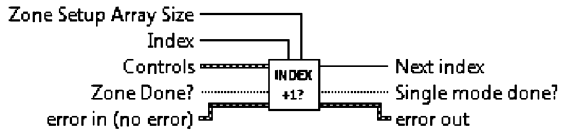


Figure 18.8: The icon of the `Increment Index` VI ■

Exercise 6.10: Create the `Service Zone` Sub-VI

We can now create the `Service Zone` sub-VI with the icon shown in Figure 18.10. This sub-VI selects the current zone from the ZONE SETUP ARRAY and performs the necessary time check using the sprinkler timer developed in Exercise 6.7. The output cluster for display by the main application is generated using the `Light Zone Lamp` sub-VI developed in Exercise 6.8. Remember to round the elapsed time to the nearest second as required by the functional specification. The index is then incremented (if needed) using the `Increment Index` sub-VI developed in Exercise 6.9.

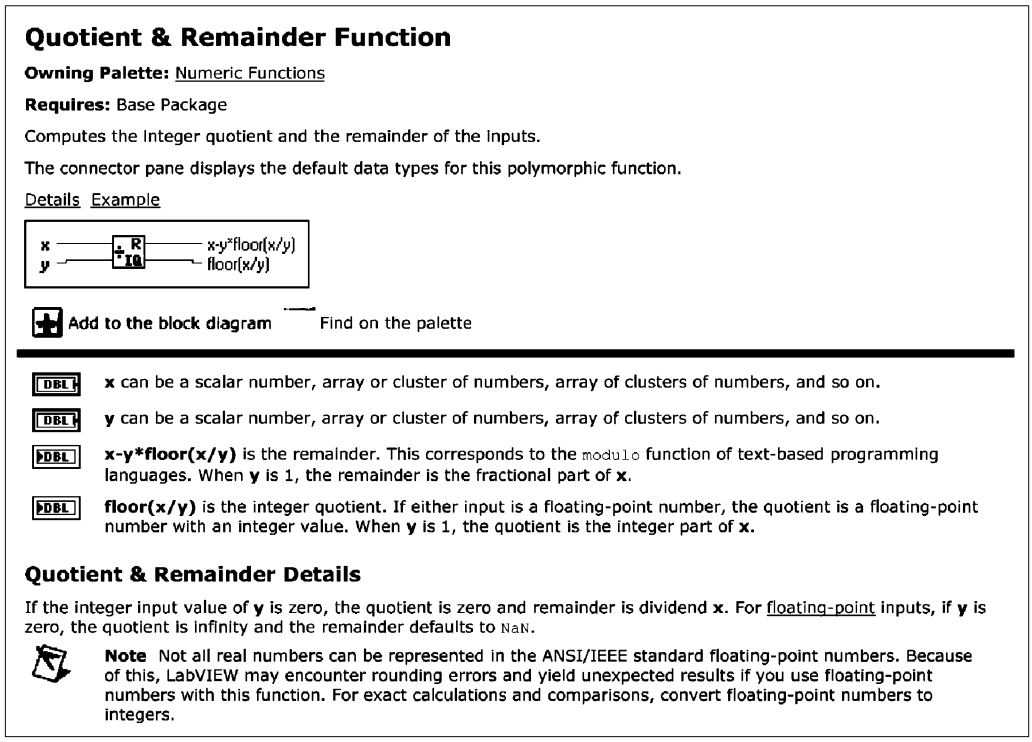


Figure 18.9: Help file for the `Quotient & Remainder` function

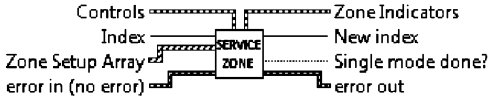


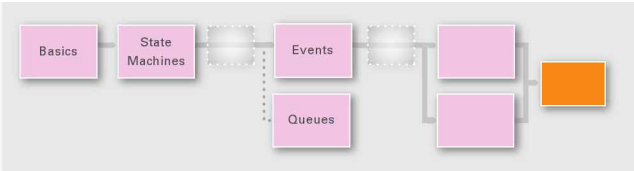
Figure 18.10: The icon of the `Service Zone` VI ■

Exercise 6.11: Create the Sprinkler Controller Main Application

Create the sprinkler controller main application using the state transition diagram, type definitions, and sub-VIs developed in the previous exercises of this problem set. ■

Exercise 6.12: Queues?

In Exercise 5.7, you were asked to convert the array-based version of the Car Wash 2 controller to a queue-based version. Using a queue simplified the main application by removing the array and index from the data highway and eliminating the need for the `Next Car Wash State` sub-VI. Would that approach be useful for the sprinkler controller? Why or why not? ■



19

Problem Set 7

This problem set guides you through a solution of the ATM controller sample Certified LabVIEW Developer exam. If you are preparing for the CLD exam, try to complete the ATM controller in four hours without reading the exercises in this problem set.

The ATM controller is a challenging exercise for a number of reasons. One of the challenges involves the personalization of the ATM messages. We need to develop a scheme for personalizing the messages using information available in the user's account. To do this, we develop a key-value parsing scheme. Message templates contain keys that act as placeholders for account information. The messages are parsed to identify the keys and replace them with the relevant account information.

Another challenge involves the ENTER button. The ENTER button is modal; the response to its value change depends on what mode it is in. There are four modes to consider:

Verify Account If the user presses ENTER after keying in an account number, the controller must verify the account.

Deposit If the user presses ENTER after keying in a deposit amount, the controller must make a deposit.

Withdraw If the user presses ENTER after keying in a withdrawal request, the controller must verify that the account has sufficient funds and then make the withdrawal.

Idle If the user presses ENTER at any other time, the controller must ignore the button press and take no action.

Read the functional specification in Appendix F carefully and keeping these challenges in mind. Then proceed with the following exercises.

Exercise 7.1: ATM Controller State Transition Diagram

Read the functional specification for the ATM controller in Appendix F. What states will the program need? How do they transition from one to another? Draw a state transition diagram for ATM controller. Pay attention to how the *Wait* state is handled

before and after a card is inserted. Add notes to the diagram to explain any actions that are difficult to capture in the diagram itself. Use the diagram to create a list of states for the ATM controller. ■

Exercise 7.2: Select an Architecture

Select a state machine architecture for the ATM controller. Would an event structure be useful? Would queues be useful? After you have selected an architecture consider the placement of the controls and indicators. Where should the CARD SIMULATOR button, the USER INPUT field, and the ENTER button be placed? Where should the left and right menu buttons be placed? Are these controls needed by multiple states? Do they change often? Do their values need to be stored?

Consider the indicators. Where should the ATM MESSAGE indicator and the left and right menu indicators be placed? Do they need to be updated by multiple states? Do they need to be updated frequently?

Consider persistent storage. Does the ATM controller need a data highway? If so, what should be stored on the data highway?

Consider the guard clause. Does it need to monitor anything other than the error status of the main error wire? ■

Exercise 7.3: ATM Controller Type Definitions

The ATM controller uses several type definitions. First, create an enumerated constant containing the list of states generated in Exercise 7.1. If you have decided to use the guard clause to monitor front panel activity, create an enumerated constant of states present in the guard clause.

The front panel has important clusters that should be saved as type definitions. Create a type-defined control containing three ATM buttons. This control is used for both the left and right ATM buttons. Create a type-defined control for containing three string indicators. This control is used for both the left and right side menu displays.

The ATM controller must personalize many different message types for each account holder, such as the *Welcome* message and the *Session Terminate* message. Create an enumerated constant that contains the names of all messages delivered by the ATM controller.

The ATM controller must deal with accounts. Accounts contain all necessary information about the ATM user. Account information is stored in a text file. Since account information is important and widely used in the controller, create a type-defined cluster that contains string controls for the data stored in the account: *First Name*, *Last Name*, *Balance*, and *Amount*. The *Amount* field is not stored in the text file. It is used to temporarily store deposit and withdrawal information.

The ENTER button of the ATM controller is modal; its function depends on the operation that is taking place when it is pressed. The ENTER button has four modes: *Idle*, *Deposit*, *Withdraw*, and *Verify Account*. Create a type-defined enumerated constant containing these modes. ■

One of the challenges of the functional specification is parsing the ATM messages. The ATM messages need to be personalized with the following information: first name

of the account holder, last name of the account holder, the current account balance, and the amount of the current transaction. This can be done using a key-value system. The keys will be pre-defined placeholders in the ATM message templates that will be replaced by the corresponding values from the account and transaction history. The resulting message will be displayed on the ATM.

To do this, we need to define a convention for identifying the keys in a given message. We use the convention where keys are enclosed by square brackets "[]". For example, the template of the main menu message would read: "Welcome to Acme Bank [First Name] [Last Name]. Please select transaction by using the buttons." The balance inquiry message template would read "Your Balance Is: $[Balance]." In these examples, the bracketed key values would be replaced by the proper account information before the message was displayed. There are four keys in the set of ATM messages: [First Name], [Last Name], [Balance], and [Amount].

Exercise 7.4: Create the `Get Message Templates` Sub-VI

Create the `Get Message Templates` sub-VI with the icon shown in Figure 19.1. The message templates are retrieved by this sub-VI. This sub-VI accepts as an input the enumerated constant listing the message types created in Exercise 7.3. The output of the sub-VI is the specified message template containing keys in square brackets.



Figure 19.1: The icon of the `Get Message Templates` VI ■

Exercise 7.5: Create the `Replace Keys With Values` Sub-VI

Create the `Replace Keys With Values` sub-VI with the icon shown in Figure 19.2. This sub-VI accepts a message template with keys contained in square brackets as an input. It also accepts an array of bracketed keys and a corresponding array of values as inputs. This sub-VI will iterate through the array of keys, searching the message template for all instances of each key and replacing them with the corresponding value. You might find the **`Search and Replace String`** function useful (see the help file in Figure 19.3).

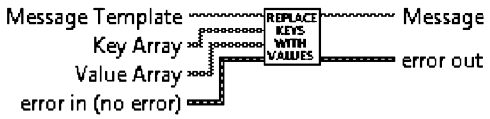


Figure 19.2: The icon of the `Replace Keys With Values` VI

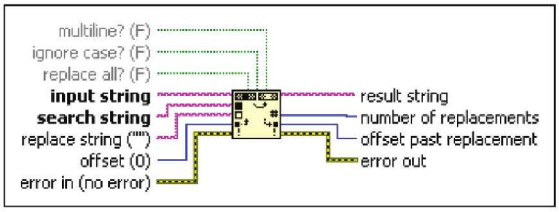
Search and Replace String Function

Owning Palette: String Functions

Requires: Base Package

Replaces one or all instances of a substring with another substring. To include the **multiline?** Boolean input, right-click the function and select **Regular Expression**.

Details Example



Add to the block diagram Find on the palette

multiline? sets whether or not to treat the text in **input string** as a multiple-line string. This affects how the ^ and $ characters handle matches. When FALSE (the default), entering "^" matches the beginning of the line in **input string** only and entering "$" matches the end of **input string** only. When TRUE, "^" matches the beginning of any line in **input string** and "$" matches the end of any line in **input string**.

ignore case? specifies whether the string search is case sensitive. If FALSE (default), the string search is case sensitive.

replace all? specifies whether the function should replace all occurrences of the **search string** or only the first occurrence. If FALSE (default), the function replaces only the first occurrence.

input string is the input string the function searches.

search string is the string you want to search for and replace. If **search string** is an empty string and **replace all?** is FALSE, the function inserts **replace string** at the beginning of **input string**. If **search string** is an empty string and **replace all?** is TRUE, the function inserts **replace string** at the beginning, in between each character, and at the end of **input string**.

replace string replaces the string in **search string**. The default is an empty string.

offset determines the number of characters into **input string** at which the function starts searching for **search string**.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

result string contains the **input string** with one or all occurrences of **search string** replaced with **replace string**. If **replace string** is empty, **result string** contains **input string** with **search string** deleted.

number of replacements returns the number of times LabVIEW replaced **search string**.

offset past replacement indicates the offset in **result string** of the character immediately following where the last match and replacement occurred. If **replace all?** is FALSE, the next match, if any, is after this point.

If the function does not find **search string**, **offset past replacement** is -1.

error out contains error information. This output provides standard error out functionality.

Figure 19.3: Help file for the `Search and Replace String` VI ■

Exercise 7.6: Create the `Get Message` Sub-VI

Create the `Get Message` sub-VI with the icon shown in Figure 19.4. This sub-VI accepts an account cluster and a message type as inputs and it produces a personalized message as an output. This sub-VI will call the `Get Message Templates` sub-VI

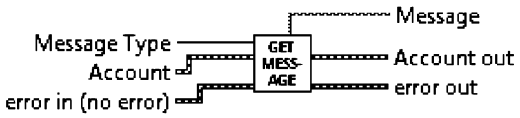


Figure 19.4: The icon of the `Get Message` VI

developed in Exercise 7.4 to get the correct message template. It will then construct the key and value arrays and pass them to the `Replace Keys With Values` sub-VI developed in Exercise 7.5 to generate the personalized message. ■

Most of the activity of the ATM controller involves user accounts. The accounts are stored in a comma-separated value (CSV) text file. Each account is given a separate row in the file. Each account consists of a five digit account number, first name, last name, and current balance (in that order). When the balance is changed, the file needs to be updated. We operate under the assumption that the record does not need to be locked during the operation of the program. We assume that the account only can be accessed from one ATM machine at a time. This is a significant assumption, but designing a security system for locking one record of a text file that is a shared resource for a network of ATMs is not in the scope of the functional specification.

The account information is represented in the controller as a type-defined cluster created in Exercise 7.3. The cluster contains five string controls: *Account Number*, *First Name*, *Last Name*, *Balance*, and *Transaction*. The first four of these items are recorded in the accounts file. The *Transaction* field is used to store pending deposit and withdrawal amounts. In the following exercises, you will develop the sub-VIs needed to manage the account file. You will also develop sub-VIs to manipulate the the account information through deposits and withdrawals.

Exercise 7.7: Create the `Get Account` Sub-VI

Create `Get Account` sub-VI with the icon shown in Figure 19.5. This sub-VI accepts an account number string and the account file path as inputs. It returns the user account cluster, its position in the file, and a Boolean indicating if an account was found as outputs.

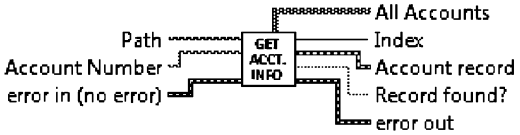


Figure 19.5: The icon of the `Get Account` VI ■

Exercise 7.8: Create the `Verify Account` Sub-VI

Create the `Verify Account` sub-VI with the icon shown in Figure 19.6. This sub-VI accepts a path to the account file and an account number string as inputs. It searches

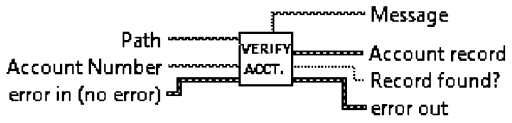


Figure 19.6: The icon of the `Verify Account` VI

the account file to see if the input account number exists within the file and outputs a RECORD FOUND? Boolean indicating the result of the search. This sub-VI also generates the appropriate ATM message informing the user whether or not the account was found. The account record is also produced as an output. ■

Exercise 7.9: Create the `Update Account File` Sub-VI

Create the `Update Account File` sub-VI with the icon shown in Figure 19.7. This sub-VI takes an account cluster and the account file path as inputs. It opens the account file, finds the record corresponding to the account number contained in the input account cluster, and updates the record using the information found in the input account cluster. Use the `Get Account` sub-VI developed in Exercise 7.7 to read the account file and generate an array of all account information, replace the old account information with the new account information, and then overwrite the accounts file with the updated array. You might find the **`Replace Array Subset`** function useful (see the help file in Figure 19.8), as well as the **`Write To Spreadsheet File`** function (see the help file in Figure 19.9).



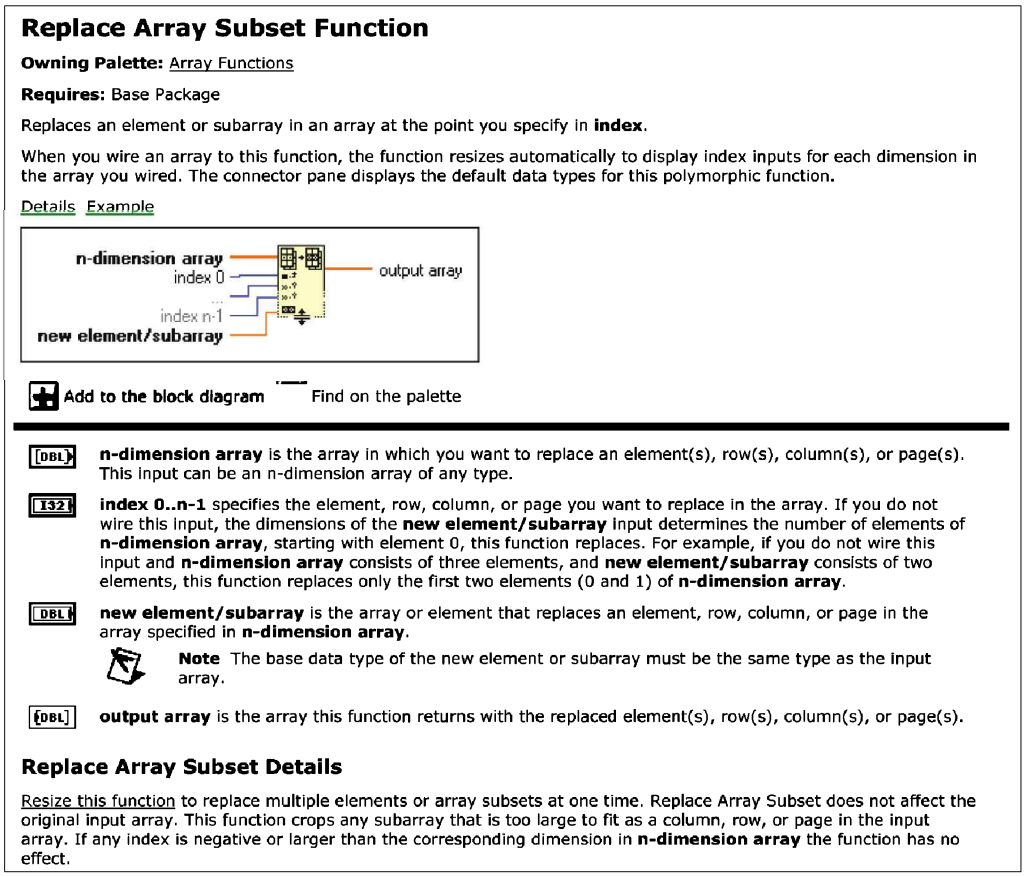
Figure 19.7: The icon of the `Update Account File` VI ■

Exercise 7.10: Create the `Deposit` Sub-VI

Create the `Deposit` sub-VI with the icon shown in Figure 19.10. This sub-VI accepts an account cluster and a deposit amount string as inputs. The deposit amount is added to the balance of the account and updates the balance of the account cluster. This sub-VI should also store the deposit amount in the *Transaction* field of the account cluster. ■

Exercise 7.11: Create the `Withdrawal` Sub-VI

Create the `Deposit` sub-VI with the icon shown in Figure 19.11. This sub-VI accepts an account cluster and a withdrawal amount string as inputs. This sub-VI subtracts the withdrawal amount from the balance of the account and updates the account cluster if there are sufficient funds in the account. The withdrawal amount also should be written to the *Transaction* field of the account cluster. If the balance is not large enough to permit the withdrawal, the account information is passed through unchanged. A SUFFICIENT FUNDS? Boolean is also output indicating if the withdrawal was successful or not. ■



Replace Array Subset Function

Owning Palette: Array Functions

Requires: Base Package

Replaces an element or subarray in an array at the point you specify in **index**.

When you wire an array to this function, the function resizes automatically to display index inputs for each dimension in the array you wired. The connector pane displays the default data types for this polymorphic function.

Details Example

Add to the block diagram Find on the palette

n-dimension array is the array in which you want to replace an element(s), row(s), column(s), or page(s). This input can be an n-dimension array of any type.

index 0..n-1 specifies the element, row, column, or page you want to replace in the array. If you do not wire this input, the dimensions of the **new element/subarray** input determines the number of elements of **n-dimension array**, starting with element 0, this function replaces. For example, if you do not wire this input and **n-dimension array** consists of three elements, and **new element/subarray** consists of two elements, this function replaces only the first two elements (0 and 1) of **n-dimension array**.

new element/subarray is the array or element that replaces an element, row, column, or page in the array specified in **n-dimension array**.

Note The base data type of the new element or subarray must be the same type as the input array.

output array is the array this function returns with the replaced element(s), row(s), column(s), or page(s).

Replace Array Subset Details

Resize this function to replace multiple elements or array subsets at one time. Replace Array Subset does not affect the original input array. This function crops any subarray that is too large to fit as a column, row, or page in the input array. If any index is negative or larger than the corresponding dimension in **n-dimension array** the function has no effect.

Figure 19.8: Help file for the `Replace Array Subset` VI

Exercise 7.12: Create the `Enter Button Next State` Sub-VI

Create the `Enter Button Next State` sub-VI with the icon shown in Figure 19.12. This sub-VI handles the modal nature of the ENTER button. It accepts the enumerated constant containing the modes of the ENTER button as an input, and it outputs the next state following the value change of the enter button. If the current mode of the ENTER button is *Verify Account*, then a value change of the ENTER button signifies that the user entered an incorrect account number and is trying again. The next state is *Verify Account*. If the current mode of the ENTER button is *Deposit*, a value change of the ENTER button signifies that the user is entering the deposit amount, and the next state is *Complete Deposit*. If the current mode of the ENTER button is *Withdraw*, then a value change of the ENTER button signifies that the user is entering the withdrawal amount, and the next state is *Complete Withdrawal*. If the current mode of the ENTER button is *Idle*, then a value change of the ENTER button should be ignored and the next state is *Wait*. ■

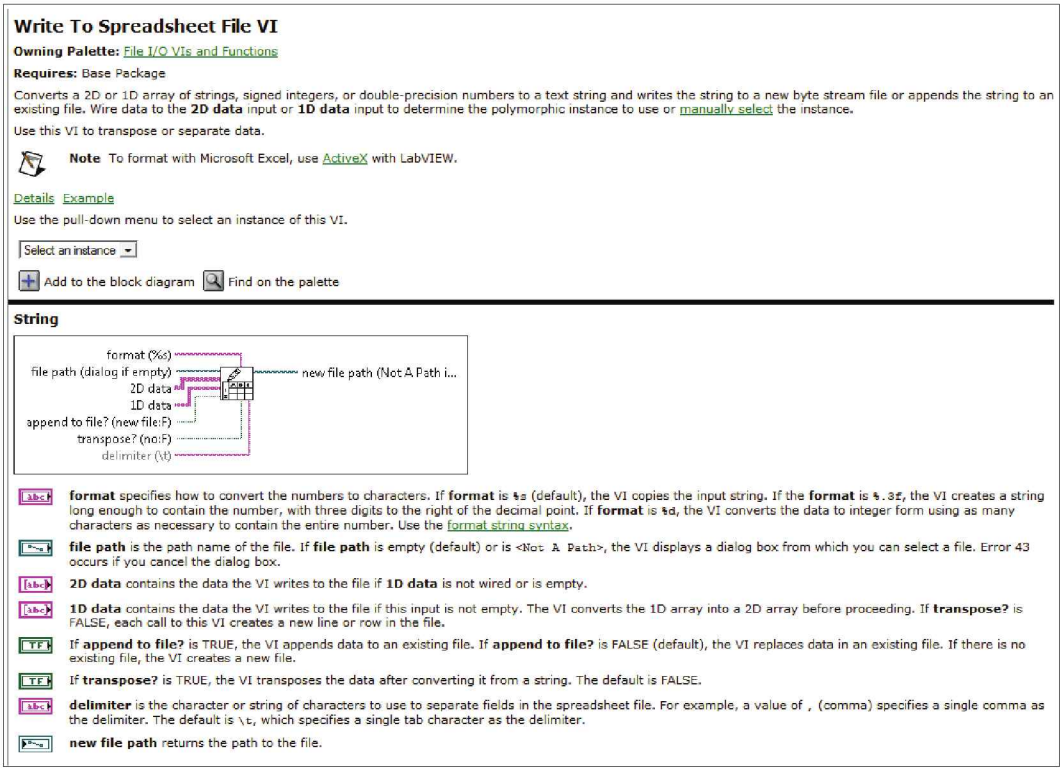


Figure 19.9: Help file for the `Write To Spreadsheet File` VI

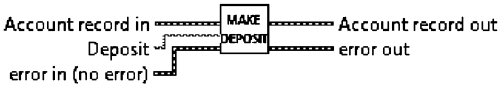


Figure 19.10: The icon of the `Deposit` VI

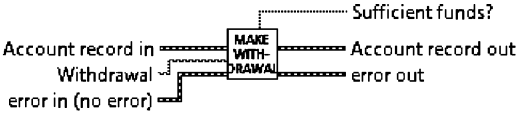


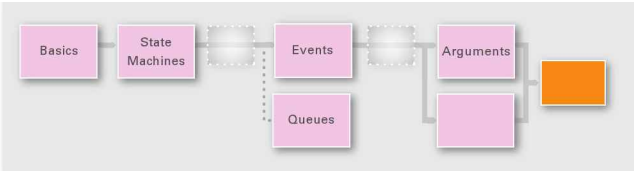
Figure 19.11: The icon of the `Withdrawal` VI



Figure 19.12: The icon of the `Enter Button Next State` VI

Exercise 7.13: Create the ATM Controller

Now that all of the sub-VIs have been created for handling message personalization, account manipulation, and the modal ENTER button, create the event-driven classic state machine implementation of the ATM controller. Place the ATM side menus on the data highway to make the application more scalable in the future. ■



20

State Machines with Arguments

Queued, event-driven state machines are powerful, flexible, and efficient and are applicable to a wide range of problems. In this chapter, we look at an upgrade to this design pattern that is not as widely applicable but is powerful when it can be used—the state machine with arguments.

States in a state machine are very much like sub-VIs. The code within a state is self-contained—just like a sub-VI. Dequeuing a state and executing it is much like calling a sub-VI. A state can be repeated over and over again—just like a sub-VI can be called repeatedly.

Information can be passed to a state—just like inputs or arguments—can be passed to a sub-VI. For example, in the car wash controller, the proximity switch cluster is passed as an input to the car wash cycle states. The main error wire and the queue reference wire also enter each state as inputs.

But what if we could pass other kinds of information to states as arguments? In the car wash controller, the five car wash cycle states are nearly identical (*Underbody Wash*, *Soap Application*, *Main Wash*, *Rinse*, and *Air Dry*). The only differences in these states are which proximity switch is monitored, what time target is used, and which indicators are lit. Imagine if we could pass that information to a generic *Cycle* state as arguments. Then we could replace five specific car wash cycle states with a single generic *Cycle* state that accepts arguments. This would be a big improvement in readability, maintainability, and scalability.

In this chapter, we examine different schemes for passing information to states as arguments using the data highway, queued clusters, and queued strings. We practice on the `Multitest` VI and then develop a full-blown car wash controller with arguments in Chapter 21.

20.1 The Data Highway

State machines use the data highway to pass information from one iteration of the main loop to the next, but it also can be used to pass information from one state to another, since the data highway can be accessed by all of the states in a state machine.

As an example of this, let's revise the queued `Multitest` VI developed in Chapter 13. Recall that in this implementation of the `Multitest` VI, there are separate states for running Test 1 and Test 2. The only difference between these states is the value of the string displayed by the one-button dialog. We can replace the *Run Test 1* and *Run Test 2* states with a single generic *Run Test* state if we use the data highway to store the value of the dialog string.

Figure 20.1 shows the new generic *Run Test* state of the revised `Multitest` VI. Notice the addition of a data highway to store the value of the *Test ID* string. The value of the *Test ID* string is retrieved from the data highway and is displayed to the user via the **`One Button Dialog`** function.

Style Guide: Notice that the data highway wire has a label. Always label wires on the data highway to improve readability of your code.

Figure 20.2 shows the event case handling the value change of the RUN TEST 1 button of the revised `Multitest` VI. The generic *Run Test* state is enqueued instead of the *Run Test 1* state used in Chapter 13. The `Run Test 1` string is placed on the data highway for use by the *Run Test* state.

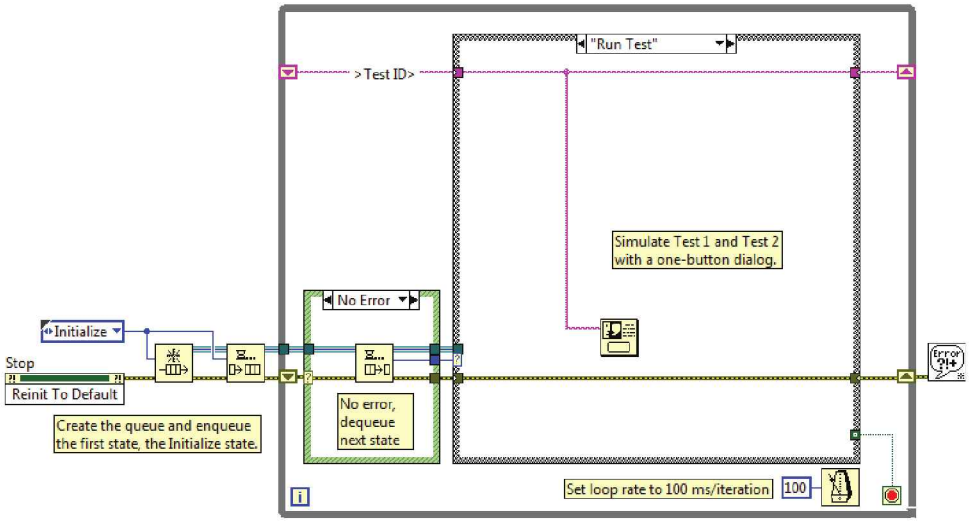


Figure 20.1: The *Run Test* state of the revised `Multitest` VI

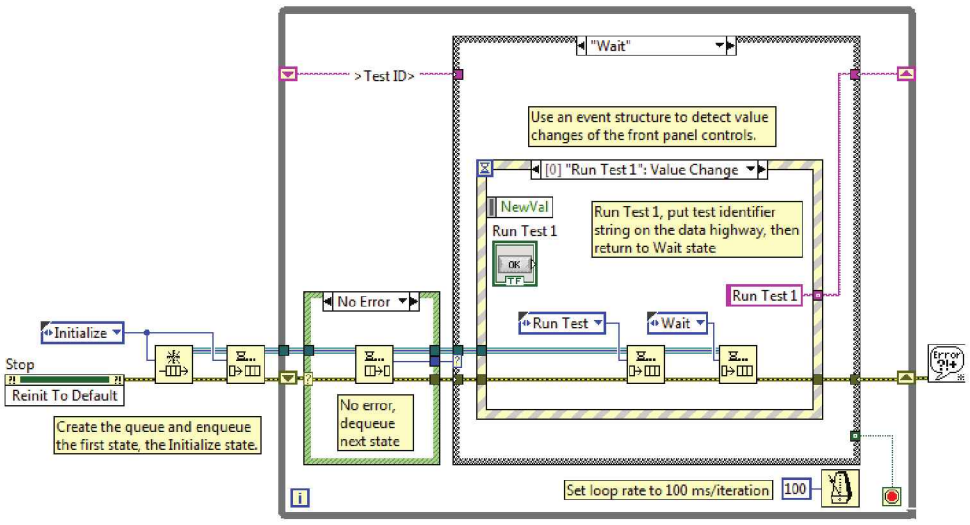


Figure 20.2: The event case handling the value change of the RUN TEST 1 button

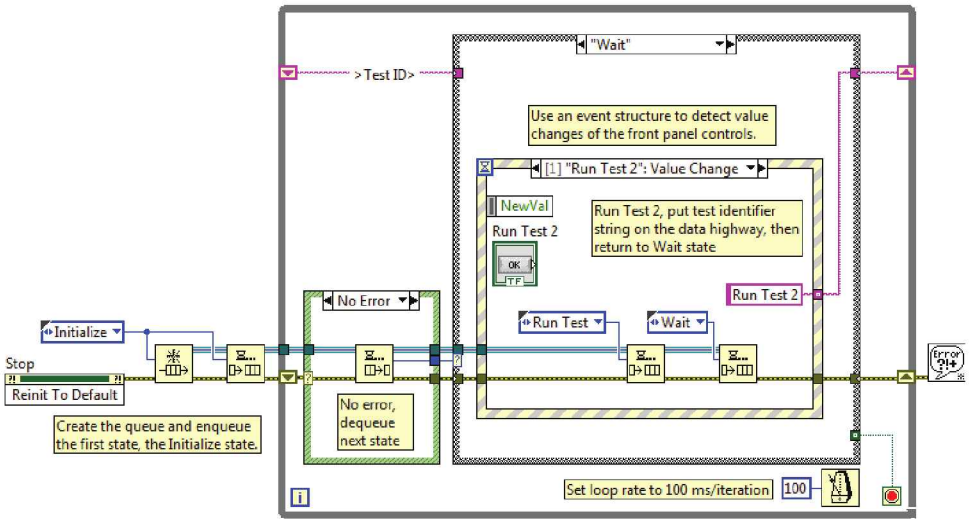


Figure 20.3: The event case handling the value change of the RUN TEST 2 button

Figure 20.3 shows the corresponding event case for the value change of the RUN TEST 2 button. This case enqueues the same *Run Test* state and places the `Run Test 2` string on the data highway for use by the *Run Test* state. The data highway acts as short-term storage for the *Test ID* string. The event cases handling the value changes of the RUN TEST 1 and RUN TEST 2 buttons use the data highway to pass the *Test ID* string as an argument to the generic *Run Test* state.

Wiring Tip: In this implementation of the `Multitest` VI, we removed the *Run Test 1* and *Run Test 2* states and replaced them with the *Run Test* state. Changing the states means changing the enumerated constant listing the states. This demonstrates why the enumerated constant should be saved as a type-definition. Changing the states in the type definition file automatically changes the states in all of the enumerated constants used in the `Multitest` VI without breaking any wires. If the enumerated constants were not saved as type definitions, the wires would break as soon as one of the constants was changed. Each constant would then have to be changed manually.

20.2 Using Arguments to Pass Information

The previous data highway approach worked because only one argument needed to be passed to one state at a time. But what if multiple arguments needed to be passed? What if a series of states required arguments? What if different states required arguments with various data types or various numbers of arguments?

The data highway would be awkward in such a scenario. The number of wires would multiply, or arrays or clusters of data would need to placed on the data highway with each state accessing a different element. This would degrade the readability, scalability, and maintainability of the code. A better method needs to be used that targets each state with the individual arguments needed by that state. We need a way to "bundle together" the state name and its arguments into a single data package that can be processed by a state machine. This idea of bundling gives a clue as how to proceed with adding arguments to a LabVIEW state machine.

20.3 Passing Arguments Using Clusters

In Chapters 13 and 14, we used a queue to store state information. That queue element was a type-defined enumerated constant containing the names of the possible states. But queue elements are not limited to enumerated constants. Queue elements can be used to carry both the name of the next state and the arguments used by that state.

One way to do this is to use a cluster as the queue element. The cluster bundles the state enumerated constant with the needed arguments. When the cluster is dequeued, the enumerated constant can be unbundled to determine the next state, and the arguments can be unbundled and passed as inputs to the main case structure of the state machine.

Let's take the `Multitest` VI as an example. Once again, the goal is to consolidate the *Run Test 1* state and the *Run Test 2* state into a generic *Run Test* state that accepts a string as an argument. The queue element will be a cluster of the enumerated constant containing the states and the string argument, as shown in Figure 20.4.



Figure 20.4: Cluster of state and argument used in the cluster-based queue of the `Multitest` VI

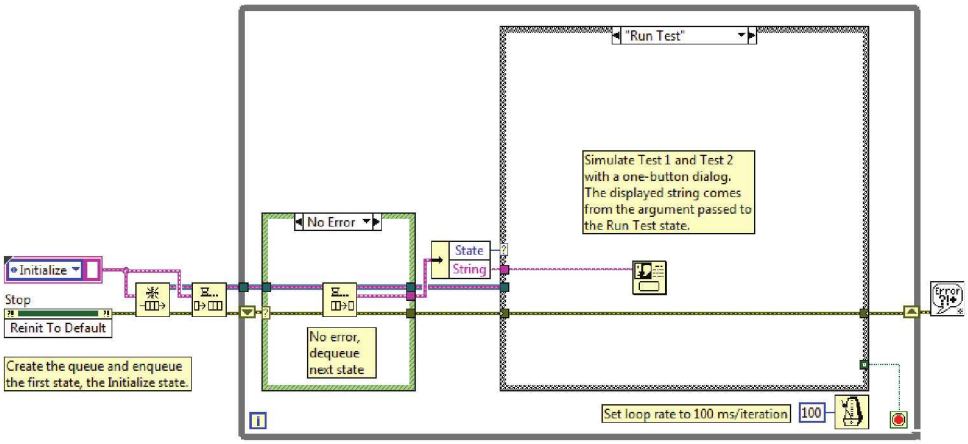


Figure 20.5: The *Run Test* state of the state machine with arguments implementation of the `Multitest` VI

Wiring Tip: It is usually a good idea to create type definitions of important clusters in a VI. In this case, the cluster that forms the basis of the queue that drives the `Multitest` state machine certainly qualifies as an important cluster. It should be saved as a type definition.

The implementation of the argument-based `Multitest` VI is straightforward. The overall structure of the cluster-based state machine with arguments is shown in Figure 20.5. The queue is created outside the loop (as in Chapter 14), but this time the cluster in Figure 20.4 is used as the queue data type.

After the queue has been created, the first element is enqueued: a cluster of the *Initialize* state and a blank string constant. This illustrates a drawback of the cluster-based queue. The *Initialize* state doesn't need a string argument, but it must be included anyway. The queue element has the data type of a cluster consisting of an enumerated constant and a string constant, and every element in the queue must have the same data type. The string constant must be included even when it is not needed.

The queue reference is then passed into the main loop. The guard clause checks to see if an error is present. If the guard clause detects no error, the next element is dequeued. The **`Unbundle By Name`** function is then used between the guard clause and the main case structure to unbundle the state information and the string argument. The state information is wired to the case structure selector to determine the next state.

The string argument is passed into the case structure as an input. The *Run Test* state shown in Figure 20.5 accepts the string as an argument and displays it to the user.

If the guard clause detects an error, the queue is bypassed, and the state machine transitions to the *Shutdown* state, as shown in Figure 20.6. The *Shutdown* state does not need an argument, so the string input tunnel is left unwired. Even though the *Shutdown* state needs no arguments, the cluster in the guard clause must still include the string constant.

The event cases handling the value change of the RUN TEST 1 or the RUN TEST 2 buttons are shown in Figures 20.7 and 20.8. Each of these states combines the *Run Test* state with its argument into a single cluster and then places that cluster on the queue.

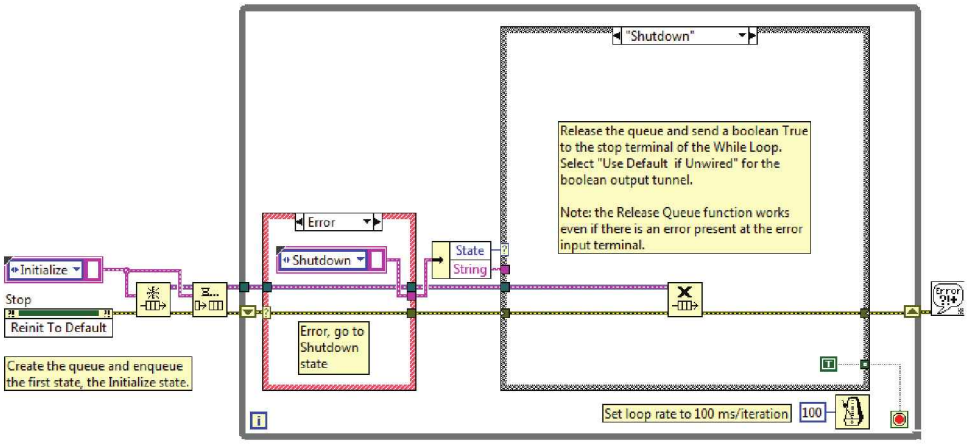


Figure 20.6: The *Shutdown* state of the state machine with arguments implementation of the `Multitest` VI

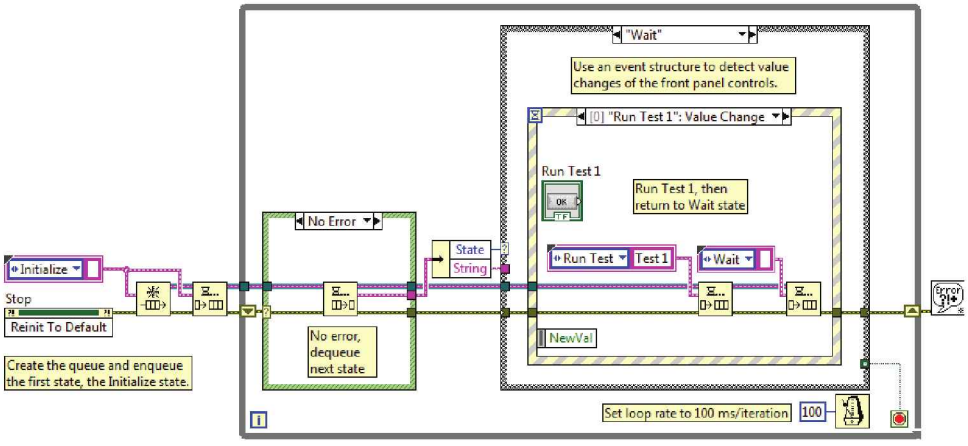


Figure 20.7: The event case handling the value change of the RUN TEST 1 button of the state machine with arguments implementation of the `Multitest` VI

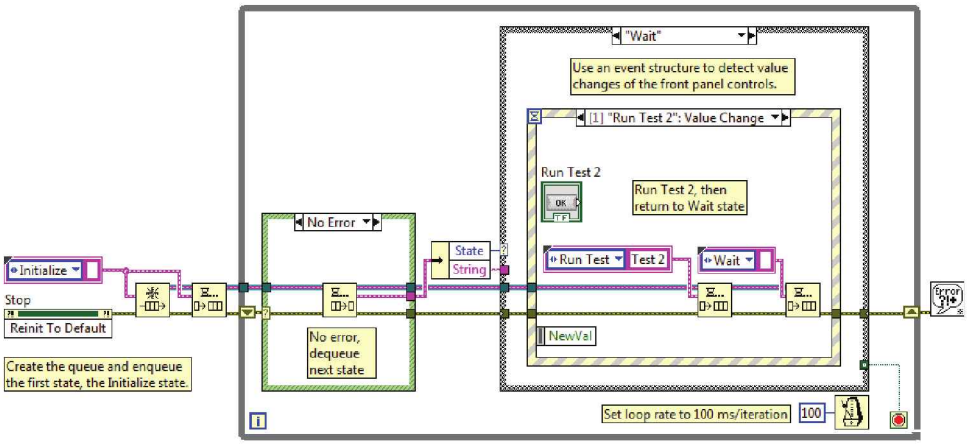


Figure 20.8: The event case handling the value change of the RUN TEST 2 button of the state machine with arguments implementation of the `Multitest` VI

20.4 Passing Arguments Using Strings

Clusters are an easy way to pass arguments to states, but they are limited by the fact that clusters have fixed data types. This means that the arguments always have to be included even if they aren't needed for a particular state. When clusters with arguments are used to create a queue, the arguments are never optional. This problem gets worse as the number of arguments increases. Clusters with large numbers of dummy arguments take up valuable screen real estate.

Another issue arises if different states need different types of arguments. Since the data type of the cluster is fixed, all of the required argument data types would need to be included in the queued cluster. For instance, if one state needs a Boolean argument and another state needs a numeric argument, the queued cluster would need to include both a Boolean and a numeric constant as well as the state enumerated constant. One state would use the Boolean argument and ignore the numeric argument, and the other state would use the numeric argument and ignore the Boolean argument.

One way around these difficulties is to use strings as the base element of the queue. Strings also have a fixed data type, but strings are more malleable than clusters. LabVIEW has many native functions for manipulating, splitting, searching, and concatenating strings. A clever LabVIEW developer can use these functions to create and parse strings that contain both state information and arguments in any format the developer desires.

20.5 Designing Parsing Sub-VIs

As an example, let's adopt a simple programming convention for separating state information from arguments in a string using a delimiter. If we use a colon as a

delimiter, we could construct a string with state information and an argument using the format `State:Argument`. The string consists of two pieces of information separated by a colon with the state first and the argument following the state.

A parsing routine could be written that accepts a string with this format as an input. The routine splits the state from the argument, outputting them as separate strings. If no delimiter is found in the input string (and thus no argument), a null string is output as the argument string.

This scheme can be implemented using the native **`Match Pattern`** function whose help file is shown in Figure 20.9. The **`Match Pattern`** function takes a string and a regular expression as inputs. The outputs include the portion of the string before the regular expression, the matching string, and the portion after the regular expression.

Regular expressions are dense, compact, and very powerful statements used for parsing text. They are beyond the scope of this book, but for our purposes, a regular expression can be thought of as a pattern string to be matched in a larger string.

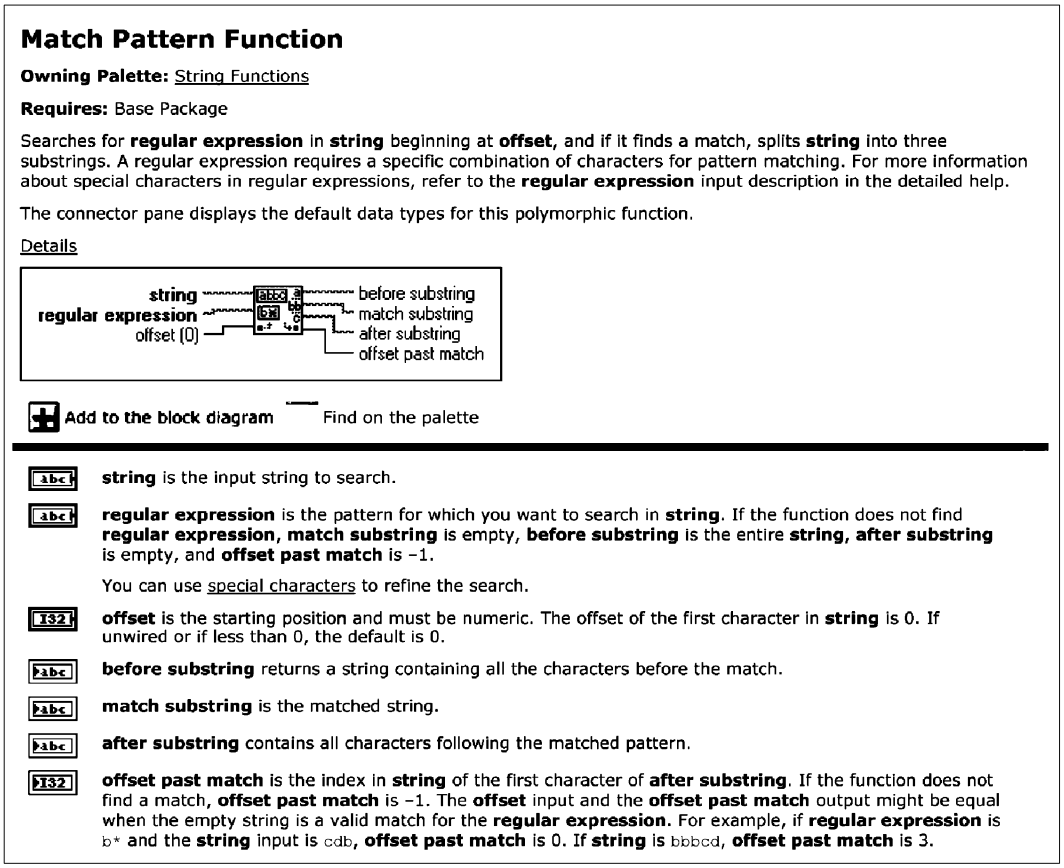


Figure 20.9: Help file for the `Match Pattern` function

To search for a colon, the regular expression is simply a colon, ":". The "before substring" output of the **Match Pattern** VI corresponds to the state, and the "after substring" output corresponds to the argument. Note that the **Match Pattern** function only finds the first instance of the search string. This will be an important consideration in the next chapter when we extend our parsing routine to handle strings with multiple arguments.

A parsing sub-VI, `Parse State and Arguments`, can be developed using the **Match Pattern** function. The icon for this sub-VI is shown in Figure 20.10. The VI takes a string as an input, splits the string at the first colon found, and returns the portions of the string before and after the colon. This is illustrated by the front panel of the sub-VI, shown in Figure 20.11. The block diagram to accomplish this task is shown in Figure 20.12. As seen in the figures, this sub-VI is simply a wrapper of the **Match Pattern** function designating a colon as the delimiter.

What happens if the `Parse State and Arguments` VI is given a string without a colon as an input? If the **Match Pattern** function cannot find the pattern string, then the "before substring" will equal the input string and the "after substring" will be the empty string. This is desired behavior for our parsing sub-VI. If the input string has no arguments, such as `Initialize`, the state output will be `Initialize` and the argument output will be the empty string.



Figure 20.10: The icon of the `Parse State and Arguments` sub-VI



Figure 20.11: The front panel of the `Parse State and Arguments` sub-VI

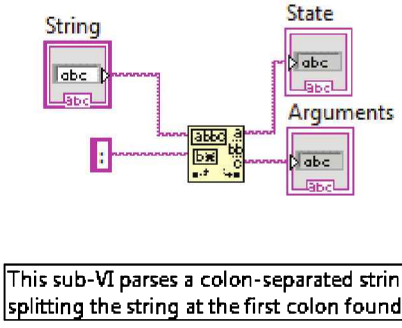


Figure 20.12: The block diagram of the `Parse State and Arguments` sub-VI

20.6 The String-Based `Multitest` VI

The parsing scheme described previously now can be applied to the `Multitest` VI. Let's look at the overall structure of the `Multitest` VI with a string-based queue by examining Figure 20.13. This figure shows the block diagram of the `Multitest` VI in the *Initialize* state.

As usual, the queue is created outside the main loop of the state machine. The `Initialize` string is used to specify the data type of the queue element and is also enqueued as the first element. Notice that there are no arguments attached to this string. Contrast this with the cluster-based queue element used to set the data type of the queue in Figure 20.5. Even though the *Initialize* state needs no arguments, the cluster in the input section of the loop must include a dummy argument for the test string. In the string-based scheme, the arguments are optional.

The guard clause then checks to see if an error has occurred. When no error is present, the next state is dequeued, parsed, and sent to the main case structure of the state machine. When an error is present, the queue is bypassed and the *Shutdown* state is sent to the main case structure of the state machine, as seen in Figure 20.14. The `Stop` string in the guard clause needs no arguments.

After the guard clause, we find the `Parse State and Arguments` sub-VI. This replaces the **`Unbundle By Name`** function used in the cluster-based version of the state machine with arguments. This sub-VI parses the input string and separates it into a state string and an argument string. The state string is wired to the selector terminal of the case structure, and the argument string is passed to the case structure as an input.

The *Initialize* case shown in Figure 20.13 resets the front panel controls and queues up the *Wait* state. The *Wait* state needs no argument, so none is provided. The *Wait* state contains the event structure that waits for user input, such as pressing

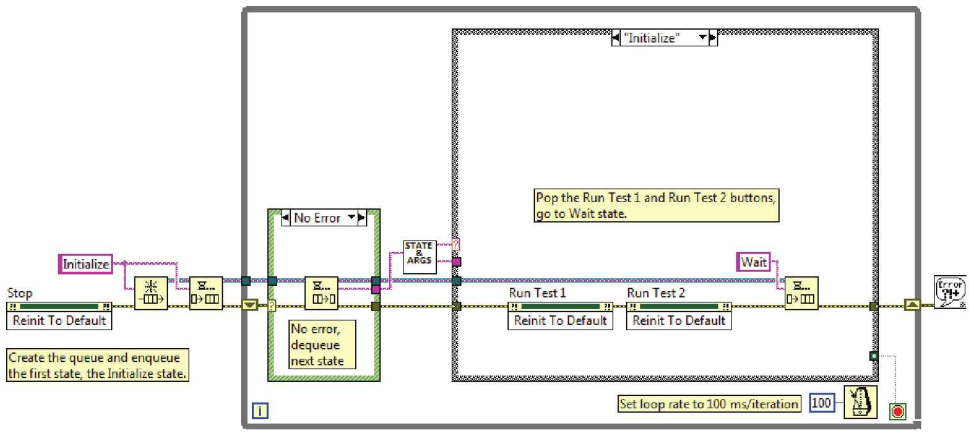


Figure 20.13: Overall structure of a queued string state machine implementation of the `Multitest` VI example where the *No Error* case of the guard clause is shown

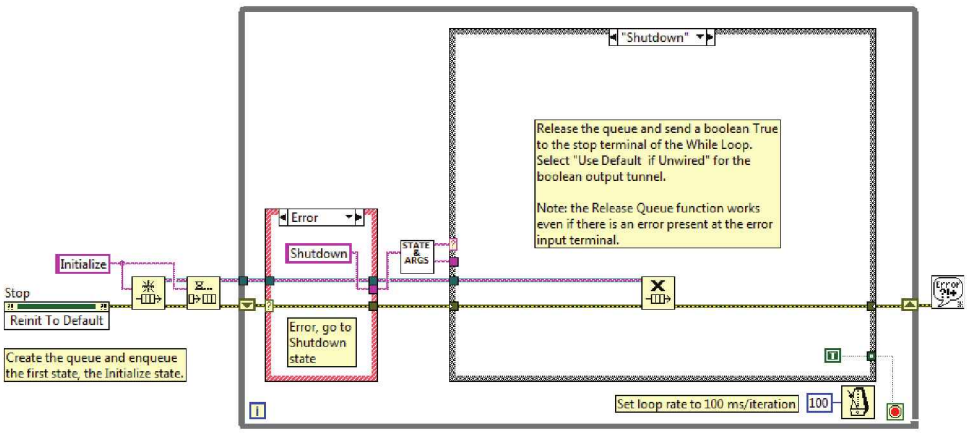


Figure 20.14: The *Shutdown* state of the queued string `Multitest` VI, where the *Error* case of the guard clause is shown

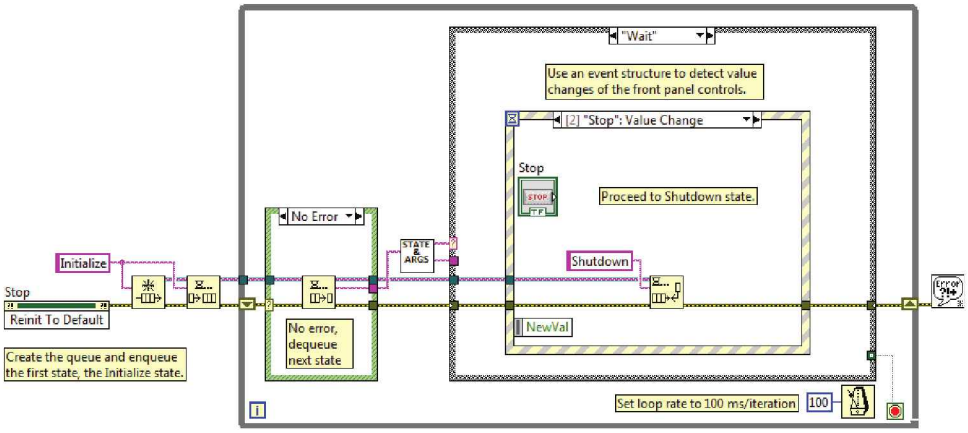


Figure 20.15: The event case handling the value change of the STOP button of the queued string `Multitest` VI

the STOP button. The event case handling the value change of the STOP button is shown in Figure 20.15. The *Shutdown* case of the state machine was shown in Figure 20.14.

The event case handling the value change of the RUN TEST 1 button is shown in Figure 20.16. Two strings are enqueued: `Run Test:Test 1` and `Wait`. Compare this with the cluster-based implementation in Figure 20.7. Notice how the string-based version is more readable. The block diagram has a "lighter" feel to it, and the elements are simple strings—not clusters. The border of the string elements are thinner, and there is less clutter within the elements. Elements that don't need arguments don't

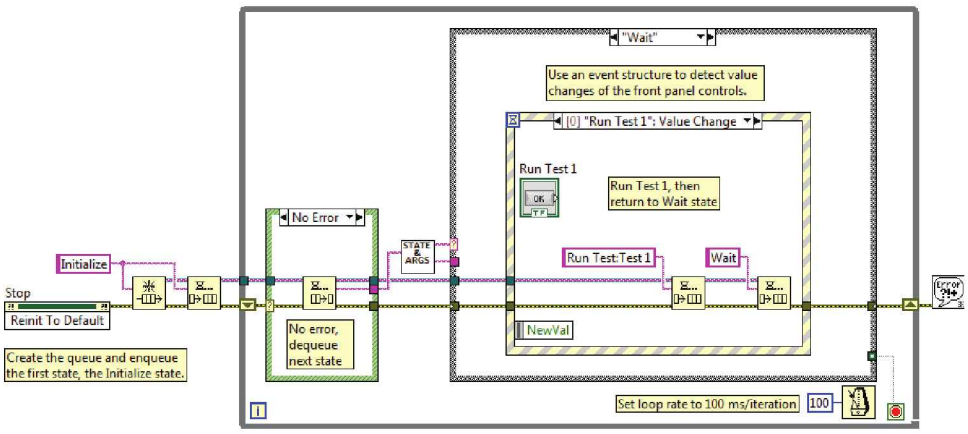


Figure 20.16: The event case handling the value change of the RUN TEST 1 button of the queued string `Multitest` VI example

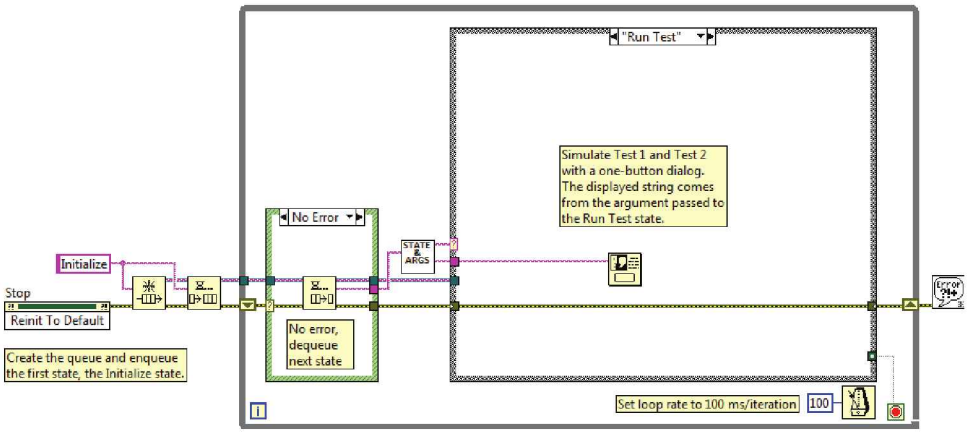


Figure 20.17: The *Run Test* state of the queued string `Multitest` VI

have arguments. All of these factors increase the readability of the code. This increase in readability will be even more evident when we apply a string-based queue to the car wash controller in the next chapter.

After the `Run Test:Test 1` and `Wait` strings have been enqueued, the state machine moves on to the next iteration. When the `Run Test:Test 1` string is dequeued and parsed, the `Run Test` string is passed to the selector of the case structure, and the `Test 1` string is passed to the input tunnel of the case structure as an argument. The *Run Test* state is shown in Figure 20.17.

The event case handling the value change of the RUN TEST 2 button, shown in Figure 20.18, is much the same as the case handling the RUN TEST 1 button. The only

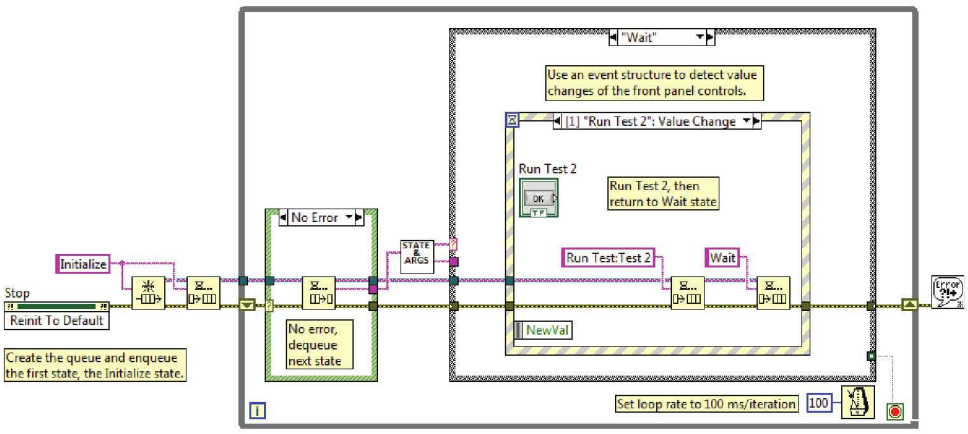


Figure 20.18: The event case handling the value change of the RUN TEST 2 button of the queued string `Multitest` VI

difference is that the argument of the of the *Run Test* state is `Run 2` instead of `Run 1`. A single *Run Test* state can be used for running either test.

20.7 Disadvantages of Using Strings

String-based queues have two disadvantages when compared to queues using clusters and enumerated constants. The major disadvantage is that you have to parse the strings yourself. The cluster-based state machine used the **`Unbundle By Name`** function to separate the state and arguments. To use strings, you must first decide on a parsing convention to separate the states and arguments. Then you must build the support Vis to perform the parsing.

A second disadvantage is that case structures in LabVIEW are not optimized to work with strings as they are with enumerated constants. When an enumerated constant is wired to a case structure, you can right-click on the case selector and choose *Add Case for Every Value* to automatically populate the case structure with every value in the enumerated constant. This is a good defensive programming technique. When an enumerated constant is used with a case structure that has been populated in this way, there is very little danger of an unknown state being sent to the selector terminal of the case structure.

This works because LabVIEW knows all of the possible values of an enumerated constant before the code runs. This is not true for strings. An enumerated constant has a fixed range of values, while a string does not. LabVIEW can't guess ahead of time what strings will be sent to a case structure, so it can't assign a case for every value. This means that there is a real danger of unknown strings being sent to the case structure (usually as a result of the LabVIEW developer misspelling one of the states).

To address this problem, a string-based state machine must always have a default state designated to trap unknown states. One way to do this is to create a default case

for the empty string. The default case should contain code to handle the unknown string, perhaps by generating a custom error.

Custom errors are useful tools, allowing the developer to create warnings or errors for situations specific to a particular VI. LabVIEW reserves the ranges of error codes −8999 through −8000, 5000 through 9999, and 500,000 through 599,999 for custom use. Custom errors can be generated using the **`Error Cluster From Error Code`** function. The help file is shown in Figure 20.19. This function accepts an error code and an error message string as inputs and creates the custom error cluster. The **`Error Cluster From Error Code`** function is useful because the error message strings can be programatically determined at run-time.

An example of this technique can be found in Figure 20.20. The default case shown here creates a custom error message when an unknown state is requested. The **`Concatenate Strings`** function is used to incorporate the unknown state string into the error message. When this custom error is generated, the state machine will respond by shutting down gracefully and notifying the user that an unknown state has been requested.

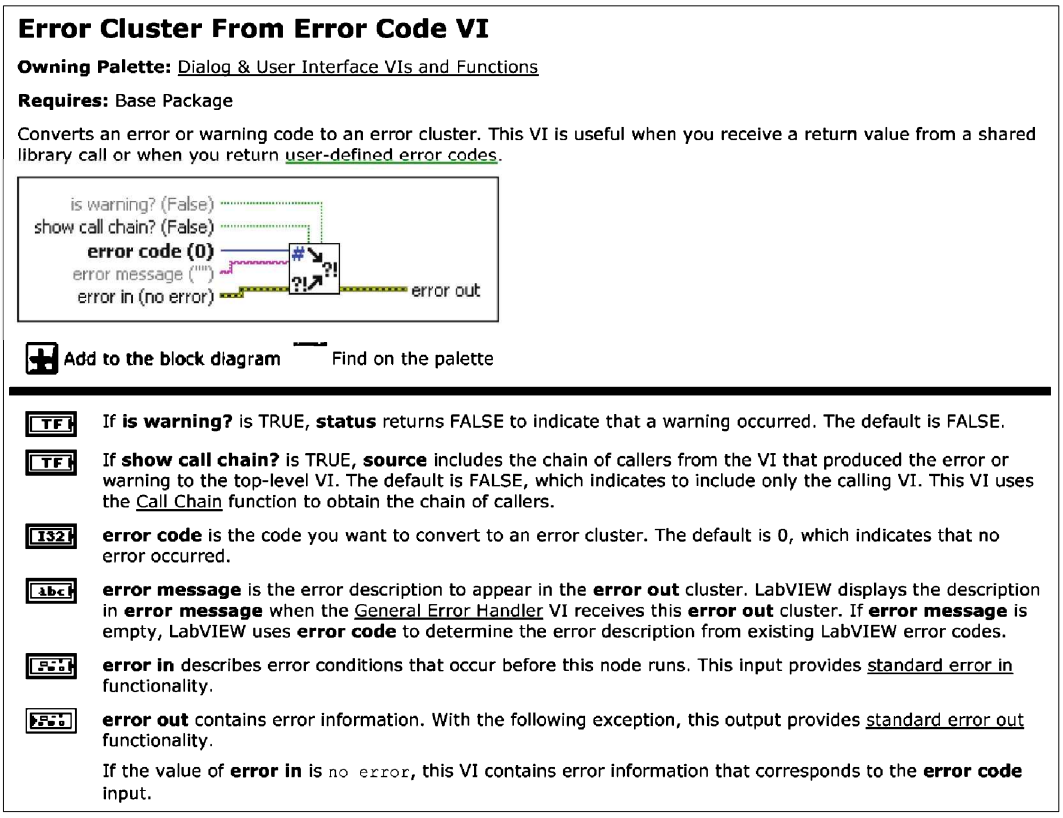


Figure 20.19: Help file for the `Error Cluster From Error Code` function

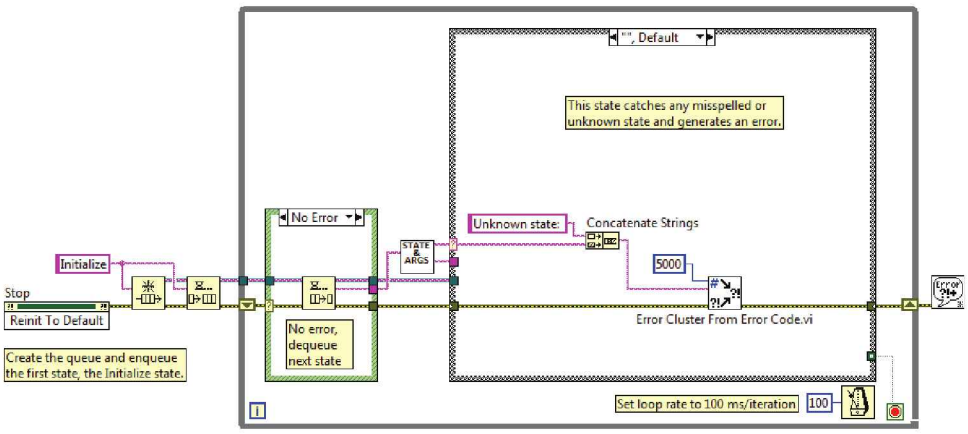


Figure 20.20: Default case of the queued string `Multitest` VI

20.8 Improving the String-Based State Machine

The `Multitest` VI that we just developed is functional, but not easily scalable. Queueing individual strings is awkward and consumes too much space on the block diagram. Look back at Figure 20.16. Imagine queueing all of the states needed to run the Deluxe Wash with each string wired to a separate **`Enqueue Element`** function. It would stretch the block diagram beyond the width of the screen and would be difficult to read.

In Chapter 14, we used an array of states and a **`For`** loop to enqueue a group of states (see Figure 14.8). This saved space and improved readability. We can do the same here for the string-based `Multitest` VI, as demonstrated in Figure 20.21.

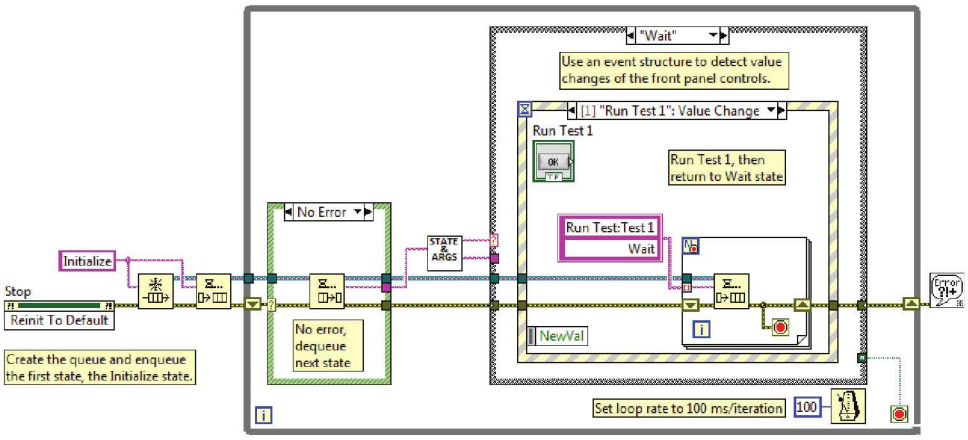


Figure 20.21: Implementing an array and `For` loop to queue states in the `Multitest` VI

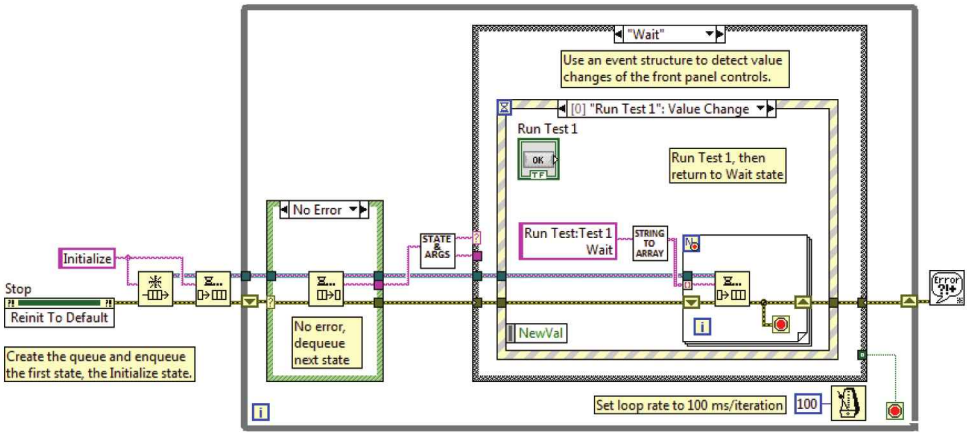


Figure 20.22: Replacing the array of string constants with a single string using the `String to Array` VI



Figure 20.23: Icon for the `String to Array` sub-VI

Now that we are working with strings, we can improve upon this technique. Instead of using an array of strings in the event structure, it is much easier to use a single multiline string that could be converted into an array. This idea is illustrated in Figure 20.22. In this version of the `Multitest` VI, we adopted a second parsing convention. Multiple state and argument strings are now represented by a single string with each state and argument string separated by a linefeed. Notice how this improves the readability of the code. It also improves maintainability, because it is much easier to edit a single multiline string than it is to edit an array of strings.

The conversion of a multiline string to a string array can be accomplished with the `String to Array` sub-VI with the icon shown in Figure 20.23. The sub-VI takes a multiline string as an input, splits it into its individual lines, and returns an array of strings made up of these individual lines. This process is illustrated by the inputs and outputs of the front panel shown in Figure 20.24.

The `String to Array` sub-VI makes use of the native **`Spreadsheet String To Array`** function. This help file is found in Figure 20.25. This function takes a string that represents tabulated data as an input. The data type of the input data is specified by a format input, and the delimiter is also specified as an input. The output of this function is an array of the tabulated data values in their correct order and in the specified data format. The `String to Array` sub-VI uses this function to split a multiline string into an array of individual strings by specifying the linefeed character as a delimiter, as shown in Figure 20.26.

We can take a further step to improve the readability of the string-based state machine with arguments. Compare Figures 14.8 and 20.22. The **`For`** loops in these

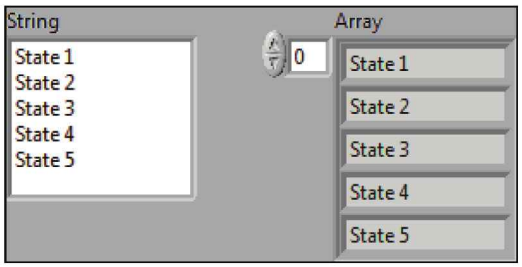


Figure 20.24: Front panel of the `String to Array` sub-VI

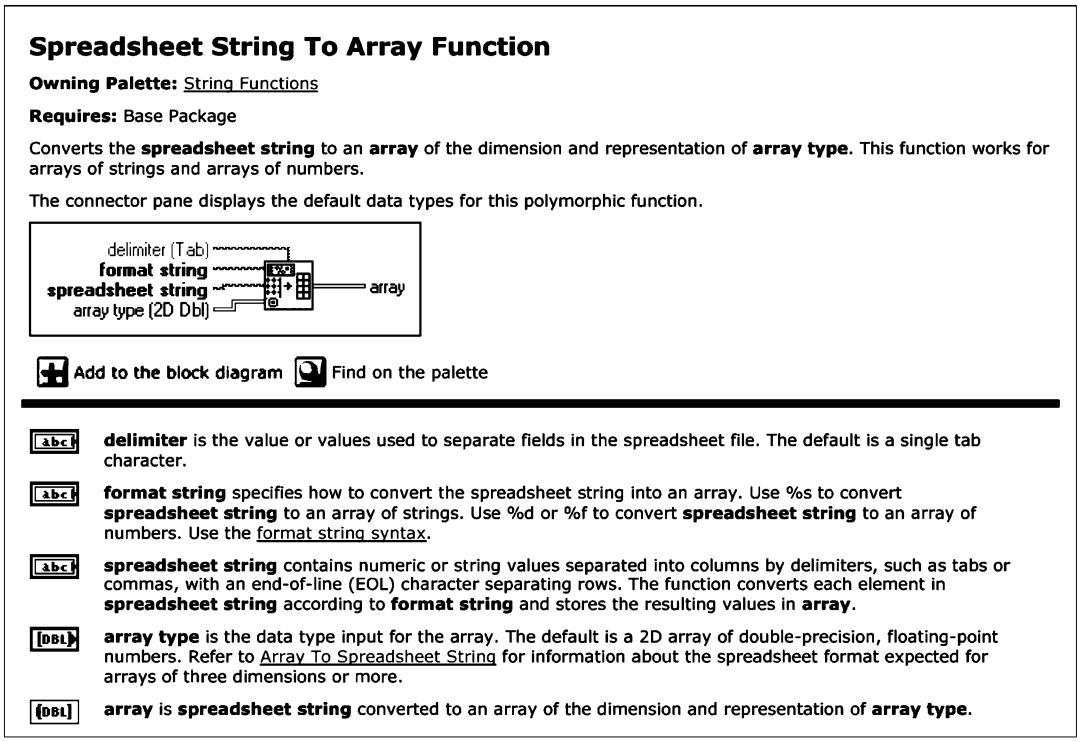


Figure 20.25: Help file for the `Spreadsheet String To Array` function

two figures look nearly identical. The only difference between them is in the data type of the queue reference input and array input tunnels. One of the benefits of string-based queues is that all string-based queues and string-based arrays have the same data type—no matter what parsing convention is used. This is not true for clusters. Clusters with different elements have different data types.

This has a big impact on code reusability. If we develop a useful sub-VI that operates on a string-based queue, we can use it on *any* string-based queue. For example, Figure 20.22 can be simplified by replacing the `String To Array` sub-VI and the

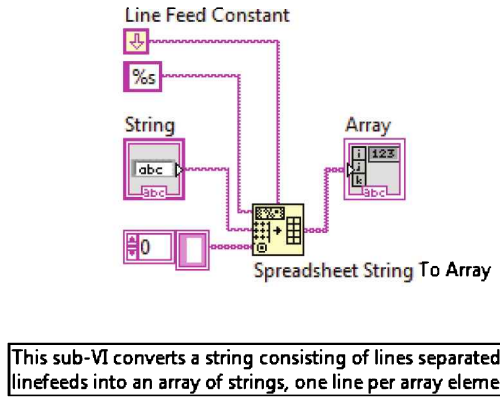


Figure 20.26: Block diagram of the `String to Array` sub-VI

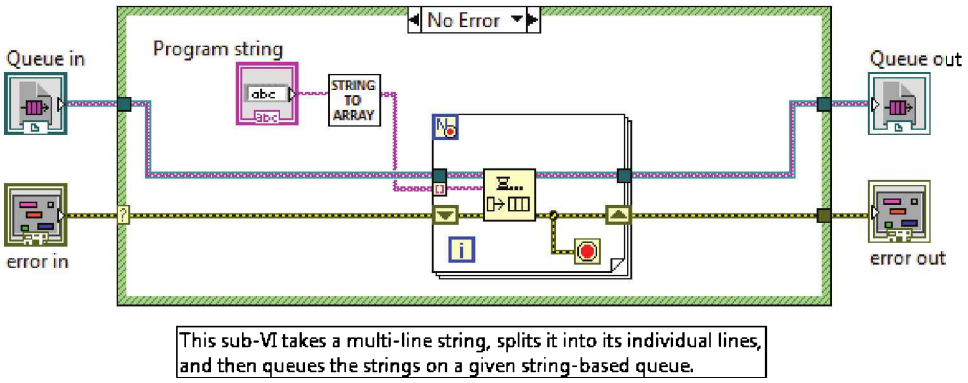


Figure 20.27: Block diagram for the `Split And Queue String` sub-VI



Figure 20.28: Icon for the `Split And Queue String` sub-VI

For loop with a reusable `Split And Queue String` sub-VI. The block diagram for this sub-VI is shown in Figure 20.27.

The icon of the `Split and Queue String` sub-VI is shown in Figure 20.28, and its front panel is shown in Figure 20.29. Notice that an output terminal for the queue reference has been included in this sub-VI. This terminal is not needed by the `Multitest` VI, but it is a good programming practice to include reference output terminals for sub-VIs with reference input terminals. This increases the reusability of the sub-VI and allows the sub-VI to be chained with other VIs that use the queue reference if the need arises.

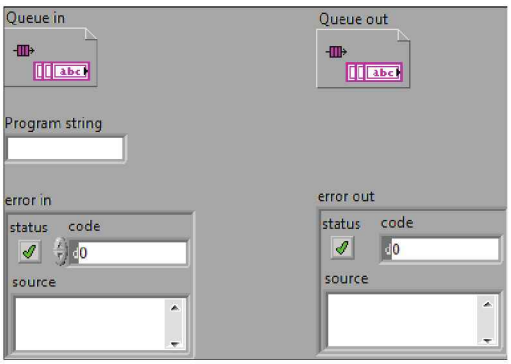


Figure 20.29: Front panel for the `Split And Queue String` sub-VI

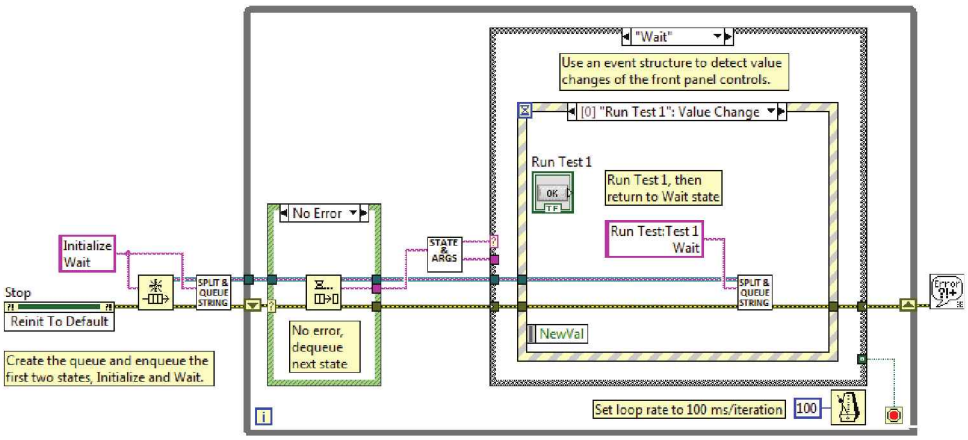


Figure 20.30: Block diagram of the `Multitest` VI using the `Split and Queue String` sub-VI

Wiring Tip: The queue reference control and indicator seen in Figure 20.29 are not found in the the standard LabVIEW palette of controls. To create the queue reference control, right-click on the queue reference input of one of the the native LabVIEW queue functions and select *Create. . . Control*. This will create a string-based queue reference control on the front panel. You can then copy the control and change it into an indicator by right-clicking and selecting *Change to Indicator*. Alternatively, you can right-click on the queue reference output of a queue function on the block diagram and select *Create. . . Indicator*.

The `Split and Queue String` sub-VI can now be used to simplify the `Multitest` VI. The event case handling the value change of the RUN TEST 1 button is shown in Figure 20.30. The multiline string within the event case is much easier to

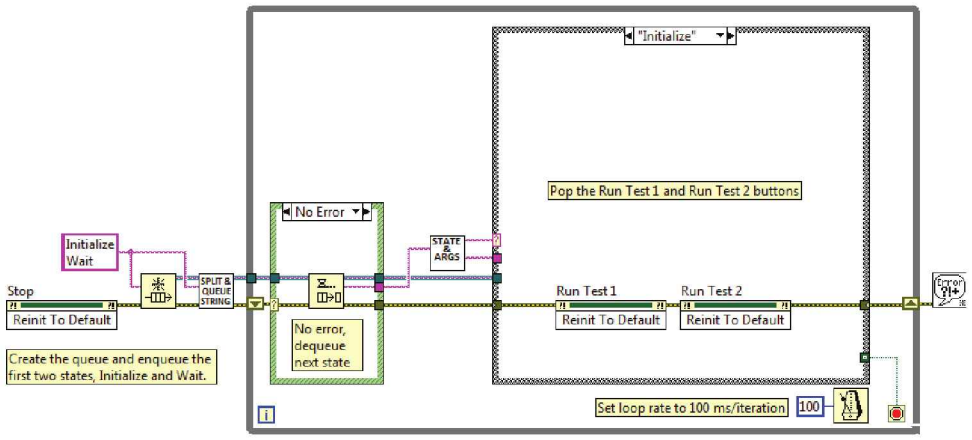
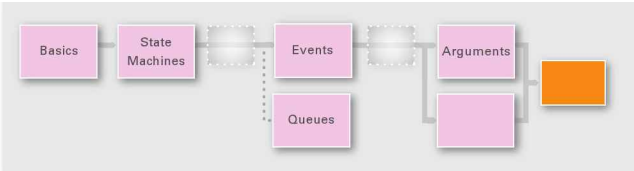


Figure 20.31: *Initialize* state of the `Multitest` VI using the `Split and Queue String` sub-VI

read and edit than an array of string constants. The `Split and Queue String` sub-VI suggests that the contents of the string constant are placed on the queue, hiding the low-level details of how the string is split and enqueued and how the errors are trapped. This increases the readability of the top-level VI. The `Split and Queue String` sub-VI can be opened and examined if the reader is interested in the details of how this is accomplished.

Notice too that the `Split and Queue String` sub-VI has been used in the input area of the loop. In Figure 20.13, the *Initialize* state is enqueued outside the main loop, and the *Wait* state is enqueued in the *Initialize* state. This has been done to save space in the input area of the loop, allowing the loop to be made larger and still fit on one screen. In Figure 20.30, the `Split and Queue String` sub-VI is used outside of the main loop instead of the native **`Enqueue Element`** function, allowing both the *Initialize* and the *Wait* state to be enqueued without taking up extra space. This increases the readability of the `Multitest` VI. The first two states of the `Multitest` VI are always *Initialize* and *Wait*. Placing these two states in the input area of the loop makes this fact obvious to the reader.

The new version of the *Initialize* state is shown in Figure 20.31. Compare this figure with Figure 20.13. The *Initialize* state no longer needs to queue the *Wait* state, since it is now queued in the input area of the loop.



21

Car Wash Controller with Arguments

In the previous chapter, we looked at how to pass arguments to individual states in a queued state machine. We did this by using a cluster as the queue element and a string as the queue element. Both approaches allowed us to combine the name of the target state with the data needed by that state. This allowed us to treat each state as a subroutine in a text-based language, calling the state by name and passing it the needed arguments at the same time. This resulted in replacing the *Run Test 1* and *Run Test 2* states with a generic *Run Test* state that accepted arguments.

The goal of this chapter is to develop a car wash controller with a generic *Cycle* state that replaces the *Under Body Wash*, *Soap Application*, *Main Wash*, *Rinse*, and *Air Dry* states. The *Cycle* state accepts a car wash cycle name and a time target as arguments. The name argument will be used to select the correct proximity switches and indicators. The time target argument is used by the car wash timer to regulate the cycle.

This upgrade improves the readability of the code, since the number of cycles are drastically reduced. It also improves maintainability. It is much easier to maintain a single *Cycle* state rather than five individual car wash states.

Another improvement is that the use of arguments promotes loose coupling. The classic state machine version of the controller developed in Chapter 8 did not use queues. Each state had to determine the transition to the next state. The value of the DELUXE WASH Boolean needed to be placed on the data highway so the *Lock Selector* state and the *Rinse* state could determine their state transitions. The car wash states were coupled to the type of wash selected by the user.

The queued controller developed in Chapter 14 fixed that problem by queueing all of the states needed to accomplish the desired wash as soon as the wash was selected. The DELUXE WASH Boolean was removed from the data highway, uncoupling

the car wash state transitions from the type of wash selected. The queued controller had looser coupling than the classic controller.

But even though the car wash state transitions were decoupled from the car wash type, the Deluxe and Economy Wash cycles were still coupled to each other in a more subtle way. The time targets of the three car wash states common to both the Deluxe and the Economy Washes (*Soap Application*, *Main Wash*, and *Rinse*) are the same in both the Deluxe and the Economy Washes. The Deluxe and Economy Wash cycles are coupled to each other through the time targets of these common states.

Suppose the owner of the car wash wanted the *Main Wash* state to run for eight seconds in the Deluxe Wash and for five seconds in the Economy Wash. The queued controller would not allow this option, since the cycle times are hardwired into the states themselves and the same *Main Wash* state is shared by both the Deluxe Wash and the Economy Wash. One way around this problem would be to replace the *Main Wash* state with two new states: a *Deluxe Main Wash* state that runs for eight seconds and an *Economy Main Wash* state that runs for five seconds. But adding more states is not desirable. We want to reduce the number of states.

A better solution is to pass the time target to each car wash cycle as an argument. This removes the hardwired target times from the cycle states and decouples the deluxe wash from the economy wash. The *Main Wash* state could then have different target times in the Deluxe and Economy Washes. The car wash states would be truly decoupled from the type of wash selected, and the Deluxe and Economy Wash cycles would be truly decoupled from each other.

21.1 Car Wash Controller with Time and Cycle Arguments

We looked at the use of both clusters and strings for passing arguments to individual states in a state machine in the previous chapter. Strings proved to have a number of advantages over clusters: optional arguments, improved readability, and sub-VIs (such as the `Split and Queue String` VI) that can be used in any program using a string-based queue. These advantages are important in the car wash controller, since two arguments will need to be passed: the cycle name and the time target. If a cluster were used, the cluster would need three elements: an enumerated constant to hold the state, a string to contain the cycle name, and a numeric to hold the time target. A single string can hold all three of these elements or just the state name if no arguments are needed.

This gives us a plan for developing a car wash controller with arguments. First we develop parsing sub-VIs that extract the state, car wash cycle type, and time target from the queued string. Then we develop a generic *Cycle* state that accepts these arguments and performs the tasks previously handled by the individual car wash cycle states. Finally, we modify the queued car wash controller from Chapter 14 to use queued strings rather than queued enumerated constants.

21.2 Developing Parsing Routines for the Car Wash Controller

In the previous chapter, we developed the `Parse State and Arguments` sub-VI to separate the state string from the arguments in the queued string version of the `Multitest` VI. This sub-VI took the queued string as an input and returned two separate strings as outputs: a state string and an argument string. The original string was destroyed in the process.

That approach isn't appropriate for the car wash controller. Remember that the car wash controller needs to requeue the car wash cycle state if the cycle isn't finished yet. This means that the original queued string (state plus arguments) needs to be placed back on the queue if the cycle isn't finished. Instead of breaking up the string into states and arguments, it makes more sense to read the state and arguments from the string while keeping the string intact. This makes it easier to requeue the original string if the cycle is not done.

We again need a convention to allow the arguments to be parsed from the state name. The "colon convention" used in the last chapter can be expanded for use in the car wash controller. The car wash controller string is formatted as `State Name:Cycle Name:Time Target`. To specify that a generic *Cycle* state is to run a *Rinse* cycle for five seconds, the command string is `Cycle:Rinse:5`.

In order to use this scheme with the car wash controller, we need two parsing sub-VIs—one to read the state from the queued string and another to read the arguments. The state is extracted from the queued string by developing a `Parse State` sub-VI. The icon for the sub-VI is shown in Figure 21.1. A non-square icon has been chosen to streamline the appearance of the block diagram. The sub-VI has a single input and a single output, so the short icon is practical in this case. It also gives the sub-VI an "in-line function" appearance rather than a "subroutine" appearance.

The block diagram of the `Parse State` sub-VI is shown in Figure 21.2. This sub-VI makes use of the native **`Match Pattern`** function to split the queued string at the first colon found. The substring before the colon is the desired state. The rest of the string is discarded. The process is illustrated by the front panel of the `Parse State`



Figure 21.1: Icon for the `Parse State` sub-VI

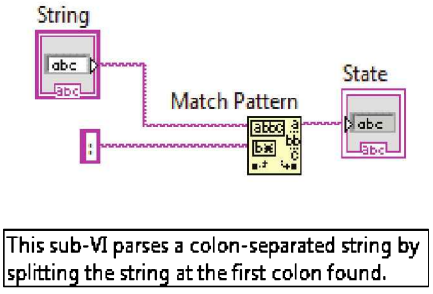


Figure 21.2: Block diagram for the `Parse State` sub-VI

VI shown in Figure 21.3. In this figure, the input string is `Cycle:Main Wash:5`, and the output string is `Cycle`, which is the desired state.

Next we need a sub-VI that will read the arguments from the queued string. This is accomplished by the `Parse Name and Time Target` sub-VI. The icon of this sub-VI is shown in Figure 21.4, and the block diagram is shown in Figure 21.5.

The `Parse Name and Time Target` VI uses the **`Match Pattern`** function to split the input string twice. The first split happens at the first colon and separates the state string from the arguments string. The state information is not needed by this sub-VI and is discarded. The argument string is then split again at the second colon. This creates a cycle name string and a string containing the target time for the cycle. The target time string is then converted to a floating-point number using the **`Fract/Exp String to Number`** function (see the help file in Figure 18.4). The parsing process is illustrated by the front panel of the `Parse Name and Time Target` VI shown in Figure 21.6. The input string is `Cycle:Air Dry:10`. The output cycle string is `Air Dry`, and the time target is `10`.



Figure 21.3: Front panel of the `Parse State` sub-VI



Figure 21.4: Icon of the `Parse Name and Time Target` VI

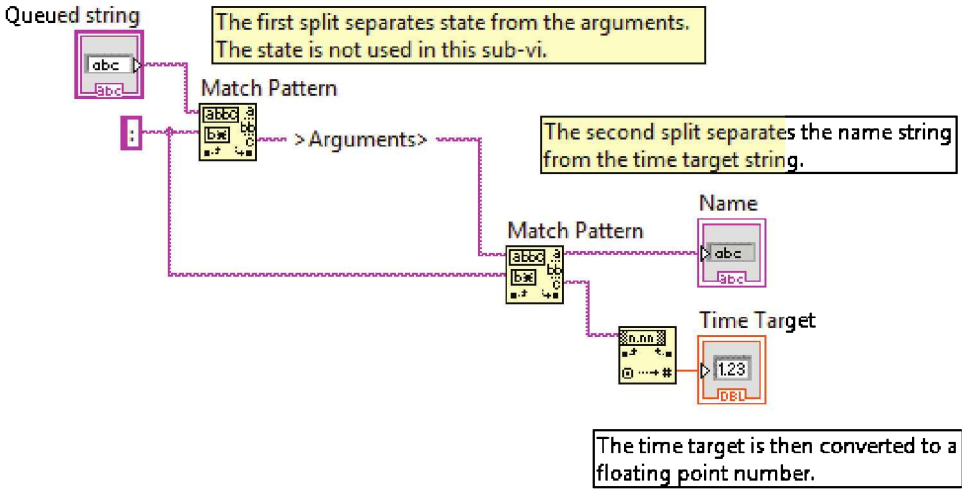


Figure 21.5: Block diagram of the `Parse Name and Time Target` VI



Figure 21.6: Front panel of the `Parse Name and Time Target` VI

21.3 Developing a Generic *Cycle* State

The following tasks are carried out in each of the five individual car wash cycle states in the following order:

1. Select the proper proximity switch from the *Switches* cluster.
2. Perform a time check using a hard-wired time target and the selected proximity switch.
3. Light the proper lamps in the *Indicators* cluster.
4. Reset the car wash timer if the state is complete, else requeue the current state.

In the queued car wash controller, the switch selection, time target, and indicator selections were all hard-wired in the individual car wash states, but the overall structure of the states was otherwise identical. This suggests a plan of attack for developing the generic *Cycle* state:

1. Read the cycle name and time target from the queued string.
2. Use the cycle name as an input to a mapping sub-VI that will select the proper switch from the *Switches* cluster.
3. Pass the time target and selected switch value to the car wash timer action engine.
4. Pass the output of the car wash timer and the cycle name to a second mapping sub-VI that will light the correct indicator lamps.
5. Use the car wash timer output to decide whether to requeue the current state or not.

Developing the Mapping Sub-VIs

The key to developing a generic cycle state are the two mapping sub-VIs. These sub-VIs will accept the cycle name argument as an input and will read the correct proximity switch and light the correct lamps. They are called "mapping" sub-VIs because they contain maps of the car wash set up. These sub-VIs know which switches are read by each cycle and which lamps are lit by each cycle.

The mapping sub-VIs are easy to develop, since they are essentially case structures that accept cycle name strings as selector inputs. The icon of the `Car Wash Switch Map` sub-VI is shown in Figure 21.7. This sub-VI accepts the car wash simulation switches and the cycle name argument string as inputs, and it produces the value of the correct switch as an output.

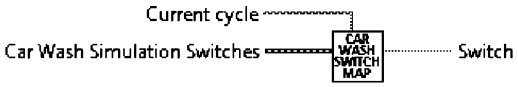


Figure 21.7: The icon for the the `Car Wash Switch Map` sub-VI

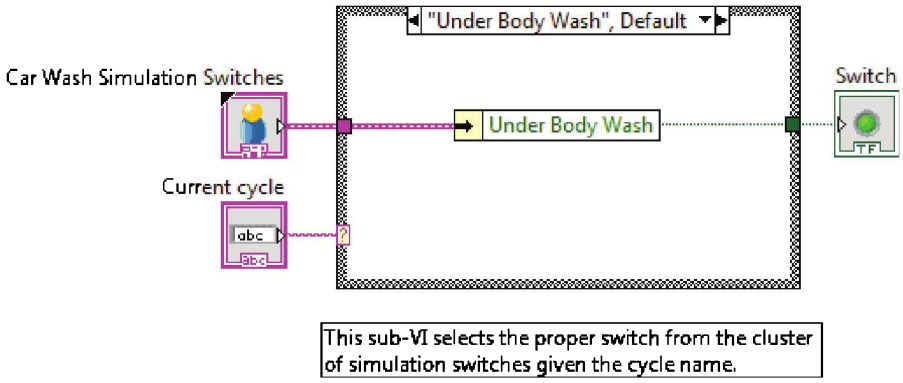


Figure 21.8: The block diagram for the `Car Wash Switch Map` sub-VI showing the *Underbody Wash* case

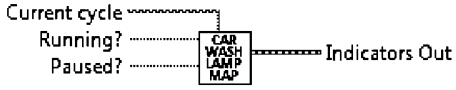


Figure 21.9: The icon and connector pane for the the `Car Wash Switch Map` sub-VI

The block diagram of the `Car Wash Switch Map` is shown in Figure 21.8. The cycle name string is wired to the selector of a case structure, and the switches cluster is passed as an input to the case structure. Each case of the structure contains an **`Unbundle By Name`** function that selects the correct switch from the cluster of proximity switches. Figure 21.8 shows the case for the *Under Body Wash* cycle. The value of the UNDER BODY WASH switch is unbundled from the switches cluster and is sent to the Boolean output. The other cases are similar, each case unbundling the switch appropriate for the given wash cycle. Notice that the *Under Body Wash* case has been made the default case of the structure. Since the input to the case structure is a string, one of the cases must be designated as a default case.

The indicators also need a mapping sub-VI that will light the correct lamps given the cycle name and the current status of the cycle. The icon of the `Car Wash Lamp Map` sub-VI is shown in Figure 21.9. The `Car Wash Lamp Map` sub-VI accepts the name of the current cycle, the Running? Boolean, and the Paused? Booleans as inputs and it outputs a copy of the indicator cluster with the appropriate lamps lit.

The block diagram for the *Underbody Wash* case is shown in Figure 21.10. A case structure uses the name of the current cycle as its selector input. A cluster constant of the car wash lamp defaults (all lamps dark) is an input to the case structure. Each case

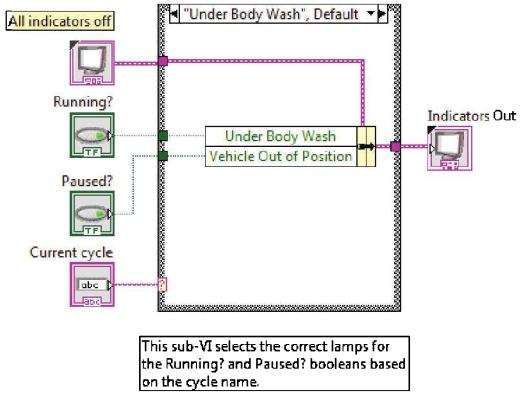


Figure 21.10: The block diagram for the `Car Wash Lamp Map` sub-VI showing the *Underbody Wash* case

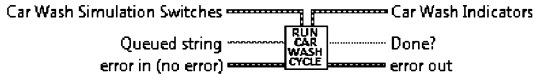


Figure 21.11: Icon for the `Run Car Wash Cycle` VI

uses the **`Bundle By Name`** function to bundle the value of the *Running?* Boolean to the appropriate cycle indicator and the value of the *Paused?* Boolean to the *Vehicle Out of Position* indicator. Since all of the indicators in the input cluster constant are set to False (dark), any indicators that may have been lit in previous cycles are automatically turned off every time the `Car Wash Lamp Map` function is called.

Developing the `Run Car Wash Cycle` Action Engine

Now that the mapping sub-VIs are developed, they can be combined with the car wash timer action engine to form a new sub-VI, the `Run Car Wash Cycle` action engine. This is the sub-VI that will do most of the work of the *Cycle* state. This sub-VI will be an action engine since it will serve as a wrapper for the car wash timer action engine.

The icon of the `Run Car Wash Cycle` action engine is shown in Figure 21.11. This sub-VI accepts the queued string and the switches cluster as inputs. The output of this sub-VI will be the indicator cluster and a *Done?* Boolean signalling if the the current cycle is done or not.

The block diagram of the `Run Car Wash Cycle` sub-VI is shown in Figure 21.12. This block diagram should look familiar, since it is very similar to the car wash cycle states seen in the queued car wash controller discussed in Chapter 14. The queued string input is parsed by the `Parse Name and Time Target` sub-VI developed previously in this chapter. The cycle name is passed to the `Car Wash Switch Map` along with the switches cluster to get the value of the appropriate proximity switch. This switch value is then passed to the car wash timer along with the time target, which was also read from the queued string input. The output of the car

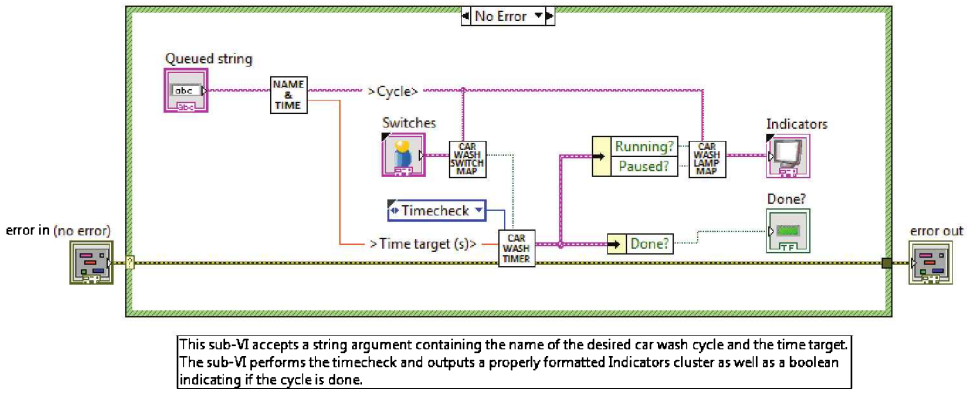


Figure 21.12: The block diagram of the `Run Car Wash Cycle` VI

wash timer is then passed to the `Car Wash Lamp Map` along with the cycle name. The indicators cluster and the value of the *Done?* Boolean are then passed as outputs of the `Car Wash Cycle` VI.

21.4 Implementing the Car Wash Controller with Arguments

The `Car Wash Cycle` action engine is a significant step forward. This sub-VI can now run any car wash cycle if the cycle name and time targets are given to it as inputs. The five individual wash cycle states can now be replaced with a single generic *Cycle* state in a queued-string implementation of the car wash controller that uses arguments. This version of the car wash controller has a number of new features, so we will examine each state of the controller carefully.

The *Initialize* state of the car wash controller with arguments can be seen in Figure 21.13. The overall structure of the state machine is similar to the queued controller developed in Chapter 14. The queue is created in the input area of the main loop, but this time the queue is given a string as its data type rather than an enumerated constant. The `Split and Queue String` sub-VI developed in Chapter 20 is then used to enqueue the first three states of the state machine: *Initialize*, *Unlock Selector*, and *Wait*.

This demonstrates one of the advantages of the string-based state machine. In the queued version of the car wash controller, these first three states were spread out within the state machine. The *Initialize* state was enqueued outside the main loop, and the *Unlock Selector* and *Wait* states were enqueued within the *Initialize* state. It would have taken up valuable screen real estate to enqueue all three of these states separately outside the main loop. In the queued-string implementation, we can put all three states into a single, multiline string using the `Split and Queue String` developed for the `Multitest` VI. This is a good use of space and a good example of code reuse. These first three states also show another advantage of strings. None of these states

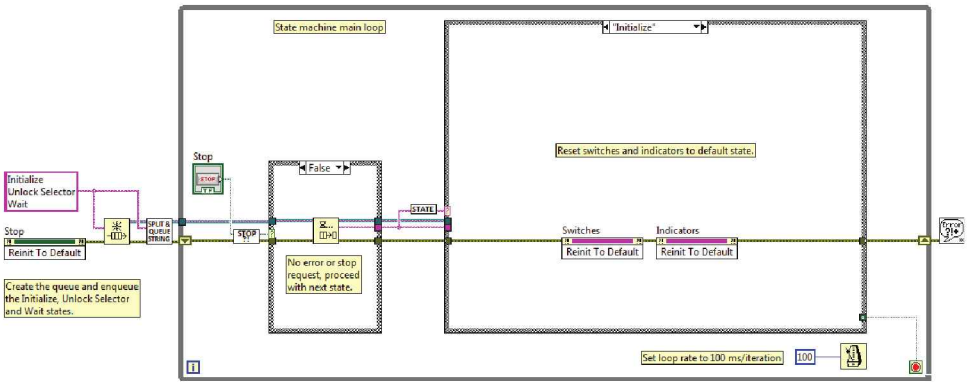


Figure 21.13: *Initialize* state for the string-based car wash controller

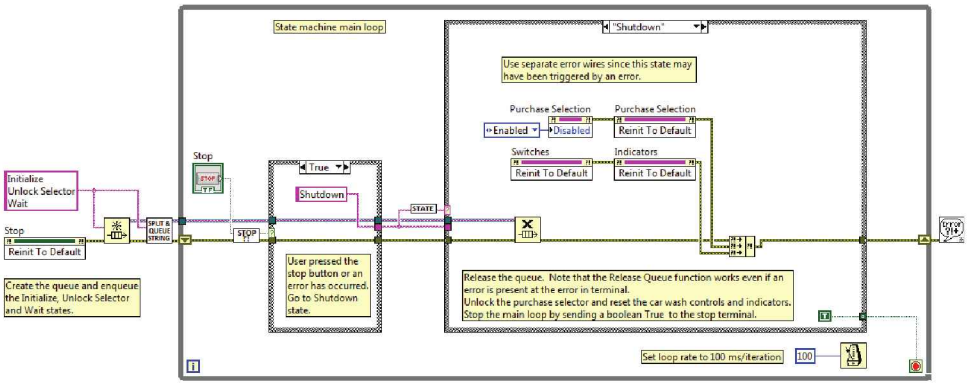


Figure 21.14: *Shutdown* state for the string-based car wash controller

need arguments, and none are provided in the string constant. The constants are optional in the chosen parsing scheme.

Moving inside the main loop, the first code to execute is the guard clause case structure. If the user has pressed the STOP button or an error is present, the guard clause bypasses the queue and sends the state machine directly to the *Shutdown* state, as seen in Figure 21.14. The *Shutdown* state releases the queue, resets the front panel controls and indicators, and stops the main loop.

Returning to Figure 21.13, if the STOP button has not been pressed and no error is present, the next string is dequeued. The next state is read from the string by the `Parse State` sub-VI. The original string is passed intact to the case structure as an input. The state machine then proceeds to dequeue the first three states enqueued outside the main loop. The *Initialize* resets the front panel controls. The *Unlock Selector* state shown in Figure 21.15 then unlocks and initializes the purchase selector. The state machine then moves to the *Wait* state and waits for user input.

The *Wait* state contains an event structure with three event cases that detect the value change of the DELUXE WASH, ECONOMY WASH, and STOP buttons. The event case

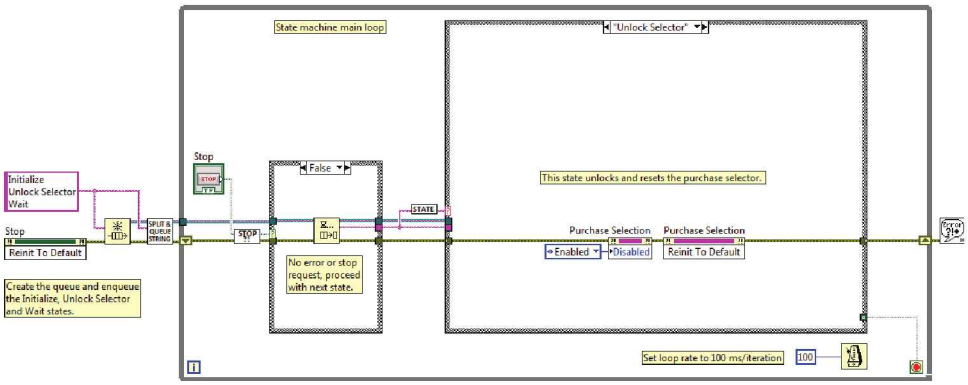


Figure 21.15: The *Unlock Selector* case of the string-based car wash controller

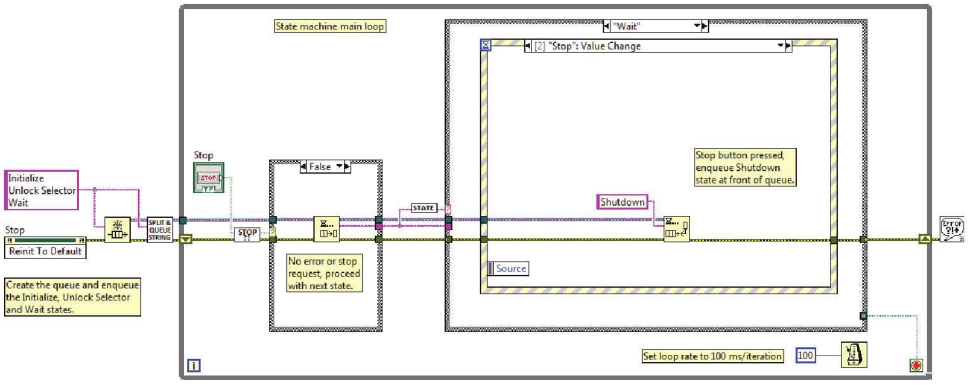


Figure 21.16: Event case handling the value change of the STOP button for the string-based car wash controller

handling the value change of the STOP button is the simplest one and is shown in Figure 21.16. This event case queues the *Shutdown* state at the front of the queue for immediate execution using the **`Enqueue Element At Opposite End`** function.

The event case handling the value change of the DELUXE WASH button is shown in Figure 21.17. The entire Deluxe Wash sequence is contained in a single multiline string constant that is enqueued by the `Split and Queue String` VI. Notice how readable this particular figure is. The first three states and the entire Deluxe Wash sequence are each contained in string constants. Each string constant is almost like a fragment of text-based pseudocode. In a single glance, the reader can see the entire start-up routine and the entire Deluxe Wash sequence of states, including the target times for each cycle state. This improves the readability of the code, making it practically self-documenting. The same readability is evident in the event case handling the value change of the ECONOMY WASH button shown in Figure 21.18.

Maintainability and scalability are also improved because the string-based version of the car wash controller is easier to edit and extend. In Chapter 14, the sequence

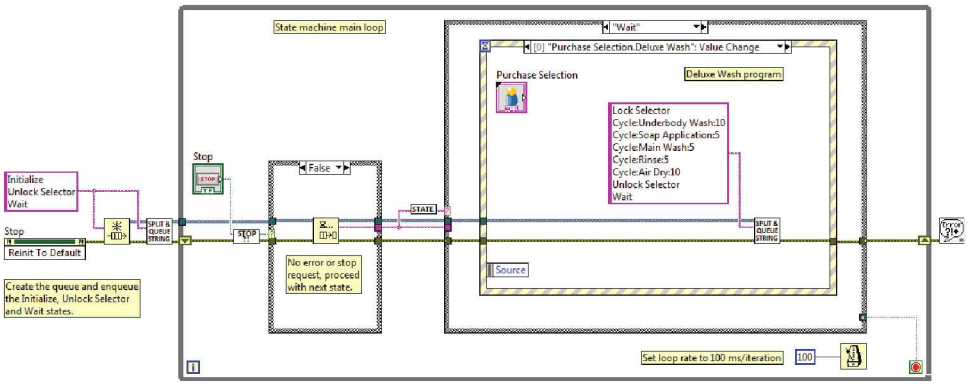


Figure 21.17: The event case for handling the value change of the DELUXE WASH button for the string-based car wash controller

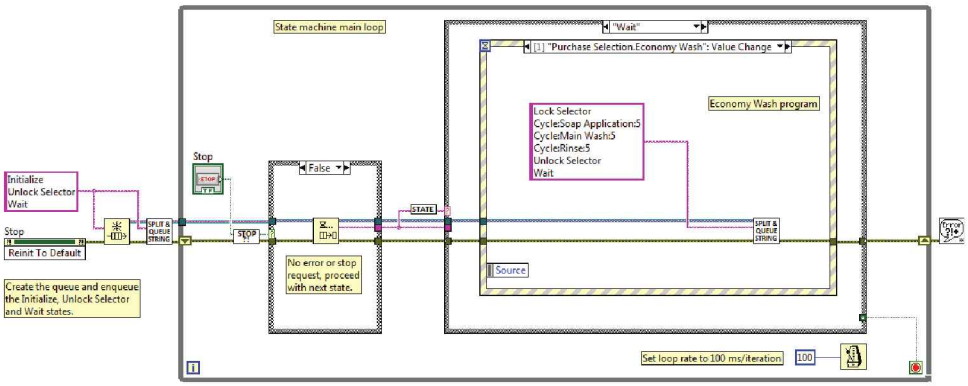


Figure 21.18: The event case for handling the value change of the ECONOMY WASH button for the string-based car wash controller

of states was contained in an array of enumerated constants. Inserting, deleting, or rearranging elements of an array constant is awkward. It is much easier to edit a string constant. The string constant also makes it easier to implement a plug-in architecture (if desired). Instead of hard-coding the deluxe and economy wash string constants into the program, they can be read from a text file on the user's computer.

Returning to the sequence of states, both the Deluxe Wash and the Economy Wash start with the *Lock Selector* state shown in Figure 21.19. This state locks the purchase selector and resets the timer prior to starting the first car wash cycle.

The sequence of car wash cycle states starts after the purchase selector has been locked. The *Cycle* state of the car wash controller is shown in Figure 21.20. The queued string containing the arguments and the cluster of proximity switches are passed to the `Car Wash Cycle` action engine. This sub-VI reads the arguments and runs the cycle.

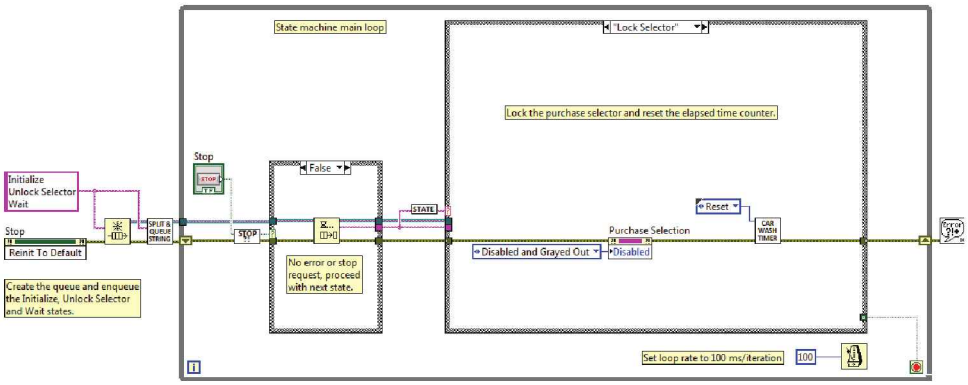


Figure 21.19: The *Lock Selector* state of the string-based car wash controller

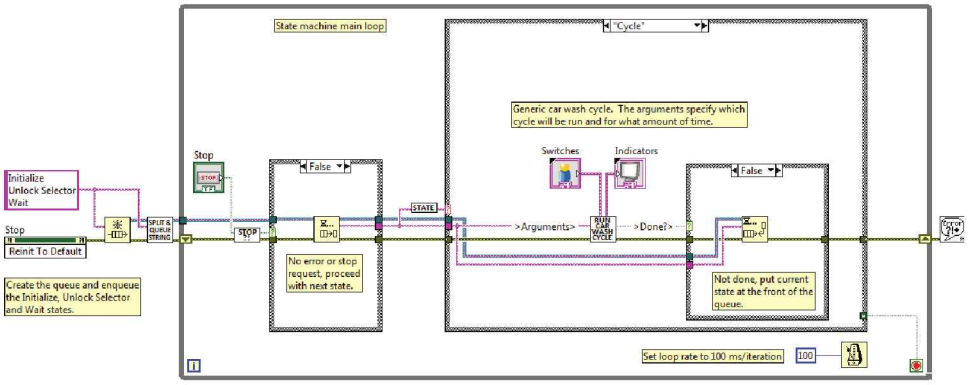


Figure 21.20: The *Cycle* state of the string-based car wash controller showing the *Not Done* case.

This sub-VI selects the appropriate proximity switch, performs a time check, and lights the appropriate indicators. The sub-VI outputs a *Done?* Boolean that is wired to the selector of a case structure that decides whether the cycle is done or not. If the cycle is not done, the queued string is replaced at the front of the queue for immediate execution, as seen in Figure 21.20. If the cycle is done, the timer is reset, as shown in Figure 21.21.

One obvious benefit of the *Cycle* state is that it replaces five other states. A second, more structural, benefit is that it removes the proximity switch cluster from the lobby and the indicator cluster from the alley of the state machine. In the previous implementations of the car wash controller, these clusters were needed by five states. This meant that the switches and indicators were global in scope and needed to be placed in the lobby and alley of the state machine. By replacing the five cycle states with a single *Cycle* state, we have reduced the scope of these two clusters. Now the

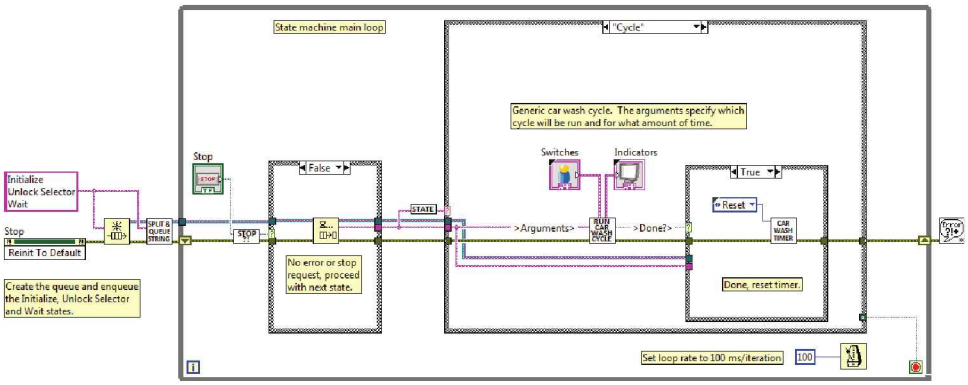


Figure 21.21: The *Cycle* state of the string-based car wash controller showing the *Done* case

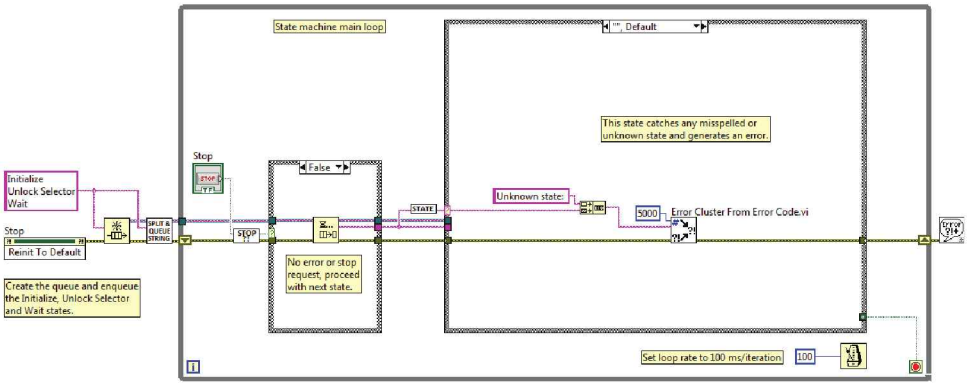
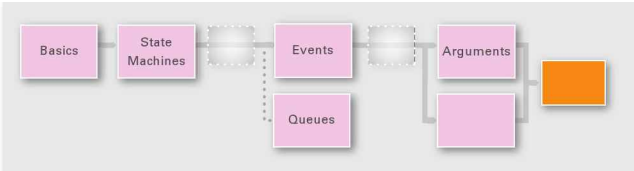


Figure 21.22: The default case of the string-based car wash controller

switches and indicators are needed by only one state and their scope is local. This allows us to pull them in from the lobby and alley and put them into the *Cycle* state. This increases the maintainability of the code, especially since the indicators are no longer in the alley. Recall that outputs of case structures need to be wired in every case. Moving the indicator cluster to the *Cycle* state removes one of the output tunnels of the case structure. This simplifies every case in the case structure and allows us to remove the default indicator cluster from the input area of the main loop. In order to make sure that the indicator cluster starts out with all lamps dark, the cluster is reinitialized to its default value in the *Initialize* state, as seen in Figure 21.13.

As mentioned in Chapter 20, string-based state machines should have a default state designed to catch unknown states due to spelling errors. The default state of the string-based controller is shown in Figure 21.22. The unknown state string is captured and incorporated into the source string of a custom error cluster. The custom error triggers the *Shutdown* state, and the error is reported to the user by the **`General Error Handler`** function.



22

Problem Set 8

22.1 DAQ State Machine Exercises

Exercise 8.1: String-Based DAQ State Machine

Convert the DAQ state machine developed in Exercise 5.10 to use a string-based queue. No arguments are necessary, simply replace the enumerated state constants with strings. Use the `Split and Queue String` sub-VI shown in Figure 20.27 to improve readability and conserve screen real estate. What are the advantages and disadvantages of using strings instead of enumerated constants in this case? ■

22.2 Car Wash Controller Exercises

Exercise 8.2: Cluster-Based Car Wash Controller with Arguments

String-based queues were used in Chapter 21 because it was claimed that cluster-based queues with arguments would be more awkward and less readable, scalable, and maintainable. These claims were made without proof. Develop a version of the car wash controller with arguments that uses a cluster-based queue similar to the cluster-based `Multitest` VI with arguments developed in the previous chapter. Compare and contrast the cluster-based version you develop with the string-based version developed in Chapter 21. ■

Exercise 8.3: Read the Car Wash Programs from Text Files

Currently, the string-based car wash controller uses string constants in the event structure to specify the sequence of steps taken when the user selects a Deluxe or Economy car wash. Modify the string-based car wash controller to read these sequences of steps from text files. Assume that the text files have the names `Deluxe.txt` and

`Economy.txt` and are stored in the same directory as the main VI. The text files should be read when the controller first starts and should be persistently stored thereafter. You might find the **`Read from Text File`** function useful (see the help file in Figure 16.6). ■

22.3 Traffic Light Controller Exercises

Exercise 8.4: Traffic Light Mapping Functions

Create a `Signal Map` sub-VI that accepts a string containing the name of a traffic light state as an input, as well as inputs representing the duration of green/red lights and the duration of yellow lights. The sub-VI should output the traffic light cluster and the signal duration that corresponds to the input traffic light state name. For instance, if the sub-VI receives the inputs `N-S Green` with green/red signal durations of four seconds and a yellow signal duration of two seconds, the output traffic light cluster should have the north and south signals green, the east and west signals red, and all left turn signals red. The output duration should be four seconds. The icon of the `Signal Map` sub-VI is shown in Figure 22.1.

Create a `Next Traffic State String` sub-VI that accepts a string containing the name of a traffic light state as an input, as well as an input of the bundle of left-turn sensors. The sub-VI should return a string containing the name of the next traffic light state. The icon of the `Next Traffic State String` sub-VI is shown in Figure 22.2.

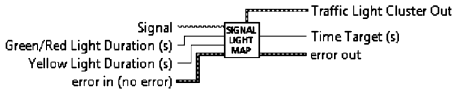


Figure 22.1: The icon of the `Signal Map` sub-VI

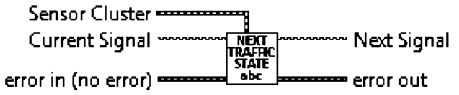


Figure 22.2: The icon of the `Next Traffic State String` sub-VI ■

Exercise 8.5: Traffic Light Controller with Arguments

Create a traffic light controller that uses arguments to replace the eight specific signal states (*N-S Green*, *E-W Yellow*, etc.) with a single generic *Signal* state. The generic *Signal* state should use the mapping functions developed in the previous exercise to set the traffic light pattern and to determine the next state. ■

22.4 Security System Controller Exercises

Exercise 8.6: Security System Controller with Arguments

In Problem Set 16, you placed the `Update Log` sub-VI in a new *Update Log* state. A consequence of this action is that the old zone status needs to be stored persistently. In

Exercise 5.2, you developed a security system controller that places both the old and new zone status clusters on the data highway. In Exercise 5.4, you created the `Update Log Action Engine` to store the old zone status internally. Another solution to this problem is to place only the new zone status on the data highway and to pass the old zone status as an argument to the *Update Log* state.

Revise the security system controller developed in Exercise 5.3 to pass the old zone status cluster as an argument to the *Update Log* state instead of placing it on the data highway. Create a type-defined cluster containing the state enumerated constant and the zone status array, and use this cluster as the data type of the queue. When a value change of the switches is detected, place the new zone status array on the data highway, bundle the old zone status cluster with the state enumerated constant, and place the cluster on the queue. ■

Exercise 8.7: Reversing the Arguments in the Security System Controller

In Exercise 8.6, you were asked to create a version of the queued, event-driven security controller that placed the new zone status array on the data highway and passed the old zone status array to the *Update Log* state as an argument. Create a version of the security system with the arguments reversed: Keep the old zone status array on the data highway and pass the new zone status array as an argument. ■

Exercise 8.8: Comparing the Security System Controllers with Arguments

Is there any functional difference between the controller developed in Exercise 8.6 and the one developed in Exercise 8.7? Is there any compelling reason to choose an argument-based implementation of the security system controller over the version developed in Exercise 5.2? ■

22.5 Boiler Controller Exercises

Exercise 8.9: String-Based Boiler Controller

In Exercise 5.5, you developed a version of the boiler controller that used a queue of enumerated constants to dictate the state transitions. Revise this program to use a string-based queue instead. Compare the two queue-based implementations. What are the advantages and disadvantages of each implementation? ■

The move to a string-based queue for the boiler controller in Exercise 8.9 was primarily cosmetic, although it did make it easier to edit the sequence of states. In this exercise, you take the use of strings a step further and add arguments and a parsing scheme to the boiler controller. The prior implementations of the boiler controller used

a cluster of constants to contain important process parameters, such as the purge time, flame sensor lower limit, and the upper and lower fuel-control limits. This scheme is not ideal. One problem is that the cluster of constants is available to every state in the state machine, while the individual constants in the cluster might only be needed by one or two states. The constants are unnecessarily exposed. Another problem is that the constants cannot be changed unless the block diagram of the controller is changed.

Both of these issues can be addressed by adding the concepts of arguments to the boiler controller. Instead of concentrating the constants into a single cluster, distribute the constants in the program string so that they become arguments to the states that need them. This reduces the exposure of the data. The only states that have access to these constants are the states that need them, yet the constants are all defined in one central location of the program. In addition, it is easy to imagine a version of the boiler controller that would read the program string from a text file rather than have it hard-coded on the block diagram. This would allow the user to change the process parameters in the text file without changing the block diagram of the code.

In the following exercises, you will modify the boiler sub-VIs that need information from the LIMITS cluster. Instead of unbundling the needed values from the LIMITS cluster, these sub-VIs parse the arguments from the queued string. You will then incorporate these modified sub-VIs into the string-based boiler controller and eliminate the LIMITS cluster altogether.

Exercise 8.10: Create the `Parse State and Argument Array` Sub-VI

An interesting feature of the problem of adapting the boiler controller to use arguments is that some states use no arguments, some use one argument, and some use two arguments. Develop the `Parse State and Argument Array` with the icon shown in Figure 22.3. This sub-VI accepts a string of the format `State:Argument 1: ...:Argument n` and returns the state string and an array of the argument strings in the order that they appeared in the original string. This sub-VI be used by the individual boiler steps to extract the argument array from the queued string.

One way to do this is to use the `Parse State and Argument` sub-VI shown in Figure 20.12. Use this sub-VI inside of a loop to parse the input string repeatedly until the argument string is empty, collecting the parsed arguments in an array. The state then can be deleted from the array using the **`Delete From Array`** function. The help file is shown in Figure 22.4, leaving just an array of argument strings.



Figure 22.3: The icon of the `Parse State and Argument Array` sub-VI ■

Exercise 8.11: Create the `Get DBL Arguments` Sub-VI

Create the `Get DBL Arguments` sub-VI with the icon shown in Figure 22.5. This sub-VI accepts the array of string arguments created by the `Parse State and`

Delete From Array Function

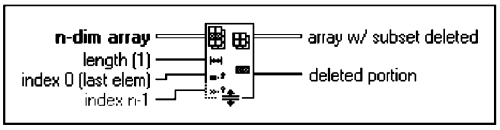
Owning Palette: Array Functions

Requires: Base Package

Deletes an element or subarray from **n-dim array** of **length** elements starting at **index**. Returns the edited array in **array w/ subset deleted** and the deleted element or subarray in **deleted portion**.

When you wire an array to this function, the function resizes automatically to display **index** inputs for each dimension in the array.

Details



Add to the block diagram Find on the palette

n-dim array is the array from which you want to delete element(s), row(s), column(s), page(s), and so on. This input can be an n-dimension array of any type.

length determines how many elements, rows, columns, or pages to delete. The default length is one element.

index 0..n-1 specifies what you want to delete from the array, such as an element, row, column, or page. Array functions in LabVIEW access array data in row-major order. In a 2D array, row is the first, major index. Column is the last, minor index. In bigger multidimensional arrays, column remains the last index and the more major indexes are added to the front. Thus, vol is the first index, followed by page, row, and column. These names are index identifiers and have no other meaning.

The default is the index of the last element in the array. You can wire only one **index** input. The index input can be positive or negative.

array w/ subset deleted is the array returned with the deleted element(s), row(s), column(s), or page(s).

deleted portion is the deleted element or array. If you wire a value to **length**, **deleted portion** is an array of the same number of dimensions as **n-dim array** containing all the elements deleted from **n-dim array**. The outer dimension size of **deleted portion** is the value of **length**, and the inner dimension size(s) match that of **n-dim array**. For example, if you wire a 3D array of dimensions 10x4x6 to **n-dim array** and a value of 2 to **length**, and do not wire values to the index inputs, **deleted portion** is a 3D array of dimensions 2x4x6 containing the last 2 pages from **n-dim array**.

If you wire a value to **length** and a negative value to the **index**, **deleted portion** is an array with an outer dimension size of the value of **length** subtracted by the absolute value of the **index**.

If you do not wire a value to **length**, **deleted portion** is an array of one less dimension than **n-dim array** containing the portion of the array deleted from **n-dim array**. For example, if you wired a 2D array of dimensions 8x5 to **n-dim array** and do not wire a value to **length**, and wire 3 to **index 0** (row), **deleted portion** is a 1D array containing all elements from row 3 of **n-dim array**.

Delete From Array Details

This function reduces the array in one dimension only, therefore, you can wire only one **index** input. For example, to delete a row in a 2D array, wire only the **row** index. To delete a column, wire only the **column** index. You can delete multiple consecutive subarrays at once by wiring **length**.

Figure 22.4: Help file for the `Delete From Array` function

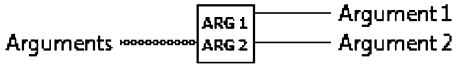


Figure 22.5: The icon of the `Get DBL Arguments` Sub-VI

`Argument Array` sub-VI shown in Exercise 8.10 as an input and returns the first two elements of the array as double-precision, floating-point numbers. ■

Exercise 8.12: Modify the `Pre-Purge Timer Done` Sub-VI

Modify the `Pre-Purge Timer Done` sub-VI developed in Exercise 1.23 to accept a string argument instead of accessing the LIMITS cluster. The icon of the `Pre-Purge Timer Done Arguments` sub-VI is shown in Figure 22.6. Use the `Parse State and Argument Array` sub-VI developed in Exercise 8.10 and the `Get DBL Arguments` sub-VI developed in Exercise 8.11 to parse the pre-purge time limit from the queued string.

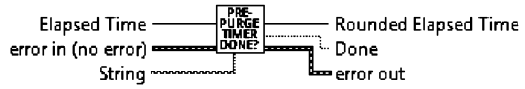


Figure 22.6: The icon of the `Pre-Purge Timer Done Arguments` sub-VI ■

Exercise 8.13: Modify the `Ignition Done` sub-VI

Modify the `Ignition Done` sub-VI developed in Exercise 1.27 to accept a string argument instead of accessing the LIMITS cluster. The icon of the `Ignition Done Arguments` sub-VI is shown in Figure 22.7. Use the `Parse State and Argument Array` sub-VI developed in Exercise 8.10 and the `Get DBL Arguments` sub-VI developed in Exercise 8.11 to parse the low-flame sensor limit from the queued string.

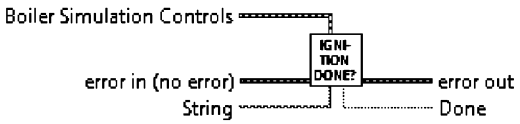


Figure 22.7: The icon of the `Ignition Done Arguments` sub-VI ■

Exercise 8.14: Modify the `Start Done` Sub-VI

Modify the `Start Done` sub-VI developed in Exercise 1.31 to accept a string argument instead of accessing the LIMITS cluster. The icon of the `Start Done Arguments` sub-VI is shown in Figure 22.8. Use the `Parse State and Argument`

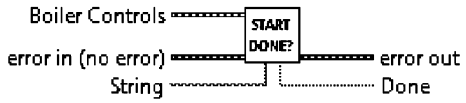


Figure 22.8: The icon of the `Start Done Arguments` sub-VI

Max & Min Function

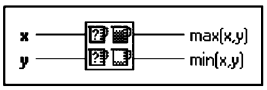
Owning Palette: Comparison Functions

Requires: Base Package

Compares **x** and **y** and returns the larger value at the top output terminal and the smaller value at the bottom output terminal. This function accepts time stamp values if all inputs are time stamp values. If the inputs are time stamp values, the function returns the later time at the top and the earlier time at the bottom. You can change the comparison mode of this function.

The connector pane displays the default data types for this polymorphic function.

Details Examples



Add to the block diagram Find on the palette

x and **y** must be of the same type.

max(x, y) is the larger value. When you compare arrays in Compare Elements mode (default), this function compares corresponding elements in each input array and returns the element with the maximum value. **max(x, y)** is an array consisting of the maximum value elements. When you compare arrays in Compare Aggregates mode, this function compares corresponding elements in each input array starting at the beginning of the array. When this function reaches a pair of unequal corresponding elements, **max(x, y)** returns the array that has the greater value for that element.

min(x, y) is the smaller value. When you compare arrays in Compare Elements mode (default), this function compares corresponding elements in each input array and returns the element with the minimum value. **min(x, y)** is an array consisting of the minimum value elements. When you compare arrays in Compare Aggregates mode, this function compares corresponding elements in each input array starting at the beginning of the array. When this function reaches a pair of unequal corresponding elements, **min(x, y)** returns the array that has the smaller value for that element.

Max & Min Details

Behavior When Comparing NaN Elements

In Compare Elements mode, when one or both inputs are NaN (Not-a-Number), this function produces the following results:

- Both inputs are NaN—Both **max(x,y)** and **min(x,y)** return NaN
- Only one input is NaN—Both **max(x,y)** and **min(x,y)** return the non-NaN value
- Inputs are arrays—The function evaluates each pair of elements according to the previous rules

In Compare Aggregates mode, when one or both inputs are NaN (Not-a-Number), this function produces the following results:

- Both inputs are NaN—Both **max(x,y)** and **min(x,y)** return NaN
- Only one input is NaN—**max(x,y)** returns **x** and **min(x,y)** returns **y**
- Inputs are arrays—If the function encounters a NaN element in either input array, **max(x,y)** returns **x** and **min (x,y)** returns **y**

Figure 22.9: Help file for the `Max & Min` function

`Array` sub-VI developed in Exercise 8.10 and the `Get DBL Arguments` sub-VI developed in Exercise 8.11 to parse the high and low fuel-control limits from the queued string. Add input-checking capability to the sub-VI to ensure that the higher of the two parsed numbers is assigned to the high fuel control limit and the lower of the two parsed numbers is assigned to the low fuel control limit. You might find the **`Max & Min`** function useful (see the help file in Figure 22.9). ■

Exercise 8.15: Modify the `Run Done` Sub-VI

Modify the `Run Done` sub-VI developed in Exercise 1.33 to accept a string argument instead of accessing the LIMITS cluster. The icon of the `Run Done Arguments` sub-VI is shown in Figure 22.10. Use the `Parse State and Argument Array` sub-VI developed in Exercise 8.10 and the `Get DBL Arguments` sub-VI developed in Exercise 8.11 to parse the high and low fuel-control limits from the queued string. Add input-checking capability to the sub-VI to ensure that the higher of the two parsed numbers is assigned to the high fuel-control limit and the lower of the two parsed numbers is assigned to the low fuel-control limit. You might find the **`Max & Min`** function useful (see the help file in Figure 22.9).

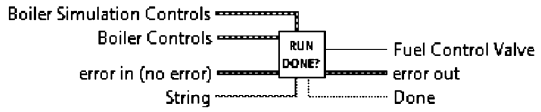


Figure 22.10: The icon of the `Run Done Arguments` sub-VI ■

Exercise 8.16: Modify the `Purge Done` Sub-VI

Modify the `Purge Done` sub-VI developed in Exercise 1.35 to accept a string argument instead of accessing the LIMITS cluster. The icon of the `Purge Done Arguments` sub-VI is shown in Figure 22.11. Use the `Parse State and Argument Array` sub-VI developed in Exercise 8.10 and the `Get DBL Arguments` sub-VI developed in Exercise 8.11 to parse the purge time limit from the queued string.

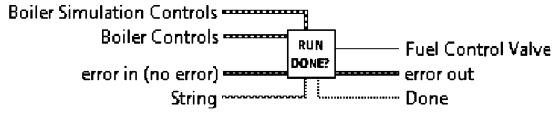


Figure 22.11: The icon of the `Purge Done Arguments` sub-VI ■

Exercise 8.17: Developing the Boiler Controller with Arguments

Now put all of the pieces together to develop the boiler controller with arguments. Revise the boiler controller with strings developed in Exercise 8.9, remove the LIMITS cluster, and distribute its contents as arguments to the states that need them. ■

22.6 Car Wash 2 Controller Exercises

In the following exercises, you will revise the queued Car Wash 2 controller developed in Exercise 5.7. The first step is to create a string-based Car Wash 2 controller with

time arguments to remove the hard-coded time target constants in the option states. Then mapping functions are developed and the eight option states are replaced with a single *Option* state in the same manner that the generic *Cycle* state replaced the five wash states in the original car wash controller.

Exercise 8.18: Developing a Parsing Sub-VI for the Car Wash 2 Controller with Time Arguments

The string-based Car Wash 2 controller with time arguments uses the `Option Name:Time Target` convention to enqueue the state and the time target in a single string. Develop the `State And Time Target` sub-VI with the icon shown in Figure 22.12. This sub-VI accepts an `Option Name:Time Target` string as an input. It parses the string into the option name and time target strings and converts the time target to a double-precision number.

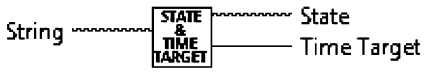


Figure 22.12: The icon of the `State And Time Target` sub-VI ■

Exercise 8.19: Create the `Build Option String Array` Sub-VI

Revise the `Build Wash Array` sub-VI developed in Exercise 5.6 to create the `Build Option String Array` with the icon shown in Figure 22.13. This sub-VI accepts the cluster of wash option buttons and a multiline string as inputs. The multiline string contains all of the available options and their time targets as arguments, as seen in Figure 22.14. The sub-VI splits the string into separate lines and creates a string array of those lines. The WASH ARRAY output array contains the option name and time argument strings corresponding to the user choices in the cluster of wash option buttons. The output array starts with the `Car Enter` string, is followed by the option strings, and ends with the `Car Exit` string in a manner analogous to the output of the `Build Wash Array` function.

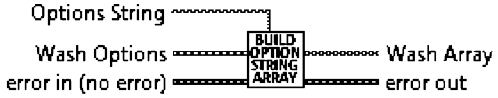


Figure 22.13: The icon of the `Build Option String Array` sub-VI ■

Exercise 8.20: Develop a String-Based Car Wash 2 Controller with Time Target Arguments

Develop a string-based version of the Car Wash 2 controller using arguments for the time targets of the option states. Starting with the queued version of the Car

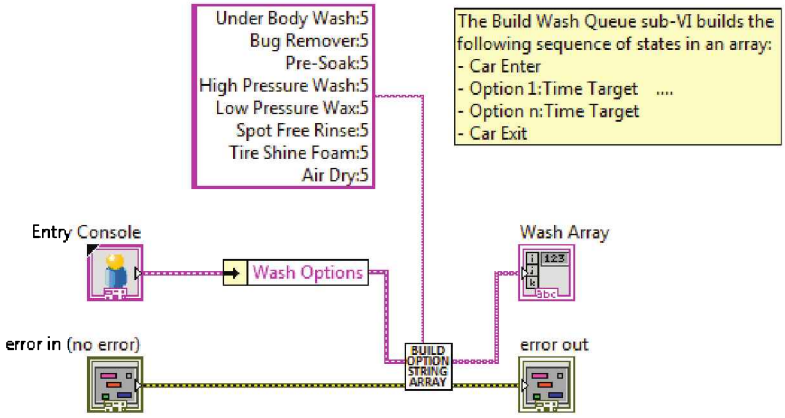


Figure 22.14: Using `Build Option String Array` sub-VI with time arguments only

Wash 2 controller developed in Exercise 5.7, change the queue element to a string data type. Use the `State And Time Target` sub-VI developed in Exercise 8.18 to parse the queued string into a state string and a double-precision time target. Use the `Build Option String Array` sub-VI to enqueue the car wash states based on user input. ■

Exercise 8.21: Create the Mapping Sub-VIs for the Car Wash 2 Controller

Now that we have created a string-based version of the Car Wash 2 controller with time arguments, we move toward a version that accepts option name arguments as well. To do this, create the `Get Option Station` sub-VI with the icon shown in Figure 22.15. This mapping sub-VI accepts the name of wash option state as an input and returns the station where that option takes place. The station output is in the form of the enumerated constant created for the `Car In Position` sub-VI developed in Exercise 1.39.

Also create the `Set Option Display` sub-VI with the icon shown in Figure 22.16. This sub-VI accepts the name of a wash option state as an input along with the current elapsed time of the option state, a RUNNING? Boolean control, and a PAUSED? Boolean control. The sub-VI uses these inputs to create a properly formatted output cluster for the CAR WASH DISPLAY indicator of the main application.



Figure 22.15: The icon of the `Get Option Station` sub-VI

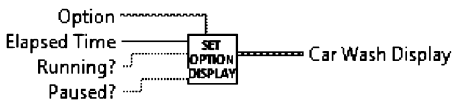


Figure 22.16: The icon of the `Set Option Display` sub-VI ■

Exercise 8.22: Create the `Run Car Wash Option` Sub-VI

Use the mapping functions developed in Exercise 8.21 to create the `Run Car Wash Option` sub-VI with the icon shown in Figure 22.17. Use the block diagram of the `Run Car Wash Cycle` sub-VI shown in Figure 21.12 as a guide.

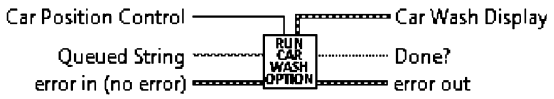


Figure 22.17: The icon of the `Run Car Wash Option` sub-VI ■

Exercise 8.23: Develop a String-Based Car Wash 2 Controller with Wash Option Arguments

Develop a string-based version of the Car Wash 2 controller using arguments for the time targets of the option states. Starting with the queued version of the Car Wash 2 controller developed in Exercise 5.7, change the queue element to a string data type. Use the `State And Time Target` sub-VI developed in Exercise 8.18 to parse the queued string into a state string and a double-precision time target. Use the `State:Name:Time Target` format to code the arguments into the queued string. Use the `Build Option String Array` sub-VI to enqueue the car wash states based on user input, as shown in Figure 22.18.

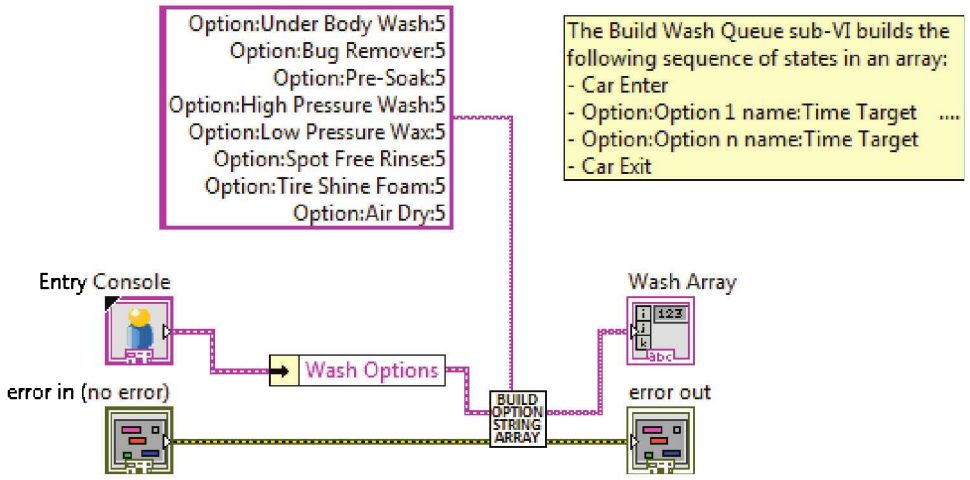
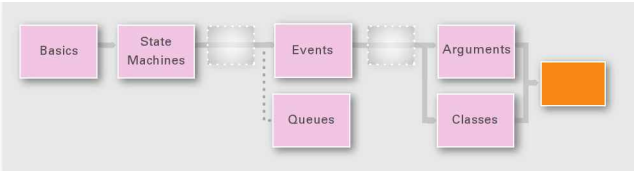


Figure 22.18: Using `Build Option String Array` sub-VI in the Car Wash 2 controller with wash option arguments ■



23

Data Encapsulation

All of the implementations of the car wash controller presented thus far have one thing in common: They all use an action engine to time the car wash cycles. This means that they all share a potential vulnerability as well. They all use a global resource to store timing information, and this resource is not protected.

Figure 23.1 shows the block diagram of a `Timer Bug` VI. This VI places the car wash timer action engine in a loop and resets it each loop iteration. If this simple VI is running in the background, it will hang any of the car wash controllers described in this book so far.

The reason for this is simple. Action engines are global. If the `Timer Bug` VI is run simultaneously on the same computer as the car wash controller, the timer state variables are accessed by both the car wash controller and the `Timer Bug`. The `Timer Bug` continuously resets the elapsed time to zero. A running car wash cycle will never complete, because the time target will never be reached.

Here is another example of the problem. Imagine running two versions of the car wash timer simultaneously on the same computer. Each version of the car wash controller calls the same action engine to control the cycle timing. Whenever one controller resets the timer, both controllers will feel the consequences. Only one controller can run at a given time on a given computer.

This issue is related to the concept of data encapsulation. Data encapsulation has a very specific definition in object-oriented programming, but in general, it can be thought of as data protection. How exposed is a piece of data? Can a piece of data be manipulated from outside the main program? Can the data be manipulated in unexpected ways from within a program? Data that is well encapsulated only can be accessed from within the main program and only by using prescribed methods.

One approach to solving this problem is to re-examine the concept of action engines. In Chapter 5, action engines were defined as sub-VIs with persistent storage. A state variable approach was proposed in which the actual work of the action engine was performed by a kernel sub-VI. The kernel sub-VI accepted a cluster of state variables as an input and provided an updated cluster of state variables as an output. The kernel itself had no persistent storage—the storage of the state variables was provided by a shell sub-VI. The shell stored the state variables and passed them to

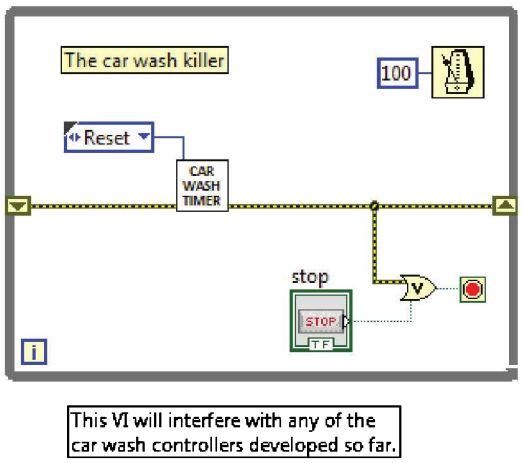


Figure 23.1: This program illustrates the danger of using a global resource to control a program

the kernel. The advantage of this method is that it separates the storage of the state variables from the methods of the action engine.

This concept can be modified to defeat the `Timer Bug` VI. The problem with the `Timer Bug` is that timer state variables are stored in a global resource, which is a sub-VI that can be called by any VI running on the same computer. Why not store the state variables in the main VI itself? If the state variables are stored locally in the main VI, they cannot be accessed globally. Multiple copies of the same program could then run on the same computer without conflict, since each copy would have its own locally stored state variables.

One way to do this is to store the state variables on the data highway. The data highway provides persistent storage, is accessible to all states, but is not accessible outside of the main VI. Let's use the queued-string car wash controller from Chapter 21 as an example. The *Cycle* state of the queued-string car wash controller was shown in Figures 21.20 and 21.21 using two functions of the car wash cycle action engine: *Run Cycle* and *Reset*.

Recall that each of these functions was implemented using a separate method sub-VI within the kernel sub-VI of the action engine. Now imagine that the separate method sub-VIs for the `Timecheck` and the `Reset` functions were liberated from the action engine and were used directly, relying on the data highway of the car wash controller to store the state variables. The results look like Figures 23.2 and 23.3. In this version of the car wash controller, the timer state variables have been saved as a type-defined cluster that is used to initialize a shift register on the data highway. This provides persistent storage of the state variables that is local to the main VI.

The `Car Wash Cycle Data Highway` sub-VI icon is shown in Figure 23.4, and the `Reset Timer Data Highway` sub-VI icon is shown in Figure 23.5. These are not action engines because they have no persistent storage. They rely on the data highway to store the state variables. This defeats the `Timer Bug` VI and also allows

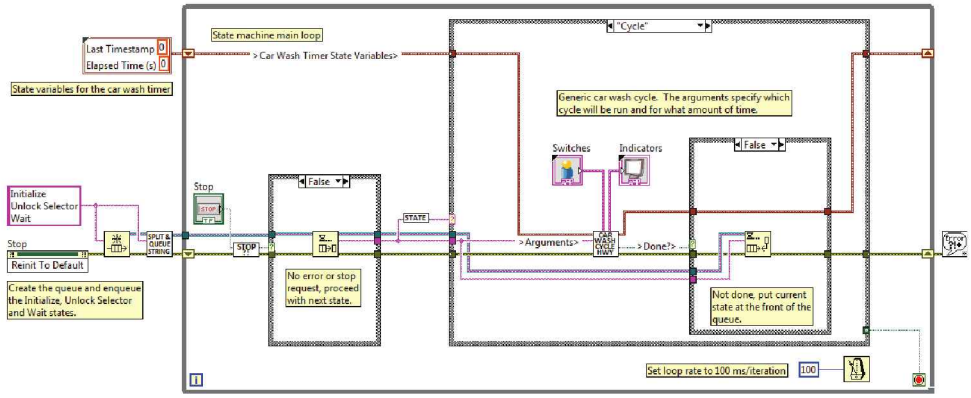


Figure 23.2: The *Cycle* state of the modified queued string car wash controller showing the *Not Done* case

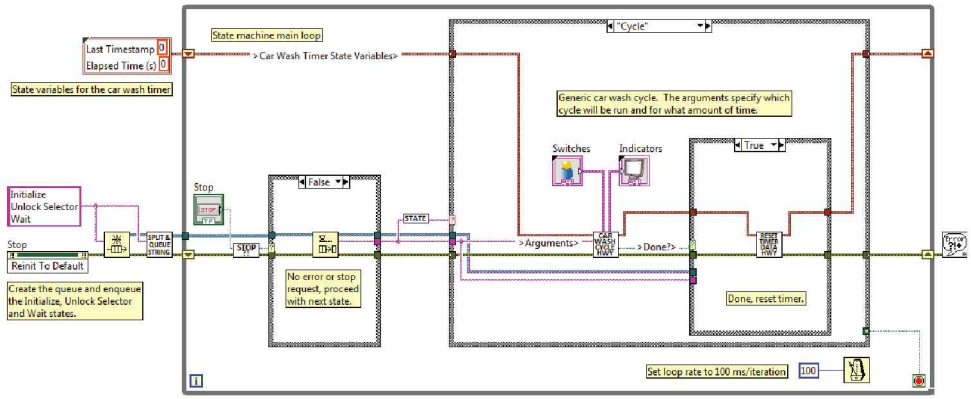


Figure 23.3: The *Cycle* state of the modified queued string car wash controller showing the *Done* case



Figure 23.4: The icon of the `Car Wash Cycle Data Highway` VI



Figure 23.5: The icon of the `Reset Timer Data Highway` VI

multiple versions of the car wash controller to run on the same computer. The state variables are now safely stored in the data highway and cannot be accessed from outside the main VI. The block diagram of the `Car Wash Cycle Data Highway` sub-VI is shown in Figure 23.6, and the block diagram of the `Reset Timer Data Highway` sub-VI is shown in Figure 23.7.

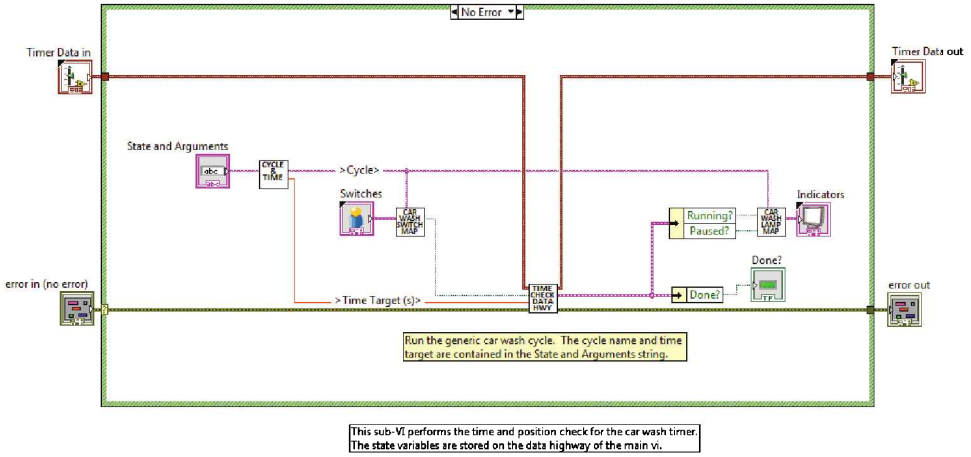


Figure 23.6: The block diagram of the `Car Wash Cycle Data Highway` VI

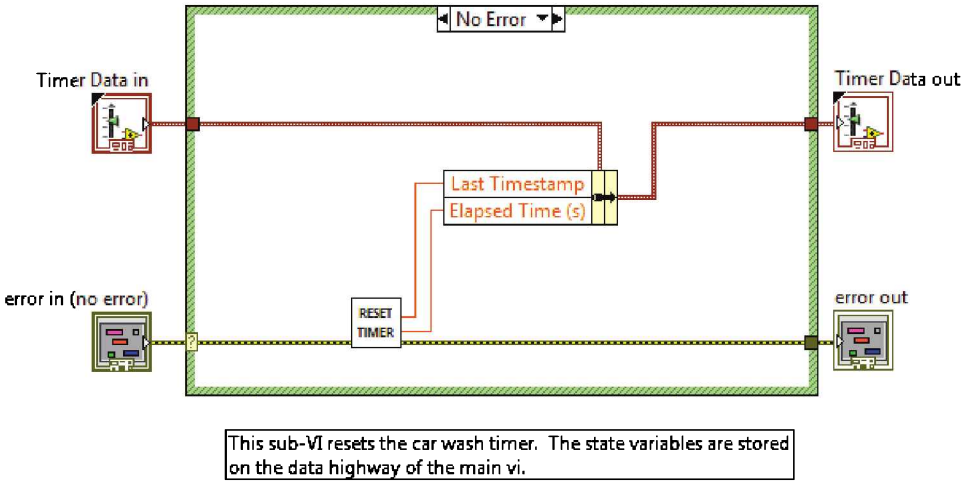


Figure 23.7: The block diagram of the `Reset Timer Data Highway` VI

Placing the state variables on the data highway keeps the state variables in the main VI and protects them from external exposure. But using this technique requires self-discipline on the part of the developer. The state variable data is still exposed internally on the data highway and can be manipulated without using the `Car Wash Cycle Data Highway` VI or the `Reset Timer Data Highway` VI. It may be tempting to manipulate the data directly, but this temptation should be resisted. An example is shown in Figure 23.8. This figure shows the *Done* case of the *Cycle* state and is very similar to Figure 23.3. But instead of using the `Reset Timer Data Highway` VI to reset the timer, the state variables are manipulated directly.

This technique is a bad style, because direct manipulation breaches the encapsulation of the data. It is much better to use the method sub-VIs designed to manipulate

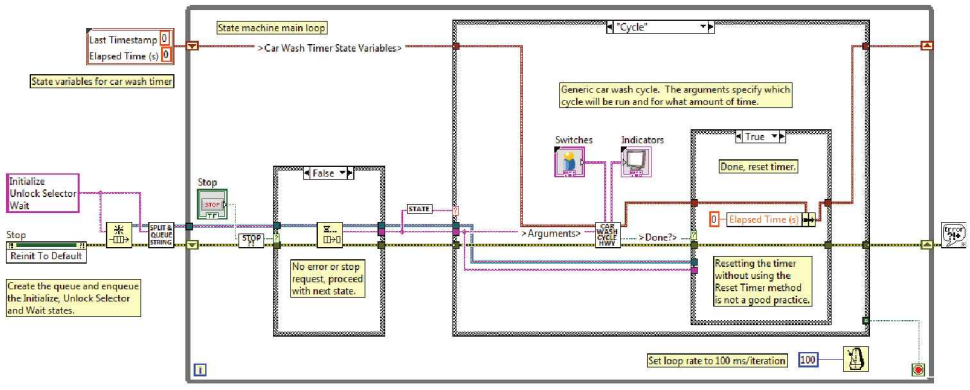


Figure 23.8: Manual manipulation of the state variable data is a bad idea

the encapsulated data. The advantage of this approach is that the format of the data and the algorithms for manipulating it are also hidden. If the developer decides to change the data format or the way in which it is manipulated, the changes are limited the type definition of the data and block diagrams of the method sub-VIs. The main VI is unaffected by these changes.

Following this discipline also helps prevent bugs. Did you notice the bug in Figure 23.8? When the timer is reset, both state variables need to be overwritten. The elapsed time needs to be set to zero (as is shown in the example), but the stored timestamp needs to be overwritten by the current timestamp as well. These details need to be remembered if the data is manually manipulated. If the method sub-VI is used to reset the timer, these details are hidden, reducing the likelihood of bugs.

23.1 LabVIEW Objects and Classes

The data highway approach defeated the `Timer Bug` by localizing the state variables to the main VI, but it did not provide true data encapsulation because the data was still exposed on the data highway. The data could still be manipulated directly without using the designated method sub-VIs.

True data encapsulation requires support from the programming language. The language itself needs to prevent any attempt to manipulate the encapsulated data except through the use of the designated method sub-VIs. This is typically done by using objects. An object can be thought of as a protected data set combined with a designated set of methods. The protected data can only be accessed and manipulated by using the designated methods.

Each object has its own unique copy of the protected data. For example, suppose we had a car wash timer object with protected state variable data and method sub-VIs for doing a time check and resetting the timer. If we had two timer objects, they could run in parallel without interfering with each other, because each one would have its own protected copy of the state variable data. Resetting one timer would not reset the other.

Even though the two timer objects are distinct, they are obviously related. They have distinct copies of the protected data, but the data types of the protected data are the same in both cases. The method sub-VIs are identical for both objects as well. Both of these timer objects can be thought of as members of a "class" of car wash timers. The class of car wash timers is an abstract idea of what a car wash timer is, what kind of protected data it has, and what kinds of methods can be used to manipulate the private data. An object is a concrete realization of the abstract idea defined by the class. An object is an "instantiation" of the class.

In LabVIEW, a class is essentially a library consisting of a special cluster control and a set of sub-VIs that access and manipulate the data in the special cluster. The elements of the cluster are the protected data and are called the *properties* of the class. The sub-VIs are called the *methods* of the class.

To create a new class, select *File...New* and select *Other Files...Class*, as shown in Figure 23.9. The class looks like the LabVIEW project file seen in Figure 23.10. The class contains a `ctl` file. This control file is a cluster that contains the private data of the class. Figure 23.10 shows the private data cluster for a `Timer` class. It contains numeric controls for the timestamp and the elapsed time.

After the class has been created, the properties of the class can be set by right-clicking on the `lvclass` icon and selecting *Properties*. This brings up a window as

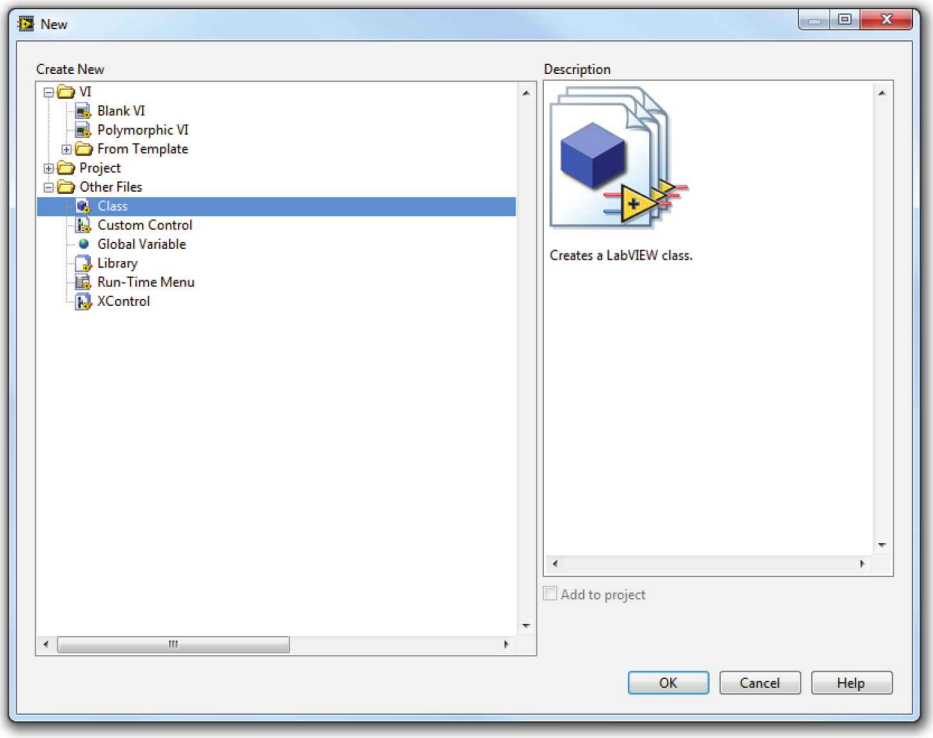


Figure 23.9: Creating a new LabVIEW class

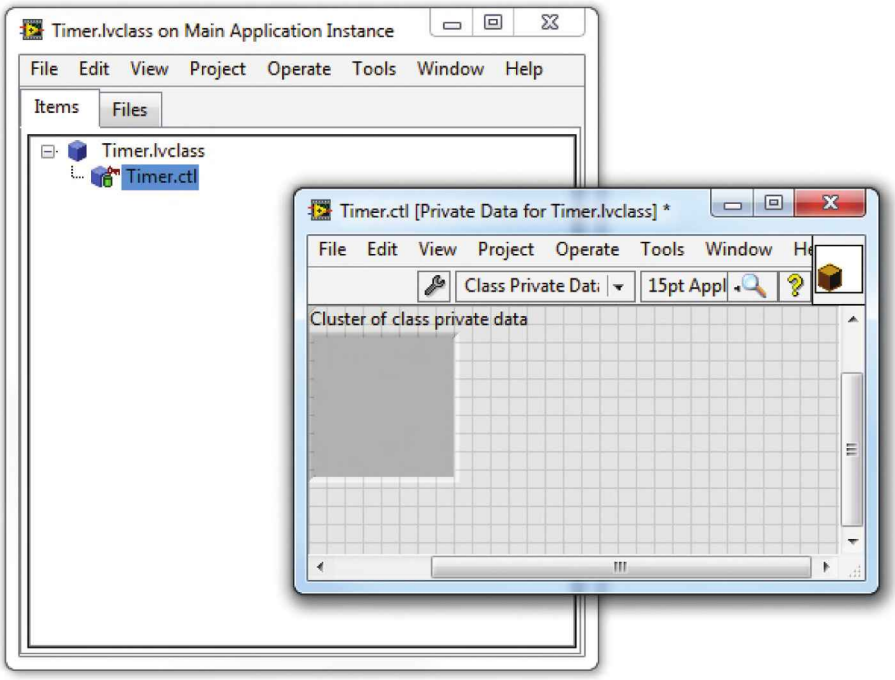


Figure 23.10: The private data cluster of the `Timer` class

seen in Figure 23.11. To create an icon template for the class, press the EDIT ICON... button. This brings up an icon editor, as shown in Figure 23.12.

The icon template is comprised of a banner with a background color. The banner is superposed on the method sub-VIs of the class, making it obvious that they belong to the same class.

Wiring Tip: As seen in Figure 23.12, the *Small Fonts* font was chosen for the icon. The anti-aliased fonts used in the rest of this book are difficult to read when the background is not white.

Method VIs for the class can be created by right-clicking on the `lvclass` icon in the class library and selecting *New*. There are four types of method sub-VIs that can be created from this menu: *Data Member Access*, *Static Dispatch*, *Dynamic Dispatch*, and *Override*. Dynamic dispatch and override sub-VIs will be covered in the next chapter.

Data member access sub-VIs are methods used to read from or write to the private data cluster. Private data is hidden from VIs outside of the class. The only way for VIs outside the class to access the private data directly is through the use of data member access.

A static dispatch sub-VI is a class sub-VI that behaves like a standard sub-VI. The name "static dispatch" will be explained in the next chapter. For now, think of a

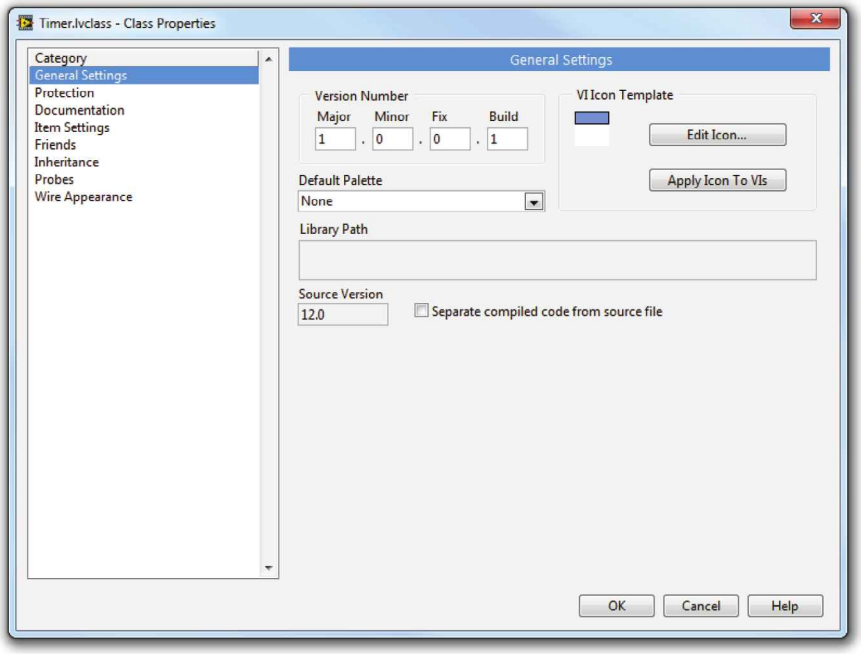


Figure 23.11: The *General Properties* tab of the *Class Properties* window

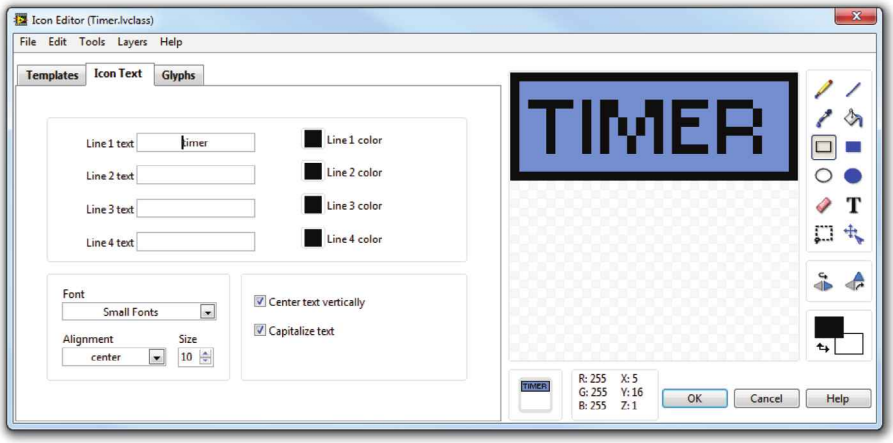


Figure 23.12: The class icon editor

static static dispatch sub-VI as a standard sub-VI that can access the private data of its class.

When a static dispatch sub-VI is created by right-clicking the `lvclass` icon of the class library, the new sub-VI is customized for the class and is automatically a member of the class. The front panel is created with a control and indicator with the

data type of an object of the class. These are the cube-like items seen in the upper corners of Figure 23.13. The front panel is also created with error input and output terminals.

The front panel objects are wired to the connector pane, as shown in Figure 23.14. The connector pane follows the standard LabVIEW conventions: the 4 × 2 × 2 × 4 pattern is used, the error terminals are wired in the lower corners, and the object terminals are wired in the upper corners. The icon of the new sub-VI consists of the class icon developed in the *Properties* window (see Figure 23.11) and an identification number.

The block diagram of a new static dispatch sub-VI is shown in Figure 23.15. The block diagram follows the error-handling template discussed in Chapter 1. If an error is present at the error input terminal, the sub-VI will not execute—instead the error will be passed directly to the error output terminal.

The new static dispatch sub-VI then can be modified and saved as a method VI for the class. This procedure was followed previously to create the `Time Check` sub-VI for the `Timer` class. The icon for this method sub-VI is shown in Figure 23.16, and the front panel is shown in Figure 23.17. The `Time Check` VI accepts a `Timer` object, a proximity switch, and a time target and performs the time and proximity check required for the car wash controller.

The block diagram for the `Time Check` VI is shown in Figure 23.18. The old timestamp and elapsed time are unbundled from the private data of the timer and are passed to the same `Time and Proximity Check` sub-VI developed in Chapter 6 and shown in Figure 6.4.

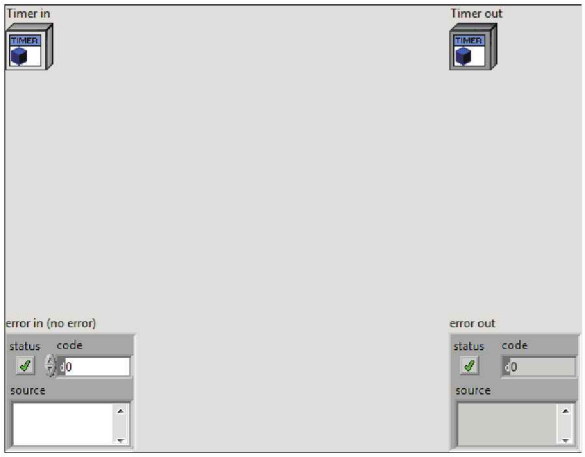


Figure 23.13: Front panel of a new static dispatch VI



Figure 23.14: Icon of a new static dispatch VI

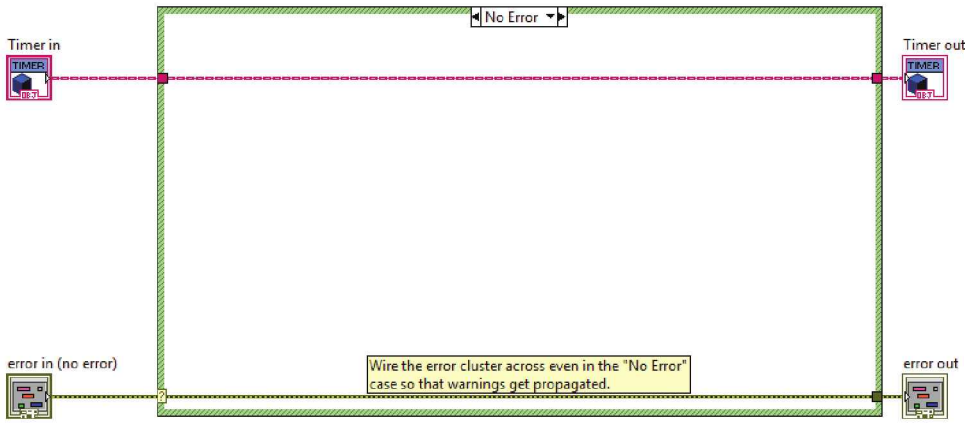


Figure 23.15: Block diagram of a new static dispatch VI

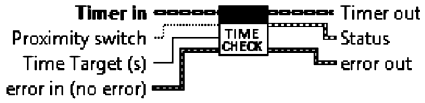


Figure 23.16: Icon of the `Time Check` method of the `Timer` class

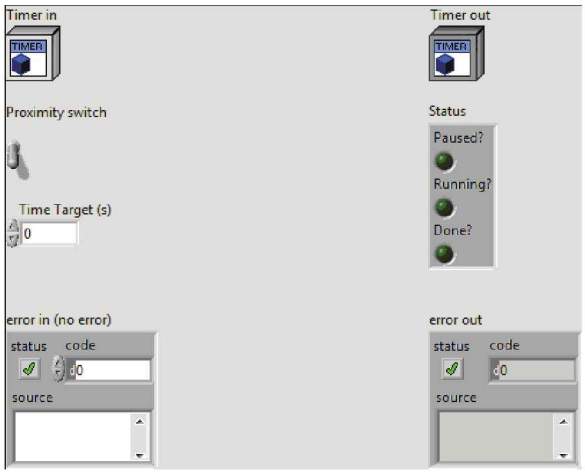


Figure 23.17: Front panel of the `Time Check` method of the `Timer` class

A second timer method needs to be created: a method that will initialize the timer. This is another static dispatch sub-VI, and it uses the `Reset Timer` kernel developed in Chapter 6 and shown in Figure 6.6. The icon for this method sub-VI is shown in Figure 23.19. The name of this new method sub-VI is `Initialize Timer` rather than `Reset Timer`. This follows a common convention of object-oriented programming.

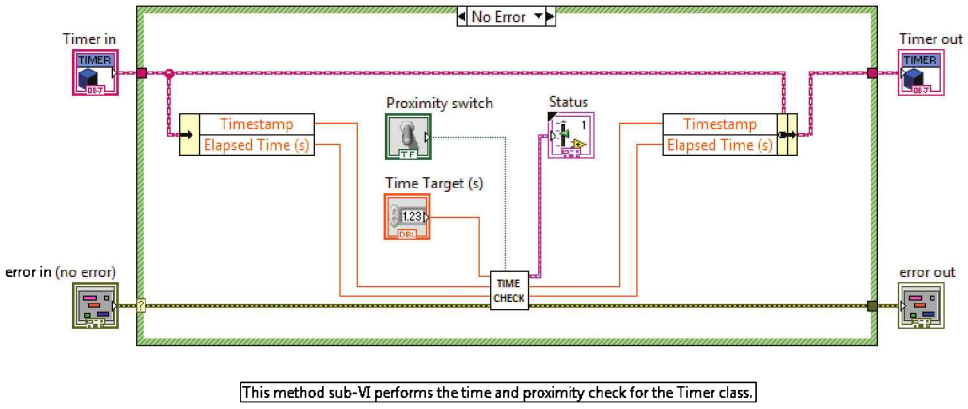


Figure 23.18: Block diagram of the `Time Check` method of the `Timer` class



Figure 23.19: Icon of the `Initialize Timer` method of the `Timer` class

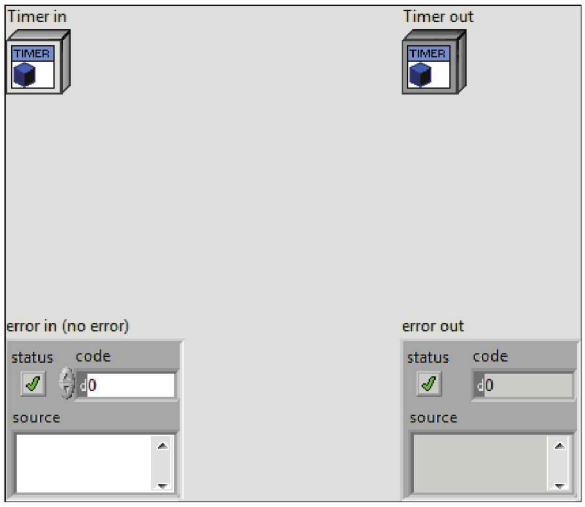


Figure 23.20: Front panel of the `Initialize Timer` method of the `Timer` class

Methods that set the private data of an object to a desired starting state are usually given a name containing the word "Initialize." The front panel of the `Initialize Timer` VI is shown in Figure 23.20, and the block diagram is shown in Figure 23.21.

We have now imported the functionality of the original car wash timer action engine into the new `Timer` class. This process is straightforward, because we use the state variable approach for the car wash timer. This enables us to import the kernel

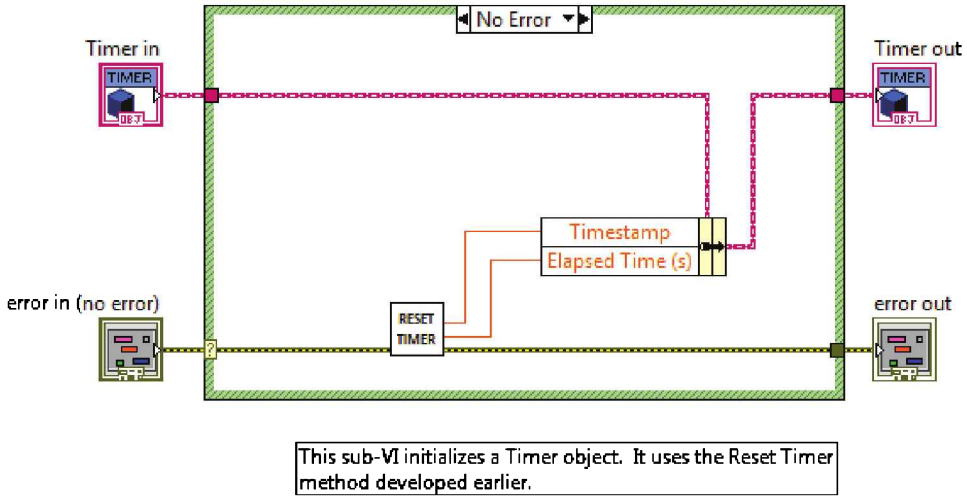


Figure 23.21: Block diagram of the `Initialize Timer` method of the `Timer` class

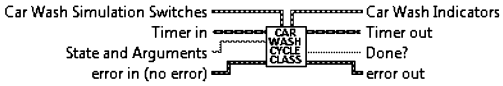


Figure 23.22: Icon of the `Car Wash Cycle Timer Class` sub-VI

sub-VIs and use them as methods in the new class, providing storage for the state variables in the private data of the class.

The last sub-VI that needs to be developed is the `Car Wash Cycle lvclass Timer`, which is a car wash cycle sub-VI that uses a `Timer` object. The icon of such a sub-VI is shown in Figure 23.22. The corresponding front panel is shown in Figure 23.23, and the block diagram is shown in Figure 23.24.

This sub-VI accepts a `Timer` object as an input and passes the object as an output. The sub-VI has to pass the object as an output, because the private data of the timer changes as it passes through the sub-VI. Remember, LabVIEW passes data by-value unless references are used. This applies to LabVIEW objects as well. The current value of the object needs to be stored persistently on the data highway to ensure that the current values of the private data are used in the next loop iteration.

The fact that the `Car Wash Cycle Timer Class` VI uses a `Timer` object raises an interesting question. Does the `Car Wash Cycle Timer Class` VI need to be a member of the `Timer` class? To answer that question, examine the block diagram of the sub-VI shown in Figure 23.24. Notice that this sub-VI is very similar to the one shown in Figure 21.12. The main difference is that a `Timer` object is passed to the time check method of the `Timer` class rather than use an action engine to perform the time check.

The key point is that the car wash cycle sub-VI does not need to access the private data of the `Timer` object. It simply passes that object on to the time check method.

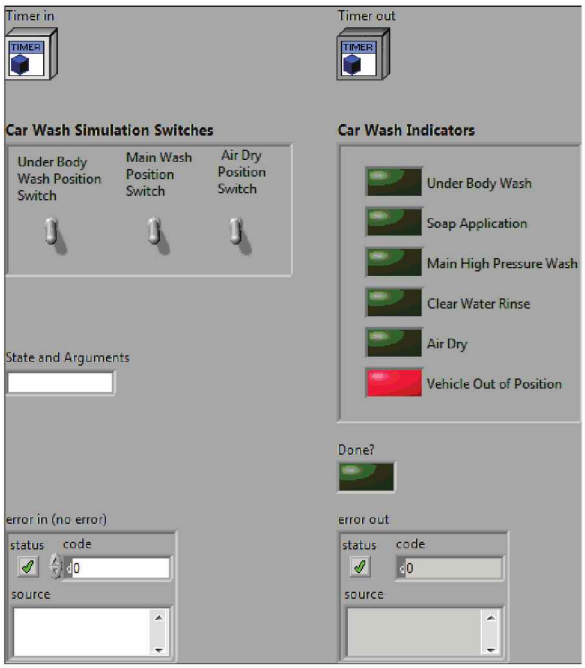


Figure 23.23: Front panel of the `Car Wash Cycle Timer Class` sub-VI

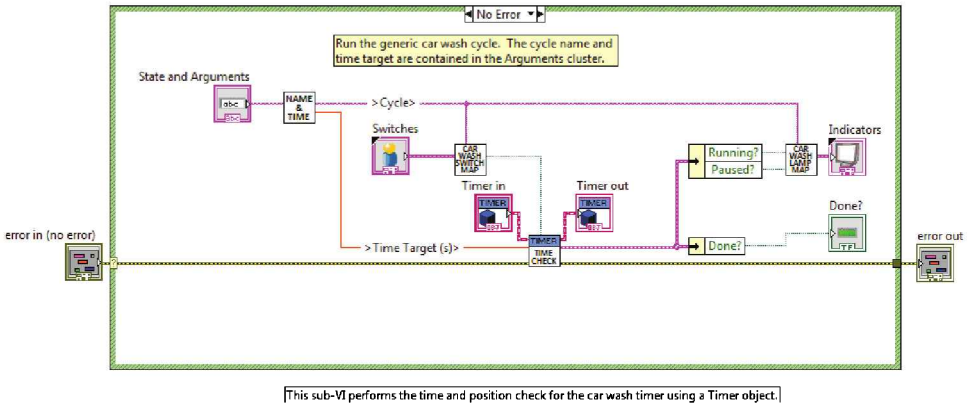


Figure 23.24: Block diagram of the `Car Wash Cycle Timer Class` sub-VI

This means that the car wash cycle sub-VI does not need to be a member of the `Timer` class. Since the car wash cycle sub-VI doesn't add new functionality to the `Timer` class itself, it is better to keep the sub-VI outside of the class. This keeps the class more focused on its central purpose and gives the class better cohesion. Classes with high cohesion are usually superior to those with low cohesion, since they are easier to understand and easier to reuse in other codes.

Now that the class has been developed, it can be deployed in the car wash controller. The *Cycle* state of the new car wash controller is shown in Figures 23.25 and 23.26.

Once again, notice the overall similarity between these figures and Figures 23.2 and 23.3. All of these figures are nearly identical. The only difference is that the state variable cluster in the data highway version has been replaced by a `Timer` object.

If the two approaches produce nearly identical programs, why bother with the effort of replacing the state variable cluster with a class? The answer can be seen in Figure 23.27. Classes provide true data encapsulation. The state variable/data highway approach described in the beginning of this chapter provides a measure of data encapsulation, but it depends on developer self-discipline. The developer needs to ensure that the state variable cluster data is only accessed through sub-VIs designed

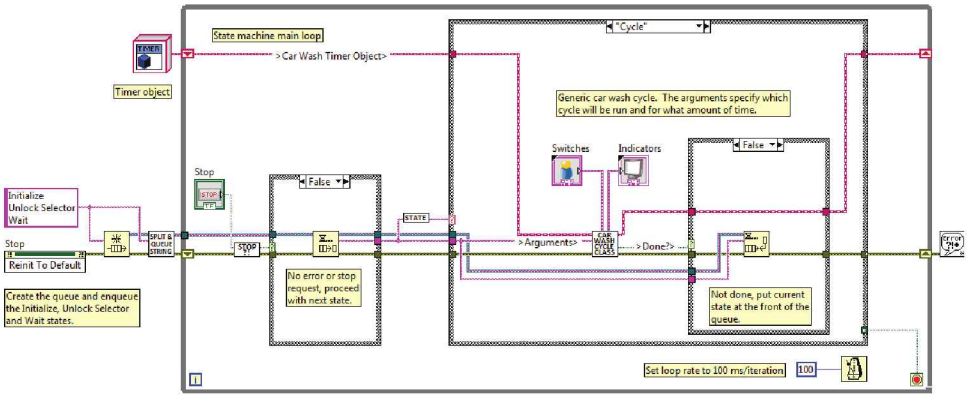


Figure 23.25: The *Cycle* state of the `Timer` class queued string car wash controller showing the *Not Done* case

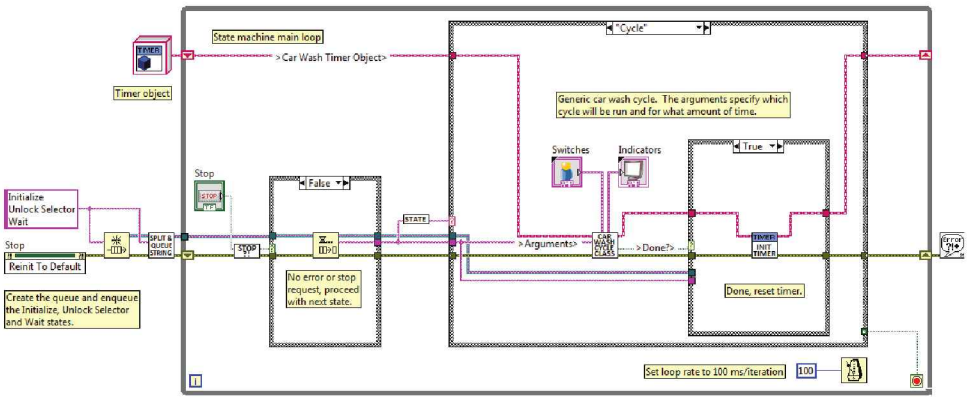


Figure 23.26: The *Cycle* state of the `Timer` class queued string car wash controller showing the *Done* case

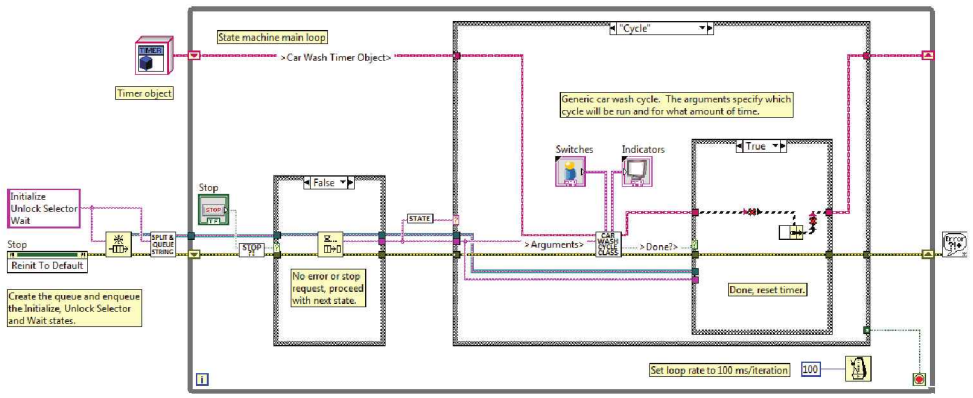
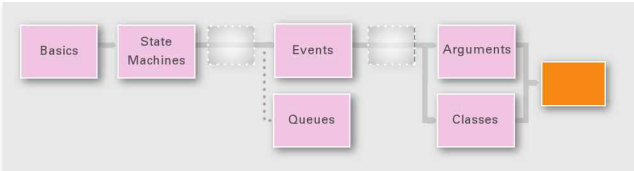


Figure 23.27: The private data of the `Timer` class can't be accessed if the VI does not belong to the `Timer` class

for that purpose. With a LabVIEW class, there is no need for this self-discipline. As can be seen in Figure 23.27, LabVIEW prevents developers from accessing the private data of a class in VIs outside of the class. Attempts to do so cause broken wires and broken run arrows.

Classes also give the developer the ability to set the access scope of method VIs. A developer can make some members of the class available for public use, allowing them to be called by VIs outside the class. Other method sub-VIs can be made private, restricting their use only to other members of the class. Community and protected scopes are also possible and will be discussed in Chapter 25. These features make classes very portable and reusable while also hiding the internal details of data formats and method algorithms.



24

Problem Set 9

24.1 Traffic Light Controller Exercises

Exercise 9.1: Auto-Reset Timer Class

Create a LabVIEW class or modify an existing class to replace the auto-reset timer action engine used in the traffic light controller. *Hint*: Compare the private data of the traffic light timer to the private data of the `Timer` class developed in Chapter 23. Do you need to start from scratch to create the auto-reset timer? Can you use or somehow extend the `Timer` class that already has been developed? ■

Exercise 9.2: Traffic Light Controller with Auto-Reset Timer Class

Revise the traffic light controller to use the new timer developed in the last exercise, placing it on the data highway to make it accessible to all states. ■

24.2 Boiler Controller Exercises

Exercise 9.3: Data Encapsulation Using the Data Highway

The boiler controller developed in Exercise 8.9 uses two action engines. One action engine is for the stopwatch timer, while the other is used to store the log string in the sub-VI that writes to the log file. This exercise focuses on the log file action engine. Create the `String Changed Kernel` sub-VI with the icon shown in Figure 24.1. This function acts like the `String Changed` action engine, except it stores its state variable on the data highway. Also create the `Write To Log File Data Highway` sub-VI with the icon shown in Figure 24.2. This sub-VI is identical to the `Write To Boiler Log File` developed in Exercise 2.4, except it stores the old log string externally on the data highway rather than internally.



Figure 24.1: The icon of the `String Changed Kernel` sub-VI



Figure 24.2: The icon of the `Write To Log Data Highway` sub-VI ■

Exercise 9.4: Boiler Controller With Data Highway State Variable

Revise the boiler controller with arguments developed in Exercise 8.17 to use the data highway storage sub-VIs developed in Exercise 9.3. ■

Exercise 9.5: Stopwatch Class

Create a LabVIEW class or modify an existing class to replace the timer action engine used in the boiler controller. The timer should simply act like a stopwatch and return the elapsed time from when it was last reset or initialized. *Hint*: Compare the private data of the stopwatch timer to the private data of the `Timer` class developed in Chapter 23. Do you need to start from scratch to create the stopwatch timer? Can you use or somehow extend the `Timer` class that has already been developed? ■

Exercise 9.6: Boiler Controller with Stopwatch Timer Class

Revise the version of the boiler controller you developed in Exercise 9.4 to use the new stopwatch timer developed in the last exercise, placing it on the data highway to make it accessible to all states. In this way, both of the of the action engines in the boiler controller will have been replaced with elements that store their persistent data on the data highway of the main VI. ■

24.3 Car Wash 2 Controller Exercises

Exercise 9.7: Create the `Car Wash 2 Time Check Timer` Method

Create the `Car Wash 2 Time Check` method sub-VI for the `Timer` class (see the icon in Figure 24.3). This method should be identical to the one shown in Figure 23.18 but with an indicator to output the elapsed time.

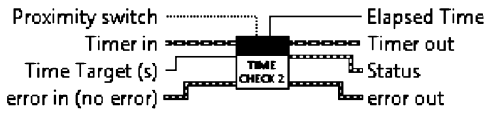


Figure 24.3: The icon of the `Car Wash 2 Time Check` sub-VI ■

Exercise 9.8: Create the `Run Car Wash Option Timer lvclass` Sub-VI

Create the `Run Car Wash Option Timer lvclass` sub-VI with the icon shown in Figure 24.4. This sub-VI is similar to the one developed in Exercise 8.22 except for the fact that it uses a `Timer` object rather than an action engine to monitor the elapsed time.

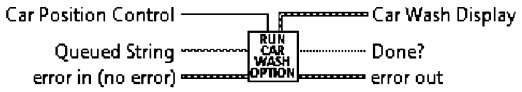
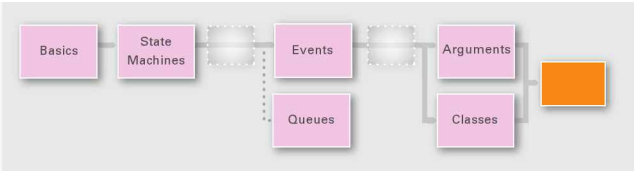


Figure 24.4: The icon of the `Run Car Wash Option Timer lvclass` sub-VI ■

Exercise 9.9: Car Wash 2 Controller with a `Timer` Object

Create a data-encapsulated version of the Car Wash 2 controller by replacing the `Car Wash 2 Timer` action engine with a `Timer` object. Use the `Run Car Wash Option Timer lvclass` sub-VI to execute the wash option timechecks. Store the `Timer` object on the data highway. ■



25 Inheritance and Composition

In the last chapter, we introduced LabVIEW classes and objects and discussed how they can be used for data encapsulation. We took advantage of the fact that an object is a self-contained member of a class with its own private data and predefined methods to access and manipulate that data.

But data encapsulation is only one of the advantages of classes. The real power of a class lies in its ability to form special relationships with other classes. Two of these special relationships are introduced in this chapter: composition and inheritance.

To illustrate these concepts, we create a new class for the car wash controller: the `Cycle` class. In the last chapter, we created a class to represent the timer used for each of the car wash cycles. In this chapter, we create a class to represent the car wash cycles themselves, such as *Main Wash* or *Rinse*. A `Cycle` object knows which proximity switch it needs to monitor and which indicator lamp it needs to light. The `Cycle` object also knows how to perform its own time check. But in order to do this, the `Cycle` object needs to have a timer that monitors a proximity switch. The `Timer` class developed in the last chapter would work perfectly if it could somehow be incorporated into the new `Cycle` class.

This is done by using a technique known as *composition*. Composition simply means that an object of one class can be placed into the private data of another class. Composition is a simple idea, but a powerful one. By placing one object inside of another, you can automatically leverage the programming effort to create one class to create another class. The `Cycle` class is a good example. If a `Cycle` object contains a `Timer` object, the time check method of the `Timer` object could be used by the `Cycle` object. The `Timer` functionality is imported to the `Cycle` class with little extra effort on the part of the developer.

Composition means that an object of one class can "contain" an object of a different class. Composition is a special relationship between two objects of different classes. Inheritance, on the other hand, is a special relationship between two objects of related classes.

What do we mean by "related" classes? Imagine that we create a set of five classes to represent the individual car wash cycles: `Underbody Wash`, `Soap Application`, `Main Wash`, `Rinse`, and `Air Dry`. These five classes would be very similar.

They would all share the same kind of private data, they would all monitor a proximity switch, they would all have time targets, they would all use the same time and proximity check method, and they would all light indicator lamps. They would be nearly identical to each other, except that each would have independent time targets, and the monitored proximity switch and output lamps would differ from one cycle to the next. These five classes are related to each other, almost like siblings in a family. And like siblings in a family, they share the same parent. In this example the parent would be the generic `Cycle` class.

This is the idea behind inheritance. Inheritance allows child classes to "inherit" from a parent class. For example, a child class can use methods defined by its parent class. This saves programming effort. In the context of the car wash controller, the main programming effort goes into developing the generic `Cycle` class. The five car wash cycles are then created as children of the `Cycle` class. The children will be much easier to develop and will benefit from the effort put into developing the `Cycle` class.

Another advantage of inheritance is that child classes also inherit the data type of the parent class. Objects of child classes can travel on wires on a block diagram that have the data type of the parent class. This means that we can write a VI in terms of a generic `Cycle` class, but all of the objects traveling on the wires when the program actually runs will be objects from the child classes.

Take the `Cycle` class, for example. A car wash cycle is an abstract concept. A generic car wash cycle has a number of features: a proximity switch that indicates if a vehicle is in position, a time target that specifies the duration of the cycle, and an indicator lamp that signifies if the cycle is running. These features are shared by every car wash cycle, but there is no "generic" cycle actually operating in the car wash. The cycles operating in the car wash are real and specific. They each monitor a specific real switch and light a specific real lamp.

Even though the generic cycle doesn't really exist in an actual car wash, it is very convenient to use the generic cycle as a placeholder in the car wash controller. Figure 25.1 shows an implementation of the car wash controller that uses the `Cycle` object. We haven't developed all of the pieces yet to fully understand this figure, but

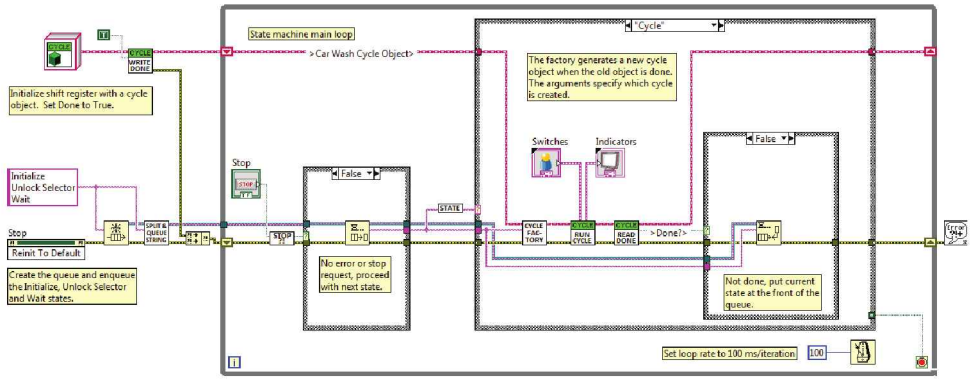


Figure 25.1: The *Cycle* state of a car wash controller using a `Cycle` object

it will serve as good overview. The data highway shift register is initialized with a `Cycle` object. The main case structure shows the *Cycle* state. This state has three sub-VIs in series that act on the `Cycle` object: `Cycle Factory`, `Run Cycle`, and `Read Done`. After these three sub-VIs execute, the `Cycle` object is placed back on the data highway.

Here is how the *Cycle* state operates. Notice that the `Cycle Factory` sub-VI accepts the arguments string as an input. This sub-VI checks to see if the current `Cycle` object is done or not. If it not done, the factory simply passes it on. If it is done, the factory generates the appropriate new child object based on the argument string. For instance, it might generate a `Main Wash` object or a `Rinse` object. It will *not* produce `Cycle` objects, since these don't really exist in the car wash. The child object is free to travel on wires that belong to its parent, though, so the child object can be passed to the next two sub-VIs in the sequence. The `Run Cycle` sub-VI executes the time and proximity check, allowing the child object to choose which switch to monitor and which lamps to light. Finally, the `Read Done` sub-VI determines if the child object is done or not. The child object is then placed back on the data highway, where it again is passed on a generic `Cycle` wire.

This example shows one of the powerful features of inheritance. The *Cycle* state was written with no knowledge of what individual states actually exist in the car wash. The main VI is written entirely in terms of the generic `Cycle` class, which will never actually be executed. The child classes will travel on the `Cycle` wires when the code is run. This makes the code very scalable. New child classes can be added without changing the structure of the main VI.

25.1 Inheritance and Composition

Inheritance is a special relationship between related classes. A child class inherits the private data and methods of the parent class. Child classes can travel on the wires of its parent class, and as we will see, child classes can override methods of their parent class with methods of their own. Inheritance can be thought of as an "is-a" relationship. If the statement "Class B is-a Class A" is true, then Class B is a good candidate to be a child of Class A. For example, a `Main Wash` is-a `Cycle`, so inheritance might be a good choice for this relationship.

Composition is a special relationship between unrelated classes. This is a "has-a" relationship. If the statement "Class A has-a Class B" is true, it might be a good idea to place an object of Class B into the private data of Class A. For example, a `Cycle` has-a `Timer`, so composition might be a good choice for this relationship.

Now plan for the car wash controller calls for seven different classes: `Timer`, `Cycle`, `Underbody Wash`, `Soap Application`, `Main Wash`, `Rinse`, and `Air Dry`. These classes all can be interrelated through composition and inheritance. One possible diagram for laying out these relationships is shown in Figure 25.2. This type of diagram is called a class diagram. Each of the rectangles represents an individual class. The contents of the class rectangles detail the private data and methods of the class. The arrows indicate the relationships between the classes.

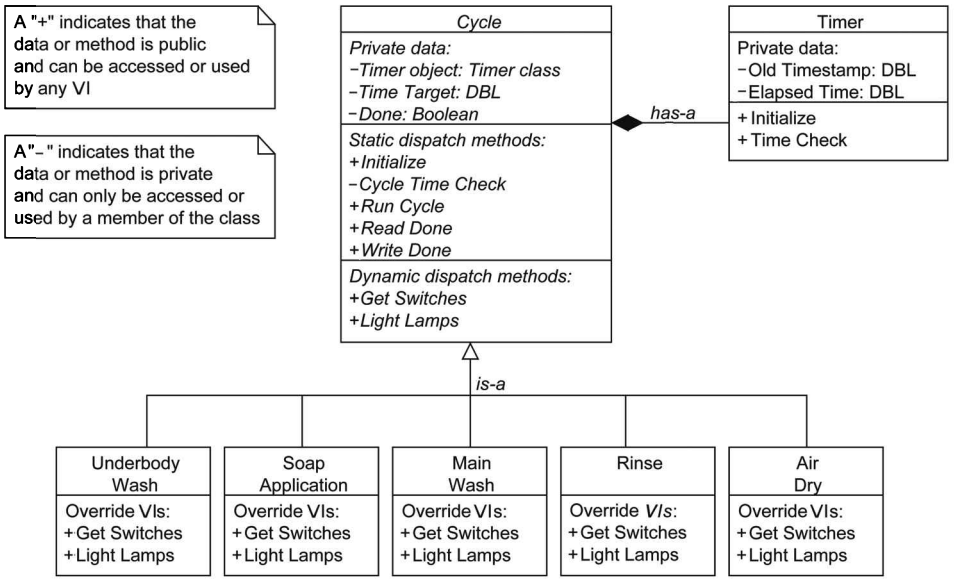


Figure 25.2: A class diagram showing the relationships between the classes of the car wash controller

The solid diamond-head arrow indicates composition. The class touching the arrowhead contains an instance of the class at the other end of the arrow. The figure shows that the generic `Cycle` class contains an object from the `Timer` class. The open triangle-head arrows indicate inheritance. The class touching the arrowhead is the parent of the class at the other end of the arrow. Notice that the child classes do not list private data elements. Each of the children inherits the private data of the parent, including the `Timer` object. Child classes can define additional private data elements not present in the parent class, but this is not necessary for the classes in the car wash controller.

Figure 25.2 also mentions different types of method sub-VIs: static dispatch, dynamic dispatch, and VIs for override. These terms apply to method sub-VIs in a parent class. Static dispatch sub-VIs are always the same, whether they are called by the parent class or by the child classes. For instance, we will develop a `Cycle Initialize` sub-VI that can initialize an object from the `Cycle` class or any of the child classes. The sub-VI does not change when it is called by a member of a child class, so it is classified as a static dispatch sub-VI.

Dynamic dispatch sub-VIs behave differently. If a parent has a dynamic dispatch method, the child class has a choice when calling it. The child class can either call the sub-VI defined by its parent, or it can override the method with a sub-VI of its own. Dynamic dispatch methods are very powerful. They allow a VI to be written in terms of a generic class (as seen in Figure 25.1) but using customized behavior for each of the child classes when they are passed along the data wire of their parents.

For example, the generic `Cycle` class has a dynamic dispatch sub-VI that selects which proximity switch will be monitored. This sub-VI acts as a placeholder with the

intent that each of the child classes will override it. Each of the child classes has an override sub-VI of the same name that selects the actual switch needed by that class. Similarly, the generic `Cycle` class has a dynamic dispatch sub-VI that selects which indicator lamp will be lit during the cycle. This sub-VI is also overridden with new sub-VIs of the same name in each of the child classes.

25.2 Creating the `Cycle` Class

In the previous section, we laid out a plan for using the `Cycle` class and diagrammed its relationship with other car wash cycle classes and the `Timer` class. The class diagram in Figure 25.2 shows that the `Cycle` class has three elements in its private data cluster: a `Timer` object, a *Time Target* numerical control, and a *Done* Boolean. These items were chosen for the parent `Cycle` class because they are needed by every child class. It is good design practice to move data elements shared by all child classes to the parent class. This allows them to be inherited by the child classes, saving programming effort.

If we follow the steps for creating a new class outlined in the last chapter, the resulting class library and private data cluster will look like the ones shown in Figure 25.3.

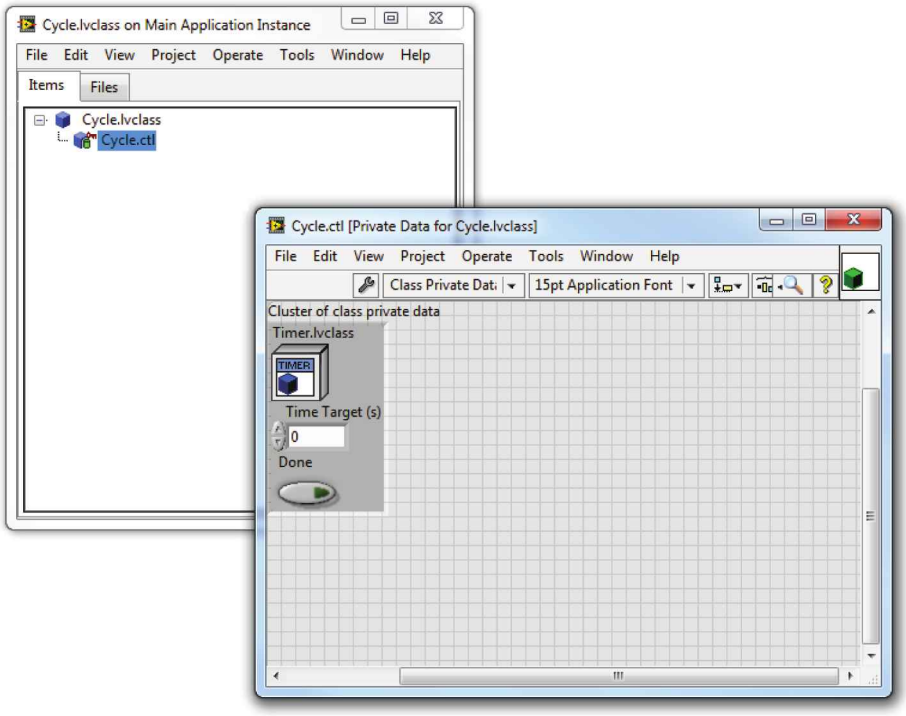


Figure 25.3: The private data of the `Cycle` class

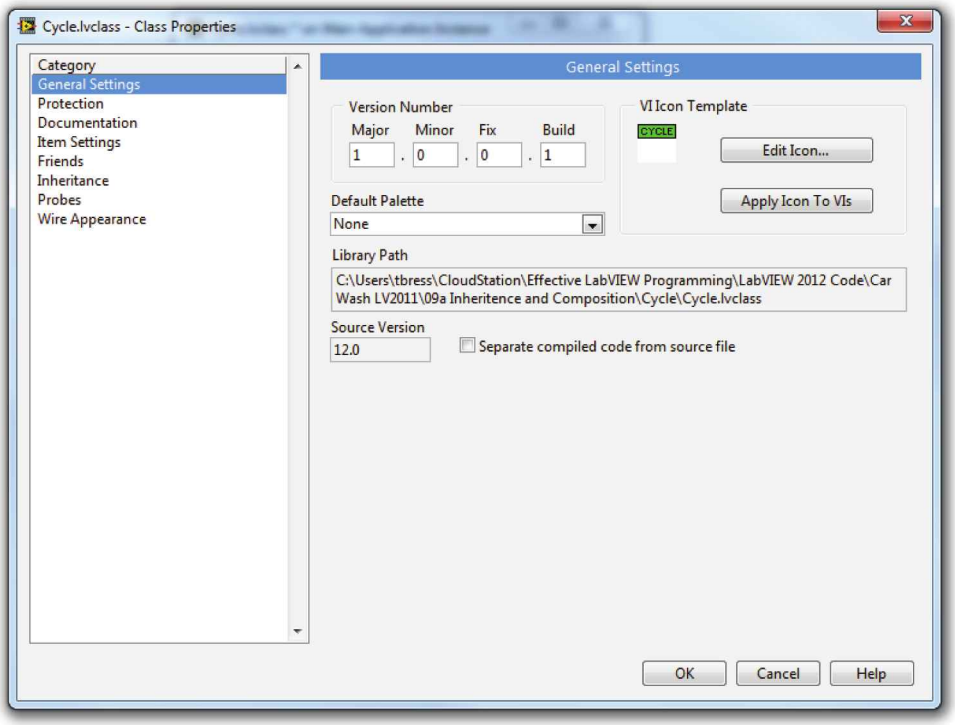


Figure 25.4: Defining the class icon of the `Cycle` class

The private data cluster of the `Cycle` class contains an object from the `Timer` class. This is an example of composition. Every time a new `Cycle` object is created, it will have its own `Timer` object as well. The new `Timer` object is tied to the `Cycle` object. It is created when the `Cycle` object is created, and it is destroyed when the `Cycle` object is destroyed. Each independent `Cycle` object has its own encapsulated timer, and multiple `Cycle` objects can operate at the same time without their timers interfering with each other.

The methods of the new `Cycle` class needs a custom icon header. This header can be created from the general properties page of the class, as shown in Figure 25.4. This figure shows that all methods of the `Cycle` class have icons with the word "Cycle" in the upper shaded portion of the icon.

25.3 Data Member Access Sub-VIs

Now that the class library and private data cluster have been defined, it is time to develop the method sub-VIs. We start with the data member access VIs. Data member access VIs are used to access the private data of a class without violating the encapsulation of the private data. They act as an intermediary between the developers who use the class and the developer who created the class.

Consulting the class diagram in Figure 25.2 and the state machine in Figure 25.1, it is evident that the *Done* Boolean in the private data of a `Cycle` object must be accessible to VIs outside of the class. The *Done* Boolean must be read by the car wash controller to see if the current cycle is done or not. The *Done* Boolean is also written to in the input area of the car wash controller.

The *Done* Boolean needs to be read from and written to by VIs outside of the `Cycle` class. The way to do this without violating data encapsulation is through the use of a data member access sub-VI. A data member access sub-VI is a method of a class with the sole purpose of reading from or writing to an element of the private data of the class. Since the data is accessed by a member of the class, data encapsulation is respected. Access to the private data is possible only through the use of intermediary method sub-VIs.

LabVIEW can automatically generate data member access sub-VIs by right-clicking on the `.lvclass` member of the class library and selecting *New... VI for Data Member Access*. LabVIEW will bring up the window shown in Figure 25.5. This window allows you to configure the VI-generation process. You can select whether LabVIEW will create a read method, a write method, or both. In the case of the *Done* Boolean, we need to create both read and write methods.

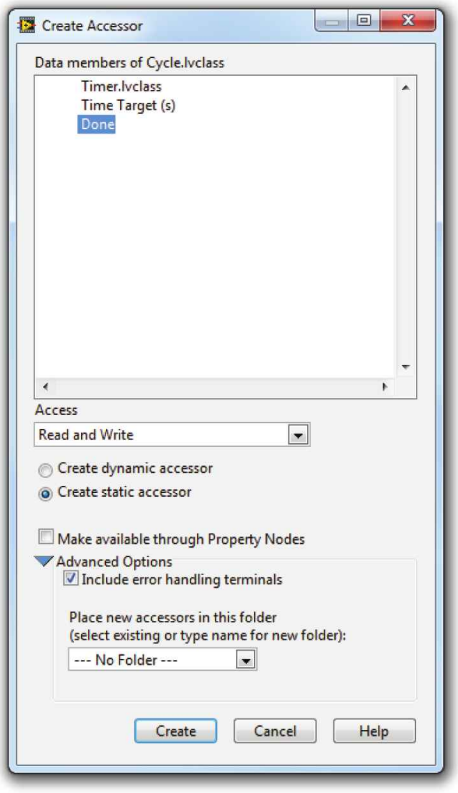


Figure 25.5: Creating data member access VIs for the *Done* Boolean

When the window shown in Figure 25.5 is exited, two data member sub-VI are created since the option was selected to create both read and write method sub-VIs. The icon of the `Read Done` sub-VI is shown in Figure 25.7, the front panel is shown in Figure 25.6, and the block diagram is shown in Figure 25.8. The new sub-VI is created with inputs and outputs for a `Cycle` object and with error input and output terminals.



Figure 25.6: The icon of the automatically generated `Read Done` VI

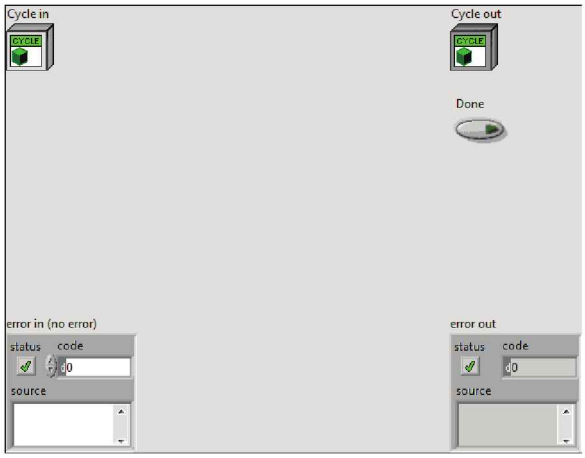


Figure 25.7: The front panel of the automatically generater `Read Done` VI

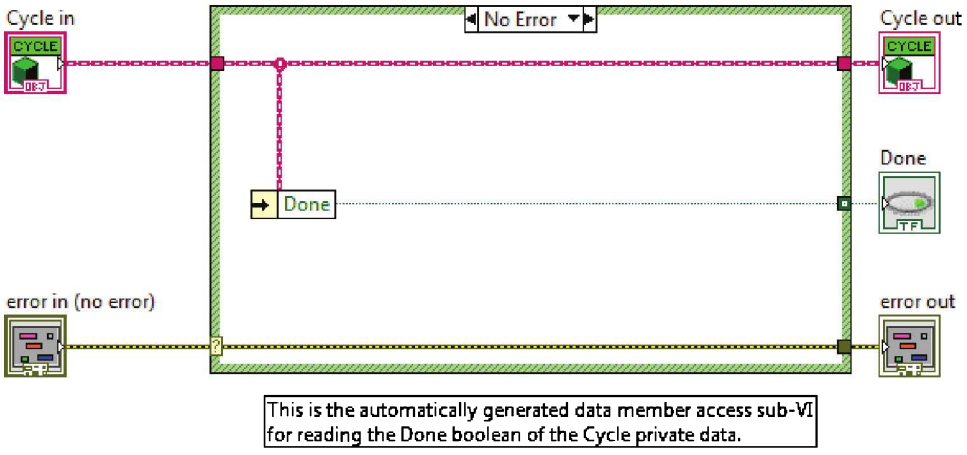


Figure 25.8: The block diagram of the automatically generater `Read Done` VI



Figure 25.9: The icon of the automatically generated `Write Done` VI

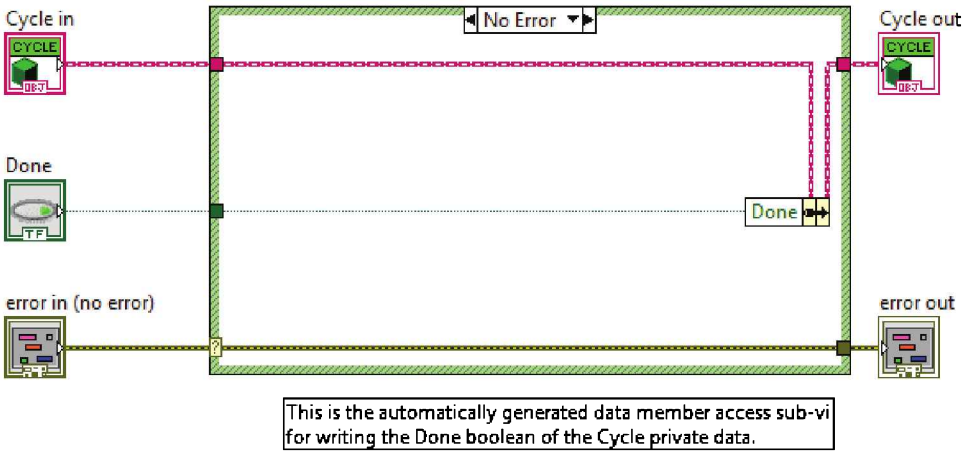


Figure 25.10: The block diagram of the automatically generater `Write Done` VI

The sub-VI also includes a Boolean indicator as an output since it was created for the purpose of reading the value of the *Done* Boolean in the private data. The `Road Done` block diagram is shown in Figure 25.8.

The `Write Done` sub-VI is also automatically generated. The icon is shown in Figure 25.9, and the block diagram is shown in Figure 25.10. The value of the input *Done* Boolean control is bundled into the private data of the `Cycle` object. LabVIEW gives you the option to represent the read and write data member access sub-VIs as property nodes on the block diagram. This option is not used here.

25.4 Dynamic Dispatch Sub-VIs

Dynamic dispatch sub-VIs are a key feature of inheritance. Dynamic dispatch sub-VIs are defined in a parent class and are inherited by children of the class—just like static dispatch methods. Unlike static dispatch methods, the child class can choose to use the parent method or it can override it with an identically named method of its own.

Let's use the `Cycle` class as an example. The generic `Cycle` class defines two dynamically dispatched methods: one to select a particular switch from the switch cluster and one to light a particular lamp in the cycle indicator cluster. These methods act as placeholders. The methods are inherited by the child car wash cycle classes and are overridden by methods defined in the child car wash cycle classes.

To create a dynamic dispatch method, right-click on the `.lvclass` entry of a class library and select *New... VI from Dynamic Dispatch Template*. This automatically creates a dynamic dispatch sub-VI. The dynamic dispatch sub-VI is very similar to the automatically generated static dispatch sub-VIs seen in Figures 23.13 and 23.15. The key difference lies in the connector pane. The connector pane of the dynamic dispatch sub-VI is shown in Figure 25.11. The red "X's" in the upper left and right terminals indicate that this is a dynamic dispatch sub-VI.

The new dynamic dispatch sub-VI then can be developed like any other sub-VI. The icon for the `Get Switch` method of the `Cycle` class is shown in Figure 25.12, and the front panel is shown in Figure 25.13. Since this is a member of the `Cycle` class, the method accepts a `Cycle` object as an input.

The block diagram of the `Get Switch` method is shown in Figure 25.14. This sub-VI accepts the cluster of car wash simulation switches, chooses one, and sends the value to an indicator. It doesn't matter which of the switches is selected in this sub-VI, since it will be overridden by all of the child classes.

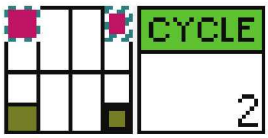


Figure 25.11: The connector pane of a new dynamic dispatch sub-VI

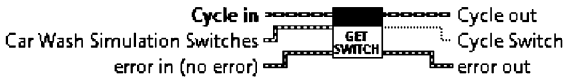


Figure 25.12: The icon of the dynamically dispatched `Get Switch` VI

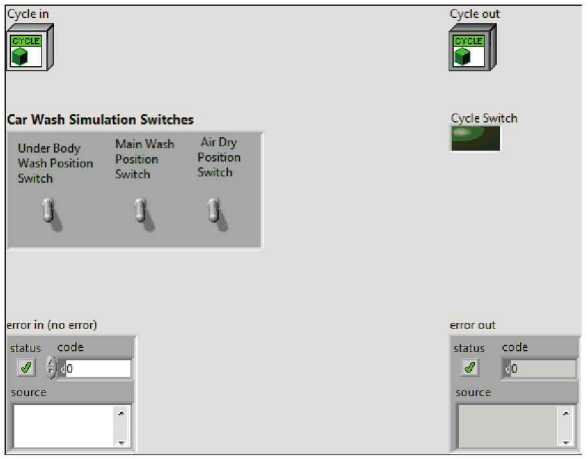


Figure 25.13: The front panel of the dynamically dispatched `Get Switch` VI

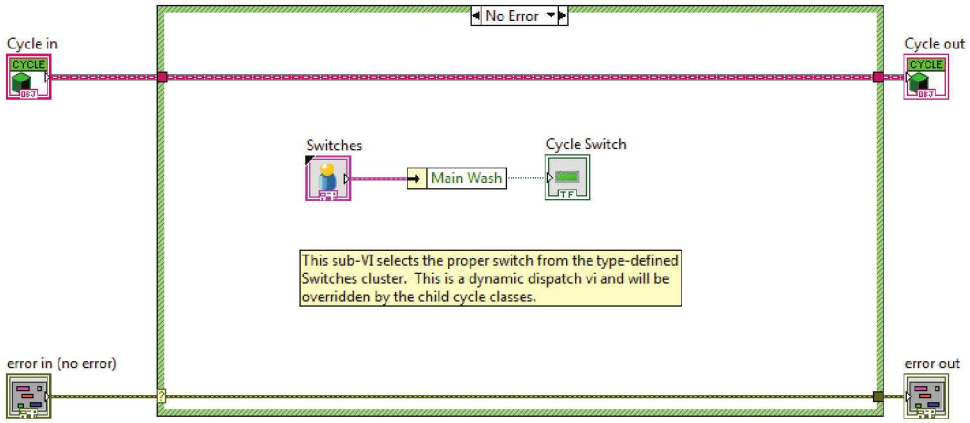


Figure 25.14: The block diagram of the dynamically dispatched `Get Switch` VI

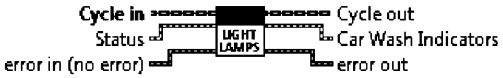


Figure 25.15: The icon of the dynamically dispatched `Light Lamps` VI

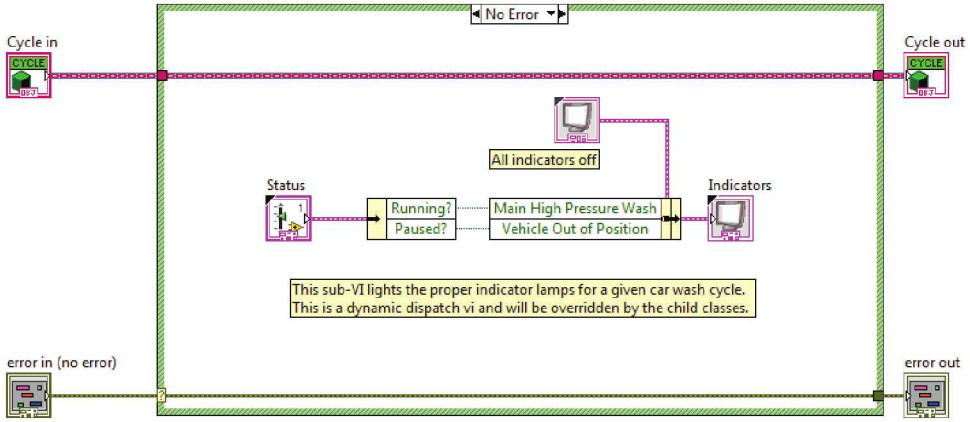


Figure 25.16: The block diagram of the dynamically dispatched `Light Lamps` VI

A dynamic dispatch sub-VI also needs to be developed to light the correct car wash cycle indicators. The icon of the `Light Lamps` method of the `Cycle` class is shown in Figure 25.15, and the block diagram is shown in Figure 25.16. This sub-VI accepts the status cluster from `Time Check` method of the `Timer` class and uses it to indicate if the cycle is running and if the vehicle is out of position. Again, it doesn't matter which of the cycle lamps is chosen in this sub-VI, since it will be overridden by all of the child classes.

25.5 Static Dispatch Sub-VIs

Static dispatch sub-VIs cannot be overridden. They are inherited and used "as-is" by child classes. They were introduced in the last chapter when the `Timer` class was developed. The `Timer` class had no child classes, so the method sub-VIs were developed with the static dispatch template.

Static dispatch methods are convenient because they only have to be defined in the parent class. They are appropriate when the action taken is the same for both parent and child. For example, Figure 25.2 shows that the child car wash cycles have the same private data as their parent class. This means that a sub-VI that initializes the private data of the parent class also can be used to initialize the private data of the child classes. This makes the initialization sub-VI a good candidate for static dispatch.

The icon for the `Initialize Cycle` method is shown in Figure 25.17 and the block diagram is shown in Figure 25.18. This method can be used by a `Cycle` object or any of its children. The sub-VI sets the private data to their initial values: the `Timer` object is initialized, the *Done* Boolean is set to False, and the time target is set to the given value.

Even though this is a simple sub-VI, it has some interesting features. Notice that the timer is reset using the `Initialize Timer` method of the `Timer` class. Initializing the timer changes the private data of the timer. This means that after the initialization is executed, the `Timer` object needs to be bundled back into the the private data of the `Cycle` object to preserve the initialized timer data.

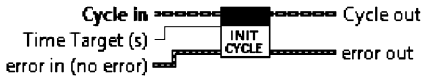


Figure 25.17: The icon of the dynamically dispatched `Initialize Cycle` VI

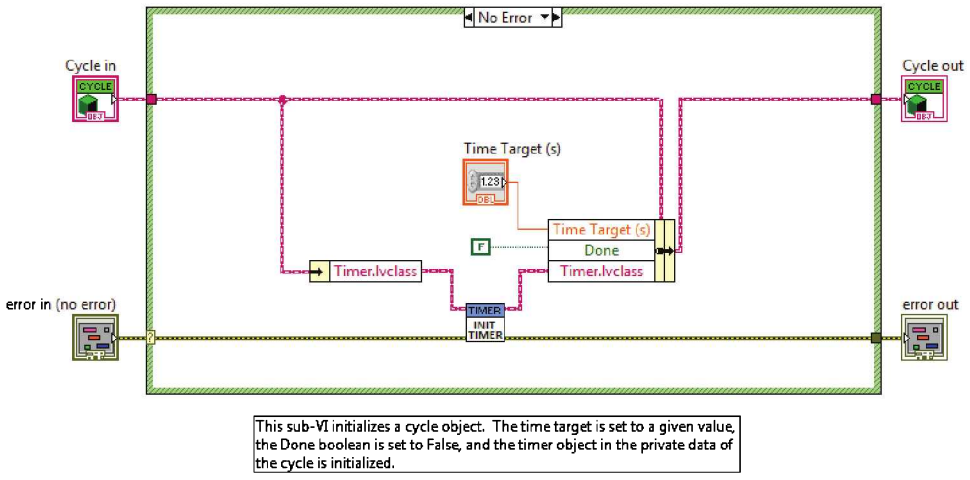


Figure 25.18: The block diagram of the dynamically dispatched `Initialize Cycle` VI

This sub-VI is another example of the use of composition and encapsulation. The `Cycle` class is designed in such a way that the `Timer` object is encapsulated and inaccessible. Since the timer is in the private data with no data member access sub-VIs, it cannot be directly accessed by a VI outside of the class. The timer must be initialized using the designated `Cycle` class method, `Initialize Cycle`.

Notice that in Figure 25.18 the *Done* Boolean is set to False without using the data member access sub-VI or property node. Remember that data member access methods are only needed by VIs outside the class. Since the `Initialize Cycle` method is a member of the `Cycle` class, it can bundle or unbundle the *Done* Boolean from the private data cluster without needing a data member access method.

The time target of the cycle is also set in the `Initialize Cycle` method. This makes the `Cycle` object self-contained. When a time check is performed, the only information that needs to be sent to the `Cycle` object is the value of the proximity switch. The time target is read from the `Cycle` object itself.

The next static dispatch method to be developed is the `Cycle Time Check` method. This method uses the time check method of the `Timer` object in its private data. The icon for this sub-VI is shown in Figure 25.19, and the block diagram is shown in Figure 25.20. This method sub-VI accepts a proximity switch and a `Cycle` object as inputs. It unbundles the `Timer` object and the time target and then passes them along with the proximity switch value to the `Time Check` method of the `Timer` object. The *Done?* Boolean output of the time check is also bundled into the *Done* Boolean in the `Cycle` object's private data. The status cluster is passed as an output of the sub-VI.

We can now finally develop the `Run Cycle` method seen in the block diagram of Figure 25.1. This method sub-VI accepts the cluster of proximity switches and a

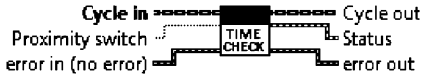


Figure 25.19: The icon of the statically dispatched `Cycle Time Check` VI

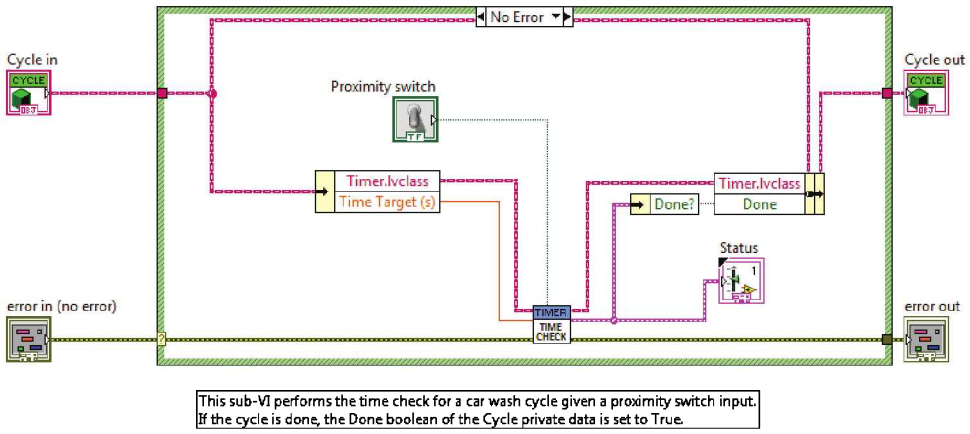


Figure 25.20: The block diagram of the statically dispatched `Cycle Time Check` VI

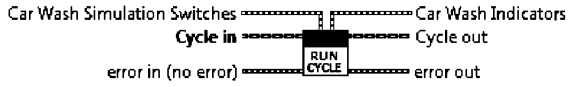


Figure 25.21: The icon of the statically dispatched `Run Cycle` VI

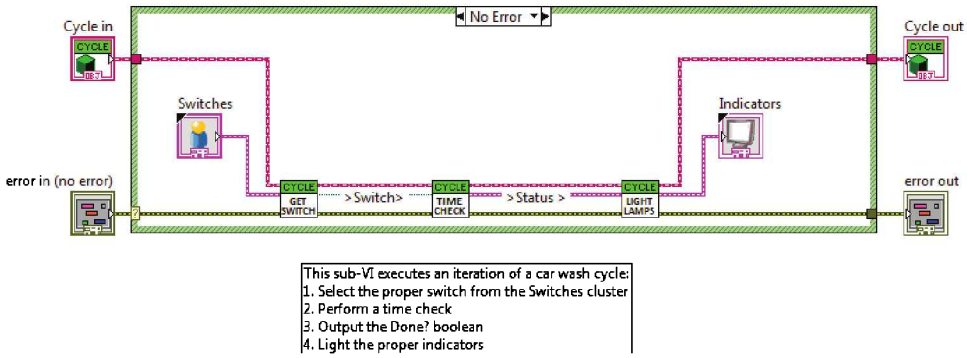


Figure 25.22: The block diagram of the statically dispatched `Run Cycle` VI

`Cycle` class object as inputs. It outputs the correct indicator cluster and the modified `Cycle` object. The `Run Cycle` method uses dynamic dispatch sub-VIs to select the proper switch and indicator lamps, but is itself a static dispatch method. Child classes will call this method without overriding it.

The icon of this method sub-VI is found in Figure 25.21. The block diagram is shown in Figure 25.22. Notice how clean the block diagram is. The `Get Switch` dynamic dispatch sub-VI selects the proper proximity switch from the switches cluster. The value of the switch is then passed to the static dispatch `Run Cycle` sub-VI to perform the time check. Then the dynamic dispatch `Light Lamps` sub-VI lights the proper indicators. All of these sub-VIs are are methods of the `Cycle` class. The `Run Cycle` method can now be inherited as a static dispatch VI and used by any of the child classes, with the child classes supplying their own override methods to select the proper switch and indicator lamps. If new child classes are developed in the future, they will also inherit the ability to run the car wash cycle even though the specifics of the child classes were unknown when the parent class was created.

25.6 Access Scope

In Chapter 23, we talked about data encapsulation. The basic idea is to hide the data and to provide a public interface to use the data without exposing the data itself. As long as the public interface remains unchanged, changes made to the way the hidden data is represented and stored have no effect on the outside world.

This concept can be extended to method sub-VIs as well. Consider the three static dispatch sub-VIs developed in this chapter. Two of the three sub-VIs are intended to be

used by VIs outside the class: `Initialize Cycle` and `Run Cycle`. But the `Cycle Time Check` method is not meant to be used by the public, it is a support sub-VI for the `Run Cycle` method. The `Cycle Time Check` VI needs to be part of the `Cycle` class since it needs access to the `Cycle` private data, but it does not need to be used as a stand-alone sub-VI by the public. In a situation like this, it is beneficial to "hide" the sub-VI by making it private. Public methods can be called by any VI, but a private method can only be called by VIs that are members of the same class.

A private VI has the same advantage that private data enjoys. The private VI is accessible only through the public VIs that use the private VI. As long as the connector pane and associated inputs and outputs of the private VI remain constant, changes made to the private VI have no effect on the outside world.

The degree of exposure of a method VI is called its *access scope*. LabVIEW provides four levels of access scope for method VIs: public, private, protected, and community. The default scope is public. A public method can be used by any VI inside or outside of the class.

Private VIs only can be used by members of their own class, while protected VIs can be used by members of the same class or by child classes. There is a subtle distinction here depending on whether the calling VI is statically or dynamically dispatched. Let's use the `Cycle` class as an example. The `Run Cycle` VI is public, and it calls the `Cycle Time Check` VI, which is private. Both of these VIs are statically dispatched. Now assume that the `Run Cycle` sub-VI is run on a `Rinse` object, which is a child of the `Cycle` class. Since the `Run Cycle` sub-VI is statically dispatched, the actual VI that will be run is the `Run Cycle` VI of the `Cycle` class. There is no `Run Cycle` VI of the `Rinse` class; the `Rinse` object uses the `Run Cycle` VI it inherits from its parent class. In this case, there is no problem when the `Run Cycle` VI calls the private `Cycle Time Check` VI, since they are both from the same class.

Now imagine a case where the `Run Cycle` VI is dynamically dispatched. In this case, the `Rinse` object would override the `Cycle` version of the `Run Cycle` VI with the `Rinse` version of the `Run Cycle` VI. The `Rinse` version of the `Run Cycle` VI is technically not part of the `Cycle` class, so it would not be able to call the private `Cycle Time Check`. This is where the protected access scope is useful. A protected method sub-VI can be called by members of the same class or by the overriden VIs of child classes.

The protected scope is also useful if child classes have extra methods not found in the parent class. Suppose that the `Rinse` class had a method called `Extra Time` that extends the rinse duration if needed, and suppose further that the `Extra Time` method needs to call the `Cycle Time Check` method. This will not work if the access scope of the `Cycle Time Check` VI is private, since the `Extra Time` method is not part of the `Cycle` class. It will work if the access scope of the `Cycle Time Check` VI is protected, since it is being called by a member of a child class of the `Cycle` class.

The last access scope is the community scope. The community scope allows one class to "friend" another class. Community scope methods of a class can be used by members of that class or by friends of that class.

Dynamically dispatched methods cannot have community scope, and "friending" is not transitive. If Class B is a friend of Class A and Class C is friend of Class B, then Class C is not automatically a friend of Class A. Friendship also does not automatically

extend to descendants of a class. If Class B is a friend of Class A and Class C is a descendant of Class B, Class C is not automatically a friend of Class A.

To set the access scope of a method sub-VI, right-click on the entry for that sub-VI in the class library and select *Access Scope*. Select the desired scope from the menu. The selected scope is indicated by a small key-shaped glyph on the icon next to the method name in the class library. Icons with no glyph are public, a red key indicates a private method, a dark yellow key indicates a protected method, and a dark blue key indicates a community method.

25.7 The Completed `Cycle` Class

The `Cycle` class library is now complete and can be seen in Figure 25.23. Virtual folders have been used to organize the class library. Virtual folders exist only for display purposes and do not reflect the file organization on disk. Virtual folders can be created by right-clicking on the `.lvclass` entry of the class library and selecting *New... Virtual Folder*.

Figure 25.23 has features that will be common to many class libraries:

- The private data control appears first and is part of every class library. Notice the red key icon indicating that the control is private and only can be accessed by members of the class or its descendants.

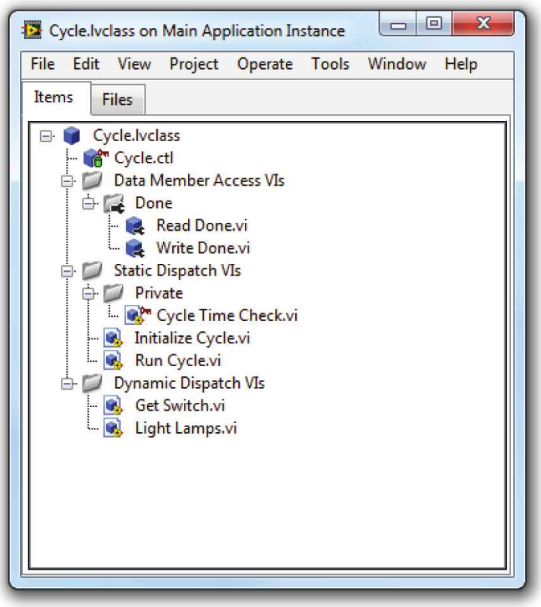


Figure 25.23: The completed of the `Cycle` class

- Data member access sub-VIs are defined for elements of the private data that need to be accessed from outside the class.
- Statically dispatched sub-VIs can be called "as-is" by members of the class or its descendants.
- Dynamically dispatched sub-VIs can be called "as-is" by members of the class or its descendants, or descendants can choose to override them with their own methods of the same name and connector pane.
- Private sub-VIs can be accessed only by other members of the same class. The red key icon signals that the access scope is private.

25.8 Creating the Child Classes

Now that the parent class has been created, it is time to create the child classes. The child classes are much easier to create, since they leverage the effort used in creating the parent class. Each child class requires the following steps.

1. Create the child class library.
2. Customize the child class icon.
3. Configure the class to inherit from the `Cycle` class.
4. Create the override VI for the `Get Switch` method.
5. Create the override VI for the `Light Lamps` method.

We use the `Underbody Wash` class as an example. The first two steps, creating the new class library and customizing the icon, should be familiar by now. The same steps are followed as in the `Timer` and `Cycle` classes. The new class has an empty private data cluster. The child car wash cycle classes do not add any private data elements to the `Cycle` class private data, so the private data cluster of the `Underbody Wash` class is empty.

Recall that the class icon was customized by bringing up the properties window for the class by right-clicking on the `.lvclass` entry of the class library and selecting *Properties*. The inheritance of the class is set using the same properties window. The *Inheritance* tab of the window is shown in Figure 25.24. In this window, you are given the opportunity to make the current class a child of any previously defined class. To change the inheritance of the `Underbody Wash`, select the `Cycle` class in the inheritance tree.

The ultimate ancestor at the top of the inheritance tree is the `LabVIEW Object`. The `LabVIEW Object` is the default parent of any new LabVIEW class and is the ultimate ancestor of every LabVIEW class. This means that the `LabVIEW Object` is the parent of both the `Timer` class and the `Cycle` class. The `LabVIEW Object` is the parent of the `Cycle` class, and the `Cycle` class is the parent of the `Underbody Wash` class. Thus, the `LabVIEW Object` is the ultimate ancestor of the `Underbody Wash` class. This means that the `Underbody Wash` class can travel on wires created to carry a `LabVIEW Object` as well as wires created to carry a `Cycle` object.

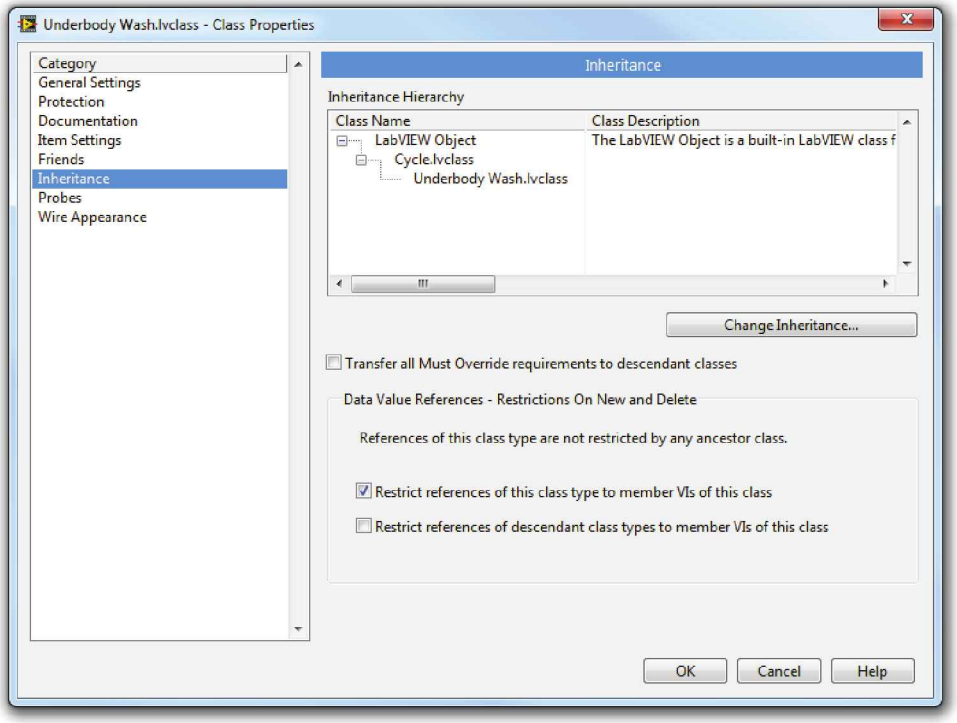


Figure 25.24: The `Underbody Wash` class inherits from the `Cycle` class

25.9 Developing Override VIs

The next step is to develop the override VIs. This can be done by right-clicking on the `.lvclass` entry of the class library and selecting *New... VI for Override*. This brings up a window like the one shown in Figure 25.25. This window allows you to select which of the dynamic dispatch methods of the parent class to override.

After exiting this window, the override VI is automatically generated. The front panel, icon, and connector pane are all correct and useful, but unfortunately, the automatically generated block diagram is not. The block diagram for the override of the `Get Switch` method is shown in Figure 25.26. The block diagram is set up to call the parent method and uses the **`Call Parent Method Node`** function, with the help file shown in Figure 25.27.

In order to override the parent method, the block diagram needs to be re-built to look like the one shown in Figure 25.28. The icon for the override sub-VI is shown in Figure 25.29. The override sub-VI must have the same connector pane and name as the parent method to be overridden. Normally, LabVIEW prevents two different sub-VIs from having the same name. However, the names of the parent

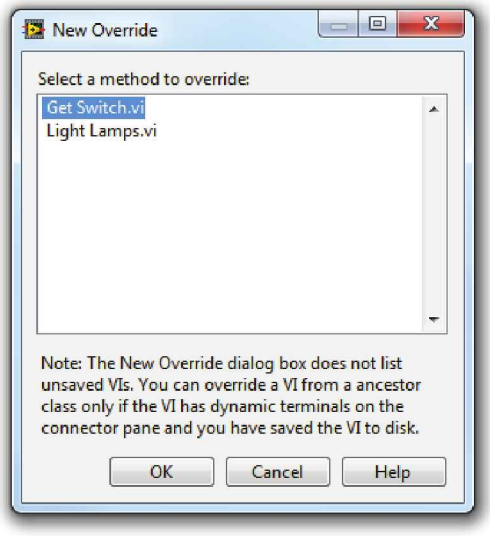


Figure 25.25: Creating a VI to override a parent method

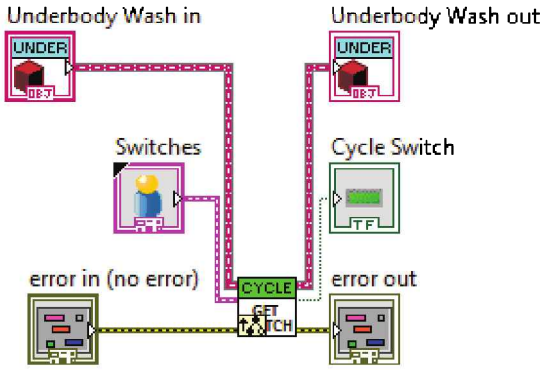


Figure 25.26: The automatically generated VI for override calls the nearest ancestor's method

method and the override VI can be the same, because LabVIEW gives each class a protected name space by internally appending the class name to the method name. Thus, the `Cycle.lvclass:Get Switch.vi` method VI is distinct from the `Underbody Wash.lvclass:Get Switch.vi`.

Wiring Tip: The override VI can also be created using the dynamic dispatch template. This provides a more useful block diagram, but the connector pane must be wired manually to match that of the parent dynamic dispatch VI. The icon also needs to be customized. A third approach for creating an override VI is to save a copy of the parent dynamic dispatch VI and add the copy to the class library of the child class. The copy would then have to be customized to accept the child object rather than the parent object, and the child class icon would need to be used instead of the icon from the parent class.

The `Underbody Wash` override of the `Light Lamps` method is similar. The icon for this override VI is shown in Figure 25.30, and the block diagram is shown in Figure 25.31. Notice that both of the override VIs select the switch and indicators that are needed by the *Underbody Wash* car wash cycle.

The `Underbody Wash` class is now complete. The class library and private data cluster are shown in Figure 25.32. This process must be repeated to create the other four

Call Parent Method Node

Owning Palette: Cluster, Class, & Variant VIs and Functions

Requires: Base Package

Calls the nearest ancestor implementation of a class method. You can use the Call Parent Method node only on the block diagram of a member VI that belongs to a class that inherits member VIs from an ancestor class. The child member VI must be a dynamic dispatching member VI and have the same name as the ancestor member VI.



Add to the block diagram Find on the palette

The parameters, or input and output terminals, of this node depend on the ancestor member VI and change to match the ancestor member VI exactly. This node adopts the VI icon of the ancestor member VI with a glyph in the bottom, left corner of the subVI to indicate the subVI is calling the ancestor member VI instead of dynamically dispatching as a dynamic subVI node does typically.

Figure 25.27: Help file for the `Call Parent Method Node` function

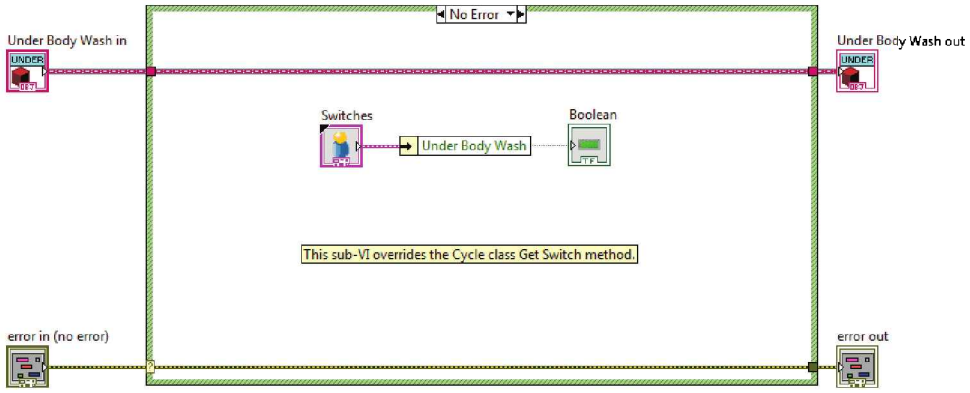


Figure 25.28: The block diagram of the `Underbody Wash` class override of the `Get Switch` VI

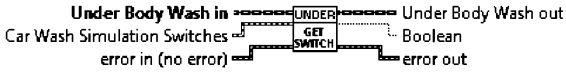


Figure 25.29: The icon of the `Underbody Wash` class override of the `Get Switch` VI

car wash cycle classes. The override VIs will be identical to those of the `Underbody Wash` class with each VI customized to choose the switches and indicators needed for its particular cycle. For comparison, the icon of the `Soap Application` version of the `Get Switch` is shown in Figure 25.33, and the block diagram is shown in Figure 25.34. The completed class libraries for the five child car wash cycles are shown in Figure 25.35.

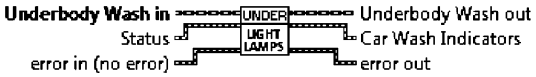


Figure 25.30: The icon of the `Underbody Wash` class override of the `Light Lamps` VI

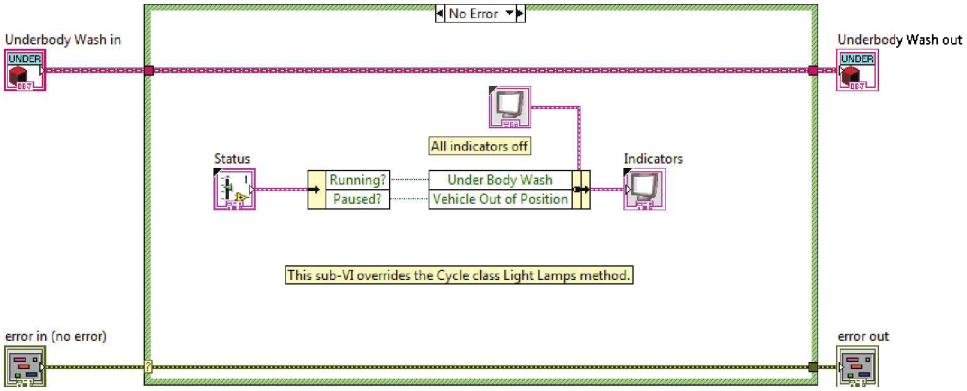


Figure 25.31: The block diagram of the `Underbody Wash` class override of the `Light Lamps` VI

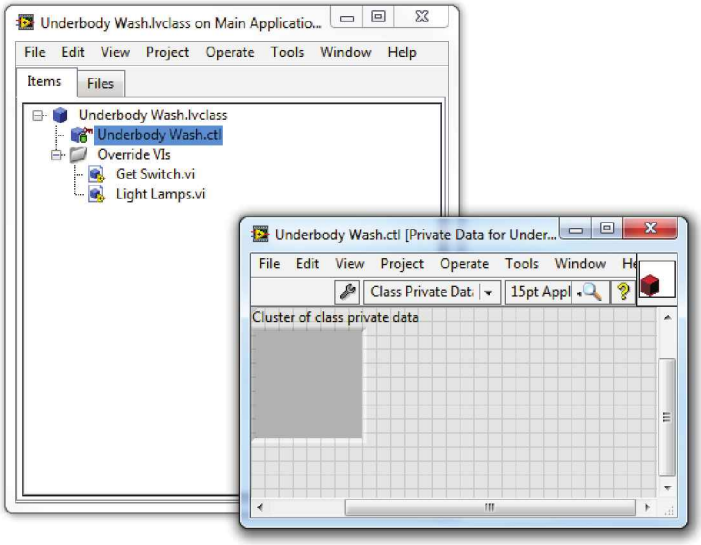


Figure 25.32: The completed library of the `Underbody Wash` class

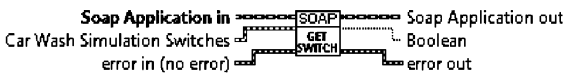


Figure 25.33: The icon of the `Soap Application` class override of the `Get Switch` VI

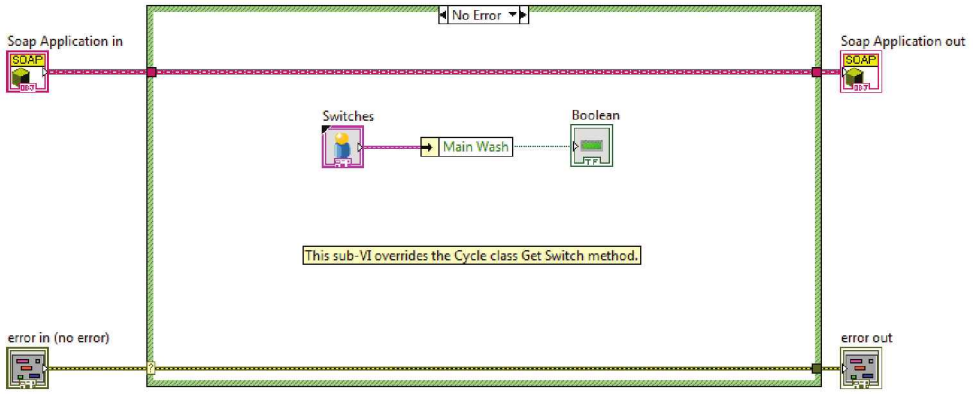


Figure 25.34: The block diagram of the `Soap Application` class override of the `Get Switch` VI

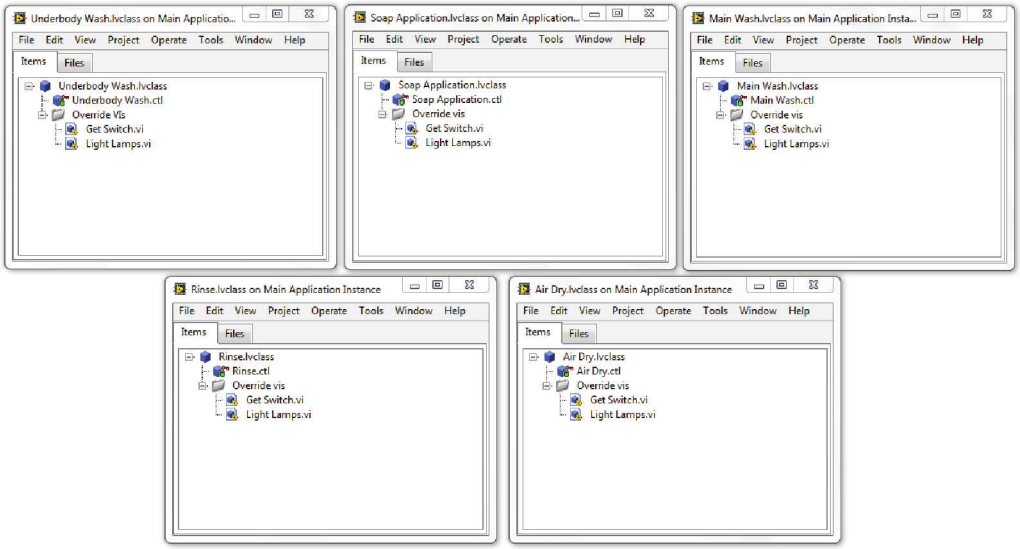


Figure 25.35: The class libraries of the five child car wash classes

The completed class hierarchy can be examined by selecting *View...LabVIEW Class Hierarchy* from the LabVIEW toolbar. The class hierarchy shows the inheritance relationships of the classes in memory. The class hierarchy for the car wash controller is shown in Figure 25.36. The item at the top of the hierarchy is the LabVIEW Object, which is the ultimate ancestor of every LabVIEW class. Compare this hierarchy to the class diagram shown in Figure 25.2.

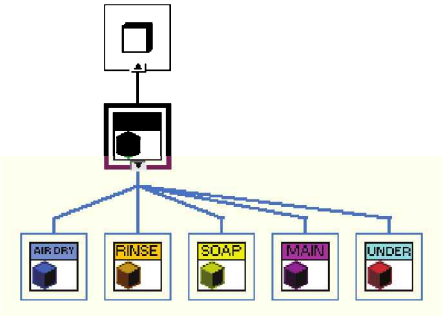


Figure 25.36: The completed class hierarchy of the car wash controller

25.10 The Factory Method

Now that the `Cycle` class and all of its children have been developed, there is only one last sub-VI that needs to be developed before implementing the new car wash controller, which is the `Cycle Factory` sub-VI seen in Figure 25.1. The purpose of this sub-VI is to generate objects from the five `Cycle` child classes: `Underbody Wash`, `Soap Application`, `Main Wash`, `Rinse`, and `Air Dry`. The factory checks to see if the current `Cycle` object is done. If it was not done, the factory passes it to the `Run Cycle` VI to execute the time check. But if the `Cycle` object is done, a new object is generated as dictated by the argument string.

LabVIEW objects are typically created by dropping an object constant on the block diagram. In our case, we need to be able to parse the argument string and use the cycle name to generate the correct object. One way to do that is to create a sub-VI similar to the mapping sub-VIs developed in Chapter 20. The `Cycle Factory` contains a case structure keyed to the cycle name. Each case of the structure corresponds to a possible car wash cycle, and each case outputs an object for that cycle. Because all of the car wash cycle objects are descendants of the `Cycle` class, they all can be passed through a single output tunnel of the case structure if the output tunnel has the data type of the `Cycle` class.

This type of scheme is known as a factory pattern. Object-oriented programmers have recognized that certain types of problems crop up again and again in object-oriented codes. The best-practice solutions to these problems have been codified as "patterns." A pattern is not a piece of code, but rather, it is a description of a good solution to a common problem.

Most programs that include a group of related objects need to be able to instantiate objects based on current conditions in the program. These objects usually need to be initialized before they are used. One solution to this problem is to encapsulate the object creation and initialization process in a separate piece of code—perhaps

in a sub-VI or in another class. This solution to the customized object creation and initialization problem is known as the factory pattern.

Rather than create a new class, we use a simple sub-VI called `Get Cycle Object` to generate the car wash cycle objects. The icon for this sub-VI is shown in Figure 25.37, and the front panel is shown in Figure 25.38. This sub-VI accepts a string containing the cycle name and generates the corresponding object. The front panel shows that the output of the `Get Cycle Object` is a `Cycle` object. This allows the sub-VI to pass any of the child objects as an output, since child objects can travel on wires with the data type of their parents.

The block diagram of the `Get Cycle Object` sub-VI is shown in Figure 25.39. This sub-VI contains a case structure with a string-based selector. The figure shows



Figure 25.37: The icon of the `Get Cycle Object` VI

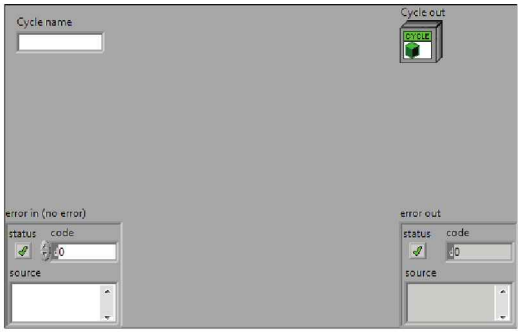


Figure 25.38: The front panel of the `Get Cycle Object` VI

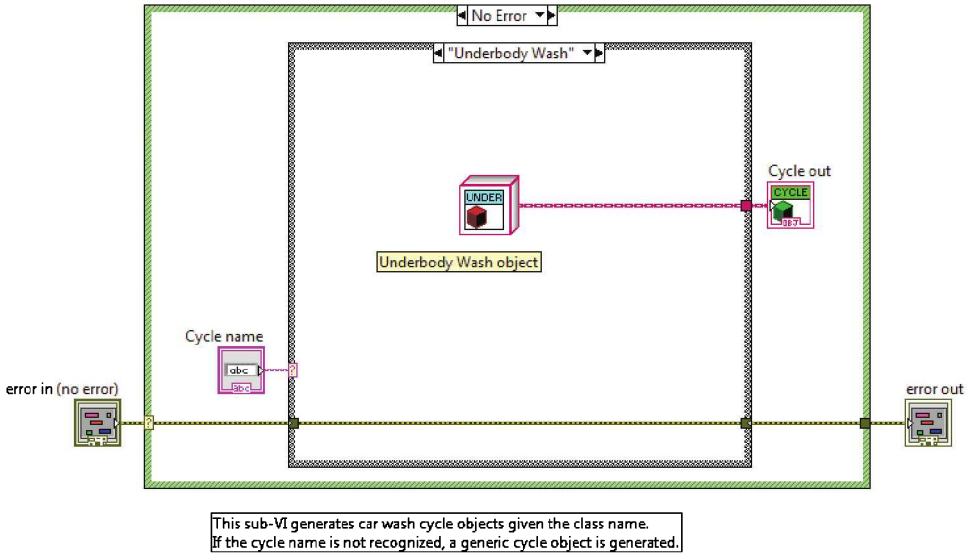


Figure 25.39: The block diagram of the `Get Cycle Object` VI showing the *Underbody Wash* case

the *Underbody Wash* case of the structure. The code within the case consists solely of an `Underbody Wash` object. The object passes through the output tunnel along a wire to the object indicator. The tunnel, the wire, and the object indicators are all of the `Cycle` class data type.

The cases for the other child classes are similar. Each case generates an object for the corresponding cycle. The objects all travel along the same wire to the same output indicator. Since the case structure selector is string-based, the case structure needs a default case as shown in Figure 25.40. The default case occurs when an unknown class is requested. The default case generates a custom error with error code 5010 to distinguish it from the custom error code 5000 that we have been using to denote an error due to an unknown state. A generic `Cycle` state is output if an error is detected.

The next step is to build the `Cycle Factory` sub-VI. The icon of the `Cycle Factory` sub-VI is shown in Figure 25.41. The block diagram is shown in Figure 25.42. The factory examines the current cycle object and checks to see if it is done. If the cycle is done, the `Parse Name and Time Target` sub-VI developed in Chapter 21 is used to extract the cycle name and time target from the argument string. The `Get Cycle Object` sub-VI is then used to generate an object for the new cycle. The object is then initialized using the time target that was parsed from the input string. If the current cycle is not done, the `Cycle Factory` sub-VI simply passes the current cycle to the output indicator, as seen in Figure 25.43.

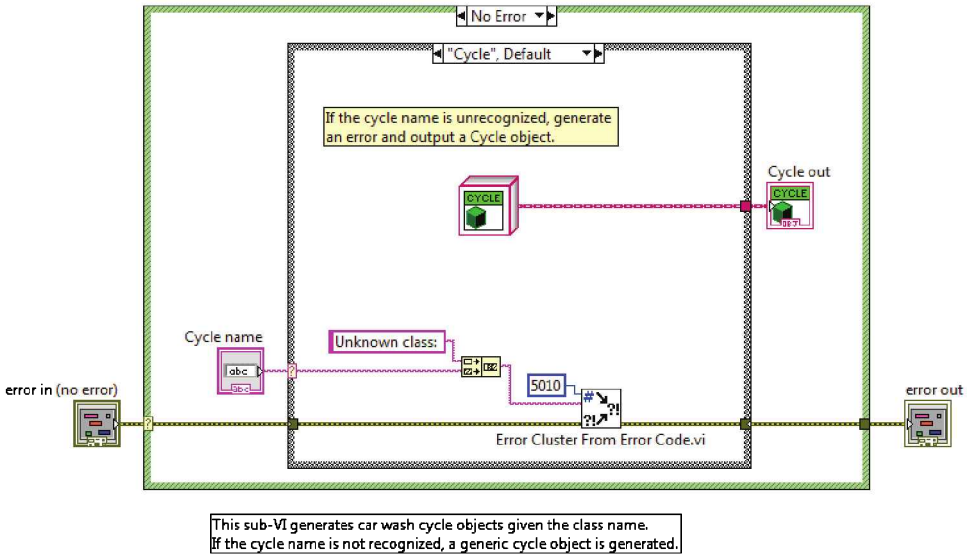


Figure 25.40: The block diagram of the `Get Cycle Object` VI showing the *Default* case

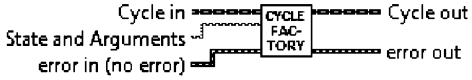


Figure 25.41: The icon of the `Cycle Factory` VI

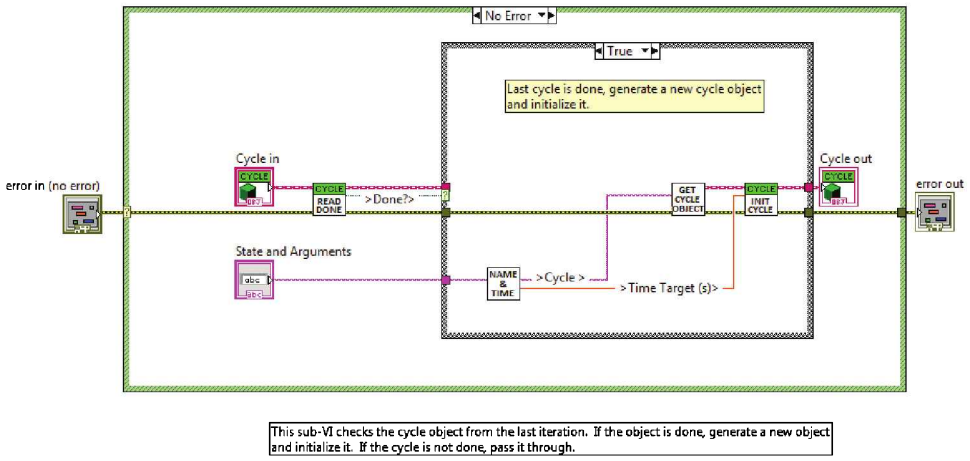


Figure 25.42: The block diagram of the `Cycle Factory` VI showing the case that executes when the cycle is done

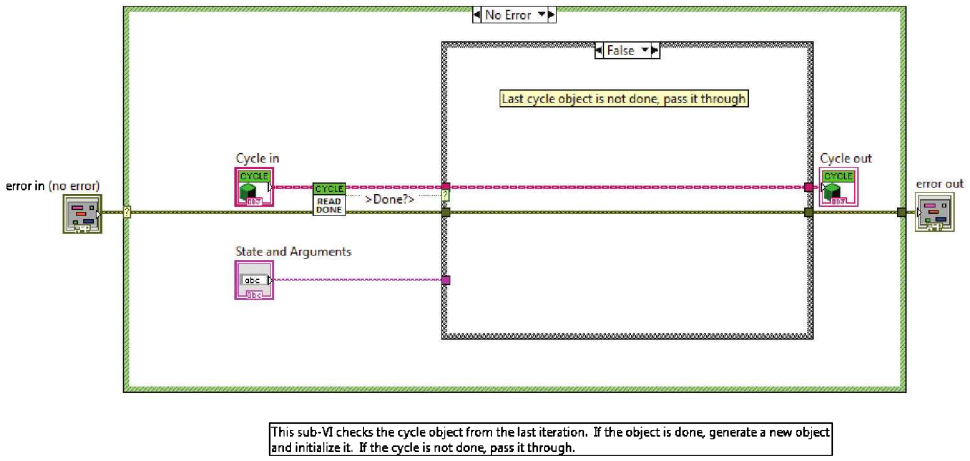


Figure 25.43: The block diagram of the `Cycle Factory` VI showing the case that executes when the cycle is not done

The `Cycle Factory` sub-VI accepts a `Cycle` object as an input and outputs a `Cycle` object as well. This means that it can both accept any child object of the `Cycle` class as an input and pass any child class as an output. Even so, this VI does not belong to any class. The `Cycle Factory` does not need to access the private data of any class, and it only executes public methods of other classes (the *Initialize* method). Therefore, it does not need to be a member of the `Cycle` class or any of it its child classes.

25.11 Dynamic Loading of Classes

The `Cycle Factory` VI is functional, but it is not very scalable. The problem lies in the `Get Cycle Object` sub-VI. This sub-VI contains an object of every car wash cycle object. If a new car wash cycle is added in the future, the `Get Cycle Object` sub-VI will have to be updated with an instance of the new cycle object.

There is also the potential for a performance issue. When a VI contains objects from classes, all of the classes are loaded into memory when the VI is loaded into memory. This process is called static loading. For VIs that use a small number of classes, this is not an issue, but it can cause delays when loading a VI that uses a large number of classes. The car wash controller uses a small number of classes, so performance is not an issue. But to illustrate the problem, imagine that the car wash controller runs all day and every customer chooses the Economy Wash. Even though the `Underbody Wash` class and the `Air Dry` class are never used, they are still loaded into memory when the car wash controller is first started and stayed resident in memory during the entire operation of the car wash.

Both of these issues can be addressed by using dynamic loading, instead of static loading. With dynamic loading, classes are not loaded into memory until they are are actually needed. This solves the performance issue. Even better, a dynamically loaded class is loaded into memory by specifying the path to the class. Since a string can be converted into a path, a class can be loaded into memory simply by knowing its name if a naming convention is used. This is true even if the developer doesn't know the names of classes that will be developed in the future, as long as the future names follow the naming convention. This means that a dynamically loading factory sub-VI does not need to be updated if new classes are created in the future.

Let's develop a sub-VI that uses dynamic loading of classes to replace the statically loading `Get Cycle Object` VI shown in Figure 25.39. Dynamic loading of classes is accomplished with the native `Get LV Class Default Value` function. The help file for this function is shown in Figure 25.44. This function accepts a path to a LabVIEW class as an input and generates an instance of the class (a class object constant) as an output.

We need a file path convention that allows us to generate an absolute file path for a class given the name of the class. For example, suppose each car wash class is stored in its own folder and the name of the folder is the same as the name of the class. Further suppose that the class folder is stored in the same directory as the car wash controller main VI. Then the path to the `Rinse` class would be `Application Directory\Rinse\Rinse.lvclass`, and the path to the `Main Wash` class would be `Application Directory\Main Wash\Main Wash.lvclass`. Using this naming convention, we can develop a `String to Path` sub-VI that will generate the path to a class given its name.

The icon for the `String to Path` VI is shown in Figure 25.45, and the block diagram is shown in Figure 25.46. This VI accepts the name of a car wash class as a string input and uses the **`Format Into String`** function to create a string containing the relative path to the class (see the help file in Figure 1.60). For example, the input string `Rinse` produces the output string `Rinse\Rinse.lvclass`. This is done using

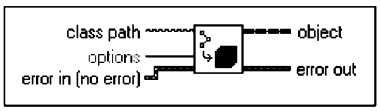
Get LV Class Default Value VI

Owning Palette: Cluster, Class, & Variant VIs and Functions

Requires: Base Package

Returns the class data for the class whose path you wire to the **path** input. If the class is not in memory, LabVIEW attempts to load it from disk. If any dependencies are missing, LabVIEW searches for the missing dependencies. If LabVIEW cannot find the dependency, LabVIEW assumes the dependency is missing and does not prompt the user to browse to the needed file.

Use this VI to dynamically create class objects, including class objects that are unknown at edit time.



Add to the block diagram Find on the palette

class path specifies the path to the LabVIEW class.

options is a bit set that specifies the behavior of LabVIEW when loading the class into memory. The default is 0x0, which means that when loading the class, LabVIEW searches for missing dependencies but does not prompt the user to browse to missing dependencies. **options** can be a combination of the following values.

0x01	This option has no effect when loading LabVIEW classes. For VIs, this is the **Record Modifications** value. When dynamically loading classes, modifications are always recorded and may prompt you to save any unsaved changes when the class unloads. This behavior has been true since classes were introduced in LabVIEW 8.2. This behavior might change in a future version of LabVIEW.
0x02	This option has no effect when loading LabVIEW classes.
0x04	This option has no effect when loading LabVIEW classes.
0x08	This option has no effect when loading LabVIEW classes.
0x10	Prompt user to find missing dependent VIs or libraries of the referenced class.
0x20	Do not display the loading dialog box when searching for missing dependencies. **Note** This option does not affect whether LabVIEW prompts you to find the missing dependencies or not.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

object displays the data of the class wired to the **class path** input.

error out contains error information. This output provides standard error out functionality.

Figure 25.44: Help file for the `Get LV Class Default Value` function

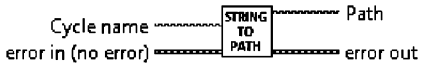


Figure 25.45: The icon of the `String to Path` VI

the format string `%s\\%s.lvclass`. The string tokens "%s" are placeholders for the class name and the "\\" characters are the backslash characters. Since the backslash is a special character in format strings, it needs to be escaped by preceding it with the escape character, which also happens to be the backslash.

This resulting relative path then needs to be converted into an absolute path. This is done using the **`Build Path`** and **`Application Directory`** functions introduced in Chapter 1. The **`Build Path`** function appends the relative path of the class to the path to the application directory, creating an absolute path to the class.

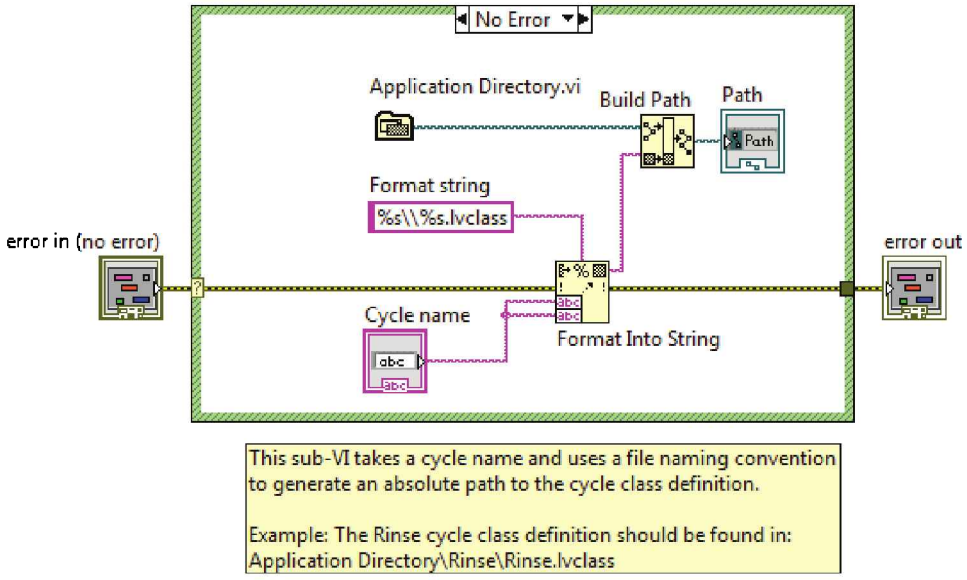


Figure 25.46: The block diagram of the `String to Path` VI



Figure 25.47: The icon of the `Get Cycle Object Dynamic Loading` VI

This sub-VI can now be used to dynamically load a class given the class name. The icon for the `Get Cycle Object Dynamic Loading` sub-VI is shown in Figure 25.47. This sub-VI accepts a string input containing the car wash cycle name and outputs the corresponding car wash cycle object using dynamic loading. The block diagram of this sub-VI is shown in Figure 25.48. The string input containing the name of the car wash cycle is converted to a path using the `String to Path` VI we just developed. The path is then passed to the **`Get LV Class Default Value`** function to instantiate an object of the named class.

A problem arises at this point. The **`Get LV Class Default Value`** function will accept the path to any LabVIEW class and produce the corresponding object. But the output of the function has a fixed data type. How can it pass any given object through a fixed data-type output, since each class has its own unique data type? The answer lies in the concept of inheritance. Remember that any LabVIEW object can travel on wires with the data type of any of its ancestors. Recall also that the `LabVIEW Object` is the ultimate ancestor of all LabVIEW classes. That means that any LabVIEW object can travel on a wire or be sent to an indicator that has the data type of the `LabVIEW Object`. Because of this, the **`Get LV Class Default Value`** function "upcasts" the output object to the data type of the `LabVIEW Object`.

But we would like the `Get Cycle Object Dynamic Loading` VI to output an object with the `Cycle` class data type. That way the resulting object can

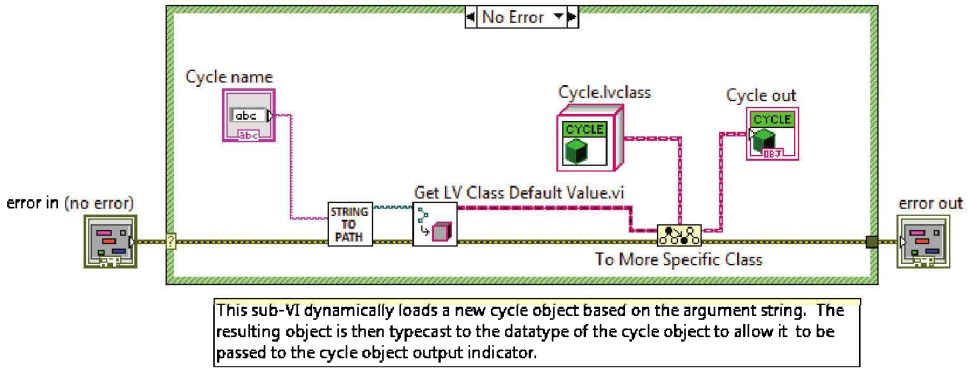


Figure 25.48: The block diagram of the `Get Cycle Object Dynamic Loading` VI

be passed to the `Run Cycle` VI, which only accepts `Cycle` objects. The way to do this is to "downcast" the `LabVIEW Object` produced by the **`Get LV Class Default Value`** function to the `Cycle` data type using the **`To More Specific Class`** function. The help file for the `To More Specific Class` function is shown in Figure 9.3. This function can be used to downcast an object to the data type of any of its descendants if possible.

Let's use the `Rinse` class as an example. The `Rinse` class inherits from the `Cycle` class, which in turn inherits from the `LabVIEW Object`. When a `Rinse` object is created by the **`Get LV Class Default Value`** function, it is upcast to the data type of its ultimate ancestor, the `LabVIEW Object`. In order to output this object as a `Cycle` object, it is downcast to the `Cycle` class data type by the **`To More Specific Class`** function. When this object is eventually passed to the `Run Cycle` VI, LabVIEW looks at its most specific class—the `Rinse` class—to determine which override VI to call when running the `Get Switch` and `Light Lamps` VIs.

The important point to remember here is that by using dynamic loading it is possible to create an object without placing an instance of the object on the block diagram of the VI. This enhances performance of the VI, and it also allows a VI to create objects based solely on the name alone if a naming convention is created and followed.

25.12 The Car Wash Controller with Inheritance and Composition

We can now implement a version of the car wash controller that uses objects with composition and inheritance. All of the object-based action takes place in a single state—the *Cycle* state shown in Figure 25.1.

When the car wash controller first starts, the data highway shift register is initialized with a generic `Cycle` object. The *Done* Boolean in the private data of this object is set to True using a data member access property node before the `Cycle` object is

placed on the shift register. This is to ensure that this generic `Cycle` object is replaced by a child class when it passes through the `Cycle Factory`.

When the *Cycle* state executes, the `Cycle Factory` takes the current object from the data highway and determines if it is done or not. The factory passes the object if it is not done. If the object is done, the factory creates and initializes a new object based on the specifications of the argument string. The object is then passed to the `Run Cycle` method, which is a public, statically dispatched member of the `Cycle` class. This method is public so that it may be called by VIs outside of its class, such as in the car wash controller. This method is statically dispatched, because each member of the child classes must use it "as-is" without overriding it. The child classes can use the `Run Cycle` method because they inherited it from the parent `Cycle` class.

The `Run Cycle` method calls a private, statically dispatched `Cycle Time Check` method. The `Cycle Time Check` method cannot be called by the car wash controller directly, since its access scope is private. The `Cycle Time Check` VI unbundles the `Timer` object from the current cycle object's private data. The `Timer` object was incorporated into the generic `Cycle` class through composition and is made available to all of the child classes through inheritance. The `Cycle Time Check` sub-VI then uses the `Time Check` method of the `Timer` class to perform the time and proximity check.

But to perform this check, the value of the correct proximity switch must be selected from the front panel cluster of proximity switches. This is done using the dynamically dispatched `Get Switch` method of the `Cycle` class. Each child class overrides this method with their own version of the `Get Switch` VI to ensure that the proper switch is selected for each child class. Once the time check is finished, the dynamically dispatched `Light Lamps` method is used to light the appropriate front panel indicators. Each child class overrides this method as well to ensure that the proper indicators are lit for each child class.

When the time check is complete, the object representing the current cycle is placed back on the data highway. This is possible only because objects of child classes can travel on wires with the data type of their parent class. Every child car wash cycle can be placed on the shift register that was initialized by the parent `Cycle` class.

If the current cycle is not finished, the state and argument string is requeued at the opposite end of the queue for immediate execution of the next loop iteration. If the cycle is done, no action is taken, and the next state and argument string are dequeued normally. Notice that this is different from previous versions of the car wash controller where the timer was reset in the *Cycle* state when the current cycle was completed. This is not necessary here. LabVIEW automatically destroys objects when they are neither stored nor "live" on a wire. When a cycle is finished and a new child object is created, the object representing the old cycle is automatically destroyed. This destroys the old timer as well, so the timer does not need to be reset when a cycle ends. Each newly created car wash cycle object contains a new timer that is initialized when the car wash cycle object is initialized.

This also impacts the *Lock Selector* state shown in Figure 25.49. In previous versions of the car wash controller, the timer was reset in the *Lock Selector* state, as this was the first state executed after a wash type was selected. But this is not necessary here

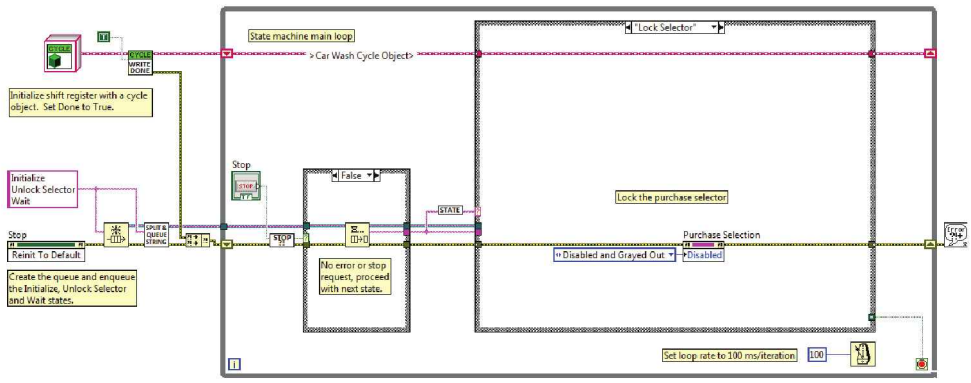


Figure 25.49: The *Lock Selector* state of a car wash controller using a `Cycle` object

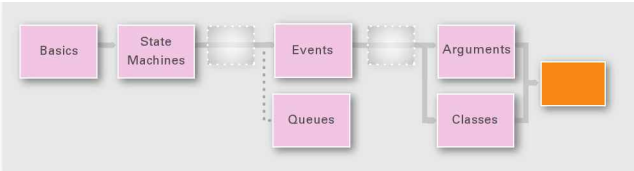
because a new timer is created and initialized with every new car wash cycle object in the *Cycle* state. Every other state of the `Cycle` class car wash controller is nearly identical to those of the `Timer` class car wash controller developed in Chapter 23. The only difference is that a `Cycle` object is on the data highway instead of a `Timer` object.

25.13 Why Bother with Classes?

At this point, you may be wondering why anyone would bother with the extra effort to create and use LabVIEW classes. We've already discussed the benefits of data encapsulation, inheritance, and composition. We've also shown how classes are self-contained and promote code reuse.

Another benefit is that classes can be used to encapsulate some of the problem-specific data and logic and move them outside of the main VI. Use the car wash controller as an example. In the previous incarnations of the car wash controller, the main VI contained all of the information about what cycles could be run, which switches each cycle needed to monitor, and which indicators each cycle used as outputs. If this information changed or new cycles were added, changes needed to be made to the main VI or its sub-VIs.

This limits the scalability of the code. For example, it makes it difficult to move to a plug-in architecture, since the main VI and its sub-VIs need to be changed when the car wash cycles are changed. But this is not the case for the object-oriented car wash controller developed in this chapter. Now the logic for a given cycle is contained in the cycle itself. The car wash controller doesn't know what switches are needed or which lamps are lit by each cycle. If the dynamic loading factory sub-VI is used, the main VI doesn't even need to know what or if car wash cycles exist. All of the cycle-specific logic has been removed from the main VI and is now encapsulated in the car wash cycle classes themselves. The main VI becomes a framework for manipulating a generic `Cycle` object, laying the groundwork for a plug-and-play architecture where car wash cycles can be added or deleted without impacting the main VI or its sub-VIs.



26

Problem Set 10

26.1 Car Wash 2 Exercises

In this series of exercises, you will incorporate inheritance and composition into the Car Wash 2 controller in a manner similar to the one used in Chapter 25.

Exercise 10.1: Create the `Option` Class and Private Data

Create the `Option` class where the private data is shown in Figure 26.1. The `Option` class will be the parent class of the individual Car Wash 2 option classes. Just as in the `Cycle` class developed in Chapter 25, the `Option` class private data contains a `Timer` object, a TIME TARGET numerical control, and a DONE Boolean. Examine the method sub-VIs shown in Figure 26.1. These sub-VIs will be developed in the following exercises. Based on the structure of the class library, what similarities exist between the `Option` class and the `Cycle` class? ■

Exercise 10.2: Create the `Option` Class Data Accessor Methods

Create the data accessor methods to read from and write to the DONE Boolean. The icon of the `Read Done` method is shown in Figure 26.2, and the icon of the `Write Done` method is shown in Figure 26.3. ■

Exercise 10.3: Create the `Option` Class Dynamic Dispatch Methods

Create the dynamic dispatch methods of the `Option` class. The icon of the `Get Position` method is shown in Figure 26.4. This method takes the current value of the car position slider as an input and it outputs the corresponding value of the car position enumerated constant. The icon of the `Set Display` method is shown in Figure 26.5. This method accepts the elapsed time and a status cluster as inputs and outputs a correctly formatted display cluster. ■

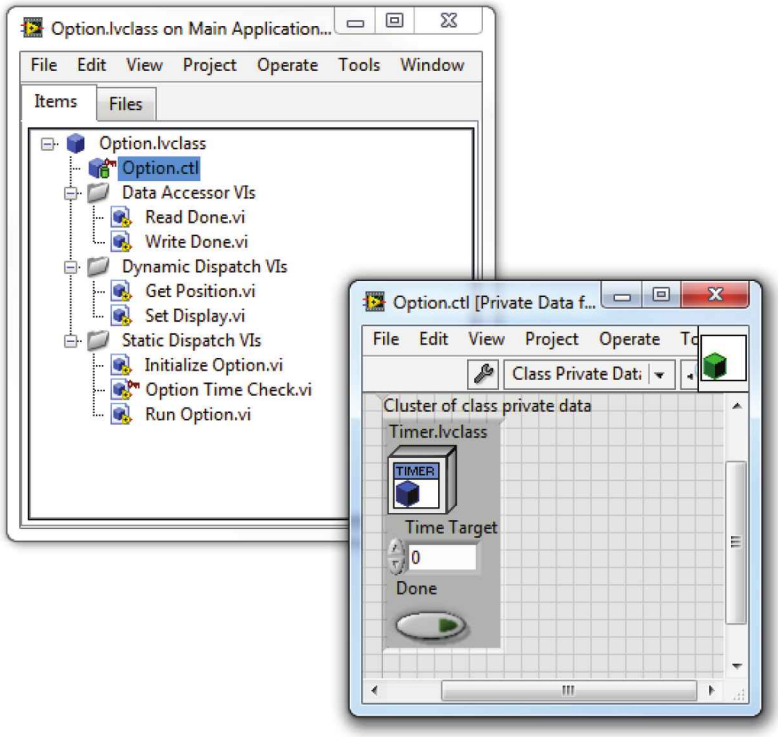


Figure 26.1: The library and private data of the `Option` class



Figure 26.2: The icon of the `Read Done` method of the `Option` class



Figure 26.3: The icon of the `Write Done` method of the `Option` class



Figure 26.4: The icon of the `Get Position` method of the `Option` class

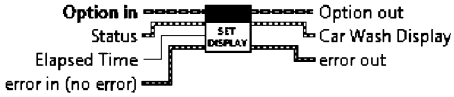


Figure 26.5: The icon of the `Set Display` method of the `Option` class

Exercise 10.4: Create the `Option` Class Static Dispatch Methods

Develop the three static dispatch methods for the `Option` class. The icon of the `Initialize Option` method is shown in Figure 26.6. This method initializes the

private data of the `Option` object, including the `Timer` object. The icon of the `Option Time Check` method is shown in Figure 26.7. This method sub-VI accepts the proximity switch as an input and performs the car wash time check. The icon of the `Run Option` method is shown in Figure 26.8. This method sub-VI accepts the car position slider as an input, extracts the car position, runs the time check, and then produces a properly formatted display cluster.

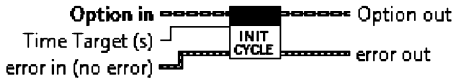


Figure 26.6: The icon of the `Initialize Option` method of the `Option` class

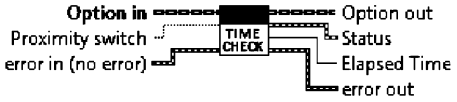


Figure 26.7: The icon of the `Option Time Check` method of the `Option` class

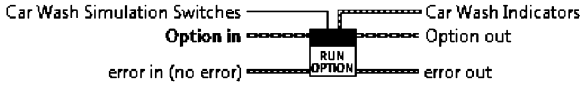


Figure 26.8: The icon of the `Run Option` method of the `Option` class ■

Exercise 10.5: Create the `Option` Child Classes

Create the child classes for the `Option` class. The child classes have empty private data and add no new methods. Each child class should override the dynamically dispatched `Get Position` and `Set Display` sub-VIs defined in Exercise 10.3. ■

Exercise 10.6: Create the `Get Option Object` Sub-VI

Create the `Get Object Object` sub-VI with the icon shown in Figure 26.9. This sub-VI is analogous to the `Get Cycle Object` sub-VI developed in Figure 25.39. This sub-VI accepts an option name string and returns an object of the corresponding class.



Figure 26.9: The icon of the `Get Option Object` sub-VI ■

Exercise 10.7: Create the `Option` Factory

Use the `Get Option Object` sub-VI developed in Exercise 10.6 to develop the `Option Factory` sub-VI with the icon shown in Figure 26.10. This sub-VI is analogous to the `Cycle Factory` developed in Figure 25.42. This sub-VI accepts a wash option string of the form `State:Option name:Time target` and an `Option` object. This sub-VI checks the DONE Boolean in the private data of the input object. If the current object is done, the sub-VI parses the input string and produces the named object. The object is initialized with the time target contained in the input string.

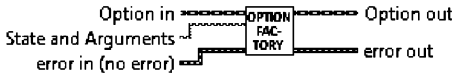


Figure 26.10: The icon of the `Option Factory` sub-VI ■

Exercise 10.8: Create the Car Wash 2 Controller Using the `Option` Class

The `Option` class, all of its children, and a factory have been developed to generate `Option` objects on demand. Use these classes and the factory to develop a version of the Car Wash 2 controller that takes advantage of inheritance and composition in the same manner as the car wash controller developed in Chapter 25. ■

26.2 Traffic Light Controller Exercises

In the following exercises, you will use inheritance and composition to create a new version of the traffic controller. This version will eliminate the individual traffic light states (*N-S Green*, *E-W Yellow*, etc.) and replace them with a single generic *Signal* state.

Exercise 10.9: Creating a `Signal` Class

Create a generic `Signal` class for the traffic light controller similar to the generic `Cycle` class for the car wash controller. The private data of the `Signal` class should contain a `Timer` object and a numeric control for the time target of the signal. The class should have two public static dispatch methods: one to initialize the signal and another to perform a time check. The initialize method will check to see if the signal will a green/red signal or a yellow signal and will initialize the time target accordingly. The class also will have a dynamic dispatch method that returns a traffic light cluster with the appropriate lamps lit. ■

Exercise 10.10: Creating Child Classes of the `Signal` Class

Create eight child classes of the `Signal` class with each representing a specific signal pattern (`N-S Green`, `E-W Left Yellow`, etc.). Each class will override the dynamic dispatch method of the `Signal` class that generates the proper signal pattern. ■

Exercise 10.11: The `Get Signal Name` Sub-VI

Create the `Get Signal Name` sub-VI with the icon shown in Figure 26.11. This sub-VI accepts a `Signal` object as an input and returns a string containing the name of the object. For instance, if a `N-S Green` object is passed to this sub-VI, the output will be the string `N-S Green`. *Hint:* use the **`Get LV Class Path`** function to get the path to the class (see the help file in Figure 26.12). The **`To More Generic Class`** function can be used to eliminate a coercion dot at the input of the **`Get LV Class Path`** function (see the help file in Figure 9.2). The name of the class (and thus the

object) then can be extracted from the path using the **`Get File Extension`** function (see the help file shown in Figure 1.31).



Figure 26.11: The icon of the `Get Signal Name` sub-VI

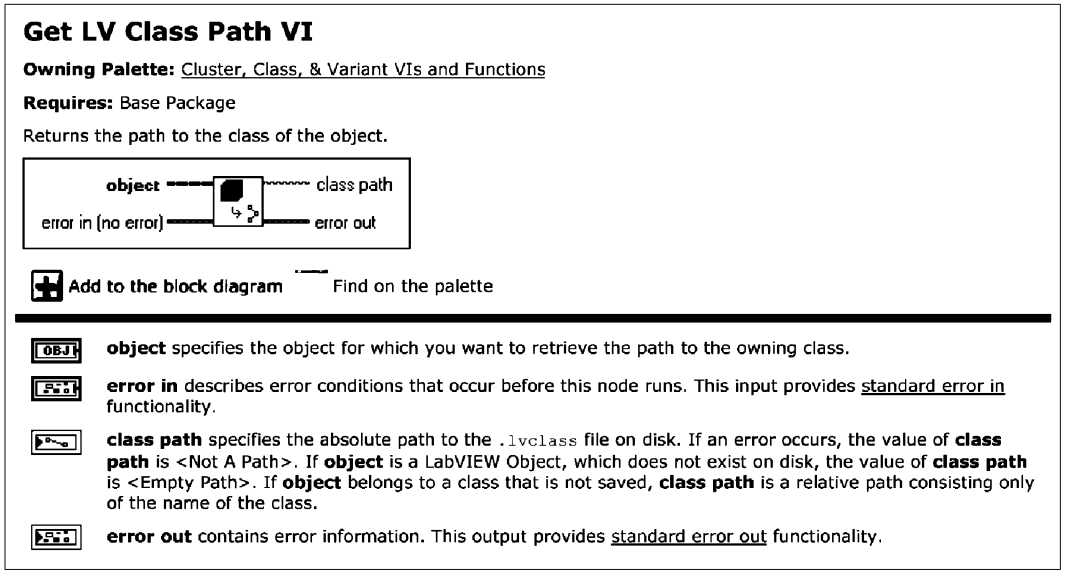


Figure 26.12: Help file for the `Get LV Class Path` function. ■

Exercise 10.12: Factory Pattern for the Traffic Light Controller

Create the `Next Signal Object` sub-VI with the icon shown in Figure 26.13. This sub-VI uses the factory pattern to generate instances of the children of the `Signal` class. This sub-VI will accept a completed `Signal` object and the cluster of vehicle sensors as inputs. This Sub-VI will use the name of the object and the state of the vehicle sensors to generate the next signal object needed by the traffic controller. The next signal object will be initialized by the factory before it is output.

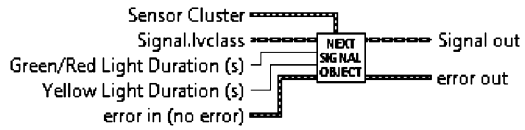


Figure 26.13: The icon of the `Next Signal Object` sub-VI ■

Exercise 10.13: Traffic Controller Using the `Signal` Class

Create a version of the traffic controller that uses the `Signal` class. The block diagram of the controller will only contain objects and methods of the generic `Signal` class. ■

26.3 Boiler Controller Exercises

In the following exercises, you will use inheritance and composition to create a new version of the boiler controller. This version will eliminate the individual boiler states (*Application Run*, *Prove The Pilot*, etc.) and replace them with a single generic *Step* state. To do this, you will create an abstract `Step` class with child classes representing the actual boiler process steps.

In many ways, the boiler controller that you develop is similar to both the traffic light controller with a abstract `Signal` class and the car wash controller with the abstract `Cycle` class. But there are a few wrinkles that make the boiler controller unique from these other two examples. In the traffic light controller and the car wash controller, each of the child states were very similar to their siblings. In the traffic light controller, each child state was nearly identical, with the only difference being what lights were lit and whether or not they needed to check the vehicle sensors to determine the next state. The car wash cycles were also very similar. They all behaved identically; they only varied in their selection of proximity switch and output lamps. The boiler steps, though, are all very different from each other. They have different conditions to determine if they are done. They access controls from multiple front panel clusters. One of the states needs to make a front panel object blink.

And yet, when you get right down to it, the boiler steps have much in common. If you review any of the previous implementations of the boiler controller, you will see that each state follows the same pattern. Each state first determines if it is completed by employing a sub-VI tailored for the requirements of that state. After determining if the state is finished, a second sub-VI tailored for that state generates the log string and writes values to the status cluster. The log file is then written to if the log file has changed. If the current state is not completed, the state and argument string is placed at the front of the queue so the state can be repeated. If the state is completed, the queue is left untouched so the next state can be dequeued during the next loop iteration.

An implementation of this plan can be seen in Figure 26.14. This version of the boiler controller uses a string-based queue with arguments to control the boiler. The same `State:Argument` parsing convention used by the car wash controller is used here. This version of the controller has an abstract `Step` object on the data highway. When the *Initialize* state executes, the queue is loaded up, and the *Done* Boolean of the `Step` object is set to True to force its replacement by a child class in the *Step* state.

The *Step* state is shown in Figure 26.15. A factory sub-VI checks to see if the current object is completed. If it is not finished, the current object is passed through. If the current object is completed, the argument string is consulted, and a new child object is generated. The factory initializes the new child object and passes it the necessary

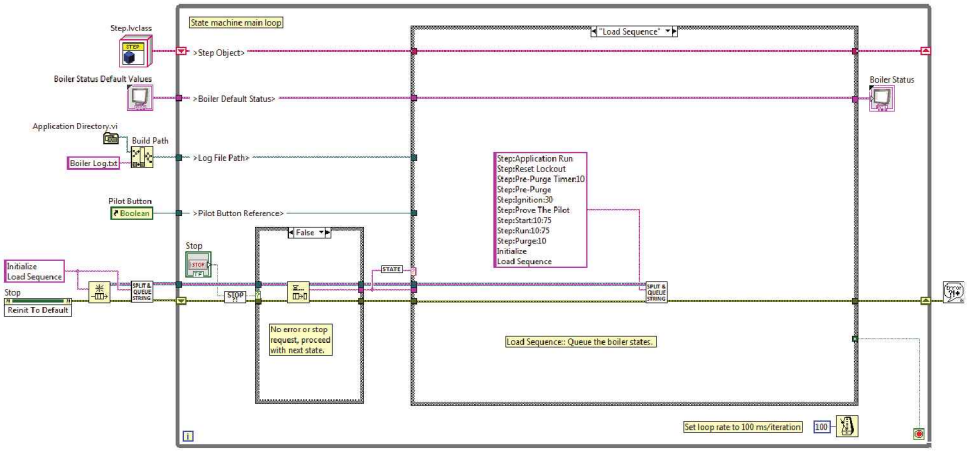


Figure 26.14: The *Load Sequence* state of the boiler controller with `Step` class

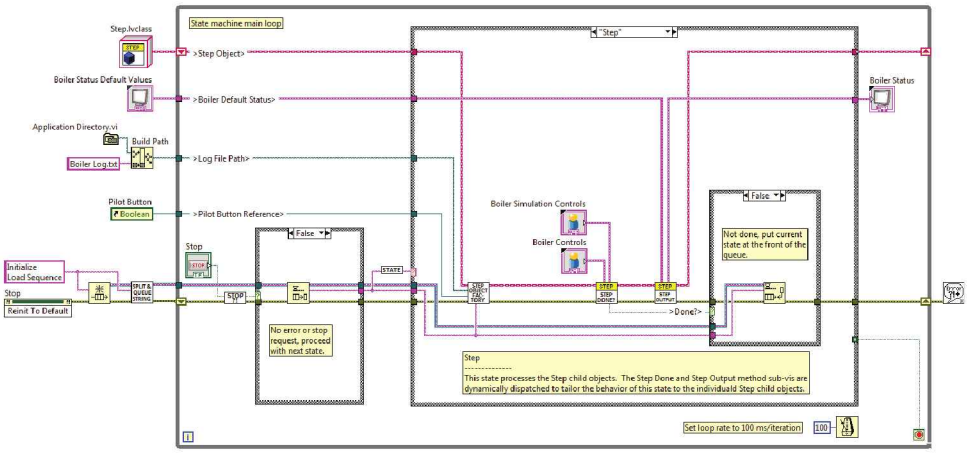


Figure 26.15: The *Step* state of the boiler controller with `Step` class

information from the constants in the input area of the loop: the path to the log file, any arguments passed from the queued string, and the reference to the PILOT button. Each child object is passed only the information that it needs. For instance, the only child object that needs the reference to the PILOT button is the `Pre-Purge` object, so it is the only object that will be initialized with that reference. The private data of each child class will be unique, and the `Step Initialize` method used by the factory will be dynamically dispatched so that each child class can initialize its unique private data.

The child object is then passed to the dynamically dispatched `Done` method. Since this method is dynamically dispatched, each child object will override it and provide a unique method for determining if the current step is completed. The `Done`

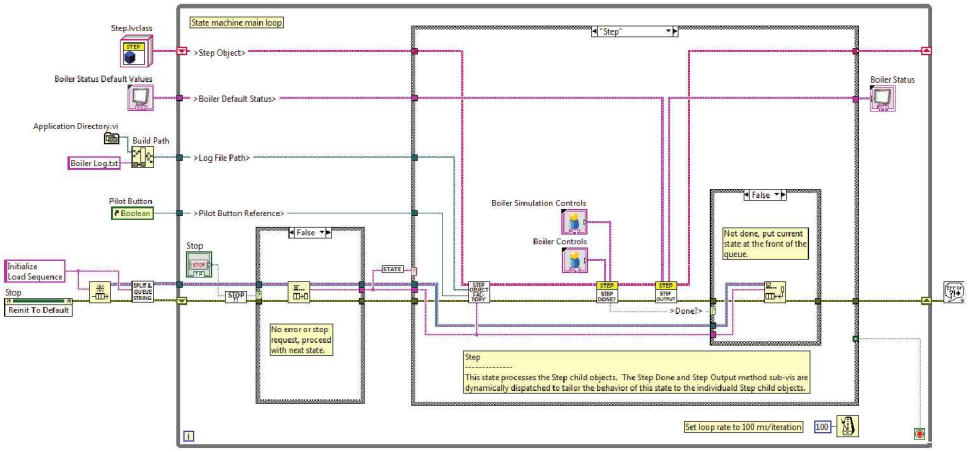


Figure 26.16: The *Shutdown* state of the boiler controller with `Step` class

method has access to the constants in the private data of the child object, and it also accepts the BOILER CONTROLS cluster and the BOILER SIMULATION CONTROLS cluster as inputs. In this way, the `Done` method has access to all of the information it needs to decide if the current step is done or not.

The child object is then passed to the dynamically dispatched `Step Output` method. This method writes to both the BOILER STATUS indicator and to the log file (if needed). Instead of storing the log string in an action engine or on the data highway, the log string is stored in the private data of the child object. The child object is then placed back on the data highway. If the current step is not completed, the state and argument string are placed back on the queue, and the current object is processed again by the *Step* state during the next loop iteration. If the current child object is finished, the queue remains untouched, and the next state will be dequeued during the next loop iteration. The factory sub-VI detects when the current object is finished and replaces it with an initialized child object corresponding to the next step in the boiler process. When user terminates the boiler process or if an error is detected, the *Shutdown* state is executed, as seen in Figure 26.16. The reference to the PILOT button is closed, and the boiler controller shuts down.

In the following set of exercises, you will be asked to develop the abstract `Step` class and the child classes. You will also develop the factory sub-VI. These classes and the factory will enable you to build and run the boiler controller shown in the previous figures.

Exercise 10.14: Create the `Step` Class Library and Private Data

Create the `Step` class and private data as shown in Figure 26.17. Create a custom icon for the class. Create static dispatch methods for reading from and writing to the DONE Boolean control in the private data of the class. The icon for the `Read Done`

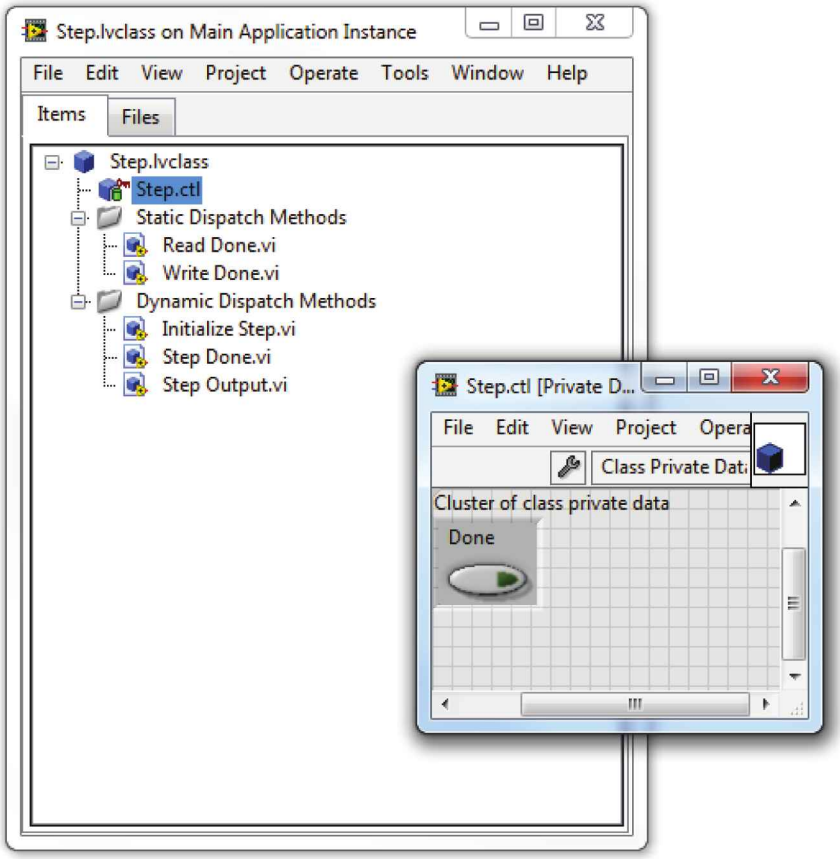


Figure 26.17: The library and private data of the `Step` class



Figure 26.18: The icon of the `Read Done` method of the `Step` class



Figure 26.19: The icon of the `Write Done` method of the `Step` class

method is shown in Figure 26.18, and the icon for the `Write Done` method is shown in Figure 26.19. Use these figures for guidance on the inputs and outputs of the methods. Do not create the dynamic dispatch methods yet, they will be developed in the next exercise. ■

Exercise 10.15: Create the `Step` Class Dynamic Dispatch Methods

Create the dynamic dispatch methods of the `Step` class. These methods should be stubs only, they should have the correct icons and connector panes, but no code should appear on the block diagram. Since these methods are only stubs, configure the class to require that descendants override these methods. The icons of these three methods are found in Figures 26.20, 26.21, and 26.22. Use these icons for guidance on the inputs and outputs of these methods.

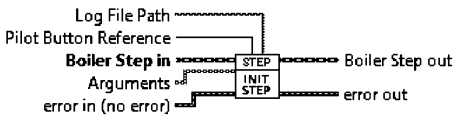


Figure 26.20: The icon of the `Initialize Step` method of the `Step` class

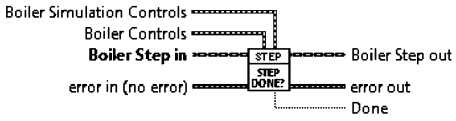


Figure 26.21: The icon of the `Step Done` method of the `Step` class



Figure 26.22: The icon of the `Step Output` method of the `Step` class ■

Wiring Tip: The control refnum for the PILOT button can be found in the *Refnum* palette of controls. The control refnum is for a generic control. In order to avoid coercion dots, cast the refnum specifically for Boolean controls. Right-click the control and select *Select VI Server Class*. Then drill down the sub-menus to *Generic...GObject...Control...Boolean*. The icon on the refnum will change to that of a vertical switch to signify that the control refnum is now specifically for Boolean controls.

Exercise 10.16: Create the `Conditional Write To Log File` Sub-VI

Revise the `Write to Boiler Log File` sub-VI developed in Exercise 2.4 to accept a log string and a Boolean control as inputs. If the Boolean control is True, the sub-VI



Figure 26.23: The icon of the `Conditional Write To Log File` sub-VI

will format the log string and write it to the boiler log file. If the Boolean is False, the sub-VI will exit without writing to the log file. The icon of the new `Conditional Write To Log File` sub-VI is shown in Figure 26.23. ■

Exercise 10.17: Create the `Application Run` Class

Create the `Application Run` class. The class library and private data are shown in Figure 26.24. The private data for the class consists of a string control for the log string and a path control for the log file path. The string control is used to store the old value of the log string in order to decide if it is necessary to write the new value of the log string to the log file.

The class library contains the three override methods corresponding to the three dynamically dispatched methods you developed in Exercise 10.15. Develop these three override methods. You might find Figures G.14 and G.15 useful. Use the `Conditional Write To Log File` sub-VI developed in Exercise 10.16 when developing the `Step Output` method.

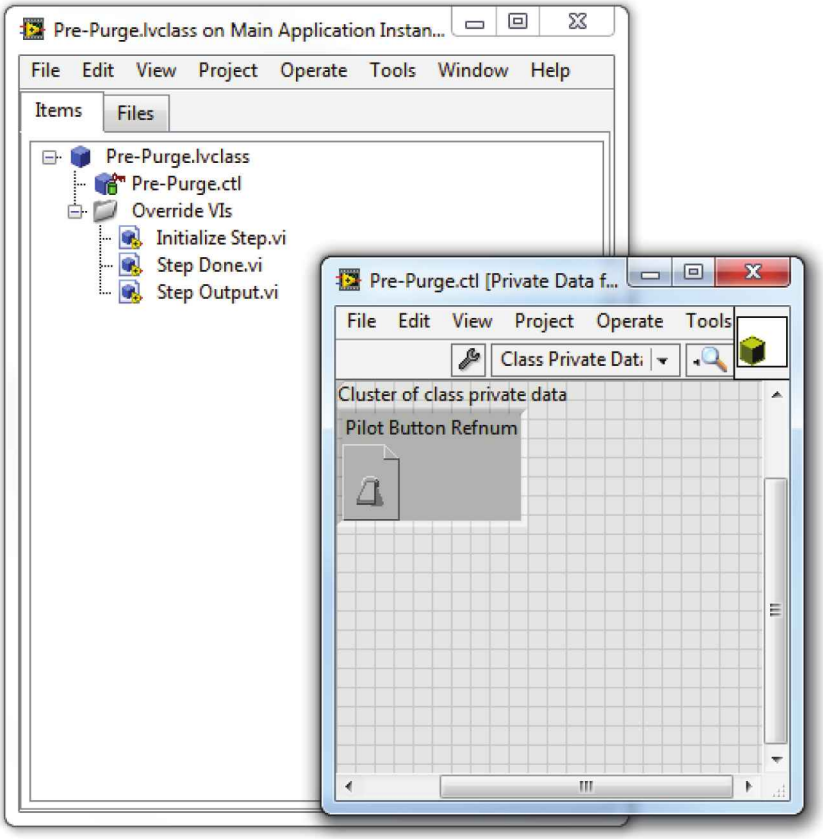


Figure 26.24: The library and private data of the `Application Run` class ■

Exercise 10.18: Create the `Reset Lockout` Class

Create the `Reset Lockout` class. The class library and private data are shown in Figure 26.25. The private data for the class consists of a string control for the log string and a path control for the log file path. The string control is used to store the old value of the log string in order to decide if it is necessary to write the new value of the log string to the log file.

The class library contains the three override methods corresponding to the three dynamically dispatched methods you developed in Exercise 10.15. Develop these three override methods. You might find Figures G.16 and G.17 useful. Use the `Conditional Write To Log File` sub-VI developed in Exercise 10.16 when developing the `Step Output` method.

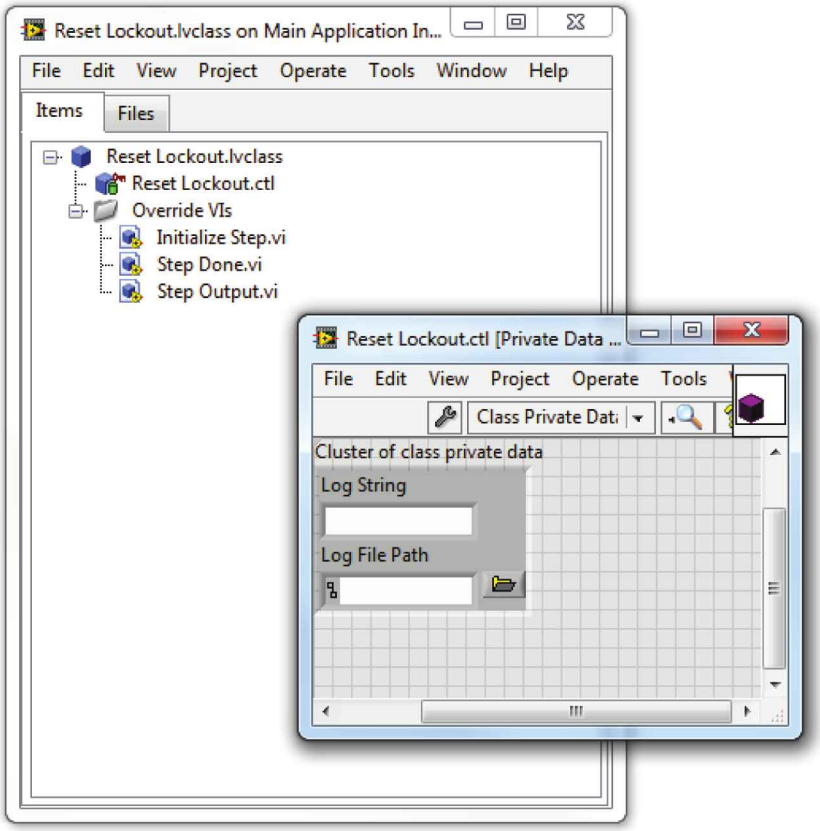


Figure 26.25: The library and private data of the `Reset Lockout` class ■

Exercise 10.19: Create the `Pre-Purge Timer` Class

Create the `Pre-Purge Timer` class. The class library and private data are shown in Figure 26.26. This step in the boiler process is timed, so the private data of the class

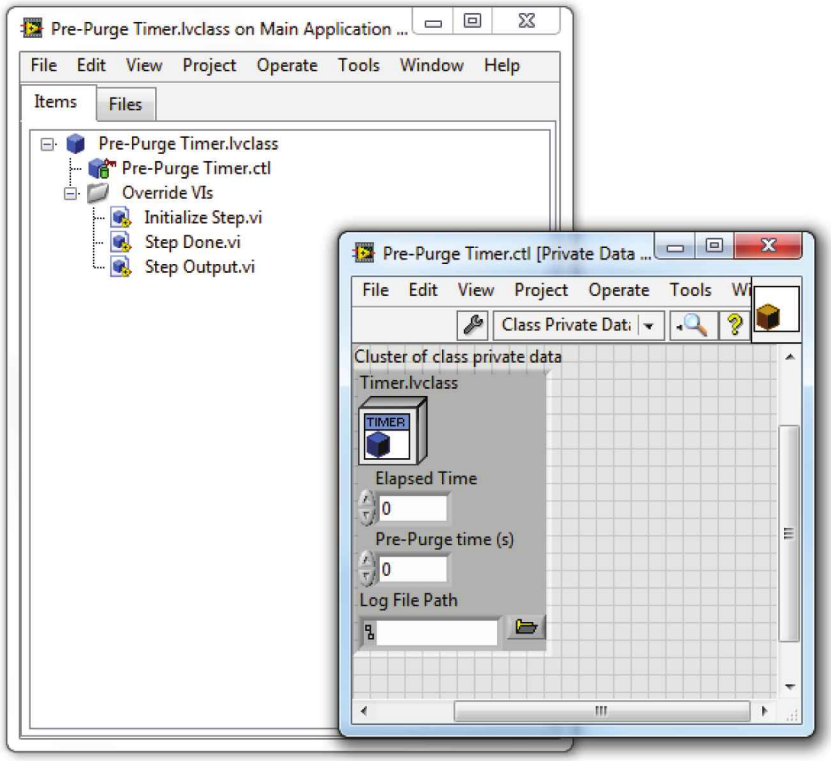


Figure 26.26: The library and private data of the `Pre-Purge Timer` class

includes an object from the `Timer` class developed in Chapter 23 as well as a numerical control for the pre-purge time limit. The private data also contains a numeric control to store the current elapsed time. A path control is also included for the log file path. Since the log string incorporates the elapsed time, the string is always unique and does not need to be stored in the private data for comparison with later log strings.

The class library contains the three override methods corresponding to the three dynamically dispatched methods you developed in Exercise 10.15. Develop these three override methods. You might find Figures G.19 and G.20 useful. Use the `Conditional Write To Log File` sub-VI developed in Exercise 10.16 when developing the `Step Output` method. ■

Exercise 10.20: Create the `Pre-Purge` Class

Create the `Pre-Purge` class. The class library and private data are shown in Figure 26.27. The private data for the class consists of a control refnum for the PILOT button. This control refnum is used to make the PILOT button blink using the *Blinking* property node. Cast the control refnum for Boolean controls to avoid coercion dots. This step of the boiler process does not write information to the log file, so the

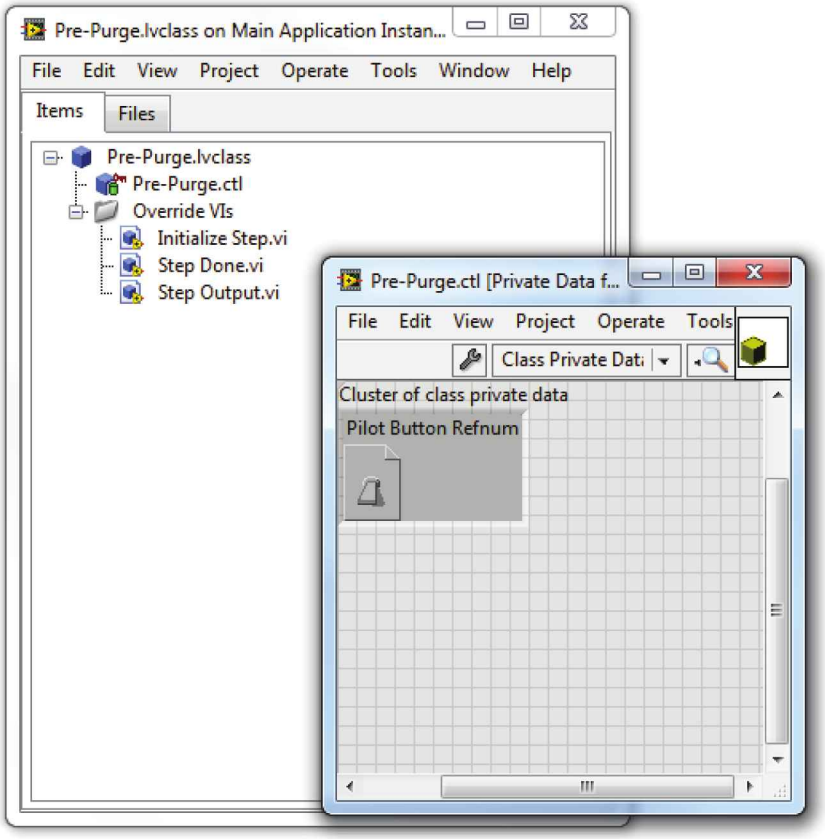


Figure 26.27: The library and private data of the `Pre-Purge` class

log file path control and log string controls used by the other child classes are not needed here.

The class library contains the three override methods corresponding to the three dynamically dispatched methods you developed in Exercise 10.15. Develop these three override methods. You might find Figures G.21 and G.22 useful. Consider carefully which of these three methods should be responsible for making the PILOT button blink, and justify your choice. ■

Exercise 10.21: Create the `Ignition` Class

Create the `Ignition` class. The class library and private data are shown in Figure 26.28. The private data for the class consists of a string control for the log string and a path control for the log file path. The string control is used to store the old value of the log string in order to decide if it is necessary to write the new value of the log string to the log file.

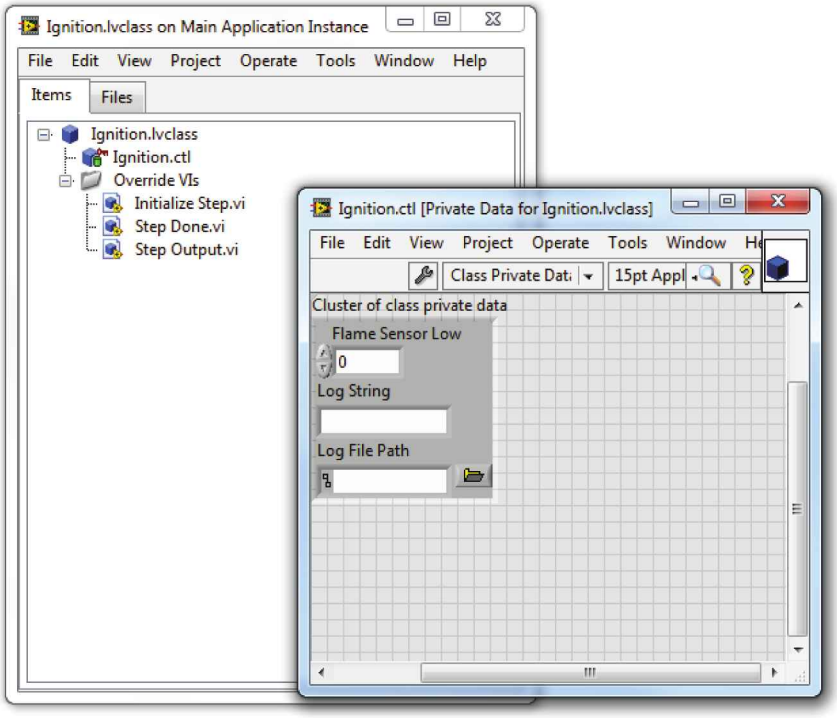


Figure 26.28: The library and private data of the `Ignition` class

The class library contains the three override methods corresponding to the three dynamically dispatched methods you developed in Exercise 10.15. Develop these three override methods. You might find Figures G.23 and G.24 useful. Use the `Conditional Write To Log File` sub-VI developed in Exercise 10.16 when developing the `Step Output` method. ■

Exercise 10.22: Create the `Prove The Pilot` Class

Create the `Prove The Pilot` class. The class library and private data are shown in Figure 26.29. The private data for the class consists of a string control for the log string and a path control for the log file path. The string control is used to store the old value of the log string in order to decide if it is necessary to write the new value of the log string to the log file.

The class library contains the three override methods corresponding to the three dynamically dispatched methods you developed in Exercise 10.15. Develop these three override methods. You might find Figures G.25 and G.26 useful. Use the `Conditional Write To Log File` sub-VI developed in Exercise 10.16 when developing the `Step Output` method.

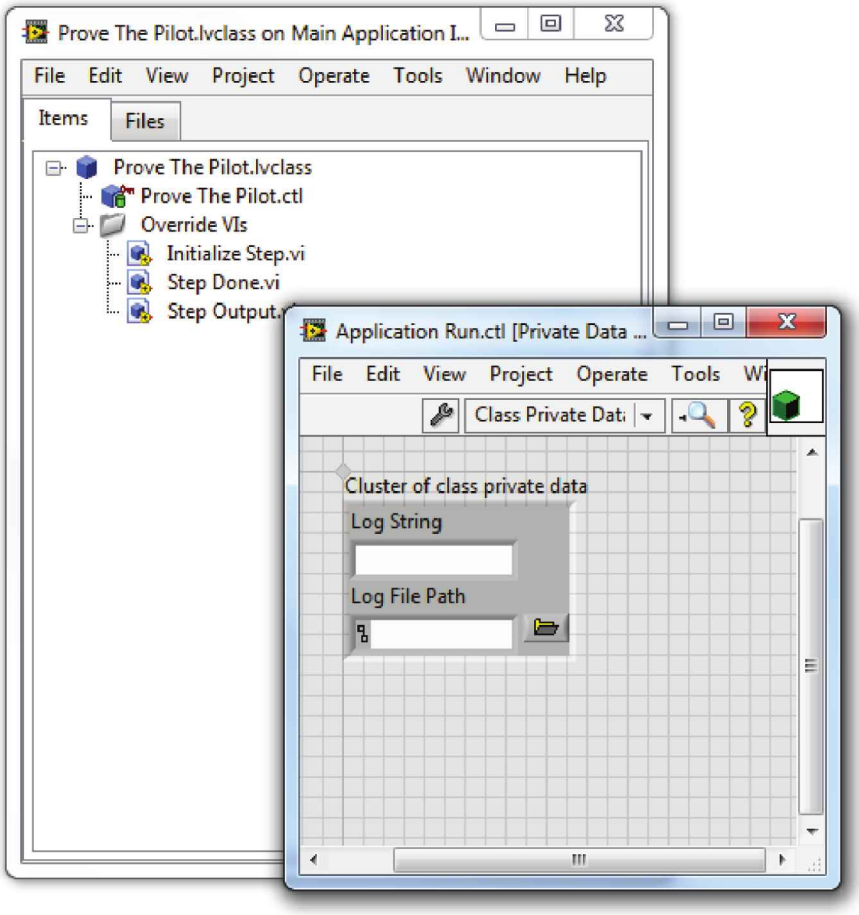


Figure 26.29: The library and private data of the `Prove The Pilot` class ■

Exercise 10.23: Create the `Start` Class

Create the `Start` class. The class library and private data are shown in Figure 26.30. The private data for the class consists of a string control for the log string and a path control for the log file path. The string control is used to store the old value of the log string in order to decide if it is necessary to write the new value of the log string to the log file.

The class library contains the three override methods corresponding to the three dynamically dispatched methods you developed in Exercise 10.15. Develop these three override methods. You might find Figures G.27 and G.28 useful. Use the `Conditional Write To Log File` sub-VI developed in Exercise 10.16 when developing the `Step Output` method.

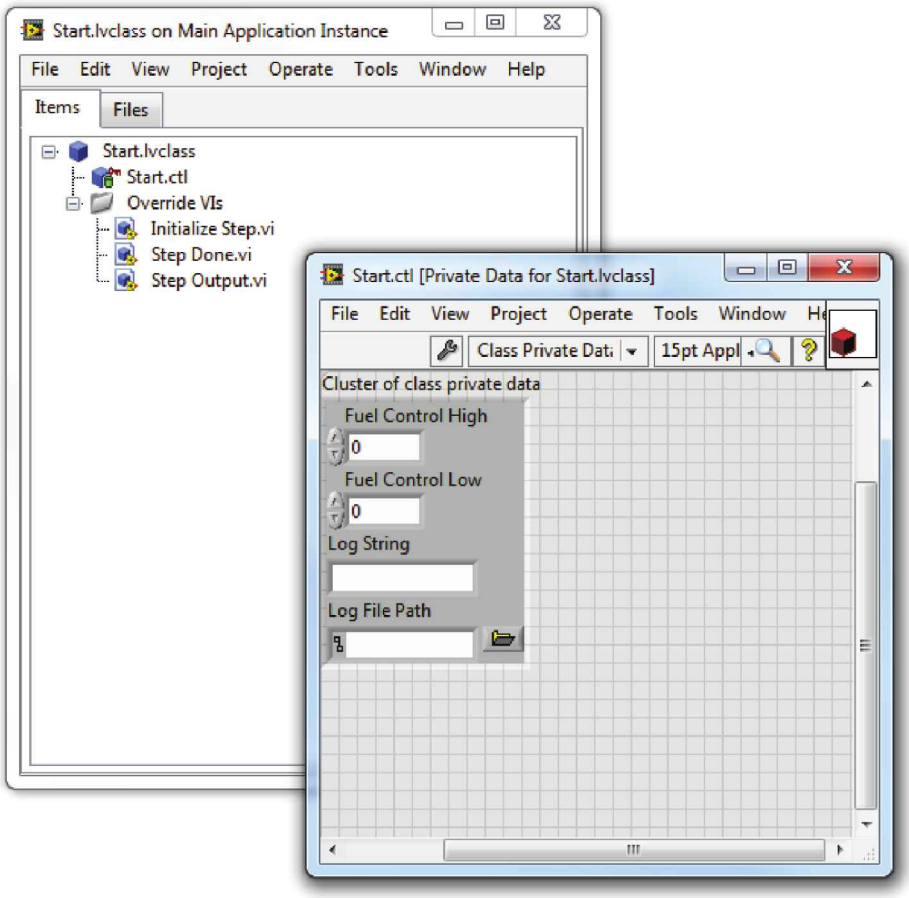


Figure 26.30: The library and private data of the `Start` class ■

Exercise 10.24: Create the `Run` Class

Create the `Run` class. The class library and private data are shown in Figure 26.31. The private data for the class contains the fuel control limits and the current value of the FUEL CONTROL VALVE. It also contains of a string control for the log string and a path control for the log file path. The string control is used to store the old value of the log string in order to decide if it is necessary to write the new value of the log string to the log file.

The class library contains the three override methods corresponding to the three dynamically dispatched methods you developed in Exercise 10.15. Develop these three override methods. You might find Figures G.29 and G.30 useful. Use the `Conditional Write To Log File` sub-VI developed in Exercise 10.16 when developing the `Step Output` method.

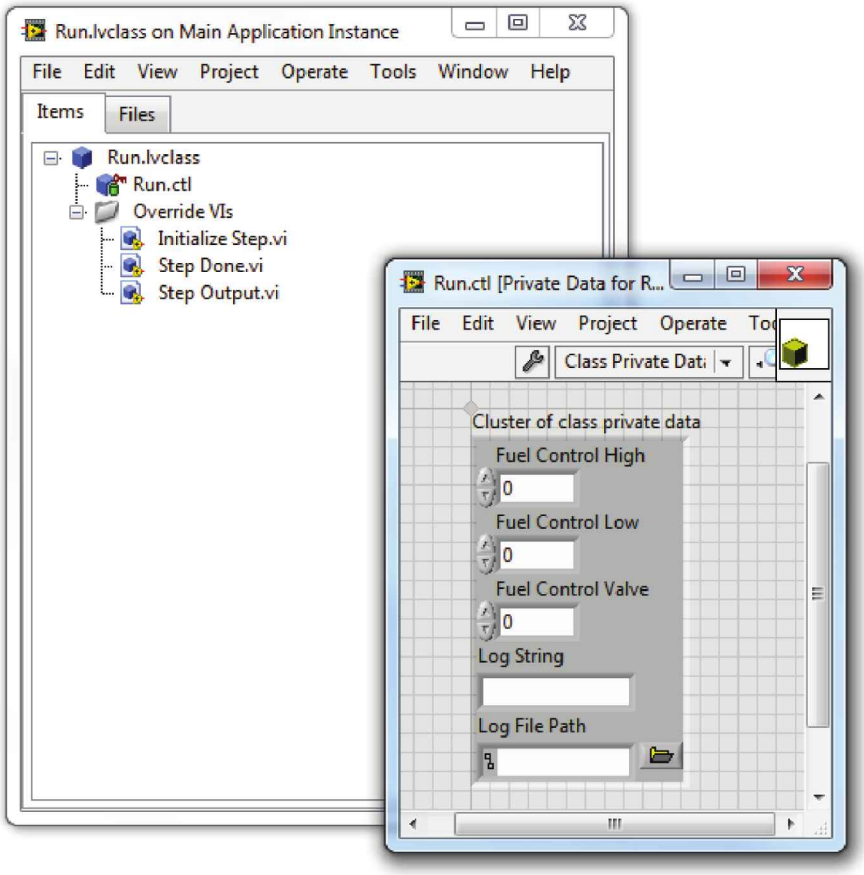


Figure 26.31: The library and private data of the `Run` class ■

Exercise 10.25: Create the `Purge` Class

Create the `Purge` class. The class library and private data are shown in Figure 26.32. This step in the boiler process is timed, so the private data of the class includes an object from the `Timer` class developed in Chapter 23 as well as a numerical control for the purge time limit. The private data also contains a numeric control to store the current elapsed time. A path control is also included for the log file path. Since the log string incorporates the elapsed time, the string is always unique and does not need to be stored in the private data for comparison with later log strings.

The class library contains the three override methods corresponding to the three dynamically dispatched methods you developed in Exercise 10.15. Develop these three override methods. You might find Figures G.31 and G.32 useful. Use the `Conditional Write To Log File` sub-VI developed in Exercise 10.16 when developing the `Step Output` method.

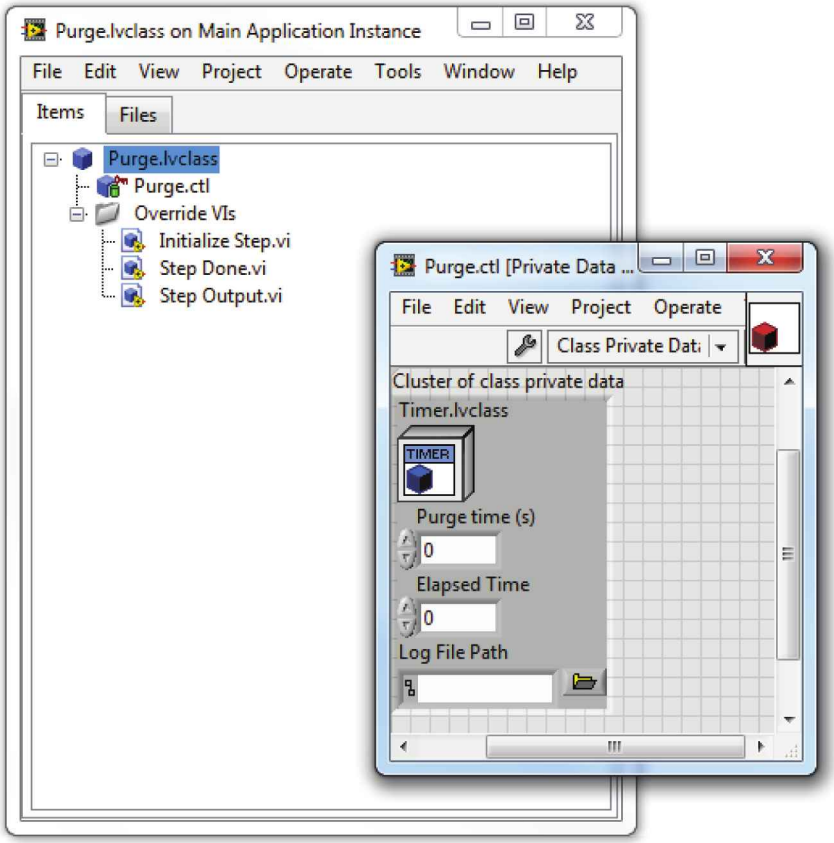


Figure 26.32: The library and private data of the `Purge` class ■

The child classes are now complete. The final step is to develop the `Step` factory sub-VI that will instantiate a `Step` or child object given the name of the object.

Exercise 10.26: Create the `Get Step Object` Sub-VI

Create the `Get Step Object` sub-VI with the icon shown in Figure 26.33. This sub-VI takes the name of a desired class as an input and produces an object of the named class as an output. If an unknown name is passed as an input, output a `Step` object and generate a custom error message. Pattern this sub-VI on the `Get Cycle Object` sub-VI shown in Figure 25.39.



Figure 26.33: The icon of the `Get Step Object` sub-VI ■

Exercise 10.27: Create the `Step Factory` Sub-VI

Create the `Step Factory` sub-VI with the icon shown in Figure 26.34. This sub-VI is a factory pattern sub-VI for `Step` objects. The factory accepts a string input in the form `State:Argument`, where the argument is the name of the desired object. The factory also accepts the current `Step` or child object as an input. The factory should first check to see if the current object is done. If it is not completed, the factory should simply pass it through. If it is finished, the factory should parse the string to extract the object name. It should produce the object using the `Get Step Object` sub-VI developed in Exercise 10.27. It should then initialize the object.

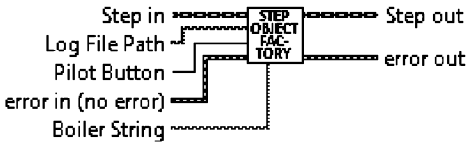
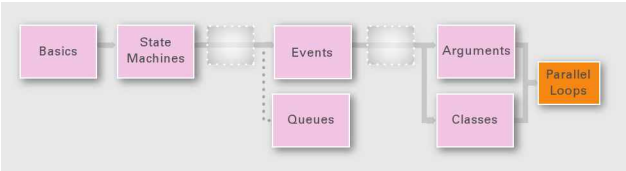


Figure 26.34: The icon of the `Step Factory` sub-VI ■

Exercise 10.28: Is It Worth It?

As you can see, creating the `Step` class and all of its children requires effort and time compared to the other implementations of the boiler controller discussed in this book. Is it worth it? Why or why not? ■



27

Producer–Consumer State Machines

All of the state machines discussed in the main body of this book are single-loop state machines. All of the Certified LabVIEW Developer exams and the exercises in this book can be done using single-loop state machines. But sometimes single-loop state machines are not sufficient to solve the problem at hand. Parallel loop architectures are a logical next step after mastering the single-loop architectures presented in this book.

Single-loop state machines execute one state at a time. This approach works well, but it has limitations. Consider the event-driven state machines that we've seen so far. The event structure was placed in the *Wait* state and was used to monitor the user interface. When the state machine was in the *Wait* state, it responded quickly to user input.

But what happens when the state machine is not in the *Wait* state? Suppose that a state machine had to execute states that took a significant amount of time after receiving user input in the *Wait* state. The user interface would not be monitored while those tasks were taking place, because the state machine would no longer be in the *Wait* state. The user interface would be sluggish and unresponsive.

In a situation like this, it would be better to monitor the user interface continually while the state machine performs other tasks. But this can't be done in a single-loop state machine. In order to execute two states at once, you need to have a parallel-loop architecture. One loop can be dedicated to monitoring the user interface, while another loop can execute the tasks requested by the user.

This design pattern is generally known as the producer–consumer architecture. The producer loop generates data and the consumer loop processes it. Both processes run in parallel. In this chapter, we examine the "producer–consumer (events)" pattern. In this design pattern, the producer loop contains an event structure to respond to user

input, and the consumer loop contains a state machine to process the user commands. The producer–consumer architecture monitors the user interface continuously as it processes prior user input.

27.1 Stopping Parallel Loops

Parallel loops are easy to create in LabVIEW, since LabVIEW is an inherently parallel language. To create parallel loops in LabVIEW, all you need to do is place parallel loops on the block diagram. "Parallel" means that one loop does not serve as an input to the other. An example of this is shown in Figure 27.1. Loop A and Loop B are totally independent of each other. The loops have no data dependency; neither loop is "upstream" of the other.

Parallel loops are easy to create, but they can be difficult to stop. In Figure 27.1, each loop has a dedicated stop button. To stop the program, both stop buttons have to be pressed. This works, but it is not the way most developers want to stop parallel loops. Usually the desire is to have both loops stop with the press of a single button.

This is not as easy to do as you might think. Take a look at Figure 27.2. The developer of this code intended for the STOP button to stop both loops, but this method does not work because the STOP button is outside of the loops. When this code runs, the value of the STOP button travels along the wires to the input tunnels of the loops. The loops won't run until data is present at their input terminals. When the code starts to run, the value of the STOP button is False, so the value False arrives at the input tunnels. This value remains unchanged while the loops run. Remember, the loops don't "look outside" the loop while the loop runs. Inputs to the loop remain constant throughout the duration of the loop. This means that the False values are

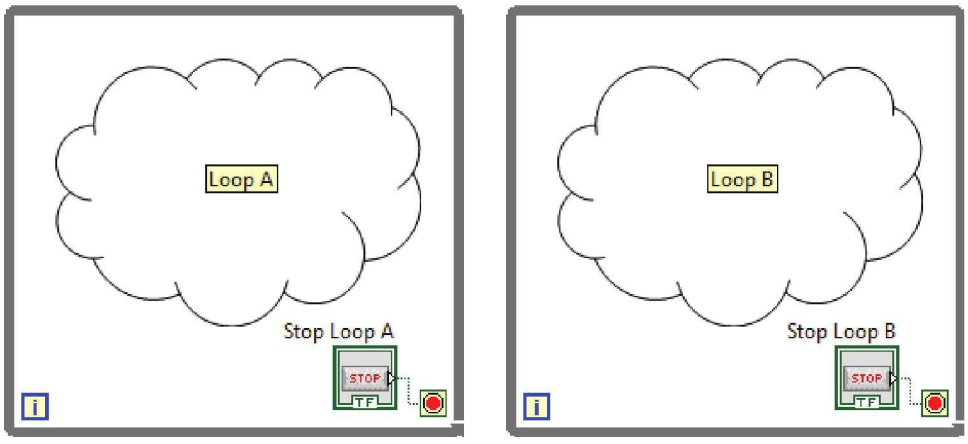


Figure 27.1: Parallel loops are easy to create in LabVIEW

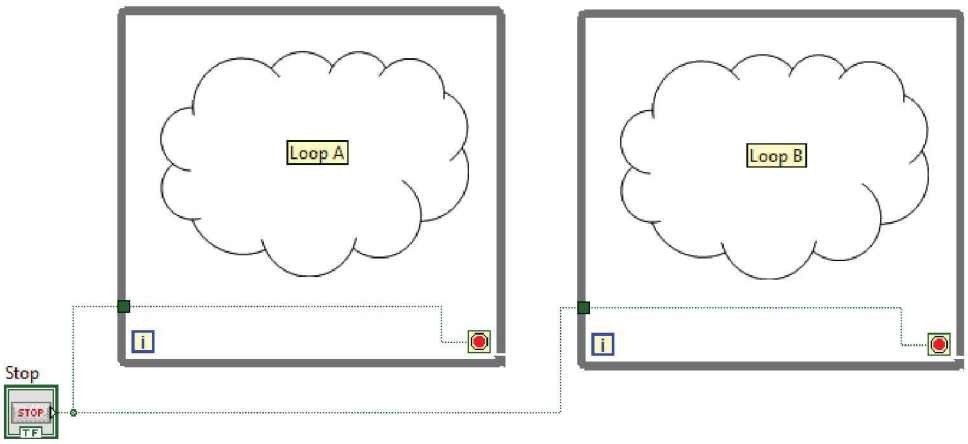


Figure 27.2: This method of stopping parallel loops won't work

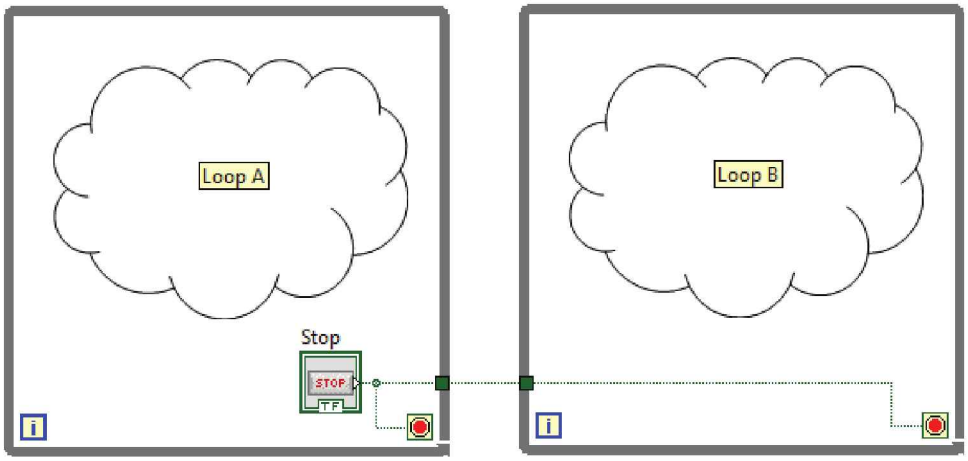


Figure 27.3: This method of stopping parallel loops won't work either

wired to the loop conditional terminals every loop iteration. The loops will never, end and the VI will have to be aborted from the toolbar.

The method shown in Figure 27.3 is an improvement, but it won't work as intended either. The STOP button is in Loop A, so its value will be read every iteration of Loop A. When the STOP button is pressed, Loop A will stop. But what about Loop B?

Loop B will not run while Loop A is running. When Loop A ends, Loop B will run for one iteration and stop. Why does this happen? The answer lies in the data flow. Notice that a wire from the STOP button exits Loop A and enters Loop B. That wire creates a data dependency between the two loops. Loop A is now an input to Loop B. Program elements in LabVIEW won't run until all of their inputs are present, so Loop B won't run until it has input data from Loop A. But data won't leave an

output tunnel of a loop until the loop is completed, so Loop B must wait until Loop A has completed before it receives its input data.

Loop B will start to run as soon as it receives its input data from Loop A. But what will that input data be? The value sent from the Loop A output tunnel will be the value of the STOP button during the last loop iteration. But since Loop A has stopped, that value will be True. The True value then travels to the Loop B conditional terminal. The loop proceeds with its first iteration and then checks the value at the conditional terminal, stopping the loop after the first iteration. Obviously, this is not what the developer intended.

In order to stop both loops at the same time, the loops need a way to communicate with each other without creating a data dependency. They need to communicate "off the page" of the LabVIEW program. A shared resource needs to be created that both loops have access too. The shared resource then can be used to send messages from one loop to the other without creating a direct data dependency between the two loops.

One way to do this is through the use of queues. Queues can be used to communicate between parallel loops, because queues are accessed by-reference, not by-value. The queue reference is a constant that can be passed to both loops without setting up a data dependency. When one loop accesses the queue, changes to the queue can be seen by the other loop. The loops can thus communicate with each other while remaining parallel.

27.2 The Producer–Consumer Architecture

The producer–consumer architecture is a common parallel loop architecture that uses queues for loop communication. The producer–consumer architecture can have many parallel loops, but in this chapter, we only study produce–consumer architectures with two loops. One loop is designated as the producer, and the other loop is designated as the consumer. The hallmark of the producer–consumer architecture is that there is a communication path between the producer loop and each of the consumer loops, typically a queue. The producer loop produces data and transmits it to the consumer loop. The data flow is primarily one-way from the producer to the consumer, although there is usually an alternate communication path to allow a consumer loop to shut down the producer loop if an error is detected in the consumer loop. Producer–consumer loops are usually designed in such a way that the communication between the loops is buffered to ensure that no data is lost.

One typical application of the producer–consumer architecture is data acquisition and analysis. The data is acquired in the producer loop and sent to the consumer loop to be analyzed, displayed, or stored while the producer loop continues to acquire data. The two loops communicate via a queue, so none of the data is lost—even if the loops run at different speeds. The producer loop can acquire the data at high speeds and dump it to the queue. The consumer can analyze the data at a slower rate with no data loss. This is known as the producer–consumer (data) architecture.

Another common use of the producer–consumer architecture is for monitoring the user interface. An event structure is placed in the producer loop to monitor the

user interface continuously. The producer loop responds to user input by placing commands on a queue. The consumer loop contains a state machine that dequeues the commands from the producer loop and processes them. In this way, the user interface remains responsive while the state machine is working. This is known as the producer–consumer (events) architecture.

27.3 The Producer–Consumer (Events) Architecture

The producer–consumer (events) achitecture is the one we examine here. To demonstrate this architecture, we convert the queued-string version of the `Multitest` VI developed in Chapter 20 to a producer–consumer (events) state machine.

Error handling is challenging in the producer–consumer design pattern. Each loop must be able to trap errors and shut itself down if an error is detected. In addition, each loop needs to be able to signal the other loop to shut down when an error is detected.

To demonstrate error handling in the producer–consumer architecture, the front panel of the `Multitest` VI will be modified, as shown in Figure 27.4. Two additional buttons have been added: PRODUCER ERROR and CONSUMER ERROR. This version of the `Multitest` VI is an exercise in creating tight error chains and merging them correctly. Pressing these buttons intentionally introduces errors into the producer and consumer loop. Pressing either of these buttons should shut down both loops of the VI. The error reported when the program shuts down should correspond to the error generated by the button press.

The general architecture of the producer–consumer `Multitest` VI is seen in Figure 27.5. The VI consists of two parallel loops. The producer loop contains an event structure followed by a guard clause case structure. The consumer loop consists of a guard clause case structure followed by the main state machine case structure. The loops communicate via a queue. The queue is created and initialized in the input area of the state machine, and the queue reference is passed as an input to both loops. The queue reference wires are not put on shift registers, because they remain constant throughout the life of the loops.

The error wire from the queue creation and initialization functions is also branched and passed to both loops. The error wires initialize shift registers that trap the errors in each loop. The error wires carry data by-value, not by-reference. This means that after the error wire splits and enters the two loops, each error wire branch

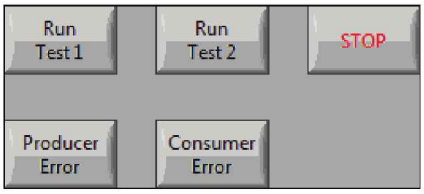


Figure 27.4: The front panel of the producer–consumer `Multitest` VI

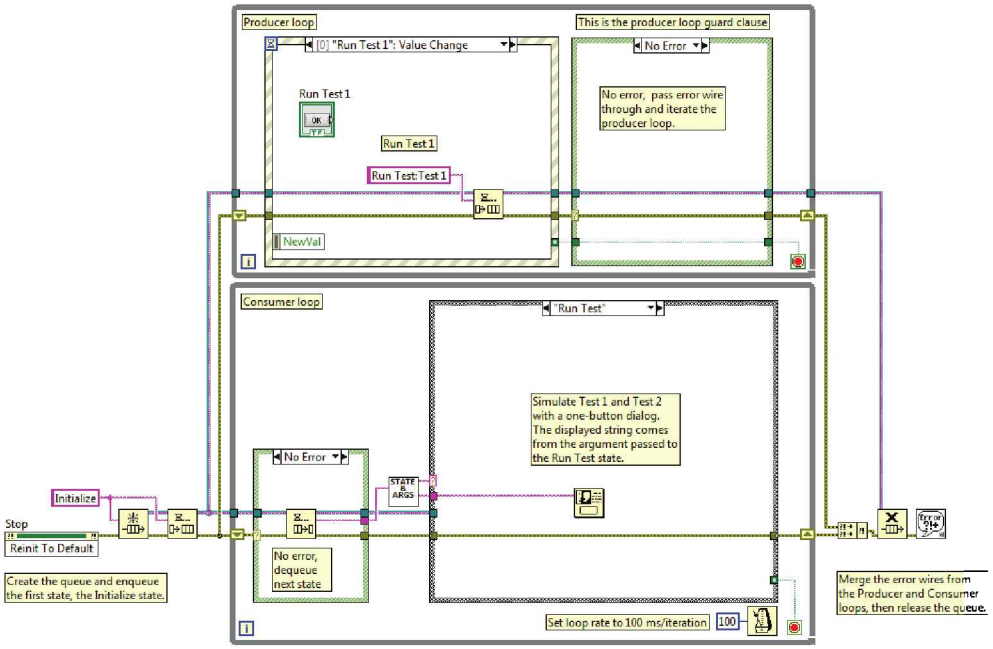


Figure 27.5: The *Run Test 1* event case and *Run Test* state of the producer–consumer `Multitest` VI

will be carrying independent error information. Errors detected by one loop will not be seen by the other loop.

Figure 27.5 shows the normal operation of the state machine. If the RUN TEST 1 button is pressed, the event structure responds to the value change by enqueueing the *Run Test* state with the `Test 1` string as an argument. If there is no error, the producer loop guard clause passes the queue and error wires through, and the producer loop iterates and waits for more input.

Notice the difference between this event case and the one shown in Figure 20.16. In the single loop `Multitest` VI, the event structure had to queue a *Wait* state after the *Run Test* state to ensure that the state machine returned to the *Wait* state after the *Run Test* state completed. This isn't necessary in the producer–consumer version, because the event structure is always active in the producer loop. After the event structure responds to an event, the producer loop iterates and automatically returns to the event structure to wait for the next input. While the producer loop is waiting, the consumer loop dequeues the *Run Test* state and acts upon it. When the queue is empty, the **`Dequeue Element`** in the consumer loop waits until a new element appears on the queue.

Now that we've seen the normal operation of the producer–consumer (events) architecture, let's examine the three ways that it can be shut down. The first way to shut it down is by pressing the STOP button. This scenario is shown in Figure 27.6. When the user presses the STOP button, a value change event is generated. The event case that

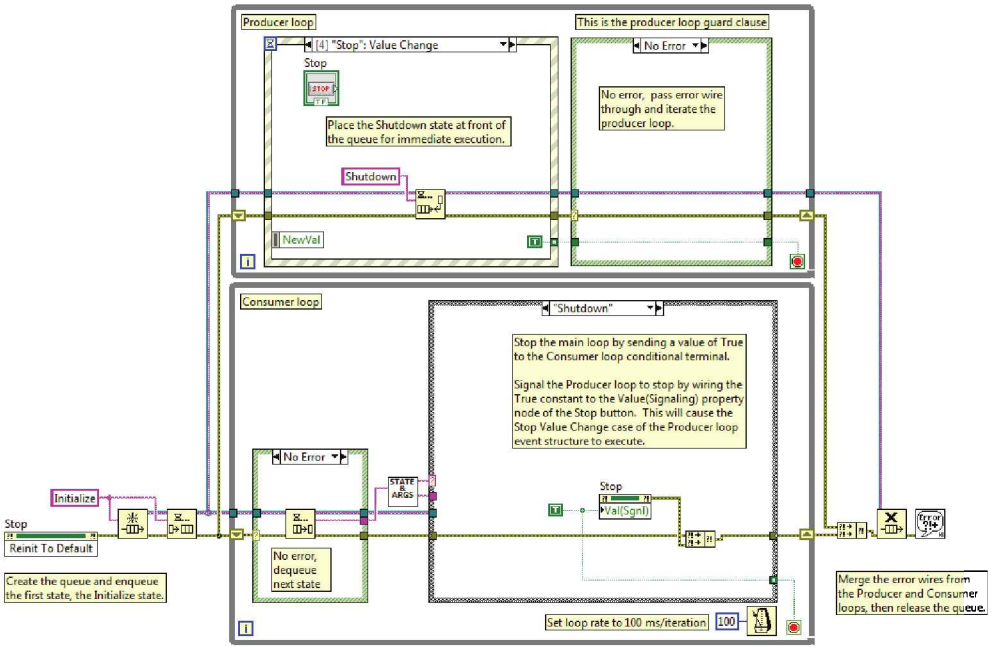


Figure 27.6: Stopping the `Multitest` VI by pressing the STOP button

handles the value change of the STOP button responds by placing the *Shutdown* state at the front of the queue using the **`Enqueue Element At Opposite End`** function. The value of the STOP button is then passed through the guard clause and is wired to the conditional terminal to stop the producer loop. An error wire then proceeds from the producer error shift register and is connected to an input terminal of the **`Merge Errors`** function.

While this is happening, the consumer loop dequeues the *Shutdown* state and executes it. The *Shutdown* state contains a Boolean True constant that is wired to the consumer loop conditional terminal, stopping the consumer loop. The Boolean True constant is also wired to a **`Value(Signalling)`** property node of the STOP button. The **`Value(Signalling)`** property node allows the developer to programatically simulate changing the value of a front panel control. This property node not only changes the value of the control, it also triggers a value change event that can be detected by an event structure. The **`Value`** property node only changes the value of a front panel control. The **`Value(Signalling)`** property node changes the value and triggers a value change event. In Figure 27.6, the **`Value(Signalling)`** property node causes the value of the STOP button to change and also generates a value change event, but nothing happens in response since the producer loop is already shut down.

After the consumer loop ends, a wire from the error shift register of the consumer loop enters the **`Merge Errors`** function. This function executes once it has received error values from both loops, sending its merged error output to the **`Release Queue`** function. The queue is then released, and the error chain terminates in a **`General`**

Error Handler function. The net effect of this sequence is that both loops shut down when the user presses the STOP button.

The second way to shut down the producer–consumer state machine is to generate an error in the producer loop. The shutdown procedure is shown in Figure 27.7. When the user simulates a producer loop error by pressing the PRODUCER ERROR button, the event case handling the value change of the PRODUCER ERROR button generates a custom error with code `5020` and the text `Producer Error`. In this figure, an **Error Ring** constant is used to generate the custom error (see the help file in Figure 4.2). The error ring is a pre-defined error cluster that can be used as an error source on the block diagram.

The guard clause downstream of the producer loop event structure immediately catches the error and executes its *Error* case. A Boolean True is sent to the producer loop conditional terminal to terminate the producer loop, and the *Shutdown* state is placed at the front of the queue for immediate execution using the **Enqueue Element At Opposite End** function. Notice that this function is placed on a parallel error wire, and that this error wire is then merged with the main error wire of the producer loop. This was done because the only time this particular **Enqueue Element At Opposite End** function will ever execute is if an error is present on the main error wire of the producer loop. If this function was placed on the main error wire, it would not execute; it would simply pass the error on to its error output terminal. If that happened, the *Shutdown* state would not be placed on the queue and the program would hang. The producer loop would shut down, but not the

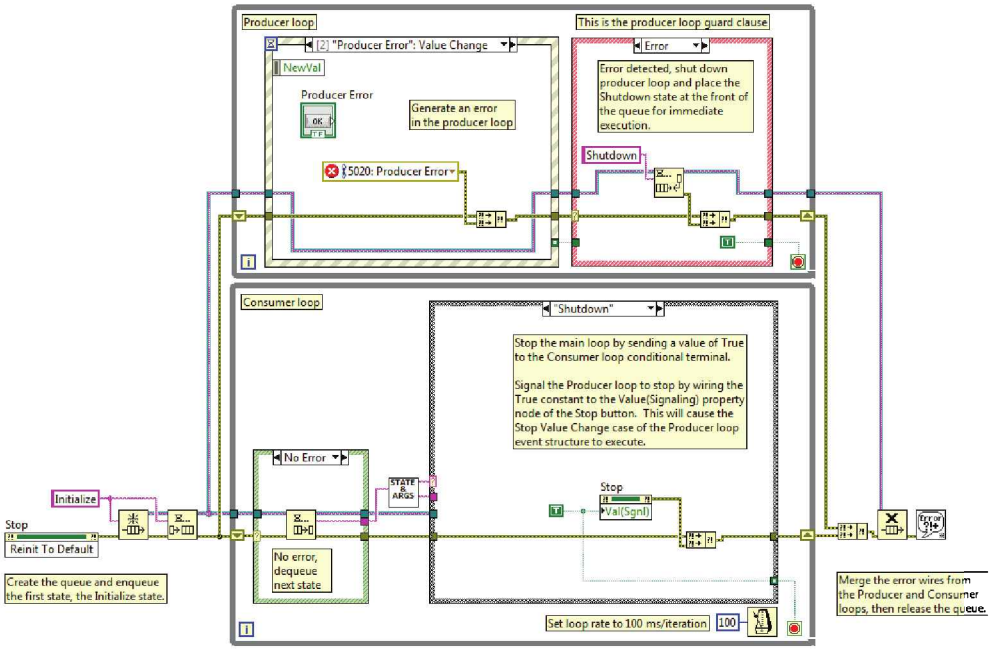


Figure 27.7: Stopping the `Multitest` VI by generating an error in the producer loop

consumer loop. Pressing the STOP button again at that point would not help, since the event structure that detects the value change of the STOP button was shut down with the producer loop.

From this point on, the program responds normally. The consumer loop dequeues the *Shutdown* state and executes it. The *Shutdown* state shuts down the consumer loop and sends a value change event to the producer loop via the **`Value(Signalling)`** property node of the STOP button. This has no effect since the producer loop already has been shut down.

When the two loops are shut down, their error wires are merged. The producer loop error is passed on to the **`General Error Handler`** function at the end of the chain. The **`Release Queue`** function still functions even with an error present at its input terminal, because it does not have the standard error functionality of most LabVIEW functions. When the **`General Error Handler`** executes, the following message is displayed in a pop-up dialog window: `Error 5020 occurred at Producer Consumer Multitest.vi.Possiblereason(s): Producer Error.` The net effect of this sequence is that both loops are shut down when an error is detected in the producer loop and the proper error is reported.

The third way to shut down the producer–consumer `Multitest` VI is to generate an error in the consumer loop. This can be simulated by pressing the CONSUMER ERROR button on the front panel. The shutdown procedure is shown in Figure 27.8. The event case that handles the value change of the CONSUMER ERROR button places the *Consumer Error* state on the queue. Since there is no error in the producer loop,

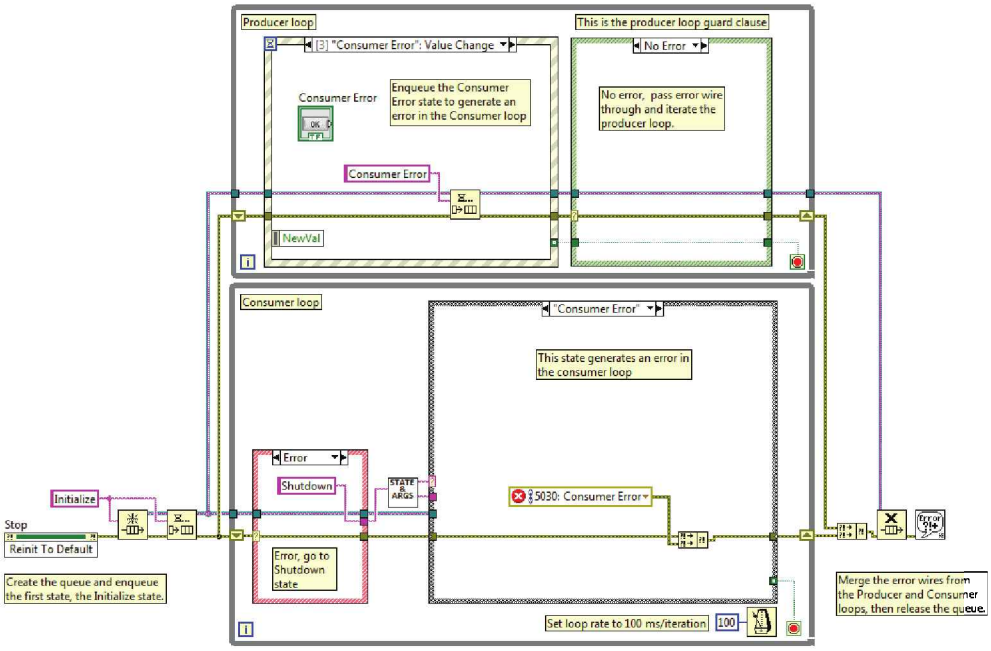


Figure 27.8: Stopping the `Multitest` VI by generating an error in the consumer loop

the guard clause executes normally, and the producer loop iterates again. The event structure waits for the next event, and the producer loop goes to sleep.

The *Consumer Error* state is then dequeued and executed in the consumer loop. The *Consumer Error* state generates a custom error with code `5030` and the text `Consumer Error`. The consumer loop then iterates, carrying the custom error on the error shift register to the next iteration of the consumer loop. The guard clause catches this error and sends the consumer loop state machine directly to the *Shutdown* state, which can be seen in Figure 27.7. The Boolean True constant in this state is used to shut down the consumer loop. This constant is also wired to the **`Value(Signalling)`** property node of the STOP button. This causes a value change event to be generated, as well as changing the value of the STOP button. The event structure in the producer loop detects this value change and enqueues the *Shutdown* state at the front of the queue using the **`Enqueue Element At Opposite End`** function. This has no effect, since the consumer loop has already been shut down. The new value of the STOP button is sent to the conditional terminal of the producer loop, causing the producer loop to also shut down. The error wires from the two loops are merged and the merged error output is passed to the **`General Error Handler`** function. When the **`General Error Handler`** executes, the following message is displayed in a pop-up dialog window: `Error 5030 occurred at Producer Consumer Multitest.vi. Possible reason(s): Consumer Error`. The net effect of this sequence is that both loops are shut down when an error is detected in the consumer loop and the proper error is reported.

Notice that the **`Value(Signalling)`** property node was put on a parallel error wire in the *Shutdown* case of the consumer loop for just this contingency. If the property node had been placed on the main consumer error wire, the consumer error would have caused the property node to skip its execution and simply pass the error on to its error output terminal. This means the value change event would not have been generated, and the producer loop would not have shut down. Unlike the producer error case, the program would not hang if this happened. The program would not respond to the RUN TEST 1, RUN TEST 2, and CONSUMER ERROR buttons, since the consumer loop would be shut down. But the program could still be terminated by pressing the STOP button or the PRODUCER ERROR button. Either of these buttons would shut down the producer loop, and from that point, on the program would terminate normally.

27.4 User Events

Signalling an event structure is an effective way to communicate between parallel loops. A limitation of the **`Value(Signalling)`** property node is that the property node must be tied to an existing front panel object. It would be useful if there was a method of signalling an event structure that didn't involve front panel objects at all.

This can be done with user events. User events are events that can be defined by the developer and generated programatically. User events are powerful and flexible, but using them involves a multi-step process:

1. Create the user event.
2. Register the user event to create an event registration reference.
3. Wire the event registration reference to the dynamic input terminals of the event structure.
4. Configure an event case to handle the user event.
5. Generate the user event when needed.
6. Unregister the user event when the event structure loop terminates.
7. Destroy the user event when the program terminates.

This process is illustrated it Figure 27.9. First, the user event must be created using the **`Create User Event`** function. The helpfile for the **`Create User Event`** function is shown in Figure 27.10. Notice the `user event data type` input at the upper-left terminal of the icon. A constant or control must be wired to this input to determine the data type of the user event. When a user event is generated, data of this data type is transmitted to the event structure along with this event. The data sent with the user event can be read using the event data node in the event case handling the user event.

The item wired to the `user event data type` input needs a label. The label is used to name the user event when an event structure is configured to handle the event. The label is not an issue if a control is wired to the `user event data type` input, since controls always have unique labels. But if a constant is used as an input,

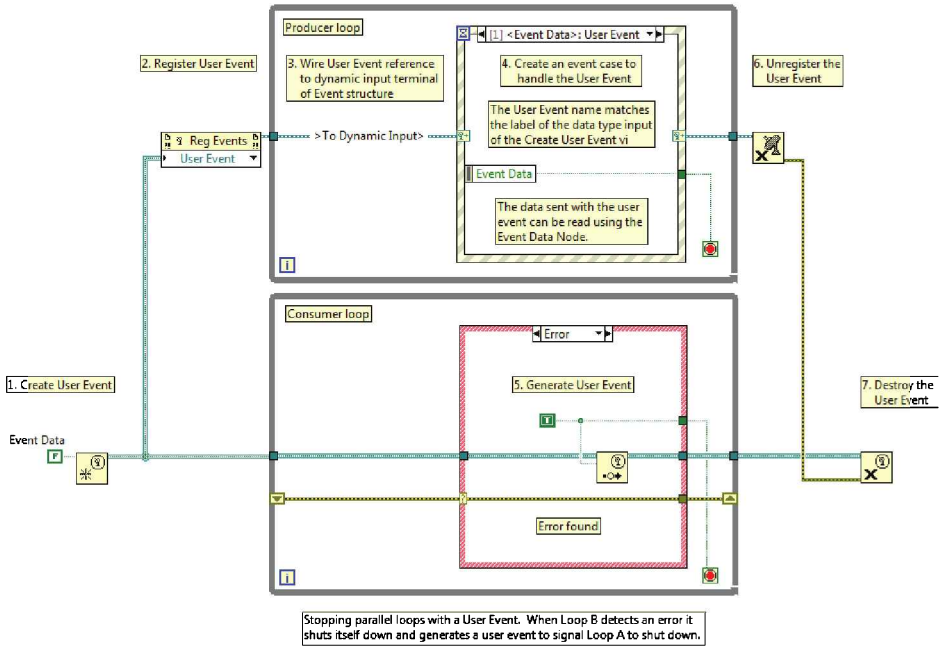


Figure 27.9: Using user events to stop parallel loops

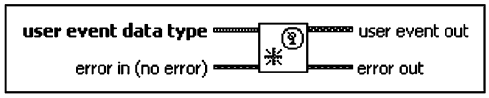
Create User Event Function

Owning Palette: Events Functions

Requires: Base Package

Returns a reference to a user event. LabVIEW uses the **user event data type** you wire to determine the event name and data type of the event. Wire the **user event out** output to a Register For Events function to register for the event. Wire the **user event out** output to a Generate User Event function to send the event and associated data to all Event structures registered for the event.

Details Examples



Add to the block diagram Find on the palette

user event data type is a cluster of elements or an individual element whose data type and label define the data type and name of the user event.

Note LabVIEW uses owned labels to identify data the user event carries, so label each element of the input with a descriptive owned label. If you do not add a label, the user event will not appear in the Event Data Node of an Event structure when added.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

user event out returns the strictly typed user event refnum.

error out contains error information. This output provides standard error out functionality.

Create User Event Details

The **user event data type** is a cluster of elements or an individual element. If you wire a cluster to the **user event data type** input, LabVIEW uses the type name of the cluster as the name of the user event. The names and data types of the cluster elements define the event data the user event carries. If you wire an individual data element to the **user event data type** input, LabVIEW uses the type name of the element as the name of the user event and as the name of the single data item it carries, which is the same data type as the element. The actual value you wire to the function is not important because LabVIEW uses only the name and data type to define the user event. The **user event out** output is a strictly typed refnum that includes the user event name and event data type.

You can wire the **user event out** output to the Generate User Event function and the Destroy User Event function. Wire the **user event out** output to the Register For Events function to dynamically register for notification when the user event occurs. When you handle a user event in an Event structure, the name of the user event appears in the event selector label at the top of the Event structure, and the user event data fields appear in the Event Data Node inside the left border of the event case.

Note If you obtain a user event reference in one application instance, you cannot use that user event reference in another application instance. If you attempt to use a user event reference in another application instance, LabVIEW returns error 1500.

Figure 27.10: Help file for the `Create User Event` function

as is done in Figure 27.9, the constant must be given a label. In Figure 27.9, the label *Event Data* is given to the user event data type constant.

After the user event has been created it needs to be registered. Events are registered using the **`Register For Events`** function. This help file is shown in Figure 27.11. Dynamic event registration allows the developer to programatically control which events will be detected by an event structure and when they will be detected. For instance, when an event structure is created, it is automatically configured to detect events related to the current LabVIEW application, events related to

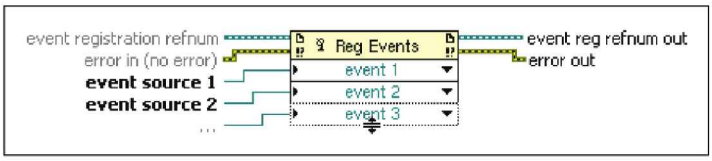
Register For Events Function

Owning Palette: Events Functions

Requires: Base Package

Dynamically registers events. The events for which you can register depend on the type of the reference you wire to each **event source** input. Wire the **event reg refnum out** output to an Event structure or to another Register For Events function.

Details Examples



Add to the block diagram Find on the palette

event registration refnum is a reference to an existing event registration a Register For Events function created.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

event source 1..n is a reference to an application, VI, control, or user event. References must be to local objects. You cannot wire a reference to a remote object. If you wire an array or cluster of refnums to this input, LabVIEW registers all elements of the array or cluster for the event.

event reg refnum out returns the reference to a new or existing event registration.

error out contains error information. This output provides standard error out functionality.

Register For Events Details

Each **event source** input is a reference to an application, VI, control, or user event. Each kind of event source can produce a variety of events. To specify which event you want LabVIEW to begin detecting for a particular **event source**, right-click the data item for the event source and select the event to detect. After this function executes, LabVIEW detects, or registers, every repetition of that event that occurs. LabVIEW stores these events in a queue so that an Event structure can handle the events later in the VI.

You can wire the **event reg refnum out** terminal of the Register For Events function to the dynamic event terminals on the Event structure border, to the Unregister For Events function, or to the top left input of another Register For Events function. If you wire the top left input of the Register For Events function, the function modifies the existing registration information associated with that refnum instead of registering the event again.

Registered events stay registered until you explicitly unregister them or until the VI that registered the events finishes running or you abort the VI. If the VI which registered for events was a subVI, events are unregistered when that VI's top-level VI finishes execution or is aborted. If you use the Run VI method to execute a subVI that registered for events, events are unregistered when the subVI finishes execution or is aborted.

Figure 27.11: Help file for the `Register For Events` function

the VI that the event structure is in, and events related to the front panel controls and indicators of the VI that the event structure is in. Dynamic event registration can be used to allow the event structure to respond to events in other VIs running on the same computer. Dynamic registration also can be used to control when an event structure can respond to front panel controls. But these topics are beyond the scope of this book. For our purposes, the important part of dynamic event registration is that event structures can only respond to user events after the events have been registered.

The output of the **`Register For Events`** function is an event registration reference. This reference can be wired to the dynamic event terminals of an event

structure to enable the event structure to detect the user event. This is demonstrated in Figure 27.9. The terminals with the satellite dish icons on the event structure are the dynamic event terminals, and the event case shown in the figure is the event case that handles the user event. Notice that the name of the event case includes the label of the data type input of the **`Create User Event`** function.

Wiring Tip: If the dynamic event terminals are hidden, right-click on the edge of the event structure and select *Show Dynamic Event Terminals*. If you create an event structure from the *Functions... Structures* palette, it will have the dynamic event terminals hidden by default. If you create an event structure from the *Functions... Dialog & User Interface... Events* palette, the dynamic event terminals will be visible by default.

User events are generated by the **`Generate User Event`** function. This help file is shown in Figure 27.12. This function has an input for the user event reference

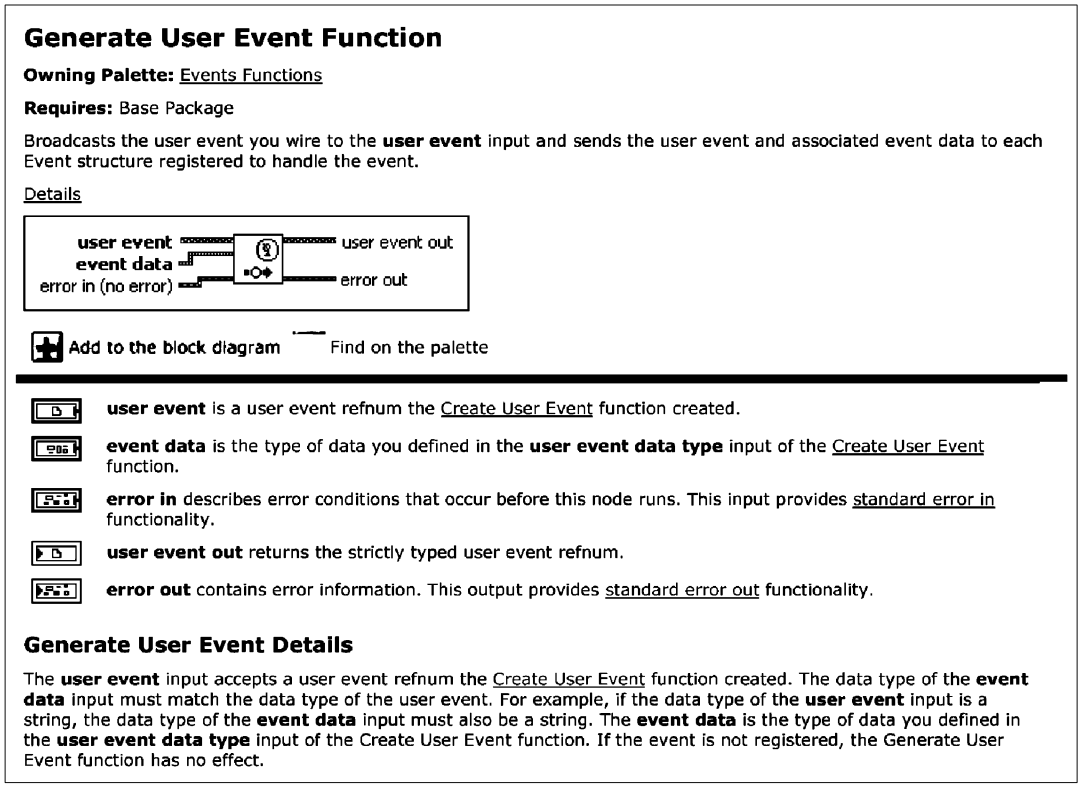


Figure 27.12: Help file for the **`Generate User Event`** function

created by the **Create User Event** function, as well as an input for the event data. The event data must be of the same data type used in the creation of the user event.

When the event is generated, the event data is transmitted along with the event. This can be seen in Figure 27.9. The event is generated when an error is detected in the consumer loop. The value of the Boolean used to stop the consumer loop is passed to the **Generate User Event** function as event data. The event data is then retrieved in the event structure in the producer loop using the event data node on the left side of the user event error case. Notice that the text in the node is identical to the label of the data type input of the **Create User Event** function. The event data is then passed to the conditional terminal of the producer loop to shut down the loop. Both loops shut down when an error is detected in the consumer loop.

It is good programming practice to close references after they are no longer needed. This is a two-step process with user events. First, the user event needs to be unregistered. This is done with the **Unregister For Event** function. This help file is shown in Figure 27.13. Its use is demonstrated in Figure 27.9. When the producer loop terminates, the event structure also terminates and there is no longer any need for the user event, so it can be safely unregistered. The **Unregister For Event** function has standard error-handling properties. It will not execute if an error is present at the error input; the error will simply be passed on the error output terminal.

Once the user event has been unregistered, it can be destroyed using the **Destroy User Event** function. The help file for this VI is shown in Figure 27.14. Unlike the **Unregister For Events** VI, the **Destroy User Event** VI will execute even if an error is present at the error input terminal. The use of the **Destroy User Event** VI is demonstrated in Figure 27.9. Notice that even though

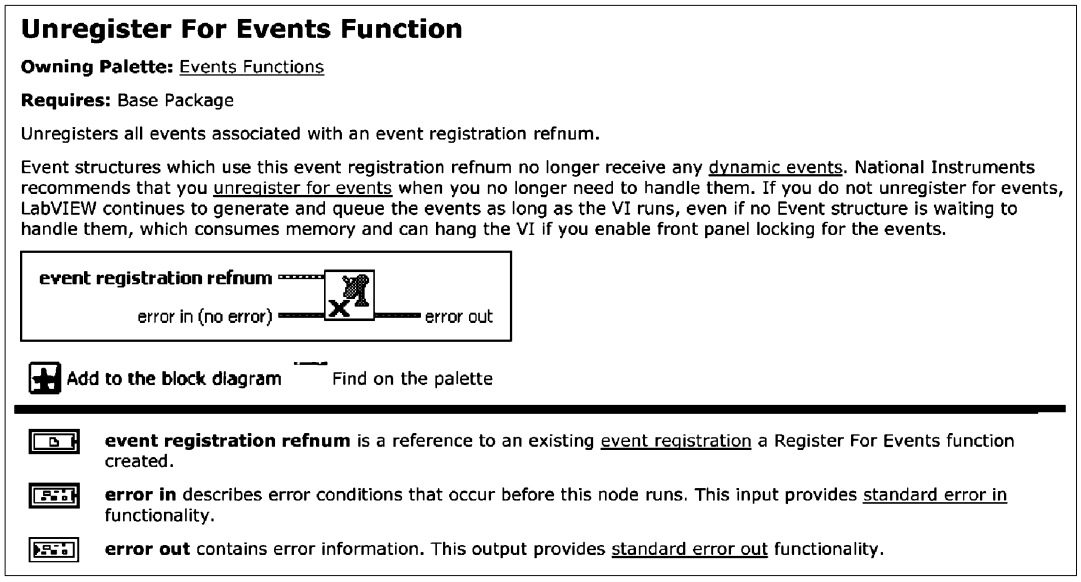


Figure 27.13: Help file for the `Unregister For Events` function

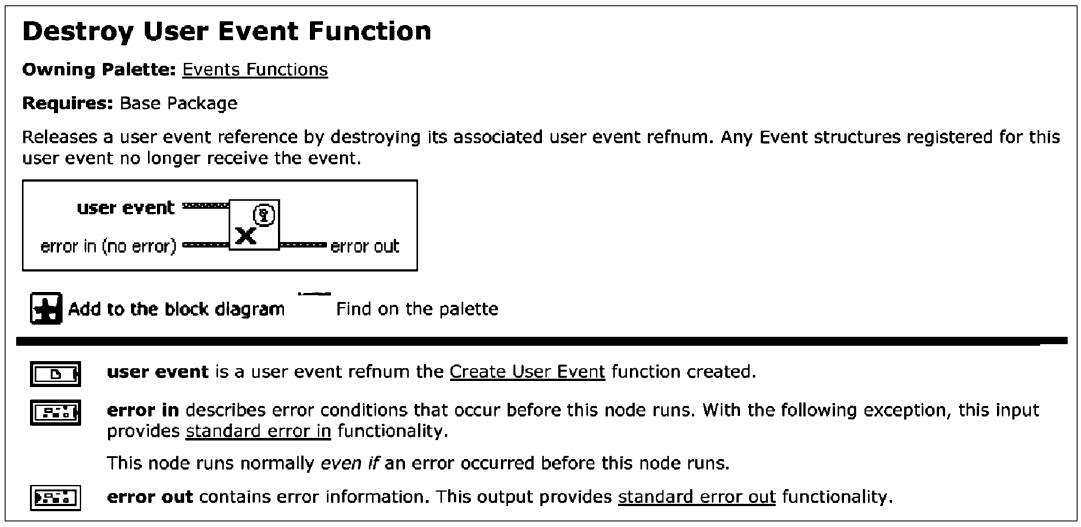


Figure 27.14: Help file for the `Destroy User Event` function

error handling has been neglected in this figure, an error wire runs from the **`Unregister For Events`** VI to the **`Destroy User Event`** function. This is to ensure that the user event is unregistered before it is destroyed.

27.5 Producer–Consumer Pattern with User Events

This version of the producer–consumer `Multitest` VI substitutes a user event for the **`Value(Signalling)`** property node of the STOP button, as shown in Figure 27.15. This VI follows the same steps illustrtated in Figure 27.9. First, the user event is created in the input area of the loops using the **`Create User Event`** function. The user event is given a Boolean data type by wiring a Boolean constant to the data type input terminal. The constant is given the label *Error* to indicate that the user event is generated when an error is detected.

The user event is then registered using the **`Register For Events`** function. The resulting registration reference is then wired to the dynamic event terminals of the event structure in the producer loop. An event case is configured to respond to the user event, as shown in Figure 27.16. The *Error* event data is extracted using the event data node on the interior left edge of the event case and is passed on to the producer loop conditional terminal.

When an error occurs in the consumer loop, the user event is generated using the **`Generate User Event`** function. A Boolean True constant is wired to this function as the event data. The Boolean True constant is also used to shut down the consumer loop. The user event then triggers the event case that handles the user event. The event data is used to shut down the producer loop.

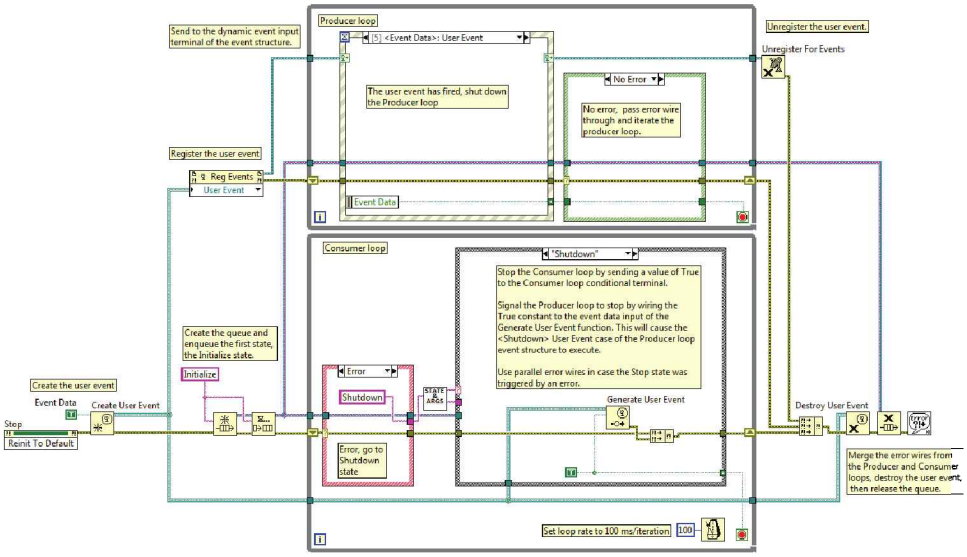


Figure 27.15: Producer–consumer version of the `Multitest` VI with user events

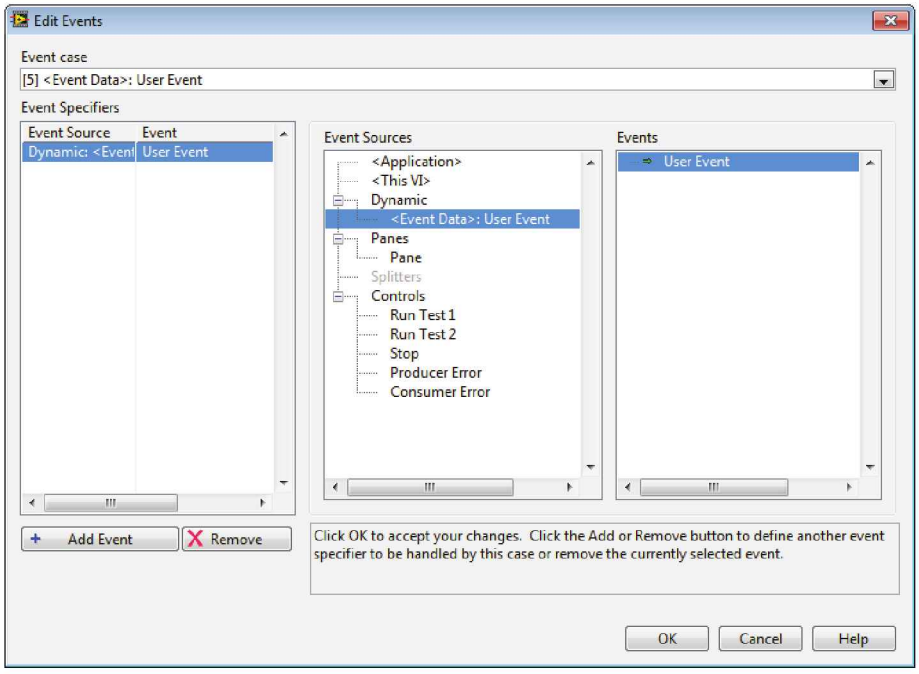


Figure 27.16: Configuration of the event case handling the user event

When the producer loop shuts down, there is no longer any need for the user event to be registered, so it is unregistered using the **`Unregister For Events`** function. Notice that this VI is placed on a parallel error line so that it will execute even if an error has developed in the producer loop. The error wire from this function is then merged with the error wires from the producer and consumer loops. This ensures that the event is unregistered before it is destroyed by the **`Destroy User Event`** function that follows the error merge. The queue is then released.

27.6 Value(Signalling) or User Events?

The previous examples demonstrate that user events require more effort and take up more screen real estate than *Value(Signalling)* property nodes. *Value(Signalling)* property nodes are easier to use and more compact, but they one major drawback: They can only be used to mimic the value change of an existing front panel control, in this case the STOP button. If an application does not have a STOP button or something like it, the *Value(Signalling)* technique won't work.

Furthermore, the action of the *Value(Signalling)* property node is indistinguishable from manipulating the corresponding front panel control. In the `Multitest` examples, this means that the producer loop cannot tell if the user has pressed the STOP button or if an error has occurred in the consumer loop. Both of these situations send the same signal to the event structure.

User events have the advantage of being independent of the front panel. They are created programatically and they are fired programatically. They also allow the developer to send unique signals to the event structure tailored for different situations. In Figure 27.15, the event structure in the producer loop has different event cases to handle the user event and the value change of the STOP button. This allows the developer to code different responses to normal stop requests and to error detection in the consumer loop.

27.7 Error Handling in the Producer–Consumer Architecture

Error handling should always be a priority when developing VIs. This is especially true for parallel-loop architectures. Parallel-loop architectures are more challenging because errors can occur in either of the loops. The parallel-loop VI needs to be designed in such a way that an error in one loop can shut down all loops. If the error wires are not carefully managed, an error may shut down only one of the loops and may cause the VI to hang.

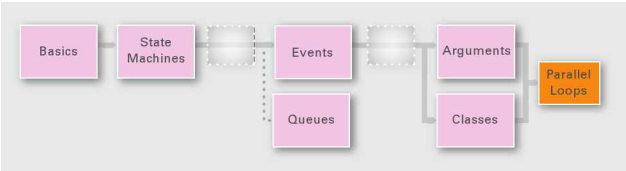
Consider the producer loop. In the examples in this chapter, the guard clause always comes after the event structure, but in the previous chapters, the guard clause always came before the case structure that contained the event structure. Why is this?

This design choice was influenced by error handling considerations. Imagine placing the guard clause before the event structure in the producer loop. If code within

the event structure generated an error, the error would be placed on the producer loop shift register, and the producer loop would iterate. The error would then be caught by the guard clause, and the *Shutdown* state would be placed at the front of the queue. But then the event structure would execute, and the producer loop would go to sleep until the next event occurred. This is undesirable behavior. Why put the producer loop to sleep right after an error has been detected? In this case, it is better to put the guard clause after the event structure. An error in the event structure code would then be caught immediately and actions could be taken to shut the loops down. The event structure would not execute again after the error had been detected; thus, the producer loop would not be put to sleep.

The difference between the guard clause in the producer loop and the guard clause case structures in the previous chapters of this book is that the producer loop guard clause does not dequeue elements from the queue. The guard clause in the single loop state machines can bypass the queue and send the state machine directly to the *Shutdown* state when an error is detected. In this way, they prevent the loop from going to sleep after an error has been detected. The producer loop can't do this. The producer loop can enqueue a *Shutdown* state for the consumer loop, but it cannot bypass its own event structure to shut the producer loop down. That's why it makes more sense in the producer loop to check for errors after the event structure rather than before.

Error wires also need to be carefully managed and merged to ensure that all errors will reach the error handler routine and that the correct error is reported. In the examples in this chapter, the errors are merged after both loops shut down using the **`Merge Errors`** function. Notice too that the errors were merged before the user event was destroyed and before the queue was released. The error wires are used to enforce dataflow. Another rule of thumb for parallel-loop architectures is that all loops should be shut down before shared resources are deallocated or destroyed. Placing the **`Destroy User Event`** and **`Release Queue`** functions downstream of the error merging function guarantees that both loops have been shut down before the shared resources are deallocated or destroyed.



28 Combatting Race Conditions

Race conditions occur when the order of execution of a piece of code affects the outcome of that piece of code. This usually happens when parallel operations share a resources such as a local variable or a queue. We've already seen two examples of race conditions in Chapter 13. Figure 13.6 shows an example where elements are placed on a queue by three parallel operations. Since the operations happen in parallel with no data dependence it is impossible to determine in what order the elements will be placed on the queue. The order may change each time the code is run, so this is a race condition. Likewise, Figure 13.7 shows an example where multiple attempts to read a queue are made by parallel dequeuing operations. Since there is no data dependence between the dequeuing operations it is impossible to predict what value will be displayed by each indicator. The values displayed read may change each time the code is run so this too is a race condition.

Race conditions are a serious issue because they behave unpredictably and can lead to incorrect results. For example, suppose that two portions of a banking program work in parallel. One portion handles withdrawals and the other portion handles deposits. Let's also assume that the account balance is a shared resource accessible to both portions of the program. A customer has $100 in his account and he wants to make a $50 withdrawal and deposit a $25 check. Here is a possible scenario as illustrated in Figure 28.1:

1. The withdrawal portion of the program reads the account balance ($100) and subtracts $50.
2. The withdrawal portion writes the new account balance ($50).
3. The deposit portion reads the balance ($50) and adds $25.
4. The deposit portion writes the new balance ($75).

This sequence gives the correct answer. We would also get the correct answer if the program followed the same sequence but performed the deposit before the withdrawal. But what if the sequence illustrated in Figure 28.2 happens instead:

1. The withdrawal portion of the program reads the account balance ($100) and subtracts $50.
2. The deposit portion reads the balance ($100) and adds $25.

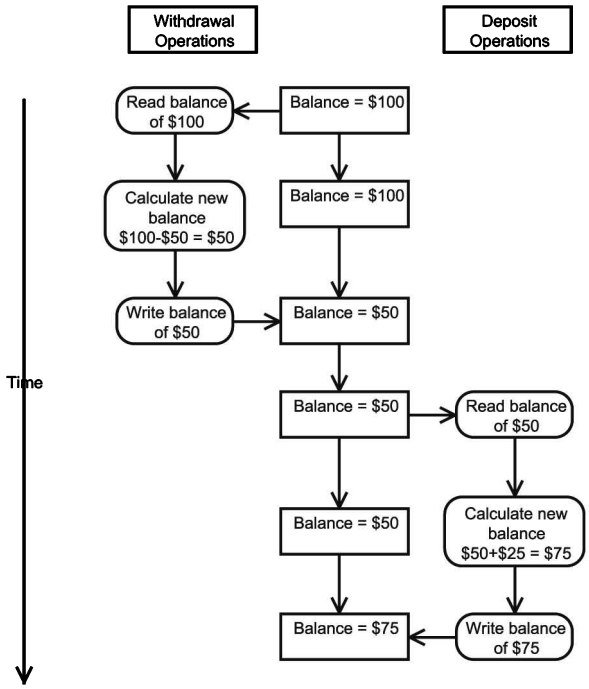


Figure 28.1: This is the intended sequence of withdrawal and deposit with a remaining balance of $75

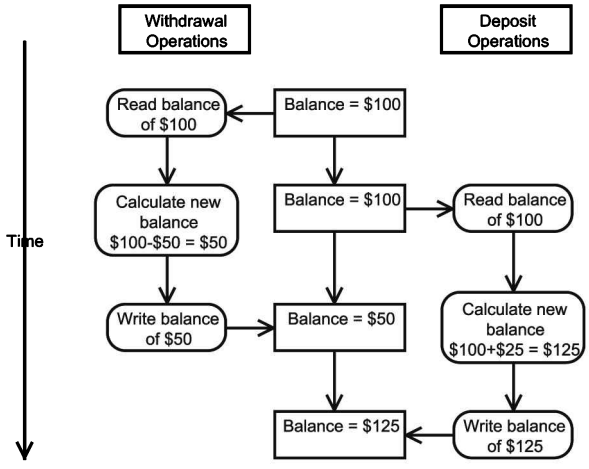


Figure 28.2: This sequence illustrates a race condition with a remaining balance of $125

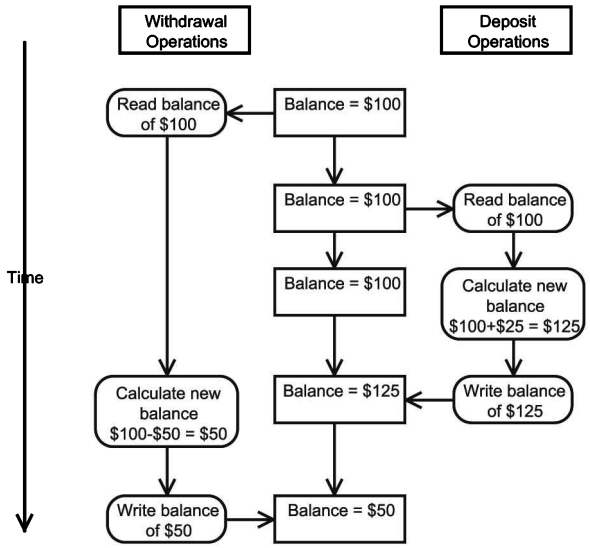


Figure 28.3: This sequence illustrates a race condition with a remaining balance of $50

3. The withdrawal portion writes the new account balance as $\$100 - \$50 = \$50$.
4. The deposit portion overwrites the new balance $\$100 + \$25 = \$125$.

This is the wrong answer. The sequence illustrated in Figure 28.3 could happen as well:

1. The withdrawal portion of the program reads the account balance ($100) and subtracts $50.
2. The deposit portion reads the balance ($100) and adds $25.
3. The deposit portion writes the new balance $\$100 + \$25 = \$125$.
4. The withdrawal portion overwrites the new account balance as $\$100 - \$50 = \$50$.

This is also incorrect. This is an example of a race condition. Three different account balances are possible, depending on the order of the parallel deposit and withdrawal operations. The problem is that the withdrawal and deposit operations are not *blocking*. The wrong answer is generated when both operations read the original balance and then proceed in parallel. What we need is a way to allow the withdrawal and deposit operations to block access to the account balance while it is being manipulated. A procedure like the one shown in Figure 28.4 would guarantee the right answer:

1. The withdrawal operation locks the account balance and blocks access to it.
2. The withdrawal operation reads the account balance ($100) and subtracts $50.

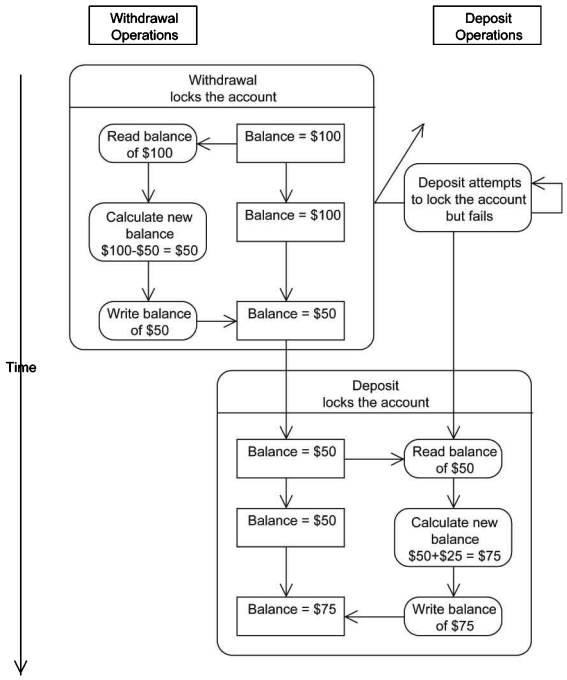


Figure 28.4: The race condition is avoided by allowing the operations to block each other

3. The deposit portion attempts to lock the account balance but can't because it is blocked.
4. The withdrawal portion writes the new account balance ($50) and releases the account.
5. The deposit portion locks the account and blocks access to it.
6. The deposit operation reads the balance ($50) and adds $25.
7. The deposit portion writes the new balance ($75) and releases the account.

This procedure will give the right answer whether the deposit happens first or the withdrawal happens first as shown in Figure 28.5. In either case, the blocking action prevents a race condition from occurring. Even though the withdrawal and deposit routines are executing in parallel, the blocking action prevents them from accessing the shared resource (the account balance) simultaneously.

Figure 28.6 shows an example of code with the potential for a race condition. A **`Flat Sequence Structure`** is used to enforce the order of execution. The code in the left frame executes first, followed by the code in the right frame. The help file for this structure is shown in Figure 28.7. A COUNTER indicator is initialized to zero in the first frame. In the second frame, two parallel **`For`** loops are used to access local variables of the COUNTER indicator and increment it. The loops each run for 1000 iterations. What value will the counter hold once the loops have completed?

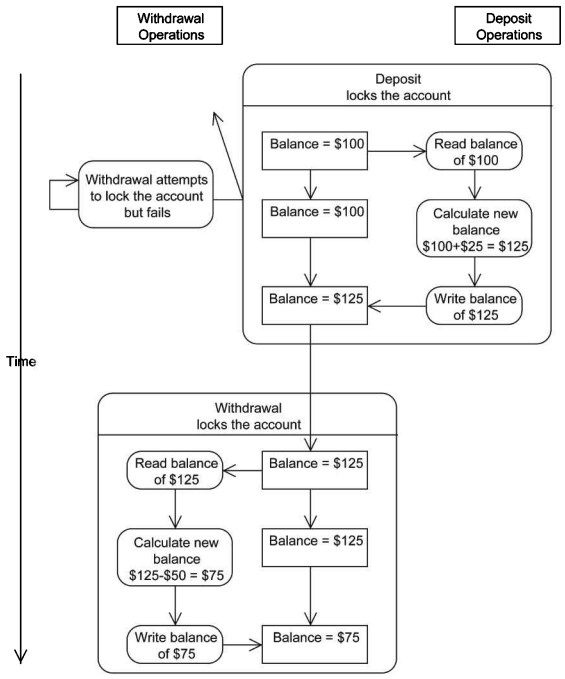


Figure 28.5: The race condition is avoided no matter which set of operations occurs first

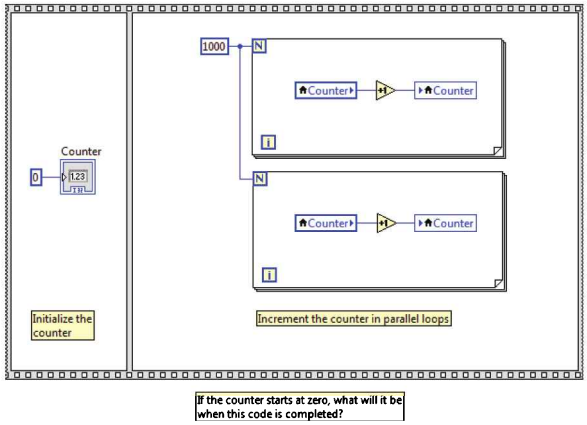


Figure 28.6: This code has the potential for a race condition

The "obvious" answer is that the counter will contain the number 2000, since the loops iterate for a total of 2000 times. The reasoning goes like this:

1. The top loop reads the value of the counter, increments it, and writes the new value to the counter.

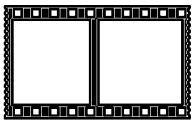
Flat Sequence Structure

Owning Palette: Structures

Requires: Base Package

Consists of one or more subdiagrams, or frames, that execute sequentially. Use the Flat Sequence structure to ensure that a subdiagram executes before or after another subdiagram.

Data flow for the Flat Sequence structure differs from data flow for other structures. Frames in a Flat Sequence structure execute from left to right and when all data values wired to a frame are available. The data leaves each frame as the frame finishes executing. This means the input of one frame can depend on the output of another frame.



Add to the block diagram Find on the palette

Unlike in the Stacked Sequence structure, you do not need to use sequence locals to pass data from frame to frame in the Flat Sequence structure. Since the Flat Sequence structure displays each frame on the block diagram, you can wire from frame to frame without using sequence locals and without hiding code.

When you add or delete frames in a Flat Sequence structure, the structure resizes automatically.

To convert a Flat Sequence structure to a Stacked Sequence structure, right-click the Flat Sequence structure and select **Replace with Stacked Sequence** from the shortcut menu. **(ETS, VxWorks, Windows)** To convert a Flat Sequence structure to a Timed Sequence structure, right-click the Flat Sequence structure and select **Replace with Timed Sequence** from the shortcut menu.

You cannot drag tunnels across the frames of a Flat Sequence structure.

To avoid overusing Flat Sequence structures, attempt to control the data flow of your VI by establishing data dependency or using flow-through parameters.

Figure 28.7: Help file for the `Flat Sequence Structure`

2. The bottom loop reads the value of the counter, increments it, and writes the new value to the counter.

The result is that each time a loop iterates, the counter variable gets incremented. But this reasoning makes an assumption that isn't wired into the code. The assumption is that the "read counter–increment counter–write counte" sequence in the top loop never interferes with the corresponding sequence in the bottom loop. But the two loops are totally unsynchronized, so the following sequence is entirely possible. Assume that the counter has been initialized to zero:

1. The top loop reads the value of the counter local variable (`counter = 0`).
2. The bottom loop reads the value of the counter local variable (`counter = 0`).
3. The top loop increments value of the counter and writes the new value to the counter local variable (`counter = 1`).
4. The bottom loop increments value of the counter and writes the new value to the counter local variable (`counter = 1`).

The second loop overwrites the result of the first loop. The end result of this sequence is that two loop iterations occurred but the counter was only incremented once.

This means that we can't predict the value of the counter after the code in Figure 28.6 executes. If the race condition happened every loop iteration the final value of the counter would be 1000. If the race condition never happened the final value of the counter would be 2000. On any given run of the code the final value of the counter could be any integer value between 1000 and 2000, inclusive. This is a race condition.

Race conditions are insidious because they can be very difficult to detect. Race conditions only happen when events occur in a particular sequence, and this sequence may happen infrequently. This means that a code may pass all of its tests during the development phase without the race condition rearing its ugly head. The race condition can then manifest itself after the code has been released. Even if a race condition is suspected during the development phase, it can be very difficult or impossible to construct a test to capture it. The best defense against race conditions is to become familiar with the kinds of situations where they can arise and to construct code in such a way to prevent their occurrence.

28.1 Guarding Against Race Conditions

The primary defense against race conditions is to identify critical sections of the code and protect them. In Figure 28.6 the critical section of the code is the "read counter–increment counter–write counter" sequence that appears in each loop. This sequence needs to block the counter from being accessed while it is being incremented.

One way to do this is to place the critical section of the code within a non-reentrant sub-VI. Sub-VIs are non-reentrant by default, which means that only one copy of the sub-VI can run at any given time. A non-reentrant sub-VI cannot be run in parallel with itself.

The counter action engine shown in Figure 5.9 contains the entire "read counter–increment counter–write counter" sequence and protects it from interruption. The action engine can be used to defeat the race condition as seen in Figure 28.8. The top loop calls the counter sub-VI and increments the stored value. While this is happening, the lower loop waits because only one version of of the counter sub-VI can run at one time. When the sub-VI in the top loop completes, the lower loop runs the sub-VI and increments the stored value again. The final value of the stored counter will equal 2000 because the two loops ran a total of two hundred times and do not interfere with each other.

In Figure 28.8, the main error wire was used to enforce the execution order using dataflow instead of a sequence structure. The error inputs to the loops ensure that the counter is reset to zero before the loops begin. The merging of the error outputs of the loops ensures that the loops have both completed before the value of the counter is reported.

Another way to protect a critical section of code is through the use of semaphores. A physical semaphore is flag used for signalling. A software semaphore has a similar purpose. A semaphore can be acquired and released, and critical code can be protected by bracking it with an acquire semaphore/release semaphore pair of functions.

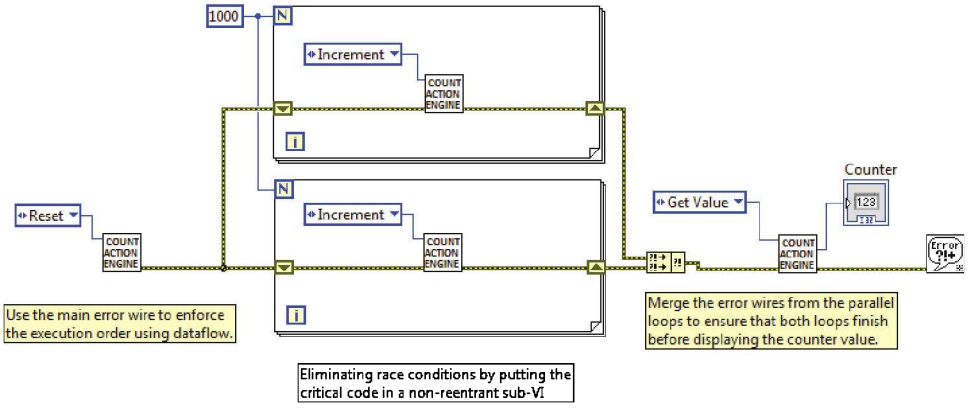


Figure 28.8: Eliminating race conditions by placing critical code in a non-reentrant sub-VI

When the critical code attempts to execute, it first tries to acquire the semaphore. If it is successful, the code executes. If another part of the code tries to acquire the semaphore while the critical code is running, it will be blocked until the critical code completes and releases the semaphore. This blocking protects the critical section of code and prevents race conditions.

Figure 28.9 demonstrates how to use semaphores to prevent the counter race condition. The semaphore is created using the **`Obtain Semaphore Reference`** function with the help file shown in Figure 28.10. The output of this function is a reference to the new semaphore. Since the semaphore is passed by-reference and not by-value, it can be used by parallel loops without creating a data dependency between the loops.

In each of the two parallel loops, the critical code is bracketed between an **`Acquire Semaphore`** function and a **`Release Semaphore`** function. The `Acquire Semaphore` function has a closed lock glyph on its icon to signify that it attempts to get a lock on the semaphore wired to its semaphore reference input. The help file for this function is found in Figure 28.11. The **`Release Semaphore`** function has an open lock glyph on its icon to signify that it releases its lock on the semaphore wired to its semaphore reference input. The help file for this function is shown in Figure 28.12.

Notice that the critical section of code has been placed inside of a case structure in each loop of Figure 28.9. This was done to use the error wires from the semaphore functions to enforce data flow. The error wire from the **`Acquire Semaphore`** function enters and exits the case structure before it enters the **`Release Semaphore`** function. This ensures that all of the code inside the case structure will execute after the semaphore is acquired and before the semaphore is released. This is important because the semaphore functions need to have a data flow dependence with the code that they protect. Data flow must establish the "acquire semaphore–critical code–release semaphore" sequence. The same technique was used upstream of the parallel loops to ensure that the counter was initialized before the loops began.

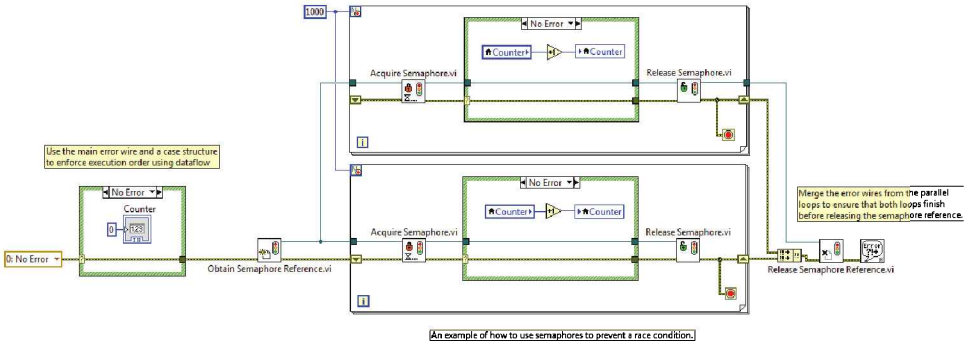


Figure 28.9: An example of eliminating a race condition through the use of semaphores

Obtain Semaphore Reference VI

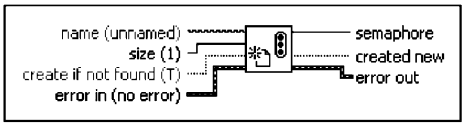
Owning Palette: Semaphore VIs

Requires: Base Package

Obtains a reference to an existing semaphore or creates a new semaphore and returns a reference to that semaphore.

Use this VI in conjunction with the other Semaphore VIs to implement a semaphore in LabVIEW.

Details Example



Add to the block diagram Find on the palette

name contains the name of the semaphore you want to look up or create. The default is an empty string to create an unnamed semaphore. If you wire **name**, LabVIEW searches for an existing semaphore with the same name and returns a unique reference to the existing semaphore. If a semaphore with the same name does not already exist and **create if not found** is TRUE, LabVIEW creates a new, named semaphore and returns a unique reference to that semaphore.

size specifies how many tasks can acquire the semaphore at the same time. If a named semaphore already exists, wiring a value to this parameter does not resize the semaphore. **size** must be greater than or equal to 1. The default is 1.

create if not found specifies whether you want to create a new semaphore if one with that name does not exist. The default is TRUE, which specifies that LabVIEW creates a semaphore if the semaphore does not exist. If **create if not found** is FALSE and LabVIEW cannot find a semaphore with the name you specify, LabVIEW returns error code 1534.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

semaphore is a reference to an existing or newly created semaphore. If you use this VI to obtain multiple references to the same named semaphore, each reference number is unique.

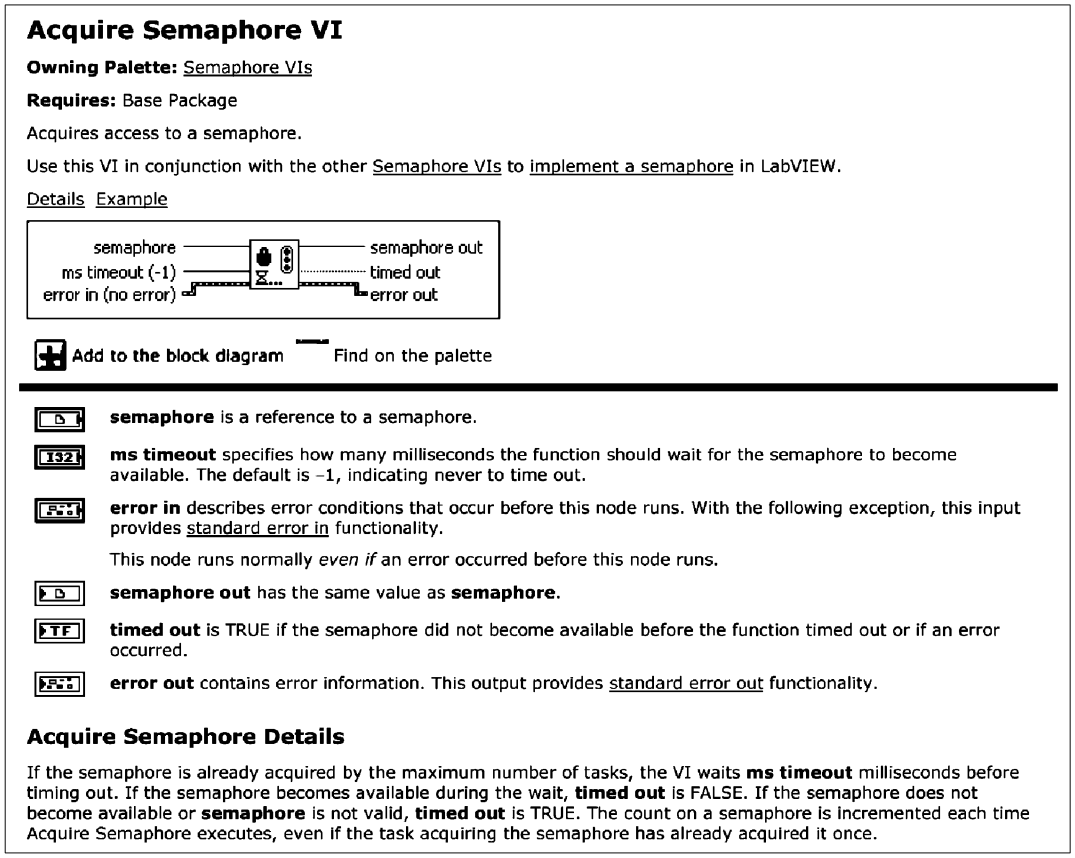
created new is TRUE if the VI created a new semaphore.

error out contains error information. This output provides standard error out functionality.

Obtain Semaphore Reference Details

You cannot use semaphores to communicate between LabVIEW application instances. If you obtain a reference to a semaphore in one application instance, you cannot use that semaphore reference in another application instance.

Figure 28.10: Help file for the `Obtain Semaphore Reference` function



Acquire Semaphore VI

Owning Palette: Semaphore VIs

Requires: Base Package

Acquires access to a semaphore.

Use this VI in conjunction with the other Semaphore VIs to implement a semaphore in LabVIEW.

Details Example

Add to the block diagram Find on the palette

semaphore is a reference to a semaphore.

ms timeout specifies how many milliseconds the function should wait for the semaphore to become available. The default is –1, indicating never to time out.

error in describes error conditions that occur before this node runs. With the following exception, this input provides standard error in functionality.

This node runs normally *even if* an error occurred before this node runs.

semaphore out has the same value as **semaphore**.

timed out is TRUE if the semaphore did not become available before the function timed out or if an error occurred.

error out contains error information. This output provides standard error out functionality.

Acquire Semaphore Details

If the semaphore is already acquired by the maximum number of tasks, the VI waits **ms timeout** milliseconds before timing out. If the semaphore becomes available during the wait, **timed out** is FALSE. If the semaphore does not become available or **semaphore** is not valid, **timed out** is TRUE. The count on a semaphore is incremented each time Acquire Semaphore executes, even if the task acquiring the semaphore has already acquired it once.

Figure 28.11: Help file for the `Acquire Semaphore` function

When both loops are finished, their error wires are merged, and the semaphore is destroyed using the `Release Semaphore Reference` function. This help file is shown in Figure 28.13. The semaphore is destroyed after the error wires are merged to ensure that both loops have completed before the semaphore is destroyed.

The code shown in Figure 28.9 will not experience a race condition. Both the top and bottom loops run in parallel, and will attempt to acquire the semaphore, but only one loop will succeed. The successful loop will then proceed to read the counter local variable, increment it, and write the new value back to the counter local. During this time the other loop will attempt to acquire the semaphore but will be unable to do so. This forces the loop to wait. When the successful loop is finished with the critical code, it releases the semaphore and iterates. The waiting loop then acquires the semaphore, forcing the other loop to wait. The second loop executes the critical code and releases the semaphore, and the pattern continues. The semaphores prevent both loops from accessing the counter local variable at the same time and thus eliminate the race condition.

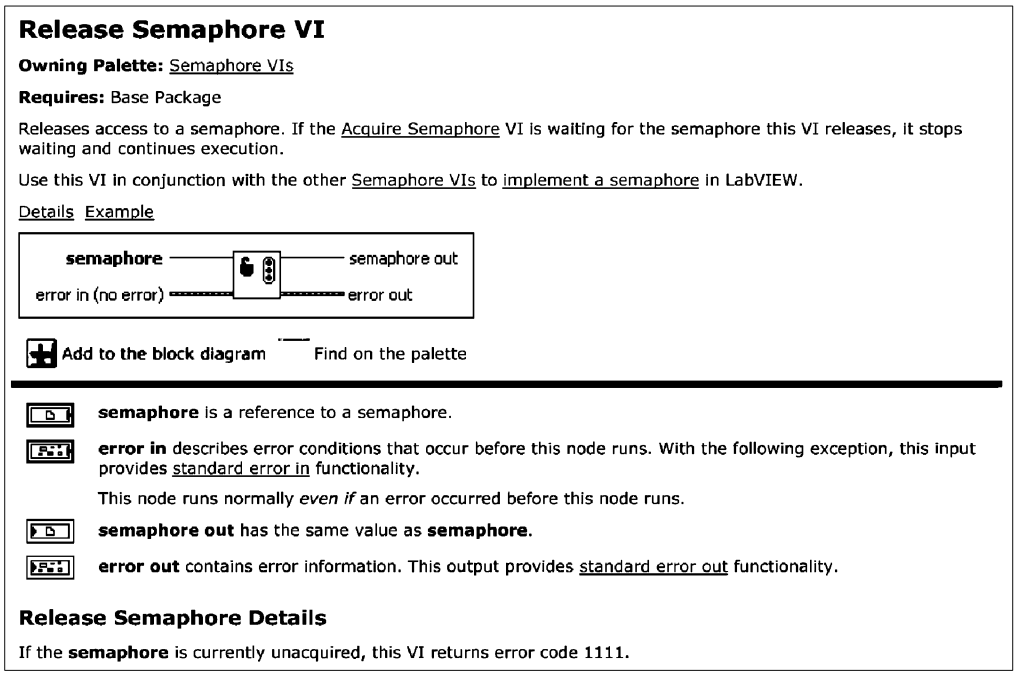


Figure 28.12: Help file for the `Release Semaphore` function

28.2 The Producer–Consumer Car Wash Controller

The previous chapter introduced the producer–consumer (events) design pattern. In this pattern, the producer loop contains an event structure to monitor the user interface. The producer loop responds to user input by placing commands on a queue for the consumer loop. The consumer loop dequeues these commands and acts on them while the producer loop continues to monitor the user interface.

The producer–consumer version of the car wash controller is shown in Figure 28.14. This version uses implementation of the car wash controller developed in Chapter 25 as a starting point. The event structure has been moved to the producer loop, eliminating the *Wait* state from the main case structure now found in the consumer loop.

The event structure shows the event case that handles the value change of the DELUXE WASH button. Compare this event case to the one shown in Figure 21.17. In the single loop version of the car wash controller both the Deluxe and Economy Wash queued states needed to end in the *Wait* state. This ensured that the state machine would transition back to the event structure to resume monitoring the purchase selector after the selected car wash was completed. This is no longer needed, since the event structure is now continuously running in parallel with the state machine in the consumer loop.

Release Semaphore Reference VI

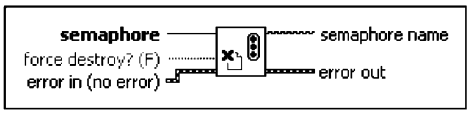
Owning Palette: Semaphore VIs

Requires: Base Package

Releases a reference to a semaphore.

Use this VI in conjunction with the other Semaphore VIs to implement a semaphore in LabVIEW.

Details Example



Add to the block diagram Find on the palette

semaphore is a reference to a semaphore.

force destroy? specifies whether to destroy the semaphore. If FALSE (default), LabVIEW destroys the semaphore only if no other references to the semaphore exist. If TRUE, LabVIEW releases all references to the semaphore and destroys the semaphore.

error in describes error conditions that occur before this node runs. With the following exception, this input provides standard error in functionality.

This node runs normally *even if* an error occurred before this node runs.

semaphore name is the name of the semaphore.

error out contains error information. This output provides standard error out functionality.

Release Semaphore Reference Details

All Acquire Semaphore VIs that are currently waiting on this reference to the semaphore time out immediately and return an error. This does not affect any Acquire Semaphore VIs that are waiting on a different reference to the same semaphore. However, if **force destroy?** is TRUE, all Acquire Semaphore VIs, including those that are waiting on a different reference to the semaphore, time out immediately and return an error.

Figure 28.13: Help file for the `Release Semaphore Reference` function

Contrast the treatment of the STOP button in Figures 28.14 and 21.17. The STOP button needs to be monitored in two places in the single-loop state machine. The STOP button needs to be polled every loop iteration during an active car wash, so it needs to be placed in the lobby of the state machine as an input to the guard clause. But the STOP also needs to be monitored while the state machine is in the *Wait* state, so an event case needs to be configured to handle the value change of the STOP button. The producer–consumer version of the car wash controller only needs to monitor the STOP button in one place: the event structure. Since the event structure is always active, we no longer need to poll the STOP button in the lobby of the consumer loop state machine. The STOP button now can be placed inside the event case handling its value change, as shown in Figure 28.15, ensuring that it will be read by LabVIEW when the event case is triggered.

Removing the STOP button from the lobby of the consumer loop state machine also simplifies the guard clause in the consumer loop. In the previous implementations of the car wash controller, we needed to insert the `Stop or Error` sub-VI upstream

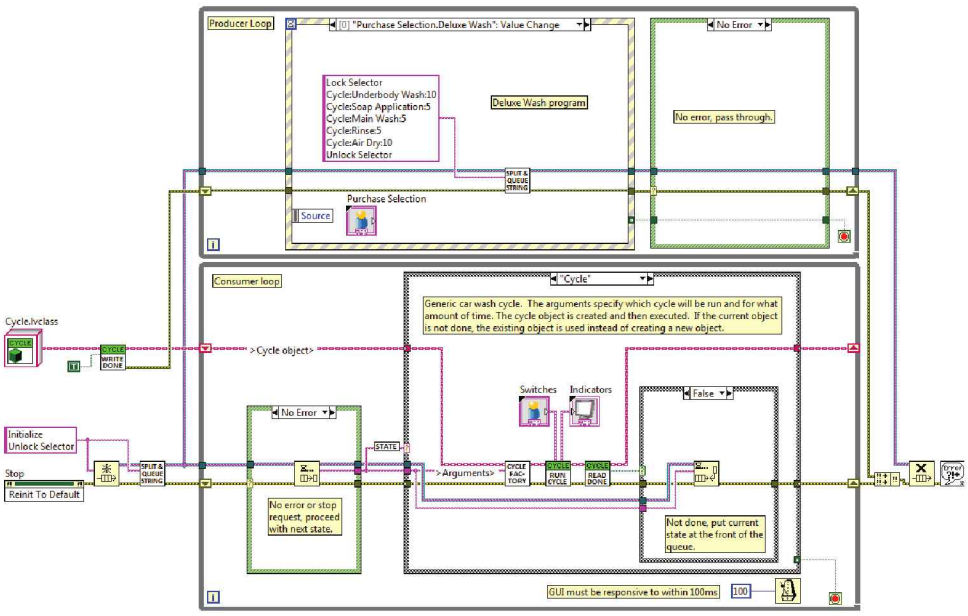


Figure 28.14: A producer–consumer (events) version of the car wash controller

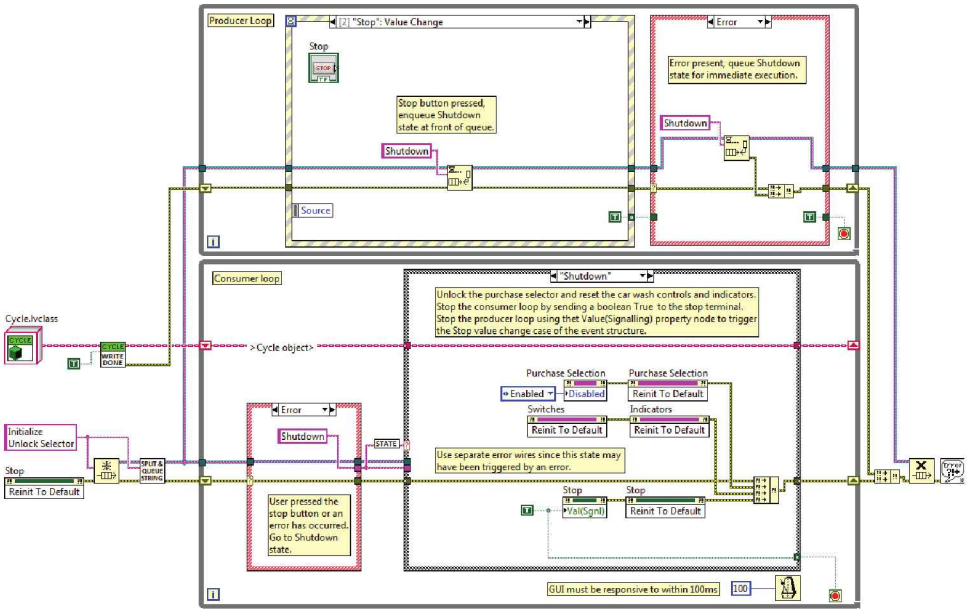


Figure 28.15: Shutting down the producer–consumer (events) version of the car wash controller

of the guard clause. The output of this sub-VI was wired to the selector input of the guard clause. But now that the STOP button has been moved out of the consumer loop, the guard clause needs to check only for errors. This means that the consumer loop main error wire can now be wired to the selector of the guard clause. This makes the guard clause more readable, because the case names are now *Error* and *No Error* rather than *True* and *False*. Using the error wire as an input also triggers the color change of the case structure boundary. The structure is now green in the *No Error* case and red in the *Error* case.

Figure 28.15 illustrates how the parallel loops are shut down. If the user presses the STOP button, the event structure enqueues the *Shutdown* state at the front of the queue for immediate execution. The new value of the STOP button is used to shut down the producer loop, and the *Shutdown* state shuts down the consumer loop. When both loops have shut down, their error wires are merged and the queue is released. The error lines are merged first to ensure that both loops have stopped before the queue is released.

If an error occurs in the producer loop, the producer guard clause following the event structure responds to the error by enqueuing the *Shutdown* state at the front of the queue for immediate execution. This shuts down the consumer loop. The **`Enqueue Element At Opposite End`** function is placed on a parallel error wire because this case of the guard clause only executes if an error is present on the main error wire. The *Error* case of the producer loop guard clause sends a Boolean True to the producer loop conditional terminal, shutting down the producer loop.

If an error occurs in the consumer loop, the consumer loop guard clause bypasses the queue and immediately sends the consumer loop to the *Shutdown* state. This shuts down the consumer loop. The *Shutdown* state also uses the *Value(Signalling)* property node of the STOP button to trigger the event case of the producer loop event structure that handles the value change of the STOP button. This causes the producer loop to shut down.

28.3 The Producer–Consumer Car Wash Controller with User Events

The producer–consumer car wash controller also can be developed with user events instead of the *Value(Signalling)* property node, as seen in Figure 28.16. This version of the code follows the template laid out in Figure 27.9. The user event is created outside the loops in the input area of the state machine with the `Create User Event` function. The event is given a Boolean data type by wiring a Boolean constant to the data type input of this function. The Boolean constant is given the label *Error* to serve as a label for the user event.

The user event reference is then registered with the **`Register For Events`** function. The event registration refnum is passed to the dynamic input terminals of the event structure to allow the event structure to respond to the user event. Back in the

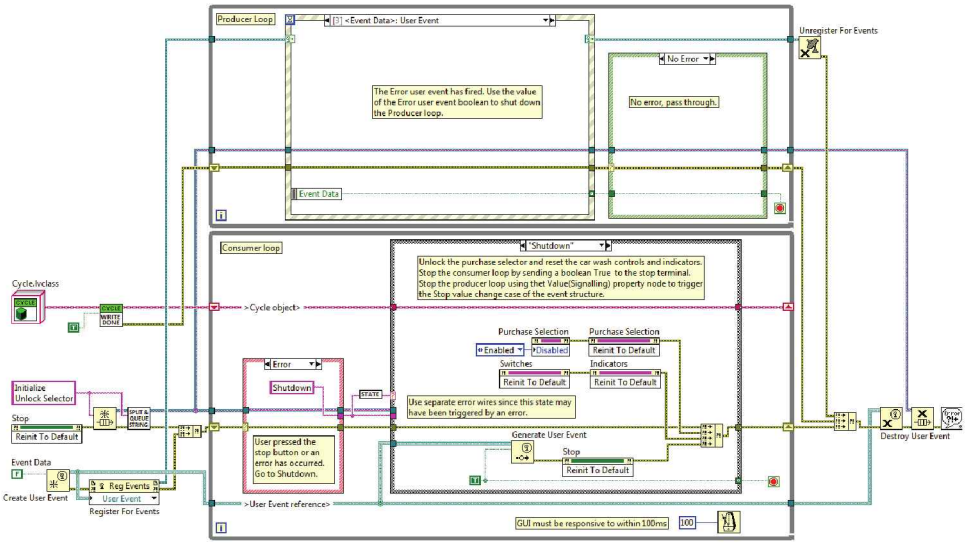


Figure 28.16: The producer–consumer (events) car wash controller with user events

consumer loop, the user event reference is passed as an input to the main case structure. When an error is detected in the consumer loop, the guard clause sends the consumer loop to the *Shutdown* state. In the *Shutdown* state, a Boolean True is wired to the consumer loop conditional terminal to shut down the consumer loop. The Boolean True is also wired to the **`Generate User Event`** function. This triggers the event case that handles the user event in the producer loop. The value of the *Error* user event is wired to the producer loop conditional terminal, shutting down the producer loop. The user event is unregistered by using the **`Unregister For Events`** function. The error wires from the producer and consumer loop are then merged, and the user event is destroyed using the **`Destroy User Event`** function.

28.4 Car Wash Race Condition

The producer–consumer car wash controllers shown in Figure 28.14 and Figure 28.16 have the potential for a race condition. Imagine that the car wash controller is in the middle of the *Rinse* state. The user presses the STOP button, expecting that the car wash controller will immediately shut down. The expected sequence is this:

1. The consumer loop dequeues a *Cycle:Rinse:5* state.
2. The consumer loop executes the *Cycle:Rinse:5* state.

3. The *Rinse* cycle is not yet done, so the *Cycle* state requeues the *Cycle:Rinse:5* state at the front of the queue.
4. The user presses the STOP button.
5. The event structure enqueues the *Shutdown* state at the front of the queue and shuts down the producer loop.
6. The consumer loop dequeues the *Shutdown* state and shuts down the consumer loop.

But the following sequence could happen instead:

1. The consumer loop dequeues a *Cycle:Rinse:5* state.
2. The user presses the STOP button.
3. The event structure enqueues the *Shutdown* state at the front of the queue and shuts down the producer loop.
4. The consumer loop executes the *Cycle:Rinse:5* state.
5. The *Rinse* cycle is not yet done, so the *Cycle* state requeues the *Cycle:Rinse:5* state at the front of the queue on top of the *Shutdown* state.
6. The consumer loop iterates and dequeues the *Cycle:Rinse:5* state once again.

The same situation could occur if a car broke down between stations of the car wash. Suppose a customer purchases a Deluxe Wash and suppose that their car breaks down between the underbody wash station and the main wash station. The car wash controller will continue to execute the *Cycle* state, waiting for the car to reach the main wash position sensor. If the operator shuts down the car wash to assist the stuck customer, the same race condition could occur.

The problem is that the *Cycle* state requeues the current state at the front of the queue if the cycle state has not finished. This dequeue–requeue sequence has the possibility of burying a *Shutdown* state placed at the front of the queue by the producer loop. This means that accessing the queue is a critical operation that must be protected. In the producer loop, placing the *Shutdown* state at the front of the queue is the critical operation. In the consumer loop, the dequeue–requeue functions—and everything between them—is the critical sequence of operations.

28.5 Semaphores and the Producer–Consumer Car Wash Controller

The race condition can be addressed by the use of semaphores. Semaphores are preferable to placing the critical code in a sub-VI in this instance for two reasons. First, the amount of critical code is large. The guard clause and the main case structure of the consumer loop would have to be placed in the sub-VI to protect the critical code. This would degrade code readability. Second, the critical code in the producer loop is different from the critical code of the consumer loop. This means that both the

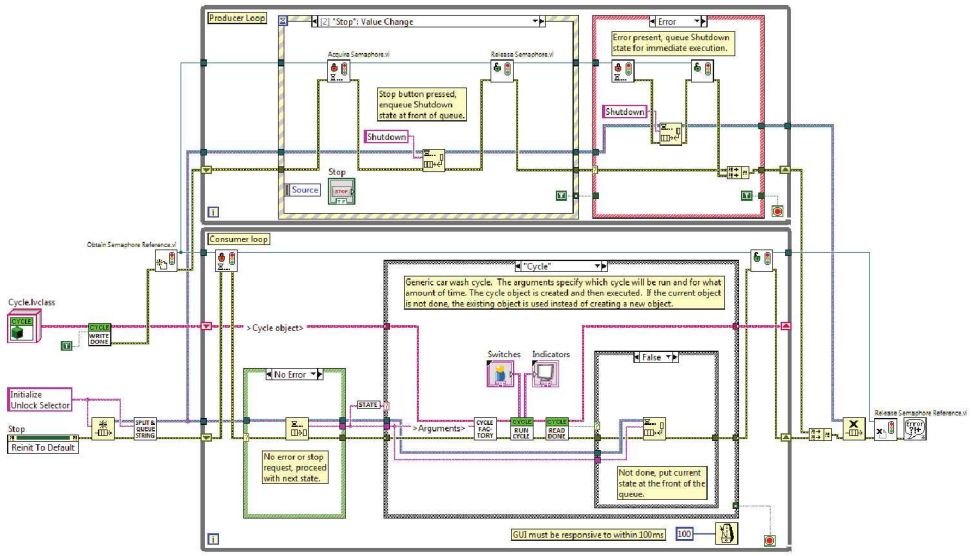


Figure 28.17: A producer–consumer (events) version of the car wash controller that uses semaphores

producer and the consumer code would have to be incorporated into the same sub-VI in order to gain the blocking that is needed to defeat the race condition. In this case, semaphores provide a cleaner and more readable solution.

Figure 28.17 shows an implementation of the producer–consumer car wash controller that uses semaphores to eliminate the race condition. The semaphore reference is obtained and sent to both the producer and consumer loops. The producer loop uses the **`Acquire Semaphore`** and **`Release Semaphore`** to protect the two operations that place the *Shutdown* state at the front of the queue: once in the event case that handles the value change of the STOP button and once in the *Error* case of the producer loop guard clause. Notice how the error wire connecting the semaphore functions is used to enforce the required dataflow: acquire semaphore–execute critical code–release semaphore.

In the consumer loop, the semaphores need to protect a much larger piece of code. The semaphore is acquired upstream of the consumer guard clause before the queue is dequeued. The semaphore is released after the main consumer case structure completes. This ensures that the requeue operation in the consumer loop is in the critical code sequence protected by the semaphores.

This version of the car wash controller successfully protects against race conditions while a Deluxe Wash or Economy Wash is in operation. The *Shutdown* state cannot be placed at the front of the queue during a *Cycle* state, because at that point, the consumer loop has acquired the semaphore. The *Shutdown* state only can be placed at the front of the queue just before the consumer loop dequeues the queue or just after the *Cycle* state requeues the current state. Either way, the next element dequeued from the queue will be the *Shutdown* state.

But the solution shown in Figure 28.17 will not work when the controller is waiting for a car wash to be purchased. Imagine that a car wash has just finished and the queue is empty. In this situation, the following series of events can occur:

1. The producer loop event structure is waiting for input so it does not acquire the semaphore.
2. The consumer loop acquires the semaphore and attempts to dequeue the currently empty queue.
3. The consumer loop waits for the producer loop to enqueue something.
4. The user presses the STOP button.
5. The event case that handles the value change of the STOP button attempts to acquire the semaphore but cannot because the consumer loop already has acquired it.

The producer loop is attempting to respond to a user shutdown request, so the producer loop has higher priority than the consumer loop which is simply waiting. Yet the producer loop cannot acquire the semaphore. The lower priority process has already acquired the semaphore and is blocking the higher priority process. This is an example of *priority inversion*, where a lower priority process blocks a higher priority process.

The situation is even worse than that. The consumer loop can't release the semaphore until the producer loop enqueues the *Shutdown* state. The producer loop can't enqueue the *Shutdown* state until the consumer loop releases the semaphore. This is an example of *deadlock*. The program waits forever because both the producer and the consumer loop are waiting for input from the other before they can proceed.

28.6 Breaking the Deadlock

A closer look at Figure 28.17 reveals the source of the deadlock. The producer loop is very focused with its use of semaphores. The semaphore is only acquired when the STOP button is pressed or when an error is detected in the producer loop. The semaphore is not acquired when the user selects a Deluxe Wash or an Economy Wash, because it is not needed in those cases.

The consumer loop is more profligate in its use of the semaphore. The semaphore is acquired at the beginning of every loop iteration whether it is needed or not. This is overkill because the semaphore only needs to be acquired when a *Cycle* state is active. If the consumer loop only acquired the semaphore during a *Cycle* state, the deadlock situation would be resolved.

This can be accomplished by developing a `Conditional Acquire Semaphore` sub-VI that checks the queue to see if the next state is a *Cycle* state

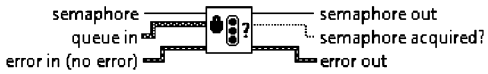


Figure 28.18: The icon of the `Conditional Acquire Semaphore` sub-VI

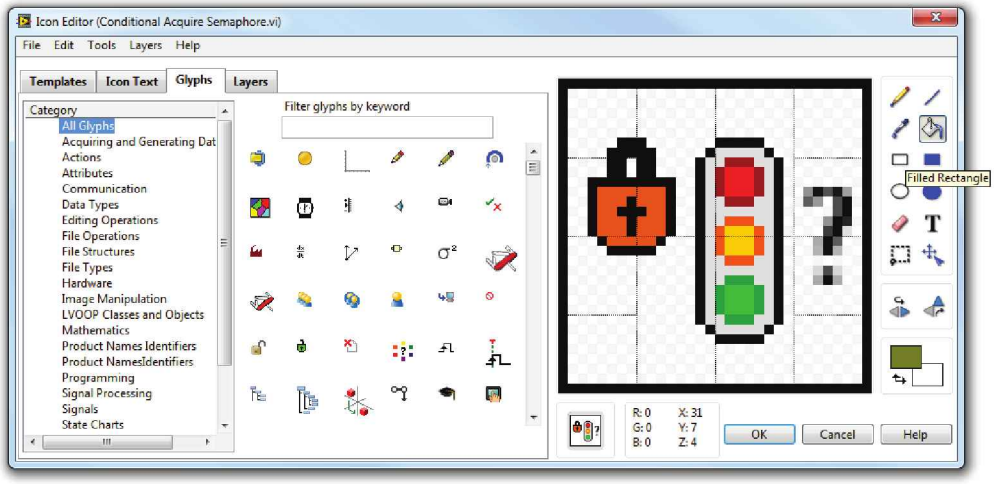


Figure 28.19: The glyphs menu of the icon editor

before acquiring the semaphore. The icon for this sub-VI is shown in Figure 28.18. The icon for this sub-VI was created using the glyph menu of the icon editor as seen in Figure 28.19. The glyphs menu contains a broad selection of pre-built glyphs for VI icons. Many of glyphs used by native LabVIEW function icons are included in this library, such as the glyphs of the lock and the semaphore symbol used in the `Conditional Acquire Semaphore` icon.

The block diagram of the `Conditional Acquire Semaphore` sub-VI is shown in Figure 28.20. This sub-VI uses the **`Preview Queue Element`** function to peek at the next element of the queue without dequeueing it (see the help file in Figure 28.21). The state is then extracted from the previewed queue element. If the state is equal to the *Cycle* state, an attempt is made to acquire the semaphore. If the previewed state is not the *Cycle* state, no attempt is made to acquire the semaphore. This resolves the deadlock situation. When the car wash controller is between car washes, the queue is empty and the consumer loop makes no attempt to acquire the semaphore. This leaves the semaphore available for acquisition when the user presses the STOP button, resolving the deadlock situation.

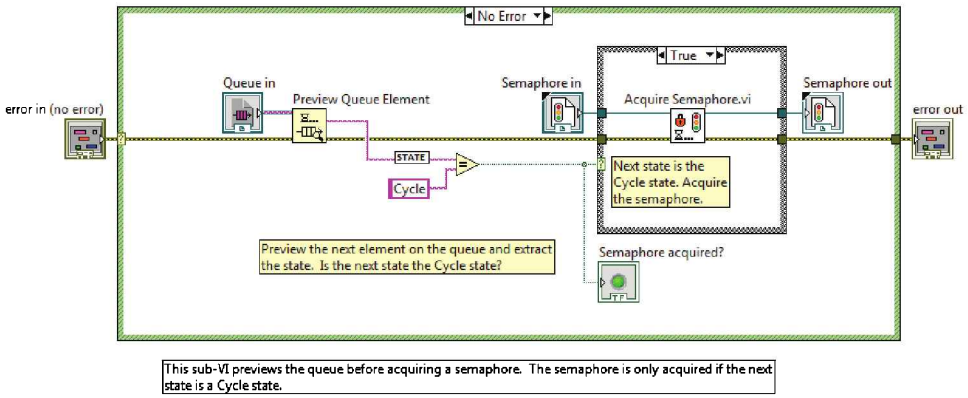


Figure 28.20: The block diagram of the `Conditional Acquire Semaphore` sub-VI

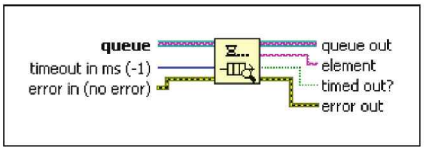
Preview Queue Element Function

Owning Palette: Queue Operations Functions

Requires: Base Package

Returns the element at the front of the queue without removing the element from the queue.

Details



Add to the block diagram Find on the palette

queue is a reference to a queue. Use the Obtain Queue function to obtain a queue reference.

timeout in ms specifies the time, in milliseconds, that the function waits for an element to become available in the queue if the queue is empty. The default is –1, indicating never to time out.

If the function waits **timeout in ms** and the queue remains empty, **timed out?** is TRUE.

error in describes error conditions that occur before this node runs. With the following exception, this input provides standard error in functionality.

This node runs normally *even if* an error occurred before this node runs.

queue out returns the reference to the queue unchanged.

element is the element at the front of the queue. This data type changes to match the subtype of **queue**.

timed out? is TRUE if an element did not become available in the queue before the function timed out or if an error occurred.

error out contains error information. This output provides standard error out functionality.

Preview Queue Element Details

Use the Dequeue Element function to return the element at the front of the queue and remove it from the queue. If the queue is empty, the function waits **timeout in ms** before timing out. If an element becomes available in the queue during the wait, the function returns the element and **timed out?** is FALSE. If **queue** becomes invalid (for example, the queue reference is released), the function stops waiting and returns error code 1122.

Figure 28.21: Help file for the `Preview Queue Element` function

An interesting feature of the `Preview Queue Element` is that it waits if the queue is empty and no timeout value has been wired. This is visually cued by the hourglass on its icon. If the queue was empty in the version of the car wash controller shown in Figure 28.14, the consumer loop would acquire the semaphore and then wait. The waiting would be due to the **`Dequeue Element`** function, which also shows an hourglass on its icon. In the new version of the car wash controller, the **`Dequeue Element`** function never needs to wait, because the **`Preview Queue Element`** function waits for the queue to be populated before deciding whether to acquire the semaphore.

The `Conditional Acquire Semaphore` sub-VI also outputs a Boolean that indicates whether the current state is a *Cycle* state or not. This is a signal to the `Conditional Release Semaphore` sub-VI with the icon shown in Figure 28.22. As the block diagram in Figure 28.23 shows, the value of the SEMAPHORE ACQUIRED? Boolean determines whether or not the semaphore is released. This is necessary to avoid an error from attempting to release a semaphore that has not been acquired.

The revised version of the producer–consumer car wash controller is shown in Figure 28.24. It is identical to the one shown in Figure 28.17—except for the use of the conditional acquire and release of the semaphores in the consumer loop. This version of the code avoids the deadlock problem.

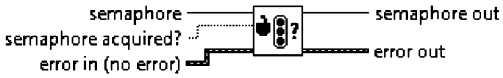


Figure 28.22: The icon of the `Conditional Release Semaphore` sub-VI

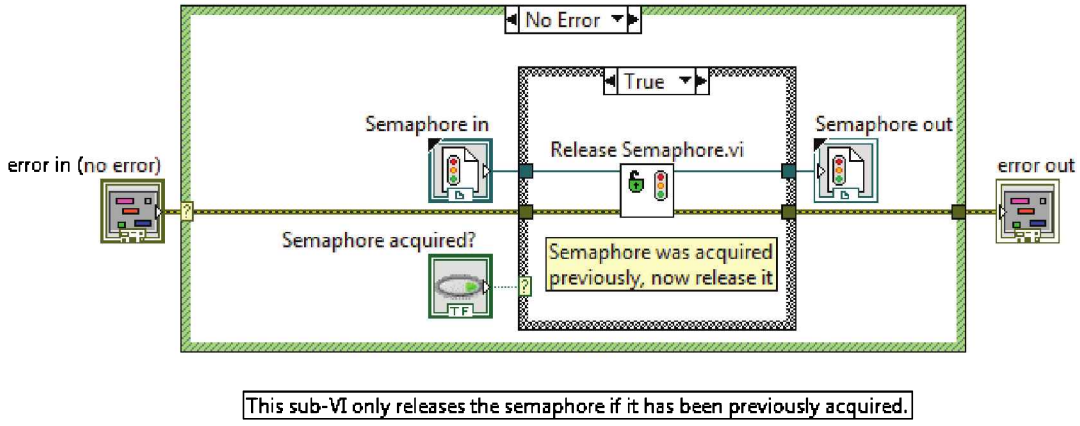


Figure 28.23: The block diagram of the `Conditional Release Semaphore` sub-VI

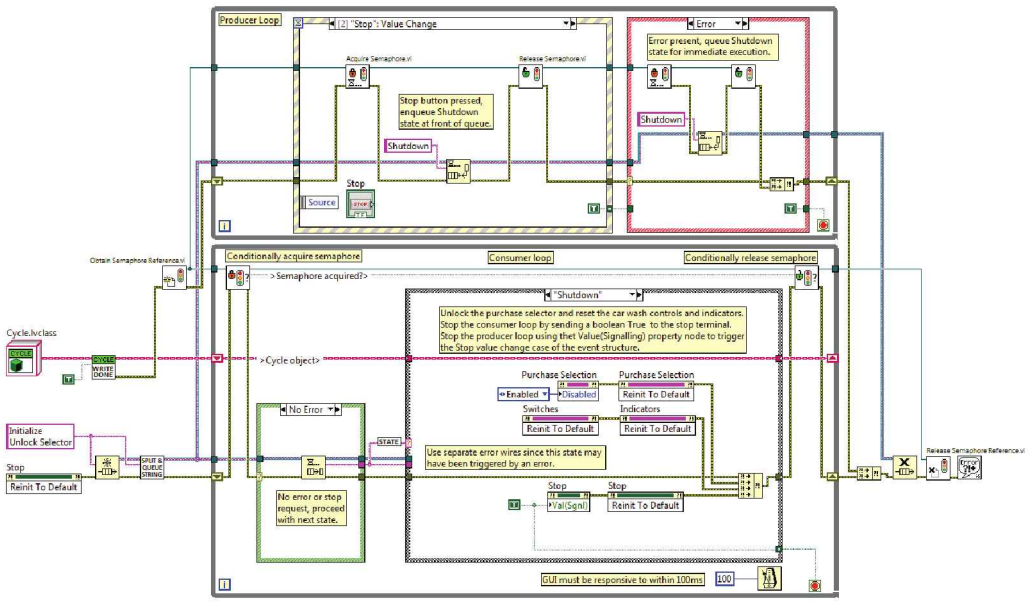


Figure 28.24: This version of the car wash controller avoids the deadlock issue

28.7 Living with the Race Condition

Another method of dealing with a race condition is to simply live with it and deal with the consequences. This isn't always possible. For example, in the parallel counter loops example shown in Figure 28.6, the race condition was very destructive. Information was lost when one loop accidentally overwrote the new value of the counter generated by the other loop. But the car wash controller doesn't use local variables, it uses queues. Information is not lost if the *Shutdown* state at the front of the queue gets covered by a *Cycle* state, it is simply misprioritized. In this case, we can plan for the race condition and deal with it accordingly.

One way to do this is to search the queue each iteration of the consumer loop to see if it contains a *Shutdown* state. There are only two possible scenarios where the queue contains a *Shutdown* state. The *Shutdown* state will either be the next state on the queue, or it will be the second state behind a *Cycle* state.

This check can be accomplished by developing a `Shutdown or Error` sub-VI that is similar to the `Stop or Error` sub-VIs of the previous chapters. The icon for this sub-VI is shown in Figure 28.25, and the block diagram is shown in Figure 28.26.



Figure 28.25: The icon of the `Shutdown or Error` sub-VI

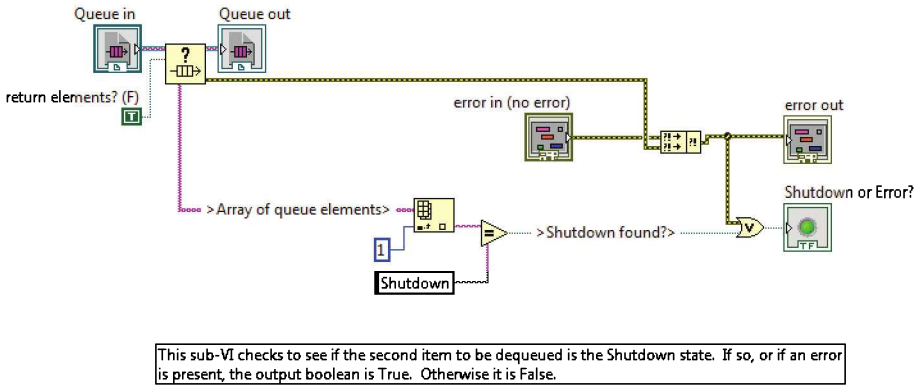


Figure 28.26: The block diagram of the `Shutdown or Error` sub-VI

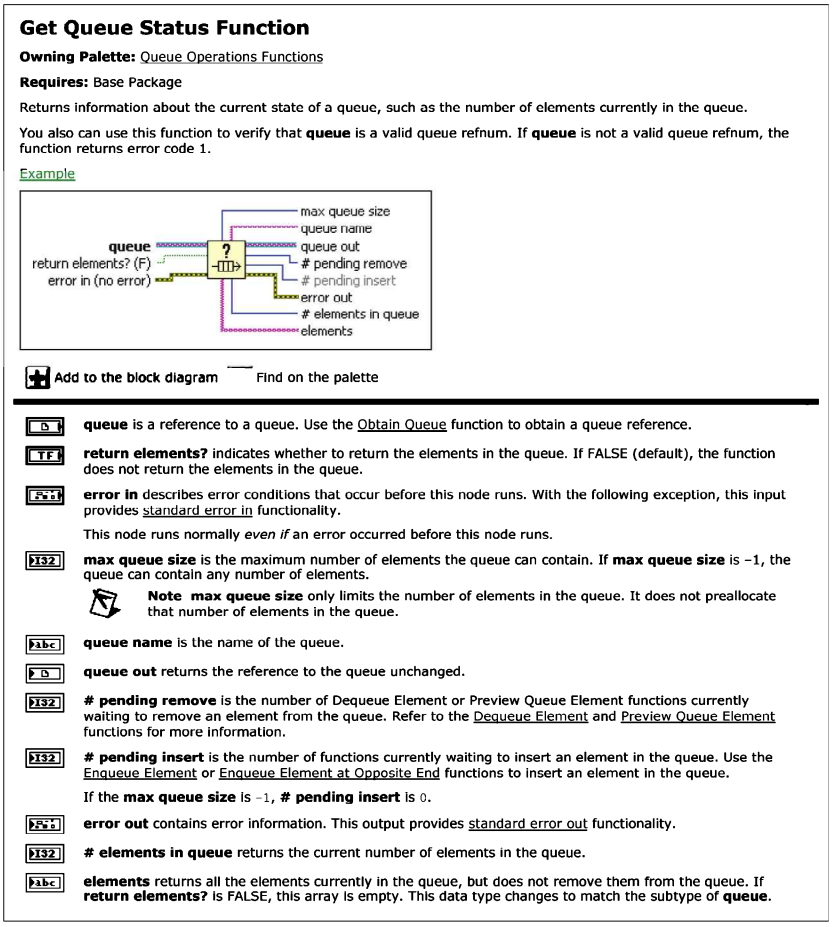


Figure 28.27: Help file for the `Get Queue Status` function

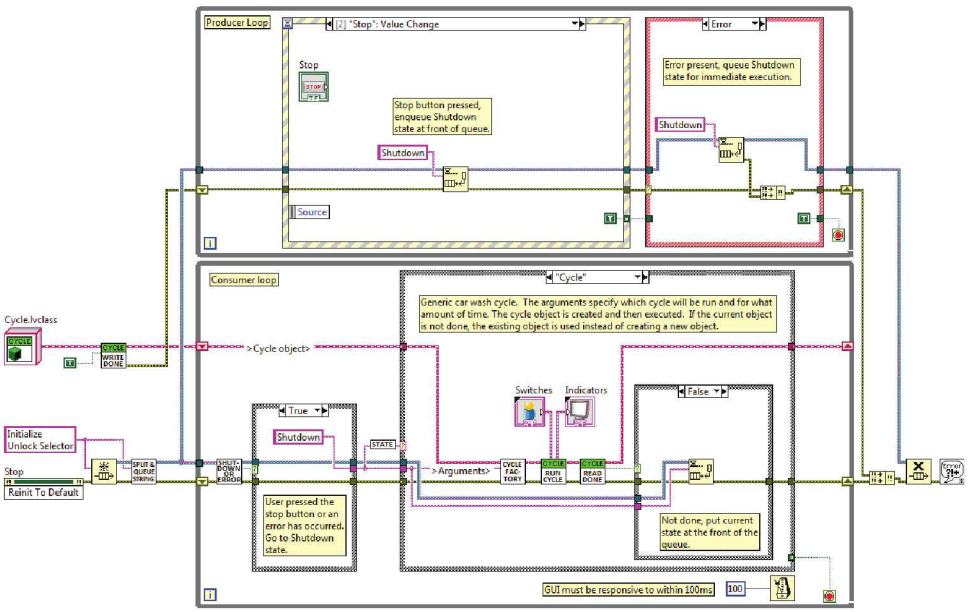
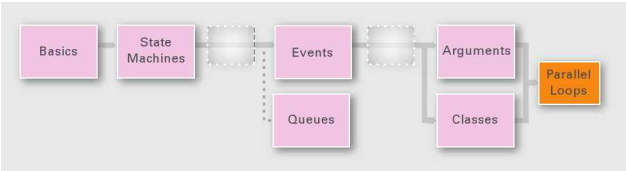


Figure 28.28: This version of the car wash controller allows the race condition to happen and corrects it

This sub-VI checks to see if the second item to be dequeued is equal to the *Shutdown* state or if an error is present.

This sub-VI uses the **`Get Queue Status`** function with the help file shown in Figure 28.27. This function can be configured to return an array of the current queue elements without dequeuing the elements. The first element of the array (index 0) is the element ready to be dequeued. This function does not wait and does not have an hourglass on its icon. If the queue is empty, an empty array is returned with the data type of the queue element. In the `Shutdown or Error` sub-VI, the second element of the array (index 1) is checked. If this element is equal to the *Shutdown* state (or if an error is present), the output Boolean is True. Otherwise, it is False.

This sub-VI can be used as an input to the consumer loop guard clause, as seen in Figure 28.28. If the output of this sub-VI is True, the queue is bypassed and the *Shutdown* state is sent to the consumer loop main case structure. If the output is False, the queue is dequeued normally. This is why the `Shutdown or Error` sub-VI only needs to check if the second item to be dequeued is the *Shutdown* state. If the first element to be dequeued is *Shutdown*, then no action needs to be taken. The consumer loop will go to the *Shutdown* state when the *Shutdown* is dequeued.



29

Extending the Language

In object-oriented programming, classes are often used to represent physical entities in the problem-space. In the car wash controller, the car wash cycles are timed so a `Timer` class was developed. This allowed us to take advantage of the data encapsulation of classes. The car wash itself is comprised of a series of car wash cycles, so generic `Cycle` class and its concrete children classes were developed. This allowed us to take advantage of inheritance and composition.

Modeling classes on physical entities is a very powerful technique because it helps align the underlying logic of the code with the underlying structure of the problem it is intended to solve. But this isn't the only way to use classes. Classes also can be used to extend the capabilities of a programming language.

A class consists of private data and methods to access and use the private data. In LabVIEW, the private data is stored in a class cluster control. This control becomes its own data type, and the methods become a library of functions that are built to take advantage of that data type. The class can be thought of as an extension of the native LabVIEW data types and functions.

In the previous chapter, we looked at race conditions and two ways to prevent them in the car wash controller: using semaphores and searching the queue for a buried *Shutdown* state. Both of these solutions treated the symptoms of the race condition without addressing the underlying cause—the dequeue–requeue pattern in the *Cycle* state. Both of the solutions worked, but treating symptoms rather than underlying causes can degrade scalability. Programs often change with time, because either the problems they are intended to solve change with time or the requirements change with time. This means that symptoms can change—even if the root cause of the problem does not. It is better to treat the root cause of the problem if possible to help future-proof the code.

The root cause of the car wash controller race condition lies in the dequeue–requeue pattern in the *Cycle* state. This implies that the problem is intimately connected to the use of queues. What if LabVIEW had a different type of queue that could be used to eliminate the race condition problem?

Specifically, what if LabVIEW had a queue with a built-in Boolean switch that could be used as a flag to activate or deactivate the requeuing process? This "switched

queue" would solve the race condition issue by preventing the race condition from happening in the first place. Unfortunately, LabVIEW does not provide queues with built-in Boolean switches. But such a queue can be created as a LabVIEW class, and the class then can be used in the car wash controller. The class "extends the language," providing a new capability to LabVIEW that has nothing to do with the physical problem space.

29.1 The `Switched Queue` Class

In this chapter, we create a `Switched Queue` class and implement it in the car wash controller. The private data of the `Switched Queue` class consists of a string-based queue reference and a Boolean control. The queue reference provides standard queue functionality, and the Boolean control provides a flag that can be set by the developer. The `Switched Queue` private data cluster is shown in Figure 29.1. This figure also shows the library of the method sub-VIs that will be developed in this chapter.

Wiring Tip: The string-based queue reference is not found on the LabVIEW function palette. In a separate sub-VI, set up a string-based queue using the `Obtain Queue` function. Create a control from the queue reference output, and save it as a custom control or type def by right-clicking on the control and selecting *Advanced... Customize*. Once the control is saved, it can be easily reused in any of the method sub-VIs.

Before developing these sub-VIs, we need to think about the way the `Switched Queue` class will be used. The intent is to have a switched queue behave like a regular queue, only with the added feature of a built-in Boolean switch that can be toggled on or off by the developer. A key feature of queues is that they are passed by-reference, not by-value. When a queue reference wire is split and is passed to the producer and consumer loops, both loops access the same queue. This violates LabVIEW's normal dataflow paradigm.

LabVIEW objects are passed by-value, not by-reference. That means that if a switched queue object is wired to parallel loops, the switched queue object will be copied and each loop will access an independent switched queue object. The two loops would not be able to communicate with each other via the switched queue, because each loop would have its own independent queue.

In order to share a switched queue between two loops, the switched queue needs to be passed by-reference instead of by-value. This allows the switched queue to be shared by both the producer and the consumer loops without establishing a data dependency between the two loops. This can be accomplished in LabVIEW using data value references.

A data value reference is a reference to an existing piece of data—much like a queue reference. When a data value reference is passed along a wire, the data itself is

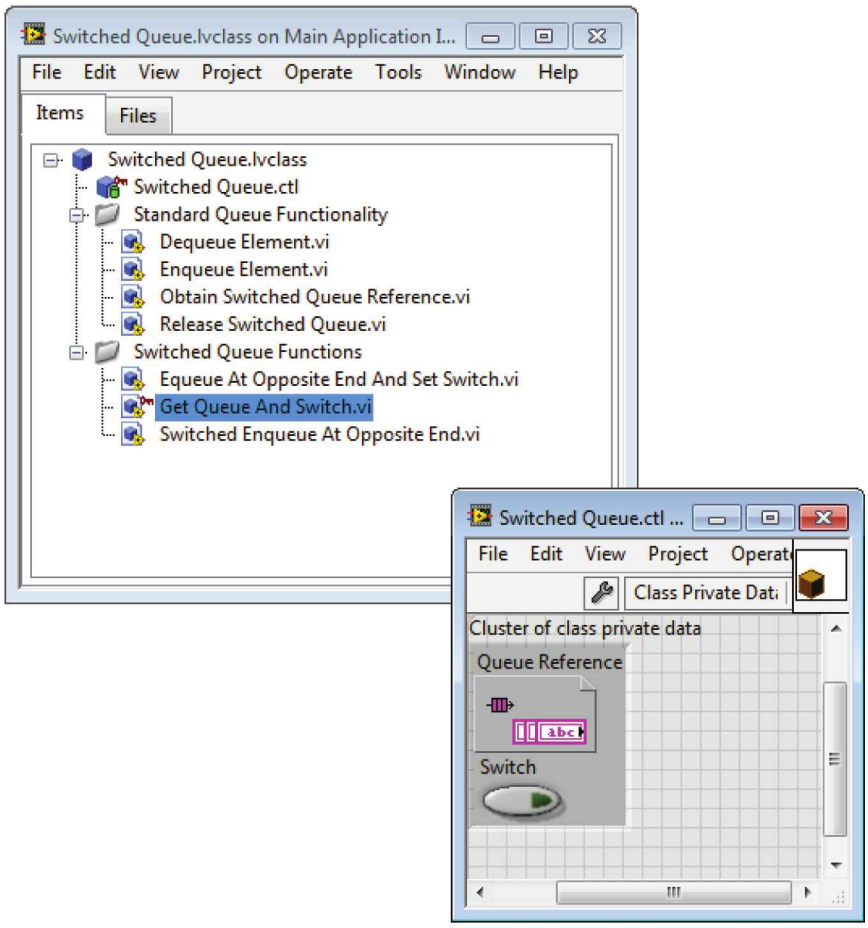


Figure 29.1: The `Switched Queue` class library and private data cluster

not on the wire. The data value reference can be thought of as the address of the data. If a data value reference wire is split, the address of the data is copied and is passed along both wires. The data itself is not copied. Both wires carry identical references to the same data.

Data that is referenced cannot be accessed directly. In order to access data that has been referenced, a three-step process is necessary.

1. De-reference the data. This process accepts the reference as an input and returns the referenced data as an output. This process should also lock the reference, blocking other attempts to access the referenced data.
2. Manipulate the data.
3. Re-reference the data. This process points the existing reference to the new value of the data. This process should also unlock the reference, allowing other portions of the program to access the referenced data.

References also need to be created and destroyed. Destruction of references is especially important. It is good programming practice to destroy all references created during the life of a program when the program closes. One situation to be especially wary of is creating references inside of a loop. Avoid this if at all possible. If it must be done, make sure that every reference created in the loop is destroyed before the program ends. Destroy the references inside the loop if possible.

29.2 Data Value References

There are two native LabVIEW functions and one native LabVIEW structure that are used to handle data references. Data value references are created with the **`New Data Value Reference`** function shown in Figure 29.2. This function accepts any LabVIEW data type as an input and provides a data value reference as an output. Existing data value references can be destroyed with the **`Delete Data Value Reference`** function shown in Figure 29.3. This function corresponds to the **`Release Queue`** function with one important difference: The **`Release Queue`** function operates normally if an error is present at its error input terminal. This ensures that the queue is released even if an error is present. The **`Delete Data Value Reference`** function does not have this property. If an error is present at its error input terminal

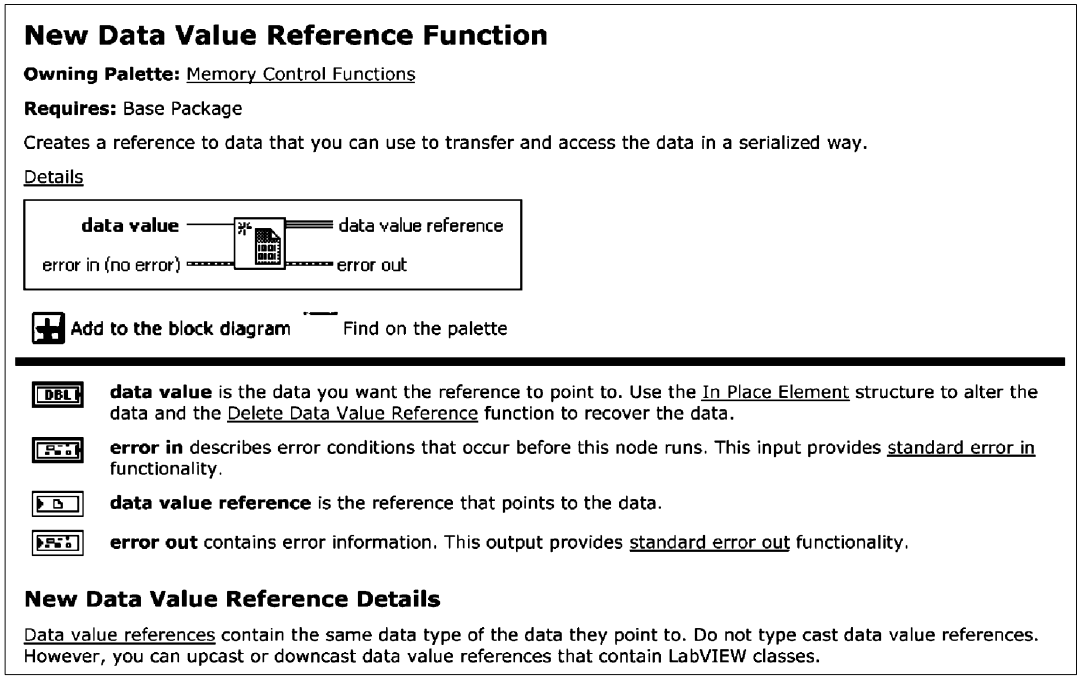
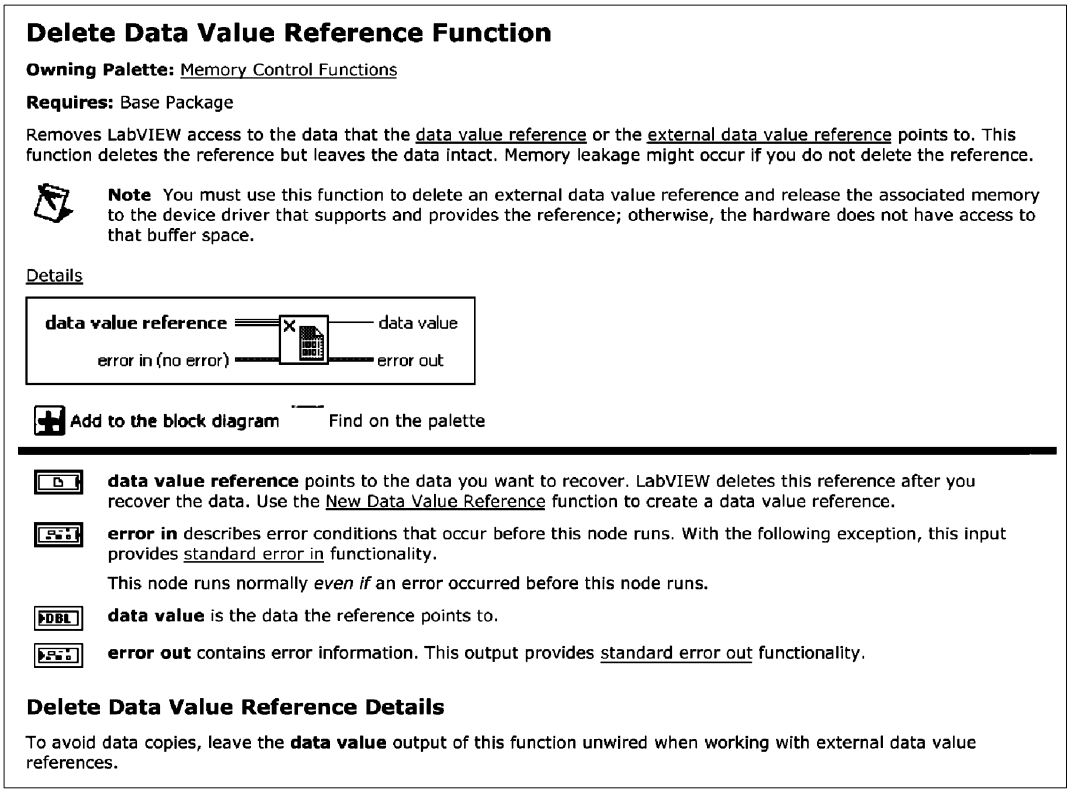


Figure 29.2: LabVIEW help file for the **`New Data Value Reference`** function



Delete Data Value Reference Function

Owning Palette: Memory Control Functions

Requires: Base Package

Removes LabVIEW access to the data that the data value reference or the external data value reference points to. This function deletes the reference but leaves the data intact. Memory leakage might occur if you do not delete the reference.

Note You must use this function to delete an external data value reference and release the associated memory to the device driver that supports and provides the reference; otherwise, the hardware does not have access to that buffer space.

Details

Add to the block diagram Find on the palette

data value reference points to the data you want to recover. LabVIEW deletes this reference after you recover the data. Use the New Data Value Reference function to create a data value reference.

error in describes error conditions that occur before this node runs. With the following exception, this input provides standard error in functionality.

This node runs normally *even if* an error occurred before this node runs.

data value is the data the reference points to.

error out contains error information. This output provides standard error out functionality.

Delete Data Value Reference Details

To avoid data copies, leave the **data value** output of this function unwired when working with external data value references.

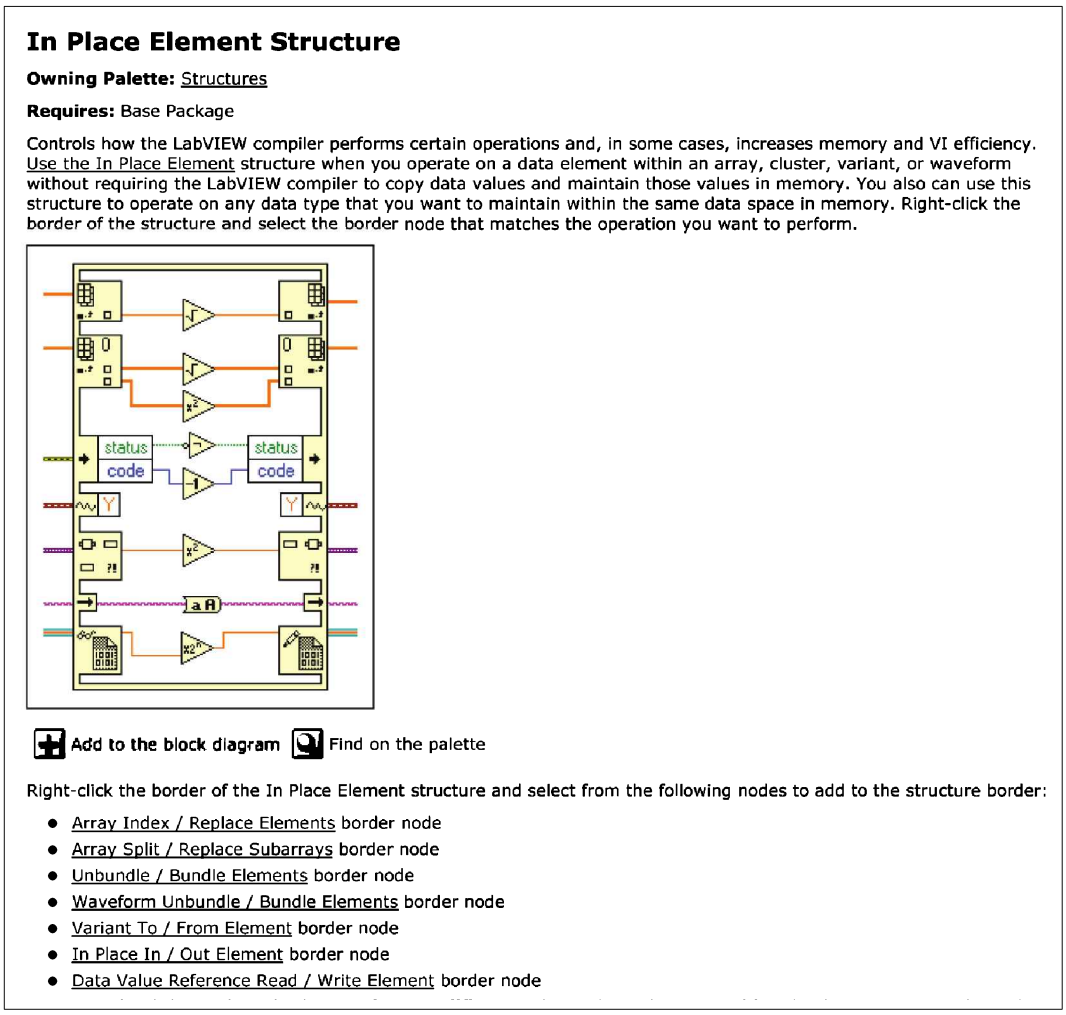
Figure 29.3: LabVIEW help file for the `Delete Data Value Reference` function

the function does not execute and simply passes the error to its error output terminal. This means that the **`Delete Data Value Reference`** function should be placed on its own parallel error wire to ensure that the data value reference is deleted. The error wire should then be merged with the main error wire of the state machine.

Referenced data must be accessed with the three-step process mentioned previously: de-reference the data, manipulate the data, re-reference the data. This is done using the in-place element structure. The in-place element structure is a versatile structure that allows operations to be done "in place" without creating extra data copies. The help file for this structure is shown in Figure 29.4.

For instance, suppose you have a large array of data and you wish to read a portion of the data, modify it, and overwrite the original values with the old values. Let's further suppose that the modification doesn't require manipulating any of the other elements of the array. This kind of operation can be done in place without disturbing or copying the rest of the array, improving the memory management of the operation.

An example is shown in Figure 29.5. In-place element structures can have various input and output elements. In this example, the top set of elements are the



In Place Element Structure

Owning Palette: Structures

Requires: Base Package

Controls how the LabVIEW compiler performs certain operations and, in some cases, increases memory and VI efficiency. Use the In Place Element structure when you operate on a data element within an array, cluster, variant, or waveform without requiring the LabVIEW compiler to copy data values and maintain those values in memory. You also can use this structure to operate on any data type that you want to maintain within the same data space in memory. Right-click the border of the structure and select the border node that matches the operation you want to perform.

Add to the block diagram Find on the palette

Right-click the border of the In Place Element structure and select from the following nodes to add to the structure border:

- Array Index / Replace Elements border node
- Array Split / Replace Subarrays border node
- Unbundle / Bundle Elements border node
- Waveform Unbundle / Bundle Elements border node
- Variant To / From Element border node
- In Place In / Out Element border node
- Data Value Reference Read / Write Element border node

Figure 29.4: LabVIEW help file for the `In-Place Element Structure`

Array Index/Replace elements. These element allow for in-place replacement of array elements. In the example, an array element is replaced with its square root without disturbing or copying the rest of the array. The lower set of elements are the *Unbundle/Bundle* elements. This example shows the Vehicle Out of Position Boolean being overwritten with its negation.

The in-place element structure also has a *Data Value Reference Read/Write* element. The *Read* element is used to de-reference a referenced piece of data. The data is then manipulated within the structure and is wired back to the *Write* element to re-reference the data. The in-place element structure also provides blocking. When a data value

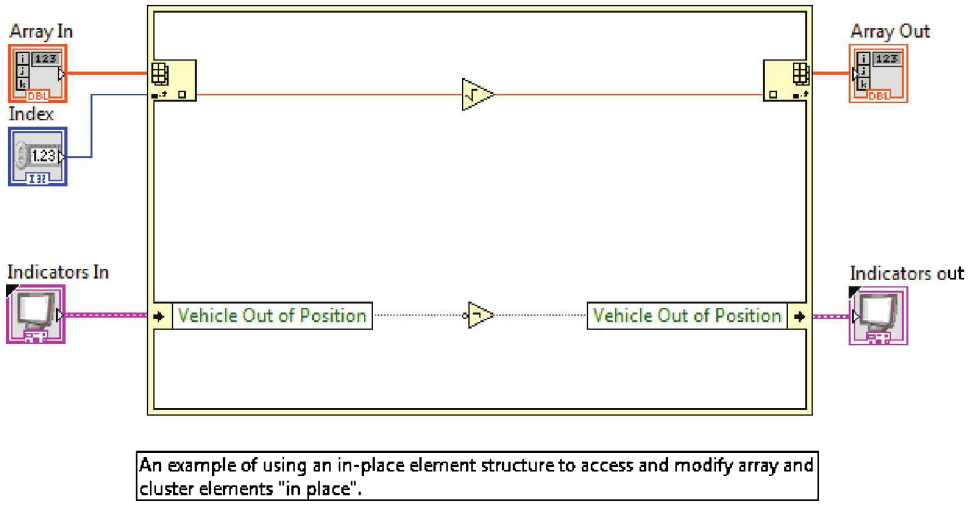


Figure 29.5: Modifying referenced data "in place" using an `In Place Element Structure`

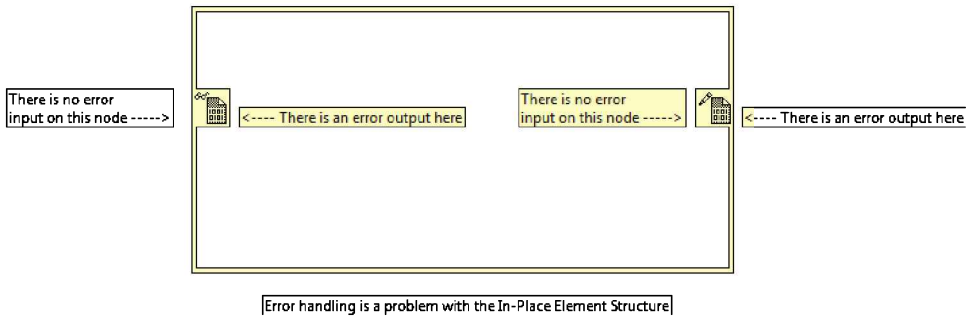


Figure 29.6: Error handling can be difficult using an `In Place Element Structure`

reference is read by an in-place element structure, the reference is locked. The reference cannot be read by other in-place element structures until the original in-place element structure finishes executing.

Error handling can be tricky when using an in-place element structure to manipulate referenced data. Figure 29.6 illustrates the problem. The *Read* element of a data value reference element pair has no input error terminal but it does have an output error terminal. The same is true for the *Write* terminal. One way to handle this is to add a set of *In Place In/Out* terminals for routing an error line and using error merging routines to trap all sources of error. A data value reference sub-VI template using this strategy is shown in Figure 29.7.

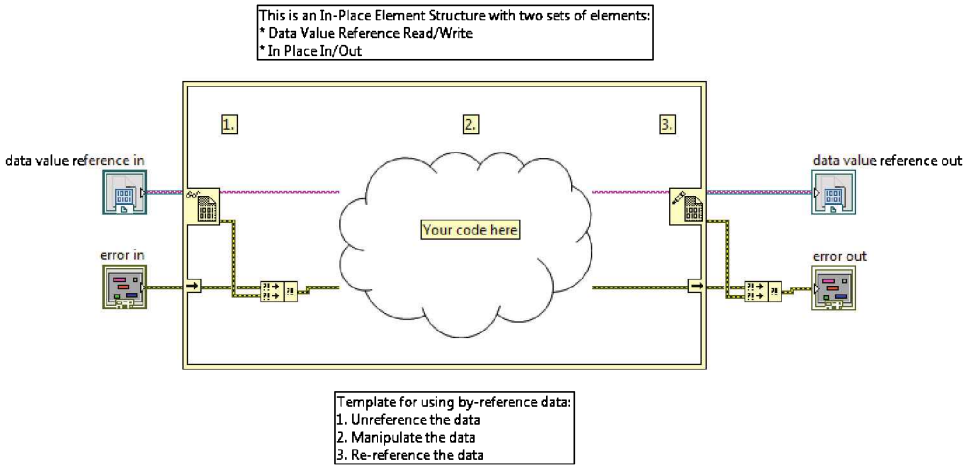


Figure 29.7: Template for accessing and manipulating data referenced by a data value reference

29.3 Data Accessor Methods of the `Switched Queue` Class

Armed with data value references, we can now develop the methods of the `Switched Queue` class. Since the switched queue is transmitted as a data value reference, the method VIs will follow the template shown in Figure 29.7. An in-place element structure will be used to de-reference the data, and the data will be re-referenced before the in-place element structure finishes its execution. Proper error handling also will be used to trap errors from all possible sources.

The `Get Queue And Switch` data access sub-VI takes the switched queue data value reference as an input and returns the queue reference and the value of the switch as outputs. The icon for this sub-VI is shown in Figure 29.8. The block diagram is shown in Figure 29.9. Notice that this sub-VI does not fit the automatically created data member access template that was created using the process shown in Figure 25.5. This data access method was created from scratch and manually added to the class library.

Wiring Tip: Once again, the data value reference control seen in Figure 29.9 is not a native LabVIEW control. In a separate sub-VI, set up a data value reference of a `Switched Queue` object using the `New Data Value Reference` function. Create a control from the data value reference output, and save it as a custom control or type definition by right-clicking on the control and selecting *Advanced... Customize*. Once the control is saved, it can be easily reused in any of the method sub-VIs.

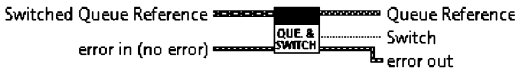


Figure 29.8: The icon of the `Get Queue And Switch` method sub-VI

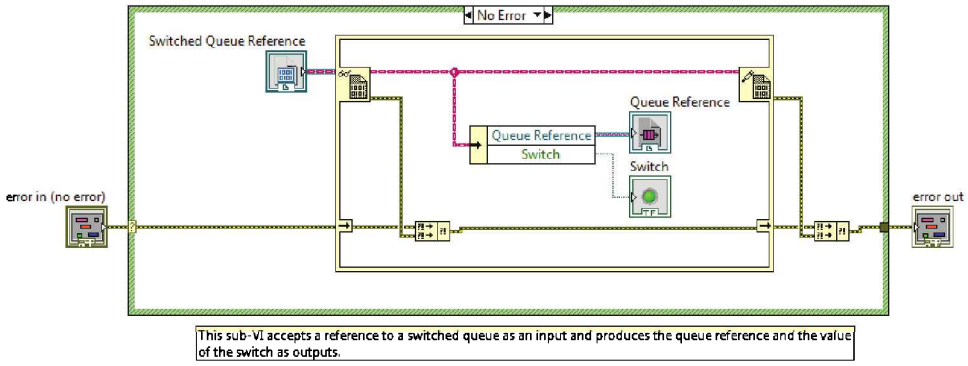


Figure 29.9: The block diagram of the `Get Queue And Switch` method sub-VI

29.4 `Switched Queue` Method Sub-VIs

The `Switched Queue` class is intended to mimic the look and feel of the standard LabVIEW queue functions, so we will use those functions as a guide in creating the `Switched Queue` method sub-VIs. First, the switched queue is created using an `Obtain Switched Queue Reference` method sub-VI. The icon for the `Obtain Switched Queue Reference` sub-VI is shown in Figure 29.10, and the block diagram is shown in Figure 29.11. This function instantiates a new instance of the `Switched Queue` class, initializes it, and then creates the data value reference. The `Switched Queue` object is initialized with a new string-based queue reference written to the private data queue reference. The value of the switch is determined by the input SWITCH Boolean control.

The opposite of the `Obtain Switched Queue Reference` function is the `Release Switched Queue` function with the icon shown in Figure 29.12. The block diagram is shown in Figure 29.13. The queue reference is obtained using the `Obtain Switched Queue Reference` method, and the queue is released using the native **`Release Queue`** function. The data value reference is then destroyed using the native **`Delete Data Value Reference`** function.

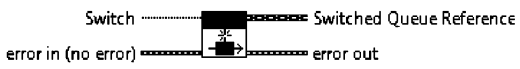


Figure 29.10: The icon of the `Obtain Switched Queue Reference` method sub-VI

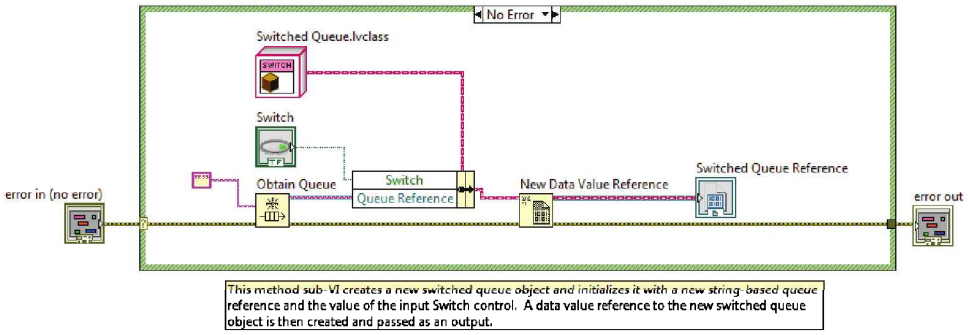


Figure 29.11: The block diagram of the `Obtain Switched Queue Reference` sub-VI

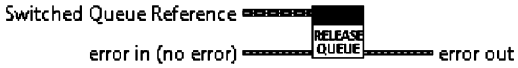


Figure 29.12: The icon of the `Release Switched Queue` method sub-VI

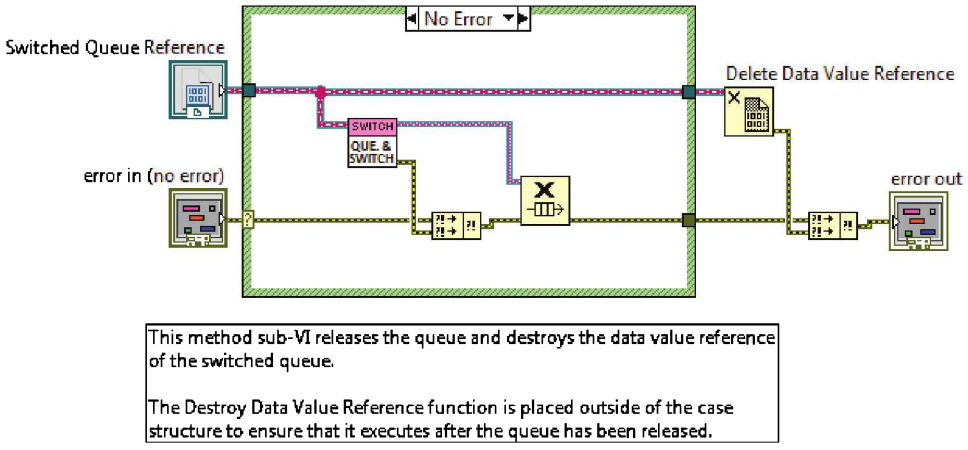


Figure 29.13: The block diagram of the `Release Switched Queue` sub-VI

Notice that the **`Delete Data Value Reference`** function is outside the case structure in Figure 29.13. This was done to avoid a race condition. If this function was placed inside the case structure, it would execute in parallel with the **`Release Queue`** function. This means that the reference could be destroyed before the queue is released, leading to an error due to attempting to release a non-existent queue. Wiring the **`Delete Data Value Reference`** function in-series with the case structure ensures that the code inside the case structure executes before the reference is deleted.

The `Dequeue Element` and `Enqueue Element` method sub-VIs are next. Care must be taken here, for our chosen form of the `Memory Queue` class has the

potential for deadlock if handled incorrectly. The danger lies in the fact that the in-place element structure blocks other attempts to de-reference the data while it is executing. Of course, this is exactly why we chose to use it, but it must be handled carefully with functions that can wait indefinitely.

An example of this issue is shown in Figure 29.14. The code in this figure attempts to enqueue an element on the switched queue and then dequeue it. The problem is that the native **`Dequeue Element`** function will wait forever if the queue has no elements, and this function has been placed in an in-place element structure. The code has no issue if the upper in-place element structure executes first. The switched queue reference is de-referenced, the queue reference is returned, and an element is placed on the queue. During this operation, the lower in-place element structure attempts to de-reference the switched queue reference but is blocked by the upper in-place element structure. When the upper in-place element structure finishes, the lower in-place element structure can act. It de-references the switched queue reference, gets the queue reference, and dequeues the element enqueued by the upper in-place element structure.

But if the lower in-place element structure executes first, deadlock results. The lower in-place element structure de-references the switched queue, gets the queue reference, and attempts to dequeue an element from the queue. Since the queue is empty, the native `Dequeue Element` function waits and the in-place element structure remains open. Meanwhile, the upper in-place element structure attempts to de-reference the switched queue to enqueue the element that the lower in-place element structure is waiting for. But it can't do this because it is blocked by the open lower in-place element structure. The bottom structure won't close until an element is enqueued,

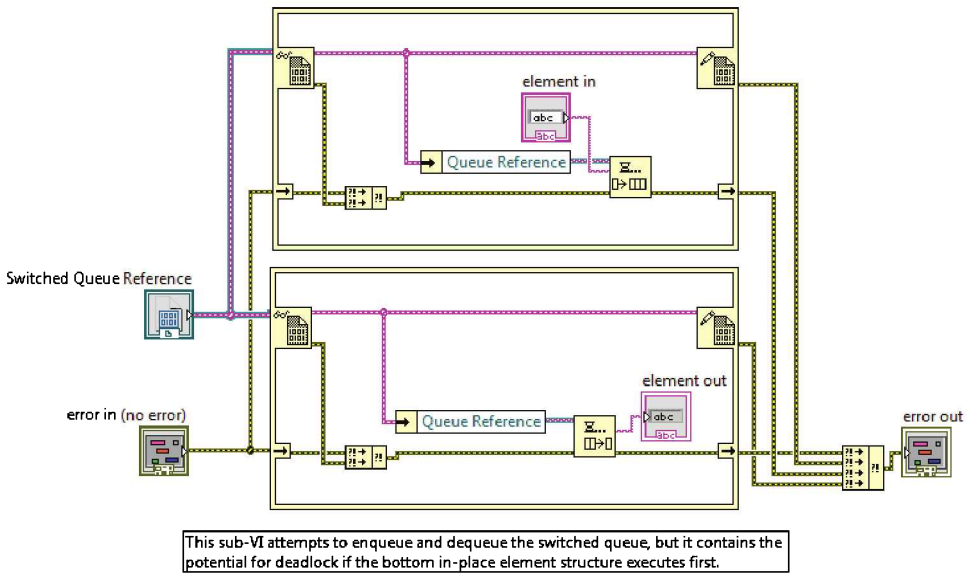


Figure 29.14: This code has the potential for deadlock

and the upper structure can't enqueue an element until the lower structure closes. This is an example of deadlock.

The deadlock is due to putting a waiting function inside a blocking in-place element. The problem here is with the `Dequeue Element` in the lower structure. Notice that the `Enqueue Element` in the upper structure also has an hourglass on its icon. This means that it can be made to wait, but this only happens if the queue was created with a fixed length and the queue is currently full. If a queue does not have a fixed length, the `Enqueue Element` function will not wait.

The solution to this problem is to take the queue functions out of the in-place element structures as shown in Figure 29.15. In this example, we use the `Get Queue And Read Switch` sub-VI shown in Figure 29.9 to get the queue reference, and then this reference is passed to the enqueue and dequeue functions. There is no deadlock issue in this code because the in-place element structures close after returning the queue reference. There is no waiting. The upper and lower path in-place element structures can still block each other, but only for the brief time that it takes to de-reference the switched queue and return the queue reference.

The lessons learned in Figure 29.15 can now be applied to the `Switched Queue` method sub-VIs for enqueuing and dequeuing data. The icon for the `Dequeue Element` sub-VI is shown in Figure 29.16 and the block diagram is shown in Figure 29.17. The `Get Queue And Switch` method sub-VI is used to get the queue reference and then the reference is used to dequeue an element outside of the in-place element structure. The same pattern is followed for the `Enqueue Element` method, whose icon is shown in Figure 29.18 and whose block diagram is shown in Figure 29.19.

As a convenience, we will also develop a sub-VI that accepts a multi-line string of states as an input. The `Split String and Enqueue` method sub-VI splits the

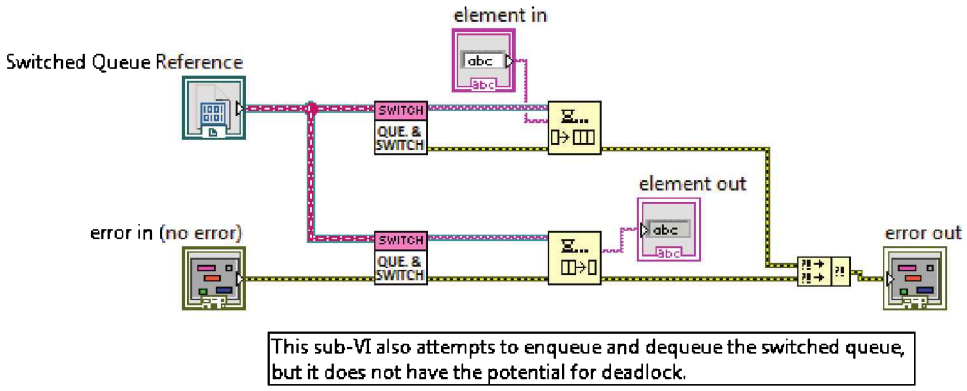


Figure 29.15: This code has no potential for deadlock



Figure 29.16: The icon of the `Dequeue Element` method sub-VI

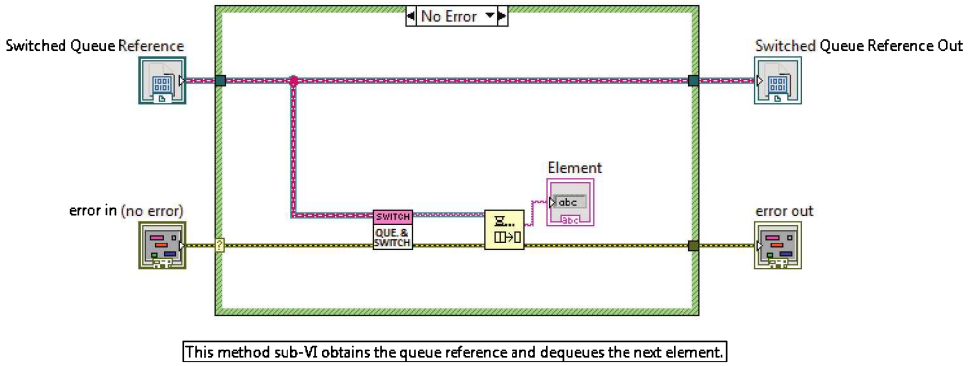


Figure 29.17: The block diagram of the `Dequeue Element` method sub-VI

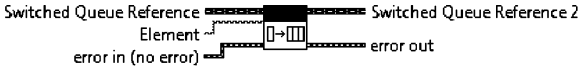


Figure 29.18: The icon of the `Enqueue Element` method sub-VI

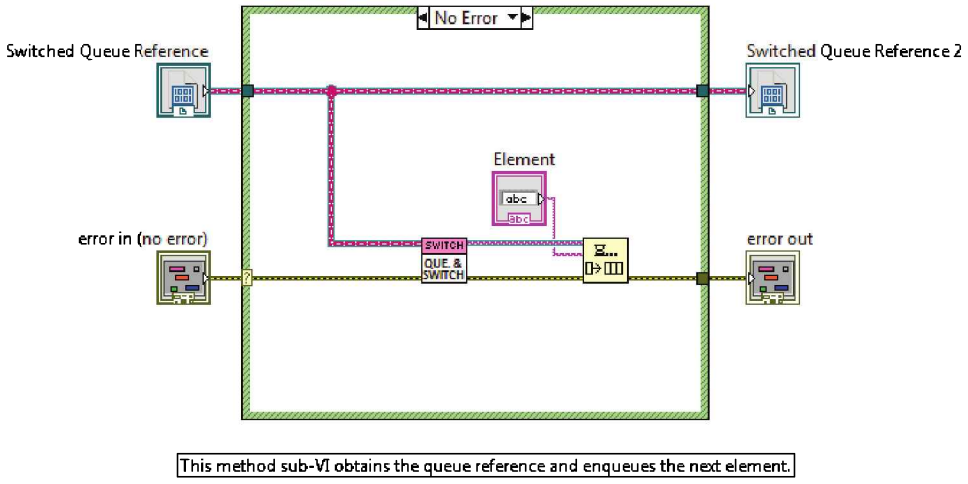


Figure 29.19: The block diagram of the `Enqueue Element` method sub-VI

string into individual elements and enqueues them. This can be done using the `String to Array` function shown in Figure 20.26. The icon for the `Split String and Enqueue` is shown in Figure 29.20, and the block diagram is shown in Figure 29.21. The multi-line string is converted to an array of strings, and this array is fed to the `Enqueue Element` method sub-VI one element at a time using a **`For`** loop with auto-indexing.

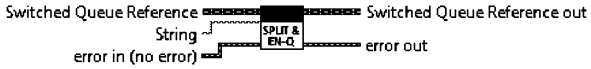


Figure 29.20: The icon of the `Split String and Enqueue` sub-VI

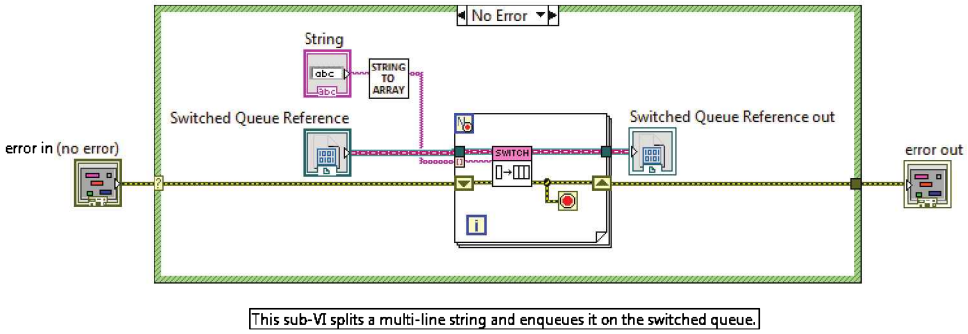


Figure 29.21: The block diagram of the `Split String and Enqueue` sub-VI

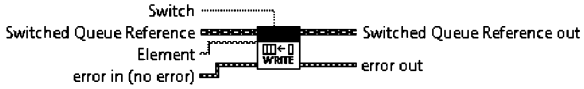


Figure 29.22: The icon of the `Enqueue At Opposite End And Set Switch` method sub-VI

This brings us to the methods that use the value of the switch in the switched queue. The `Enqueue At Opposite End And Set Switch` method uses the blocking feature of the in-place element structure to enqueue an element at the front of the queue and set the switch at the same time. "At the same time" means that the two operations are done in parallel while all other attempts at accessing the switched queue are blocked.

The icon of the `Enqueue At Opposite End And Set Switch` method is shown in Figure 29.22 and the block diagram is shown in Figure 29.23. The switched queue reference is dereferenced, blocking all other attempts to access the switched queue. Inside the in-place element structure, the queue reference is generated and the desired element is placed at the front of the queue. The switch is also set in parallel with this operation. The switched queue is then re-referenced, restoring access to the switched queue. The fact that the element is enqueued and the switch is set while access to the switched queue is blocked helps prevent a race condition. The in-place element structure guarantees that the switch is set when that particular element is enqueued. Notice too that the `Enqueue At Opposite End And Set Switch` method will execute normally even when an error is present at the error input terminal. The **`Enqueue Element At Opposite End`** function

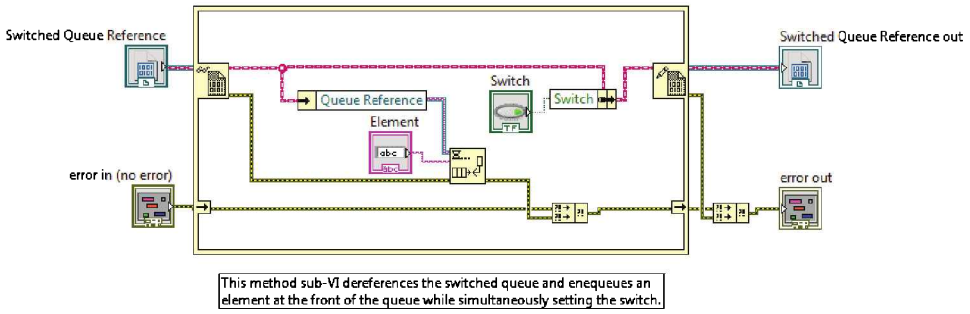


Figure 29.23: The block diagram of the `Enqueue At Opposite End And Set Switch` method sub-VI



Figure 29.24: The icon of the `Switched Enqueue At Opposite End` method sub-VI

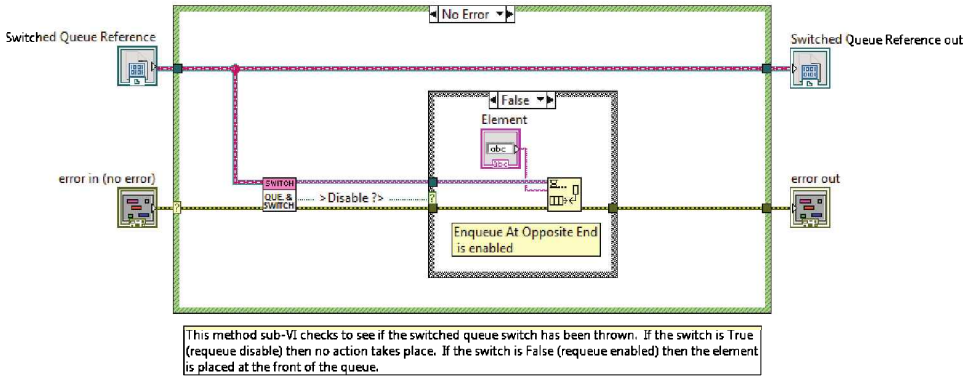


Figure 29.25: The block diagram of the `Switched Enqueue At Opposite End` method sub-VI, requeue enabled

is placed on a parallel error wire and so it will execute even in the presence of an error.

The final method is the `Switched Enqueue At Opposite End` sub-VI. The icon for this sub-VI is shown in Figure 29.24. The block diagram is shown in Figure 29.25. This sub-VI checks the value of the switch before enqueuing an element at the front of the switched queue. If the switch is False (requeue enabled), then the element is placed at the front of the queue. If the switch is True (requeue disabled), then the element is not enqueued. This is shown in Figure 29.26.

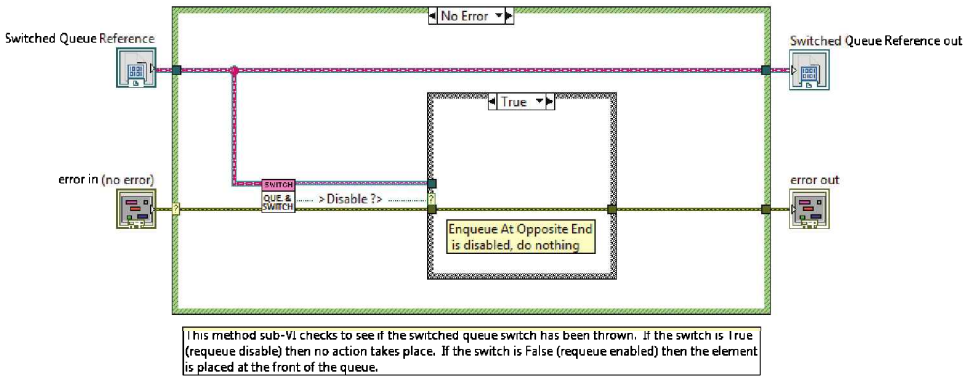


Figure 29.26: The block diagram of the `Switched Enqueue At Opposite End` method sub-VI, requeue disabled

29.5 Using the Switched Queue

Now that the general-purpose method sub-VIs have been developed, we can use the switched queue to combat the car wash controller race condition. The use of the switched queue is illustrated in Figure 29.27. Imagine that the car wash controller is in the middle of executing a car wash cycle. If the user presses the STOP button or if an error is detected in the producer loop, the `Enqueue At Opposite End And Set Switch` method sub-VI is used to simultaneously place the *Shutdown* state at the front of the queue and to set the queue switch to True. Because these events take place within the same in-place element structure, they cannot be interrupted by the consumer loop dequeuing or requeuing the switched queue. Recall that the `Dequeue Element` sub-VI and the `Switched Enqueue At Opposite End` sub-VI in the consumer loop need to dereference the switched queue reference in order to access the reference of the main queue. Since the in-place element structure blocks this operation, the producer loop cannot be interrupted as it enqueues the *Shutdown* state and sets the queue switch. Notice that the `Enqueue At Opposite End And Set Switch` method sub-VIs do not need to be placed on parallel error wires, since they were constructed to execute normally even if an error is present at the error input terminal.

If the consumer loop then tries to requeue the current state at the front of the queue, covering the *Shutdown* state, it will be prevented from doing so by the queue switch. The `Switched Enqueue At Opposite End` sub-VI checks the value of the queue switch before enqueuing the new element at the front of the queue. Since the producer loop set the switch to True, the requeue operation is disabled and the current cycle is not placed at the front of the queue. When the consumer loop executes again, the *Shutdown* state will be dequeued normally and the state machine will shut down. By disabling the requeue operation when the *Shutdown* state is enqueued, the switched queue defeats the potential race condition.

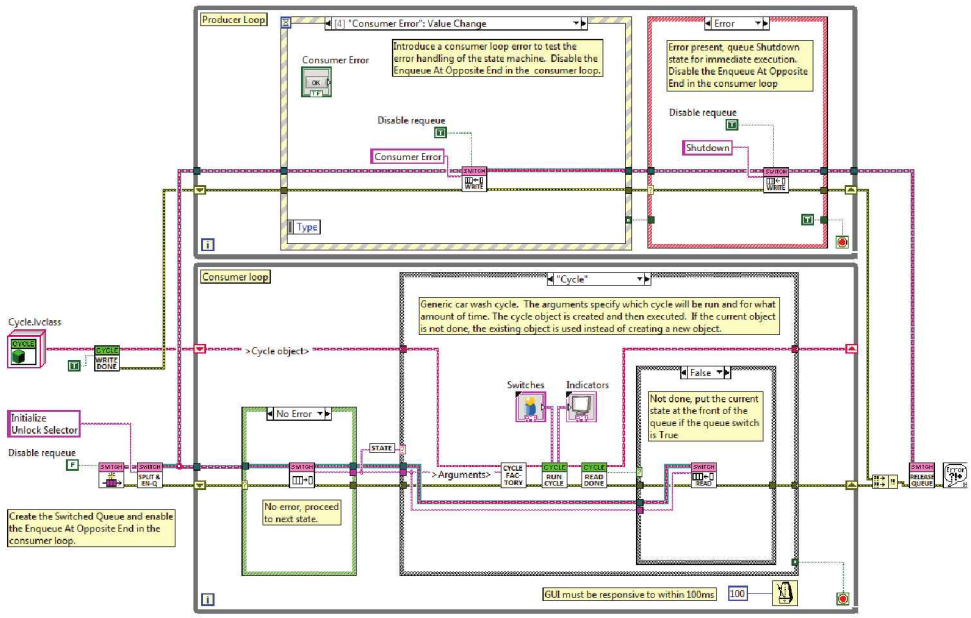


Figure 29.27: The block diagram of the `Producer Consumer Controller Switched Queue` controller

29.6 The Pros and Cons

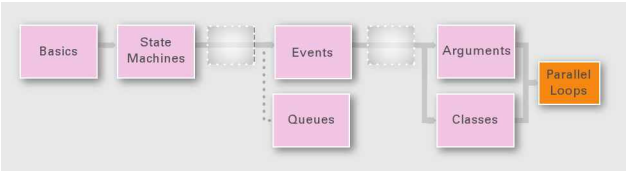
As you can see, using classes to extend the language with new capabilities requires planning and work. In the example in this chapter we had opportunities for creating new race conditions and even deadlocks in our attempt to prevent the original car wash race condition. Was it worth the effort?

Here are some of the benefits to weigh against the costs. First, the `Switched Queue` class is reusable. The effort put into developing this class will likely be leveraged in future projects since the dequeue–requeue technique is fairly common. Second, the `Switched Queue` class is very flexible. In this chapter, we used the queue switch to disable the requeue operation to remove the car wash race condition. But the switch could be used for other purposes in other applications. The switch is a general-purpose addition to the string-based queue that can be used in a number of different ways, making the `Switched Queue` class very reusable.

Most importantly, the new class increases the readability, maintainability, and scalability of the car wash controller. The switched queue solution is not nearly as cluttered (or as ugly) as the semaphore solution. The code maintainability is improved because the class is self-contained. The method VIs can be created, debugged, and tested apart from the car wash controller itself. Scalability is also improved because new functionality can be added to the switched queue class in the future with minimal

impact on the car wash controller itself. It is also easy to extend the protection of the *Shutdown* state to other states that might be added in the future. For example, suppose that a new state called *Error* is added in the future. If the user presses the STOP button, the *Shutdown* state is still put at the front of the queue, but if the producer loop detects an error, the *Error* state would be put in front of the queue. In this scenario, the *Error* state is also in danger of being covered up by a requeued *Cycle* state if the race condition manifests itself. Extending the protection offered by the switched queue to the *Error* state would be a simple matter. The requeue operation easily can be disabled when enqueuing an *Error* state—just as it was disabled for the *Shutdown* state.

In the end, the developer must decide if developing a new class is worth the effort. The scale of the project, the likelihood of code reuse, and the emphasis placed on readability, scalability, and maintainability of the code that would use the class must be considered.



30

Problem Set 11

30.1 Security System Controller Exercises

Exercise 11.1: Demonstrating the Need for Parallel Architectures

In the following exercises, you will develop a producer–consumer state machine. Of all the projects in this book, the only one that really cries out for a parallel architecture is the security system controller. The producer loop can be used to constantly monitor the switches and the consumer loop can be used to respond to incidents. In this way, the system can continue to monitor for security breaches while it is updating the log and doing whatever else is necessary to respond to previous security breaches.

Why would the security system controller benefit from a parallel loop architecture? Imagine that in the future a new requirement is added to the functional specification. Whenever a zone changes status an email needs to be sent to a supervisor, or perhaps a diagnostic test on the affected zone must be run. These activities will likely be time consuming, certainly longer than the 100ms loop period we have been using in this book.

To simulate this, create the `Update Log Slowly` sub-VI with the icon shown in Figure 30.1. This sub-VI updates the security log slowly by adding a **`Wait (ms)`** function to the block diagram (see the help file in Figure 30.2). Wire a five second (5000 millisecond) wait to this function to force the `Update Log Slowly` sub-VI to update the log file slowly. Substitute this sub-VI for the `Update Log` sub-VI in your single-loop implementations of the security system controller and run the program. What happens? What do you think would happen if you tried this test on a producer–consumer version of the security system controller?

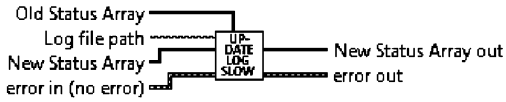
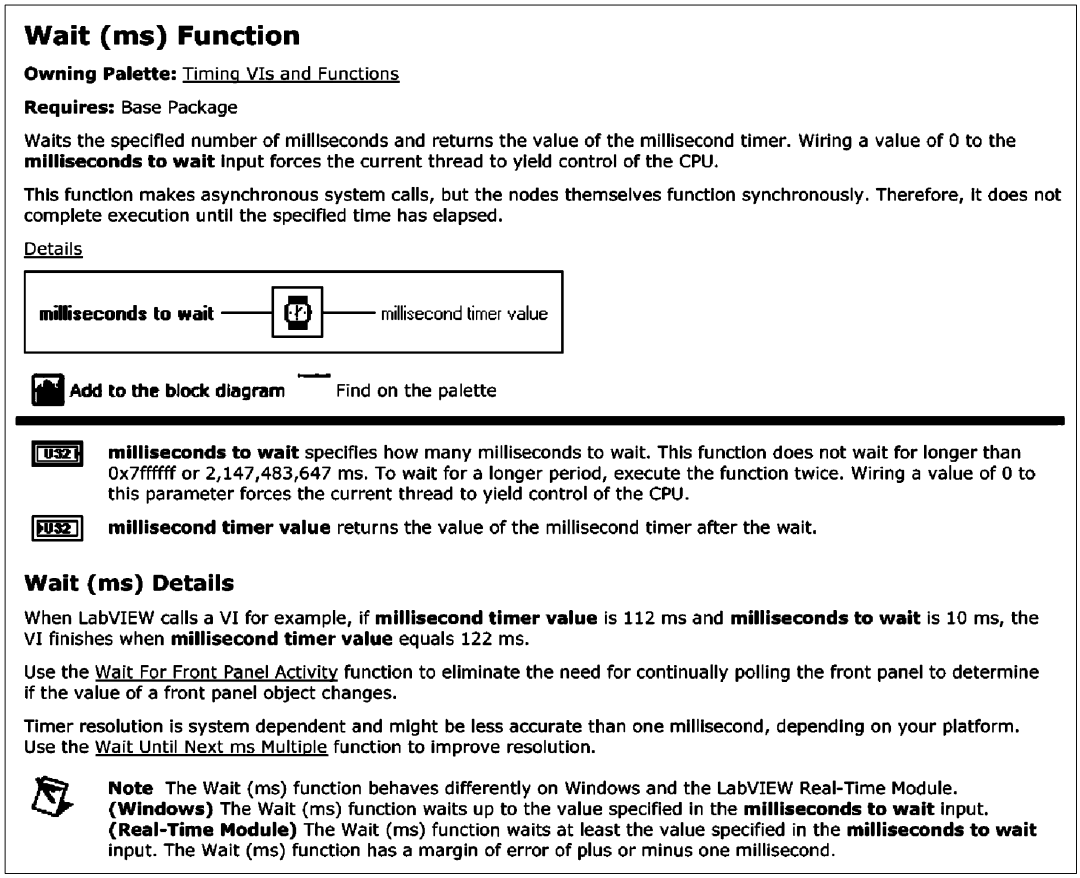


Figure 30.1: The icon of the `Update Log Slowly` sub-VI



Wait (ms) Function

Owning Palette: Timing VIs and Functions

Requires: Base Package

Waits the specified number of milliseconds and returns the value of the millisecond timer. Wiring a value of 0 to the **milliseconds to wait** input forces the current thread to yield control of the CPU.

This function makes asynchronous system calls, but the nodes themselves function synchronously. Therefore, it does not complete execution until the specified time has elapsed.

Details

Add to the block diagram Find on the palette

U32 **milliseconds to wait** specifies how many milliseconds to wait. This function does not wait for longer than 0x7fffffff or 2,147,483,647 ms. To wait for a longer period, execute the function twice. Wiring a value of 0 to this parameter forces the current thread to yield control of the CPU.

U32 **millisecond timer value** returns the value of the millisecond timer after the wait.

Wait (ms) Details

When LabVIEW calls a VI for example, if **millisecond timer value** is 112 ms and **milliseconds to wait** is 10 ms, the VI finishes when **millisecond timer value** equals 122 ms.

Use the Wait For Front Panel Activity function to eliminate the need for continually polling the front panel to determine if the value of a front panel object changes.

Timer resolution is system dependent and might be less accurate than one millisecond, depending on your platform. Use the Wait Until Next ms Multiple function to improve resolution.

Note The Wait (ms) function behaves differently on Windows and the LabVIEW Real-Time Module. **(Windows)** The Wait (ms) function waits up to the value specified in the **milliseconds to wait** input. **(Real-Time Module)** The Wait (ms) function waits at least the value specified in the **milliseconds to wait** input. The Wait (ms) function has a margin of error of plus or minus one millisecond.

Figure 30.2: Help file for the `Wait (ms)` function ■

Exercise 11.2: Producer–Consumer Security System

Revise the security system controller with arguments to use a producer–consumer architecture. Monitor the switches and calculate the new zone status in the producer loop and pass the new zone status to the consumer loop as an argument. Make sure that the system will shut down both the producer and consumer loops if an error is detected in either loop. If an error is detected in the consumer loop, use a VALUE(SIGNALLING) property node of the STOP SIMULATION button to shut down the producer loop.

Following the example in Chapter 27, add PRODUCER ERROR and CONSUMER ERROR buttons to the front panel of the security system controller to allow the user to intentionally generate errors in the producer and consumer loops. Use these buttons to test the final VI.

Be mindful of the following questions and issues:

- Where should the `Set Zone Colors` sub-VI be placed, in the producer loop or in the consumer loop?

- When you simulate an error in either the producer or consumer loops, does the VI shut down gracefully? Is the correct error reported?
- Does the consumer loop timing need to be regulated with the **`Wait Until Next ms Multiple`** function?
- Be mindful of startup and shutdown conditions. At startup, you should preallocate the zone status array and ensure that the zone indicator lamps are all set to green. At shutdown, the zone indicators should be reset to green.
- The front panel switches should be reinitialized at the beginning of the program and as it exits. All references should be closed and the queue released before the program exits.
- Make sure the program is scalable. If the number of zones is changed on the front panel, the block diagram should handle the change without being modified. ■

Exercise 11.3: Testing the Parallelism of the Producer–Consumer Security System

To test the parallelism of the producer–consumer security system developed in Exercise 11.2, replace the `Update Log` sub-VI in the consumer loop with the `Update Log Slowly` sub-VI developed in Exercise 11.1. How does the performance of the parallel loop security system compare the performance of the single loop security system? You should notice a discernable time lag after you press the STOP button before the program ends. What causes this lag? ■

Exercise 11.4: Merging the Producer and Consumer Loop Errors

Why is it important to merge the producer and consumer loop error wires before releasing the queue? ■

Exercise 11.5: Moving Toward a String-Based Producer–Consumer Security System Controller

The producer–consumer version of the security system controller developed in Exercise 11.2 is functional, but the block diagram can be improved. The cluster queue elements are large and awkward. The zone status argument appears every time the queue is accessed whether the zone status array is needed or not.

The block diagram would be easier to code and easier to read if a string-based queue was used instead of a cluster-based queue. Developing a string-based version of the security system controller is an easier task than you might expect, since the queue cluster is accessed in only two places in the cluster-based controller. The zone status array is bundled into the queue cluster in the event case handling the value change of the switches cluster, and the zone status array is unbundled from the queue cluster immediately following the guard clause in the consumer loop. To convert the security system controller to a string-based queue, we only need to replace these two bundling and unbundling operations.

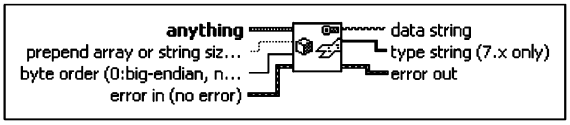
Flatten To String Function

Owning Palette: Data Manipulation Functions

Requires: Base Package

Converts the **anything** input to a flattened data string of binary values. You also can use this function to convert the byte order, or endian format, of the data in the flattened **data string**.

Details



Add to the block diagram Find on the palette

anything can be any data type.

prepend array or string size? indicates whether LabVIEW includes data size information at the beginning of **data string** when **anything** is an array or string. If **prepend array or string size?** is FALSE, LabVIEW does not include the size information. The default is TRUE. **prepend array or string size?** only controls the top-level data size information. Arrays and strings in hierarchical data types such as clusters always include size information.

byte order sets the endian format of the data in the resulting flattened string. Byte order, or endian form, indicates whether integers are represented in memory from most significant byte to least significant byte or vice versa.

0	**big-endian, network order** (default)—The most significant byte occupies the lowest memory address.
1	**native, host order**—Uses the byte-ordering format of the host computer.
2	**little-endian**—The least significant byte occupies the lowest memory address.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

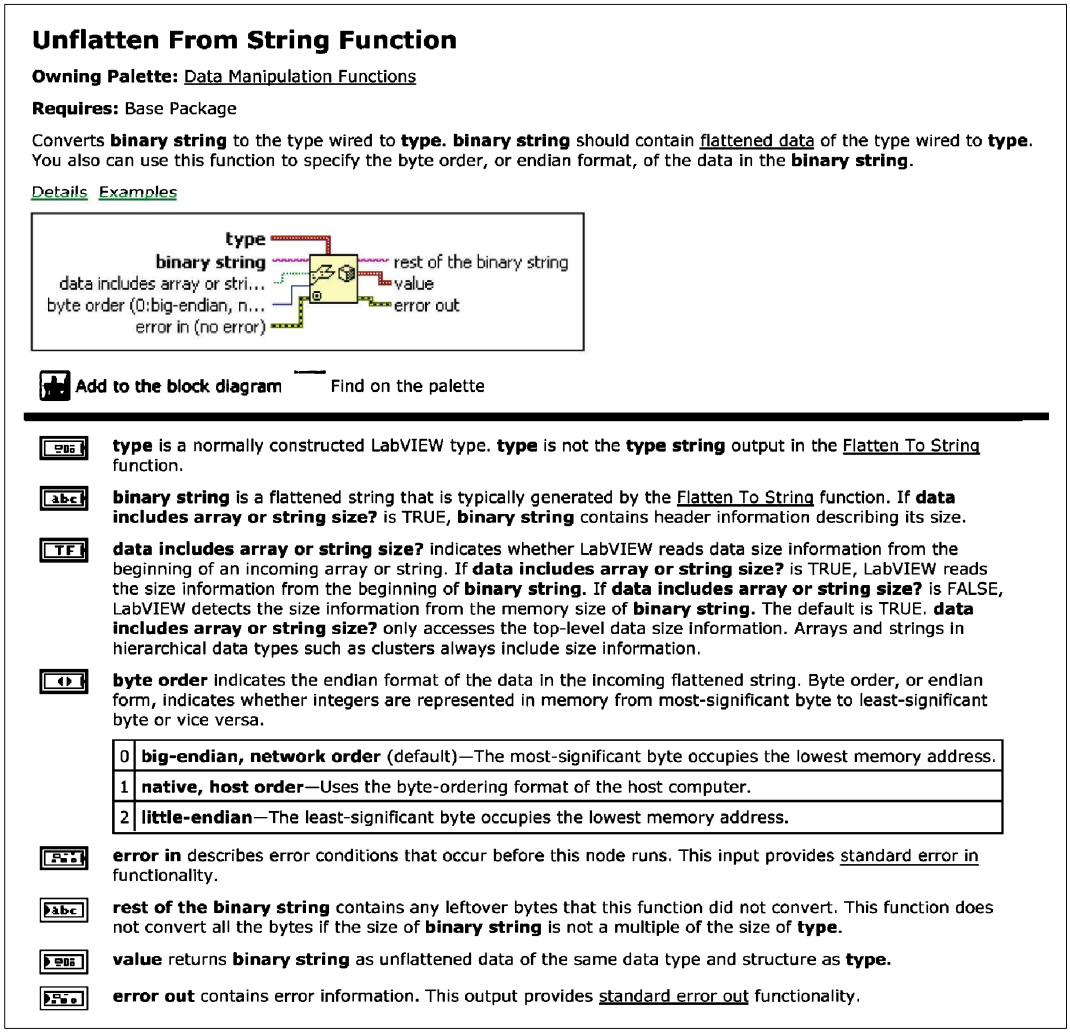
data string is the flattened data generated by the function. **data string** might contain header information before each non-scalar component describing its size. Such a string can be stored in a file or sent over a network. If you send the string over a network, the receiver must be able to interpret it. Usually, LabVIEW stores data as noncontiguous, indirectly referenced pieces. This function copies the data in LabVIEW form into a contiguous buffer **data string**. Use the Unflatten From String function to convert a data string back to any data type.

type string (7.x only) is visible only if the terminal is already wired from a previous version of LabVIEW or if you right-click the function and select **Convert 7.x Data** from the shortcut menu. **type string (7.x only)** is an encoded binary description of **data string**. **type string (7.x only)** is not the same as the **type** input in the Unflatten From String function. If **type string (7.x only)** cannot represent the data type wired to **anything**, this function returns an error.

error out contains error information. This output provides standard error out functionality.

Figure 30.3: Help file for the `Flatten To String` function

In this exercise, you will develop two sub-VIs to handle the transmission of the zone status array in a string-based queue. The queued string will again have the format `State:Argument`, where the argument in this case will be a string representing the zone status array. In this exercise, you will use the native LabVIEW functions **`Flatten To String`** with the help file shown in Figure 30.3 and **`Unflatten From String`** with the help file shown in Figure 30.4. These functions can be used to convert any LabVIEW data type to and from a string, even LabVIEW objects.



Unflatten From String Function

Owning Palette: Data Manipulation Functions

Requires: Base Package

Converts **binary string** to the type wired to **type. binary string** should contain flattened data of the type wired to **type**. You also can use this function to specify the byte order, or endian format, of the data in the **binary string**.

Details Examples

Add to the block diagram Find on the palette

type is a normally constructed LabVIEW type. **type** is not the **type string** output in the Flatten To String function.

binary string is a flattened string that is typically generated by the Flatten To String function. If **data includes array or string size?** is TRUE, **binary string** contains header information describing its size.

data includes array or string size? indicates whether LabVIEW reads data size information from the beginning of an incoming array or string. If **data includes array or string size?** is TRUE, LabVIEW reads the size information from the beginning of **binary string**. If **data includes array or string size?** is FALSE, LabVIEW detects the size information from the memory size of **binary string**. The default is TRUE. **data includes array or string size?** only accesses the top-level data size information. Arrays and strings in hierarchical data types such as clusters always include size information.

byte order indicates the endian format of the data in the incoming flattened string. Byte order, or endian form, indicates whether integers are represented in memory from most-significant byte to least-significant byte or vice versa.

0	**big-endian, network order** (default)—The most-significant byte occupies the lowest memory address.
1	**native, host order**—Uses the byte-ordering format of the host computer.
2	**little-endian**—The least-significant byte occupies the lowest memory address.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

rest of the binary string contains any leftover bytes that this function did not convert. This function does not convert all the bytes if the size of **binary string** is not a multiple of the size of **type**.

value returns **binary string** as unflattened data of the same data type and structure as **type.**

error out contains error information. This output provides standard error out functionality.

Figure 30.4: Help file for the `Unflatten From String` function

Develop the `Create State and Argument String` sub-VI whose icon is shown in Figure 30.5. This sub-VI accepts a string containing the name of the next state and a zone status array as arguments. Flatten the zone status array to a string, and concatenate it with the state string into an output string of the form `State:Argument`.

Also, develop the `Parse State and Zone Status Array` sub-VI with the icon shown in Figure 30.6. This sub-VI accepts a string from the queue and parses it to return the state string and a zone status array. Care needs to be taken with this sub-VI. The **`Unflatten From String`** function will generate an error if it is given an empty string to unflatten, so it is important to check to see if the input string actually has an argument before you attempt to unflatten it. Also, it is very important that this sub-VI

operate normally—even if an error is present at its error input terminal. Do you see why?

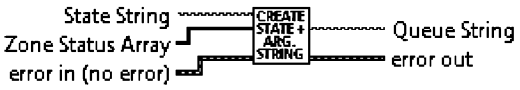


Figure 30.5: The icon of the `Create State and Argument String` sub-VI

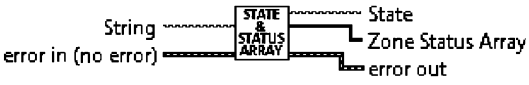


Figure 30.6: The icon of the `Parse State and Zone Status Array` sub-VI ■

Exercise 11.6: Implement a String-Based Producer–Consumer Security System

Revise the cluster-based producer–consumer security system controller developed in Exercise 11.2 to use a string-based queue. Use the flattening and unflattening sub-VIs developed in Exercise 11.5 to transmit the zone status array as a string from the producer loop to the consumer loop. Compare the readability of the cluster-based and string-based producer–consumer security system controllers.

Test the final code to ensure that the conversion from clusters to strings did not impact the error handling, parallelism, or scalability of the code. Are the simulated producer loop errors and consumer loop errors captured and reported correctly? Can you still change the number of zones on the front panel without modifying the block diagram? If you substitute the `Update Log Slowly` sub-VI for the `Update Log` sub-VI, is the front panel still responsive? ■

Exercise 11.7: Implementing a More Lightweight Flattening and Unflattening Scheme

The native LabVIEW flattening and unflattening functions have the advantage of universality: They can flatten and unflatten any LabVIEW data types, including objects. But they also have disadvantages. The primary disadvantage is that the process involved is a black box that is not under the control of the developer. The output is a binary string that cannot be read by the developer. The flattening and unflattening routines also have error terminals. This suggests that they might produces errors if not used correctly, and error-producing black boxes can be frustrating to deal with. Recall also that in Exercise 11.5 the unflattening sub-VI had to be specially constructed to execute correctly even if an error was present at its error input terminal.

In this exercise, you will create your own flattening and unflattening routines for the zone status array. These routines will not be universally applicable, but they will be lightweight, will produce human-readable output, and will have no error terminals. To do this, we will use the following scheme. Represent each possible zone status with a single letter: `N` for `Normal`, `A` for `Alarm`, `B` for `Bypass`, and `T` for `Tamper`. The zone status array can then be represented by a single string with one character for each zone. The position of a character in the string corresponds to the position of the zone within the array. For example, if there are six zones and all zones are normal, the string representation would be `NNNNNN`. Seven zones with all normal except for an alarm in Zone 3 and a Tamper in Zone 6 would be represented as `NNANNTN`.

First create two helper sub-VIs. Create the `Character Array To String` with the icon shown in Figure 30.7. This sub-VI accepts an array of zone status characters and concatenates them into a single string. Also create the `String To Character Array` sub-VI with the icon shown in Figure 30.8. This sub-VI performs the inverse operation, it accepts a string of zone status characters and breaks them up into an array of zone status characters. You might find the **`String Length`** and **`String Subset`** functions useful (see the help files in Figures 30.9 and 30.10). The helper sub-VIs should make no assumptions about the number of zones in the security system.

Then use these helper sub-VIs to to create the custom flattening and unflattening sub-VIs. Create the `Zone Status Array To String` sub-VI with the icon shown in Figure 30.11. This sub-VI accepts a zone status array and outputs a single string representing the elements of the array using the scheme described above. Also, create the `String To Zone Status Array` sub-VI with the icon shown in Figure 30.12. This sub-VI performs the inverse operation, it accepts a zone status string as an input and outputs a zone status array. The flattening and unflattening sub-VIs should make no assumptions about the number of zones in the security system.



Figure 30.7: The icon of the `Character Array To String` sub-VI



Figure 30.8: The icon of the `String To Character Array` sub-VI

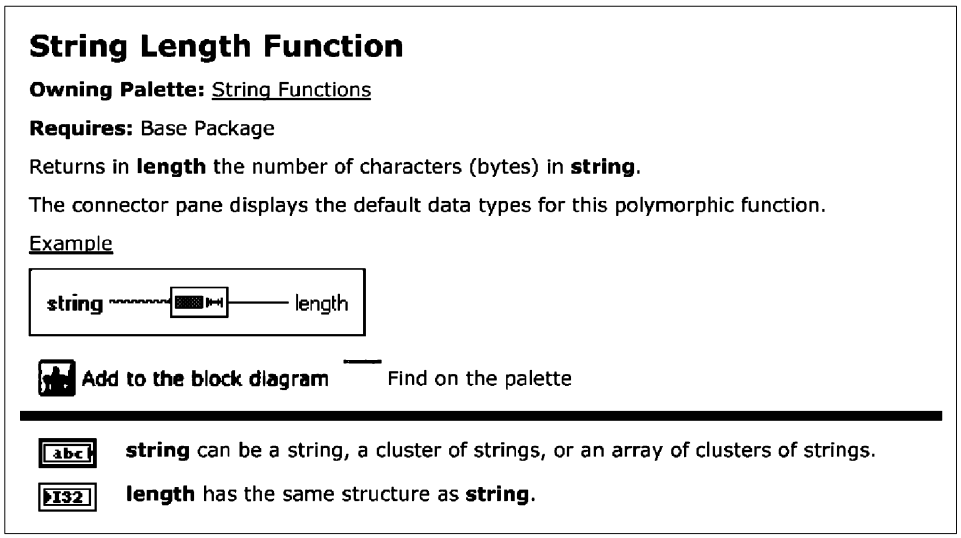


Figure 30.9: Help file for the `String Length` function

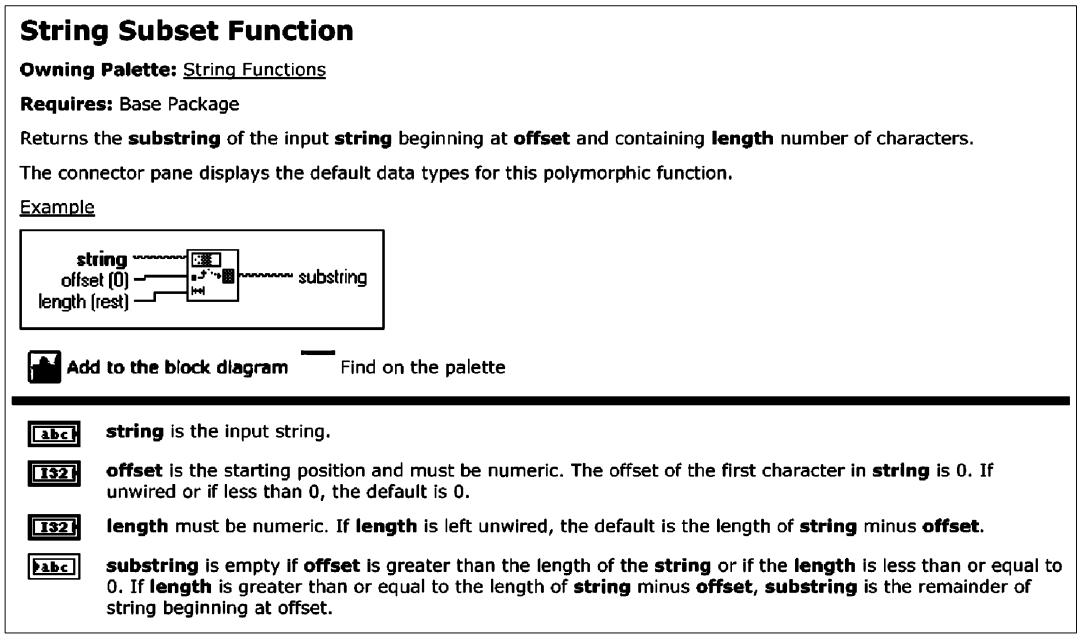


Figure 30.10: Help file for the `String Subset` function



Figure 30.11: The icon of the `Zone Status Array To String` sub-VI

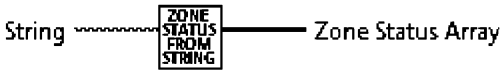


Figure 30.12: The icon of the `String To Zone Status Array` sub-VI

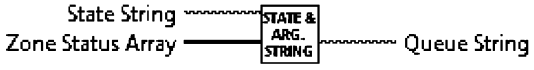


Figure 30.13: The icon of the `Create State and Argument String Custom Flattening` sub-VI

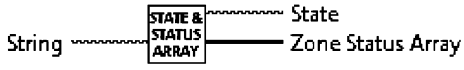


Figure 30.14: The icon of the `Parse State and Zone Status Array Custom Unflattening` sub-VI

Finally, create the custom sub-VIs that will encode and decode the actual queued strings. Create the `Create State and Argument String Custom Flattening` sub-VI with the icon shown in Figure 30.13. This sub-VI will replace the one shown in Figure 30.5. Then create the `Parse State and Zone Status Array Custom Unflattening` sub-VI with the icon shown in Figure 30.14. It will replace the one shown in Figure 30.6.

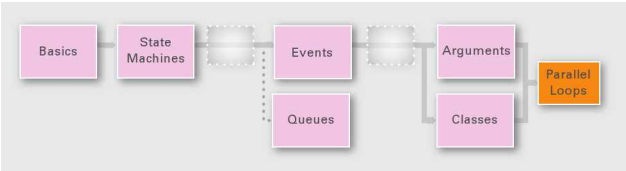
Revise the string-based producer–consumer security controller developed in Exercise 11.6, and replace the flattening and unflattening sub-VIs developed in Exercise 11.5 with the ones developed in this exercise. Test the scalability of the

controller by changing the number of zones on the front panel. Can you still change the number of zones on the front panel without modifying the block diagram? What are the advantages and disadvantages of using a custom flattening and unflattening scheme? ■

30.2 Car Wash 2 Exercises

Exercise 11.8: Implementing a Producer–Consumer Car Wash 2 Controller

Implement a producer-consumer version of the Car Wash 2 controller developed in Exercise 10.8. The Car Wash 2 controller has the same dequeue–requeue structure as the original car wash controller and it has the same potential for a race condition if it is shut down in the middle of executing the *Option* state. Use the switched queue developed in Chapter 29 to eliminate the potential race condition. Add front panel buttons to allow the user to simulate errors in the producer and consumer loops. Test your code with these buttons. Does the code shut down properly? Is the correct error reported? ■



31 Wrapping It Up

We've come a long way from Figure 0.1. Let's review the architectures we have studied:

31.1 The Classic State Machine

Pros: The classic state machine was a big step forward from spaghetti code. The classic state machine architecture is flexible and well-organized. It allows the code to fit on one screen comfortably. It incorporates good error handling techniques, checking for errors each iteration, and reporting errors to an error handling routine when the program ends. The classic state machine provides persistent storage through the use of the data highway. The classic state machine architecture provides space for initialization code and shutdown code. The classic state machine allows the developer to shut the program down gracefully in response to user input or if an error is detected.

Cons: The classic state machine can only specify one state transition at a time. Each state is responsible for determining the transition to the next state. This architecture can only think one step ahead. Also, the classic state machine is awkward if there is a *Wait* state. The classic state machine must poll all front panel objects to detect user input. The classic state machine iterates continuously, even while waiting for user input.

31.2 Event-Driven State Machine

Pros: The event-driven state machine uses an event structure to detect user input. Event structures are much easier to use than polling front panel objects. They are easily scalable as well. New event cases can be added without disturbing the code in existing cases. The event case also has the benefit of putting LabVIEW to sleep while the event case is waiting for input. This frees up the CPU to perform other tasks while the event-driven state machine is waiting for user input.

Cons: The event-driven state machine architecture is not useful if the state machine doesn't have a *Wait* state. An event queue is maintained by LabVIEW "behind the scenes," and the event queue is not accessible to the developer. This can lead to unexpected behavior, particularly in single-loop architectures. LabVIEW places events on the event queue—even if the event structure is not actively waiting for events. When the event structure finally becomes active, it can respond to events that happened at an earlier point in the program. Event structures also allow the developer to lock the front panel while the event is being processed, and this too can lead to unexpected behavior.

31.3 Queued State Machine

Pros: Queued state machines allow the developer to specify sequences of states rather than a single state. The state transitions are then dictated by the queue rather than the individual states. Queues allow the developer to program at a higher level of abstraction than the classic state machine. For example, the developer can think in terms of Deluxe Wash and Economy Wash rather than thinking in terms of individual states. Queues also improve readability, since the reader can can see lists of states grouped in their natural sequences rather than having to skip from one state to the next to figure out the likely order of state execution. Queues also can be used to communicate between parallel loops without creating a data dependency (see Chapter 27).

LabVIEW provides many functions for queue management. Strings can be used as the queue element, promoting code reuse, and making it easier to queue large groups of states in a readable fashion.

Cons: Queues are passed by-reference rather than by-value. This is not a bad thing, but it does violate LabVIEW's normal dataflow philosophy. It can also lead to race conditions when queue operations are carried out in parallel (see Figure 13.6 and Figure 13.7). Queues are not as useful in programs that do not require long sequences of states to be specified at once.

31.4 State Machines with Arguments

Pros: The addition of arguments to a state machine can reduce the number of states significantly. Groups of structurally similar states can be replaced with a single state that accepts arguments, increasing the readability and maintainability of the code. In this book, we examined schemes for passing arguments using clusters or strings. The string-based argument scheme allowed arguments to be optional for any given state.

Cons: Arguments are not useful for many state machines, but they are powerful if they can be used. String-based arguments require the developer to define and implement a parsing scheme, requiring the creation of parsing sub-VIs.

31.5 State Machines with Classes

Pros: The addition of classes to a state machine provides true data encapsulation for persistent data. The data that is stored in a class can not be accessed from outside the class. All manipulation of the data is done through method VIs of the class. The access scope of the method VI also can be specified by the developer.

Classes can have special relationships with other classes. Composition allows one class to contain instances of other classes. Inheritance allows one class to inherit the methods and data of a parent class. These techniques increase code reusability. Inheritance allows a code to be written in terms of the parent class, even if the children classes actually ride on the wires at runtime. This increases the scalability of the code. New children classes can be added in the future without requiring changes to the main VI.

Cons: Classes require a different programming mindset from standard LabVIEW programming techniques. Classes can require substantial planning and development effort.

31.6 Producer–Consumer State Machines

Pros: The producer–consumer state machine is a parallel architecture and is fundamentally different from all of the other architectures discussed in this book. The advantage of a parallel architecture is that multiple tasks can be performed at the same time without interfering with each other.

Cons: The disadvantage of a parallel architecture is that it is more complex than single-loop architectures. A messaging system needs to be established between the loops that does not create a data dependency between the loops. Parallel loops also create opportunities for race conditions and deadlocks.

31.7 Coupling

It is useful to review the coupling of the various state machine architectures in terms of car wash controller. The classic state machine and event-driven state machine implementations of the car wash controller did not allow sequences of states to be specified. Each state determined the transition to the next state. Some of the car wash states needed to know whether the Deluxe or Economy Wash had been chosen to determine which state to go to next. The value of the DELUXE WASH button was placed on the data highway for this purpose. This meant that the car wash states were coupled to the type of wash chosen.

The entire wash sequence of states were queued when a wash was selected, allowing the removal of the DELUXE WASH button from the data highway. The car wash states were uncoupled from the type of wash chosen. But this scheme only

worked because the three shared car wash states (*Soap Application*, *Main Wash*, and *Rinse*) had the same time targets in both the Deluxe and Economy Wash sequences. The wash types were still coupled to each other through the identical time targets of these three states.

The use of arguments broke this final coupling. The addition of arguments to the state machine architecture allowed the time targets to be removed from the individual states. The time targets were passed to the generic *Cycle* state as an argument instead. This allowed the time targets of the common wash states to be different for the Deluxe Wash and Economy Wash (if desired). The wash types were fully decoupled from each other.

The state machine with arguments implemented a generic *Cycle* state that depended on two mapping sub-VIs for its operation. One sub-VI contained a map of which proximity switch was checked by each car wash cycle. The other sub-VI contained a map of which indicator was lit by each car wash cycle. The physical layout of the car wash was contained in these two sub-VIs. The layout of the car wash was coupled to the main VI through these sub-VIs. If a new car wash state was added in the future, these sub-VIs would have to be modified. If the car wash controller was being used as an executable, the executable would have to be recompiled.

The state machine with classes breaks this last form of coupling. Each individual car wash class used dynamic override vis to select a proximity switch and an indicator lamp. The physical layout of the car wash, once concentrated in the two mapping sub-VIs, was distributed among the various car wash classes. No single class or VI contained the entire layout of the car wash. This decoupled the physical layout of the car wash from the main VI. The existing classes could be revised, or new car wash classes could be added in the future without requiring a modification of the main VI.

31.8 Selecting an Architecture

Each of the architectures that we've studied has different strengths. Each has been an upgrade to the ones that preceded it, but this is true only because each "upgrade" was beneficial to the car wash controller. Each problem is unique, and the merits of each architecture must be weighed against the requirements of the given problem.

For example, event structures may not be useful for programs that don't need to wait for user input, such as the traffic light controller described in Appendix A. Queues may not have much benefit if the sequences of states do not need to be specified. Again, the traffic light controller fits this description. This combination suggests that a classic state machine architecture might be the best choice for this program.

Event structures are usually worthwhile if the program waits for input, such as the security system controller described in Appendix B and the ATM controller described in Appendix F. They are not as useful for programs that don't need to wait for user input, such as the traffic light controller described in Appendix A. Queues were definitely an advantage in the car wash controller with its long sequences of states, but for programs like the traffic light controller, they are not an advantage. In fact, since the traffic light controller does not wait for input and does not need to queue long series of states, the classic state machine pattern is a fine choice of architecture.

Arguments can be a powerful technique, but they are only useful for certain kinds of problems. Look for situations like the car wash controller where a set of nearly identical states can be replaced by a single state that accepts arguments. Arrays sometimes can be used instead of clusters or strings for this kind of problem as well. The sprinkler controller described in Appendix E is an example of this. In this problem, the sprinkler zones to be serviced and the time targets are stored in an array on the front panel. When solving this problem, try to design a single state to service a generic sprinkler zone and use the array of zones and target times as an argument.

Finally, consider whether a proposed solution to a problem would benefit from an object-oriented approach. In this book, classes were discussed in two separate chapters highlighting the benefits of data encapsulation and inheritance. This is because the two chapters represent very different programming philosophies about the use of classes. The implementation of the car wash controller developed in Chapter 23 was very much like the ones developed in the earlier chapters of this book. The main difference was that a `Timer` object was used instead of the car wash timer action engine. The `Timer` object existed in isolation, it had no relationships with objects from other classes.

This approach defeated the `Timer Bug` bug shown in Figure 23.1. It encapsulated the timer data and protected it. Multiple copies of the car wash controller could run on the same computer without interfering with each other. Developing the class did not take much more effort than developing the action engine, yet the class was more reusable and it provided data encapsulation. As a general rule of thumb, consider using classes instead of action engines.

The use of classes in Chapter 25 was significantly different than the use of classes in Chapter 23. In this chapter, the classes are worked in relationship with other classes using inheritance and composition, as diagrammed in Figure 25.2. The programming philosophy was also quite different. Much of the problem-specific logic was moved out of the main VI and into the classes. A reader needs to understand both the class diagram and the main VI to fully understand the workings of the car wash controller. The amount of planning and programming effort in creating the classes was significantly greater than the effort needed to create the car wash timer action engine. The costs and benefits of this approach must be weighed carefully in light of the requirements of a given functional specification.

31.9 How Can You Improve?

Now that you've reached the end of this book, what's next? How can you improve your LabVIEW programming? There are a number of things you can do. First, try to duplicate the programs you see in this book. Start with a blank VI and use the figures as your guide. Can you make the programs work? Second, try your hand at the exercises provided in this book. Don't give up too quickly. Get as far as you can before you look at the solutions. Your solution may be different from the ones presented here, and they may even be better.

Hopefully, you learned something from this book. If you have already created working LabVIEW codes, go back and review them critically in light of what you have

learned from this book. Does your code look as good to you now as it did when you first created it? If not, revise it. Scrap it if you have to and start from scratch. Choose an appropriate architecture and rebuild it from the ground up if you have to. Reworking your own code is a valuable exercise and it may not take as much time as you think it will. You are already familiar with the challenges presented by the problems you've solved, and if you have created sub-VIs, they likely will be useful in your new code with little or no revision. The result of your rework should be a code that is more readable, maintainable, and scalable. It will be an investment in your skill set, and the final product will be more valuable to you or your employer than your original code.

31.10 Where to Go from Here

In Chapter 1, it was noted that LabVIEW codes are easier to read LabVIEW codes on a computer screen than on paper. This may well be the reason that it is easier to study LabVIEW on the Internet than from books. A quick search on Amazon will show depressingly few LabVIEW books. Of those that are available, few are widely known and praised in the LabVIEW community. There are many LabVIEW books dedicated to specific topics (data acquisition, image processing, user interfaces, etc.). Two books of broad interest with material that would appeal to most LabVIEW programmers are *LabvIEW for Everyone* by Travis and Kring and *The LabVIEW Style Book* by Peter Blume. *LabVIEW for Everyone* is an exhaustive reference covering most of LabVIEW's functionality. *The LabVIEW Style Book* is an excellent treatise on LabVIEW style from the viewpoint of practicing LabVIEW professionals.

The best sources of LabVIEW information and advice can be found online. The National Instruments website has the *NI Developer Zone* at `www.zone.ni.com`. LAVA (LabVIEW Advanced Virtual Architects), `www.lavag.org` is a vibrant online community of LabVIEW professionals and enthusiasts. OpenG (`www.openg.org`) is the ome of the LabVIEW open source community. The name "OpenG" derives from the fact that the LabVIEW is not the true name of the programming language. LabVIEW is the name of the development environment that allows you to program in the G programming language, where "G" stands for "graphical". OpenG is an organization dedicated to providing open-source G code for the LabVIEW community.

There are also a number of blogs that are recommended for students of LabVIEW. Blogs are ephemeral creatures, but at the time of publication of this book, these four blogs are recommended, check out their blogrolls for even more links:

- *Eyes on VIs* by Christina Rogers, software enginerr in LabVIEW R&D (`eyesonvis.com`).
- *Software Engineering for LabVIEW* by Elijah Kerry, Serior Product Manager for LabVIEW and Certified LabVIEW Architect (`ekerry.com`).
- *LabVIEW Artisan* by Darren Nattinger, Senior Software Engineer, LabVIEW R&D, Certified LabVIEW Architect (`labviewartisan.blogspot.com`).
- *Thinking in G* by Jim Kring, co-author of *LabVIEW for Everyone*, Certified LabVIEW Architect, OpenG administrator (`thinkinging.com`).

A

Sample CLD Exam—Traffic Light Controller

Task

Your company has received a contract to produce a traffic light controller for a local city government. The purpose of the traffic light controller is to maintain traffic at a heavily used far-way intersection. You have been assigned to create the software for this system using LabVIEW. You will create the software for one intersection that includes north, south, east, and west bound traffic with each direction having a left-hand turn lane.

General Operation

The traffic light controller provides for green, yellow, and red lights for each direction. In addition, a second green-yellow-red light must be provided for the left-hand turn lane. The period of time the lights should be ON are:

- Red: 4 seconds
- Yellow: 2 seconds
- Green: 4 seconds

The pattern of the lights with no left-turn sequence should be according to the following table:

North Bound	South Bound	East Bound	West Bound
Green	Green	Red	Red
Yellow	Yellow	Red	Red
Red	Red	Green	Green
Red	Red	Yellow	Yellow

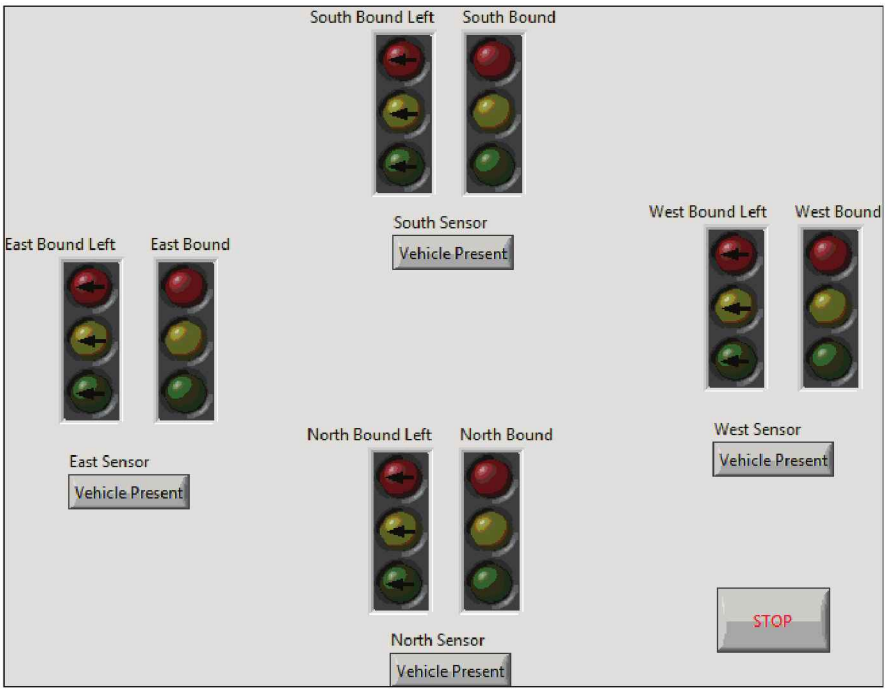


Figure A.1: Front panel for the `Traffic Light Controller` VI

The pattern of lights with a left-turn sequence for an East Bound vehicle should be according to the following table:

North Bound	South Bound	East Bound	West Bound
Green	Green	Red	Red
Yellow	Yellow	Red	Red
Red	Red	Left Green	Left Green
Red	Red	Left Yellow	Left Yellow
Red	Red	Green	Green
Red	Red	Yellow	Yellow

The left-turn sequence should only be inserted if a vehicle is positioned over the left-turn lane sensor at the time the left-turn sequence should start.

B

Sample CLD Exam—Security System

Task

Your company has received a contract to produce a security system for a local manufacturer. The purpose of the security system is to monitor the perimeter of the manufacturing complex to prevent unauthorized personnel from entering the grounds. Your job is to design the security system using LabVIEW that satisfies the provided specifications.

Instructions

Using a front panel similar to the graphic provided in Figure B.1, create a LabVIEW application that implements the actions of the security system. You will create the software for six perimeter zones with the actual hardware being simulated by Boolean switches on the front panel. The application should be easily scalable to add more zones without having to manually update the hierarchy.

Definitions

Zone A perimeter area with its own security status indication.

Alarm A zone condition indicating an intrusion into a zone.

Bypass A state in which a zone will not indicate an alarm condition. Placing a zone in bypass prevents nuisance alarms when maintenance work is performed in the area near the perimeter fence or when severe weather may cause false alarms.

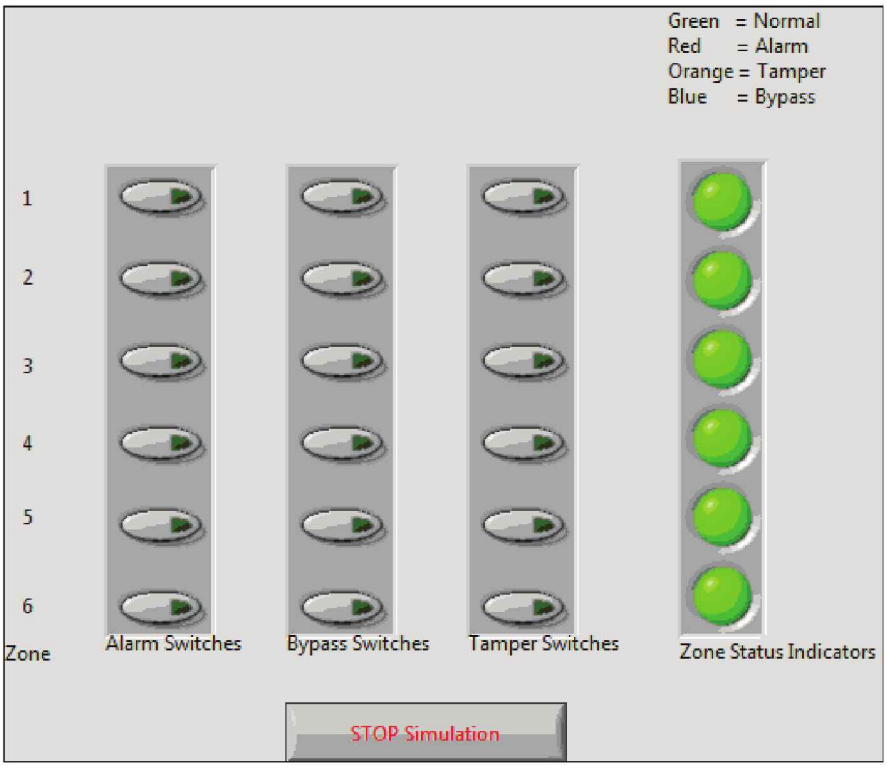


Figure B.1: Front panel for the `Security System` VI

Tamper A condition where the wiring of a zone has been altered in some way. All zone wiring is supervised. That is, if an individual attempts to circumvent the system by altering the wiring, that zone will indicate a tamper condition.

Description of Controls/Indicators

The security system software accepts Boolean inputs via front panel switches (one set for each zone) for the following functions:

- ALARM input
- BYPASS input
- TAMPER input

The security system provides for one indicator light for each zone. The color of the light provides the status information for that zone.

Green Normal
Red Alarm
Blue Bypass
Orange Tamper

Operation of the Security System

System

- A BYPASS input should always override an ALARM input but should not turn off an existing *Alarm* condition. An *Alarm* condition should not be indicated while a zone in a *Bypass* condition.
- A TAMPER input should always override both an ALARM input and/or a BYPASS input but should not turn off existing *Alarm* and/or *Bypass* conditions. *Alarm* and *Bypass* conditions should not be indicated while a zone is in a *Tamper* condition.
- An existing condition should be indicated when an overriding input is removed.

File Logging

The security system should produce an ASCII disk file in a spreadsheet-readable format. The data, when imported into a spreadsheet, should be in the following format (where XXXX represents logged data).

Date	Time	Zone	Status
XXXXXXX	XXXXXXX	XXXXXXX	XXXXXXX
XXXXXXX	XXXXXXX	XXXXXXX	XXXXXXX

The `Status` field should contain a string indicating the condition of the zone: `Normal`, `Alarm`, `Bypass`, or `Tamper`. The log file should be stored in a location relative to the location of the security application, updated only when the status of a zone changes, and closed after every transaction.



Sample CLD Exam—Boiler Controller

General Operation

The boiler startup controller allows a user to start up and shut down a boiler. The user interacts with controls on the front panel (Figure C.1) to start up and shut down the boiler and simulate conditions in the system. Indicators on the front panel display the status and the current step in the startup and shutdown process. The controller also logs events as they occur during the process.

Sequence of Operation

Note: See the Event Log File Specification section later in the document for file format and update policies.

Start (Application Run): When the application starts, the STATUS string should indicate `Lockout` and all indicator LEDs should be OFF. All controls should be initialized to the state shown on the previous front panel.

The following data should be logged to the event log file:

Time String: Absolute date and time at event

Event String: `Boiler Initialized`

Event Data String: 0

Reset Lockout: Turn the RUN INTERLOCK switch to the closed position and click the BOILER RESET button. This action should change the STATUS string to indicate Ready.

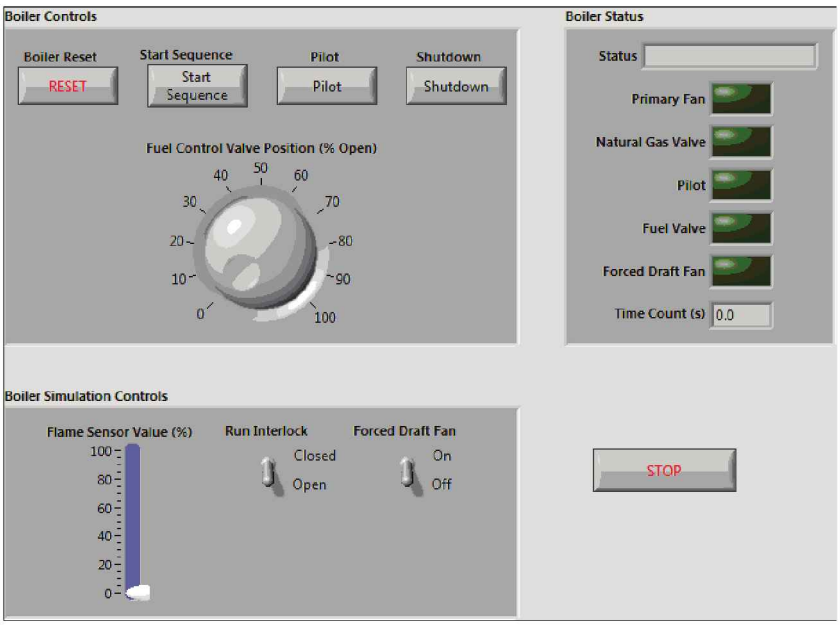


Figure C.1: Front panel for the `Boiler Controller` VI

Note: The purpose of the RUN INTERLOCK switch is to simulate external condition(s) that need to be satisfied before beginning the start sequence.

The following data should be logged to the event log file:

Time String: Absolute date and time at event
Event String: `Boiler Ready`
Event Data String: 0

Pre-Purge: Click the START SEQUENCE button. This action starts the primary fan for the pre-purge cycle. This step should last for 10 seconds, during which:

- The PRIMARY FAN LED should turn ON.
- The TIME COUNT indicator should count up to indicate elapsed time in seconds.
- The STATUS string should indicate `Pre-Purge`.

The following data should be logged to the event log file:

Time String: Absolute date and time at event
Event String: `Start Pre-Purge`
Event Data String: Pre-purge elapsed time

Pre-Purge (continued): Upon completion of this step, the following should occur:

- The STATUS string should indicate `Pre-Purge Complete`.
- The PRIMARY FAN LED should turn OFF.
- The TIME COUNT indicator should reset to 0.
- The PILOT button should blink to signal that it is safe to ignite the pilot.

The following data should be logged to the event log file:

Time String: Absolute date and time at event
Event String: `Pre-Purge Complete`
Event Data String: Pre-purge elapsed time

Ignition (Ignite Pilot): Click the PILOT button. This action starts the flow of natural gas to provide the fuel and simultaneously creates the spark to ignite the pilot. This action should:

- Turn off the blinking of the PILOT button (initiated in previous step).
- Turn ON the NATURAL GAS VALVE LED and PILOT LED.
- Change the STATUS string to indicate `Pilot On`.

Note: The pilot should ignite only after the boiler completes the pre-purge step.
The following data should be logged to the event log file:

Time String: Absolute date and time at event
Event String: `Pilot ON`
Event Data String: True

Prove the Pilot: This step verifies an adequate pilot flame for firing the boiler and is simulated by manually incrementing the FLAME SENSOR VALUE slider. When the FLAME SENSOR VALUE is greater than 30%, the STATUS string should indicate `Boiler Ready`.
The following data should be logged to the event log file:

Time String: Absolute date and time at event
Event String: `Pilot Proved`
Event Data String: Value of FLAME SENSOR VALUE slider

Start and Run: Step 1: Turn ON the FORCED DRAFT FAN switch. The FORCED DRAFT FAN LED should turn ON.
The following data should be logged to the event log file:

Time String: Absolute date and time at event
Event String: `Forced Draft Fan ON`
Event Data String: True

Step 2: Move the FUEL CONTROL VALVE POSITION knob. The boiler should not start when the FUEL CONTROL VALVE POSITION is between 0% and 10%. The boiler should start only when the FUEL CONTROL VALVE POSITION is greater than 10%. At this time:

- The FUEL VALVE indicator should turn ON.
- The NATURAL GAS VALVE LED and PILOT LED should turn OFF.
- The STATUS string should indicate `Boiler Running`.

The following data should be logged to the event log file:

Time String: Absolute date and time at event
Event String: `Boiler Running`
Event Data String: Value of FUEL CONTROL VALVE POSITION

The boiler should operate normally when the FUEL CONTROL VALVE POSITION is between 10% and 75%.

Shutdown and Purge: Any of the following conditions can shut down the boiler after the *Start* and *Run* step:

- Boiler SHUTDOWN switch is clicked.
- FUEL CONTROL VALVE POSITION is less than 10% or greater than 75%.
- RUN INTERLOCK switch opens.
- FORCED DRAFT FAN switches OFF.

The shutdown process initiates a purge cycle. The purge cycle should last for 10 seconds and begin by:

- Turning OFF the FUEL VALVE indicator.
- Turning ON the PRIMARY FAN indicator.
- Changing the STATUS to indicate `Purge`.

The following data should be logged to the event log file:

Time String: Absolute date and time at event
Event String: `Start Shutdown Purge`
Event Data String: Purge elapsed time

Increment the TIME COUNT indicator to indicate elapsed time in seconds. After the purge cycle is complete, the STATUS string should indicate `Lockout` and all indicator LEDs should turn OFF.

Time String: Absolute date and time at event
Event String: `Shutdown Purge Complete`
Event Data String: Purge elapsed time

Description of Controls and Indicators

Control Name	Control Description—Function
BOILER RESET	Button—Resets the boiler
START SEQUENCE	Button—Initiates the pre-purge step
PILOT	Button—Initiates the ignition step
FUEL CONTROL VALVE POSITION (% Open)	Knob—Controls amount of fuel flowing into the boiler
SHUTDOWN	Button—Initiates the boiler shutdown
RUN INTERLOCK	Switch—Simulates external condition(s) that must be satisfied before the start sequence can initiate
FLAME SENSOR VALUE (%)	Slider—Simulates the percentage of pilot flame
FORCED DRAFT FAN	Switch—Controls the forced draft fan
STOP	Button—Stops the application within the specified time frame

Indicator Name	Indicator Description—Function
STATUS	String—Indicates boiler status (current step)
PRIMARY FAN	LED—Indicates ON/OFF status of the primary fan
NATURAL GAS VALVE	LED—Indicates ON/OFF status of the natural gas valve
PILOT	LED—Indicates ON/OFF status of the pilot
FUEL VALVE	LED—Indicates normal operating conditions of the fuel valve
FORCED DRAFT FAN	LED—Indicates ON/OFF status of the forced draft fan
TIME COUNT	Numeric—Indicates elapsed time

Event Log File Specification

File Name: `Boiler Log.txt`

File Location: Relative, same location as the main VI

Format: Comma Separated Value (.csv)

File Header: Timestamp, Event, Event Data

Event Data String Format: Absolute date and time string, Event string, Event data string

File Data: See Sequence of Operation section above

File Creation and Modification: If file does not exist, create new, write header and log event data. If file exists, append event data.



Sample CLD Exam—Car Wash Controller 2

Task

Design a car wash controller using LabVIEW. The front panel of the controller should look similar to the one shown in Figure D.1.

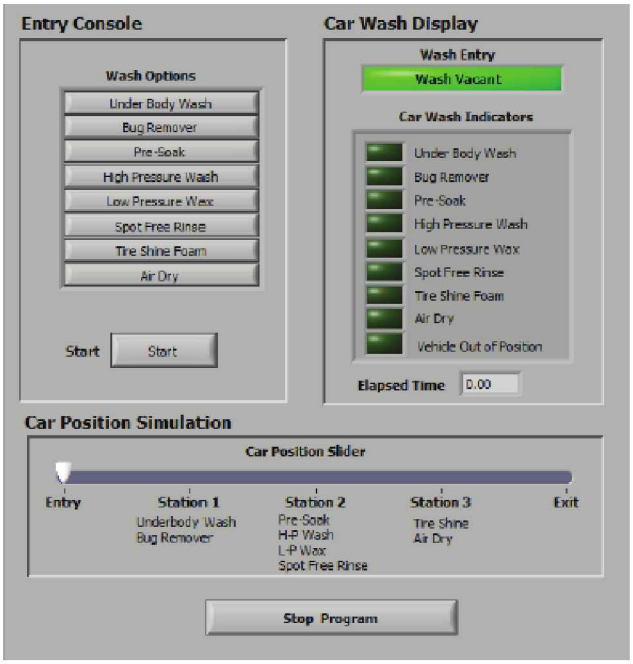


Figure D.1: Front panel for the `Car Wash Controller 2` VI

General Operation

The car wash controller simulates the control system of an automated car wash. The user interacts with controls and indicators on the front panel to select the car wash options and simulate the travel of the vehicle in the car wash.

The controller should perform the following general operations:

- Indicate on the Wash Entry LED if the car wash is vacant or a wash is in progress.
- Allow the user to select Wash Options.
- Start the washing process when the user clicks Start.
- Indicate the current wash step on the Car Wash Indicators LEDs and display the Elapsed Time on the indicator.
- Turn ON the VEHICLE OUT OF POSITION LED and pause the elapsing time if the vehicle moves from the designated position during a cycle.

Sequence of Operation

Start (Application Run): When the application starts, the front panel controls and indicators should be in the following states:

- **Entry Console:** All controls should be enabled.
- **Car Wash Display:** The Wash Entry LED should be green and display Wash Vacant. All Car Wash Indicators should be turned OFF and the Elapsed Time indicator should display `0.00`.
- **Car Position Simulation:** The Car Position Slider should be at the Entry position.

Select Wash Options: Click the Wash Options buttons to select the washing steps. The buttons should remain clicked until the completion of the wash cycles.

Start: Click the Start button to begin the car wash process. This should configure the controller to execute the steps selected by the Wash Options buttons.

Note: The High Pressure Wash step is the default wash. If the user did not select High Pressure Wash or did not select any options, High Pressure Wash must be added to the options programmatically after the use clicks Start.

The controller should disable the Wash Options buttons. The controller should signal the start of the wash by changing the color of the Wash Entry LED to red and displaying `Wash In Progress`. The controller should check if the vehicle is at the appropriate station for the first step in the selected wash type by monitoring the CAR POSITION SLIDER.

Note: Refer to the *Car Wash Step Timing and Station Positions* table at end of the specification for a list of stations that correspond to the car wash steps.

If the vehicle is not at the appropriate station, the controller should turn ON the VEHICLE OUT OF POSITION LED and not begin timing.

If the vehicle is at the appropriate station, the appropriate wash step LED should turn ON and the ELAPSED TIME indicator should count up from zero to indicate elapsed time for that step.

Upon completion of a step, the ELAPSED TIME indicator should reset to zero. Each wash step should not last for more than the time allocated in the *Car Wash Step Timing and Station Positions* table at the end of the specification.

Upon completion of a step, if the vehicle is in the appropriate station, the controller should continue to the next wash step by resetting and restarting the elapsed timer.

Upon completion of a step, if the vehicle is not in the appropriate station, the controller should turn ON the VEHICLE OUT OF POSITION LED and not begin timing.

If the vehicle moves away from the wash station in the middle of a wash step, the elapsed time should pause, the VEHICLE OUT OF POSITION LED should turn ON and the appropriate wash step LED should turn OFF. When the vehicle position is restored to the correct position by moving the CAR POSITION SLIDER, the VEHICLE OUT OF POSITION LED should turn OFF, the appropriate wash step LED should turn ON, and timing should continue from the paused time until completion of the step.

When all the wash steps are complete, the VEHICLE OUT OF POSITION LED should turn ON, indicating to the user to exit the car wash. When the user moves the CAR POSITION SLIDER to the *Exit* position, all CAR WASH INDICATORS should turn OFF and the CAR POSITION SLIDER should reset to the *Entry* position. The WASH ENTRY indicator should turn green and display `Wash Vacant`, and the controls in the WASH ENTRY CONSOLE should be enabled to allow the user to select and start another car wash process.

Stop Program: Click STOP PROGRAM to abort the car wash process at any time and stop the application. When the application stops, the front panel controls and indicators should be in the following states:

Entry Console All controls should be enabled.

Car Wash Display The WASH ENTRY LED should be green and display Wash Vacant. All CAR WASH INDICATORS should be turned OFF, and the ELAPSED TIME indicator should reset to zero.

Car Position Simulation The CAR POSITION SLIDER should reset to the *Entry* position.

Description of Controls and Indicators

Wash Options: Cluster of Buttons allows the user to customize the car wash.

Start: Button allows user to start car wash process.

Car Position Slider: Horizontal Pointer Slide allows user to simulate travel of car during the car wash process. (*Note:* Use Text Labels for Scale Markers.)

Labels	Values
Entry	0
Station 1	1
Station 2	2
Station 3	3
Exit	4

Note: Refer to the front panel figure provided for car wash steps corresponding to the stations.

Stop Program: Button stops the program.

Wash Entry: Square LED indicates if the car wash is operational.

- OFF Color: Green
- OFF Text: `Wash Vacant`
- ON Color: Red
- ON Text: `Wash in Progress`

Car Wash Indicators: Cluster of square LEDs indicates the current step in the wash cycle.

Cluster Order	Square LED Label
0	`Under Body Wash`
1	`Bug Remover`
2	`Pre-Soak`
3	`High Pressure Wash`
4	`Low Pressure Wax`
5	`Spot Free Rinse`
6	`Tire Shine Foam`
7	`Air Dry`
8	`Vehicle Out of Position`

Elapsed Time: Numeric representation: DBL, Digits of Precision: 2

Car Wash Step Timing and Station Positions:

Wash Steps	Step Time	Station
Under Body Wash	5 seconds	Station 1
Bug Remover	5 seconds	Station 1
Pre-Soak	5 seconds	Station 2
High Pressure Wash	5 seconds	Station 2
Low Pressure Wax	5 seconds	Station 2
Spot Free Rinse	5 seconds	Station 2
Tire Shine Foam	5 seconds	Station 3
Air Dry	5 seconds	Station 3

E

Sample CLD Exam—Sprinkler Controller

Task

Design a four zone sprinkler controller using LabVIEW. The front panel of the controller should look similar to the front panel in Figure E.1.

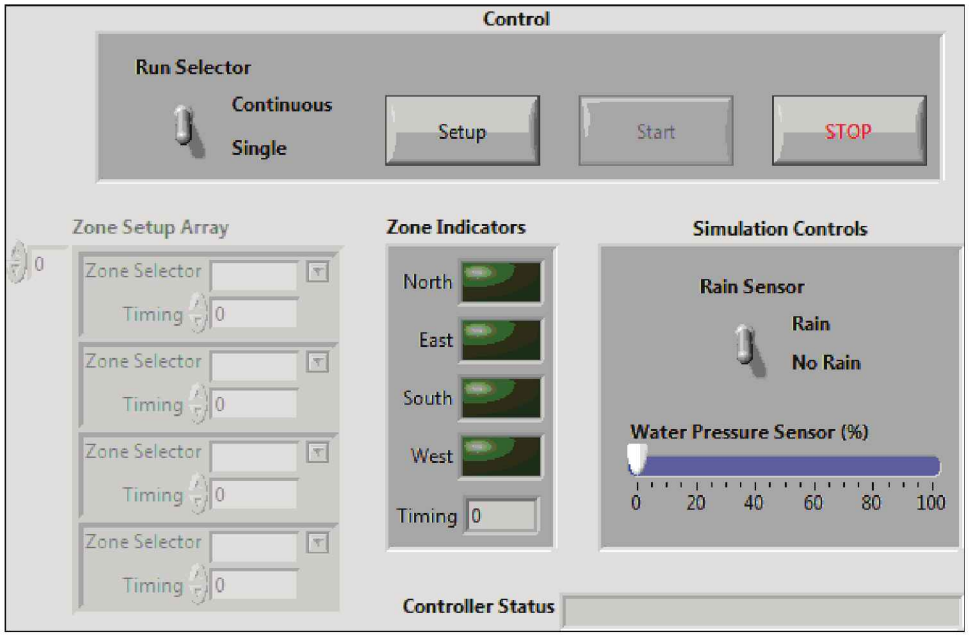


Figure E.1: Front panel for the `Sprinkler Controller` VI

General Operation

The sprinkler controller simulates a lawn sprinkler control system that services four zones: *North*, *South*, *East*, and *West*. Controls on the front panel control the zone selection and simulate external conditions. Indicators on the front panel indicate the zone that is currently being serviced and the elapsed time for that zone.

The controller must load the ZONE SETUP ARRAY data from a Comma Separated Value (.csv) file when the VI starts. The controller runs when the user clicks the START button. When the RUN SELECTOR switch is in the *Single* run position, the controller should service each zone for the set TIMING and wait for user input. When the RUN SELECTOR switch is in the *Continuous* run position, the controller should continually cycle through the ZONE SETUP ARRAY by restarting with the first zone after servicing the last zone. The controller (application) should stop when the user clicks the STOP button.

Note: The application should be scalable to handle additional elements of the ZONE SETUP ARRAY without any modification of the program.

Sequence of Operation

VI Load: When the application loads, all controls should initialize to the default states as shown on the front panel.

Start (Application Run): When the application runs, the sprinkler controller program should read the Sprinkler Configuration file (see format in Sprinkler Configuration File section) and load the ZONE SETUP ARRAY with the data from the file.

- ZONE SETUP ARRAY should be dimmed.
- All indicator LEDs should be set to OFF.
- The TIMING indicator should indicate zero.
- CONTROLLER STATUS should display `Controller Initialized`.

Setup: Click the SETUP button to activate the ZONE SETUP ARRAY control and allow the user to change the Zone Selector and the TIMING for each zone.

The Zone Selector should allow a user to select and change the zone: *North*, *East*, *South*, or *West*. The controller should handle zones in any order and allow setting of a zone more than once in the ZONE SETUP ARRAY.

The TIMING setting should be in seconds (no decimals). If TIMING is zero, the zone should be not be serviced when the controller runs.

In this mode, the CONTROLLER STATUS should display `Setup Mode`.

Clicking the SETUP button during a run should immediately stop the controller and activate the ZONE SETUP ARRAY.

Select *Continuous* or *Single* Run: Switch to the desired RUN SELECTOR mode.

When *Single* is selected, the controller should service the zones as setup in the ZONE SETUP ARRAY.

When *Continuous* is selected, the controller should continually cycle through the zones as setup in the ZONE SETUP ARRAY by repeating the cycle until the user clicks the STOP button.

Start: Click the START button to start the controller. This action should complete the following:

- Dim the ZONE SETUP ARRAY control.
- Check the WATER PRESSURE SENSOR simulation control. If the water pressure is below 50%, the CONTROLLER STATUS should display `Low Water Pressure`. If the water pressure is restored to a value greater than 50%, the controller should start operating.
- Run the controller in the mode selected by the RUN SELECTOR switch.
- Display `Running` in the CONTROLLER STATUS indicator.

When the RUN SELECTOR switch is in the *Single* position, the controller should make one pass, sequentially, through the zones in the ZONE SETUP ARRAY. The controller should service one zone at a time, beginning with the first zone. When a single run completes, the controller should wait for a user action. The CONTROLLER STATUS indicator should continue to display `Running` even after the single run is complete.

When the RUN SELECTOR switch is in the *Continuous* position, the controller should continue with the first zone after the last zone has been serviced.

If the RUN SELECTOR switch is moved from the *Continuous* position to the *Single* position during a continuous run, the controller should stop after servicing the last zone and wait for a user action.

When a zone is being serviced, the following should occur:

- Only its respective indicator should be ON.
- All other indicators should be OFF.
- The TIMING indicator should display the elapsed time in seconds.
- The CONTROLLER STATUS indicator should display `Running`.

Note: If zone TIMING is set to zero, the zone should be skipped and the next zone in order should be serviced.

If it begins to rain during the operation of the controller, simulated by the RAIN SENSOR switch, the controller must stop servicing the zones and display `Rain` in the CONTROLLER STATUS indicator, even if a zone has not completed its timing cycle.

If it stops raining and the RUN SELECTOR switch is in the *Single* position, the controller should make a single pass and wait for user input. If the RUN SELECTOR switch is in the *Continuous* position, the controller should start servicing the zone beginning with the first zone.

If the water pressure sensor detects a loss in water pressure during the operation of the controller ñ simulated by the WATER PRESSURE SENSOR slider, the controller must stop servicing the zone and display `Low Water Pressure` in the CONTROLLER

STATUS indicator. If the pressure is restored, the controller should start operating beginning with the first zone.

Stop: Click the STOP button to stop the controller (application). This action should complete the following:

- Stop servicing the zones.
- Display `Stopped` in the CONTROLLER STATUS indicator.
- Stop the application.

Description of Controls/Indicators

Control Name	Control Description—Function
RUN SELECTOR	Switch—Selects between *Single* and *Continuous* run modes
SETUP	Button—Activates setup operation
START	Button—Starts the sprinkler controller operation
STOP	Button—Stops the application
ZONE SETUP Array	Array of ZONE SETUP clusters
ZONE SETUP Cluster	Cluster of two elements: • Combo Box Zone Selector • Numeric (DBL) Zone TIMING (seconds)
RAIN SENSOR	Switch—Simulates presence of rain
WATER PRESSURE SENSOR	Slider—Simulates available water pressure

Indicator Name	Indicator Description—Function
NORTH	LED—Indicates ON when the North zone is being serviced by the controller
EAST	LED—Indicates ON when the East zone is being serviced by the controller
SOUTH	LED—Indicates ON when the South zone is being serviced by the controller
WEST	LED—Indicates ON when the West zone is being serviced by the controller
TIMING	Numeric—Indicates elapsed time (seconds)

CONTROLLER STATUS	String—Displays controller status. • `Controller Initialized` Indicates that the application run button was clicked • `Setup Mode` ñ Indicates that the SETUP button was clicked • `Running` ñ Indicates that the controller is running • `Low Water Pressure` ñ Indicates that the WATER PRESSURE SENSOR slider is at less than 50% • `Rain` ñ Indicates that the RAIN SENSOR switch is in the `Rain` position

Sprinkler Configuration File

Format: Comma Separated Value (.csv). See the Figure E.2 screenshot for the initial data.

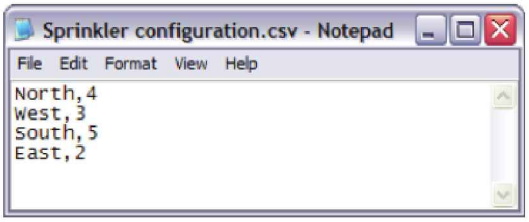


Figure E.2: The sprinkler configuration file

F

Sample CLD Exam—ATM Controller

Task

Design an Automated Teller Machine (ATM) controller using LabVIEW. The front panel of the controller should look similar to the front panel in Figure F.1.

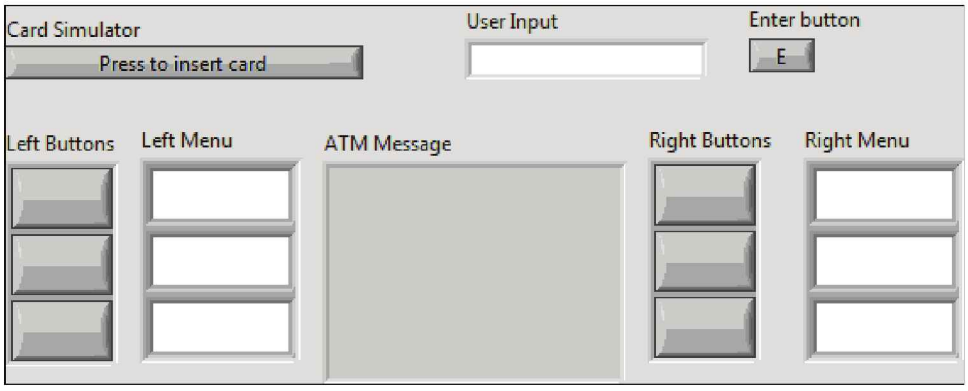


Figure F.1: Front panel for the `ATM Controller` VI

General Operation

The ATM controller simulates the control system of an Automated Teller Machine. The user interacts with controls and indicators on the front panel to perform common ATM functions such as deposit funds, withdraw funds, and inquire about the balance of funds from the user account.

The ATM controller has access to user account information stored in the `Accounts.txt` file. You, the developer, are responsible for creating the `Accounts.txt` file, which the ATM controller reads from or writes to, depending on user-initiated transactions. Refer to the ATM Accounts File specification section for more information about the initial content and data format of the `Accounts.txt` file.

The controller should perform the following general operations:

- Obtain user input through the USER INPUT, LEFT BUTTONS, and RIGHT BUTTONS.
- Manage the LEFT MENU and RIGHT MENU strings and take appropriate action by responding to the corresponding LEFT BUTTONS and RIGHT BUTTONS.
- Prompt user actions and confirm transactions by displaying appropriate messages in the ATM MESSAGES indicator.
- Monitor user inactivity and terminate the user session after the specified time expires.

Sequence of Operation

Start (Application Run): When the application starts, the USER INPUT, ENTER (E), LEFT MENU, RIGHT MENU, LEFT BUTTONS, and RIGHT BUTTONS should be disabled.

The CARD SIMULATOR button should be enabled, and the ATM MESSAGES indicator should display the `Welcome Message`.

Refer to the ATM Message Types section of the specification for message types and font specifications for the messages.

Insert ATM Card: Click the CARD SIMULATOR button. The button should remain pressed and the Boolean text should display `Card Inserted`. This action enables the ENTER (E) button and the USER INPUT string control, focuses the cursor to the USER INPUT control, and waits for the user to enter the account number in the control. The CARD SIMULATOR button should remain pressed until the user completes the session or a timeout (10 seconds) occurs.

After the user completes keying in the account number and presses the ENTER (E) button, the ATM controller should access the ATM Accounts file and verify that the account exists.

If the account does not exist, the ATM MESSAGES indicator should display the `Account Verification Failed Message` and prompt the user to re-enter the account number.

If the account exists, the ATM controller should enable the LEFT MENU, RIGHT MENU, LEFT and RIGHT BUTTONS, and display the `Main Menu Message` on the ATM MESSAGES indicator. The ATM controller should also populate the LEFT MENU and RIGHT MENU with the Main Menu and wait for user action.

Refer to the ATM Menus section of the specification for more information about the strings that appear in the menus.

Note: If at anytime after the insertion of the ATM card the ATM controller does not detect use of front panel controls for 10 seconds, the CARD SIMULATOR button should be released and the user session should be terminated by stopping the application. The ATM MESSAGES indicator should display the `Session Terminate Message`.

Deposit: Click the button next to the corresponding RIGHT MENU item. This action should display the `Deposit` Message, focus the cursor to the USER INPUT, and allow the user to enter the amount to be deposited. After the user enters an amount in the USER INPUT and presses the ENTER (E) button, the `Deposit Complete Message` should display on the ATM MESSAGES indicator, and the ATM Accounts file should update with the new balance.

Withdraw: Click the button next to the corresponding RIGHT MENU item. This action should display the `Withdrawal Message`, focus the cursor to the USER INPUT control, and allow the user to enter the amount to withdraw from the account. After the user enters an amount in the USER INPUT control and presses the ENTER (E) button, the ATM controller should check if the account has sufficient funds to complete the transaction.

If the account has sufficient funds, the controller should deduct the keyed in amount from the current balance, update the ATM Accounts file with the new balance, and display the `Withdrawal Complete Message` on the ATM MESSAGES indicator.

If the account has insufficient funds, the `Withdrawal Failed Message` should display on the ATM MESSAGES indicator.

Fast Cash $50: Click the button next to the corresponding RIGHT MENU item. The ATM controller should check if the account has sufficient funds to complete the transaction. If the account has at least $50, the ATM controller should deduct $50 from the current balance, update the ATM Accounts file with the new balance, and display the `Withdrawal Complete Message` on the ATM MESSAGES indicator.

If the account has insufficient funds, the `Withdrawal Failed Message` should display on the ATM MESSAGES indicator.

None: This button is reserved for future use. Clicking the button next to the corresponding LEFT MENU item should not produce any results. The ATM MESSAGES indicator should continue to display the current message.

Balance Inquiry: Click the button next to the corresponding LEFT MENU item to get the current balance from the ATM Accounts file. The ATM MESSAGES indicator should display the `Balance Inquiry Message`.

Return Card and Terminate: Click the button next to the corresponding LEFT MENU item to terminate the user session. The ATM MESSAGES indicator should display the `Session Terminate Message`. The ATM controller should release the CARD SIMULATOR button to indicate the return of the ATM card, disable the USER INPUT control, and terminate the application.

Description of Controls/Indicators

Control Name	Control Description—Function
CARD SIMULATOR	Button—Simulates insertion of an ATM card OFF text: Press to inset card ON text: Card inserted
LEFT BUTTONS	Cluster of Buttons—Allows user to select ATM transaction
RIGHT BUTTONS	Cluster of Buttons—Allows user to select ATM transaction
USER INPUT	String—Allows user to enter Account number, Deposit, and Withdrawal amounts
ENTER Button	Button—Commits the entry in the USER INPUT

Indicator Name	Indicator Description—Function
ATM MESSAGES	String—Displays messages to guide and provide feedback to user. Refer to the ATM Message Types section for the list of messages
LEFT MENU	Cluster of strings—Used to display menu. Refer to the ATM Menus section for the list of menus.
RIGHT MENU	Cluster of Strings—Used to display menu. Refer to the ATM Menus section for the list of menus.

ATM Message Types

Message Type	Message
Welcome Message	Welcome to Acme Bank. Please insert your card and enter your account number in the USER INPUT. Press ENTER (E) when done.
Main Menu Message	Welcome to Acme Bank First Name, Last Name (from file) Please select transaction by using the buttons.
Account Verification Failed Message	Account Information Incorrect Please Re-Enter Account Number. Press Enter (E) when done.
Session Terminate Message	Your session has been terminated due to inactivity or menu selection. Please take your card. Goodbye!

Deposit Message	Please enter amount to deposit and press Enter (E) when done.
Deposit Complete Message	$ (Amount) Deposited
Withdrawal Message	Please enter amount to withdraw and press Enter (E) when done.
Withdrawal Complete Message	$ (Amount) Withdrawn
Withdrawal Failed Message	Insufficient funds in account. Please check your balance and try again.
Balance Inquiry Message	Your Balance Is: $ (Balance Amount from file)

The font for the ATM Message indicator should be 14 point and bold. The message should be centered in the display, as shown in Figure F.2.

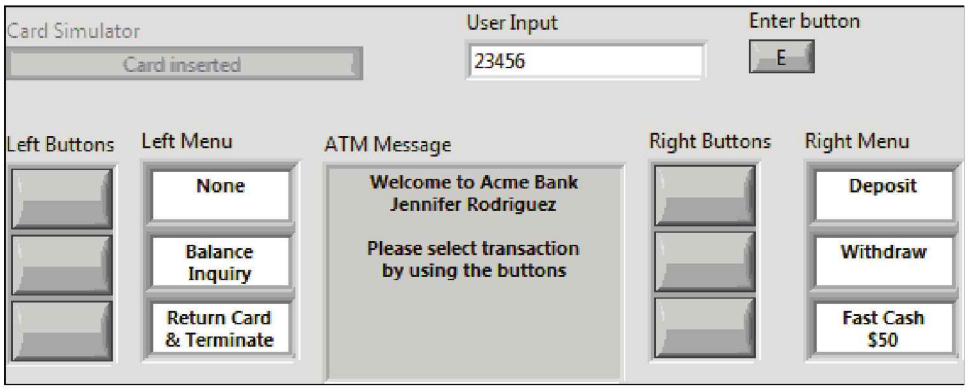


Figure F.2: Sample screen shot for ATM messages

ATM Menus

Main menu (Figure F.3)

Left Menu (Strings)	Right Menu (Strings)
None	Deposit
Balance Inquiry	Withdraw
Return Card and Terminate	Fast Cash $50

Note: The application should be scalable to add other menus without major modifications to the application.

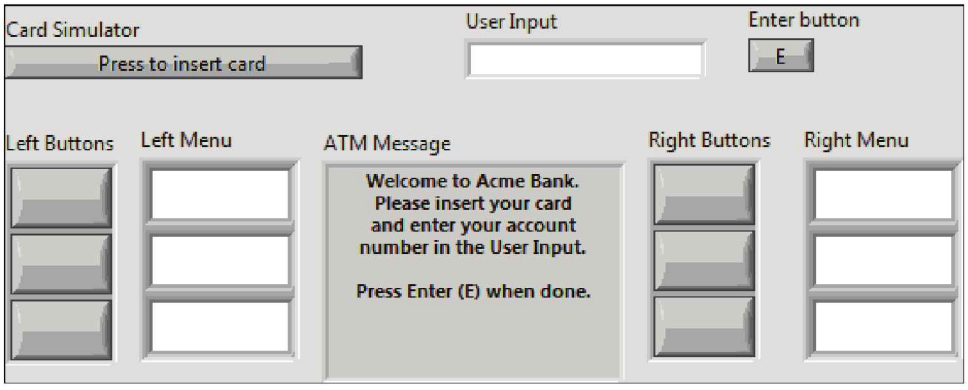


Figure F.3: Sample screen shot with Main Menu displayed

ATM Accounts File Specification

The file should be a text file containing ATM account information in a Comma Separated Value (CSV) format. The file should be located in the same folder as the main VI. The ATM controller should not prompt the user for a filename or prompt the user to update the file.

The account information should be in the following format: Account Number (5 digits), First Name, Last Name, and Balance.

Each account should be identified by a unique account number only. The following four records should exist to test the operation of the ATM:

```
12345,John,Doe,550
23456,Jennifer,Rodriguez,1000
34567,Bryan,Smith,750
45678,Julie,Ramirez,900
```



Problem Set 1 Solutions

Exercise 1.3

There are more than one possible lists of states for this problem, but most lists would include the following states. Most state machines include *Initialize* and *Shutdown* states for starting and stopping the state machine. Looking at the two tables in the functional specification, there are eight unique light combinations listed. These are natural choices for states in the state machine. A good way to name the states is to look at each unique light combination and pick out a unique element. For example, "N-S Green" for the combination of north and south green and east and west red. The eight light states would be: *N-S Green*, *N-S Yellow*, *E-W Green*, *E-W Yellow*, *N-S Left Green*, *N-S Left Yellow*, *E-W Left Green* and *E-W Left Yellow*.

Exercise 1.4

One possible state diagram for the traffic light controller is shown in Figure G.1. The state diagram shows that there are four possible loops through the states. One loop occurs if there are no cars at the vehicle sensors, a second loop occurs if there are vehicles present at the North-South vehicle sensors, the third loop occurs if there are vehicles at the East-West vehicle sensors, and the fourth loop occurs if there are vehicles at both the North-South and East-West sensors. The decision points occur at the *North-South Yellow* and *East-West Yellow* states. The diagram also shows that a continuous check is made to see if the STOP button has been pressed or to see if an error has occurred.

Exercise 1.5

The block diagram of the `Bundle Sensors` sub-VI is shown in Figure G.2. The result of the logical **`Or`** of the NORTH SENSOR and SOUTH SENSOR switches is bundled into a sensor cluster, overwriting the value of the N-S SENSORS Boolean. The result of the logical **`Or`** of the EAST SENSOR and WEST SENSOR switches is bundled into a sensor cluster, overwriting the value of the E-W SENSORS Boolean. The **`Bundle By Name`** function is used rather than the **`Bundle`** function to improve readability. The cluster

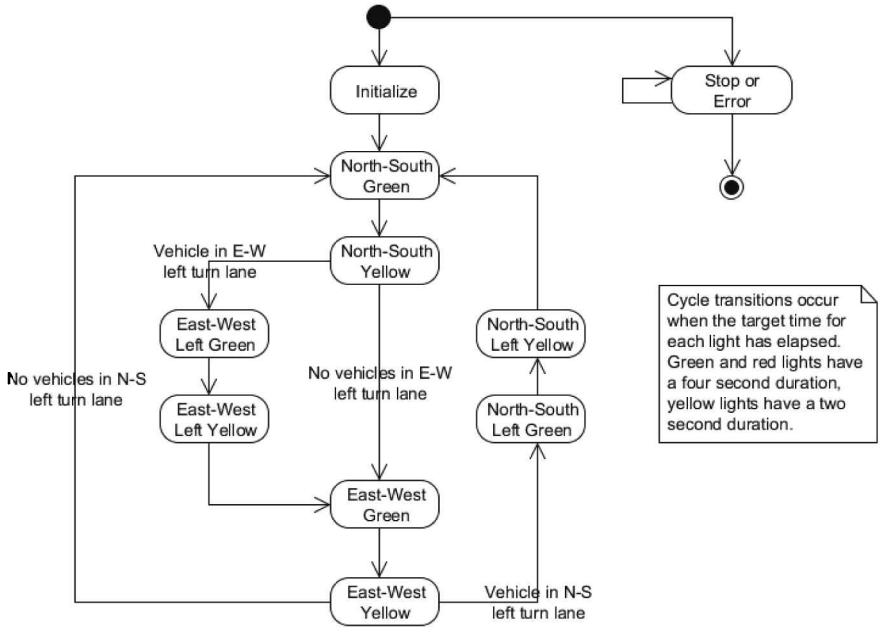


Figure G.1: State diagram for the `Traffic Light Controller` VI

constant feeding the central input of the **`Bundle By Name`** function is viewed as an icon, also to improve readability.

Exercise 1.6

If the current light state is not done, the`Next Traffic State` sub-VI passes the CURRENT STATE value through to the NEXT STATE output as seen in Figure G.3. If the current state is done, then the sub-VI selects the next state using a case structure keyed to the CURRENT STATE input. Usually the state transition is fixed, as seen in the case for the *N-S Green* state seen in Figure G.4. But when the non-left turn yellow lights are lit, the next state is determined by the vehicle sensors. This is shown in Figure G.5. The **`Select`** function is used to choose the next state based on the value of the appropriate vehicle sensors.

Exercise 1.7

The list of states will vary depending on the level of abstraction you choose. At a high level of abstraction, only three states are needed. Like most state machines, the security system will likely include an *Initialization* state and a *Shutdown* state. The system will also need a *Monitor* state in which the TAMPER, BYPASS and ALARM switches are monitored.

Other states could be added, such as a state for writing incident reports to the log file. In the solutions provided in this book the logging will be included in the *Monitor* state.

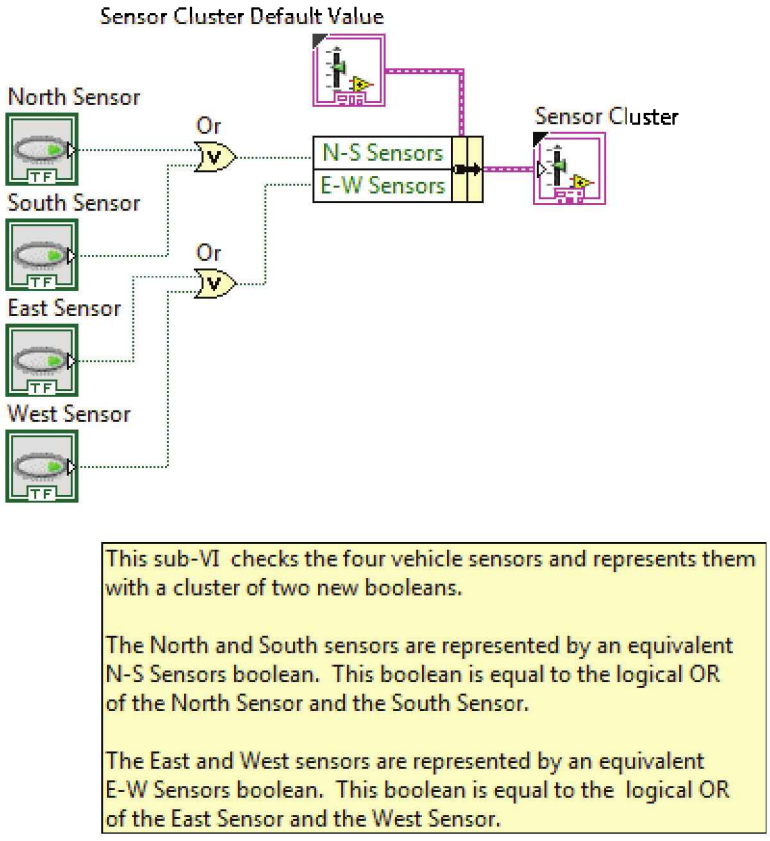


Figure G.2: Block diagram of the `Bundle Sensors` VI

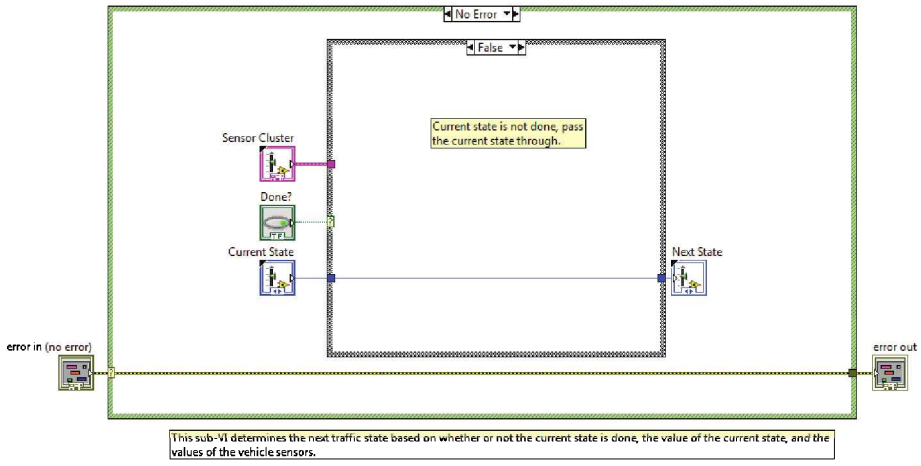


Figure G.3: The `Next Traffic State` block diagram passing the current cycle through

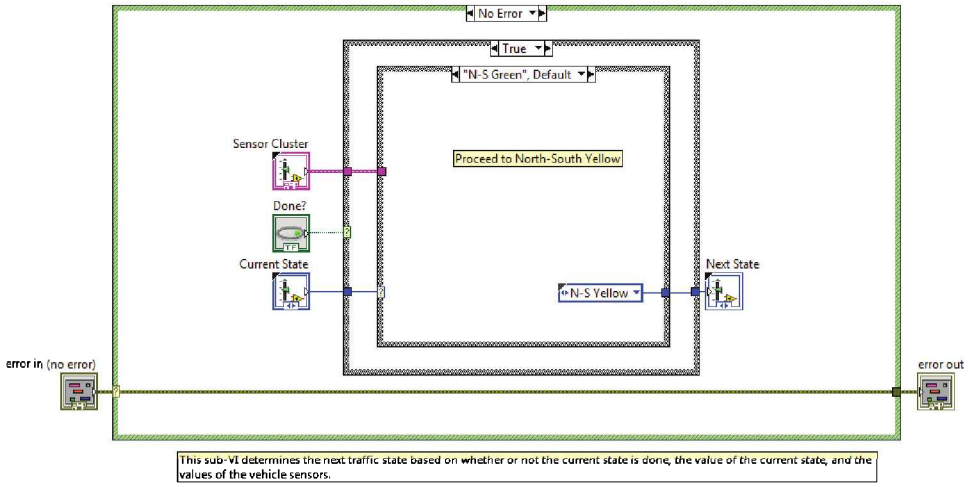


Figure G.4: The `Next Traffic State` block diagram for state with a fixed state transition

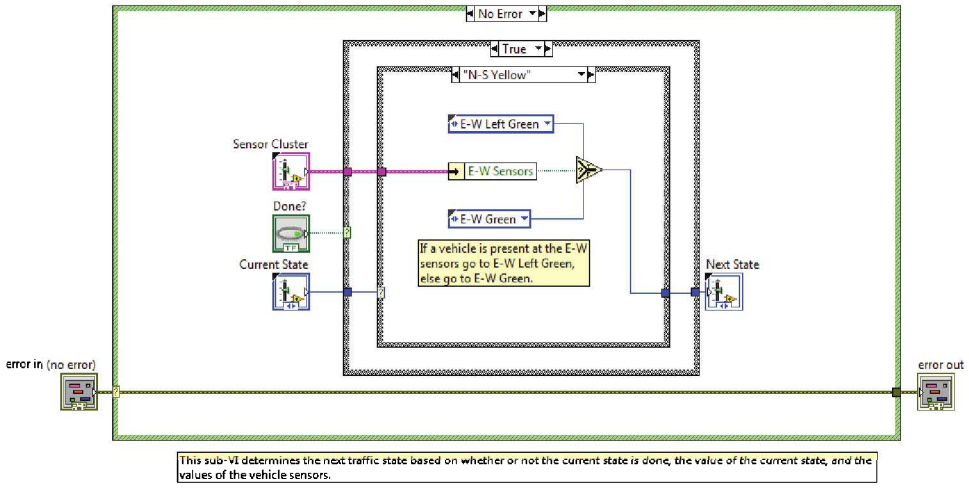


Figure G.5: The `Next Traffic State` block diagram for state with a variable state transition

Exercise 1.8

A high-level state diagram with only the three states described in Exercise 1.7 would look something like the one in Figure G.6. A lower-level state diagram would look something like the one in Figure G.7. There is more detail in the lower level diagram. The lower level diagram breaks the monitoring step into smaller steps.

One of the interesting things about the security system program is that triggering a switch does not necessarily mean that the display and log file are updated since the switch types have a priority ordering. For example, if a zone is currently in bypass an alarm in that zone is ignored. In that case the program would resume monitoring

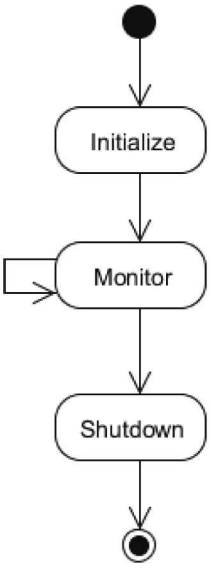


Figure G.6: High-level state diagram for the `Security System Controller` VI

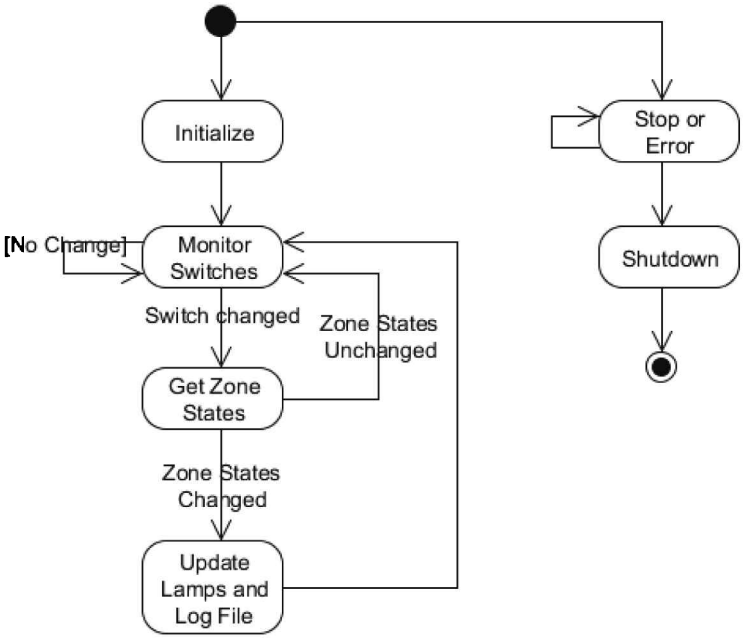


Figure G.7: Low-level state diagram for the `Security System Controller` VI

without changing the indicators and without writing to the log file. If an alarm is triggered when the system is not in bypass, then the front panel and log file would be updated before returning to the monitoring state.

Both of these diagrams are helpful. In the implementation developed later, the organization of the actual states in the program matches that in Figure G.6. But the logic of the program is more easily understood using Figure G.7.

Exercise 1.10

One possible implementation of the `Initialize Status Array From Cluster` sub-VI is shown in Figure G.8. The key to this exercise is determining the number of elements in the switches cluster. This is done by first converting the cluster to an array using **`Cluster To Array`** function. The size of the resulting array is determined using the **`Array Size`** function. This size is then passed to the **`Initialize Array`** function along with a type-defined enumerated zone status cluster as the array element. The output of this function is an array of zone status enumerated constants with the same number of elements as the input cluster.

Exercise 1.11

Figure G.9 shows one possible implementation of the `Reset Zone Array`. The input array is passed as an auto-indexed input to a **`For`** loop. The loop runs for a number

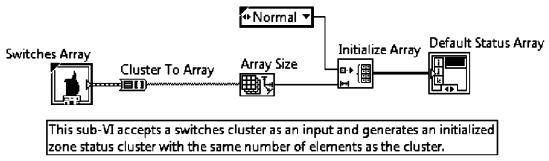


Figure G.8: The block diagram for the `Initialize Status Array From Cluster` sub-VI

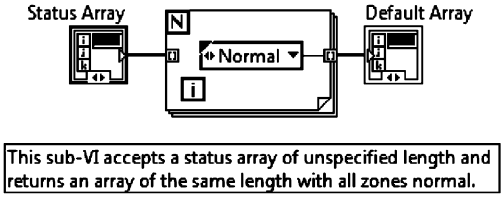


Figure G.9: The block diagram for the `Reset Zone Array` sub-VI

iterations equal to the size of the input array. A zone status enumerated constant set to *Normal* is placed within the loop and is passed to an auto-indexed output. This creates an output array of *Normal* enumerated constants with the same size as the input array.

Exercise 1.12

One possible implementation of the `Get Zone Status` sub-VI is shown in Figure G.10. This sub-VI uses a **`Select`** ladder to implement the desired logic. Starting from the top of the ladder, if TAMPER is True, the output is *Tamper*. If TAMPER is False and BYPASS is True, the output is *Bypass*. If both TAMPER and BYPASS are False and ALARM is True, the output is *Alarm*. If all three inputs are False, the output is *Normal*.

Exercise 1.13

The block diagram of the `Get Status Of All Zones` sub-VI is shown in Figure G.11. The three clusters of switches are converted to arrays using the **`Cluster To Array`** function. The arrays are then passed to auto-indexing inputs of a **`For`** loop. The switches for each zone are then passed to the `Get Zone Status` sub-VI developed in Exercise 1.12. The zone status outputs of this sub-VI are collected into an array by an auto-indexed output tunnel. The use of auto-indexed inputs and outputs ensures that the output status array will have the same number of elements as the input switch clusters. The use of auto-indexing also makes the sub-VI scalable. The sub-VI will work for clusters with any number of switches.

Exercise 1.14

One possible implementation of the `Write To Log File` sub-VI is shown in Figure G.12. The timestamp is formatted into a date string and a time string with seconds by the **`Get Date/Time String`** function. The resulting strings are passed to the **`Format Into String`** function along with the zone number and the zone status enumerated constant.

The **`Format Into String`** function needs to format these four inputs into a single comma-separated string followed by a carriage return or newline. This is accomplished by a format statement wired to the function. The format statement used in this sub-VI is `%s, %s, %d, %s`. This means that the output string will consist of the date string, a comma, the time string, a comma, the zone number, a comma, and the

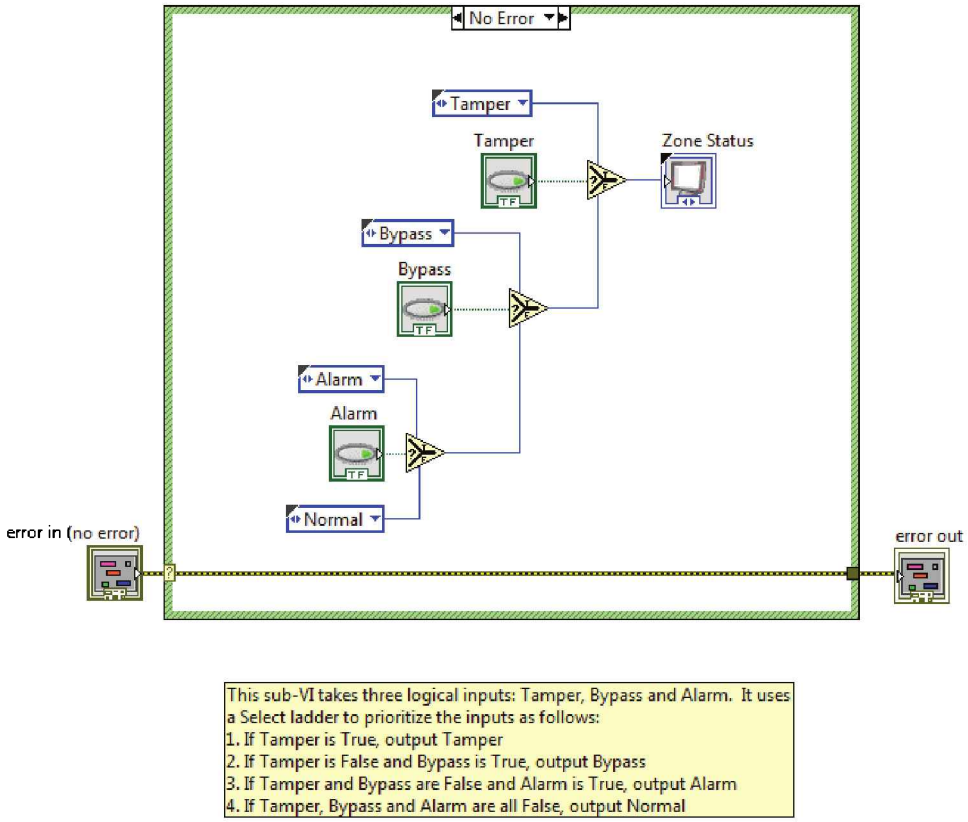


Figure G.10: The block diagram for the `Get Zone Status` sub-VI

status string. Notice that the format token for the enumerated constant is `%s`. Recall that an enumerated constant pairs strings with sequential integers. By specifying `%s` in the format statement we are requesting the string value associated with the enumerated constant. If we had specified `%d` instead we would be requesting the integer value of the enumerated constant. The output of the **`Format Into String`** is then written to the log file using the `Append To Text File` function shown in Figure 1.45. The `Append To Text File` sub-VI appends the string to the end of the file and adds a carriage return.

Exercise 1.15

The boiler controller functional specification lists a series of steps the boiler controller takes during normal operation. Each of these steps can be represented by a single state in the boiler controller. Including startup and shutdown code, a possible list of states is: *Initialize*, *Application Run*, *Reset Lockout*, *Pre-Purge Timer*, *Pre-Purge*, *Ignition*, *Prove The Pilot*, *Start*, *Run*, *Purge* and *Shutdown*.

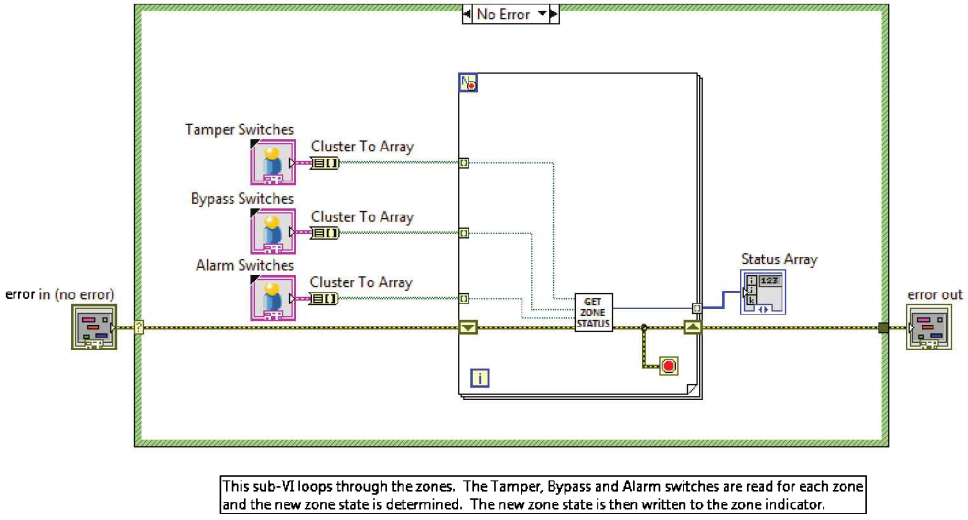


Figure G.11: The block diagram for the `Get Status Of All Zones` sub-VI

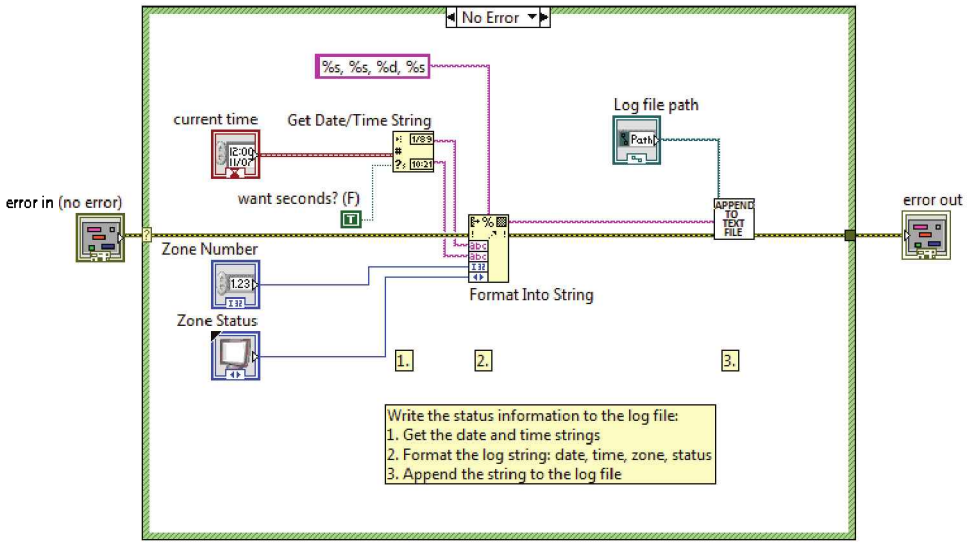


Figure G.12: The block diagram for the `Write To Log File` sub-VI

Exercise 1.16

The state transition diagram for the boiler controller is straightforward since the sequence of states is very linear. The state transitions are fixed, each state either transitions back to itself or to one other fixed state. A possible state transition diagram is shown in Figure G.13.

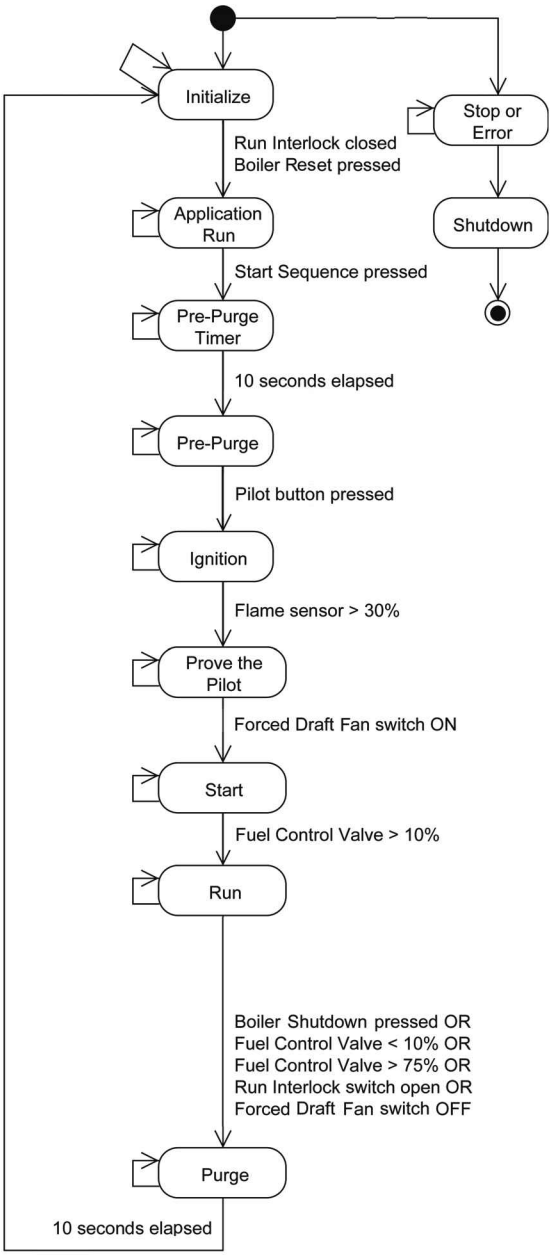


Figure G.13: State diagram for the `Boiler Controller` VI

Exercise 1.18

The block diagram of one possible implementation of the `Application Run Done` sub-VI is shown in Figure G.14. The BOILER RESET control is unbundled from the BOILER CONTROLS cluster and the RUN INTERLOCK control is unbundled from the BOILER SIMULATION CONTROLS cluster. Both operations us the **`Unbundle By Name`** function to improve readability. The problem statement says that DONE should be True if and only if both input controls are True. This means that DONE is the result of the logical And of the two controls, so the **`And`** function is used to assign a value to the DONE indicator.

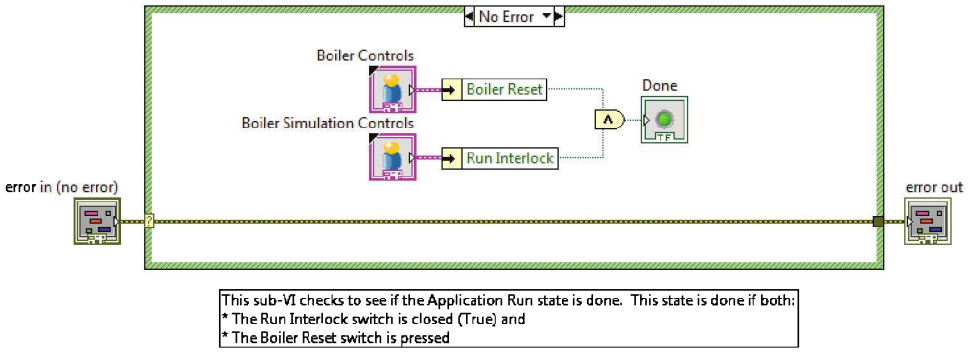


Figure G.14: The block diagram of the `Application Run Done` VI

Exercise 1.19

The block diagram of one possible implementation of the `Application Run Output` sub-VI is shown in Figure G.15. This sub-VI bundles the string `Lockout` into the BOILER STATUS cluster using the **`Bundle By Name`** function for increased readability.

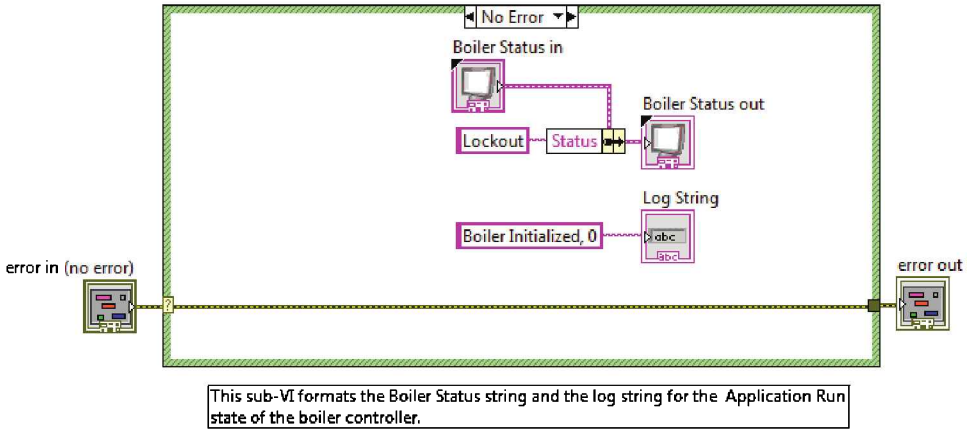


Figure G.15: The block diagram of the `Application Run Output` VI

Since the requested output string is constant, a string constant is used to output `Boiler Initialized, 0` to the LOG STRING indicator.

Exercise 1.20

The block diagram of one possible implementation of the `Reset Lockout Done` sub-VI is shown in Figure G.16. This very simple sub-VI uses the **`Unbundle By Name`** function to access the START SEQUENCE control from the BOILER CONTROLS cluster and assigns its value to the DONE output.

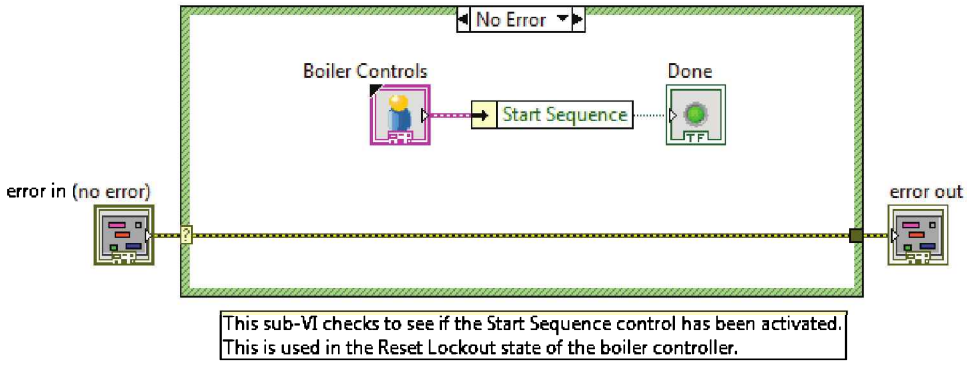


Figure G.16: The block diagram of the `Reset Lockout Done` VI

Exercise 1.21

The block diagram of one possible implementation of the `Reset Lockout Output` sub-VI is shown in Figure G.17. The `Ready` string is bundled into the BOILER STATUS

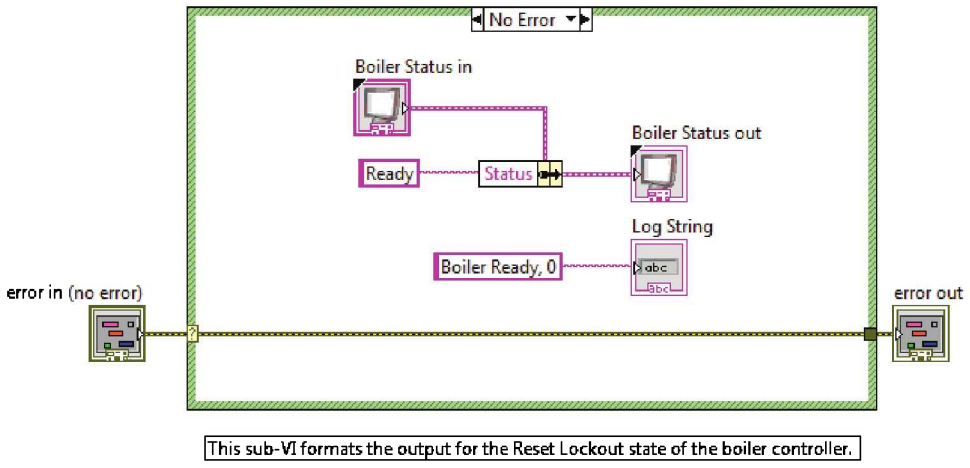


Figure G.17: The block diagram of the `Reset Lockout Output` VI

cluster. Since the output log string is constant, a string constant is used to output the the `Boiler Ready, 0` log string.

Exercise 1.22

The block diagram of one possible implementation of the `Round To Nearest Tenths Place` sub-VI is shown in Figure G.18. The input value is multiplied by ten, rounded to the nearest integer using the **`Round To Nearest`** function, and then divided by ten to provide the rounded result.

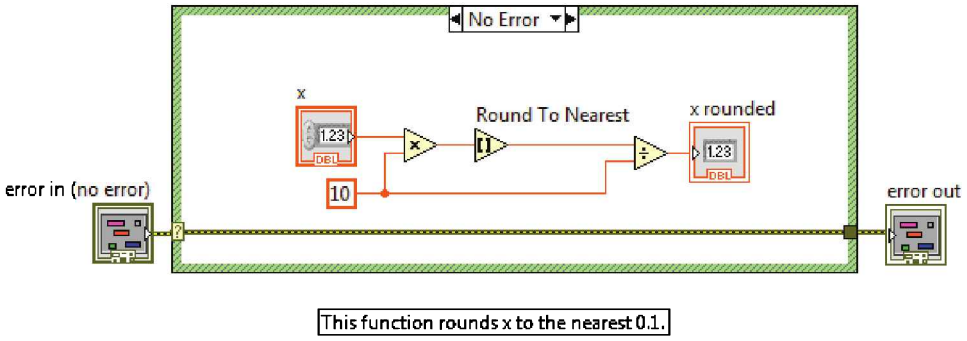


Figure G.18: The block diagram of the `Round To Nearest Tenths Place` VI

Exercise 1.23

The block diagram of one possible implementation of the `Pre-Purge Timer Done` sub-VI is shown in Figure G.19. The value of the ELAPSED TIME control is rounded to the nearest tenths place and is then compared to the PRE-PURGE TIME (S) control that is unbundled from the LIMITS cluster using the **`Greater?`** function. The rounded elapsed time and the DONE Boolean are passed as outputs.

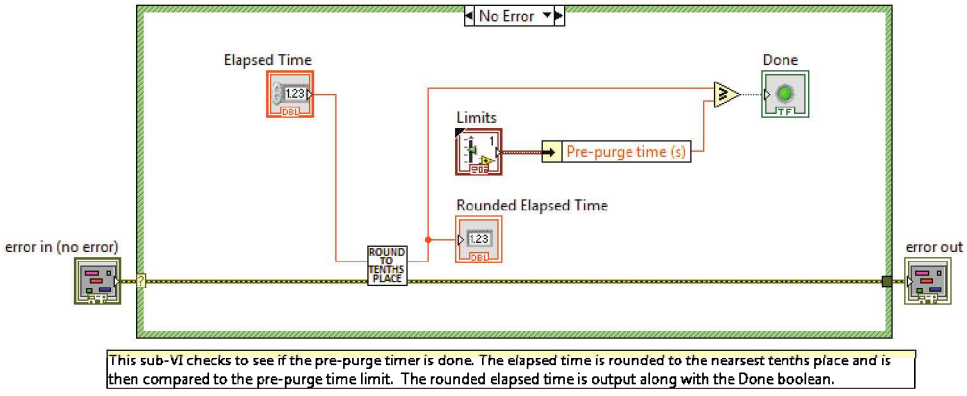


Figure G.19: The block diagram of the `Pre-Purge Timer Done` VI

Exercise 1.24

The block diagram of one possible implementation of the `Pre-Purge Timer Output` sub-VI is shown in Figure G.20. The specified values are bundled into the BOILER STATUS cluster using the **`Bundle By Name`** function.

A **`Select`** function is used to decide which string will be used in the log string. The string is then sent to the **`Format Into String`** function along with the value of the ELAPSED TIME numeric control. The **`Format Into String`** function converts this numeric value into a string when it creates the log string. The number of decimals represented in the output string is controlled by the format statement wired to the top input of the function. The format string `%s, %.1f` means that the output will consist of the input string followed by a comma, followed by a space, followed by a floating-point number with one digit past the decimal point.

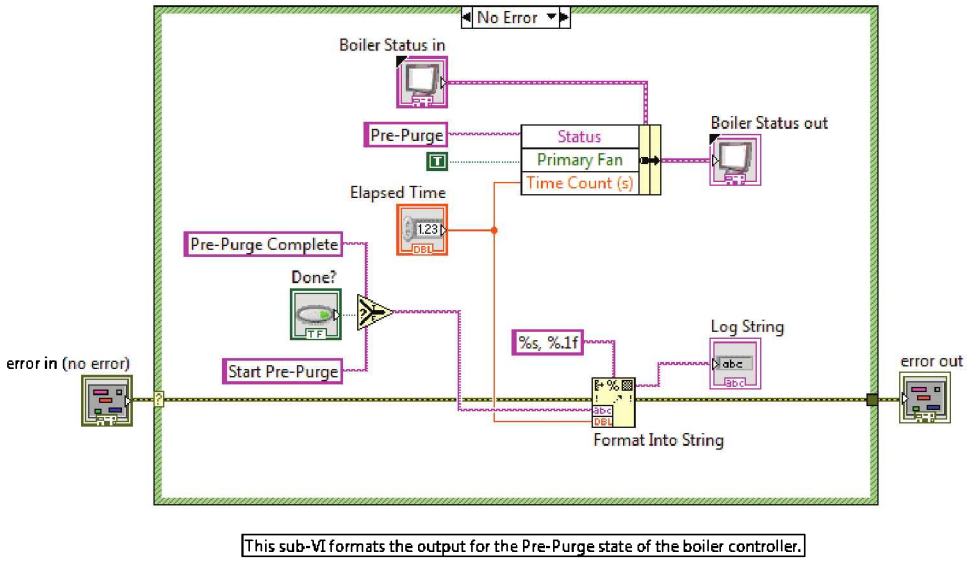


Figure G.20: The block diagram of the `Pre-Purge Timer Output` VI

Exercise 1.25

The block diagram of one possible implementation of the `Pre-Purge Done` sub-VI is shown in Figure G.21. This very simple sub-VI simply uses the **`Unbundle By Name`** function to access the PILOT BUTTON control from the BOILER CONTROLS cluster and assigns its value to the DONE output.

Exercise 1.26

The block diagram of one possible implementation of the `Pre-Purge Output` sub-VI is shown in Figure G.22. This sub-VI bundles the string `Pre-Purge Complete` into the STATUS string of the BOILER STATUS cluster.

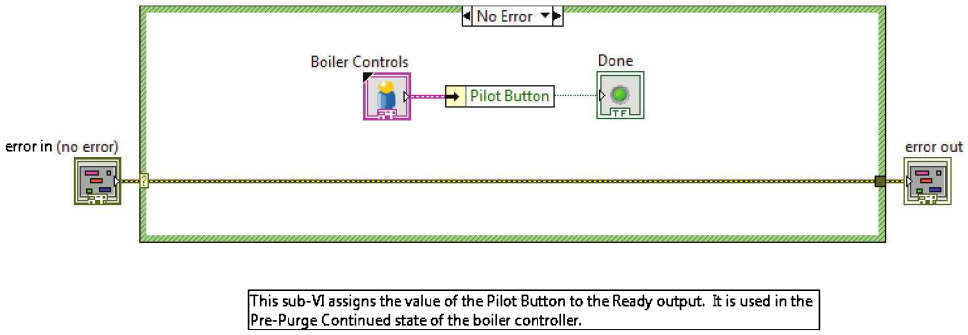


Figure G.21: The block diagram of the `Pre-Purge Done` VI

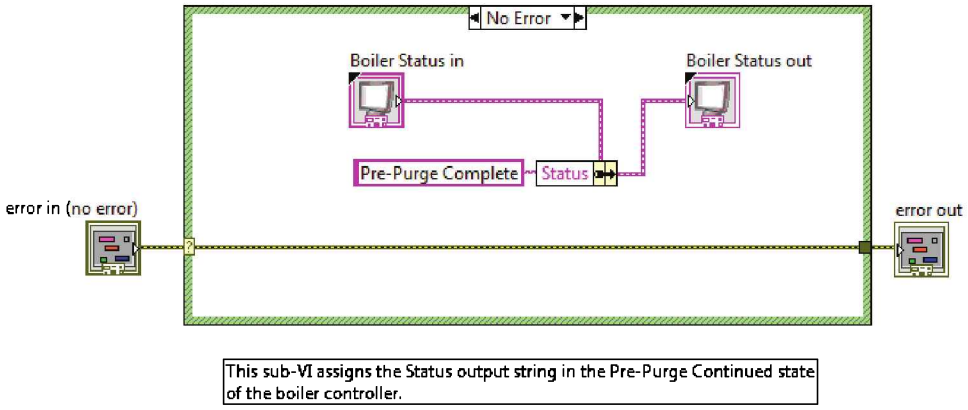


Figure G.22: The block diagram of the `Pre-Purge Output` VI

Exercise 1.27

The block diagram of one possible implementation of the `Ignition Done` sub-VI is shown in Figure G.23. The FLAME SENSOR VALUE (%) control is unbundled from the BOILER SIMULATION CONTROLS cluster and is compared to the FLAME SENSOR LOW control unbundled from the LIMITS cluster.

Exercise 1.28

The block diagram of one possible implementation of the `Ignition Output` sub-VI is shown in Figure G.24. This sub-VI uses the **`Bundle By Name`** function to bundle the desired values into the BOILER STATUS cluster. Since the output log string is constant, a string constant is used to output the the `Pilot ON, True` log string.

Exercise 1.29

The block diagram of one possible implementation of the `Prove The Pilot Done` sub-VI is shown in Figure G.25. The FORCED DRAFT FAN control is unbundled from the

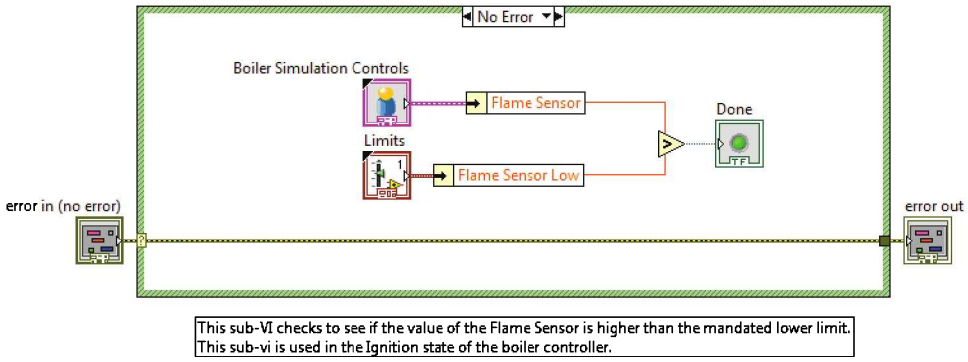


Figure G.23: The block diagram of the `Ignition Done` VI

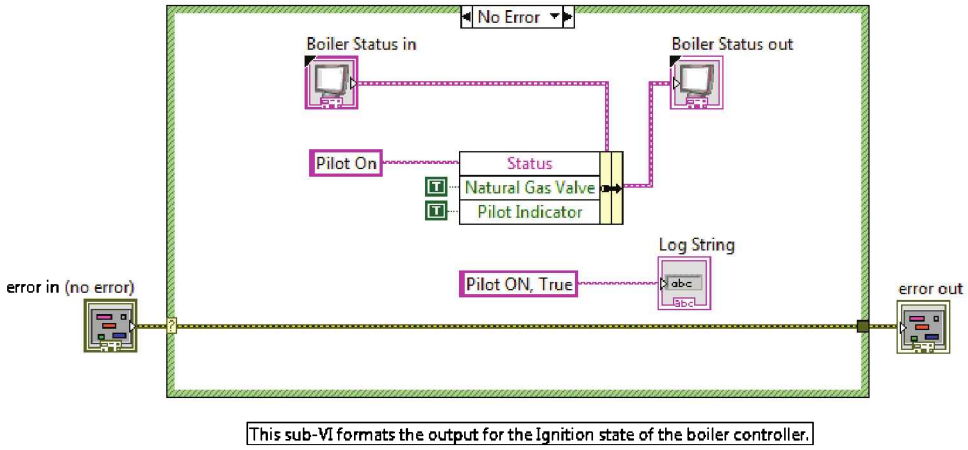


Figure G.24: The block diagram of the `Ignition Output` VI

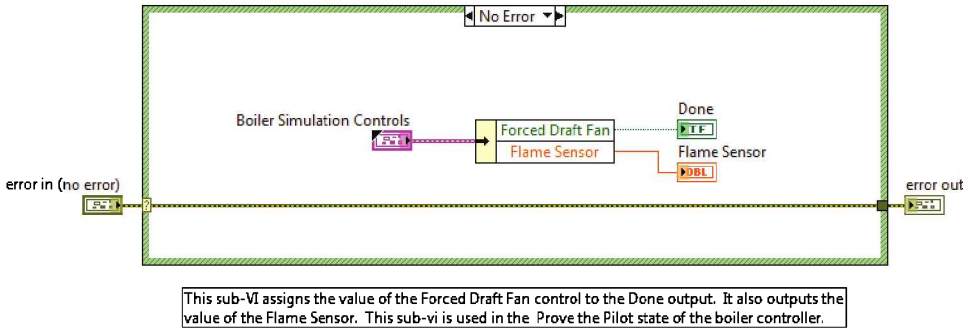


Figure G.25: The block diagram of the `Prove The Pilot Done` VI

BOILER SIMULATION CONTROLS cluster and is assigned to the DONE output. The FLAME SENSOR VALUE (%) control is also unbundled and is passed as an output.

Exercise 1.30

The block diagram of one possible implementation of the `Prove The Pilot Output` sub-VI is shown in Figure G.26. The specified values are bundled into the BOILER STATUS cluster using the **`Bundle By Name`** function.

The **`Format Into String`** function creates the log string. The number of decimals of the FLAME SENSOR value included in the output string is controlled by the format statement wired to the top input of the function. The format string `%s, %.1f` means that the output will consist of the input string followed by a comma, followed by a space, followed by a floating-point number with one digit past the decimal point.

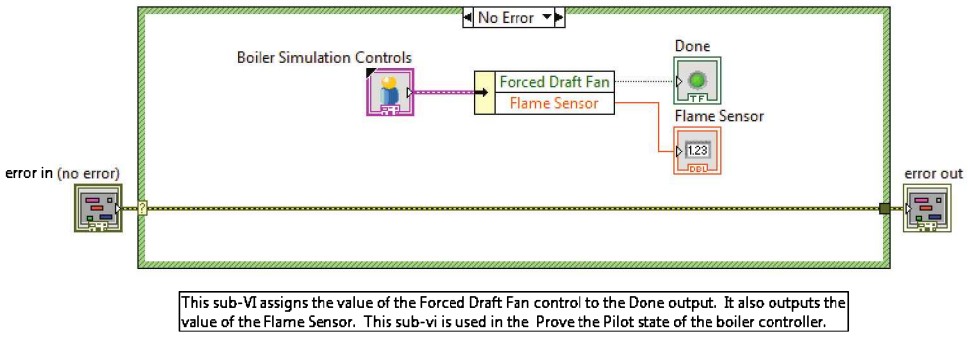


Figure G.26: The block diagram of the `Prove The Pilot Output` VI

Exercise 1.31

The block diagram of one possible implementation of the `Start Done` sub-VI is shown in Figure G.27. This sub-VI unbundles the fuel control high and low limits

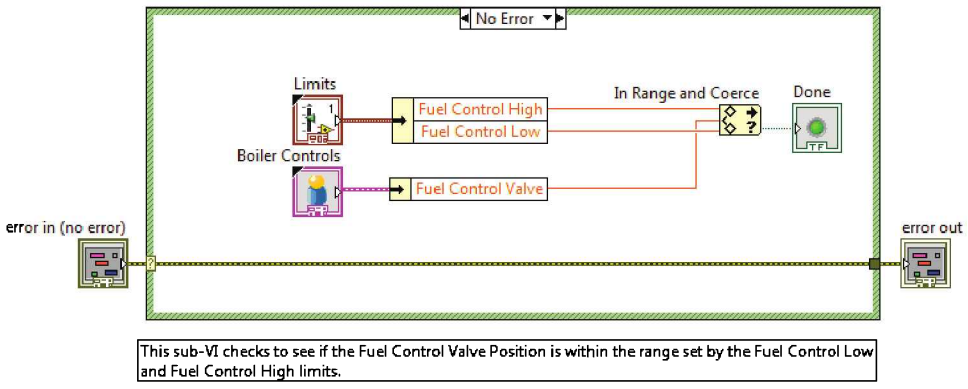


Figure G.27: The block diagram of the `Start Done` VI

and the current fuel-control valve position from the input clusters. It then uses the **In Range and Coerce** function to see if the current fuel control valve position is in the prescribed range.

Exercise 1.32

The block diagram of one possible implementation of the `Start Output` sub-VI is shown in Figure G.28. The specified values are bundled into the BOILER STATUS cluster using the **Bundle By Name** function. Since the output log string is constant, a string constant is used to output the the `Forced Draft Fan ON, True` log string.

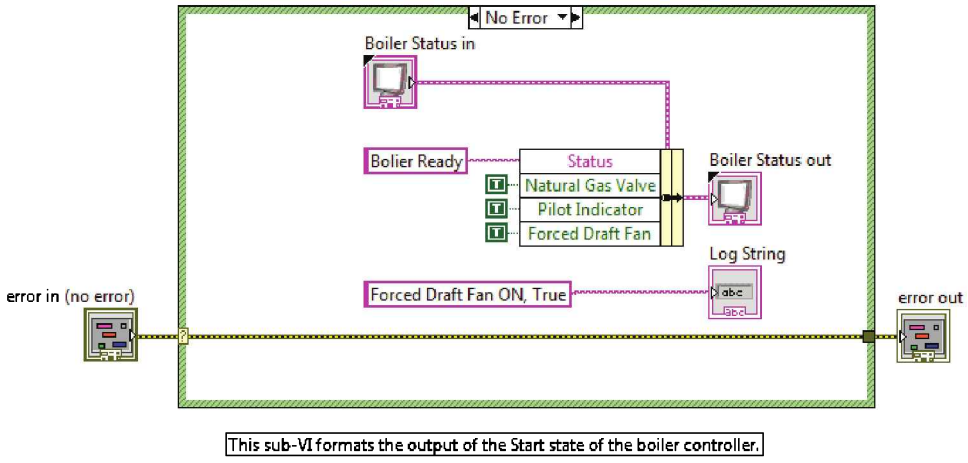


Figure G.28: The block diagram of the `Start Output` VI

Exercise 1.33

The block diagram of one possible implementation of the `Run Done` sub-VI is shown in Figure G.29. This sub-VI unbundles the fuel control high and low limits and the current fuel control valve position from the input clusters. It also unbundles the SHUTDOWN, RUN INTERLOCK and FORCED DRAFT FAN buttons. The sub-VI uses a compound **Compound Arithmetic** function configured to `Or` mode to check to see if any of the four specified conditions are satisfied.

Exercise 1.34

The block diagram of one possible implementation of the `Run Output` sub-VI is shown in Figure G.30. The specified values are bundled into the BOILER STATUS cluster using the **Bundle By Name** function.

The **Format Into String** function creates the log string. The number of decimals of the FUEL CONTROL VALVE value included in the output string is controlled by the format statement wired to the top input of the function. The format string `%s, %.1f`

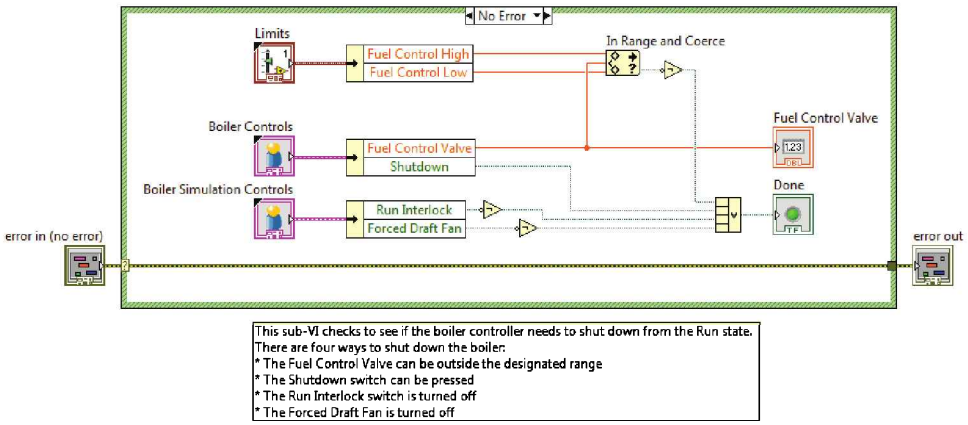


Figure G.29: The block diagram of the `Run Done` VI

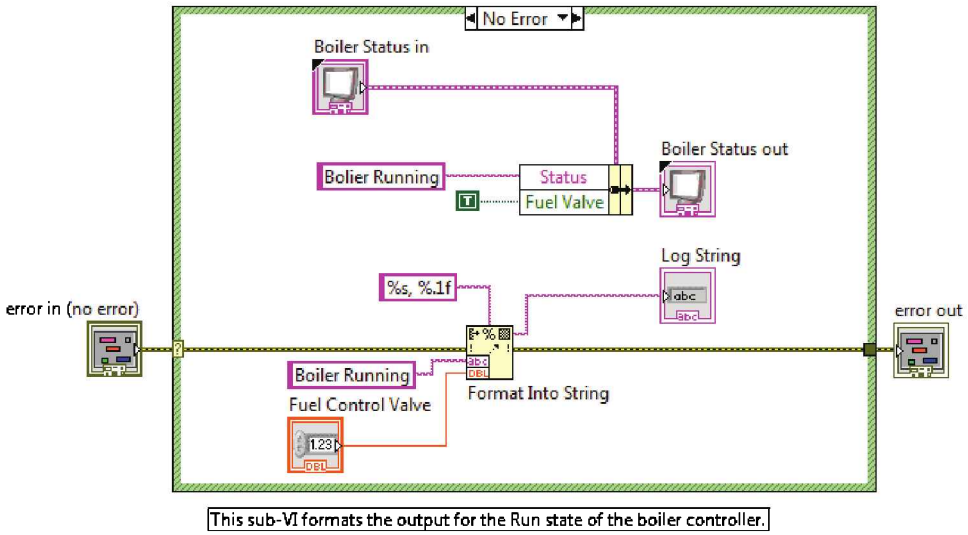


Figure G.30: The block diagram of the `Run Output` VI

means that the output will consist of the input string followed by a comma, followed by a space, followed by a floating-point number with one digit past the decimal point.

Exercise 1.35

The block diagram of one possible implementation of the `Purge Done` sub-VI is shown in Figure G.31. This sub-VI rounds the elapsed time to the nearest tenths place using the `Round to Nearest Tenths Place` sub-VI developed in Exercise 1.22. It then compares the rounded elapsed time to the value of the PURGE TIME (S) control

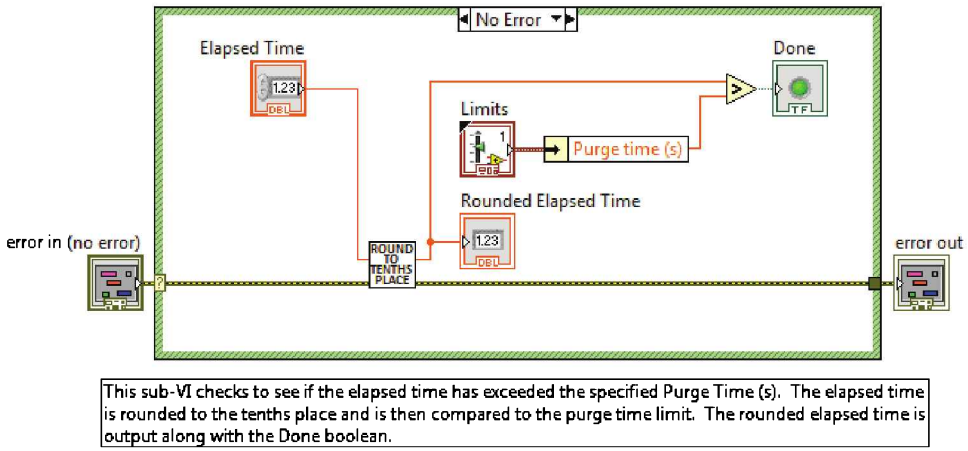


Figure G.31: The block diagram of the `Purge Done` VI

of the LIMITS cluster using the **`Greater?`** function and assigns the output to the DONE Boolean indicator. The rounded elapsed time is passed as an output as well.

Exercise 1.36

The block diagram of one possible implementation of the `Purge Output` sub-VI is shown in Figure G.32. The specified values are bundled into the BOILER STATUS cluster using the **`Bundle By Name`** function.

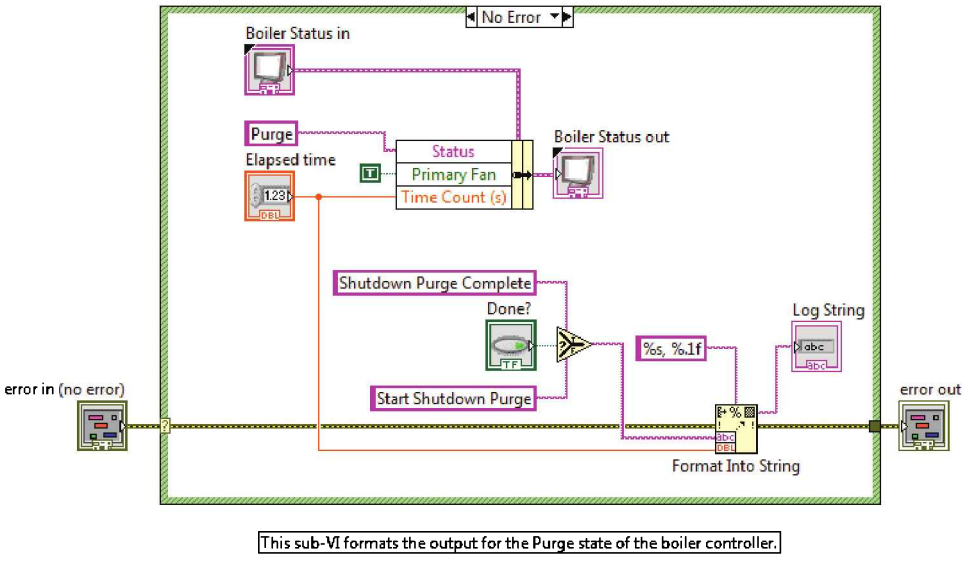


Figure G.32: The block diagram of the `Purge Output` VI

A **Select** function is used to decide which string will be used in the log string. The string is then sent to the **Format Into String** function along with the value of the ELAPSED TIME numeric control. The **Format Into String** function converts this numeric value into a string when it creates the log string. The number of decimals represented in the output string is controlled by the format statement wired to the top input of the function. The format string `%s, %.1f` means that the output will consist of the input string followed by a comma, followed by a space, followed by a floating-point number with one digit past the decimal point.

Exercise 1.37

Examination of the functional specification leads to a number of obvious states. There will be *Initialize* and *Shutdown* states to allow the state machine to initialize front panel controls and to exit gracefully. The functional specification calls for the WASH OPTIONS control to be disabled during an active car wash, so the *Enable Options* and *Disable Options* states are called for. The state machine must wait while the options are being selected, so a *Wait* state would be helpful. The state machine must wait for the car to enter and to leave the car wash, so this suggests that a *Car Entry* and a *Car Exit* state would be useful. Each of the wash options will merit a separate state: *Under Body Wash*, *Bug Remover*, *Pre-Soak*, *High Pressure Wash*, *Low Pressure Wax*, *Spot Free Rinse*, *Tire Shine Foam*, and *Air Dry*. Finally, the selected options must be assembled into some kind of sequence for the state machine to follow. In the implementation, we will develop here the options are collected in an array, so a *Build Option Array* state will be needed.

Exercise 1.38

One possible state diagram for the Car Wash 2 controller is shown in Figure G.33. The code starts in the *Initialize* state and moves to the *Enable Options* state where the wash options buttons are enabled. The *Wait* state comes while the user selects options. When the user presses the START button, the state machine transitions to the *Disable Options* state to prevent the user from entering options during an active wash. The options are then collected into an array in the *Build Option Array* state. The state machine waits for the car to enter the car wash. The state machine works its way through the states in the array of options built in the *Build Option Array* states. Since the sequence of options is variable, the entire sequence is represented on the state diagram as a single large state. When the options are exhausted, the state machine waits for the car to exit in the *Car Exit* state. The state machine then starts over again at the *Initialize* state. For every loop iteration, an error check is also made and the STOP button is polled. If either an error occurs or a stop request is made, the state machine immediately transitions to the *Shutdown* state.

Exercise 1.39

One possible implementation of the `Car In Position` sub-VI is shown in Figure G.34. This implementation is made particularly simple by the **Type Cast** function. The **Type Cast** function is used to cast the slider to a type-defined

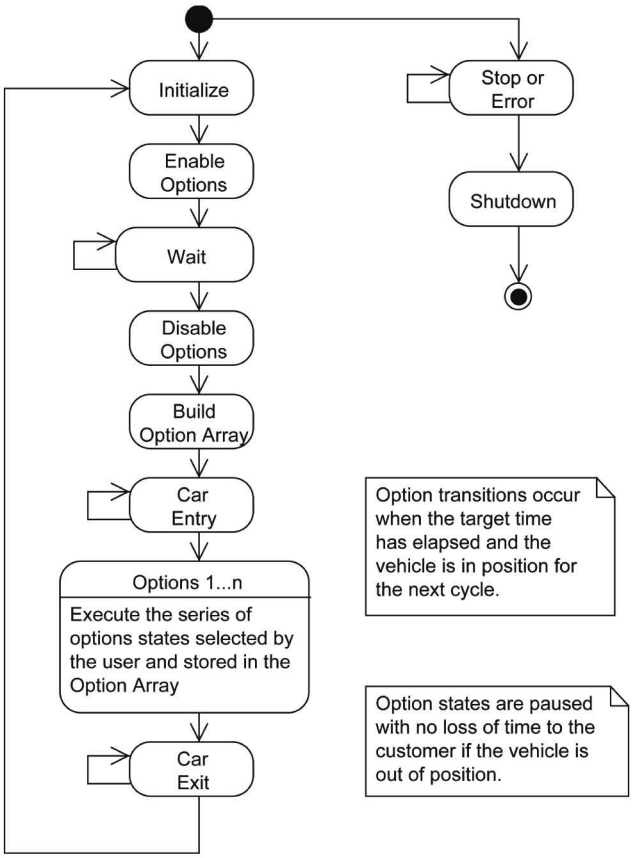


Figure G.33: State transition diagram for the Car Wash 2 controller

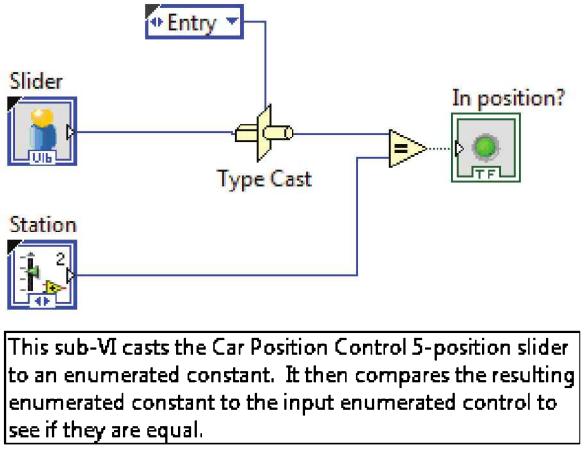


Figure G.34: The block diagram of the `Car In Position` VI

enumerated constant containing the slider text labels. This allows the slider output to be directly compared with a second type-defined enumerated constant without coercion dots.

Exercise 1.40

One possible implementation of the `Build Option Array` sub-VI is shown in Figure G.35. This sub-VI accepts the WASH OPTIONS cluster containing the values of the front panel option buttons. The first step taken is to bundle a True value for the HIGH PRESSURE WASH button. This will ensure that this option is included in the final OPTION ARRAY output even if this option was not selected by the user. The cluster is then converted to an array of Boolean values using the **`Cluster To Array`** function.

The array of Booleans is then wired to an auto-indexed tunnel of a **`For`** loop. An array constant is constructed consisting of wash option state enumerated constants. The array contains all of the car wash option states in the same order that they are listed in the WASH OPTIONS cluster and in the converted Boolean array. This array is also passed to the **`For`** loop through an auto-indexing input tunnel. A conditional output tunnel is used to filter the wash option array. The output tunnel only collects those wash option elements corresponding to the Boolean switches selected by the user.

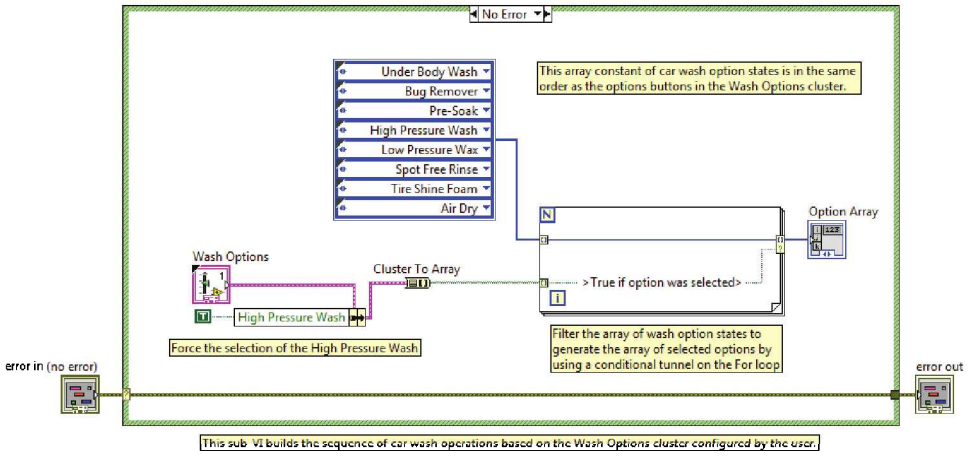


Figure G.35: The block diagram of the `Build Option Array` VI

Exercise 1.41

One possible implementation of the `Next Car Wash State` sub-VI is shown in Figure G.36. If the DONE? Boolean control is False, the current state and index are passed through as outputs. The current state is retrieved using the current index, the array of options, and the **`Index Array`** function.

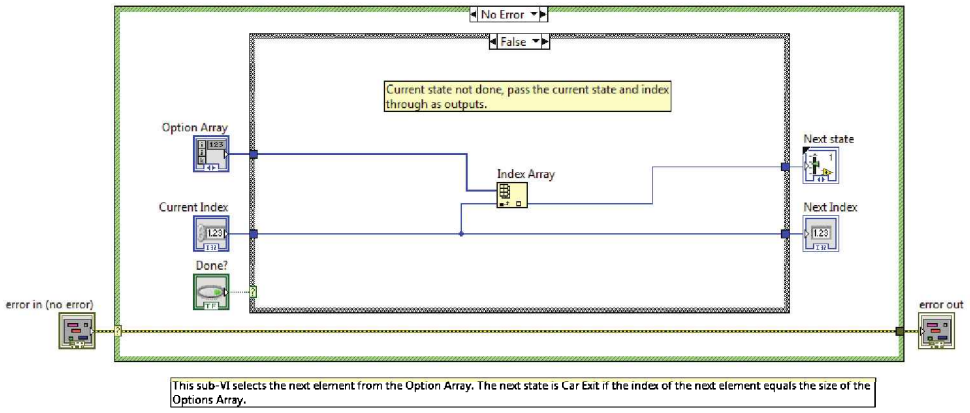


Figure G.36: The block diagram of the `Next Car Wash State` VI with the current state not done

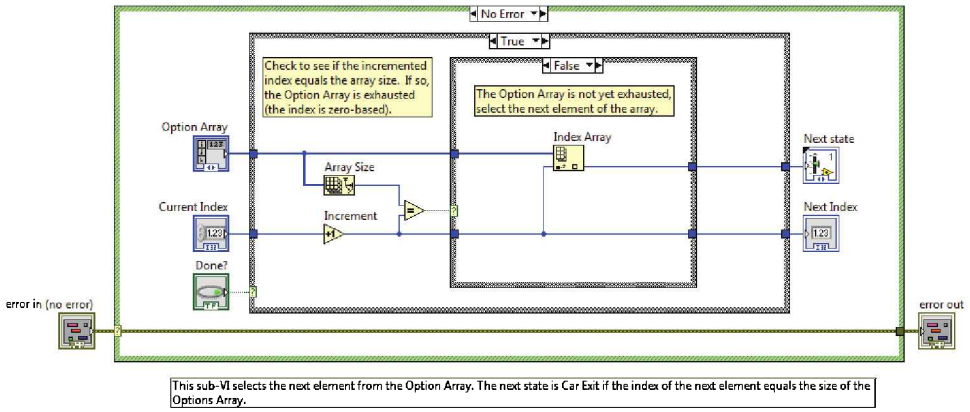


Figure G.37: The block diagram of the `Next Car Wash State` VI with the array of options not exhausted

If the DONE? Boolean control is True and the array of options is not exhausted, the index is incremented and the next state is selected from the options array as seen in Figure G.37.

The **`Array Size`** function is used to generate the array size. The array size is then compared to the incremented index. If the two are equal, the option array has been exhausted. This is due to the fact that the array index is zero-based. If there are five elements in the options array, the index of the last element is `4`. The **`Index Array`** function is used to select the next state from the array of options.

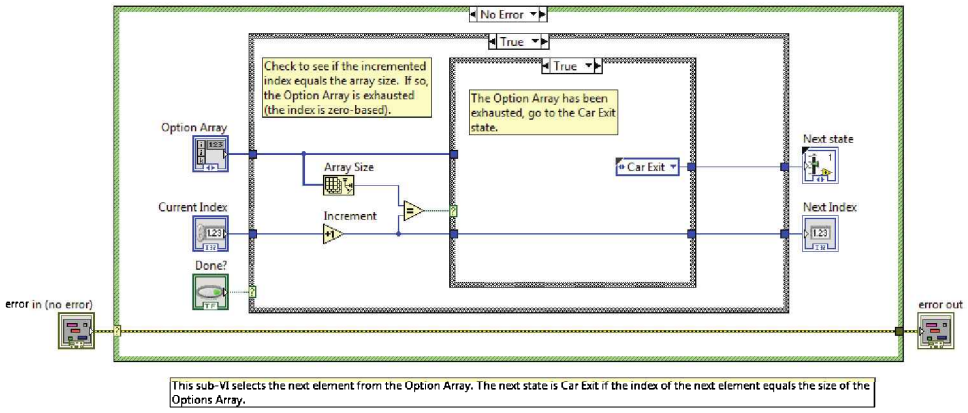


Figure G.38: The block diagram of the `Next Car Wash State` VI with the array of options exhausted

If the DONE? Boolean is True and the array of options is exhausted, the *Car Exit* state is output as shown in Figure G.38. The incremented index is also passed as an output, although this value will not be used by the main application, since the array of options has already been exhausted.



Problem Set 2 Solutions

Exercise 2.1

The block diagram of the `Car Wash 2 Timer Kernel` is shown in Figure H.1. Note the similarity with the original car wash timer kernel shown in Figure 6.8. The only difference is the addition of an indicator for the elapsed time. The *Reset* case of the case structure is unchanged, as are the method sub-VIs for performing the timecheck and resetting the timer.

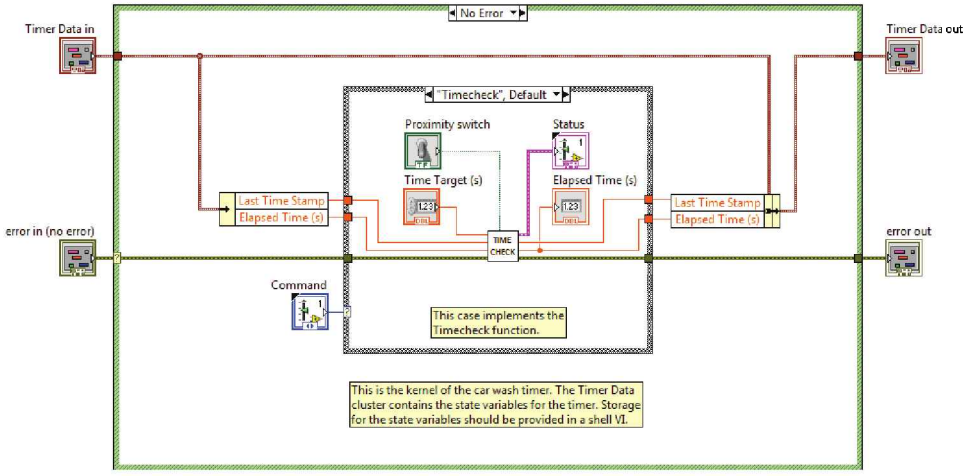


Figure H.1: The *Timecheck* case of the `Car Wash 2 Timer Kernel` sub-VI

Exercise 2.2

The block diagram of a feedback node implementation of the `Car Wash 2 Timer` sub-VI is shown in Figure H.2. This sub-VI uses a feedback node to store the state variables of the `Car Wash 2 Timer Kernel` sub-VI developed in Exercise 2.1.

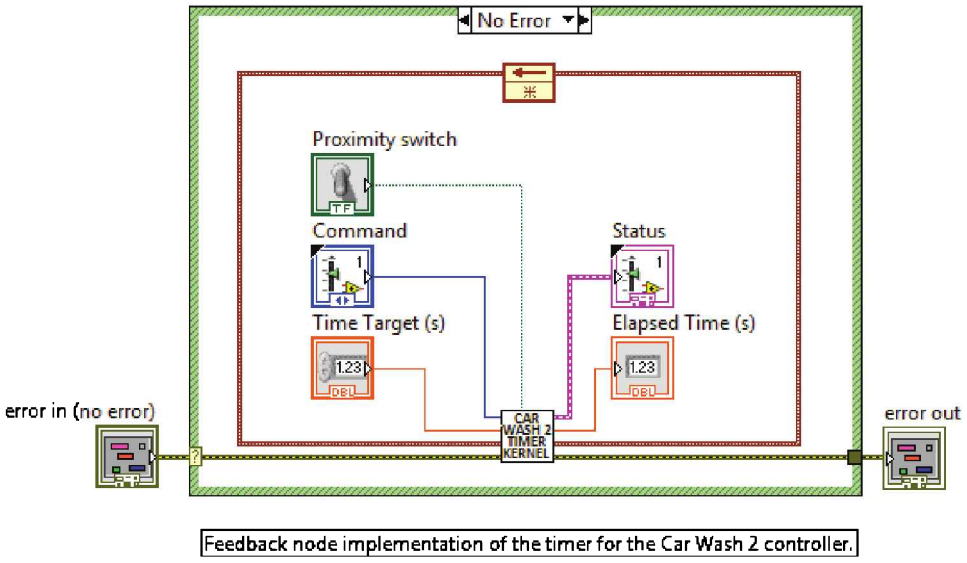


Figure H.2: The block diagram for the `Car Wash 2 Timer` sub-VI

Exercise 2.3

The `String Changed` sub-VI is an extremely simple action engine. The block diagram is shown in Figure H.3. Every time the sub-VI is called, the new value of the string is stored in the feedback node. The old value of the string is recalled from the feedback node and compared to the string input using the **`Not Equal?`** function. If the old and new strings are not equal, the Boolean output is True; otherwise, it is False.

Since the feedback node is not initialized, it will deliver the default value of the string data type when it is first called. The default of the string data type is the null string, so if the input string is the null string the first time this sub-VI is called, its output will be False. If any other string is used other than the null string, the output will be True.

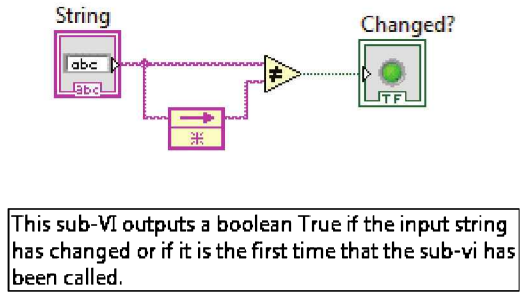


Figure H.3: The block diagram of the `String Changed` VI

Exercise 2.4

The block diagram of the `Write To Boiler Log File` sub-VI is shown in Figure H.4. This sub-VI uses the `String Changed` action engine developed in Exercise 2.3 to monitor the event string. If the string has changed, the **`Format Into String`** function is used to add the time stamp in the correct format. The string is then appended to the log file specified by the path control using the `Append To Text File` sub-VI shown in Figure 1.45. Nothing is written to the file if the event string string has not changed, as seen in Figure H.5.

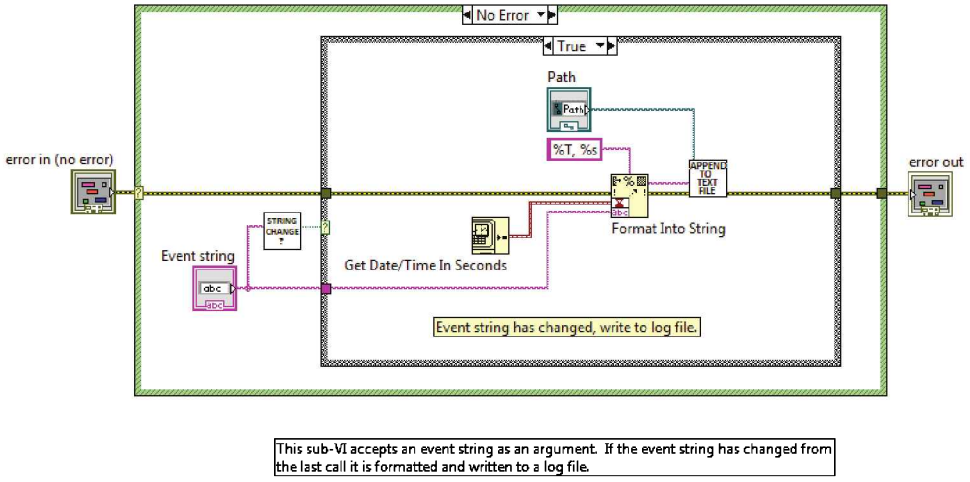


Figure H.4: The block diagram of the `Write To Boiler Log File` VI with the input string changed

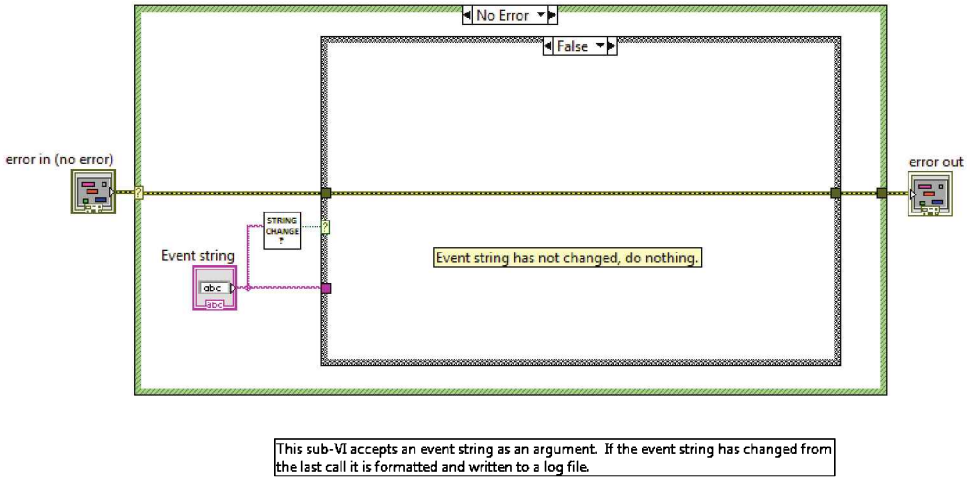


Figure H.5: The block diagram of the `Write To Boiler Log File` VI with the input string unchanged

Exercise 2.5

The goal of this exercise is to construct a `Stopwatch Timer Kernel` sub-VI for a timer action engine. The timer calculates the elapsed time since it was last reset. This means that two state variables are needed: the total elapsed time and the time that it was last called. This also means that the timer needs two commands: *Elapsed Time* and *Reset Timer*. The commands are input to timer kernel using an enumerated control. The enumerated control is wired to the selector of a case structure, which contains the code for the two functions. The elapsed time is output using a double-precision indicator.

An implementation of the *Elapsed Time* function is shown in Figure H.6 using the standard sub-VI with error handling template introduced in Chapter 1. An implementation of the *Reset Timer* function is shown in Figure H.7.

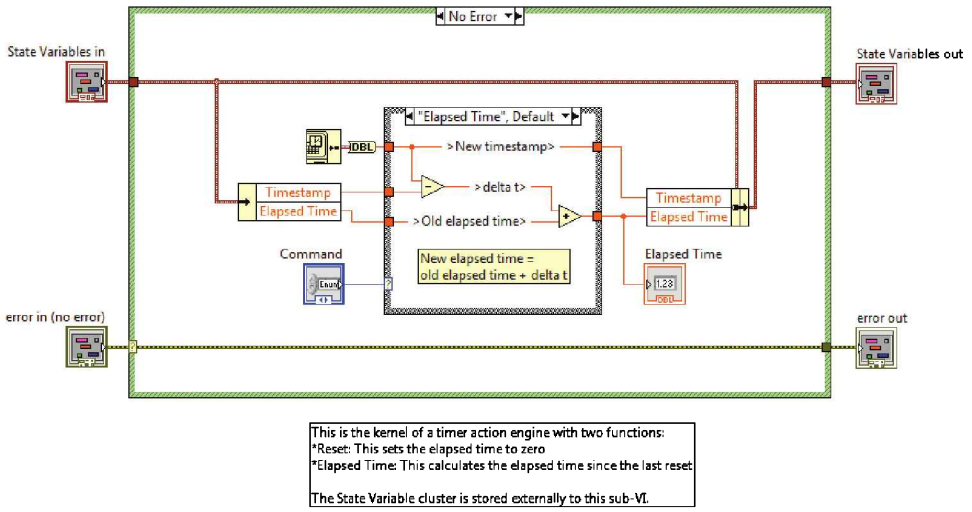


Figure H.6: The *Elapsed Time* case of the `Stopwatch Timer Kernel` sub-VI

Exercise 2.6

The block diagram of the feedback node implementation of the `Stopwatch Timer` action engine is shown in Figure H.8. A feedback node is used to provide persistent storage for the `Stopwatch Timer Kernel` sub-VI developed in Exercise 2.5.

Exercise 2.7

The block diagram for one possible implementation of the `Auto Reset Time Check Method` sub-VI is shown in Figure H.9. The current timestamp is acquired and converted to a double-precision scalar. The old timestamp is subtracted from it, and the result is added to the current elapsed time. The new elapsed time is then compared to the time target. If the time target is met or exceeded, the DONE? Boolean is set to True. The value of the DONE? Boolean is then sent to a **`Select`** function. If DONE?

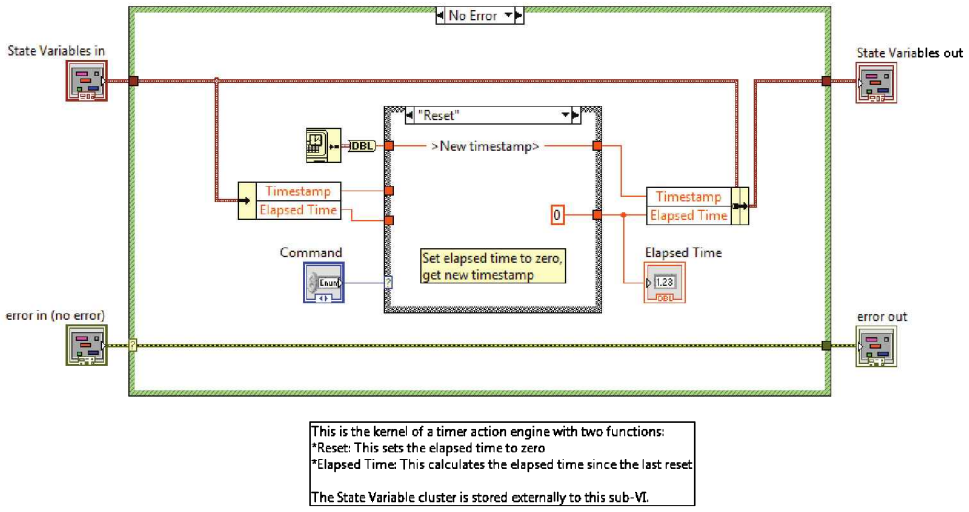


Figure H.7: The *Reset Timer* case of the `Stopwatch Timer Kernel` sub-VI

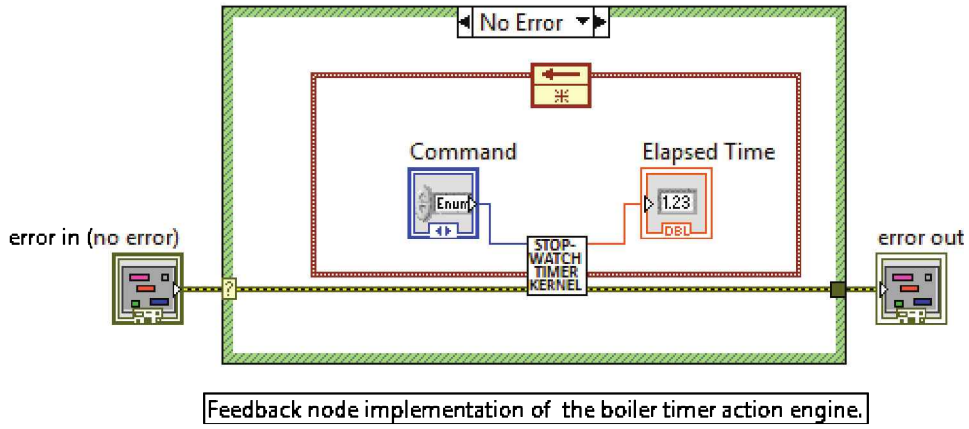


Figure H.8: Block diagram of the `Stopwatch Timer` action engine

is False, the new elapsed time is output to overwrite the old elapsed time. If DONE? is True, the current elapsed time is set to zero, and is output to overwrite the old elapsed time, automatically resetting the timer.

Exercise 2.8

The *Elapsed Time* action of the `Auto Reset Timer Kernel` sub-VI is shown in Figure H.10, and the *Reset* action is shown in Figure H.11. This sub-VI accepts the command input and uses a case structure to select the proper method sub-VI. The kernel sub-VI unbundles the state variables from the input state variable cluster and

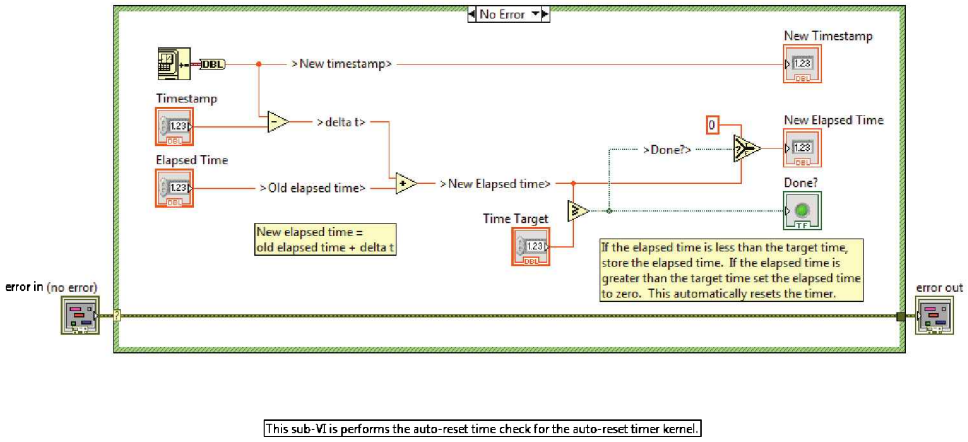


Figure H.9: The block diagram for the `Auto Reset Timer Method` VI

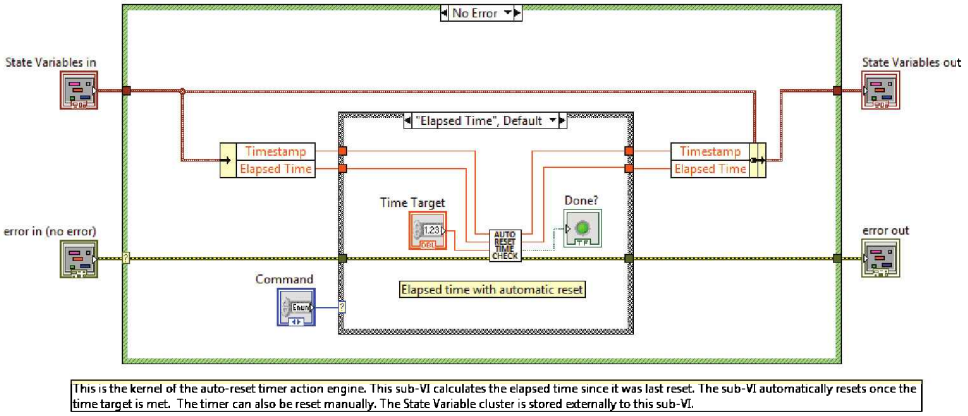


Figure H.10: The *Elapsed Time* case of the `Auto Reset Timer Kernel` sub-VI

passes them as inputs to the case structure. The method sub-VIs operate on the state variables and pass their new values as outputs to the case structure. The kernel sub-VI then bundles the new values of the state variables back into the state variable cluster and outputs the cluster for persistent storage elsewhere.

Exercise 2.9

The `Auto Reset Timer Kernel` state variables are stored externally by the `Auto Reset Timer` action engine as shown in Figure H.12. In this implementation of the action engine, a feedback node is used to provide the storage, but other storage options would also work.

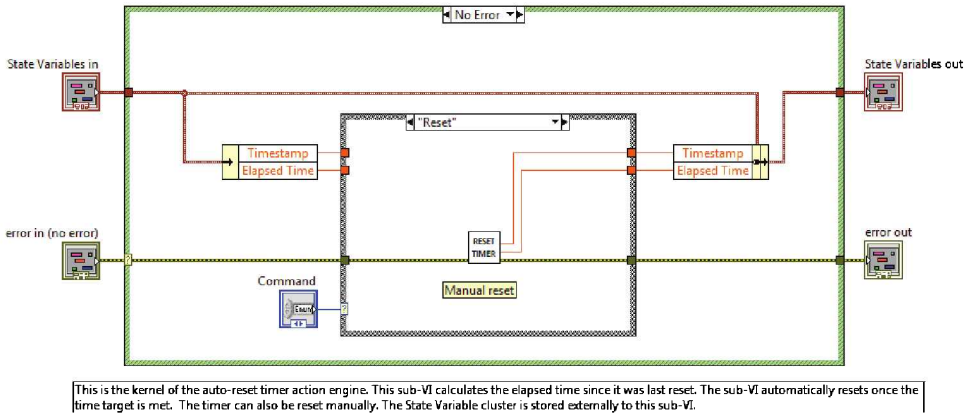


Figure H.11: The *Reset* case of the `Auto Reset Timer Kernel` sub-VI

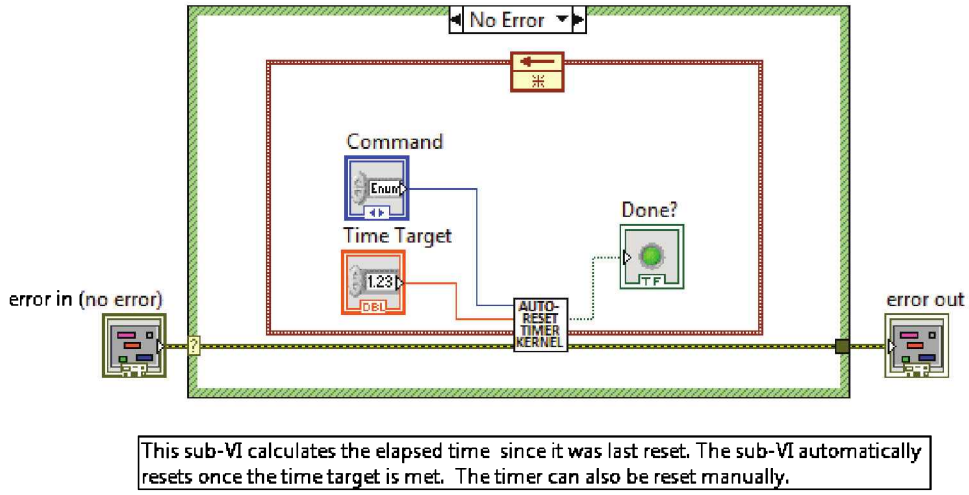


Figure H.12: Feedback node implementation of the auto reset timer

Exercise 2.10

The auto-reset feature would not be useful in the car wash timer because the timing between cycles is not continuous. When one cycle ends, the next cycle doesn't begin until vehicle is in position at the next station. Consider the Deluxe wash. The Deluxe wash starts with the *Underbody Wash* for ten seconds. After the ten seconds expires, the new timer does not begin automatically. The new timer begins only when the car is in position for the *Soap Application* cycle. Thus, an automatically resetting timer would not be useful in the car wash controller.

The same is true for the boiler controller. Only two steps of the boiler controller are timed, and there are a number of other states between them. Resetting the timer

automatically after the first timed state does no good; it would have to be manually reset just before executing the second timed state.

Exercise 2.11

The block diagram for one possible implementation of the `Diff Status Arrays` sub-VI is shown in Figure H.13. The **`Not Equal?`** function produces an element-by-element comparison of the two input arrays. The element-by-element comparison array and the NEW STATUS ARRAY are passed as arguments to auto-indexed input tunnels of a **`For`** loop. The loop contains two conditional output tunnels. The top output tunnel collects the elements of the NEW STATUS ARRAY that are different from the corresponding elements of the OLD STATUS ARRAY. The bottom output tunnel collects the indices of these elements.

After the **`For`** loop has completed, the array of indices is incremented to produces the array of zone numbers that have changed. This is needed because LabVIEW arrays are zero-based, so Zone 1 has an index of zero. The size of the array of changed elements is also calculated. If the the array size is greater than zero, the CHANGED? Boolean is set to True; otherwise, it is False.

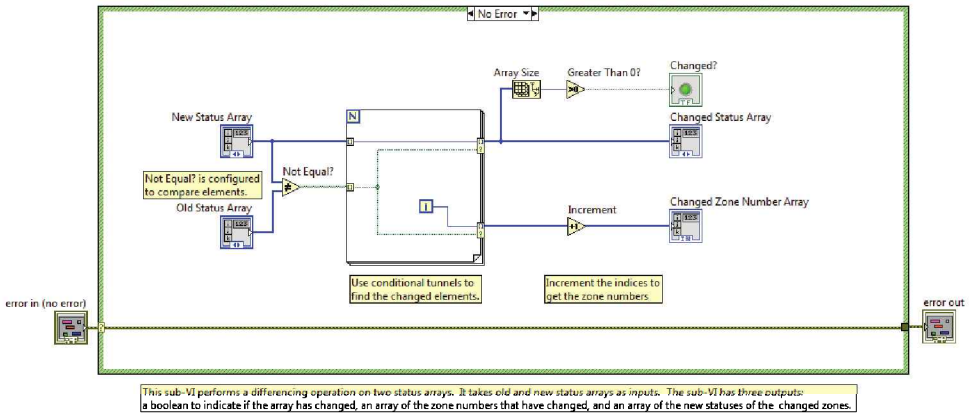


Figure H.13: The block diagram for the `Diff Status Arrays` sub-VI

Exercise 2.12

One possible implementation of the `Update Log` sub-VI is shown in Figure H.14. The new and old status arrays are passed to the `Diff Status Arrays` sub-VI developed in Exercise 2.11. This sub-VI determines if any of the zones have changed status. It also outputs arrays of the zone numbers that have changed and the new status of these zones.

The buttonChanged? Boolean output of the `Diff Status Arrays` is wired to the selector of a case structure. If the status of at least one zone has changed, the zone and status arrays are wired to auto-indexing input tunnels of a **`For`** loop. The timestamp is acquired and passed into the loop as an input to ensure that all log

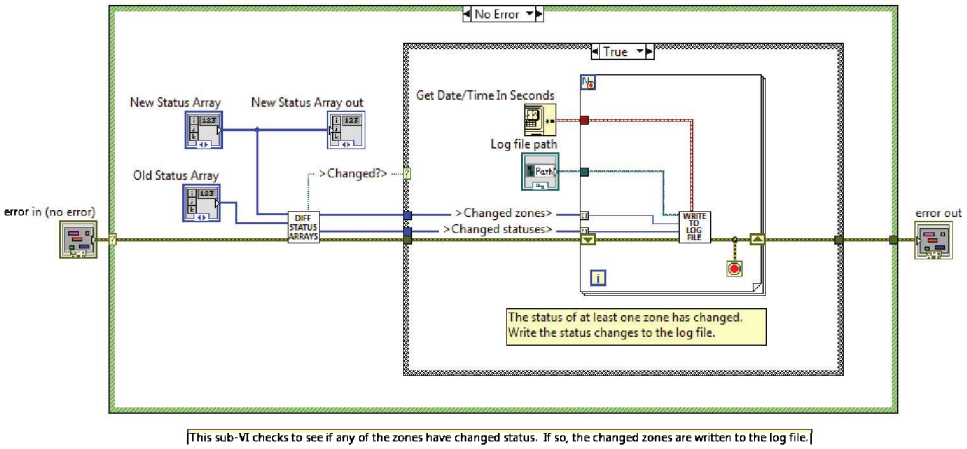


Figure H.14: The block diagram for the `Update Log` sub-VI, status of a zone has changed

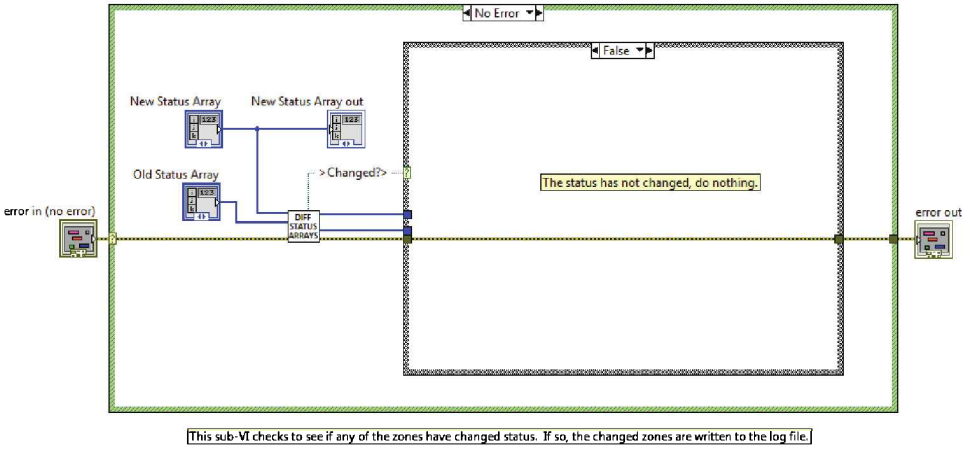


Figure H.15: The block diagram for the `Update Log` sub-VI, no zones changed

strings generated inside the loop will use the same timestamp. The `Write To Log File` sub-VI is placed inside the loop. The timestamp, log file path, and individual zone numbers and zone statuses are passed to this sub-VI. The `Write To Log File` sub-VI then formats this information into log strings and writes them to the log file with one changed zone per line in a comma-separated format. If none of the zones have changed, the log file is not accessed as seen in Figure H.15.

This sub-VI is scalable and will work for zone status arrays of arbitrary length. This is due to two factors: the sub-VI utilizes auto-indexing and the two sub-VIs it calls were also written to work for arrays of arbitrary length.

Problem Set 3 Solutions

Exercise 3.1

The car wash controller has no need for a data highway since there is no need to persistently store data from one loop iteration to the next. There does need to be a shift register for the state enumerated constant. The controller needs to check the STOP button every iteration, and it also needs to check to see if an error has occurred every iteration. This means that the STOP button should be placed in the lobby of the state machine with a guard clause case structure—just as was done in the car wash controller. The vehicle sensors also need to be checked in each loop iteration. Since the sensors need to be available to each of the eight traffic light states, the best place to put them is also in the lobby of the state machine along with the `Bundle Sensors` sub-VI. The indicators also need to be updated every iteration and need to be accessible to all states. Since they are outputs, they should be placed in the alley of the state machine.

One possible implementation of the classic state machine traffic light controller is shown in Figure I.1. Notice the extensive use of clusters to make the program more readable. The cluster of traffic light clusters developed in Exercise 1.5 allows each individual lamp of each traffic light to be accessed by name. For example, in the *N-S Green* state the NORTH-SOUTH GREEN lamp is lit, the NORTH-SOUTH LEFT RED lamp is lit, the EAST-WEST RED lamp is lit, and the EAST-WEST LEFT RED lamp is lit with the rest of the lamps being dark. Examine Figure I.1 to see how this was done. A cluster constant of the all-included traffic light cluster is used as the center input for an **`Unbundle by Name`** function. Since the default of the cluster is all traffic lights dark, the only individual lamps that need to be lit are four lamps that are red or green for that state. These four lamps are lit by wiring True values to them and then bundling them into the all-encompassing traffic light cluster.

Notice the use of the dot notion to signify different levels of the cluster. When *N-S.Green* is set to True, it means that the *Green* Boolean indicator *North-South* cluster in the all-included light cluster is set to True. After the individual lamps are lit, the individual traffic light clusters are unbundled from the all-included light cluster. Each individual traffic light cluster is then wired to the two appropriate front panel traffic light indicators.

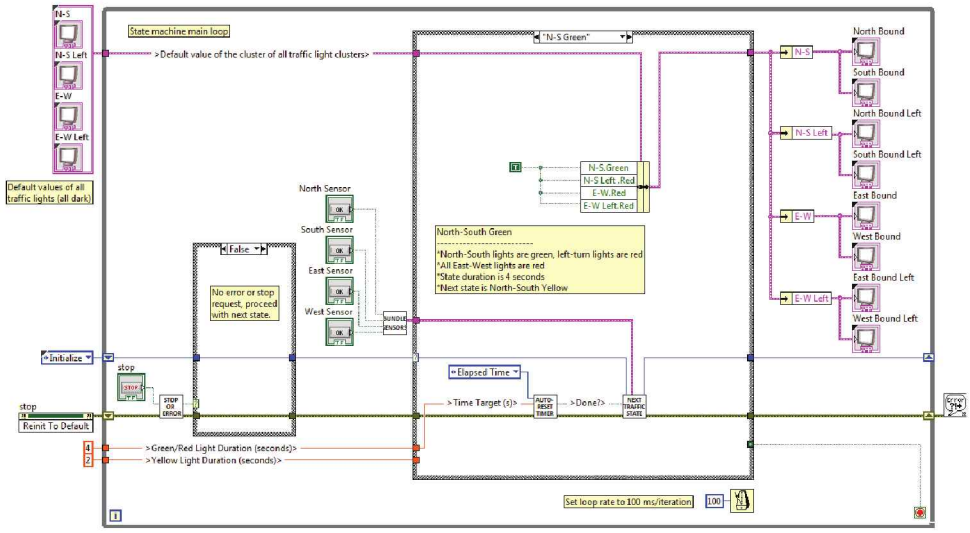


Figure I.1: The *N-S Green* state of the traffic timer

Notice that the all-included traffic light cluster constant has been placed in the input area of the state machine. This cluster could have been placed in the lobby instead, producing a shorter wire. But here we are following the general principal proposed in Chapter 4 that items that vary and are needed by multiple states should be in the lobby, while items that are constant and needed by multiple states should be placed in the input area of the state machine. This also explains why the duration of the green/red lights and the duration of the yellow lights were placed in the input area. These constants are shared by all the traffic light states. Placing them in the input area allows two constants to be used by all the states, making the code more maintainable if these constants need to be changed in the future.

The vehicle sensors are placed in the lobby of the state machine since they are variable and are needed by multiple states. They are then bundled into a cluster by the `Bundle Sensors` sub-VI before they are passed into the case structure. This is done to reduce the number of wires entering the case structure, improving the readability of the VI.

The basic flow of the program is as follows. When a new loop iteration begins, the same `Stop or Error` sub-VI used in the car wash controller checks to see if the STOP button was pressed or if an error has occurred. The output of this sub-VI is then used to trigger the guard clause. If no error is present and no stop request has been made, the next state is passed through to the case structure. The first state to execute is the *Initialize* state shown in Figure I.2. The traffic lights are all set to their default values using implicitly linked property nodes, and the timer is reset manually to ensure that the first traffic light cycle is run for its full duration.

If an error is present or if a stop request has been made, the guard clause bypasses the state shift register output and transitions the state machine to the *Shutdown* state shown in Figure I.3. The vehicle sensors are reset, the traffic light indicators are all

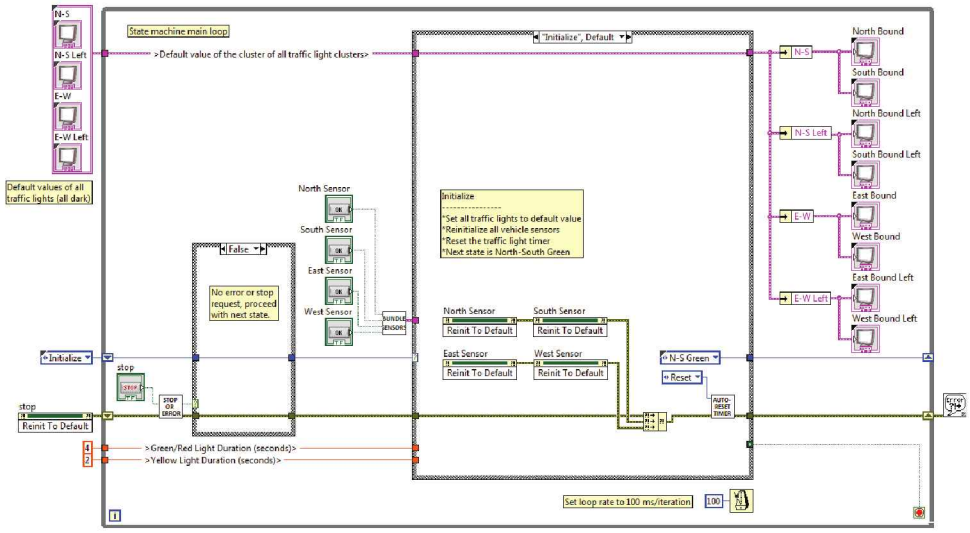


Figure I.2: The *Initialize* state of the traffic timer

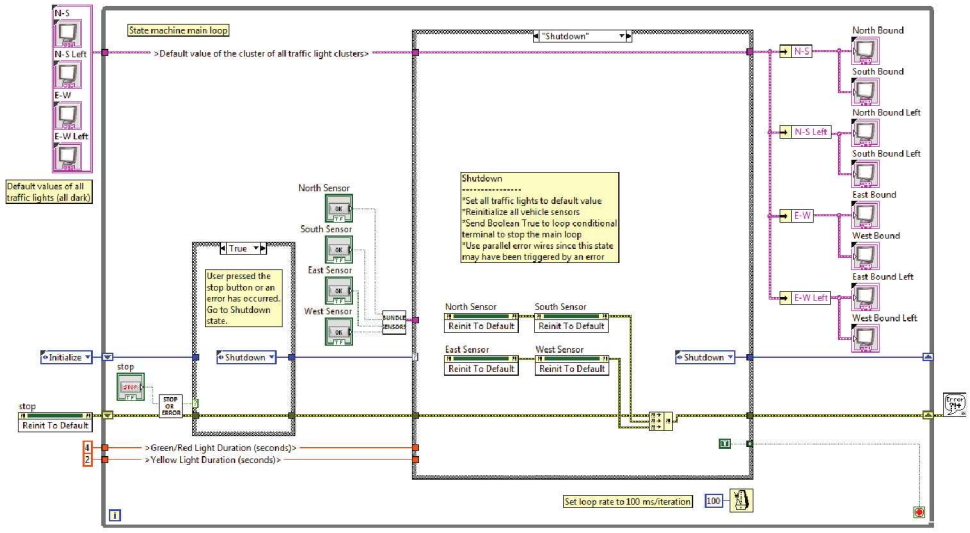


Figure I.3: The *Shutdown* state of the traffic timer

reset to their default value, and loop is terminated by sending a True to the stop terminal.

After the *Initialize* state executes, the state machine transitions to the *N-S Green* state shown in Figure I.1. The proper lamps are lit, and a time check is performed. The DONE? output of the `Auto Reset Timer`, the vehicle sensor cluster, and the current state are then passed to the `Next Traffic State` sub-VI to determine the next state transition.

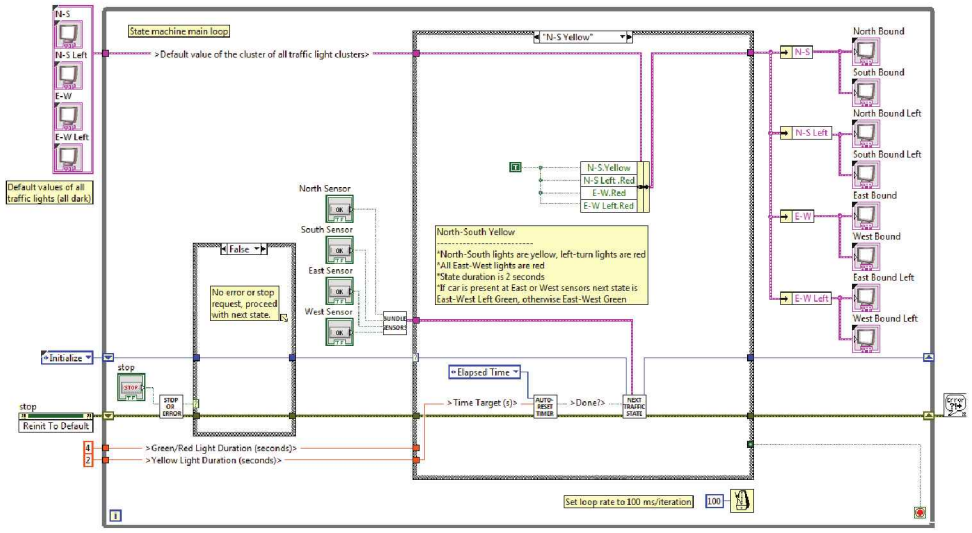


Figure I.4: The *N-S Yellow* state of the traffic timer

All of the traffic light states are nearly identical to the one shown in Figure I.1 with the only difference being which lamps are lit. The *N-S Yellow* state is shown in Figure I.4 as an example. The only difference between this state and the *N-S Green* state is in which lights are lit and which time target is chosen. This demonstrates one of the benefits of creating a `Next Traffic State` sub-VI. By hiding the details of how the next state is chosen, the appearance of each state is nearly identical. This improves the readability of the main VI and highlights the top-level logic of the code. For each light state, particular lamps are lit, a time check is made, and the vehicle sensors are checked to determine the next state. This concludes the development of the classic state machine implementation of the traffic controller.

Exercise 3.2

The classic state machine implementation of the security system is straightforward since there are only three states to implement: *Initialize*, *Monitor*, and *Shutdown*. The overall architecture and the *Initialize* state can be seen in Figure I.5. The STOP button is initialized in the input area of the state machine. The *Initialize* state is placed on the shift register to dictate the first state transition. The log file path is constructed using the **`Build Path`** function and the **`Application Directory`** constant. This builds a path to the logfile assuming that it is found in the application directory. The zone status indicator array is preallocated in the input area using the `Initialize Zone Status Array From Cluster` sub-VI developed in Exercise 1.10. The initialized and preallocated array is then placed on the data highway. As the code runs, the zone status array from the previous iteration will be stored on the data highway. This will allow the code to compare the new value of the zone indicator array with the old

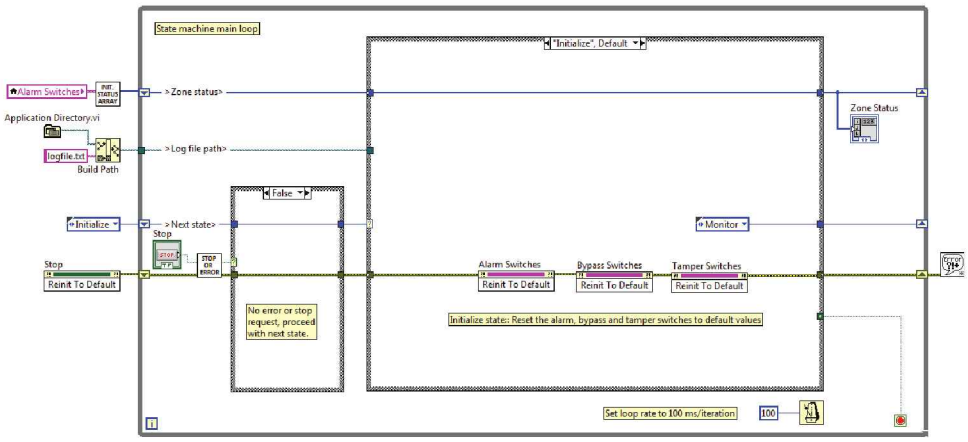


Figure I.5: The *Initialize* state of the simplified security system controller

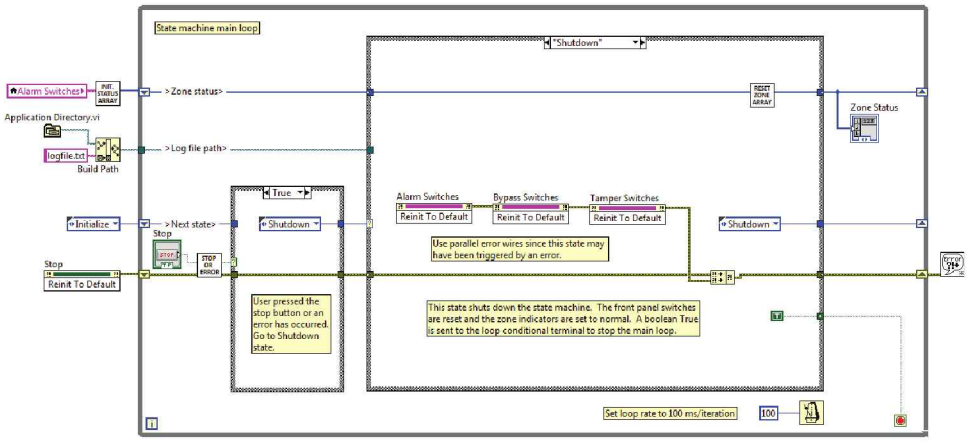


Figure I.6: The *Shutdown* state of the simplified security system controller

value to see if any of the zone indicators have changed. If they have changed, the log file will be updated.

The STOP button is placed in the lobby of the state machine so that it will be read every loop iteration. The lobby has the standard guard clause that checks to see if an error is present or if the STOP button has been pressed. This check is done by the `Stop or Error` sub-VI shown in Figure 8.7. If the STOP button has not been pressed and no error is present, the current state is passed to the main case structure. If an error is present or if the STOP button has been pressed, the current state is ignored and the state machine transitions directly to the *Shutdown* state seen in Figure I.6. This state resets the front panel switches using *Reinitialize To Default* invoke nodes and resets the zone indicator array using the `Reset Zone Array` sub-VI developed

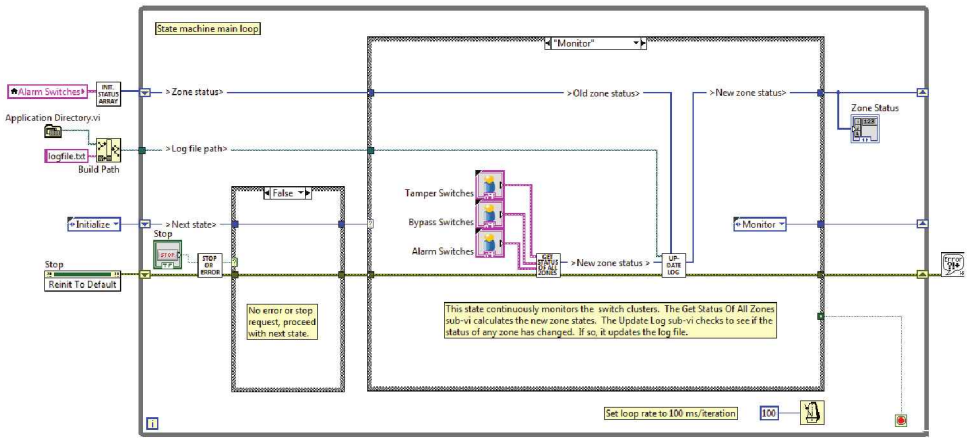


Figure I.7: The *Monitor* state of the simplified security system controller

in Exercise 1.11. The invoke nodes are placed on a parallel error wire, since the *Shutdown* state can be triggered by an error on the main error wire and an error would cause the invoke nodes to forego execution. The parallel error wire is merged back into the main error wire using the **`Merge Errors`** function. The error on the main error wire is given priority. Of course, the *Shutdown* state also shuts down the state machine by wiring a Boolean True to the stop terminal of the loop. Notice that the *Shutdown* state is wired to the state shift register. This state will not execute, since the loop will have been stopped. But output tunnels to case structures need to be wired in every case, so the *Shutdown* case was wired here.

Returning to Figure I.5, the *Initialize* state reinitializes the front panel switches using *Reinitialize To Default* invoke nodes. The default value of the zone status array generated by the `Initialize Zone Status Array From Cluster` sub-VI is passed through this state to the ZONE STATUS indicator in the alley of the state machine. The ZONE STATUS indicator is placed in the alley to ensure that it can be modified by every state in the state machine. The *Monitor* state is then placed on the state shift register.

The only other state is the *Monitor* state shown in Figure I.7. The values of the the TAMPER, BYPASS, and ALARM switches are monitored each loop iteration by the `Get Status Of All Zones` sub-VI developed in Exercise 1.13. This sub-VI calculates the current zone states and passes them to the `Update Log` sub-VI developed in Exercise 2.12. This sub-VI compares the current zone status array to the old zone status array stored on the data highway. The log file is updated if any of the states have changed. The new values of the zone indicator states are then placed on the data highway and the state machine returns to the *Monitor* state. This process repeats until the STOP button is pressed or an error is detected. This concludes the classic state machine implementation of the security system. Notice that the state machine itself is quite simple—all of the detailed work in monitoring the security system is performed by the sub-VIs.

Exercise 3.3

The simplified security system developed in Exercise 3.2 is highly scalable. The scalability can be tested by opening the type-definition file of the switches cluster and adding a seventh switch and by expanding the front panel ZONE STATUS ARRAY indicator to show a seventh zone. If this is done, the run arrow will not break and the code will operate as expected. The seventh zone will be just as functional as the first six zones. The log file will update as expected when the status of the seventh zone changes.

What features ensured the scalability of the simplified security system controller? The **Cluster To Array** function plays a large role. The determining factor for the number of zones is the number of switches in the switches cluster. Unlike arrays, cluster sizes are fixed at run time so the cluster serves as the anchor for the security system controller. The number of zones can be programatically determined by converting the switches cluster to an array and taking the size of the array using the **Array Size**. This is the approach taken by the `Initialize Zone Status Array From Cluster` sub-VI. Another way the security system controller promotes scalability is through the use of **For** loops and auto-indexing. Auto-indexing allows the loop to execute a fixed number times equal to the number of zones without knowing ahead of time how many zones are in the system. Again, this technique is anchored in the switches cluster size. The `Get Status Of All Zones` sub-VI also uses the **Cluster To Array** function to convert the switches clusters to arrays and then feeds these arrays to a **For** loop through auto-indexed input tunnels (see Figure G.11). The loop can then examine each zone in turn without knowing in advance how many zones are in the system.

The `Reset Zone Array` sub-VI takes the same approach. This sub-VI wires the existing status array to an auto-index input tunnel of a **For** loop and the output array of enumerated constants is generated by an auto-indexed output loop. This ensures that the output array is the same size as the input array. Since the size of the input array was determined by the size of the switches cluster, this scalability of this sub-VI is also traced back to the fixed size of the switches cluster.

The `Diff Status Array` function shown in Figure H.13 also relies ultimately on the fixed size of the switches cluster. The size of the status array is fixed by the size of the switches cluster. The `Diff Status Array` compares the old and new status arrays using the **Not Equal?** function. This function is configured to compare elements, so it produces an array of Booleans the same size of the status array which is the same size as the fixed switches cluster. The scalability of this sub-VI relies on the fact that the **Not Equal?** function is itself scalable; it will produce an output array that matches the size of whatever input arrays are given to it. The `Diff Status Array` also uses an auto-indexed **For** loop, which also adapts to the size of the input arrays. The output of the `Diff Status Array` is used by the `Write To Log File` sub-VI, which also uses a **For** loop with auto-indexing to ensure scalability.

To summarize, the scalability of the simplified security system controller is based on a simple strategy: Determine the number of zones from the fixed length switches clusters, convert the switches clusters to arrays, and then use auto-indexed **For** loops and native LabVIEW array-handling functions to process the arbitrarily long zone arrays.

Exercise 3.4

The use of local variables is acceptable here. The general prohibition against the use of local variables is intended to make you think carefully before using them. Local variables should be used carefully because they violate dataflow. The local variables allow the contents of front panel controls and indicators to be accessed at any point in the program without extending a wire to or from the terminal of the control. They also allow the normal flow of data to be reversed, reading data from an indicator or writing data to a control. The danger here is that local variables can be overused or misused by a developer to serve as a named variable similar to variables in a standard text-based language. The local variable becomes "x", which is a variable that stores information and can be called from anywhere in the program simply by dropping a local variable on the block diagram. The developer is then violating the spirit of dataflow.

The general prohibition against the use of local and global variables is intended to prevent the developer from being tempted to violate the spirit of dataflow. Looked at in this light, the use of a local variable to preallocate the zone status array is an acceptable exception to the general prohibition of local variables. In Exercise 3.3, we saw how the scalability of the entire program was based on determining the fixed length of the switches cluster and how the dataflow of the program used that information to handle an arbitrary number of zones. Thus, the decision to use a local variable to determine the number swtiches in the cluster serves the dataflow and scalability of the program and does not violate the spirit of dataflow. Another mitigating factor is that the local variable is only used once in the input area to the loop. It plays a key role at the very beginning of the program but then exits the stage and is not used again for the duration of the program. This indicates that the local variable is not being overused in this program. Local variables inside the main body of a program that get executed multiple times should receive careful scrutiny, but this is not the case here.

Exercise 3.5

The block diagram of one possible implementation of the `Initialize Status Array From Cluster Reference` is shown in Figure I.8. A control refnum cast as a cluster refnum accepts a generic cluster reference. An array of references to the controls inside the cluster is generated by the *Controls[]* property node. The size of this array is obtained using the **`Array Size`** function, and this size is used to generate the default zone status array using the **`Initialize Array`** function.

One interesting feature of this block diagram is the method in which it closes the references. The error wires are used to enforce the correct order of operations with dataflow. The cluster reference is closed immediately after the array of control references is generated. The error wire ensures that the *Controls[]* property node executes before the **`Close Reference`** function. This is easy to do, since both the property node and the **`Close Reference`** function have error input and output terminals. We also need to ensure that the **`Array Size`** function reads the size of the control reference array before the references are closed by the second **`Close Reference`** function. But this time, the situation is different because the **`Array Size`** function does not have error terminals. The solution to this problem is to wrap the **`Array Size`**

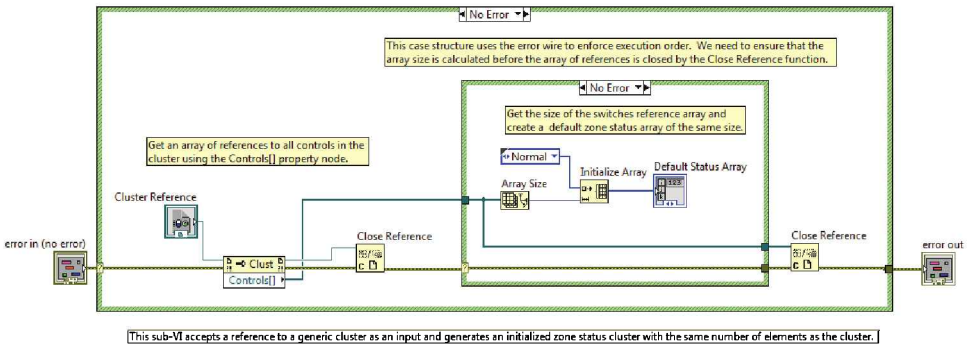


Figure I.8: The block diagram for the `Initialize Status Array From Cluster Reference` sub-VI

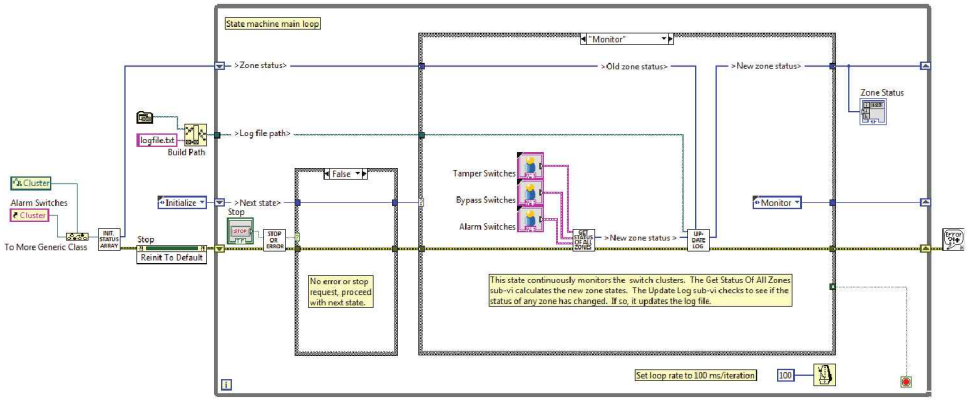


Figure I.9: The *Monitor* state of the revised simplified security system controller

function in a case structure whose selector is wired to the error wire. The case structure will not execute until the error information arrives at the selector terminal. The contents of the case structure will then execute and error information will be passed to the output tunnel. The error information is then passed to the second **`Close Reference`** function. This technique uses dataflow to ensure that the array size is obtained before the array of references is closed.

The `Initialize Status Array From Cluster Reference` can now be used in the revised simplified security system controller to remove the local variable of the switches cluster. The *Monitor* state of the revised controller is shown in Figure I.9. It is identical to Figure I.7 with the exception of how the zone status array is preallocated. In Figure I.9, a reference to the ALARM SWITCHES cluster is used to determine the number of zones and preallocate the zone status array. This reference is cast to a generic cluster reference using the **`To More Generic Class`** function and is passed to the `Initialize Status Array From Cluster Reference` sub-VI to preallocate the zone status array.

If you test the scalability of the revised security system controller, you will find that it also works as expected when extra zones are added. The `Initialize Status Array From Cluster Reference` sub-VI is constructed using the same procedure as the rest of the controller. It is anchored in the fixed size of the referenced switches cluster. It then creates an array of the same size as the switches cluster using the *Controls[]* property node and proceeds to get the size of the array and create the zone array using native LabVIEW array functions that work on arrays of arbitrary length.

Another more subtle feature that increases scalability is the fact that the switches cluster reference is cast to a generic cluster reference before it is used to preallocate the array. The `Initialize Status Array From Cluster Reference` could have been written to use a strictly typed switches cluster refnum instead of a generic cluster refnum. Strictly typed refnums only work with references to the exact type of control that were created for. In this case, a strictly typed refnum would expect a reference to a cluster of six Boolean controls. This would allow the developer to wire a switches cluster directly to the preallocation sub-VI without coercion dots and without having to cast it as a generic cluster. However, this would restrict the sub-VI to working with clusters with only six Boolean controls and would thus sacrifice the scalability of the main application. Working with generic cluster references allows the preallocation sub-VI to work with clusters of arbitrary numbers of zones, preserving the scalability of the main application.

Exercise 3.6

The block diagram of one possible implementation of the `Get Lamp Refnums` sub-VI is shown in Figure I.10. The ZONE STATUS REFNUM is a generic control refnum from the *Refnum* sub-palette that has been cast to the cluster class by right-clicking the control, selecting *Select VI Server Class*, and navigating through the class hierarchy as follows: *Generic...GObject...Control...Cluster...Cluster*. This reference is passed to an explicitly linked *Controls[]* property node to generate an array of generic control

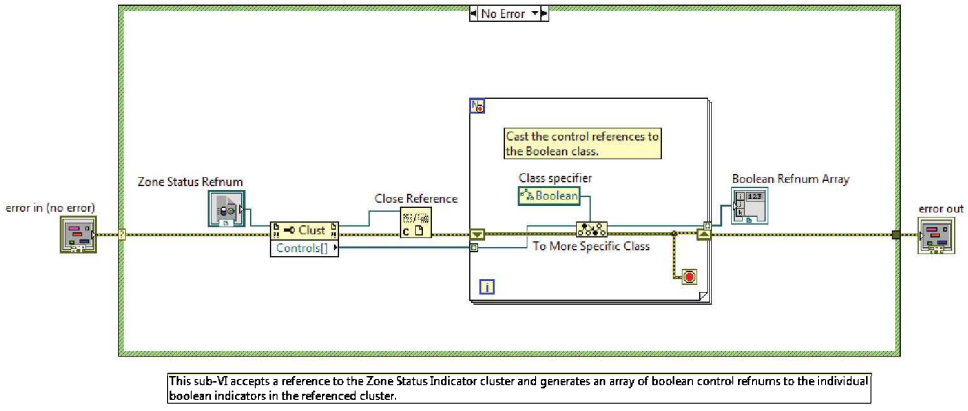


Figure I.10: The block diagram of the `Get Lamp Refnums` sub-VI

references to the zone indicator lamps inside the cluster. The cluster reference is then closed with the **`Close Reference`** function. This function can remain on the main error wire, since it will execute even if an error is present at its input terminals.

The array of control references is then passed to an auto-indexed input tunnel of a **`For`** loop. The individual control references are cast to the Boolean class using the **`To More Specific Class`** function. The Boolean class specifier constant has the following path in the class hierarchy: *Generic...GObject...Control...Boolean*. The individual Boolean control references are collected into the desired output array by an auto-indexed **`For`** loop output tunnel.

Exercise 3.7

The block diagram of one possible implementation of the `Initialize Status Array From Boolean Reference Array` sub-VI is shown in Figure I.11. This is a very simple sub-VI, since the size of the output array can be read directly from the size of the input array using the **`Array Size`** function. The array size is then used to preallocate the desired array using the **`Initialize Array`** function.

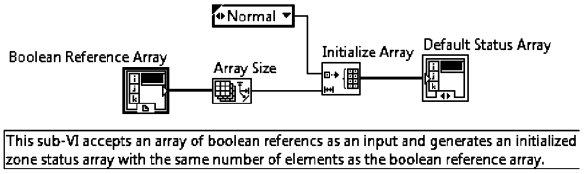


Figure I.11: The block diagram for the `Initialize Status Array From Boolean Reference Array` sub-VI

Exercise 3.8

The *Monitor* state of the full version of the security system controller is shown in Figure I.12. It is very nearly identical to the simplified version shown in Figure I.7.

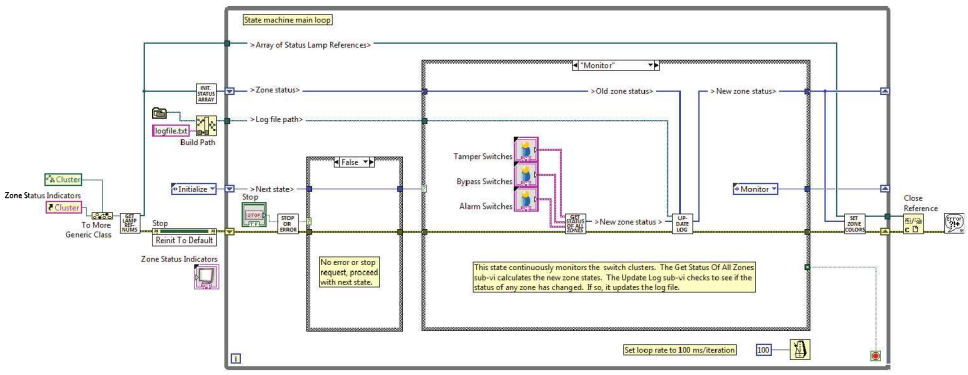


Figure I.12: The *Monitor* state of the security system controller

A reference to one of the switches clusters is cast to a generic cluster reference using the **To More Generic Class** function in the input area of the main loop. This reference is then passed to the `Get Lamp Refnums` sub-VI to generate an array of Boolean control references for the indicator lamps in the ZONE STATUS INDICATORS cluster on the front panel. This array is used to preallocate the zone status array using the `Initialize Status Array From Boolean Reference Array` sub-VI. The array of Boolean references is passed as a constant input to the main loop. The references to the indicators do not change as the program runs and do not need to be placed on the data highway.

The guard clause and the main case structure of the state machine are unaffected by these changes. The last change is found in the alley of the state machine. The `Set Zone Colors` sub-VI resides in the alley, taking the place of the zone indicator array in the simplified form of the security system. This sub-VI accepts the array of Boolean control references and the new value of the zone status array as inputs and changes the color of the front panel indicators to match the statuses designated by the zone status array. The array of Boolean control references are then passed to an output tunnel of the main loop. When the main loop terminates, this array of references is passed to the **Close Reference** function, which closes all of the Boolean control references in the array before the program terminates. This concludes the development of the classic state machine implementation of the security system controller.

Notice the similarity between the simplified and full versions of the security system controllers. The state machine architecture is quite modular. The major difference between the two programs concerns details of the user interface and not differences in program logic, so the main differences in the block diagram were localized to the input area, alley, and output area of the state machine. The lobby, data highway, guard clause, and main case structure were unaffected by the changes.

Also notice that the ZONE STATUS INDICATORS cluster is not wired in the main program. The color changes of the indicators in this cluster are carried out indirectly through the use of references and explicitly linked property nodes, so this terminal never needs to be read by the main program. It is placed in the input area of the main loop for convenience.

Exercise 3.9

The full version of the security system behaves as expected when the number of zones is increased. No changes need to be made to the block diagram. All three of the added sub-VIs were written in a scalable fashion. The `Get Lamp Refnums` sub-VI uses the *Controls[]* property node to get an array of references to the indicator lamps. This property node will work for an arbitrary number of lamps. A **For** loop with auto-indexing is used to cast the control references to the Boolean class, so this too will work for an arbitrary number of lamps. The `Initialize Status Array From Boolean Reference Array` is written using the array-handling native LabVIEW functions, so these too are inherently scalable. The `Set Zone Colors` also uses an auto-indexed **For** loop and native array-handling functions to set the zone colors in a scalable fashion. The **Close Reference** function is also scalable and is capable of closing all of the references in a one-dimensional array of unspecified length.

As mentioned in the solution to Exercise 3.5, another more subtle factor contributing to the scalability of the security system controller is the fact that the reference to the switches cluster is not strictly typed. Casting the reference to a generic cluster reference avoids coercion dots and also helps maintain the scalability of the main application.

Exercise 3.10

The completed configuration window for the `Initialize Status Array` polymorphic VI is shown in Figure I.13. The three instance VIs are the `Initialize Status Array From Cluster` sub-VI, the `Initialize Status Array From Cluster Reference` sub-VI, and the `Initialize Status Array From Boolean Reference Array` sub-VI. The connector panes of these three sub-VIs are compatible. These three sub-VIs all share the same connector pane pattern. They all wire the same terminal as an input for the cluster, cluster reference, or array of Boolean control references. They all wire the same terminal for the output preallocated array. The `Initialize Status Array From Cluster Reference` sub-VI also wires error input and output terminals, but since the other instance VIs leave these terminals unwired, the three connector panes remain compatible.

An icon was created for the polymorphic VI, and the radio button was selected to draw the polymorphic VI icon instead of the instance VI icons. The correct checkbox was selected to configure the polymorphic VI to adapt to the input data type. Appropriate menu names and selector names were assigned. To see the menu names, drop the polymorphic VI on a block diagram and right-click it and choose *Select Type*. To see the selector names, open the polymorphic VI and check *Show Selector By Default*. Drop a new instance of the polymorphic VI on a block diagram and you

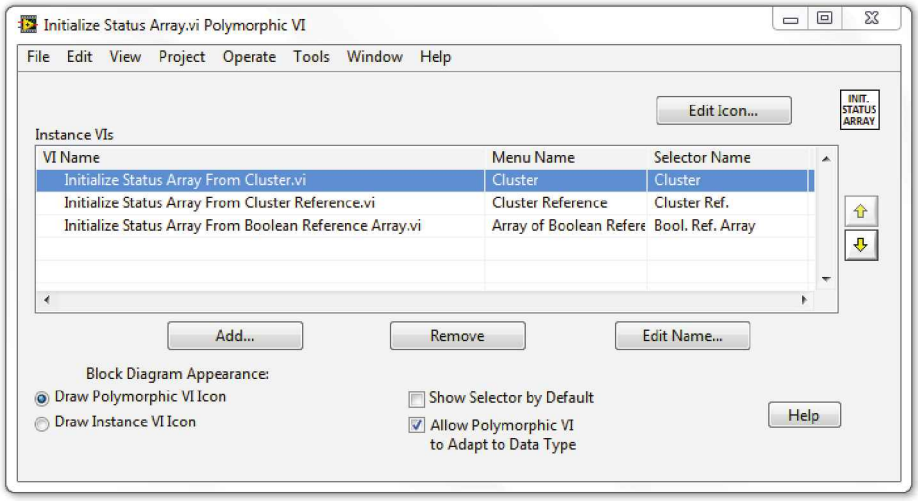


Figure I.13: Configuration window for the `Initialize Status Array` polymorphic VI

see the selector beneath the icon of the VI. Click on the selector to manually select a particular instance VI.

Exercise 3.11

The boiler controller is very different from the other projects in this book. Most of the other projects have state transition diagrams that look like a spider web with a central *Wait* state from which the program departs and returns. The state transition diagram for the boiler controller is more like a ladder. Each step of the ladder waits for a certain condition to be fulfilled before advancing to the next step, and the conditions for advancement are unique for each step. This is a common design pattern for monitoring and controlling multi-step industrial processes.

The overall structure of the boiler controller and the *Initialize* state are shown in Figure I.14. The path to the log file is constructed in the input area of the loop and is passed to the main case structure as a constant input. The default value of the BOILER STATUS cluster is also passed as a constant input. Each stage of the boiler process will use this constant as a baseline, modifying it as necessary. A cluster of numeric constants called the LIMITS cluster is also placed in the input area of the loop. These constants are fixed by the functional specification and never vary, so it makes sense to place them in the input area of the loop. The STOP button is also reinitialized in the input area of the loop, and the state constant shift register is initialized to the *Initialize* state.

The lobby of the state machine contains the guard clause case structure, the BOILER SIMULATION control, the BOILER CONTROLS control, and the STOP button. The BOILER SIMULATION control, the BOILER CONTROLS control, and the STOP button all need to be polled each loop iteration, so the lobby of the state machine is the proper location for these controls. The BOILER STATUS indicator cluster must be updated each loop iteration, so it is placed in the alley of the state machine. Notice that there is no data highway. The boiler controller does not need to store data persistently from one loop iteration to the next.

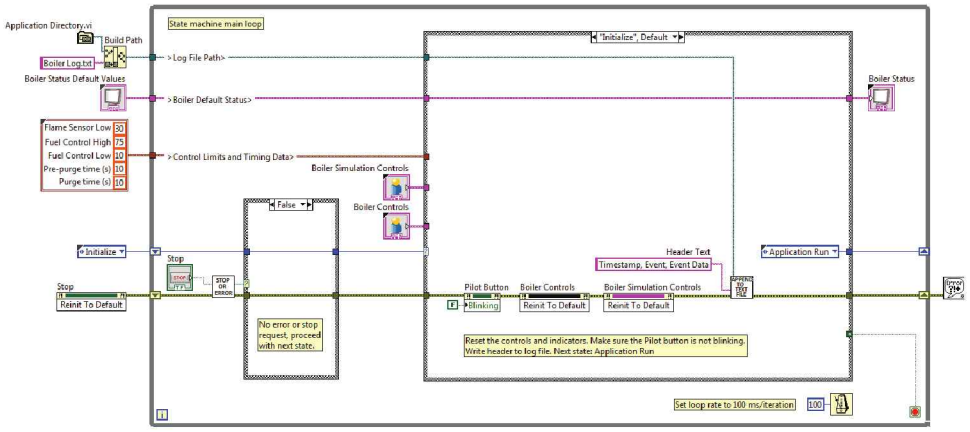


Figure I.14: The *Initialize* state of the boiler controller

The `Stop or Error` sub-VI checks to see if the STOP button has been pressed or if an error has occurred. If neither has occurred, the current state is passed from the state shift register to the main case structure. The *Initialize* state reinitializes the BOILER CONTROLS cluster and the BOILER SIMULATION CONTROLS cluster. This state also ensures that the PILOT BUTTON is not blinking. It does this by passing a Boolean False to the *Blinking* property node of the PILOT BUTTON control. The *Initialize* state also opens or creates the log file and writes the header information to the file using the `Append To Text File` sub-VI shown in Figure 1.45.

The *Initialize* state transitions immediately to the *Application Run* state, shown in Figure I.15. This state first checks to see if it is done using the `Application Run Done` sub-VI developed in Exercise 1.18. The output for this state is generated by the `Application Run Output` sub-VI developed in Exercise 1.19. Notice that this sub-VI writes the output generated by the *Initialize* state. For instance, it writes the string `Lockout` to the STATUS indicator. The output generated in each step signifies the satisfactory conclusion of the completion conditions of the previous step.

The `Application Run Output` sub-VI also generates a string for the log file. This string is written to the file using the `Write To Boiler Log File` sub-VI developed in Exercise 2.4. This sub-VI only writes the string if the string is unchanged from the last time the sub-VI was called. Since the *Application Run* log string never changes, this ensures that it is written to the log file only once.

If the *Application Run* state is not finished, it is placed back on the state shift register to execute again. This continues until the state is finished, as shown in Figure I.16. When the *Application Run* state is complete, the *Reset Lockout* state is placed on the shift register. This will be a common pattern for the states representing stages of the boiler process. If the current state is not finished, it is placed on the shift register again. If the current state is finished, the next state is placed on the shift register. For the rest of the boiler controller states, only the code for the "current state done" case will be shown.

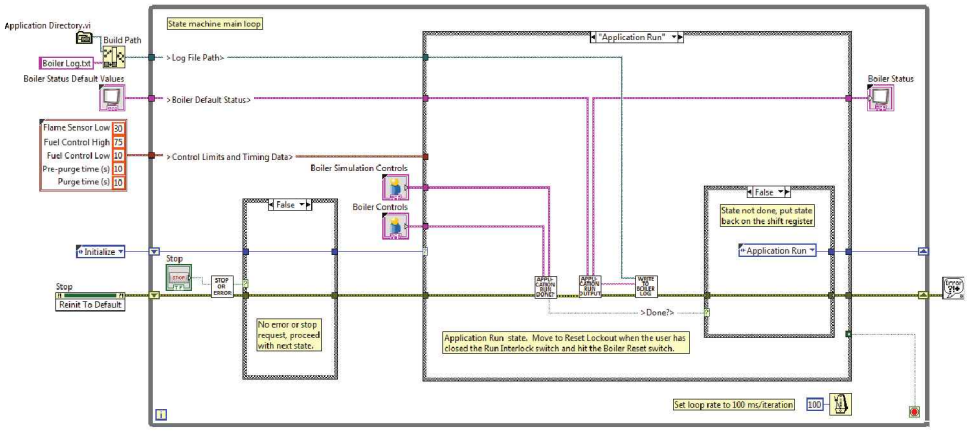


Figure I.15: The *Application Run* state of the boiler controller, state not done

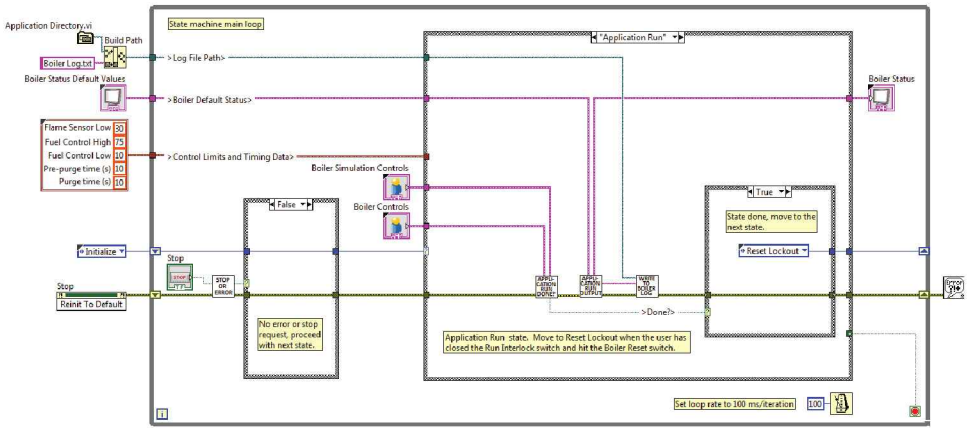


Figure I.16: The *Application Run* state of the boiler controller, state done

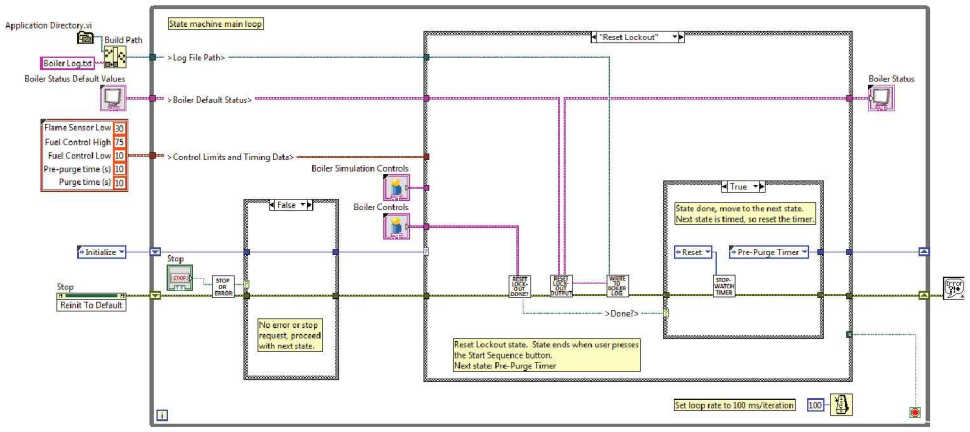


Figure I.17: The *Reset Lockout* state of the boiler controller

The *Reset Lockout* state is shown in Figure I.17. The `Reset Lockout Done` sub-VI developed in Exercise 1.20 checks to see if the state is done. The `Reset Lockout Output` sub-VI developed in Exercise 1.21 generates the BOILER STATUS cluster and the log file entry for the state. When the state is finished, the *Pre-Purge* state is placed on the state shift register. This state is timed, and the `Stopwatch Timer` action engine developed in Exercise 2.6 is used to keep track of the elapsed time. The timer is reset here in preparation for the *Pre-Purge Timer* state.

The *Pre-Purge Timer* state of the boiler controller is shown in Figure I.18. The elapsed time is calculated and passed to the `Pre-Purge Timer Done` sub-VI developed in Exercise 1.23. The `Pre-Purge Timer Output` sub-VI developed in Exercise 1.24 generates the BOILER STATUS cluster and the log file entry for the state. When the state is completed, the *Pre-Purge* state is placed on the state shift register.

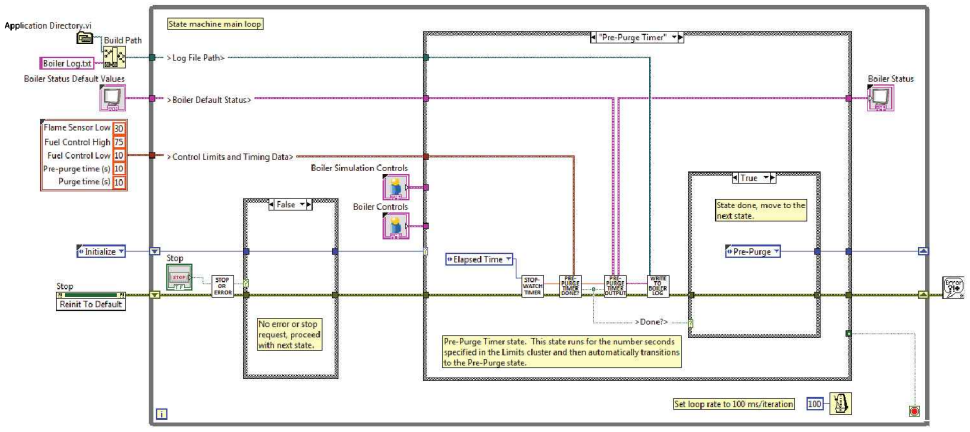


Figure I.18: The *Pre-Purge Timer* state of the boiler controller

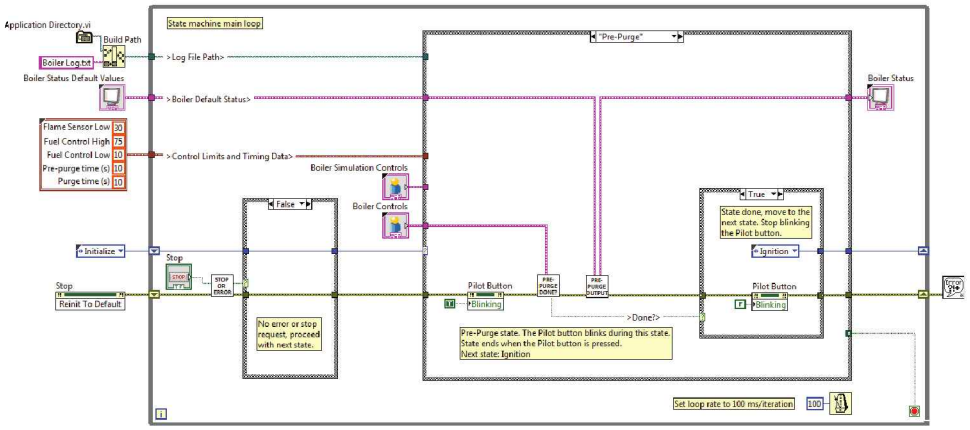


Figure I.19: The *Pre-Purge* state of the boiler controller

When the timed pre-purge is completed, the *Pre-Purge* state executes as shown in Figure I.19. This state causes the PILOT BUTTON control to blink by passing a Boolean True to its *Blinking* property node. The `Pre-Purge Done` sub-VI developed in Exercise 1.25 checks to see if the state is done. The `Pre-Purge Output` sub-VI developed in Exercise 1.26 generates the BOILER STATUS cluster and the log file entry for the state. When the state is complete, the *Ignition* state is placed on the state shift register. The PILOT BUTTON blinking is stopped by wiring a Boolean False to the *Blinking* property node of the control.

The *Ignition* state of the boiler controller is shown in Figure I.20. The `Ignition Done` sub-VI developed in Exercise 1.27 checks to see if the state is finished. The `Ignition Output` sub-VI developed in Exercise 1.28 generates the BOILER STATUS

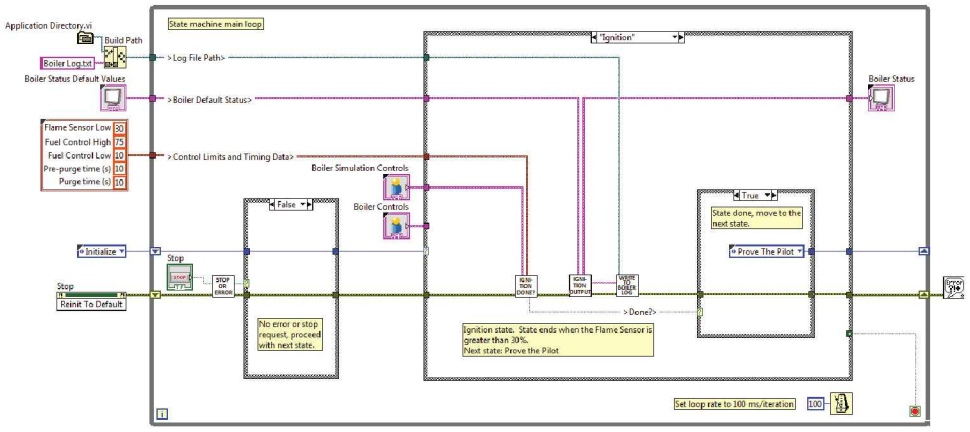


Figure I.20: The *Ignition* state of the boiler controller

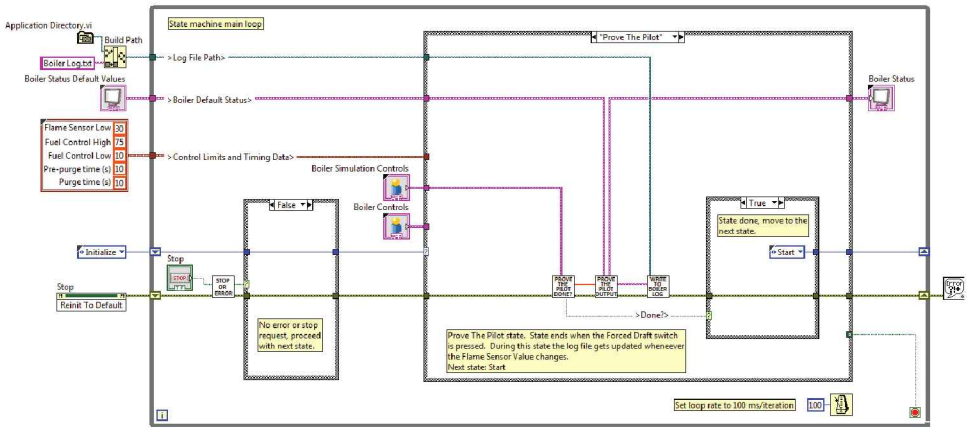


Figure I.21: The *Prove The Pilot* state of the boiler controller

cluster and the log file entry for the state. When the state is complete, the *Prove The Pilot* state is placed on the state shift register.

The *Prove The Pilot* state of the boiler controller is shown in Figure I.21. The `Prove The Pilot Done` sub-VI developed in Exercise 1.29 checks to see if the state is complete. The `Prove The Pilot Output` sub-VI developed in Exercise 1.30 generates the BOILER STATUS cluster and the log file entry for the state. When the state is finished, the *Start* state is placed on the state shift register.

The *Start* state of the boiler controller is shown in Figure I.22. The `Start Done` sub-VI developed in Exercise 1.31 checks to see if the state is done. The `Start Output` sub-VI developed in Exercise 1.32 generates the BOILER STATUS cluster and the log file entry for the state. When the state is finished, the *Run* state is placed on the state shift register.

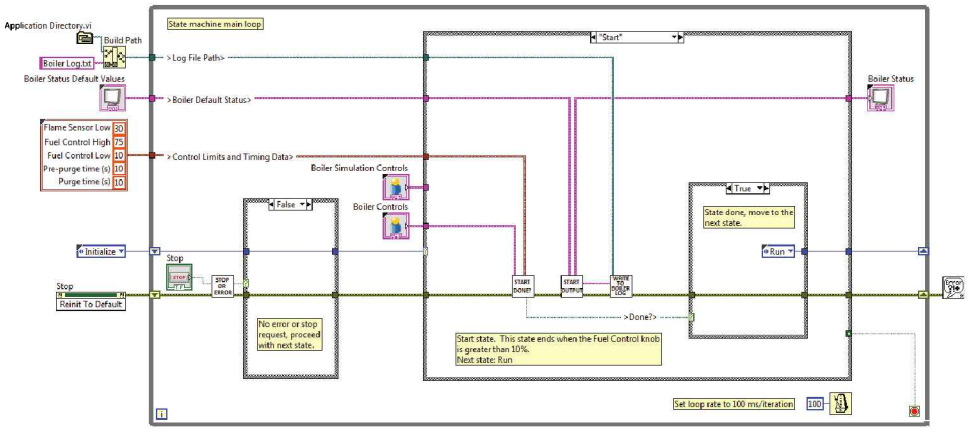


Figure I.22: The *Start* state of the boiler controller

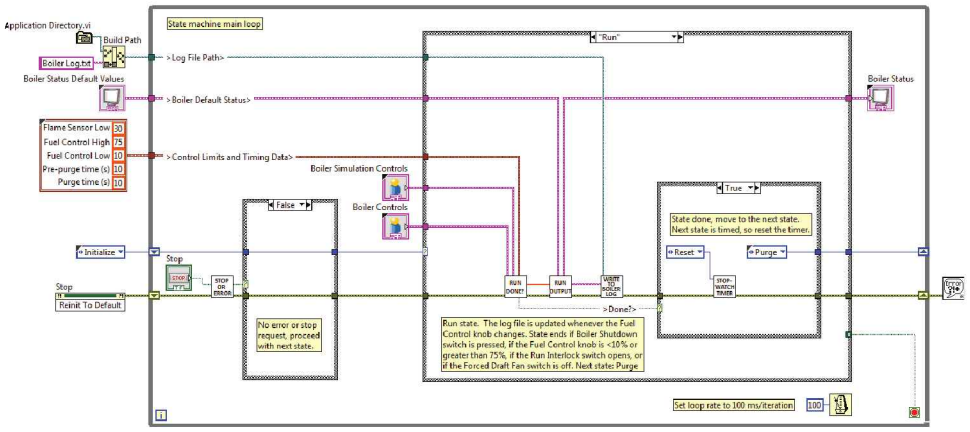


Figure I.23: The *Run* state of the boiler controller

The *Run* state of the boiler controller is shown in Figure I.23. The `Run Done` sub-VI developed in Exercise 1.33 checks to see if the state is finished. The `Run Output` sub-VI developed in Exercise 1.34 generates the BOILER STATUS cluster and the log file entry for the state. When the state is finished, the *Purge* state is placed on the state shift register. The timer is also reset to prepare for the *Purge* state, which is timed.

The *Purge* state of the boiler controller is shown in Figure I.24. The elapsed time is calculated and is passed to the `Purge Done` sub-VI developed in Exercise 1.35. The `Purge Output` sub-VI developed in Exercise 1.36 generates the BOILER STATUS cluster and the log file entry for the state. When the state is completed, the *Initialize* state is placed on the state shift register, and the boiler start-up sequence begins again.

If an error is detected or if the user presses the STOP button, the *Shutdown* state executes as shown in Figure I.25. The *Shutdown* state reinitializes the BOILER CONTROLS

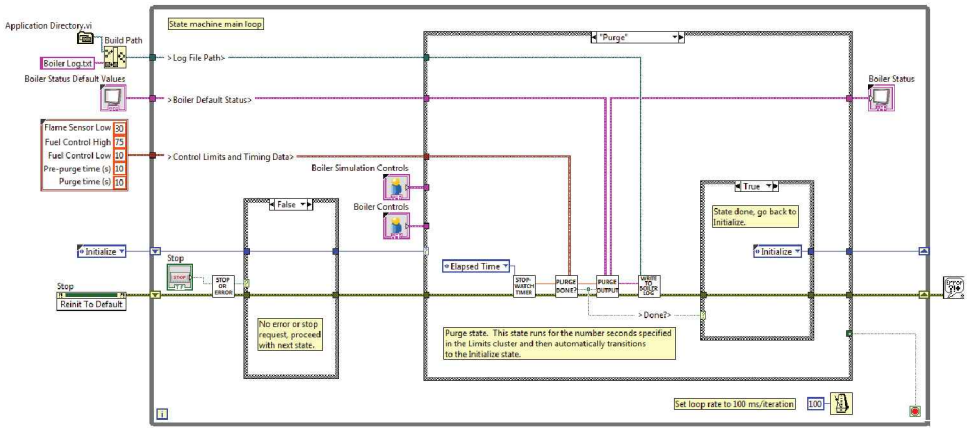


Figure I.24: The *Purge* state of the boiler controller

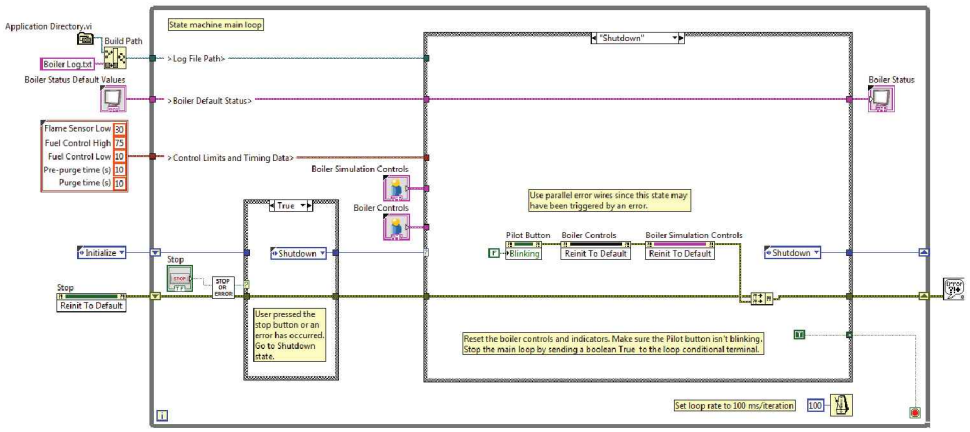


Figure I.25: The *Shutdown* state of the boiler controller

cluster and the BOILER SIMULATION CONTROLS cluster. This state also ensures that the PILOT BUTTON is not blinking. These actions are done on a parallel error wire to ensure that they execute if an error is detected on the main error wire. The main loop is shut down by wiring a Boolean True to the stop terminal of the main loop, and the error wire terminates with the **`General Error Handler`** function. This concludes the classic state machine implementation of the boiler controller.

Exercise 3.12

Although the two car wash controllers are superficially similar they are quite different. The main difference between the two is that the available car washes are fixed in the original car wash controller but they are variable in the second car wash controller.

This means that the state transitions in the first car wash controller are known ahead of time, but this is not the case in the second controller.

This makes the state transitions simpler in the first car wash controller. Many of the transitions were fixed, and some only needed the value of the DELUXE WASH Boolean to determine the next state. In the Car Wash 2 controller, we need to take a different approach. In this controller, we build an array of car wash option states based on user input. We store this array on the data highway along with an index variable. The state transitions are determined by the current index and the options array.

In the following implementation of the Car Wash 2 controller, the ENTRY CONSOLE also is placed on the data highway. The ENTRY CONSOLE does not need to be stored persistently throughout an entire wash, but it does need to be stored temporarily by one state and is then accessed by another. The data highway serves as a temporary storage location to place the ENTRY CONSOLE until it is needed the second state.

The main similarity between the two car wash controllers is in the similarity of the timing scheme. Both use essentially the same timing action engine. Both timers reference a position switch which can be used to pause the timer. Both have the same status cluster output to indicate if the current state is completed, paused, or running. Both block diagrams have the same structure as well. The CAR POSITION CONTROL, like the vehicle position switches in the original car wash controller, need to be read every loop iteration and need to be available to multiple states. This dictates that they should be placed in the lobby of the state machine. The STOP button must also be read every loop iteration, so it too should be placed in the lobby. The CAR WASH DISPLAY indicator cluster must be updated each loop iteration and needs to be accessed by multiple states. This suggests that it should be placed in the alley of the state machine.

The overall structure of the Car Wash 2 state machine and the *Initialize* state are shown in Figure I.26. The STOP button is reinitialized in the input area of the loop. This is important because the STOP button will be polled in the lobby of the state machine before the *Initialize* state executes. So if the STOP button is depressed before the program runs, it needs to be reinitialized before it is read during the first loop iteration.

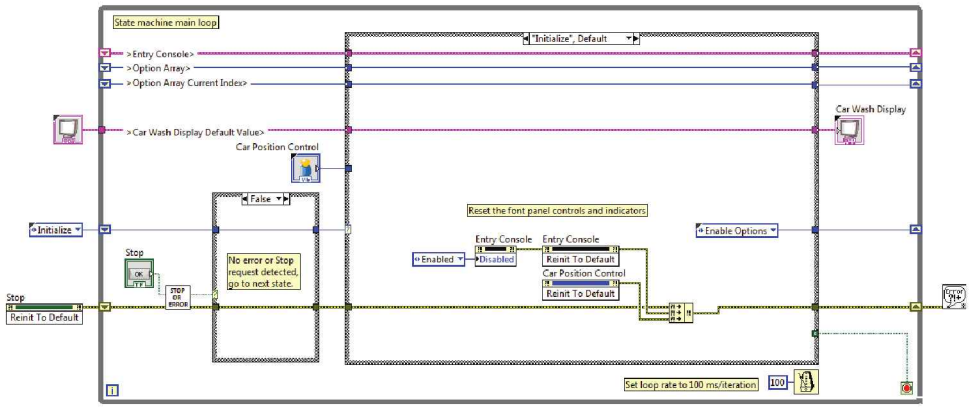


Figure I.26: The *Initialize* state of the Car Wash 2 controller

The state shift register is also initialized in the input area of the loop. A default constant of the CAR WASH DISPLAY indicator is also passed into the main loop as an input in the input area of the loop. Each state will access that cluster and make any necessary modifications and then pass the cluster to the CAR WASH DISPLAY in the alley of the state machine. Since the default value is all indicators dark, indicators that are lit in a previous loop iteration will be automatically reset in the next loop iteration due to the use of the default cluster as the baseline for setting the value of the CAR WASH DISPLAY.

The items in the lobby of the state machine are executed each loop iteration. The CAR POSITION CONTROL is placed in the lobby to ensure that it can be accessed by all of the car wash states. A check is made for errors or stop requests by the `Stop or Error` sub-VI. The output of this sub-VI is passed to the guard clause case structure. If no error is present, the current state is passed through.

If a stop request is made or if an error is detected, the guard clause bypasses the state shift register and sends the state machine directly to the *Shutdown* state, as shown in Figure I.27. This state enables the ENTRY CONSOLE, resets the front panel controls, and shuts down the main loop by wiring a Boolean True to the loop continuation terminal. The ENTRY CONSOLE property node is placed on a parallel error wire to ensure that it executes even if an error is present on the main error wire. The `Reinitialize All Controls` sub-VI was constructed to operate normally in the presence of an input error, so it is left on the main error wire. The *Shutdown* state wires the *Shutdown* state to the state shift register. This second *Shutdown* state will not be executed, because the main loop has been shut down, but something needs to be wired to the case structure output tunnel or the Run arrow will break.

Returning to Figure I.26, the *Initialize* state resets the front panel controls and then transitions to the *Enable Options* state shown in Figure I.28. The state transition takes place within 100 milliseconds or a multiple thereof due to the **`Wait Until Next ms Multiple`** function. The *Enable Options* state uses an implicitly linked property node to enable the ENTRY CONSOLE control. The state machine then transitions to the *Wait* state shown in Figure I.29.

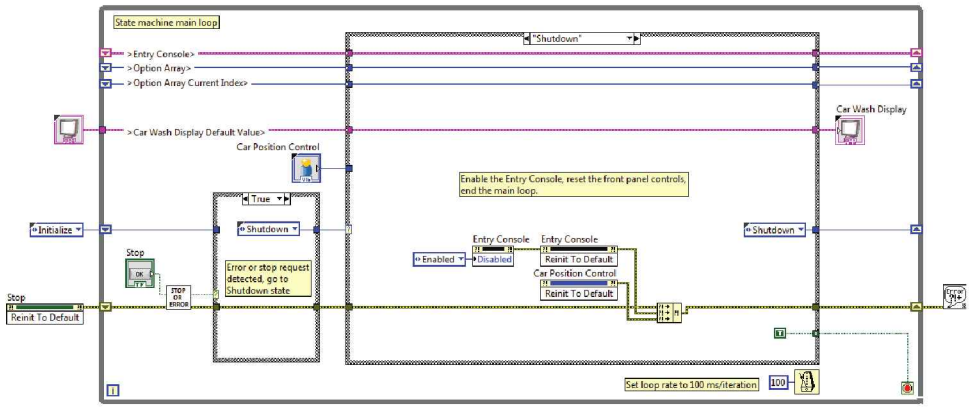


Figure I.27: The *Shutdown* state of the Car Wash 2 controller

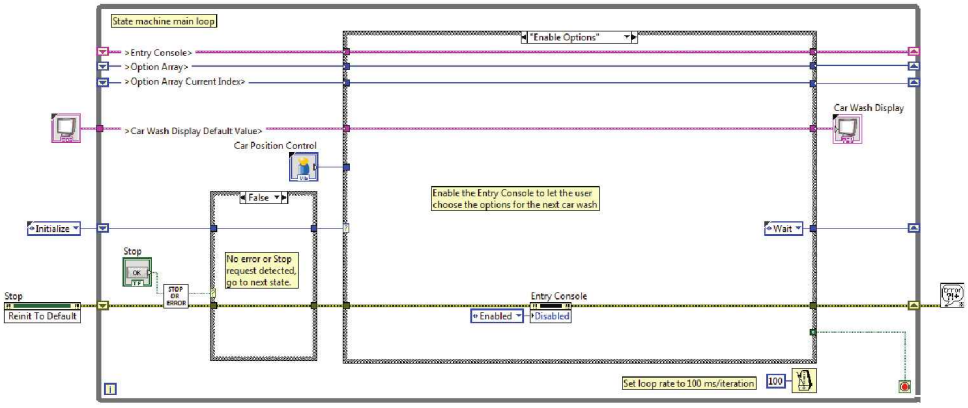


Figure I.28: The *Enable Options* state of the Car Wash 2 controller

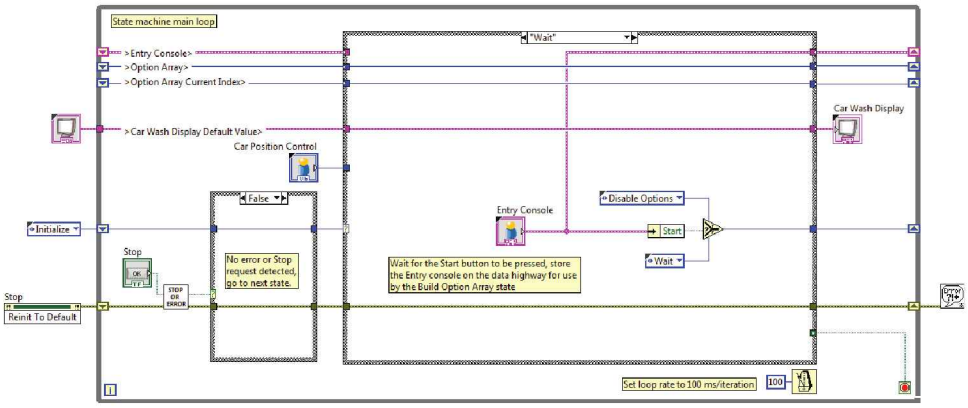


Figure I.29: The *Wait* state of the Car Wash 2 controller

The *Wait* state executes while the user is selecting wash options. The user signals that the options have been selected by pressing the START button. The controller continually cycles back to the *Wait* state until this button has been pressed. With each loop iteration, the value of the ENTRY CONSOLE is placed on the data highway. This value will be needed later by the *Build Option Array* state. The data highway serves as a temporary storage location for the ENTRY CONSOLE value. When the START button is pressed, the controller transitions to the *Disable Options* state shown in Figure I.30. This state uses an implicitly linked property node to disable and gray out the ENTRY CONSOLE control.

The controller then transitions to the `Build Option Array` sub-VI shown in Figure I.31. The WASH OPTIONS cluster is unbundled from the ENTRY CONSOLE cluster that was stored on the data highway in the *Wait* state. The `Build Option Array` sub-VI is then used to convert the wash option selections into an array of wash option states. This array of states is then placed on the data highway.

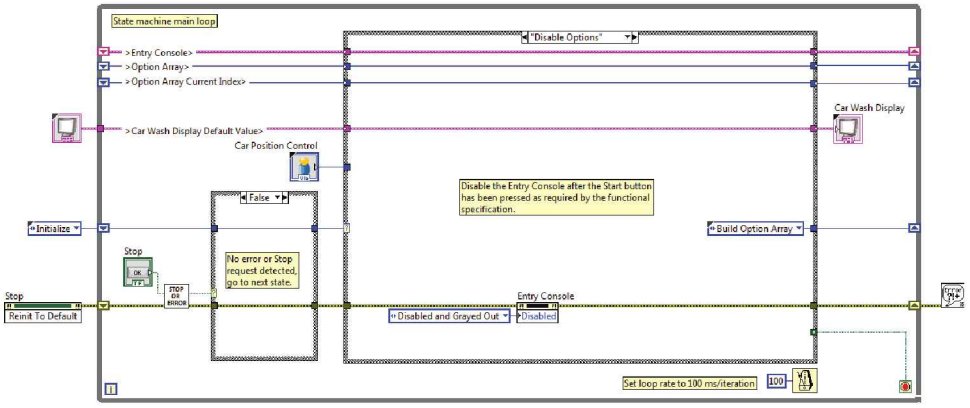


Figure I.30: The *Disable Options* state of the Car Wash 2 controller

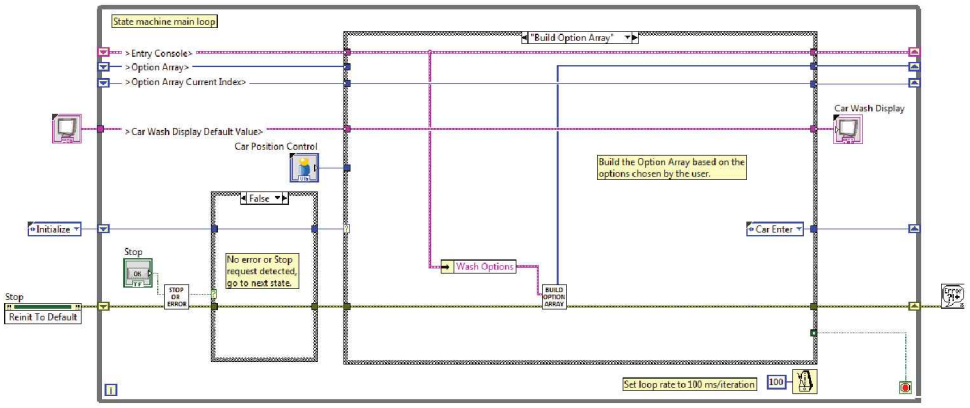


Figure I.31: The *Build Option Array* state of the Car Wash 2 controller

The controller then transitions to the *Car Enter* state shown in Figure I.32. This state uses the `Car In Position` sub-VI to check if the CAR POSITION CONTROL slider is at the *Entry* position. Notice how readable this is. Rather than check to see if the slider is at zero and require the reader to remember what position zero signifies, the `Car In Position` sub-VI compares the slider position to an enumerated constant. This makes it more obvious that the code is checking the slider to see if it is in the *Entry* position. If the vehicle is not in the *Entry* position, a Boolean True is sent to the WASH ENTRY and OUT OF POSITION indicators.

The *Car Enter* state is placed back on the state shift register until the CAR POSITION CONTROL is in the *Entry* position. When the vehicle is in position, a Boolean False is sent to the WASH ENTRY and OUT OF POSITION indicators as seen in Figure I.33. The **`Index Array`** function is then used to select the first element of the options array and the timer is reset to prepare for the first option state. The first index, `0`, is also placed on the data highway.

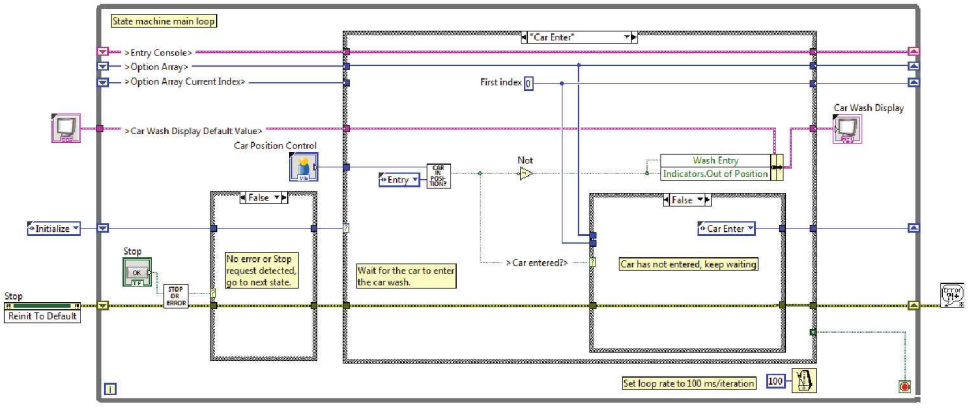


Figure I.32: The *Car Enter* state of the Car Wash 2 controller when the car has not entered

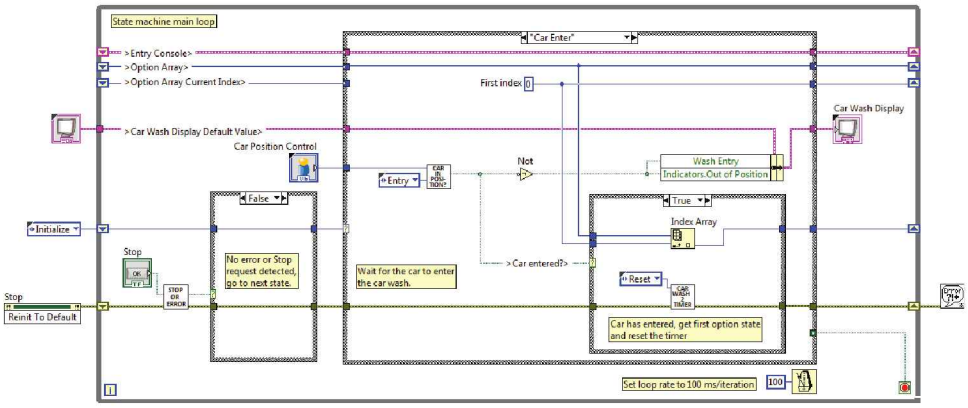


Figure I.33: The *Car Enter* state of the Car Wash 2 controller when the car has entered

At this point, the controller state transitions are dictated by the array of options on the data highway. All of the wash option states are similar, and only a few will be shown here. The *Under Body Wash* state is shown in Figure I.34. This state demonstrates the general structure of all of the option states. First, the `Car In Position` sub-VI is used to check if the CAR POSITION CONTROL value is correct for the current option state. The slider needs to be in the *Station 1* position for the *Under Body Wash* state to execute. The Boolean output of the `Car In Position` sub-VI is passed to the `Car Wash 2 Timer` action engine. This sub-VI performs the necessary time check and produces a status cluster with *Running?*, *Paused?*, and *Done?* Boolean elements. The *Running?* element is assigned to the indicator lamp appropriate for this option state—in this case the UNDER BODY lamp. The *Paused?* output is assigned to the OUT OF POSITION indicator lamp. The *Done?* output is wired to the selector of a case structure. If the current state is not finished, the value of the *Done?* Boolean is then passed to the `Next Car Wash State` sub-VI along with the options array and current index

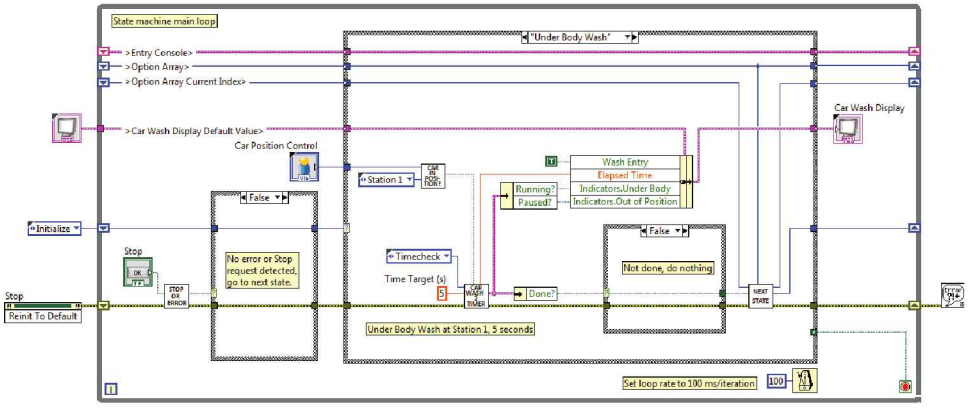


Figure I.34: The *Under Body Wash* state of the Car Wash 2 controller with the state not done

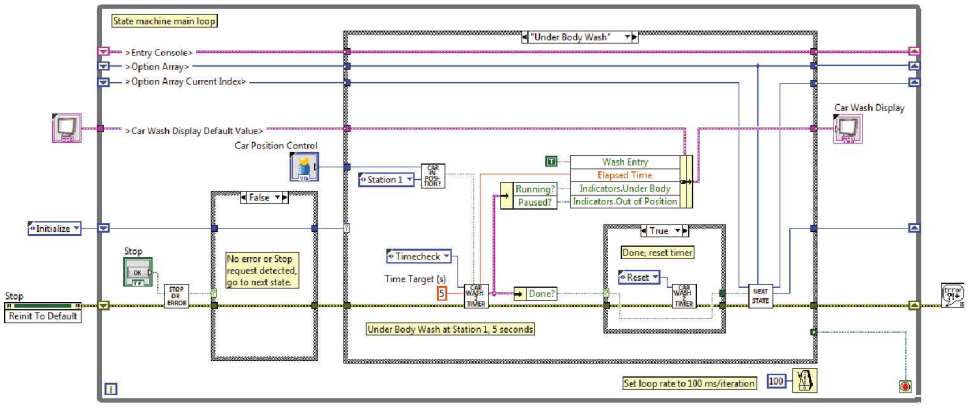


Figure I.35: The *Under Body Wash* state of the Car Wash 2 controller with the state done

from the data highway. The same thing happens if the current state is finished, but in addition, the car wash timer is reset to prepare for the next option state. This is shown in Figure I.35.

For comparison, the *High Pressure Wash* state is shown in Figure I.36, and the *Tire Shine Foam* state is shown in Figure I.37. These states are identical to the *Under Body Wash* state, where the only differences are in which slider position is referenced and which indicator lamps are lit.

When the options array is exhausted, the `Next Car Wash State` transitions the controller to the *Car Exit* state shown in Figure I.38. This state checks the value of the `Car Position Control` to see if it is at the *Exit* position. If not, the controller keeps looping back to the *Car Exit* state. When the vehicle exits, the controller returns to the *Initialize* state where the process begins again. This concludes the development of the classic state machine implementation of the Car Wash 2 controller.

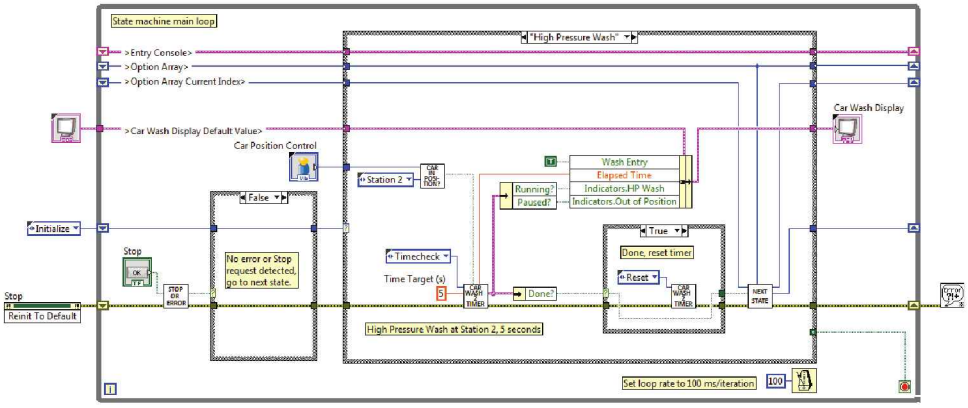


Figure I.36: The *High Pressure Wash* state of the Car Wash 2 controller with the state done

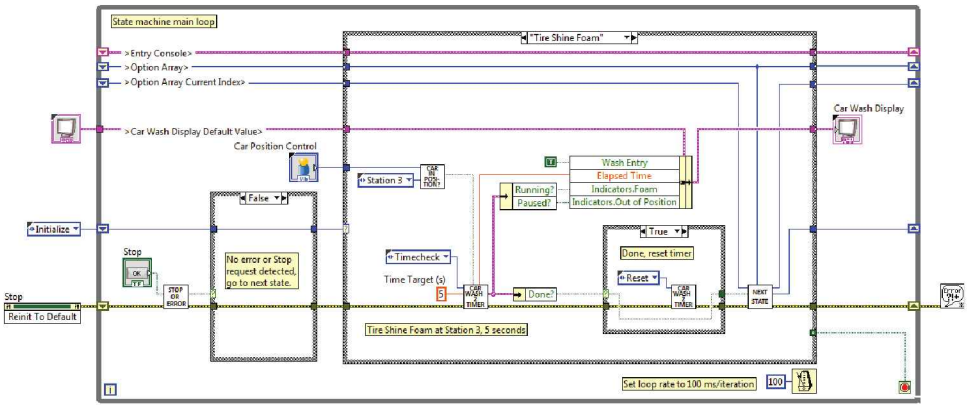


Figure I.37: The *Tire Shine Foam* state of the Car Wash 2 controller with the state done

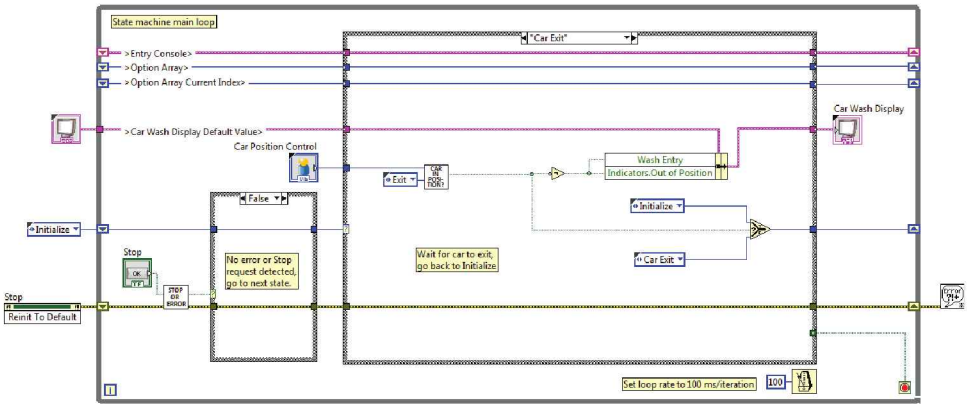


Figure I.38: The *Car Exit* state of the Car Wash 2 controller

J

Problem Set 4 Solutions

Exercise 4.1

The traffic light controller is not a good candidate for an event structure because the code never needs to wait for events. The controller cycles continuously. The flow of the program can be altered by the vehicle sensors, but these sensors are monitored every loop iteration without waiting. As a general rule of thumb, if a code does not have a *Wait* state then it is not a good candidate for an event structure.

Exercise 4.2

The security system controller is an excellent candidate for an event structure upgrade. The controller spends most of its time monitoring the switches waiting for changes to occur. The classic controller cycles continuously through the *Monitor* state, even when no switches have been tripped. An event structure would allow the controller to sleep until a switch is tripped.

Upgrading the controller is very simple. The only change that needs to made is the addition of an event structure in the *Monitor* state, as shown in Figure J.1. This figure shows the event case handling the value change of the TAMPER, BYPASS, and ALARM switches.

The configuration of this event case is shown in Figure J.2. Notice that the *<All Elements>* entry under the ALARM switches cluster has been selected. Any of the ALARM switches in the cluster will trigger this event case. Likewise for the BYPASS and TAMPER switches.

An event case also needs to be added to handle the value change of the STOP button. This event case is shown in Figure J.3.

Notice that the STOP button is now checked in two places in the program: the event case and the lobby of the state machine. Is this really necessary? The security system controller spends the vast majority of its time in the *Monitor* state. If the STOP button is pressed, it will almost certainly be while the controller is in the *Monitor* state, and the event structure will send the state machine directly to *Shutdown*. There is only one scenario where this will not happen. Suppose the user starts the program and immediately hits the stop button within the first 100 ms loop iteration. If the STOP

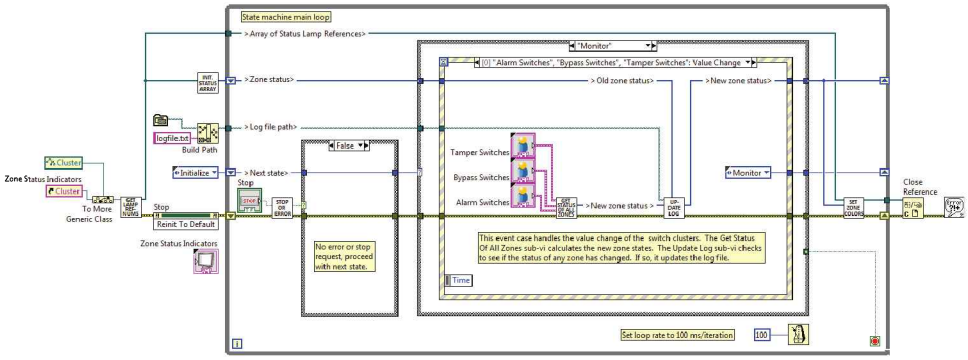


Figure J.1: The event case handling the value change of the switches, STOP button in lobby

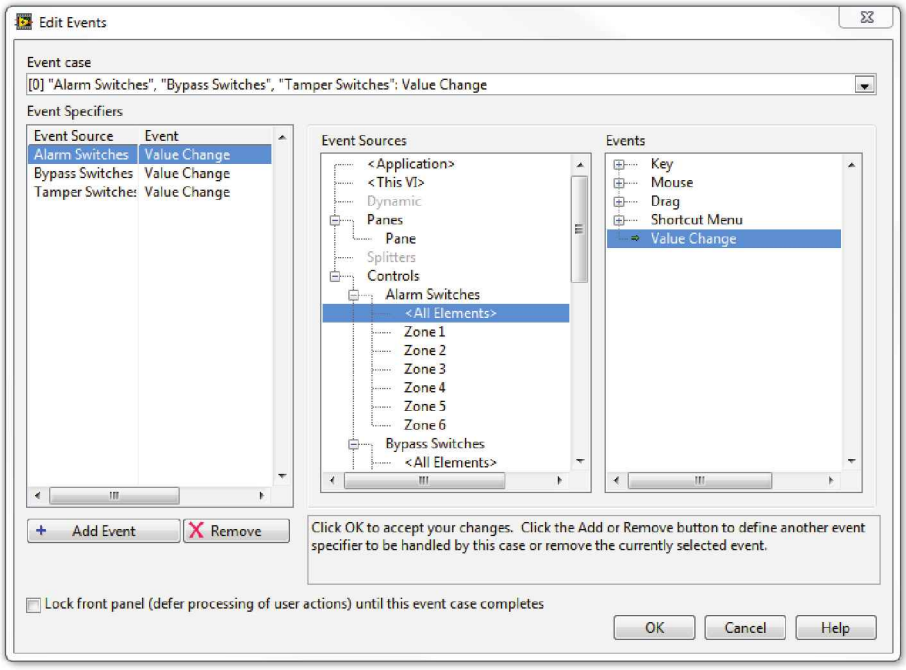


Figure J.2: Configuration of the event case handling the value change of the switches, STOP button in lobby

button is in the lobby, there is a chance that the value change will be caught in the first loop iteration, sending the state machine to the *Shutdown* state during the first loop iteration. Or perhaps the value change would be caught after executing the *Initialize* state, sending the state machine to *Shutdown* on the second loop iteration.

If the STOP button is placed in the event structure instead, the state machine will first execute the *Initialize* state and then execute the *Monitor* state. The value

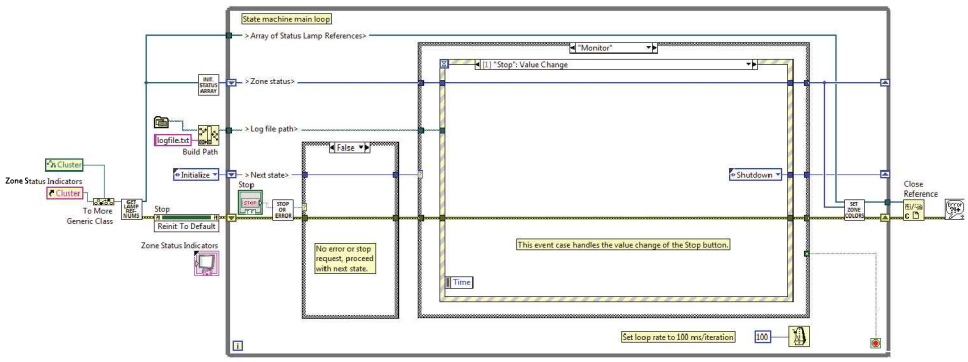


Figure J.3: The event case handling the value change of the STOP button of the security system controller

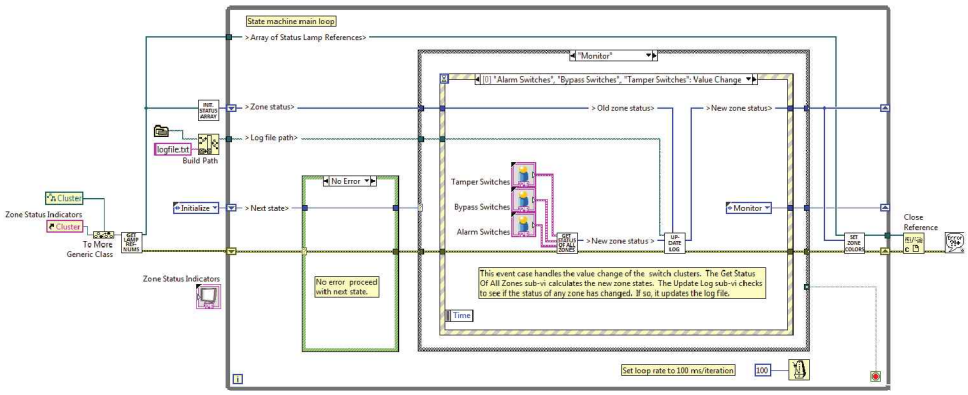


Figure J.4: The event case handling the value change of the switches of the security system controller

change of the STOP button will be processed, and the state machine will execute the *Shutdown* state on the third loop iteration. So, there is a very small chance of a very small performance hit in terms of responsiveness to the STOP button at the very start of the program if the STOP button is moved into the event structure. The functional specification does not cover this particular point. If we assume that this is acceptable, we can move the STOP button to the event case handling its value change. This ensures that the button will be read when it is pressed and that its mechanical action will be triggered.

The benefit of making this move is that can simplify the guard clause to check for errors only as seen in Figure J.4. The `Stop or Error` sub-VI can be removed entirely. The *Reinitialize To Default* invoke node of the STOP button can also be removed. The purpose of this node was to reset the STOP button before it was checked in the lobby during the first loop iteration. If the STOP button is not in the lobby, this is no longer necessary. All of these factors improve the readability of the code. The STOP button is

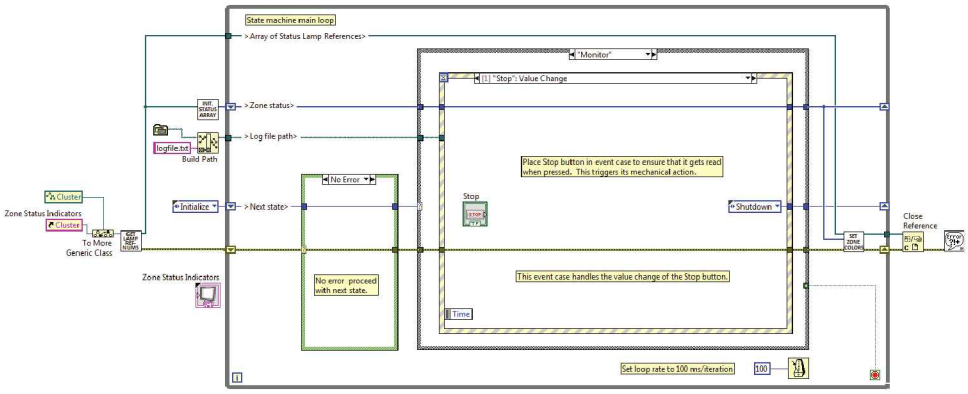


Figure J.5: The event case handling the value change of the STOP button of the security system controller

now in the event case handling its value change as seen in Figure J.5. This completes the event-driven upgrade of the security system controller.

Exercise 4.3

The boiler controller is not a good candidate for an event structure. The state transition diagram for the boiler controller is like a ladder, and each step of the ladder waits for a unique condition to be fulfilled before advancing to the next step. The boiler controller would need event structures in almost every boiler process step, and that is not recommended.

Exercise 4.4

The upgrade to an event-driven Car Wash 2 controller is very easy, simply add an event case to the *Wait* state as shown in Figure J.6. The event case shown handles the value change of the START button. Notice the "dot" notation in the event case selector. This event case does not handle the value change of the entire ENTRY CONSOLE cluster, it simply handles the value change of the START button within that cluster.

The user could press the STOP button while this event case is active, so an event case needs to be configured to handle this possibility. The event case handling the value change of the STOP button is shown in Figure J.7. This event case sends the controller to the *Shutdown* state.

Adding event structures to the *Car Enter* and *Car Exit* states is not recommended. It is generally not recommended to have more than one event structure in a code. For example, event structures in these states would both need to monitor the value change of the CAR POSITION CONTROL slider. Any changes to the slider would be placed on the event queues, even when their states were not active. The changes would be processed once the event structures became active. Since the slider changes often during the course of the car wash, this could lead to unintended consequences.

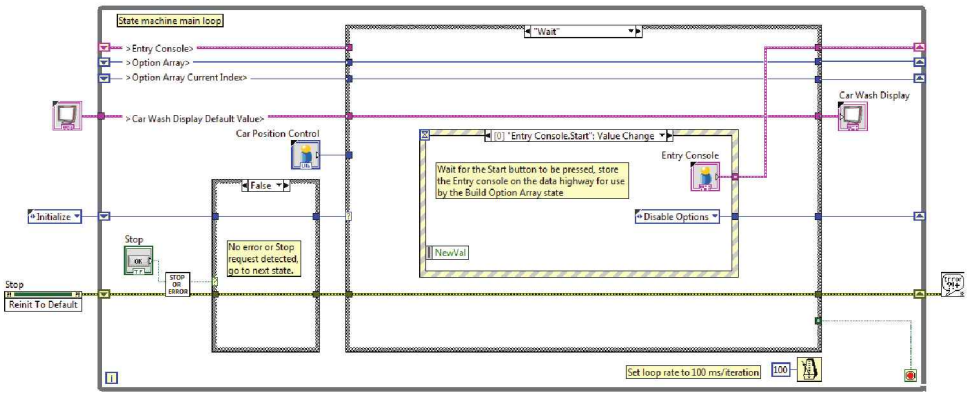


Figure J.6: The event case of the Car Wash 2 controller handling the value change of the START button

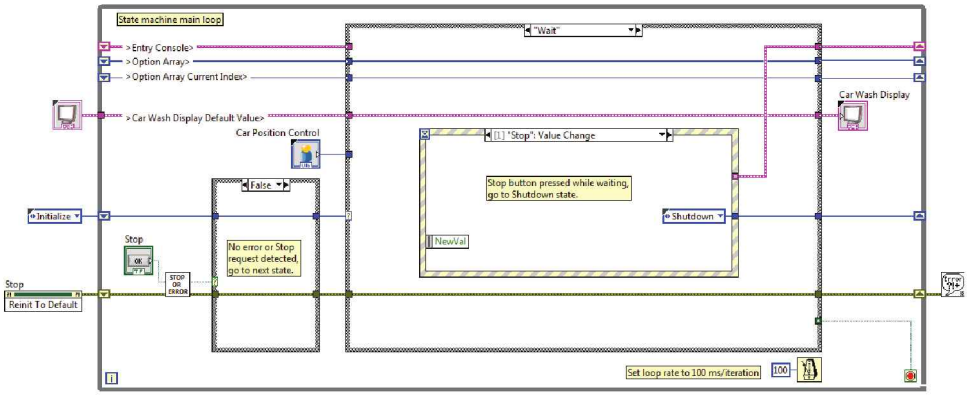


Figure J.7: The event case of the Car Wash 2 controller handling the value change of the STOP button

Exercise 4.5

The event case handling the value change of the START button for the revised event-driven Car Wash 2 controller is shown in Figure J.8. The ENTRY CONSOLE has been removed from the event case and is no longer on the data highway. The control has been moved to the *Build Option Array* state, as shown in Figure J.9.

The advantage of moving the ENTRY CONSOLE is that the control is now located where it is needed and is no longer on the data highway, reducing the persistent storage in the main application and simplifying the code. The advantage of the implementation in Exercise 4.4 is more philosophical. When a program receives a command, it is generally good practice to freeze the data needed by that command at the time that the command was issued. In this case, this means that the wash options should be locked down as soon as the START button is pressed. Putting the wash options on

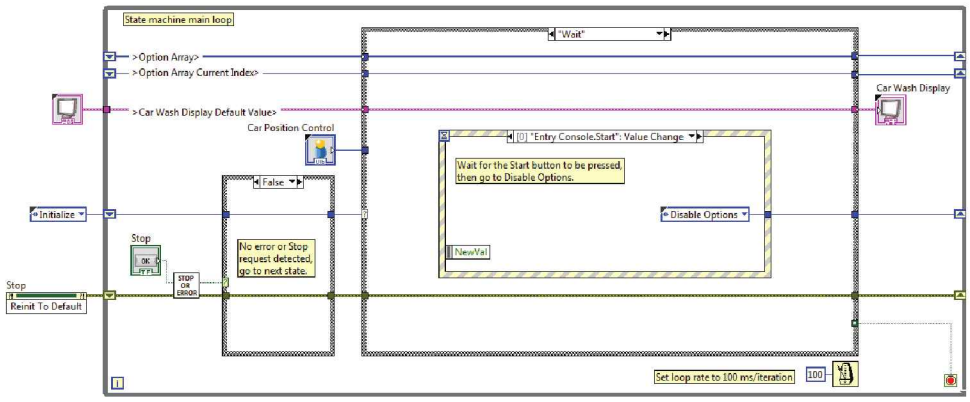


Figure J.8: The event case of the revised Car Wash 2 controller handling the value change of the START button

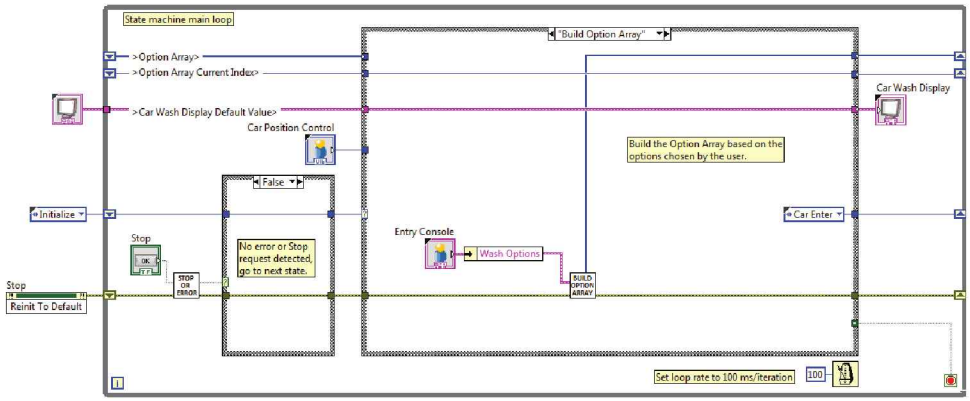
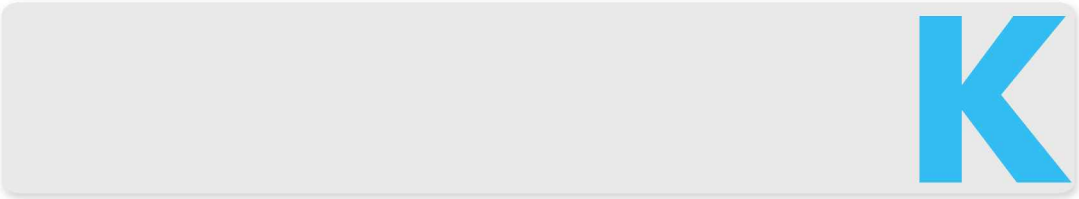


Figure J.9: The *Build Option Array* state of the revised Car Wash 2 controller

the data highway preserves the value of the wash options at the time that the START button was pressed.

In this case though, locking down the data on the data highway is not as important. The functional specification requires that the ENTRY CONSOLE control be disabled as soon as the START button is pressed. Disabling the control locks its data and preserves the value of the selected wash options until the control is accessed in the *Build Option Array*. This makes the revised version of the event-driven Car Wash 2 controller safe to use. If the wash options were not required to be disabled when the START button was pressed, it would be essential to lock them down in some other way such as by placing them on the data highway.



Problem Set 5 Solutions

Exercise 5.1

The traffic light controller switches back and forth between two fixed sets of states: the normal light sequence and the left-turn sequence. As these sequences are short, the controller never waits, so there is no good place in the code to load up a series of states. There is no real advantage in readability or maintainability in introducing queues to the traffic light controller.

Exercise 5.2

Creating a dedicated state for updating the log file has advantages and disadvantages, but it is beneficial on the whole. The main benefit is that separating the task of monitoring the front panel from the task of updating the log file increases the cohesion of the code. Each task is performed by a dedicated state. This makes the code more maintainable in the future. The disadvantages are a slight performance degradation (an extra loop iteration is needed to update the log file) and the need to provide storage for both the old and new zone status arrays.

The *Initialize* state of an implementation of a queued security system controller is shown in Figure K.1. The queue is created in the input area of the loop. The queue element is an enumerated constant containing the states: *Initialize*, *Monitor*, *Update Log* and *Shutdown*. After the queue is created *Initialize* state is enqueued.

The references to the indicator lamps are also created in the input area. The references are used by the polymorphic `Initialize Status Array` VI to preallocate the zone status array as discussed in Exercise 3.10. The preallocated default zone status array is then used to initialize the shift registers of the two items on the data highway: the old and new zone status arrays.

The guard clause in the main loop checks to see if an error has occurred. If no error has been detected, the guard clause dequeues the next state and passes it to the main case structure. The first state to be dequeued is the *Initialize* state. This state resets the front panel switches and enqueues the *Monitor* state.

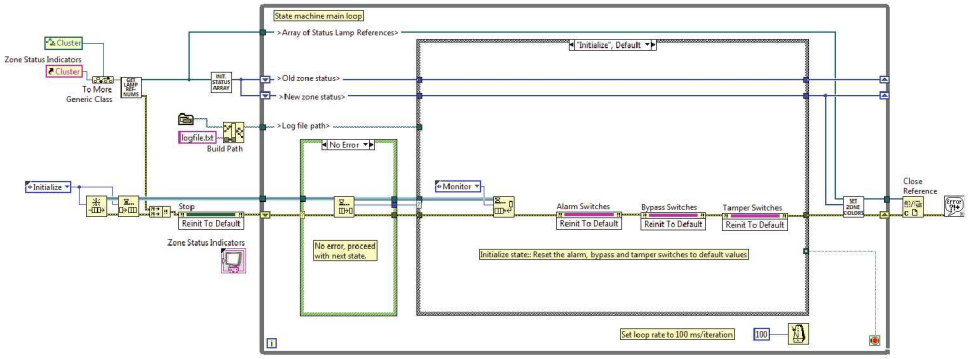


Figure K.1: The *Initialize* state of the queued security system controller

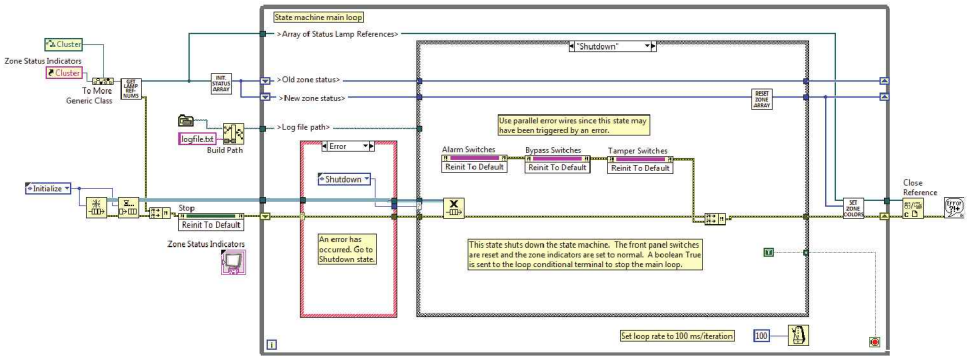


Figure K.2: The *Shutdown* state of the queued security system controller

If the guard clause detects an error, it bypasses the queue and sends the state machine directly to the *Shutdown* state, as seen in Figure K.2. The *Shutdown* state releases the queue, resets the front panel switches and the zone status array, and shuts down the loop.

The security system spends most of its time in the *Monitor* state, which contains the event structure monitoring the user interface. The event case handling the value change of the front panel switches is shown in Figure K.3. The new zone states are calculated and are placed on the data highway. The *Update Log* state is then placed on the queue. The *Update Log* state updates the log file and returns to the *Monitor* state, as shown in Figure K.4.

The *Shutdown* state is enqueued if the STOP button is pressed while the controller is in the *Monitor* state, as seen in Figure K.5. The *Shutdown* state is enueued normally; it is not put at the front of the queue for immediate execution. This is to allow any log requests already on the queue to be processed before the state machine shuts down. This concludes the queued version of the security system controller.

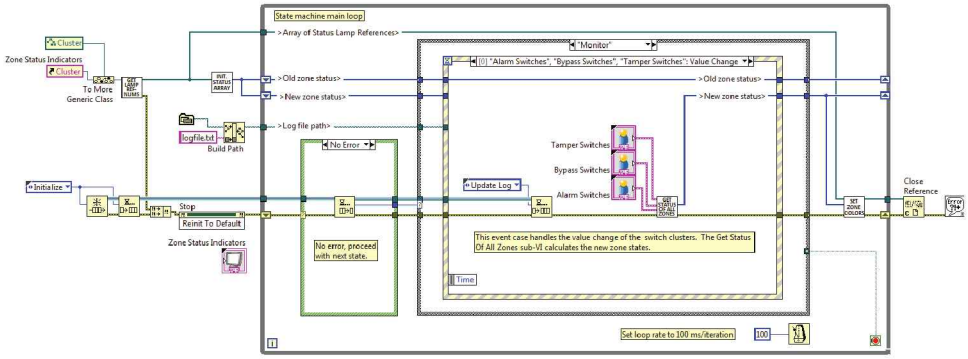


Figure K.3: The event case handling the value change of the switches of the queued security system controller

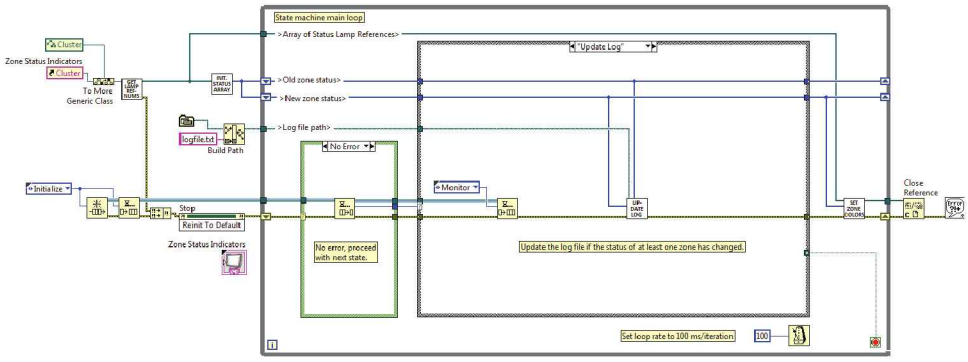


Figure K.4: The *Update Log* state of the queued security system controller

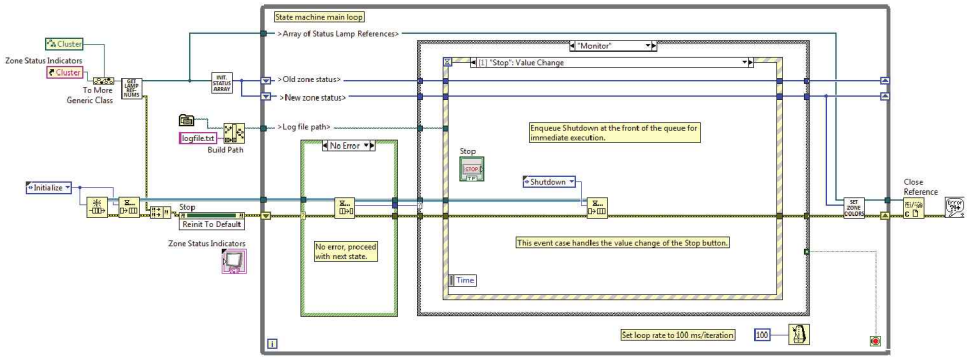


Figure K.5: The event case handling the value change of the STOP button of the queued security system controller

Exercise 5.3

The *Set Status* action of one possible implementation of the `Update Log Action Engine` sub-VI is shown in Figure K.6. The action engine uses a feedback node to provide persistent storage of the old zone state. When the *Set Status* action is performed, the new zone status input overwrites the stored zone state.

The *Update Log* action of the `Update Log Action Engine` sub-VI is shown in Figure K.7. When this action is performed, both the old and new zone status arrays are passed to the `Update Log` sub-VI used in previous versions of the security system controller. The `Update Log` sub-VI acts like a kernel for the action engine. It accepts the old zone status array as a state variable, acts upon it, and outputs an updated state variable. The new value of the state variable is written to the feedback node persistent storage.

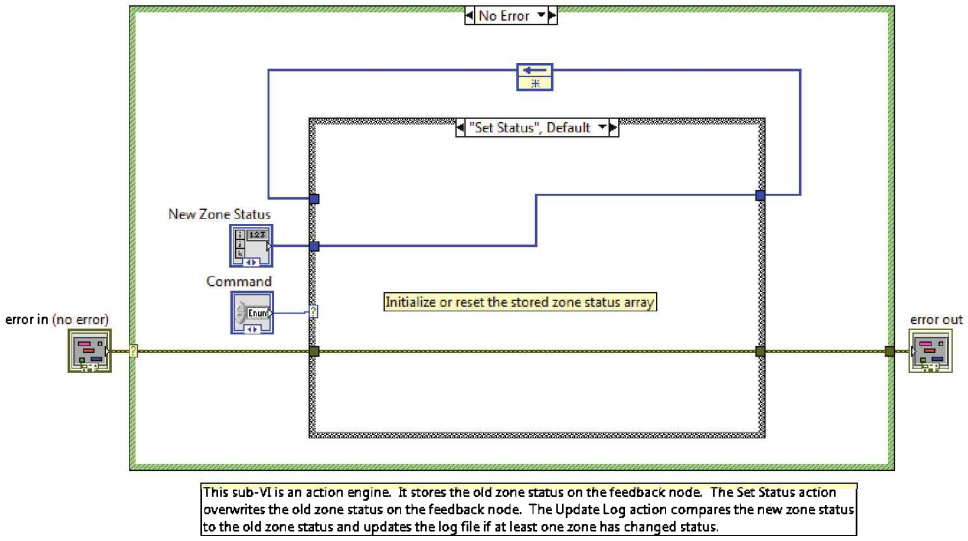


Figure K.6: The *Set Status* action of the `Update Log Action Engine` action engine

Exercise 5.4

Adding the `Update Log Action Engine` sub-VI to the queued security system controller is straightforward. The *Initialize* state of the revised security system controller is shown in Figure K.8. The old zone status has been removed from the data highway since it will now be stored in the action engine. The memory of the action engine needs to be initialized when the controller starts, so the preallocated default zone status cluster is passed to the action engine along with the *Set Status* command.

The *Update Log* state of the security system controller is shown in Figure K.9. The log file path and the new zone status array are passed to the action engine with

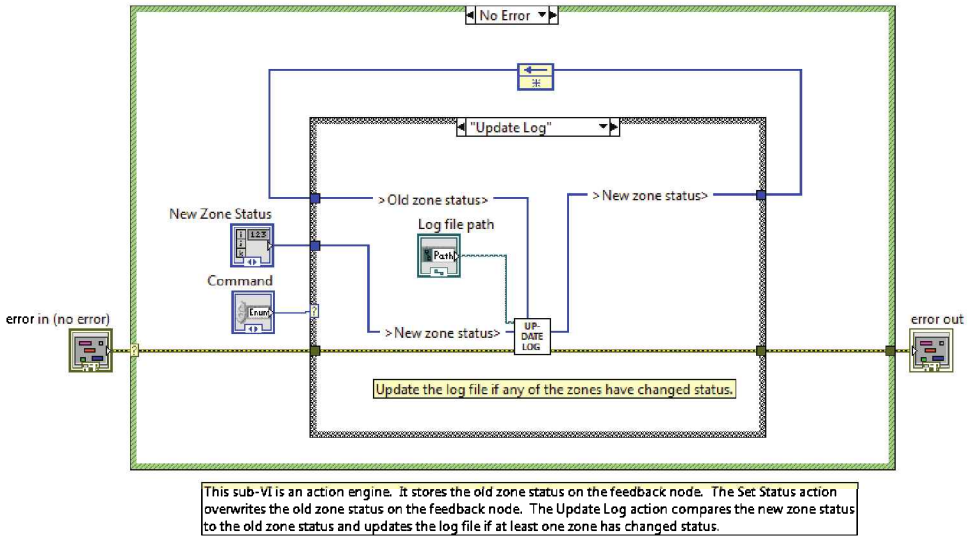


Figure K.7: The *Update Log* action of the `Update Log Action Engine` action engine

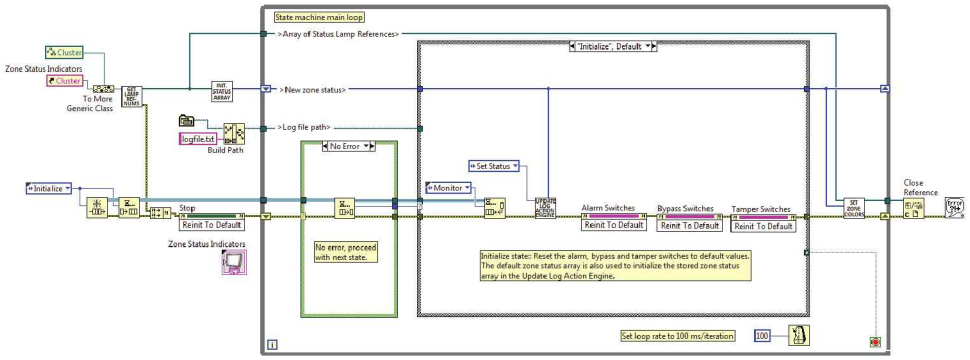


Figure K.8: The *Initialize* state of the queued security system controller with action engine

the *Update Log* command. The action engine compares the new zone status array with its stored zone status array and updates the log file if at least one zone has changed status.

The main difference between using the data highway to store the old zone status and using an action engine is the level of data exposure. The data highway is local to the main application. Data stored on the data highway cannot be accessed or changed from outside the program. However, an action engine is global by nature. Any VI running on the same computer can call the action engine and alter the stored data. The data is better encapsulated on the data highway. This topic was addressed in more detail in Chapter 23.

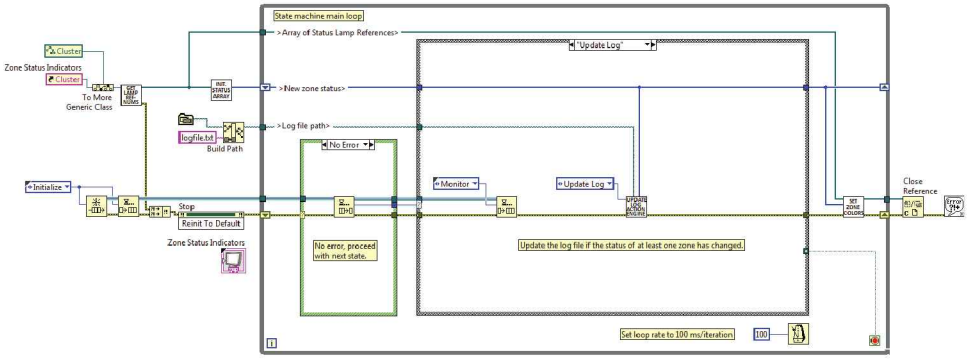


Figure K.9: The *Update Log* state of the queued security system controller with action engine

Exercise 5.5

The queued boiler controller is very similar to the classic state machine implementation, as can be seen in Figure K.10. This figure shows the *Initialize* state of the boiler controller. The queue is created in the input area of the loop using an enumerated constant containing the boiler controller states. The queue is initialized with the *Initialize* state. The queue is dequeued in the lobby of the state machine by the guard clause, assuming that the STOP button has not been pressed and no error has been detected.

The *Initialize* state resets the front panel controls using implicitly linked property nodes and ensures that the PILOT button is not blinking. The **`Enqueue Element`** function is used to enqueue the *Load Sequence* state that will load the sequence of states needed to run the boiler.

The *Load Sequence* state is shown in Figure K.11. This state enqueues the entire boiler sequence using a **`For`** loop and an array of boiler controller states. The states are

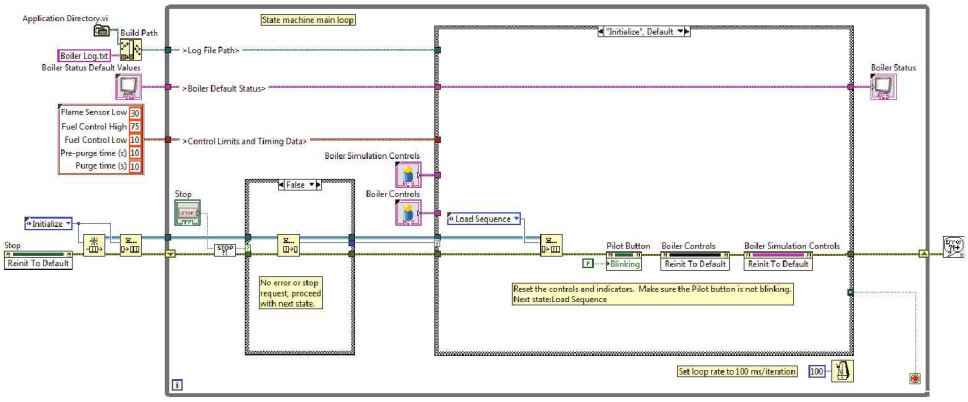


Figure K.10: The *Initialize* state of the queued boiler controller

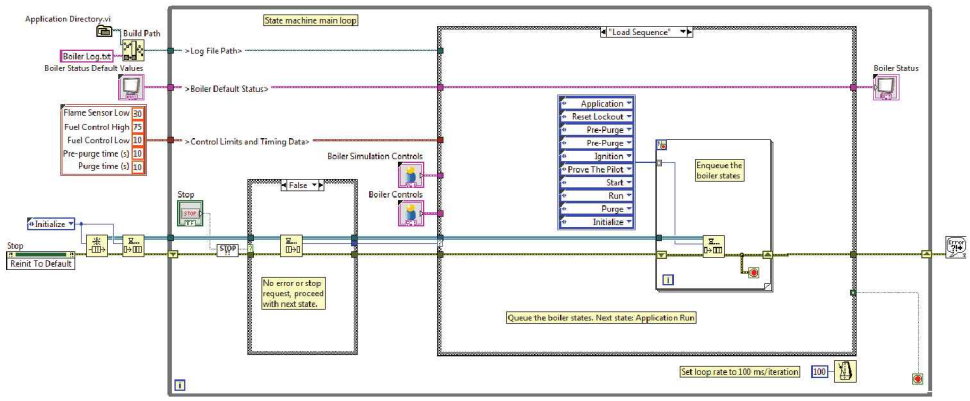


Figure K.11: The *Load Sequence* state of the queued boiler controller

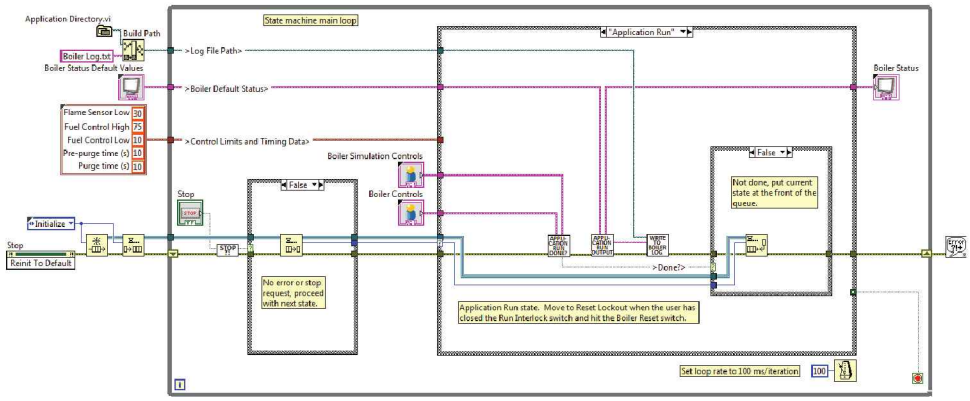


Figure K.12: The *Application Run* state of the queued boiler controller with the state not done

fed into the loop using auto-indexing and are enqueued using the **`Enqueue Element`** function. The *Initialize* state is placed at the end of the queue to allow the boiler to be re-run after it is stopped by the user.

The *Application Run* state is shown in Figure K.12. If the state is not finished, the current state is placed back on the front of the queue using the **`Enqueue Element At Opposite End`** function. If the state is finished, then the current state is allowed to expire, and a new state is dequeued in the guard clause at the next loop iteration, as seen in Figure K.13. The other boiler process states are similar and will not be shown here.

If an error is detected or the user presses the STOP button, the guard clause bypasses the queue and sends the state machine directly to the *Shutdown* state, as shown in Figure K.14. The queue is released using the **`Release Queue`** function. This function executes even if an error is present at its error input terminal, so it does not need to be placed on a parallel error wire. This concludes the queued version of the boiler controller.

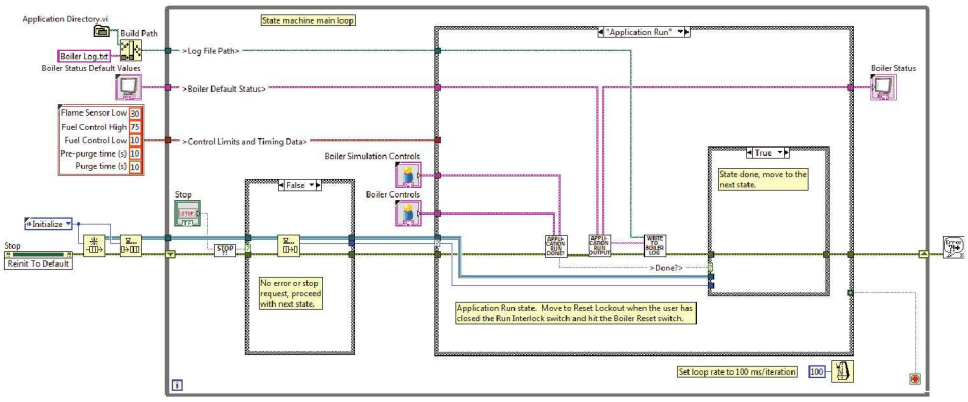


Figure K.13: The *Application Run* state of the queued boiler controller with the state done

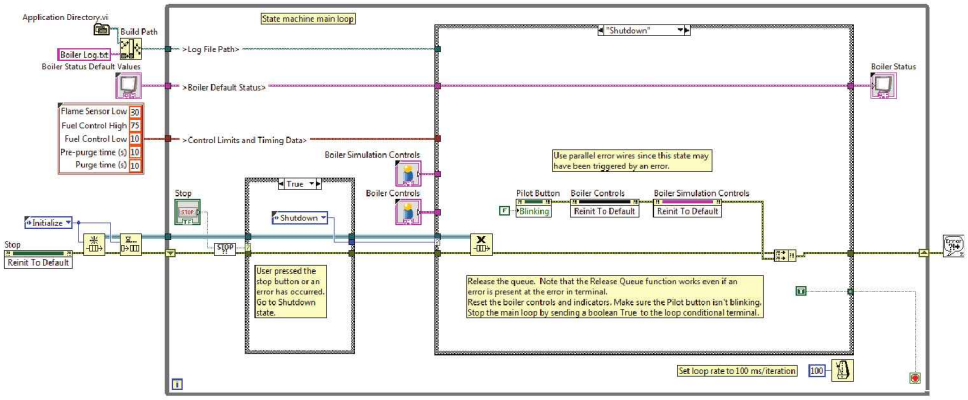


Figure K.14: The *Shutdown* state of the queued boiler controller

Notice that queues do not add functionality in this case. The series of states in the boiler controller is fixed, so the flexibiity added by queues is not needed. The queue does decouple each state from the other states, though. The individual states do not need to know what state comes next. Also, readability is enhanced because the entire sequence of states is visible in the *Initialize* state. This makes it easier for the reader to visualize the entire boiler process.

Exercise 5.6

One possible implementation of the `Build Wash Array` sub-VI is shown in Figure K.15. The logic is identical to that of the `Build Option Array` sub-VI shown in Figure G.35, except the array constant of states contains the new version of the state enumerated constant that includes the *Queue Wash Sequence* state. The **`Build Array`** is used to place the *Car Entry* state before the first selected option state and to place the *Car Exit* state after the last selected option state in the output array.

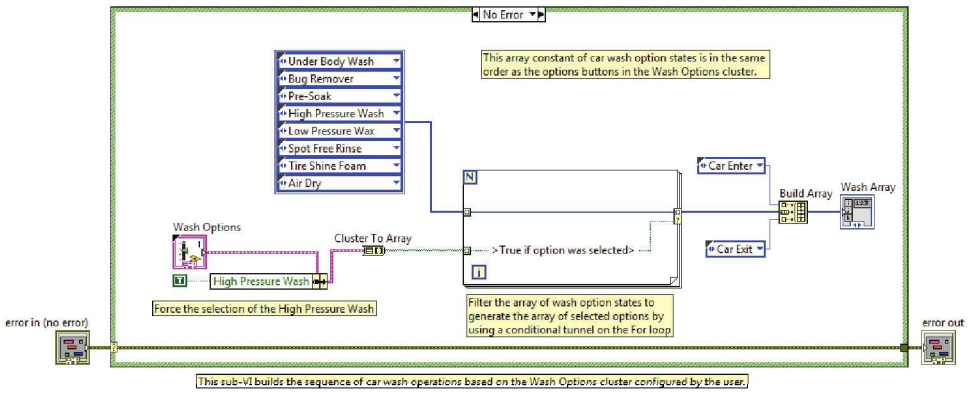


Figure K.15: The block diagram of the `Build Wash Array` sub-VI

Exercise 5.7

The overall structure of the queued Car Wash 2 controller is shown in Figure K.16 along with the *Initialize* state. The queue is created in the input area of the main loop and the *Initialize* state is enqueued. The enumerated constant containing the states is different than the one used in the classic state machine and event-driven implementations of the Car Wash 2 controller. The *Build Option Array* state has been replaced with the *Queue Wash Sequence State*. Notice that the data highway has been removed entirely, since the option array and index are no longer needed.

The short-icon version of the `Stop or Error` sub-VI is used in the lobby of the state machine in order to not interfere with the queue reference wire. If no error or stop request is detected, the guard clause case structure dequeues the next state. If an error is detected or if a stop request is made, the guard clause bypasses the queue and sends the state machine to the *Shutdown* state, as shown in Figure K.17. This state releases the queue, enables the ENTRY CONSOLE, resets the front panel controls, and stops the main loop.

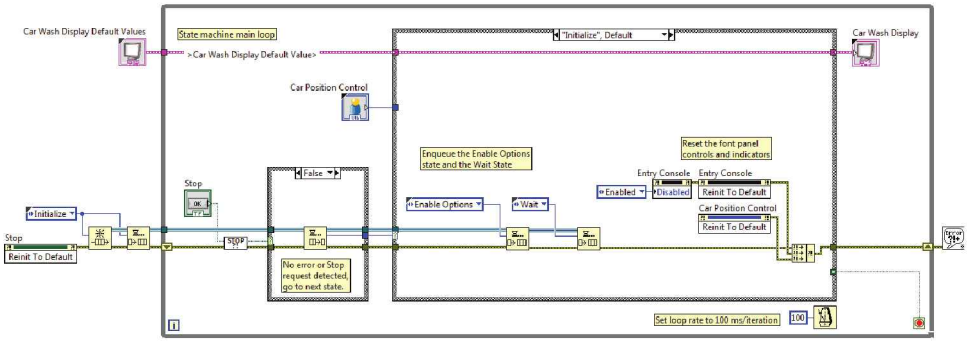


Figure K.16: The *Initialize* state of the queued Car Wash 2 controller

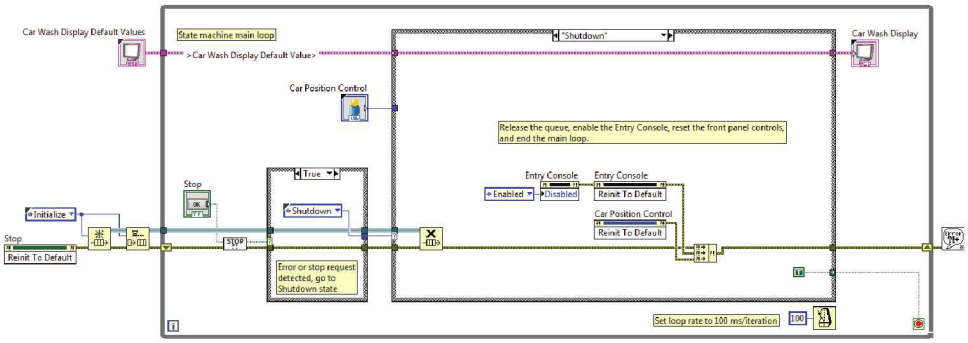


Figure K.17: The *Shutdown* state of the queued Car Wash 2 controller

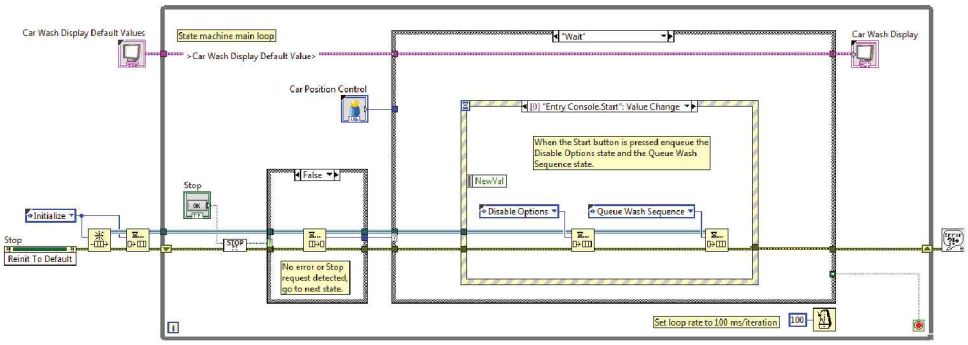


Figure K.18: The event case handling the value change of the START button in the queued Car Wash 2 controller

Returning to Figure K.16, the *Initialize* state initializes the front panel controls and enqueues the *Enable Options* and *Wait* states. The states enabling and disabling the options are not affected by the change to queues and will not be shown here.

The event case handling the value change of the START button in the *Wait* state is shown in Figure K.18. The car wash will always transition to the *Disable Options* state and the *Queue Wash Sequence States*, so these two states are placed on the queue. The event case handling the value change of the STOP button in the *Wait* state is shown in Figure K.19. This event case places the *Shutdown* state at the front of the queue for immediate execution using the **`Enqueue At Opposite End`** function.

The *Queue Wash Sequence* state is shown in Figure K.20. The `Build Wash Array` sub-VI developed in Exercise 5.6 is used to create an array of car wash states that starts with the *Car Entry* state, is followed by the selected wash option states, and ends with the *Car Exit* state. This array is then wired to an auto-indexed input tunnel of a **`For`** loop containing an **`Enqueue Element`** function, enqueuing each element of the array in its proper order.

The first state to execute after the *Queue Wash Sequence* state is the *Car Entry* state shown in Figure K.21. If the `Car Position Control` slider is not in the *Entry*

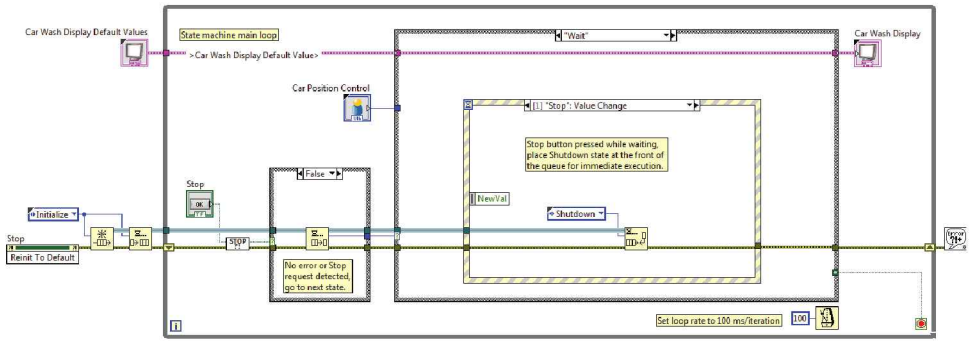


Figure K.19: The event case handling the value change of the STOP button in the queued Car Wash 2 controller

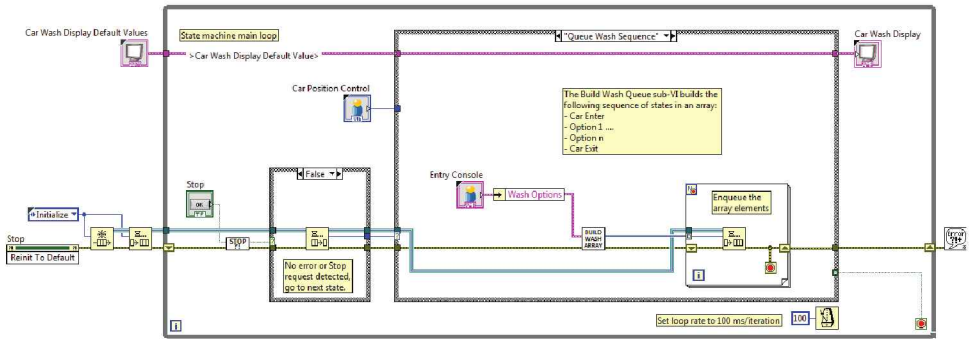


Figure K.20: The *Queue Wash Sequence* state of the queued Car Wash 2 controller

position then the *Car Entry* state is requeued at the front of the queue using the **`Enqueue Element At Opposite End`** function. This continues until the car is in the entry position, as shown in Figure K.22. In this case, the timer is reset and the next state (the first option state) is dequeued normally.

The selected wash option states are executed next. As an example, the *Under Body Wash* state is shown in Figure K.23. If the state is not completely it is requeued at the front of the queue using the **`Enqueue Element At Opposite End`** function. If the state is finished, the timer is reset but the queue is not touched, as shown in Figure K.24. The next state will dequeue normally. The other wash option states are nearly identical and will not be shown here. The only difference in the other wash option states is which slider position is referenced and which indicator lamps are lit.

When the option states have all been completed, the *Car Exit* state executes as shown in Figure K.25. If the `Car Position Control` slider is not in the *Exit* position, the *Car Exit* state is requeued at the front of the queue using the **`Enqueue Element At Opposite End`** function. This continues until the car is in the exit position, as shown in Figure K.26. In this case, the controller transitions back to the *Initialize* state. This concludes the development of the queued Car Wash 2 controller.

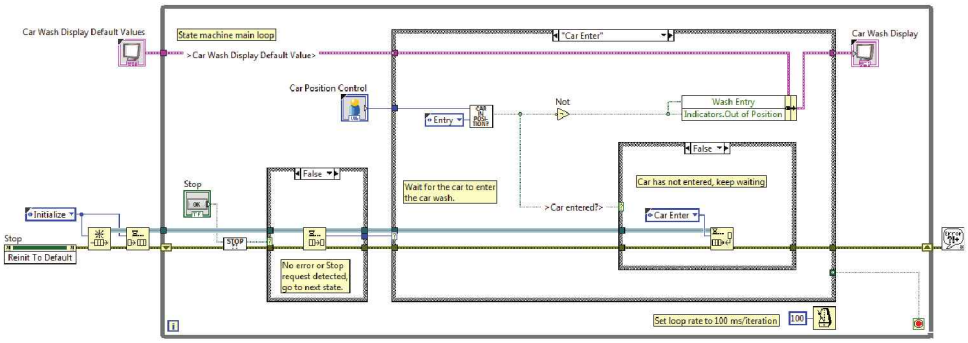


Figure K.21: The *Car Entry* state of the queued Car Wash 2 controller with the vehicle not yet entered

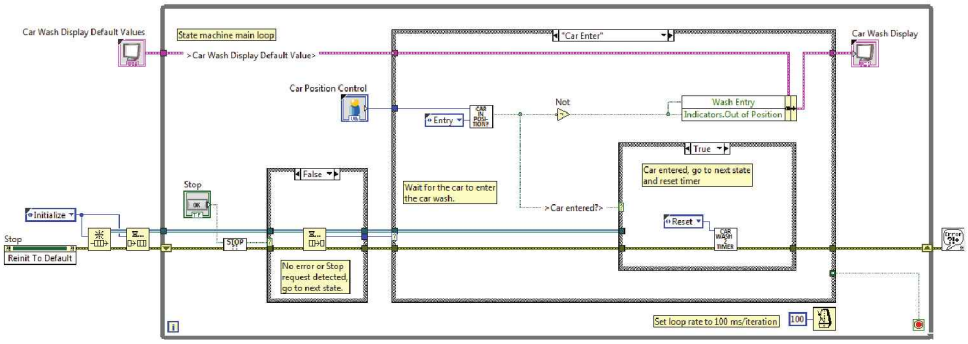


Figure K.22: The *Car Entry* state of the queued Car Wash 2 controller with the vehicle entered

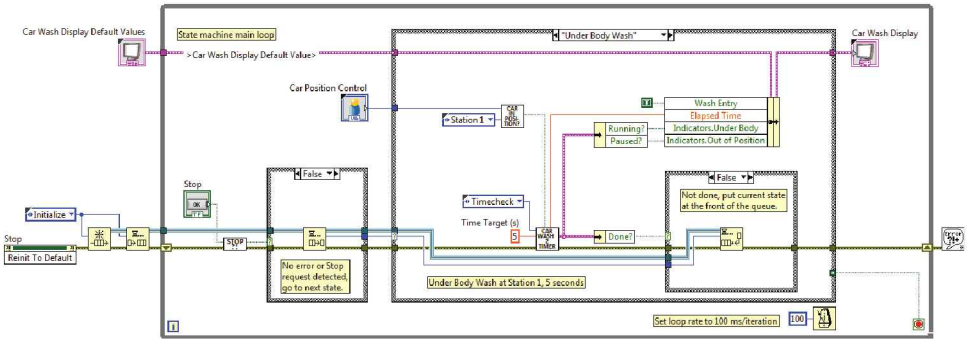


Figure K.23: The *Under Body Wash* state of the queued Car Wash 2 controller with the state not done

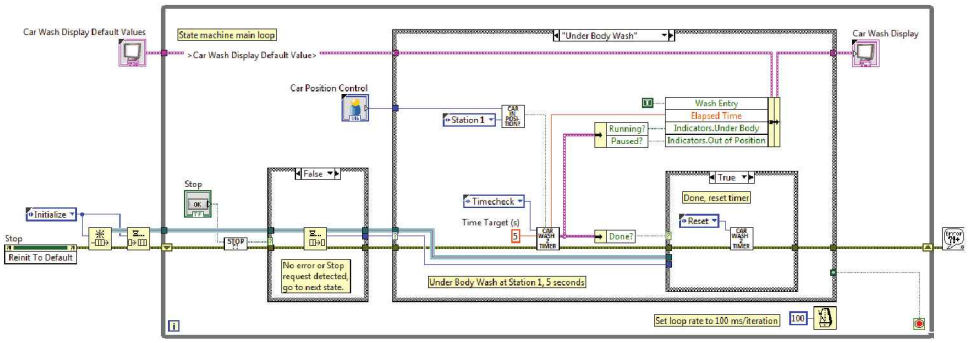


Figure K.24: The *Under Body Wash* state of the queued Car Wash 2 controller with the state done

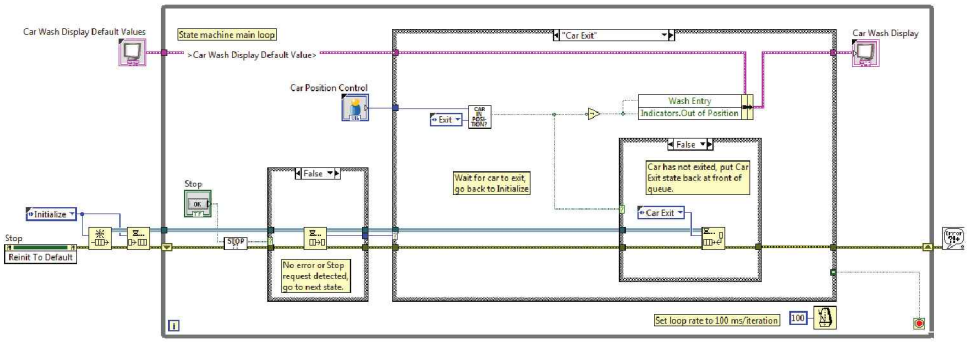


Figure K.25: The *Car Exit* state of the queued Car Wash 2 controller with the vehicle not yet exited

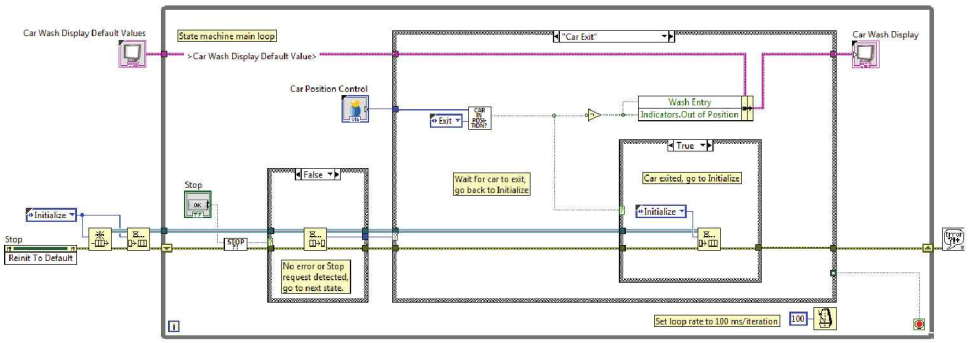


Figure K.26: The *Car Exit* state of the queued Car Wash 2 controller with the vehicle exited

Exercise 5.8

The block diagram of the `Clear Error Code` sub-VI is shown in Figure K.27. The status and code are unbundled from the error cluster using the **`Unbundle By Name`** function. If the status is True and the code is equal to the target error code, the **`Clear Errors`** function is used to clear the error. The error cluster is simply passed through if there is no error or if a non-matching error code is detected (see Figure K.28).

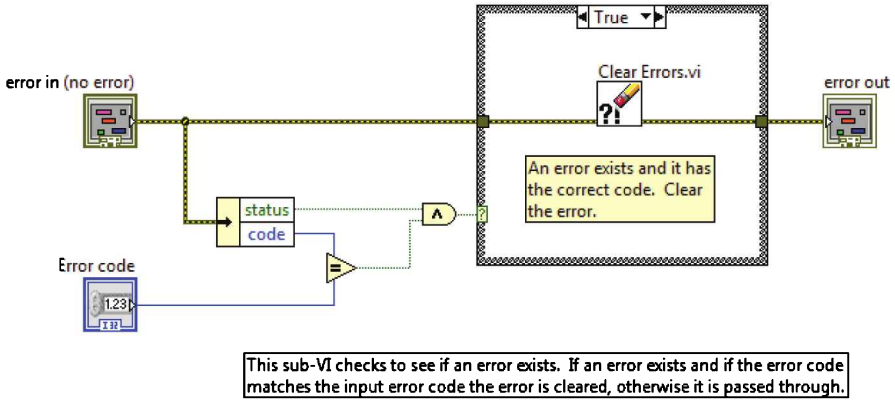


Figure K.27: The block diagram of the `Clear Error Code` VI with the target error detected

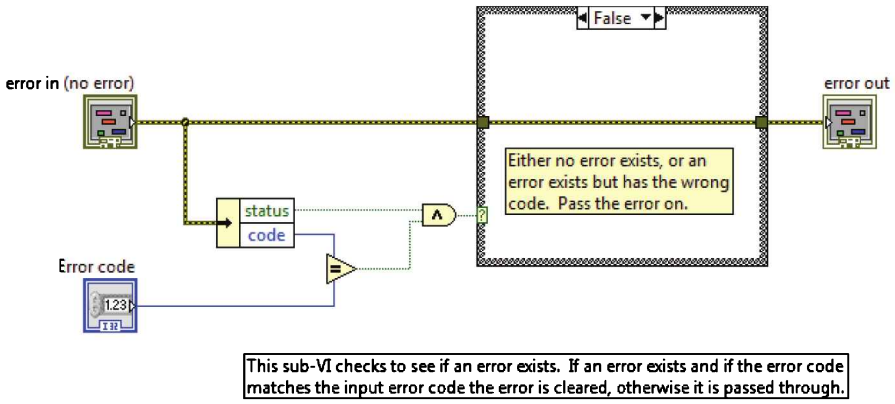


Figure K.28: The block diagram of the `Clear Error Code` VI with the target error not detected

Exercise 5.9

The front panel of the revised data acquisition VI showing the two new buttons for reading and writing the configuration information is shown in Figure K.29. The front panel includes two new buttons, a WRITE CONFIGURATION button, and a READ CONFIGURATION button. The event structure will have two new event cases to handle



Figure K.29: Front panel of the revised `Multitest` data acquisition VI

the value changes of these buttons, and the state machine will have two new states to perform the work: the *Write Configuration* and the *Read Configuration*.

Since the VI already has an event structure, it is easy to modify it to handle the value changes of these new buttons. The event case handling the value change of the WRITE CONFIGURATION button is shown in Figure K.30. This event case sends the state machine to the *Write Configuration* state and then returns to the *Wait* state. The event case handling the value change of the READ CONFIGURATION button is similar and is shown in Figure K.31. This event case sends the state machine to the *Read Configuration* state and then returns to the *Wait* state. Note that in both of these event cases the terminals for the controls have been placed in the event cases that handle their value changes to ensure that their mechanical actions are triggered when the event cases execute.

The *Write Configuration* state is shown in Figure K.32. The configuration cluster is read from the lobby of the state machine and is flattened to an XML string using the **`Flatten To XML`** function. The string is then written to a text file. Since no path was wired to the **`Write to Text File`** function, a dialog window will be brought up so the user can specify the desired file name and location. If the user cancels this window without saving the file, error number `43` is generated. Since the state machine is designed to capture all errors and to exit gracefully when an error is detected, this trivial error will shut down the state machine unless it is immediately cleared. Figure K.33 shows how the `Clear Error Code` sub-VI developed in Exercise 5.8 can be used to clear this error if the user cancels the dialog window.

The *Read Configuration* state is shown in Figure K.34. The flattened configuration text is read from the text file. Since no path is specified, the user will be presented with a dialog window to navigate to the configuration text file. The XML-formatted string is then converted back to a cluster using the **`Unflatten From XML`** function.

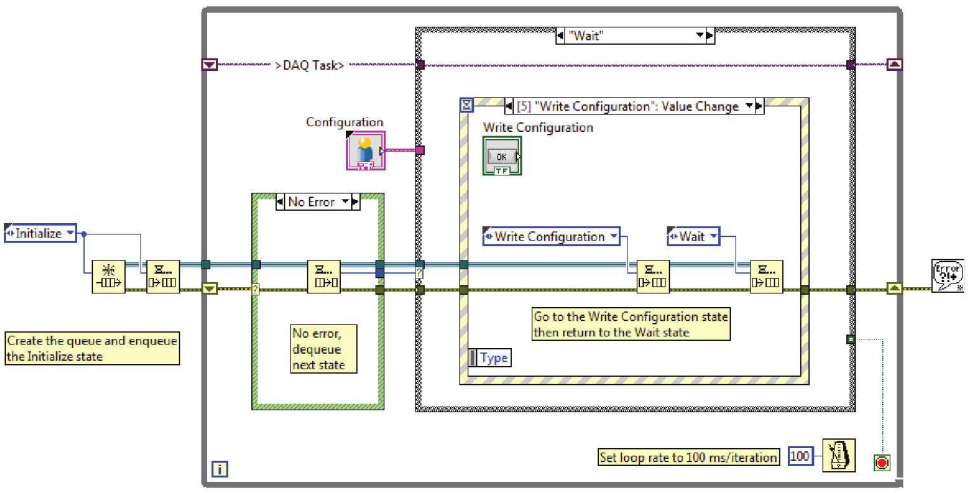


Figure K.30: Event case handling the value change of the WRITE CONFIGURATION button of the revised `Multitest` data acquisition VI

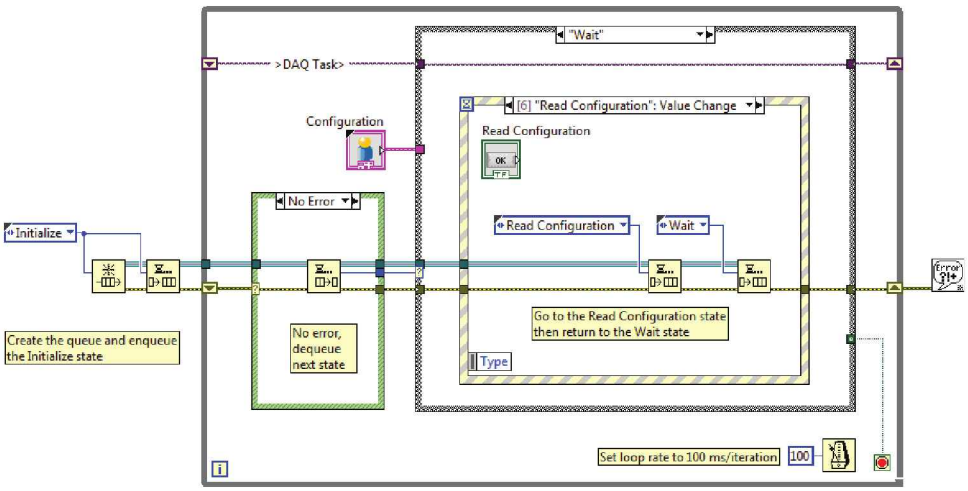


Figure K.31: Event case handling the value change of the READ CONFIGURATION button of the revised `Multitest` data acquisition VI

The cluster data is then written to the front panel control using a *Value(Signalling)* property node.

The *Value(Signalling)* property node serves two purposes. First, it reverses the standard data flow direction and lets us send data *to* a control rather than read data *from* it. Second, it triggers the event case handling the value change of the configuration cluster and causes the DAQ task to be rebuilt (see Figure 15.18). This is important because the DAQ task needs to be rebuilt now that the configuration data has changed.

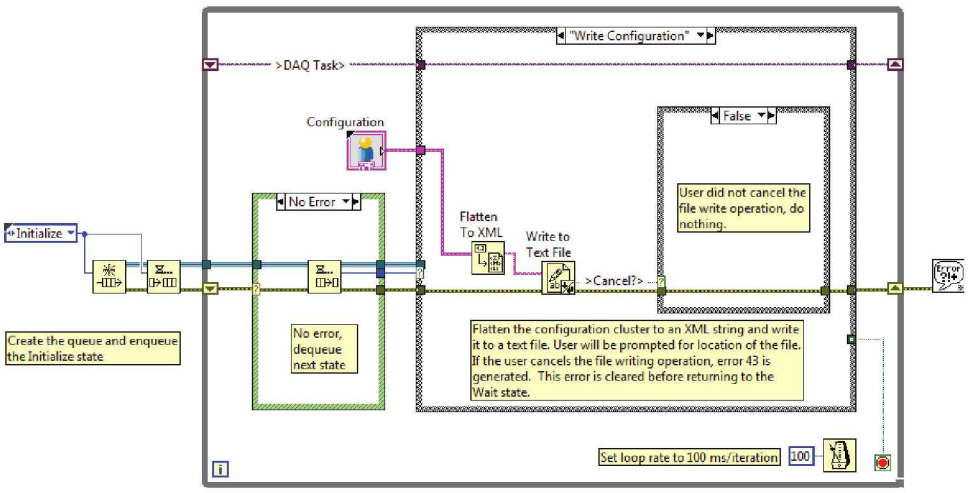


Figure K.32: The *Write Configuration* state of the revised `Multitest` data acquisition VI, user does not cancel dialog window

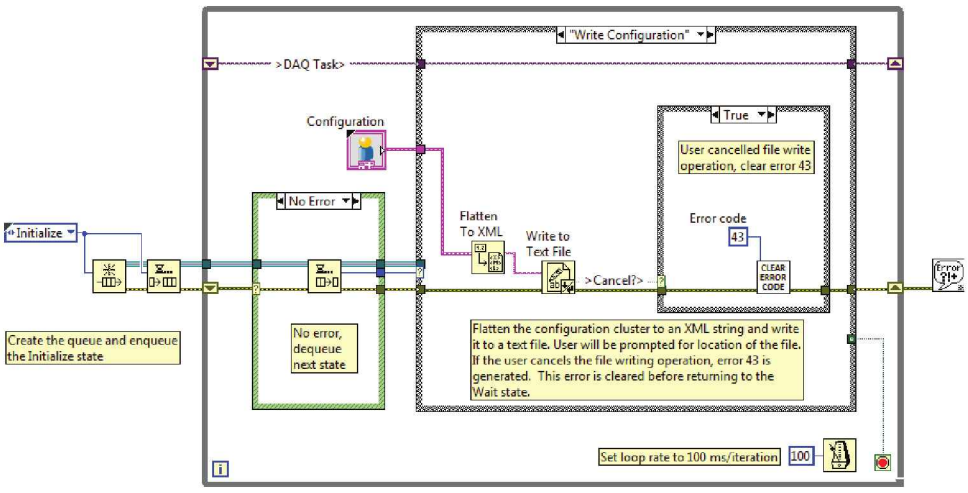


Figure K.33: The *Write Configuration* state of the revised `Multitest` data acquisition VI where the user cancels the dialog window

Pressing the READ CONFIGURATION button sends the state machine to the *Read Configuration* state, followed by the *Wait* state, then the *Build DAQ Task* state, and finally back to the *Wait* state.

If the user cancels the dialog window before the configuration data is read, error number `43` is generated. This error should be cleared to avoid shutting down the state machine. Figure K.35 shows how the `Clear Error Code` sub-VI developed in Exercise 5.8 can be used to clear this error if the user cancels the dialog window.

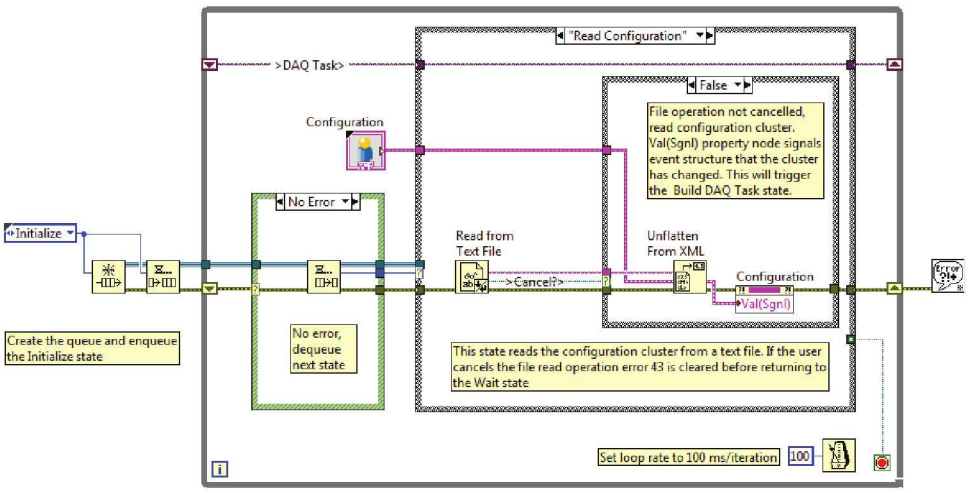


Figure K.34: The *Read Configuration* state of the revised `Multitest` data acquisition VI where the user does not cancel the dialog window

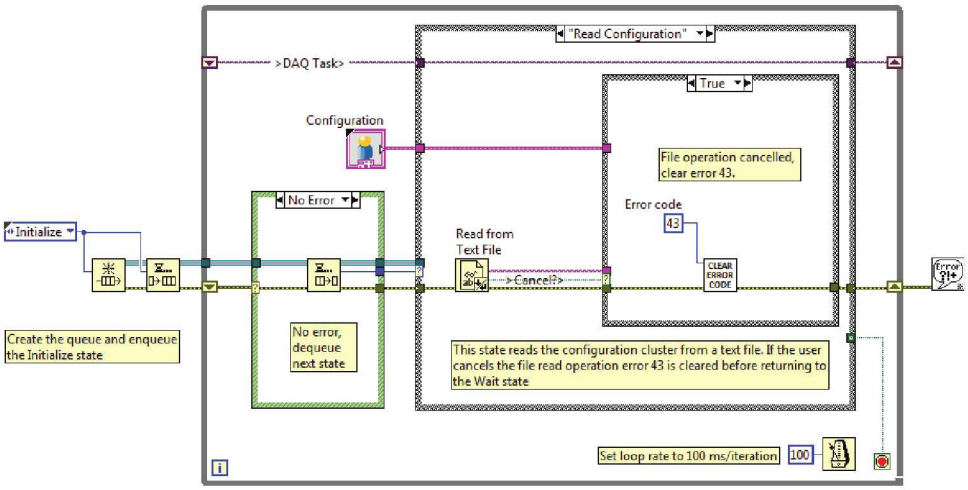


Figure K.35: The *Read Configuration* state of the revised `Multitest` data acquisition VI, user cancels dialog window

This concludes the revised version of the data acquisition state machine. Notice how easy it was to add functionality to the event-driven state machine pattern. We simply added front panel controls, configured event cases to handle the value changes of the front panel controls, and added states that implemented the desired functionality.

Exercise 5.10

The strain data will be logged by the `Save Strain Data` sub-VI with the icon shown in Figure K.36 and the block diagram shown in Figure K.37. This sub-VI accepts the time stamp that the data was taken and the configuration cluster used to take the data as inputs, as well as the data itself. The **`Format Into String`** function is used to create a comma-separated string that is then appended to the designated file using the `Append To Text File` sub-VI developed in Figure 1.45.

The key to this VI is the fact that the time stamp used by the `Save Strain Data` sub-VI must be the time stamp that reflects when the data was taken, not when it was saved. The same is true for the configuration cluster; it must be the cluster that was used to build the DAQ task used to generate the data, not the cluster available when the data was saved. Imagine the following sequence of events:

1. The user takes some data.
2. The user waits for an hour or so and then makes some changes to the configuration cluster for the next test.
3. The user remembers that the data from the previous test was not yet saved.
4. The user saves the data.

The user saves the data an hour after it was taken, and the configuration cluster has changed. The VI needs to anticipate this possibility. One solution to this problem is to save the time stamp and the configuration cluster on the data highway. Figure K.38

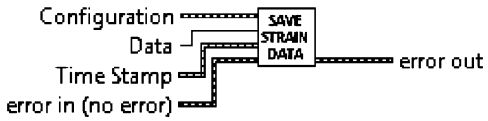


Figure K.36: The icon of the `Save Strain Data` sub-VI

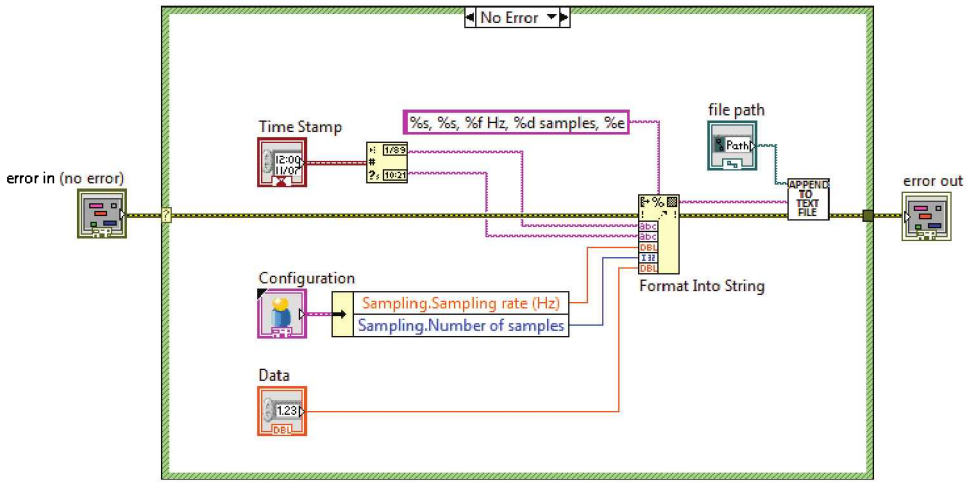


Figure K.37: The block diagram of the `Save Strain Data` sub-VI

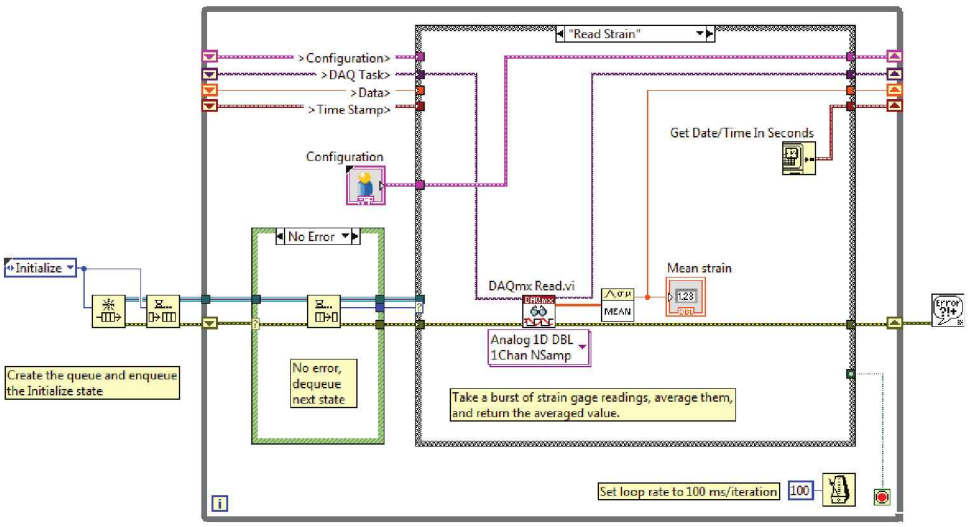


Figure K.38: The *Read Strain* state of the revised `Multitest` data acquisition VI

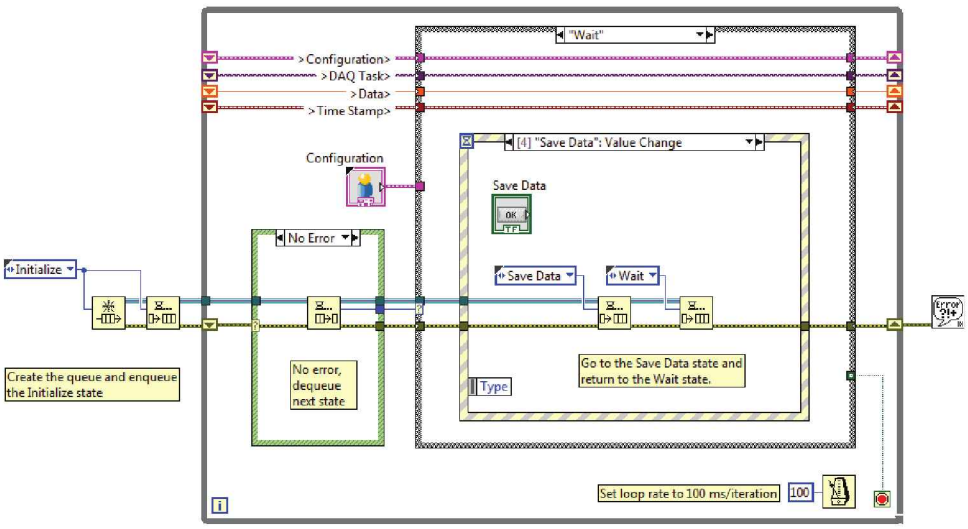


Figure K.39: The event case handling the value change of the SAVE DATA button in the revised `Multitest` data acquisition VI

shows the *Read Strain* state of the revised DAQ state machine. Whenever the strain is read, the configuration cluster is placed on the data highway along with the data and a time stamp that indicates when the data was taken. The configuration cluster and time stamp are stored with the data. When the user presses the SAVE DATA button, the event case catches the value change and sends the state machine to the *Save Data* state, as shown in Figure K.39. The *Save Data* state is shown in Figure K.40. The stored

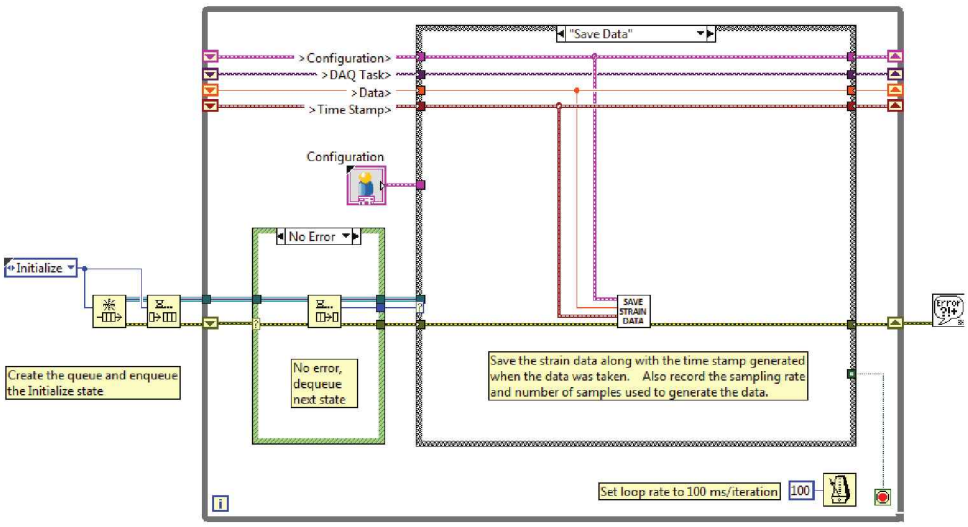


Figure K.40: The *Save Data* state of the revised `Multitest` data acquisition VI

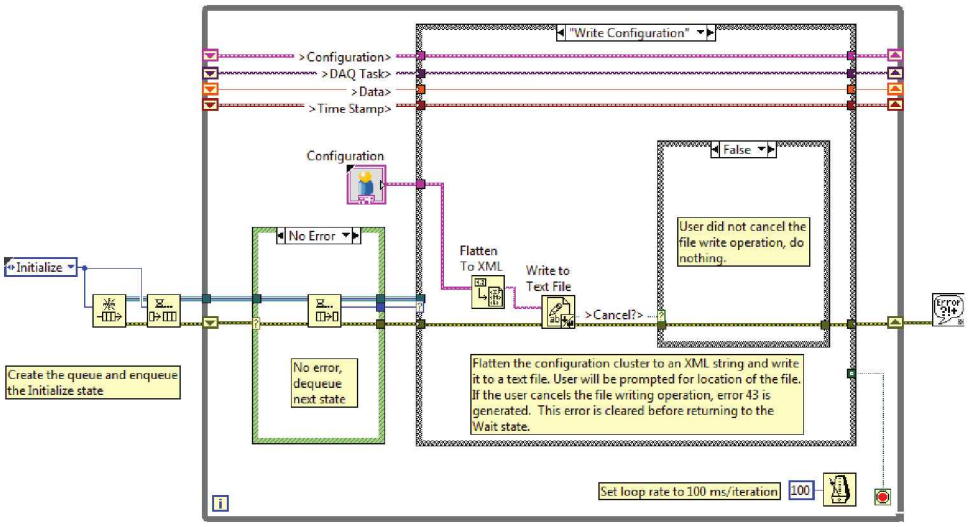


Figure K.41: The *Write Configuration* state of the revised `Multitest` data acquisition VI

version of the configuration cluster and the stored time stamp are passed to the `Save Strain Data` sub-VI to create the string written to the log file. This allows access to both the old and new versions of the configuration cluster simultaneously. The *Save Data* state can access the version of the configuration cluster that was stored on the data highway with the data, as shown in Figure K.40. The *Write Configuration* state can ignore the version on the data highway and access the most current contents of the configuration control in the lobby, as shown in Figure K.41.

L

Problem Set 6 Solutions

Exercise 6.1

A state transition diagram for the sprinkler controller is shown in Figure L.1. The controller starts in the *Initialize* state and resets the front panel controls. The ZONE SETUP ARRAY control is disabled and greyed out in this state. The controller then moves to the *Wait* state and waits for user input. If the user presses the SETUP button, the state machine transitions to the *Setup* state to enable the ZONE SETUP ARRAY control and then moves back to the *Wait* state.

When the user presses the START button, the state machine moves to the *Start* state to disable and gray out the ZONE SETUP ARRAY and to perform whatever tasks are needed to start a new sprinkler cycle. The program then starts servicing the zones by transitioning to the *Service Zone* state. The program continuously cycles through this state following the sprinkler program laid out in the ZONE SETUP ARRAY control.

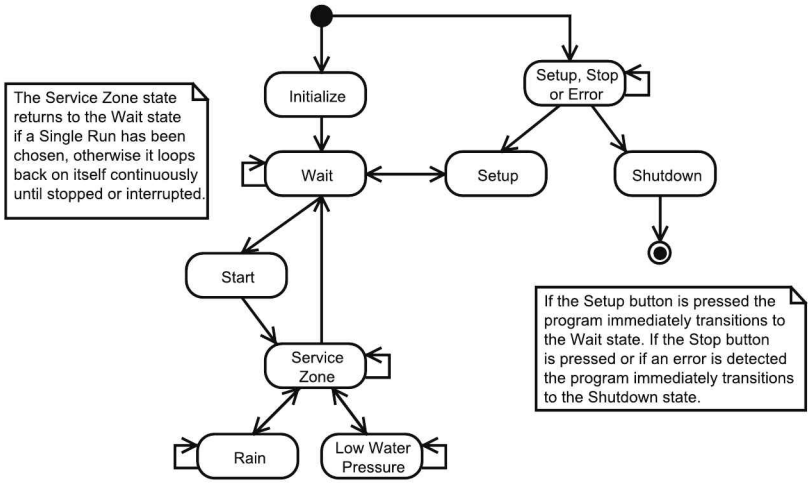


Figure L.1: A possible state transition diagram for the sprinkler controller

If the RAIN simulation switch is pressed, the state machine transitions to the *Rain* state and stays there until the RAIN switch is flipped back to its normal position. The state machine then transitions back to the *Service Zone* state. If the water pressure drops below 50%, the state machine transitions to the *Low Water Pressure* state. It stays in this state until the water pressure increases to more than 50%, when it then transitions back to the *Service Zone* state. If the state machine is in *Single* mode, it will return to the *Wait* state when all of the zones in the array have been serviced. If the state machine is in *Continuous* mode, it will stay in the *Service Zone* state until the run is interrupted or aborted. The *Continuous* mode can be interrupted by rain or low water pressure, or it can be aborted by detection of an error of by the user pressing the SETUP button or STOP button.

When the state machine is running, it is constantly checking to see if an error has occurred or if either the STOP button or the SETUP button has been pressed. If the STOP button is pressed or if an error is detected, the state machine immediately transitions to the *Shutdown* state and the program ends. If the user presses the SETUP button, the state machine immediately transitions to the *Setup* state, followed by the *Wait* state. The check for errors, for pressing the SETUP button, and for pressing the STOP button take precedence over the actions of the other states.

Exercise 6.2

The state transition diagram in Figure L.1 has a *Wait* state, so an event structure would be useful. Queues would not be useful here. The zone transitions are dictated by the ZONE SETUP ARRAY, so it is more natural to store an array index on the data highway and use it to step through the array rather than use a queue. This is particularly true since the sprinkler controller can run in a continuous mode and cycle through the array over and over again. This would be difficult with a queue because a dequeued state is removed from the queue and cannot be revisited. The event-driven classic state machine is thus a good choice for the sprinkler controller.

The controls and indicators can be placed in the state machine using the rules of thumb suggested in Chapter 4. The CONTROLS cluster contains both the STOP and the SETUP buttons. If either of these buttons are pressed, the state machine must react immediately. This means that they must be read every loop iteration. In addition, the CONTROLS cluster will be used as an input to the *Service Zone* state since this state needs to know whether the controller is in single or continuous mode when deciding what state to go to next. These facts suggest that the CONTROLS control should be placed in the lobby of the state machine. The SIMULATION CONTROLS cluster can change frequently and is needed as an input by the *Service Zone*, *Rain* and *Low Water Pressure* states. This suggests that it should also be placed in the lobby of the state machine and wired to the main case structure as an input.

The ZONE SETUP ARRAY control is only used by the *Service Zone* zone state to dictate the current zone and its time target. According to the functional specification, the user can change this array while the state machine is waiting. When the sprinkler cycle starts, the array is disabled, locking in its value and protecting it until it is read in the *Service Zone* state. This suggests that the ZONE SETUP ARRAY should be placed in the *Service Zone* state.

The ZONE INDICATORS cluster and the CONTROLLER STATUS need to be accessed by all states, and they need to be updated each loop iteration. This suggests that both of these indicators should be placed in the alley of the state machine. The ZONE INDICATORS cluster contains indicators for all zones, but only one zone will be active at a time. The other indicators will be at their default values. This suggests that a default value of the ZONE INDICATORS cluster be placed in the input area of the main loop. This cluster constant does not change and has all indicators dark. It can then be used as the central input of the **`Bundle By Name`** function in the individual states. The correct lamps can be lit for a given zone, and all of the other lamps will be dark automatically.

Data storage also needs to be considered. The sprinkler will need a timer. The timer state variables will be stored persistently in an action engine. The zone data is stored in an array, so the index of the currently serviced zone also needs to be stored persistently. In the code developed here, the index will be stored on the data highway.

Exercise 6.3

The front panel controls and indicators should be saved as type definitions, as described in the problem statement. Two type-defined enumerated controls should also be created. The enumerated control containing the sprinkler controller states will have the following elements: *Initialize*, *Wait*, *Setup*, *Start*, *Service Zone*, *Low Water Pressure*, *Rain*, and *Shutdown*. The enumerated control containing the guard clause states will have the following elements: *Stop or Error*, *Setup*, and *Continue*.

Exercise 6.4

The block diagram of the `Setup Stop or Error` sub-VI is shown in Figure L.2. A **`Select`** ladder is used to perform the necessary checks. If an error is present or if

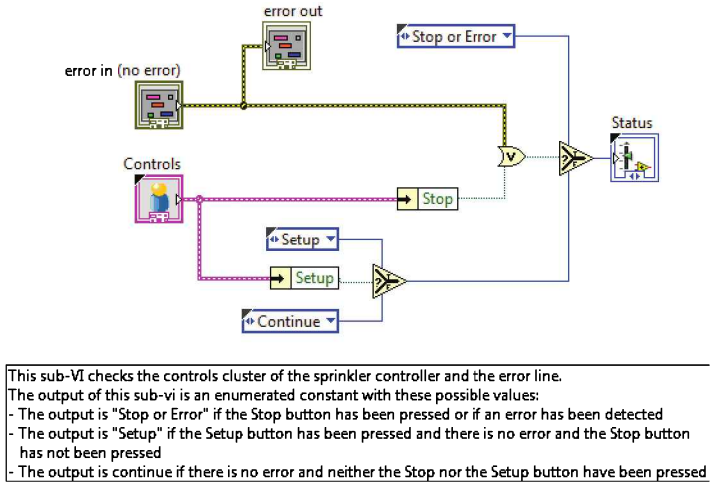


Figure L.2: Block diagram of the `Setup Stop or Error` VI

the STOP button is pressed, the *Stop or Error* state is returned. If the STOP button has not been pressed and no error is present, the SETUP button is checked. If it has been pressed, the *Setup* state is returned. If it has not been pressed, the *Continue* state is returned. Notice that this sub-VI will function normally even if an error is present at the error input terminal. This is important since this sub-VI is responsible for sending the state machine to the *Shutdown* state if an error is detected.

Exercise 6.5

One possible implementation of the `Initialize Zone Array` sub-VI is shown in Figure L.3. The **`Read From Spreadsheet File`** function is used to read the comma-separated input file. A comma is specified as the delimiter, and the polymorphic menu shows that the output will be a two-dimensional string array. This array is then wired to an auto-indexing input tunnel of a **`For`** loop. The auto-indexing tunnel passes each row of the string array into the loop—one row per loop iteration. The **`Index Array`** function is then used to retrieve the two elements of each row: a string containing the name of a zone and a string containing the timing information. The name string is cast to the type-defined zone combo box type using the **`Type Cast`** function. The timing string is converted to a double-precision number using the **`Fract/Exp String To Number`** function. These two items are then bundled into the ZONE SETUP ARRAY cluster element. The cluster elements are then collected into the output array by an auto-indexed output tunnel of the **`For`** loop.

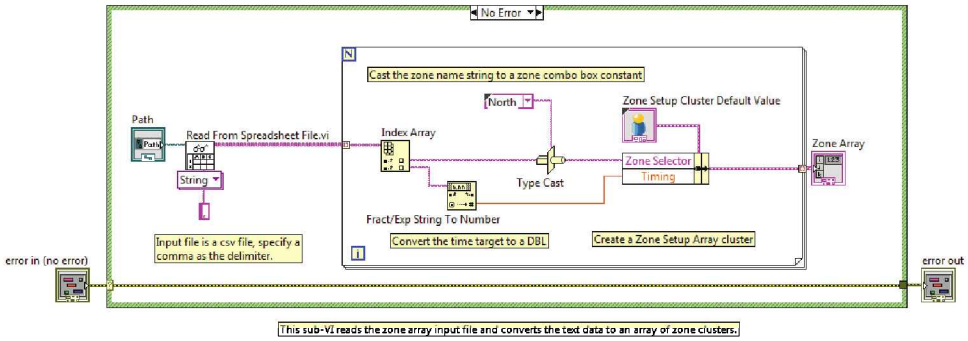


Figure L.3: The block diagram of the `Initialize Zone Array` VI

Exercise 6.6

One possible implementation of the `Next Sprinkler State` sub-VI is shown in Figure L.4. This sub-VI uses a **`Select`** ladder to determine the next sprinkler state using priorities:

1. If the water pressure is below 50% go to the *Low Water Pressure* state, else
2. If it is raining go to the *Rain* state, else

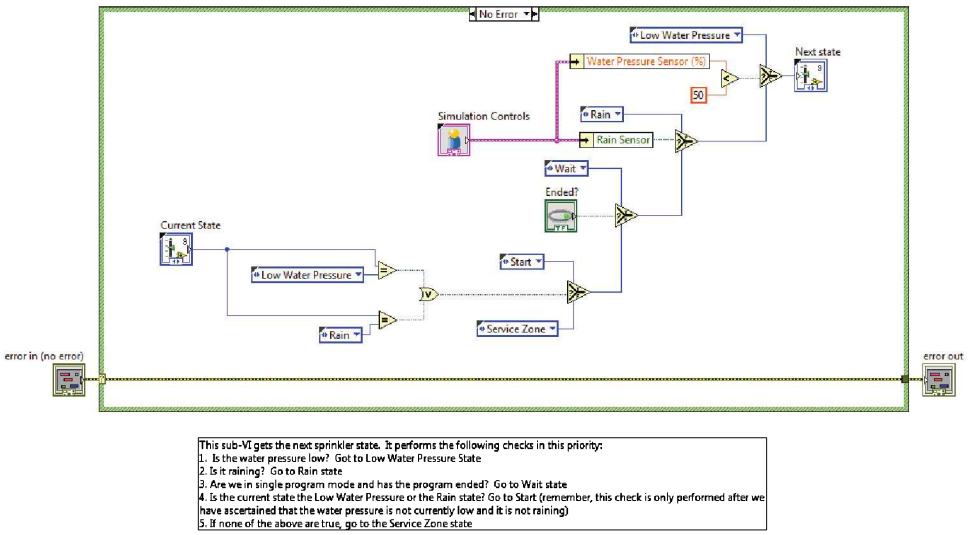


Figure L.4: The block diagram of the `Next Sprinkler State` VI

3. If the controller is running a single cycle sprinkler program and the program has ended go to the *Wait* state, else
4. If it has just stopped raining or the water pressure has just been restored go to the *Start* state, else
5. Go to the *Service Zone* state.

The fourth point checks to see if the rain has stopped or if the water pressure has been restored. How do we know if the rain has just stopped or if the water pressure has just been restored? If the current rain sensor value is False and yet the current state is *Rain*, the rain sensor must have been switched off between the last loop iteration and the current loop iteration. Likewise, if the current water pressure sensor is above 50% and the current state is *Low Water Pressure*, the water pressure must have been restored between the last loop iteration and the current loop iteration. The first two steps of the ladder check to see if it is raining or if the water pressure is low, so if we make it to the bottom of the ladder and the current state is *Rain* or *Low Water Pressure*, we know that those conditions have just ended, and the the state machine should go back to the *Start* state.

Exercise 6.7

An implementation of the `Auto Reset Timer With Elapsed Time` is shown in Figure L.5. This block diagram is identical to the block diagram of the auto-reset traffic timer shown in Figure H.12, except for the addition of an output for the elapsed time. Since the elapsed time is already stored in the state variables of the auto-reset timer, it is simply unbundled from the state variable cluster and wired to an indicator.

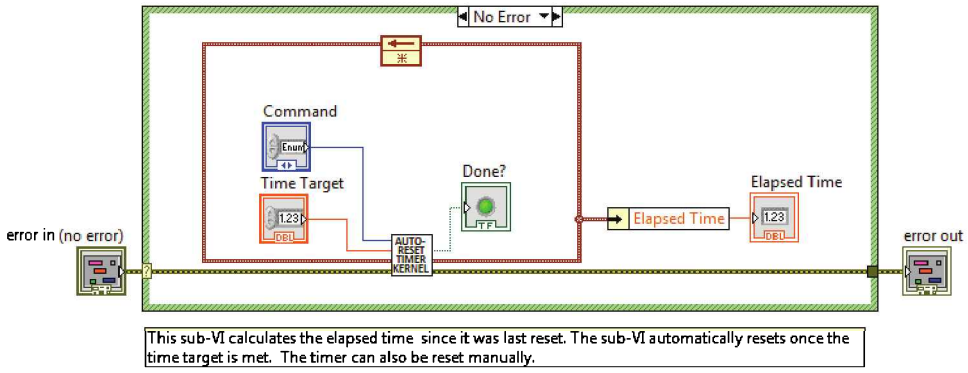


Figure L.5: The block diagram of the `Auto Reset Timer With Elapsed Time` action engine

Exercise 6.8

An implementation of the `Light Zone Lamp` sub-VI is shown in Figure L.6. This mapping sub-VI bundles the TIMING input into a ZONE SETUP ARRAY cluster element and then passes that element as an input to a case structure. The zone name is wired to the selector of the case structure. Each case of the structure bundles the value of

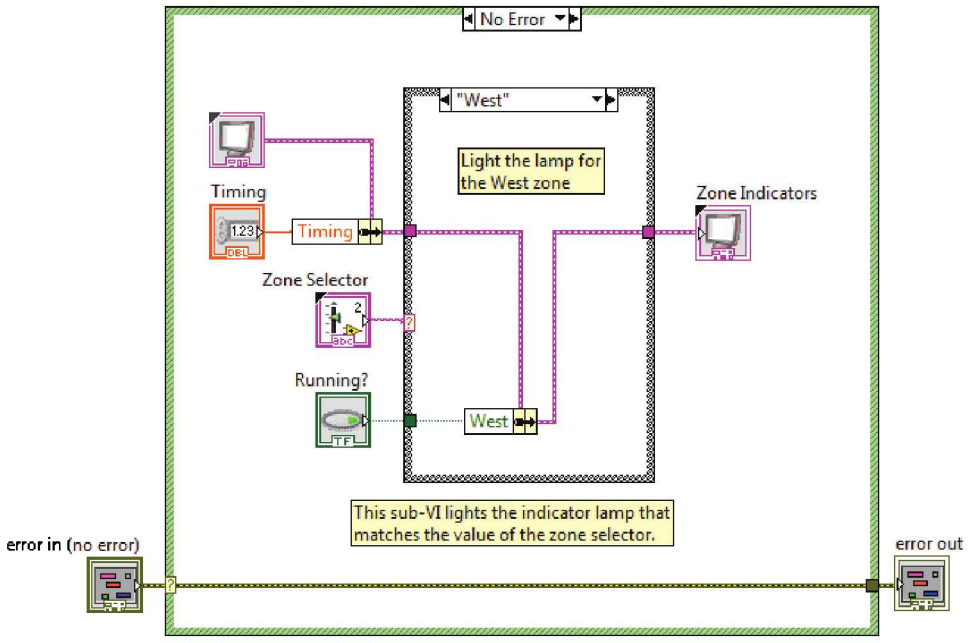


Figure L.6: Block diagram of the `Light Zone Lamp` VI showing the *West* case

the RUNNING? Boolean to the appropriate zone lamp in the cluster element. Figure L.6 shows the case that handles the *West* zone, and the other cases are similar.

Exercise 6.9

The last support sub-VI is the `Increment Index` sub-VI. The purpose of this index is to increment the index of the zone array if the current zone is done. This is a straightforward operation until the index reaches the size of the array minus one (the end of the zero-indexed array). Incrementing the index will then cause the index to "roll over" back to zero. If the sprinkler is running in single cycle mode, this is the end of the sprinkler operation, and the output *Single mode done?* Boolean is set to True. If the sprinkler mode is continuous, the counter rolls over, and the output Boolean is set to False. If the current zone is not finished, the current value of the index is passed through, and the output Boolean is set to False.

An implementation of the `Increment Index` sub-VI is shown in Figure L.7. If the current zone is finished, the index is incremented. The index is then compared to the size of the ZONE SETUP ARRAY. The index is divided by the array size and the remainder calculated using the native **`Quotient and Remainder`** function. This remainder is the value of the incremented index.

The incremented index is equal to zero when the array "rolls over." If the sprinkler is in continuous mode, this restarts the cycle at the first zone. But if the sprinkler is in single mode, this is a signal that the cycle is complete and that the controller should go back to the wait state. The *Single mode done?* Boolean is set to True if the controller is in single mode and if the new index is equal to zero. If the current zone is not done, then the current index is passed through unchanged and the *Single mode done?* Boolean is set to False, as seen in Figure L.8.

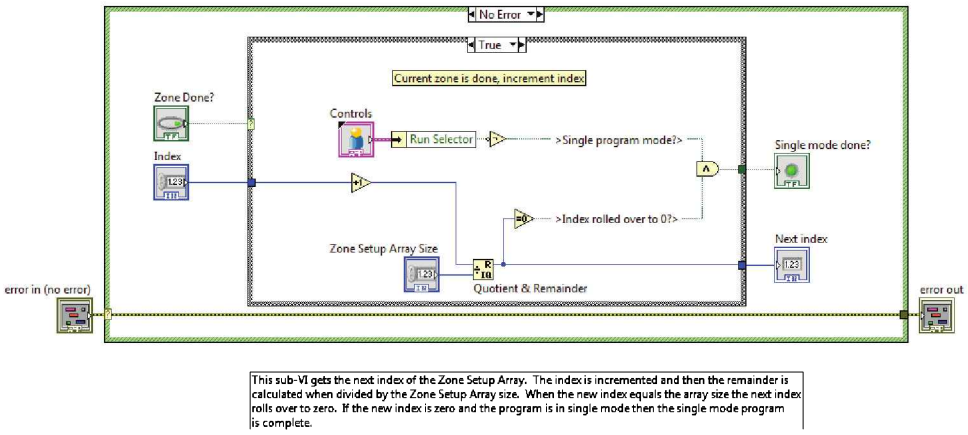


Figure L.7: Block diagram of the `Increment Index` VI showing the *Done* case

Exercise 6.10

An implementation of the `Service Zone` sub-VI is shown in Figure L.9. This sub-VI performs all of the tasks necessary to service a sprinkler zone. First, the current zone

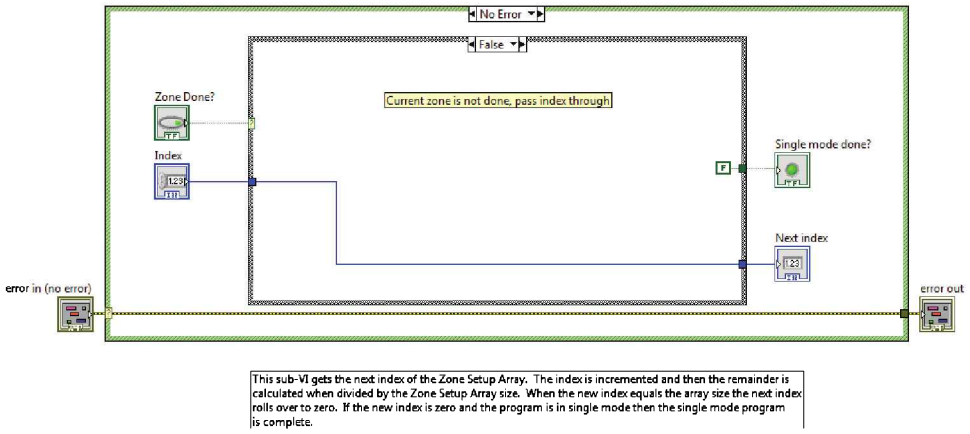


Figure L.8: Block diagram of the `Increment Index` VI showing the *Not Done* case

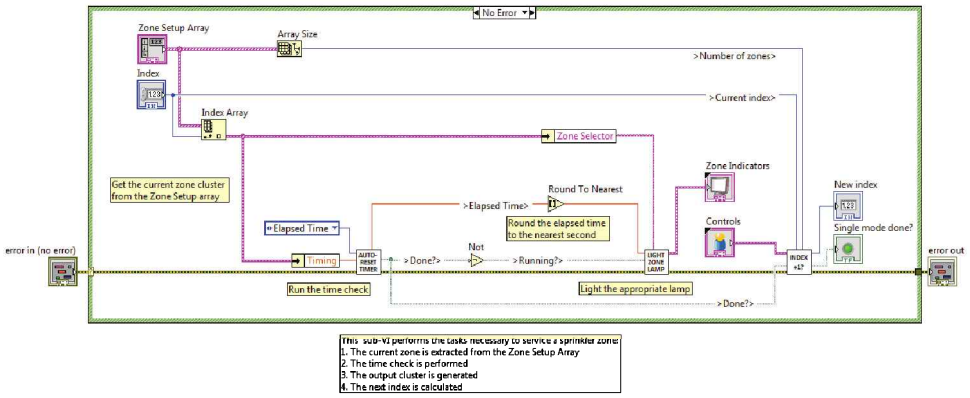


Figure L.9: The block diagram of the `Service Zone` VI

cluster is extracted from the ZONE SETUP ARRAY. The time target is unbundled from the zone cluster and is passed to the `Auto Reset Timer With Elapsed Time` developed in Exercise 6.7. A time check is performed, producing the elapsed time and a *Done?* Boolean output. The elapsed time is rounded to the nearest second using the **`Round To Nearest`** function and the *Done?* Boolean is negated, resulting in a *Running?* Boolean. These values are then passed as inputs along with the zone name to the `Light Zone Lamps` sub-VI developed in Exercise 6.8. This sub-VI generates a ZONE INDICATORS cluster which is displayed by the main application. Finally, the CONTROLS cluster, the number of zones, the current index, and the *Done?* Boolean are all passed to the `Increment Index` sub-VI developed in Exercise 6.9. This sub-VI produces the index for the next *Service Zone* iteration as well as a SINGLE MODE DONE? Boolean. This Boolean is True if the current state is the completion of the single cycle program.

Exercise 6.11

The overall structure of the state machine and the *Initialize* state are shown in Figure L.10. The CONTROLS cluster is polled in the lobby of the state machine. It is passed as an input to the `Setup Stop or Error` sub-VI. If no error is detected and if neither the STOP nor SETUP buttons have been pressed, the *Continue* case of the guard clause is executed as shown in Figure L.10. The current state is passed through to the selector of the main case structure.

The *Initialize* state enables the START button and disables the ZONE SETUP ARRAY control. This state also reads the contents of the `Sprinkler Configuration.txt` file, which is found in the same directory as the sprinkler controller. The contents of this file are converted into the ZONE SETUP ARRAY format by the `Initialize Zone Array` sub-VI developed in Exercise 6.5. The output of this sub-VI is written to the ZONE SETUP ARRAY using a local variable. The message `Controller Initialized` is sent to the CONTROLLER STATUS indicator.

The *Initialize* state then transitions to the *Wait* state. The event case handling the value change of the SETUP button is shown in Figure L.11. This event case displays the `Setup Mode` message and sends the state machine to the *Setup* state shown in Figure L.12. Note that the *Wait* state always enables the START button no matter which event case executes, and it is enabled before the event case executes. This is because the START button may have previously been disabled by other states. The event structure needs to respond to a value change of the START button, so the button must be enabled before the event structure executes.

The *Setup* state shown in Figure L.12 enables the zone array for editing, displays the `Setup Mode` message, and returns to the *Wait* state. Figure L.12 also shows the *Setup* case of the guard clause. If the user presses the SETUP button while a sprinkler cycle is executing, the state machine automatically transitions to the *Setup* state.

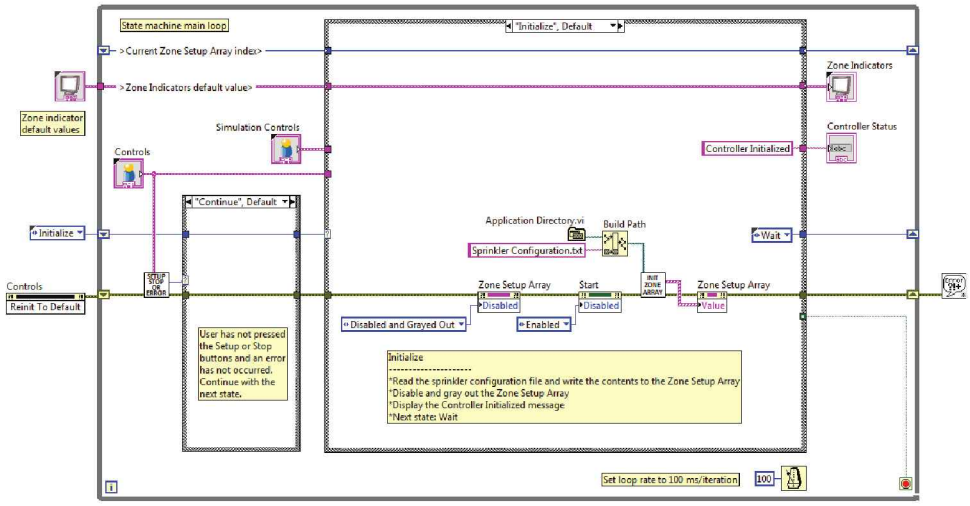


Figure L.10: The *Initialize* state of the sprinkler controller

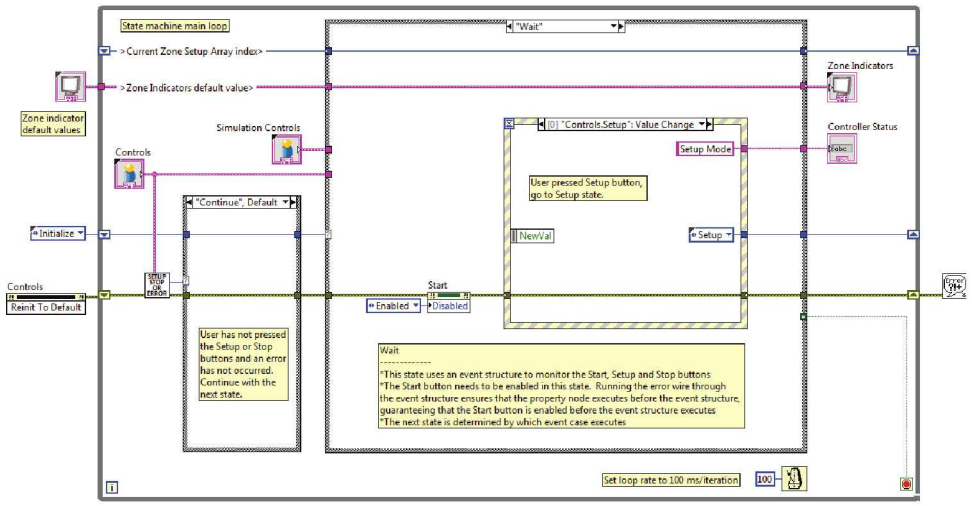


Figure L.11: The event case handling the value change of the SETUP button

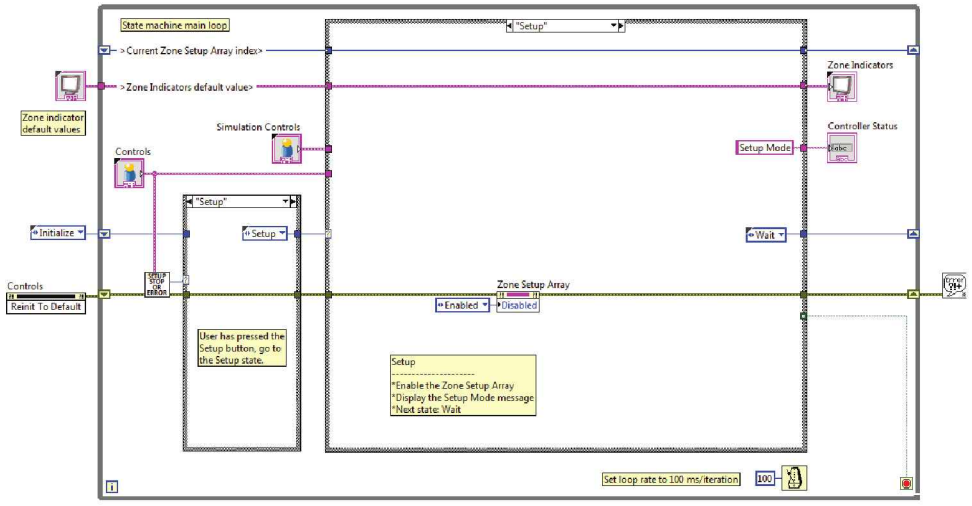


Figure L.12: The *Setup* state of the sprinkler controller

Returning to the *Wait* state, the event case handling the value change of the STOP button is shown in Figure L.13. This event case displays the `Stopped` message and sends the state machine to the *Shutdown* state shown in Figure L.14. The *Shutdown* state also displays the `Stopped` message and stops the main loop by wiring a Boolean True to the loop conditional terminal. This state places the *Shutdown* state on the state shift register because something must be wired to the case structure

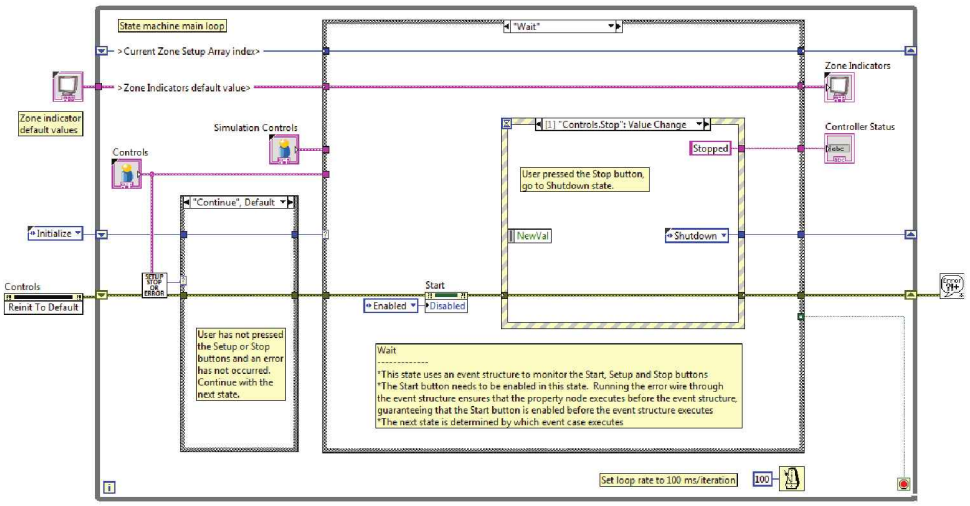


Figure L.13: The event case handling the value change of the STOP button

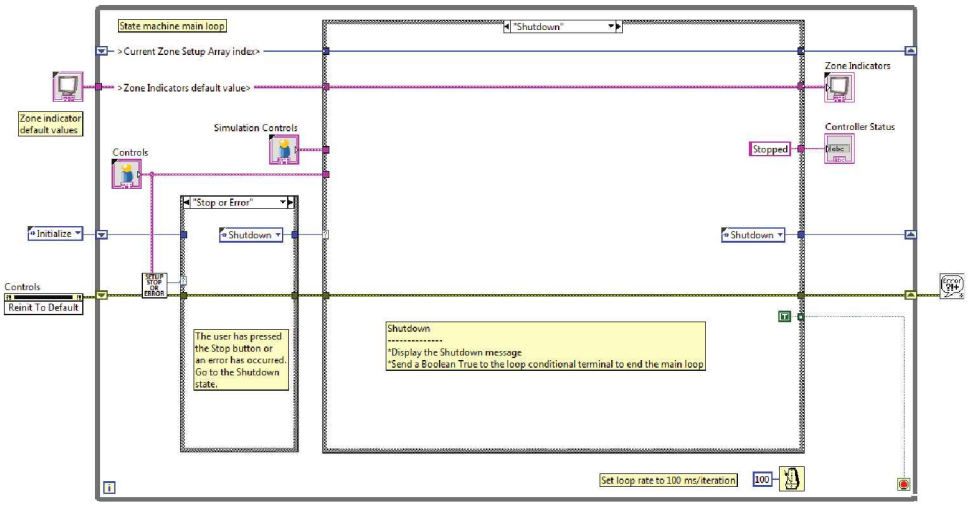


Figure L.14: The *Shutdown* state of the sprinkler controller

output tunnel, but this state will not execute, because the loop has already been shut down.

Figure L.14 also shows the *Stop or Error* case of the guard clause. If an error is detected or if the STOP button is pressed while a sprinkler cycle is executing, the state machine automatically transitions to the *Shutdown* state.

The final event case in the *Wait* state handles the value change of the *Start* button, as seen in Figure L.15. This event case displays the `Running` message and sends the

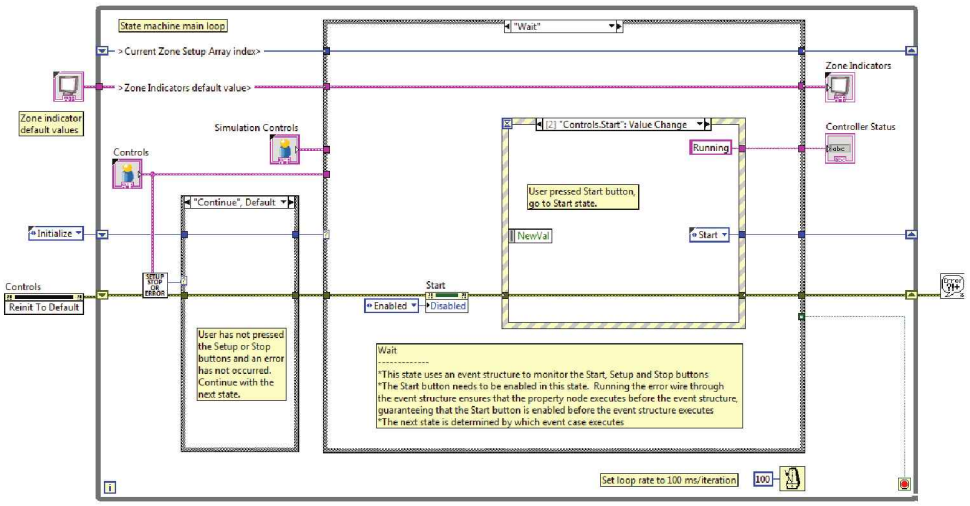


Figure L.15: The event case handling the value change of the START button

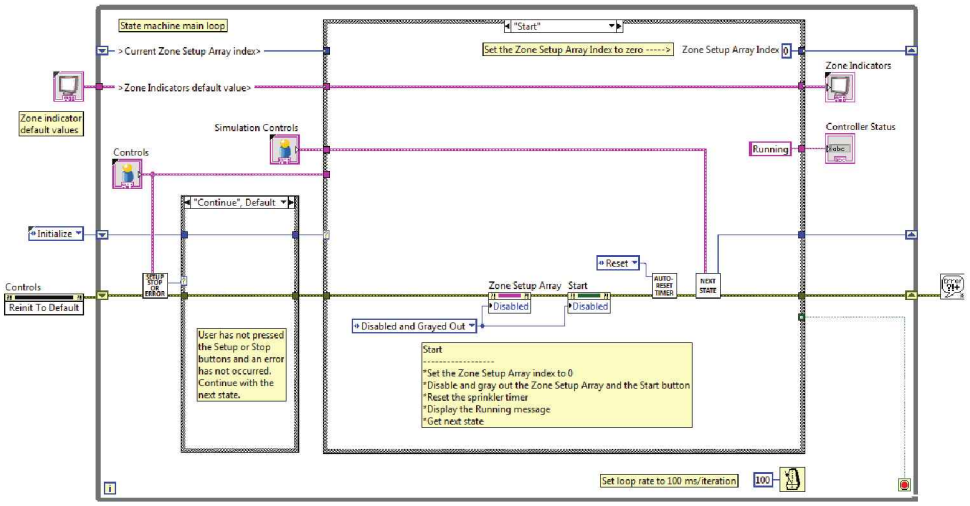


Figure L.16: The *Start* state of the sprinkler controller

state machine to the *Start* state shown in Figure L.16. Notice the way that the event structure is used in this application. The event structure serves as an event-driven switching station. The individual event cases contained no code—no sub-VIs. The individual event cases react to user input by sending the state machine to the state designed to handle the user request. This is good programming practice. The event structure is nimble and responsive. Code readability is good since each event case is

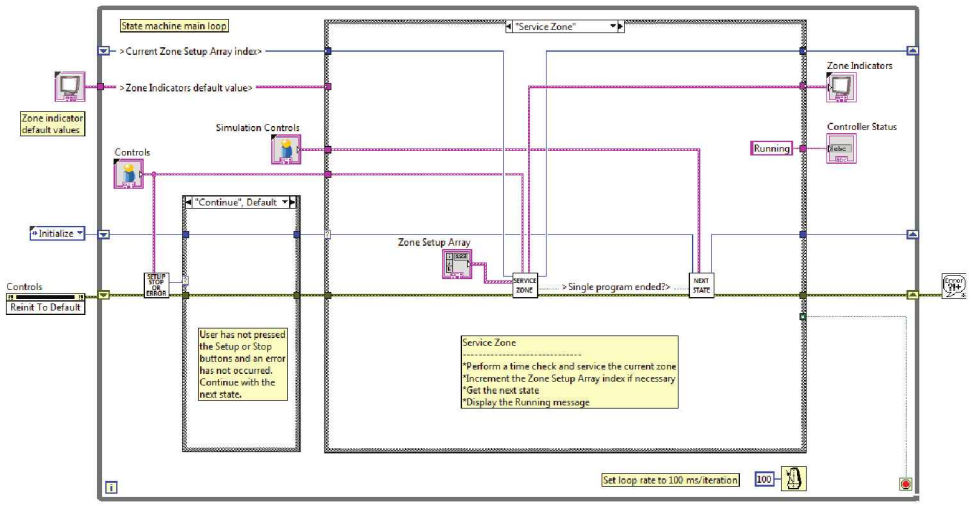


Figure L.17: The *Service Zone* state of the sprinkler controller

simple. The event structure acts like a map, telling the reader which state responds to each kind of user input.

The *Start* state does a number of things to prepare for a sprinkler cycle run. The index of the zone array is set to zero and is placed on the data highway. The ZONE SETUP ARRAY control and the START button are disabled and grayed out. The `Running` message is displayed, and the next state is calculated using the `Next Sprinkler State` sub-VI.

The *Start* state also manually resets the timer. The timer is designed to be auto-resetting, whenever the time target is met or exceeded, the elapsed time is automatically set to zero. But at the start of a new sprinkler cycle, the timer needs to be reset manually to ensure that the first sprinkler zone starts with an elapsed time of zero.

The sprinkler cycle is run by the *Service Zone* state seen in Figure L.17. This state also displays the `Running` message. The `Service Zone` sub-VI developed in Exercise 6.10 perfoms three tasks to service the current zone. First, the `Service Zone` sub-VI performs a time check. Then it creates the ZONE INDICATORS indicator cluster. Finally, it increments the index if needed. The next state is then calculated by the `Next Sprinkler State` sub-VI developed in Exercise 6.6 and is then placed on the state shift register.

The *Service Zone* state can be interrupted by low water pressure or by rain. The *Low Water Pressure* state is shown in Figure L.18. The `Low Water Pressure` message is displayed and the next state is calculated. The `Next Sprinkler State` sub-VI will keep returning the state machine to this state as long as the water pressure is less than 50%. When the water pressure is restored, the state machine will transition back to the *Start* state to restart the sprinkler zone sequence.

The *Rain* state is very similar to the *Low Water Pressure* and is shown in Figure L.19. The `Rain` message is displayed and the next state is calculated. The

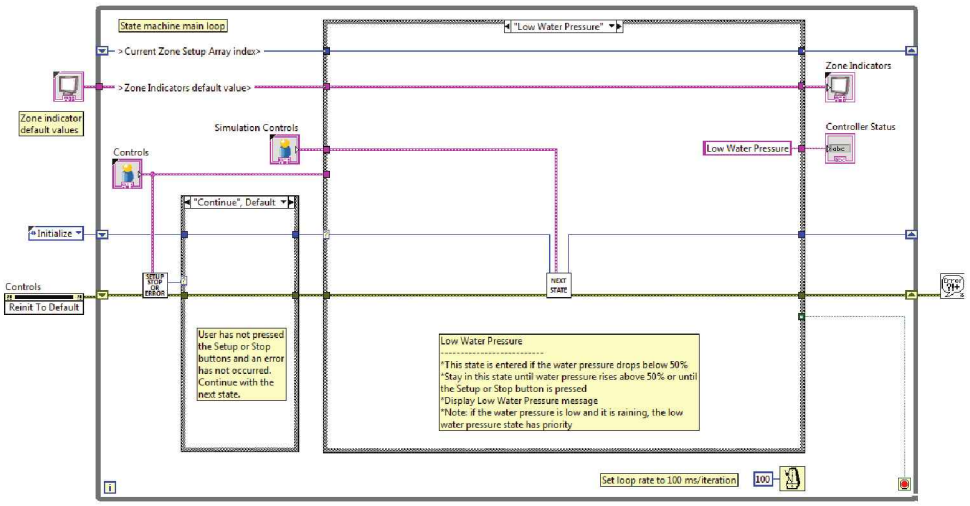


Figure L.18: The *Low Water Pressure* state of the sprinkler controller

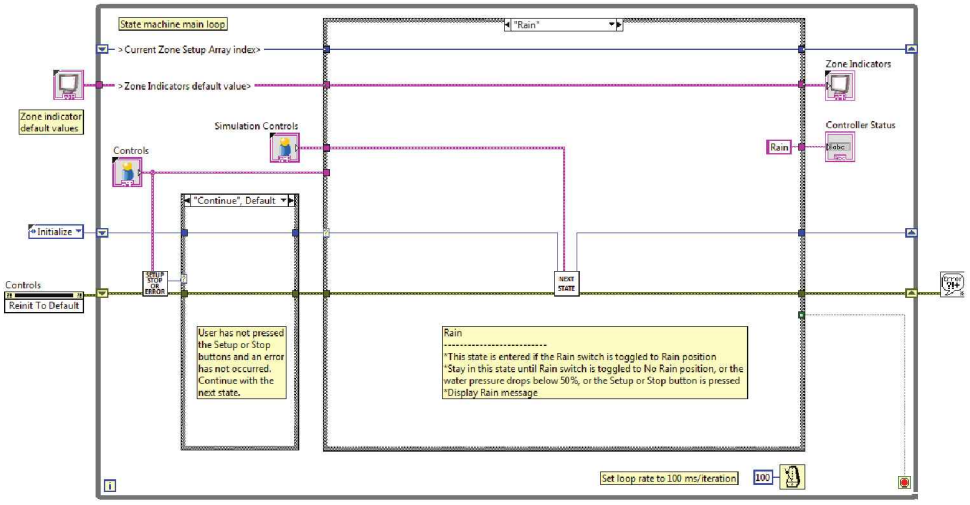


Figure L.19: The *Rain* state of the sprinkler controller

`Next Sprinkler State` sub-VI will keep returning the state machine to this state as long as it is raining. When it stops raining, the state machine will transition back to the *Start* state to restart the sprinkler zone sequence. This concludes the development of the event-driven sprinkler controller.



Problem Set 7 Solutions

Exercise 7.1

One possible state transition diagram for the ATM controller is shown in Figure M.1. When the controller starts, it initializes the front panel in the *Initialize* state and then passes to the *Wait* state. This state waits for the user to insert a bank card. The functional specification requires that after a card has been inserted that a ten second timeout be enforced, but the specification is unclear about timeouts before a card is inserted. In this implementation, we will assume that the ten second timeout is in force for the entire run of the program.

When a card has been inserted, the controller moves to the *Card Entered* state. This disables the card simulator and enables the user input field. When the user hits the enter button, the controller moves to the *Verify Account* state. If the account is verified, the controller moves to the *Activate Menus* state. This state enables the side menu buttons and menus and displays the menu messages. The controller then moves to the *Wait* state and waits for user input. If the account is not verified, the controller returns to the pre-activation *Wait* state.

Notice the *Wait* state appears twice on the state transition diagram. This will be the same state in the state machine, but it is convenient to represent it twice on the state diagram because the options for leaving the *Wait* state change after an account has been verified. It is easier to represent this graphically by duplicating the *Wait* state on the state transition diagram.

The LEFT MENU control has three buttons, but only two are currently active. If the user presses the BALANCE INQUIRY button, the controller moves to the *Balance Inquiry* state, displays the current balance, and returns to the *Wait* state. If the user presses the RETURN CARD & TERMINATE button, the controller transitions to the *Session Terminate* state and then to the *Shutdown* state.

The right menu has three active buttons. If the user presses the DEPOSIT button, the state machine transitions to the *Initiate Deposit* state. The controller then waits for the user to enter the deposit amount and then transitions to the *Complete Deposit* state and then back to the *Wait* state. Likewise, if the user presses the WITHDRAW button, the state machine transitions to the *Initiate Withdrawal* state. The controller then waits for the user to enter the withdrawal amount and then transitions to the *Complete*

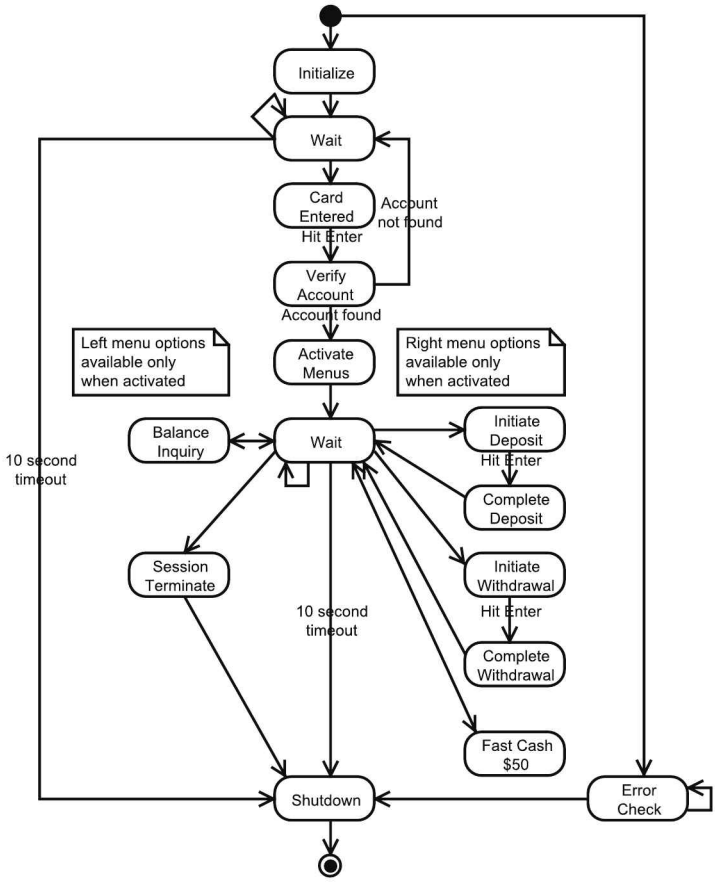


Figure M.1: A possible state transition diagram for the ATM controller

Withdrawal state and then back to the *Wait* state. The last right menu button is the FAST CASH $50 button. If this is pressed, the controller transitions to the *Fast Cash $50* state and then back to the *Wait* state.

The state diagram shows a parallel path for error checking. At the start of every loop iteration a check will be made for errors. If an error is detected, the controller will automatically transition to the *Shutdown* state. The state machine also moves to the *Shutdown* state if ten seconds pass with no user activity.

Exercise 7.2

The ATM controller has a *Wait* state that must time out after ten seconds of inactivity. This suggests that an event structure be used with an event case configured to handle a timeout state. The ATM controller does not need to queue up long, predefined series of states, so a queue would not add much value. For these reasons, an event-driven classic state machine would make a good choice for the ATM controller.

The central ATM MESSAGE indicator must be accessed by multiple states and so it should be placed in the alley of the state machine. The left and right menu indicators require a bit more thought. Technically, these menus only need to be written to once when the controller activates the menus in the *Activate Menus* state. The menus do not change after they have been activated and so the menu indicators could justifiably be placed in the *Activate Menus* state. However, the functional specification does state that "The application should be scalable to add other menus without major modificaitons to the application." This statement is vague, but the general intent seems clear. The side menus are intended to be "soft", they can be changed by the controller as circumstances dictate. It is not unrealistic to suppose that in the future the side menus might change in response to user input. In that case, it would be better to put the side menu terminals in the alley of the state machine where they can be accessed by any state in the state machine. This is the approach that will be taken here. Note that this means that the values of the left and right menu displays will have to be stored on the data highway so that the indicators will be refreshed every loop iteration.

The left and right ATM buttons will be handled in the event structure in the *Wait* state. Each menu cluster contains three buttons. Placing the cluster in a particular event case would mean that the cluster would only be read when that particular case executed. Since the value of the button doesn't matter as much as the fact that the value changed, the clusters will not be wired to anything in the program. It is convenient to place them somewhere in the lobby of the state machine where they will be read each loop iteration. This will trigger the mechanical action of the buttons in the cluster and will keep the terminals out of the way of the main program.

The CARD SIMULATOR is similar in the fact that the actual value of the control doesn't matter, only the fact that the value has changed. An event case in the *Wait* case will be configured to handle the value change of the CARD SIMULATOR button, and the control should be placed in that event case.

The USER INPUT field is also similar to the CARD SIMULATOR control. The control only needs to be read when the ENTER button is pressed. It should be placed in the event case handling the value change of the ENTER button. This event case is also the natural place to put the ENTER button itself. The control will be unwired, but putting it in the event case handling its value change ensures that the mechanical action of the button will be triggered.

The ATM controller is unusual in the fact that none of its controls need to be wired to the main case structure as an input. This also means that the guard clause can be configured to simply check for errors. There are no buttons in the lobby of the state machine that need to be monitored by the guard clause case structure.

The ATM controller has a number of data items that need to be stored persistently on the data highway. As mentioned previously, the left and right menu values need to be stored persistently if the menu terminals are included in the alley of the state machine. Likewise, the value of the central ATM MESSAGE indicator needs to be stored, since some operations (such as deposit and withdraw) take more than one state to accomplish and the ATM message must be displayed by all states within those operations. Likewise, the account information needs to be stored, since it needs to be accessed by multiple states. The current mode of the ENTER button also needs to be stored. Finally, user input (such as the amount of a deposit or

withdrawal) needs to be stored, because these operations take more than one state to accomplish.

Exercise 7.3

The states of the ATM controller are *Initialize*, *Card Entered*, *Wait*, *Verify Account*, *Activate Menus*, *Balance Inquiry*, *Session Terminate*, *Initiate Deposit*, *Complete Deposit*, *Initiate Withdrawal*, *Complete Withdrawal*, *Fast Cash* $50, and *Shutdown*.

Exercise 7.4

An implementation of the `Get Message Templates` sub-VI is shown in Figure M.2. The enumerated control listing the various message types is wired to the selector of a case structure. Each case of the structure contains a string constant with the template for the given message type. Figure M.2 shows the template for the *Main Menu* message. This message contains two keys: `[First Name]` and `[Last Name]`.

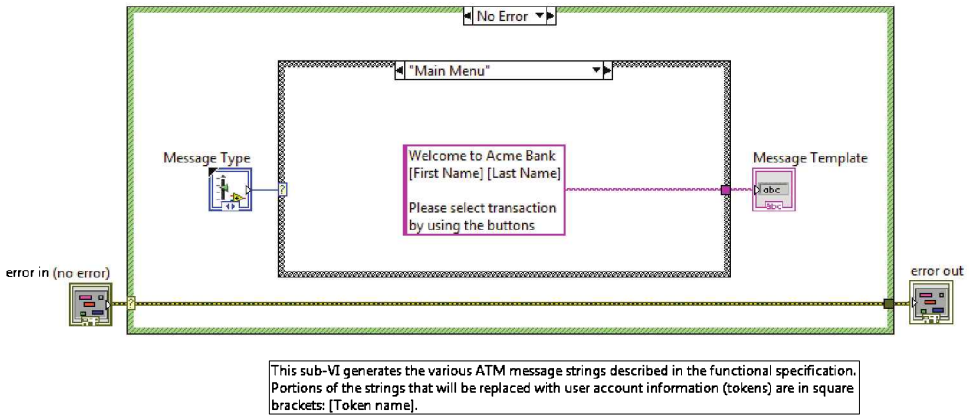


Figure M.2: Block diagram of the `Get Message Templates` VI

Exercise 7.5

An implementation of the `Replace Keys With Values` sub-VI is shown in Figure M.3. This sub-VI accepts a message template as an array of keys and an array of values as inputs. The message template initializes a shift register on a **`For`** loop. The key and value arrays are each passed to auto-indexing input tunnels of the **`For`** loop. The loop runs for a number of iterations equal to the number of keys (or values). On a given loop iteration, the **`Search and Replace String`** is used to search for all instances of a given key and to replace them with the corresponding value. The modified template is then placed on the shift register and is passed on to the next loop iteration. This is done to preserve all of the changes made to the template. When the key and value arrays are exhausted, the loop ends and the modified template is passed as an output.

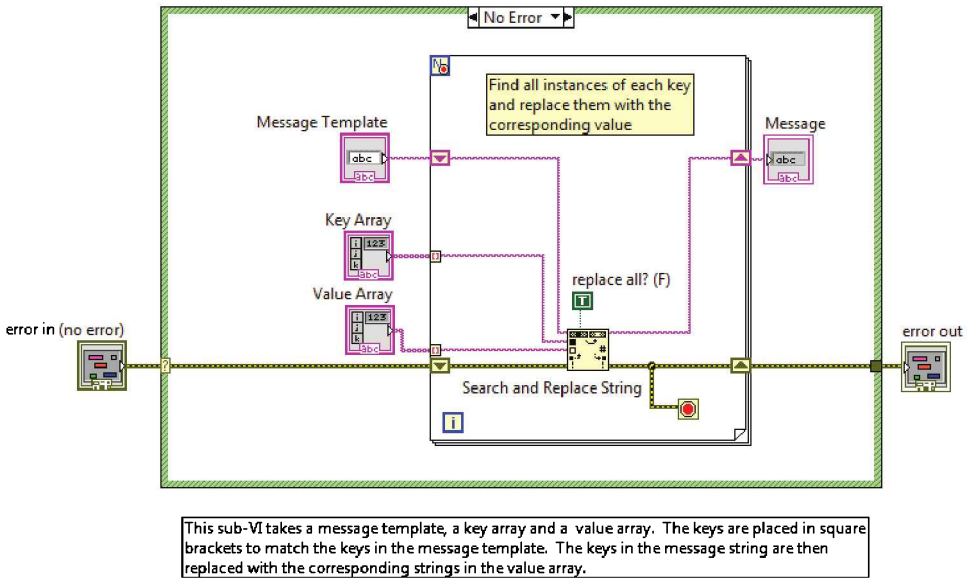


Figure M.3: Block diagram of the `Replace Keys With Values` VI

Exercise 7.6

An implementation of the `Get Message` sub-VI is shown in Figure M.4. The desired message template is retrieved using the `Get Message Templates` developed in Exercise 7.4. The value array is constructed by unbundling the relevant items from the account cluster and creating an array using the **`Build Array`** function. The key array constant and the value array are then passed to the `Replace Keys With Values` sub-VI developed in Exercise 7.5. The resulting personalized message is then passed as an output.

Exercise 7.7

An implementation of the `Get Account` sub-VI is shown in Figure M.5. The **`Read From Spreadsheet`** function is used with a comma delimiter to read the comma-separated value accounts file at the given path. The resulting two-dimensional array of strings is wired to an auto-indexing input tunnel of a **`For`** loop. This tunnel passes each row into the loop—one row per loop iteration. The **`Index Array`** function is then used to extract the individual element of the row. The account number of each row is compared to the input account number until a match is found, stopping the conditional **`For`** loop. The elements of the matching account are then output from the loop and are bundled into an account cluster with a zero balance. The index of the matching account is also passed as an output, along with a Boolean indicating whether or not a matching record was found. A two-dimensional array containing the entire contents of the accounts file is also passed as an output.

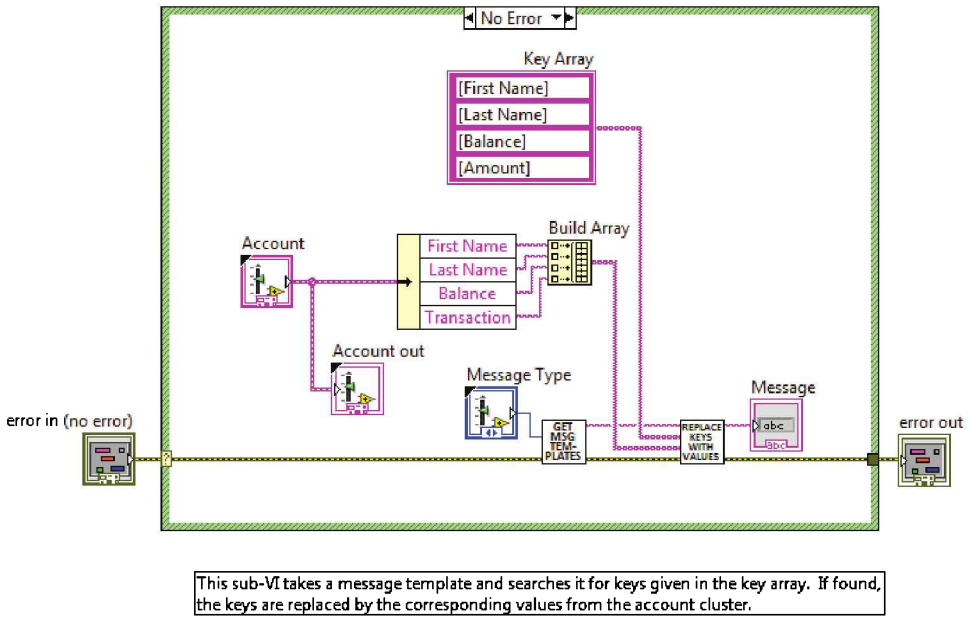


Figure M.4: Block diagram of the `Get Message` VI

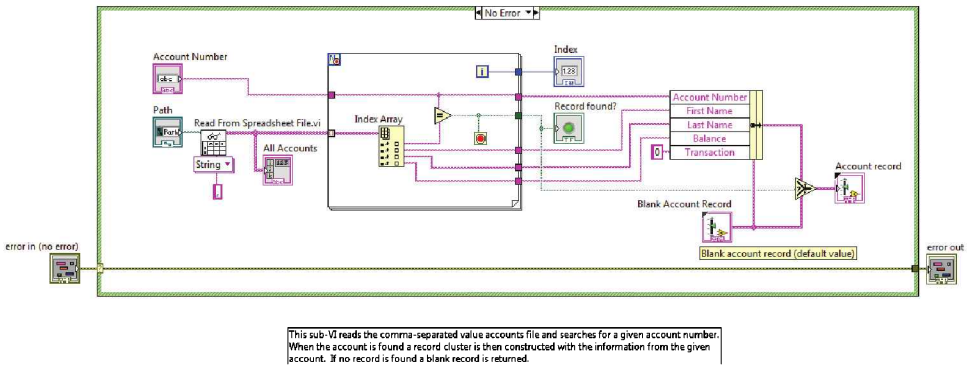


Figure M.5: Block diagram of the `Get Account` VI

Exercise 7.8

The block diagram of the `Verify Account` sub-VI is shown in Figure M.6. This sub-VI uses the `Get Account` sub-VI developed in Exercise 7.7 to attempt retrieval of the given account number. The *Record found?* output is used to determine whether or not the input account number exists in the accounts file. If it does exist, the *Main Menu* message is generated using the account information and the `Get Message` sub-VI developed in Exercise 7.6. If the account does not exist, the *Account Verification Failed* message is generated instead. The account record is passed as an output as is the *Record found?* Boolean.

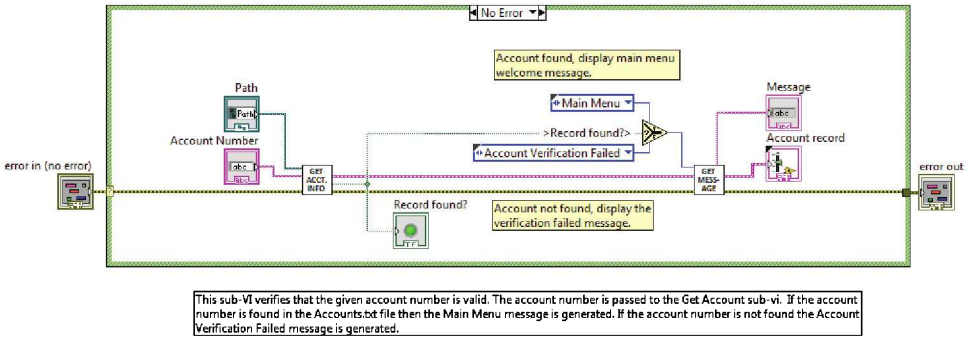


Figure M.6: Block diagram of the `Verify Account` VI

Exercise 7.9

An implementation of the `Update Account File` sub-VI is shown in Figure M.7. The account number of the input account is passed to the `Get Account` sub-VI developed in Exercise 7.7. This sub-VI generates an array of all of the accounts in the file and identifies the row index of the input account in the array. The input account cluster is converted to an array using the **`Cluster To Array`** function, and the resulting array is passed to the **`Replace Array Subset`** function along with the whole accounts array and the row index of the input account. The **`Replace Array Subset`** overwrites the account information at the given row index with the new

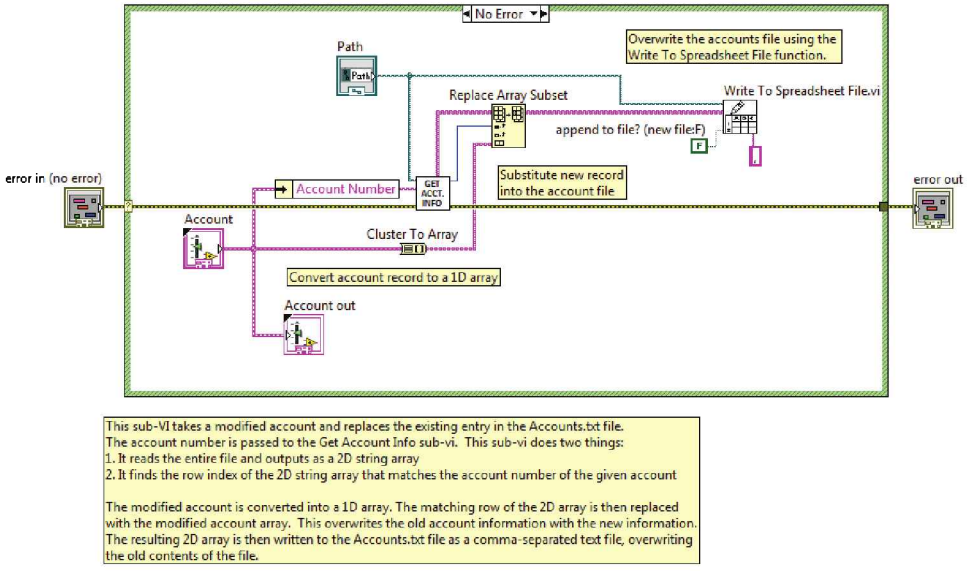


Figure M.7: Block diagram of the `Update Account File` VI

account information. The modified array is then wired to the **`Write To Spreadsheet File`** function. This function is configured with comma delimiter to produce a comma separated value text file. The function is also configure to overwrite the existing file, creating a new accounts file.

Exercise 7.10

An implementation of the `Deposit` sub-VI is shown in Figure M.8. The account balance is unbundled for the account cluster and is converted to a double-precision number using the **`Fract/Exp String To Number`** function. The deposit amount is also converted to a double-precision number using the same function. The two numbers are added, and the sum is converted to a string using the **`Format Into String`** function. This string is then bundled into the *Balance* field of the account cluster. The deposit string is also bundled into the *Transaction* field of the account cluster.

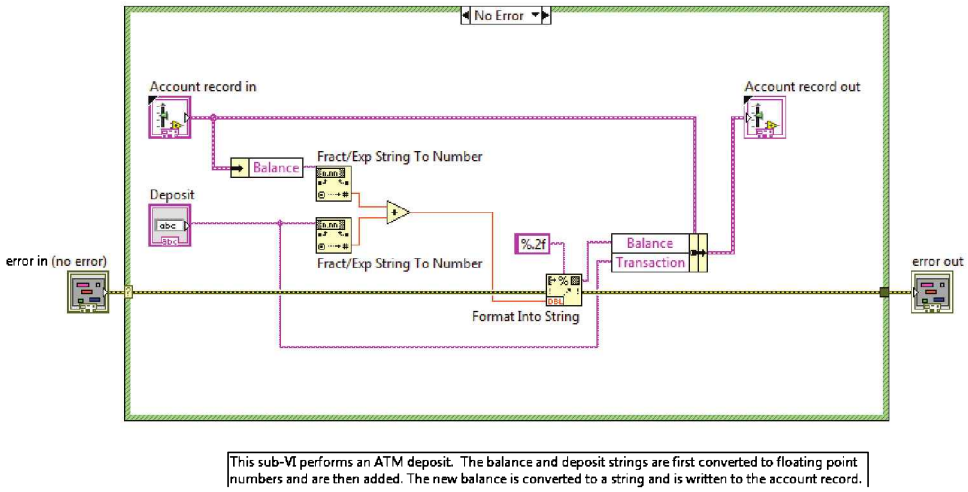


Figure M.8: Block diagram of the `Deposit` VI

Exercise 7.11

The `Withdrawal` sub-VI is similar to the `Deposit` sub-VI developed in Exercise 7.10 with the added complication that there may be insufficient funds in the account to cover the withdrawal. A implementation of the `Withdrawal` sub-VI is shown in Figure M.9. This figure shows the case where the withdrawal is successful. The balance string and the withdrawal string are converted to double-precision numbers using the **`Fract/Exp String To Number`** function and are subtracted. If the remaining funds after the withdrawal are greater than zero, the new balance is formatted into a string using the **`Format Into String`** function. The new balance string and the transaction string are then bundled into the account cluster, which is passed as an output. The *Sufficient funds?* Boolean output is set to True. Figure M.10 shows the case

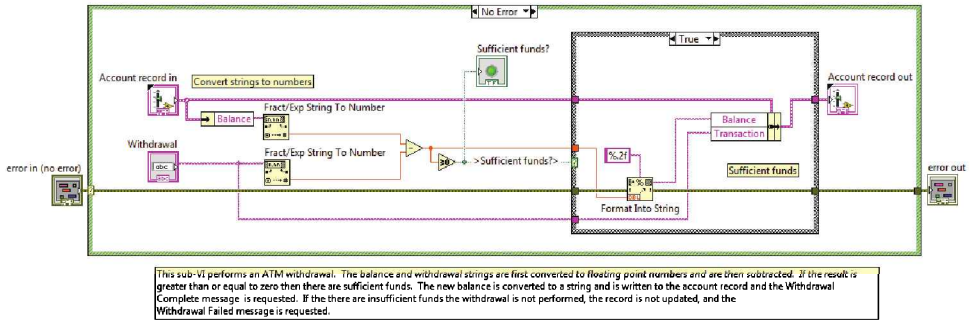


Figure M.9: Block diagram of the `Withdrawal` VI for a successful withdrawal

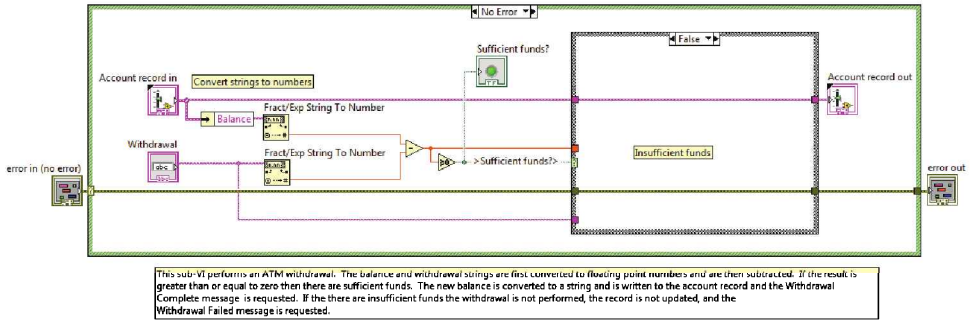


Figure M.10: Block diagram of the `Withdrawal` VI for an unsuccessful withdrawal

where there are insufficient funds for the withdrawal. In this case, the account cluster is not updated and the *Sufficient funds?* Boolean output is set to False.

Exercise 7.12

An implementation of the `Enter Button Next State` sub-VI is shown in Figure M.11. This sub-VI accepts an enumerated constant containing the mode of the ENTER button as an input and outputs the state following a value change of the ENTER button. The buttonEnter Mode enumerated constant is wired to the selector of a case structure. Figure M.11 shows that when the mode of the ENTER button is *Deposit*, the state following the value change of the ENTER button is *Complete Deposit*.

Exercise 7.13

The overall structure of the ATM controller and the *Initialize* state are shown in Figure M.12. The path to the accounts file is built in the input area of the loop and is passed as input to the loop. The left and right menu defaults are also defined in the input area and are used to initialize the shift registers for these indicators. The state shift register is initialized with the *Iniitialize* state.

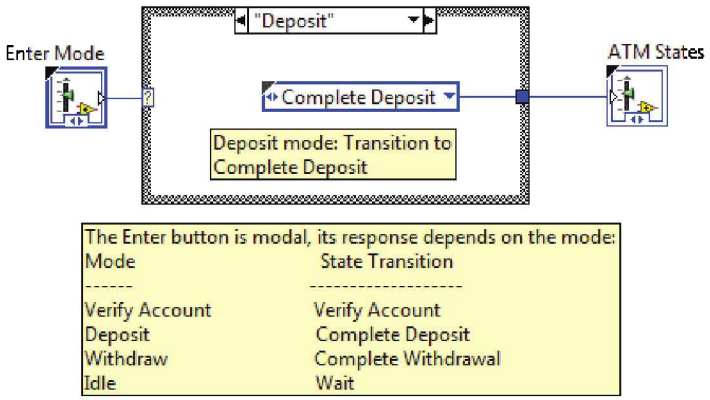


Figure M.11: Block diagram of the `Enter Button Next State` VI showing the *Deposit* case

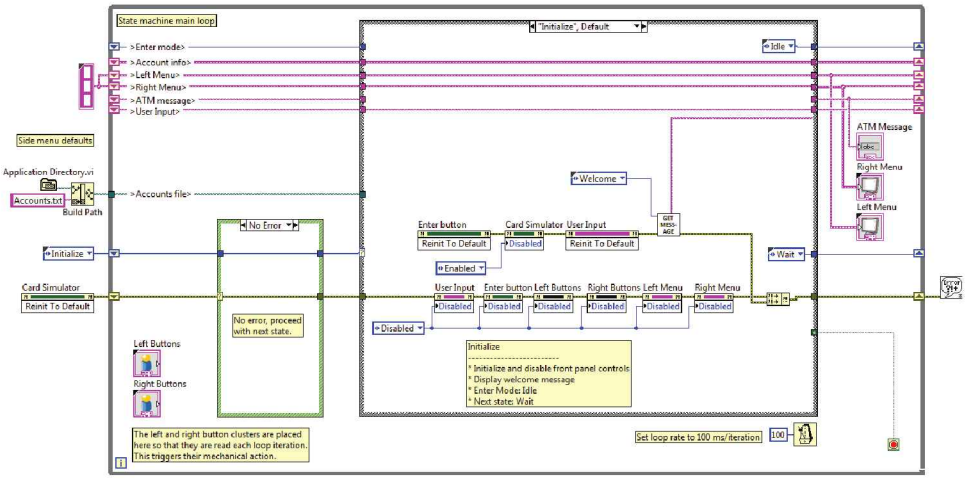


Figure M.12: The *Initialize* state of the ATM controller

Moving into the main loop, the guard clause checks to see if an error is present. If no error is present, the next state is passed to the selector of the main case structure. The *Initialize* state initializes the front panel controls. The CARD SIMULATOR button is enabled and the USER INPUT field, ENTER button, and side buttons and menus are disabled. The mode of the ENTER button is set to *Idle*, and the *Welcome* message is displayed on the central display. The *Wait* state is placed on the state shift register.

If an error is detected by the guard clause, the state machine moves immediately to the *Shutdown* state as seen in Figure M.13. This state shuts down the main loop. It also resets the CARD SIMULATOR button and the side menus and displays the *Session Terminate* message on the central display. The *Shutdown* state is wired to the shift register because the output tunnel of the case structure must have something wired to it, but since the state machine ends with this state, this stete will not be executed.

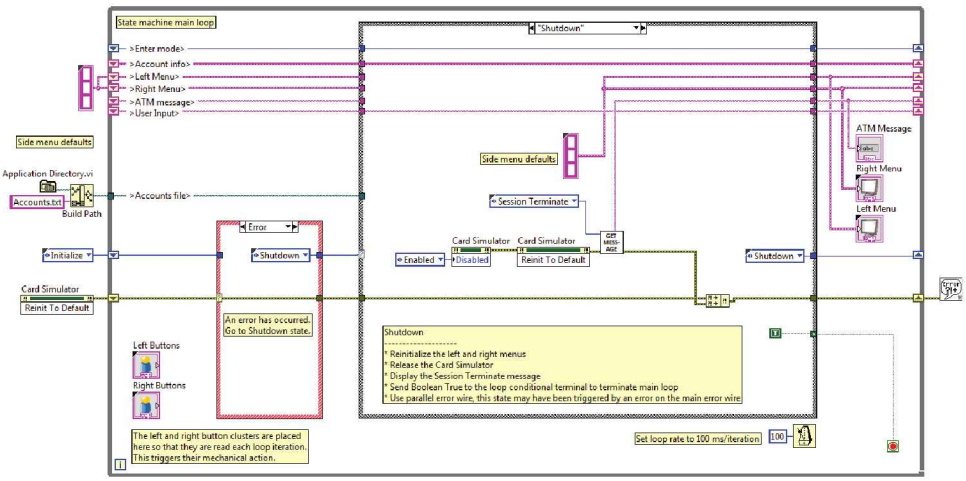


Figure M.13: The *Shutdown* state of the ATM controller

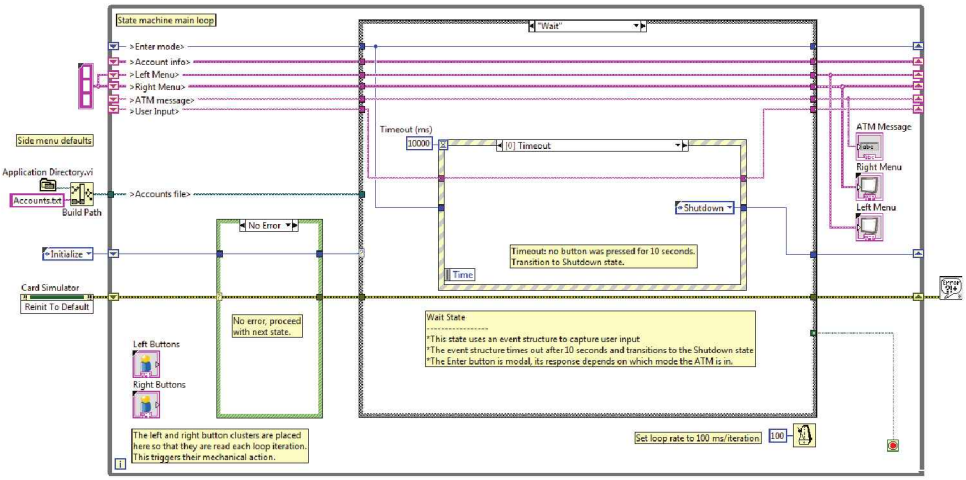


Figure M.14: The event case handling the *Timeout* of the event structure

The *Timeout* event case of the *Wait* state is shown in Figure M.14. The event structure is set to timeout after 10,000 milliseconds (or ten seconds). If the event structure times out, the *Timeout* event case executes and the *Shutdown* state is placed on the state shift register.

The event case handling the value change of the CARD SIMULATOR is shown in Figure M.15. This state places the *Card Entered* state on the shift register. This state is seen in Figure M.16. This state enables the USER INPUT field and the ENTER button and disables and grays out the CARD SIMULATOR button. The key focus is placed on the USER INPUT field using the *Key Focus* property node, satisfying a requirement of the

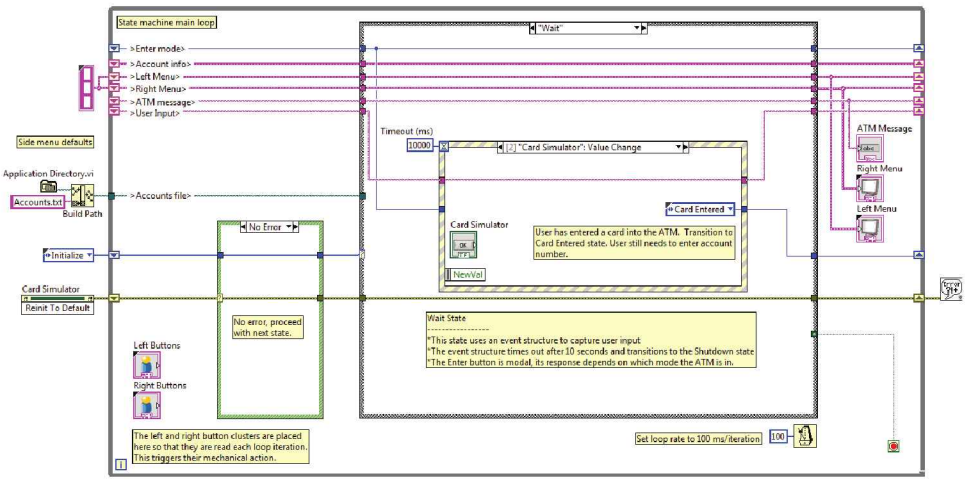


Figure M.15: The event case handling the value change of the CARD SIMULATOR control

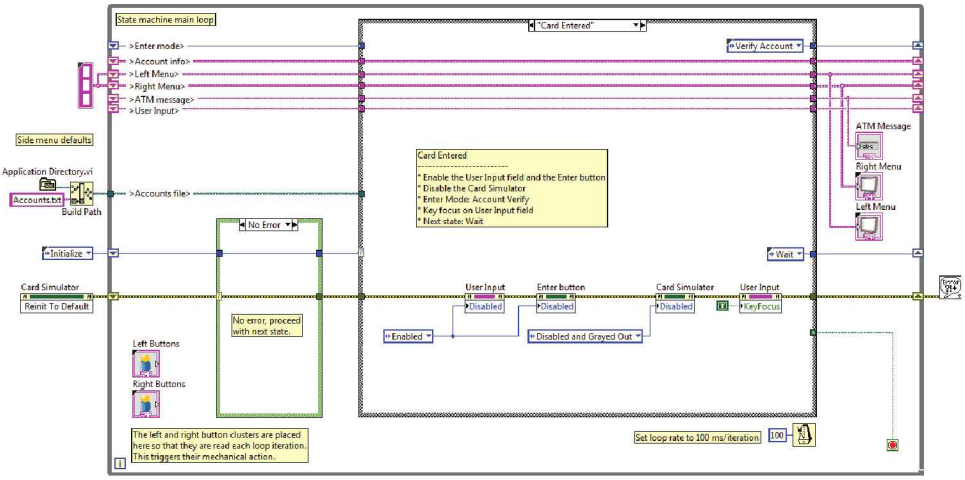


Figure M.16: The *Card Entered* state of the ATM controller

functional specification. The mode of the ENTER button is set to *Verify Account*, and the *Wait* state is placed on the state shift register.

The *Wait* state now waits for the user to enter an account number by pressing the ENTER button when the data entry is complete. The event case handling the value change of the ENTER button is shown in Figure M.17. This event case reads the mode of the ENTER button from the data highway (currently *Verify Account*) and uses the `Enter Button Next State` sub-VI shown in Figure M.11 to determine the next state. In this case, the next state is the *Verify Account* state shown in Figure M.18. This state uses the `Verify Account` sub-VI shown in Figure M.6 to search for an account matching the one entered by the user.

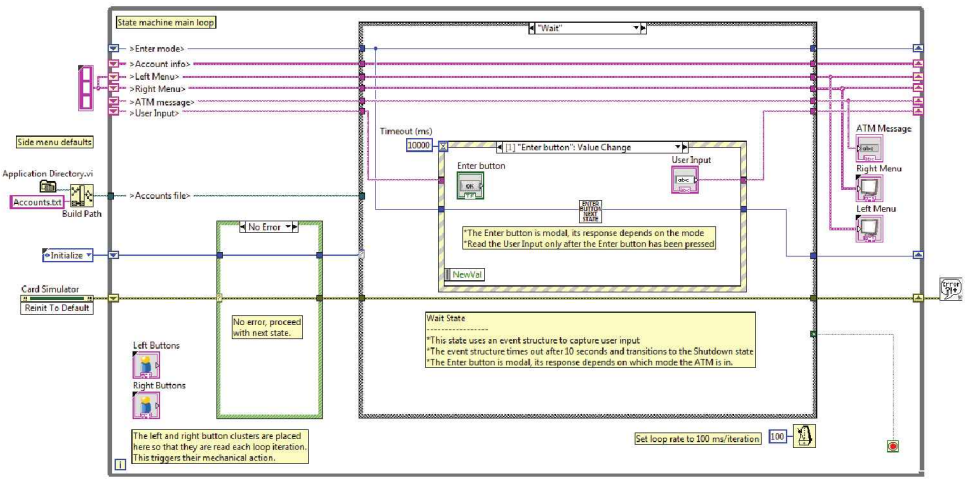


Figure M.17: The event case handling the value change of the ENTER button

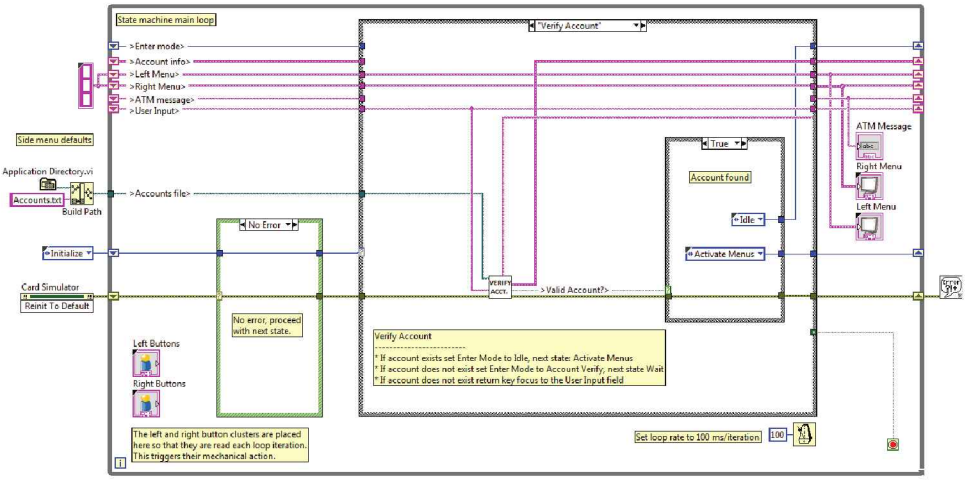


Figure M.18: The *Verify Account* state of the ATM controller showing the valid account case

If an account is found, the mode of the ENTER button is set to *Idle*, and the *Activate Menus* state is placed on the shift register. The *Main Menu* message is displayed on the center ATM display by the `Verify Account` VI. The *Idle* mode allows the ENTER button to be pressed with no effect, so if the user presses ENTER while the state machine is simply waiting, nothing will happen. If an account is not found, the mode of the ENTER button is set to *Verify Account* and the *Wait* state is placed on the shift register, as shown in Figure M.19. The *Account Verification Failed* message is displayed on the central ATM display by the `Verify Account` VI, and the state machine once again

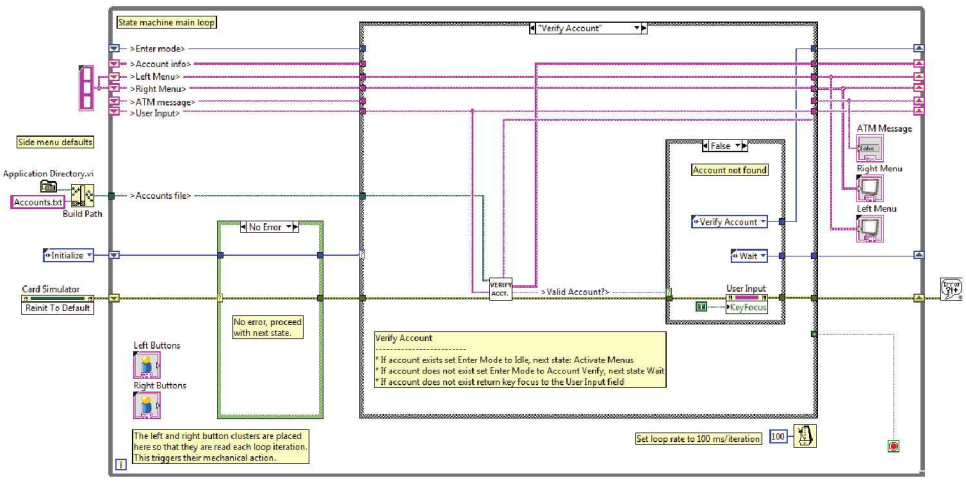


Figure M.19: The *Verify Account* state of the ATM controller showing the invalid account case

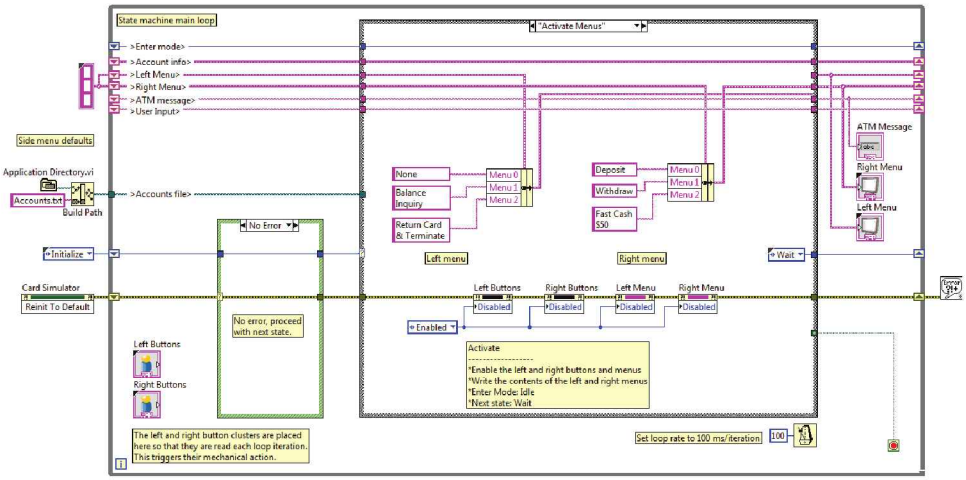


Figure M.20: The *Activate Menus* state of the ATM controller

waits for the user to enter an account number. The key focus is placed back on the USER INPUT control using an implicitly linked KEY FOCUS property node of the USER INPUT button.

When a valid account number has been entered, the state machine moves to the *Activate Menus* state shown in Figure M.20. This state bundles all of the left and right menu strings into their corresponding clusters and places them on the data highway. The left and right buttons and menu displays are also enabled in this state. The mode of the ENTER button is still *Idle*, so it is passed through this state unchanged. The *Wait* state is placed on the state shift register.

The state machine is now at the second *Wait* state shown in Figure M.1. The side menus are activated and new transitions from this state are now available. Starting with the left ATM buttons, the top button has no current function and is reserved for future use. Pressing this button should have no effect, as shown in Figure M.21. This event case takes no action other than placing the *Wait* state back on the state shift register.

The event case handling the value change of the BALANCE INQUIRY button is shown in Figure M.22. This event case places the *Balance Inquiry* state on the state shift register, as shown in Figure M.23. This state uses the current account information and

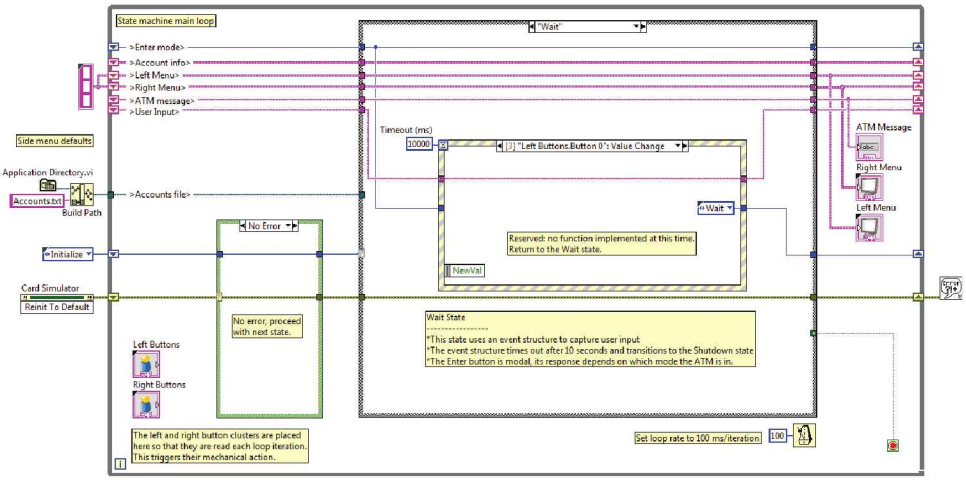


Figure M.21: The event case handling the value change of the NONE button

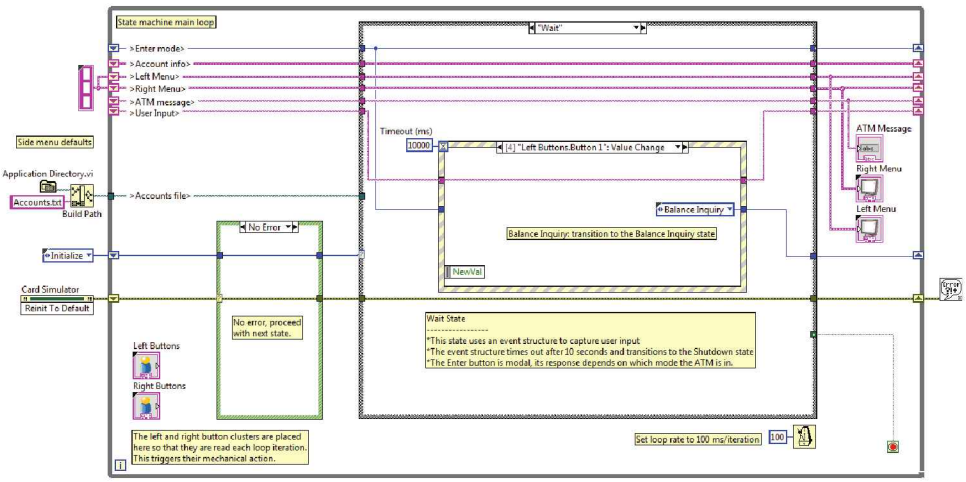


Figure M.22: The event case handling the value change of the BALANCE INQUIRY button

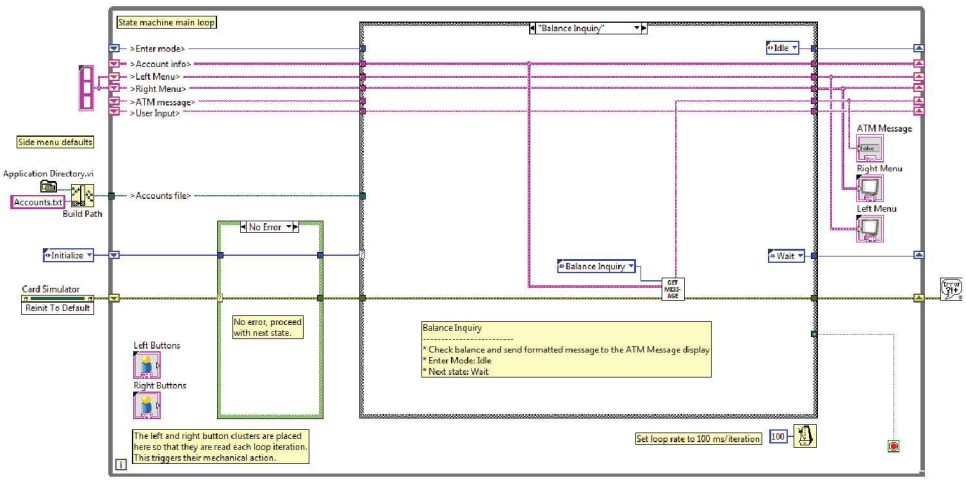


Figure M.23: The *Balance Inquiry* state of the ATM controller

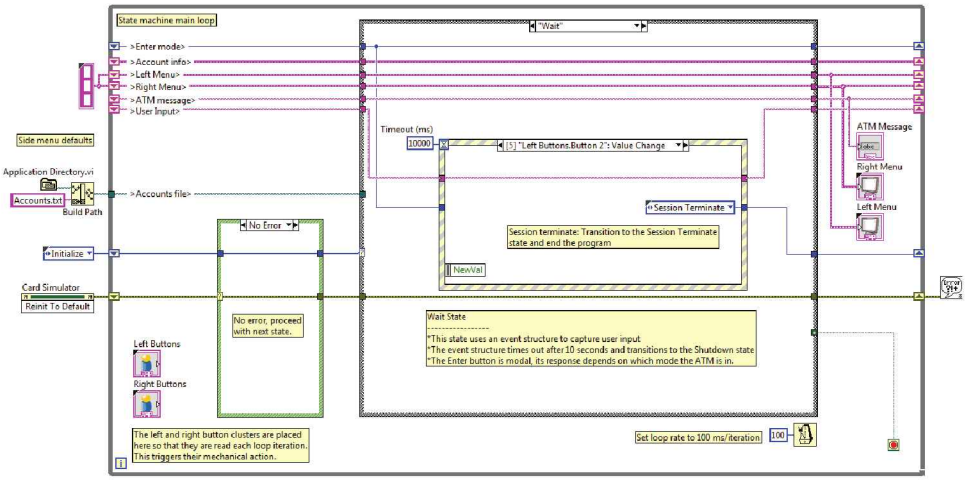


Figure M.24: The event case handling the value change of the RETURN CARD & TERMINATE button

the `Get Message` sub-VI to display a *Balance Inquiry* message personalized with the current account balance. The mode of the ENTER button is set to *Idle*, and the *Wait* state is placed on the state shift register.

The event case handling the value change of the RETURN CARD & TERMINATE button is shown in Figure M.24. This state puts the *Session Terminate* state shown in on the state shift register. Figure M.25 the *Session Terminate* message is displayed on the central ATM screen, and the *Shutdown* state is placed on the state shift register.

We now move on to the right ATM buttons. The event case handling the value change of the DEPOSIT button is shown in Figure M.26. This state places the *Initiate*

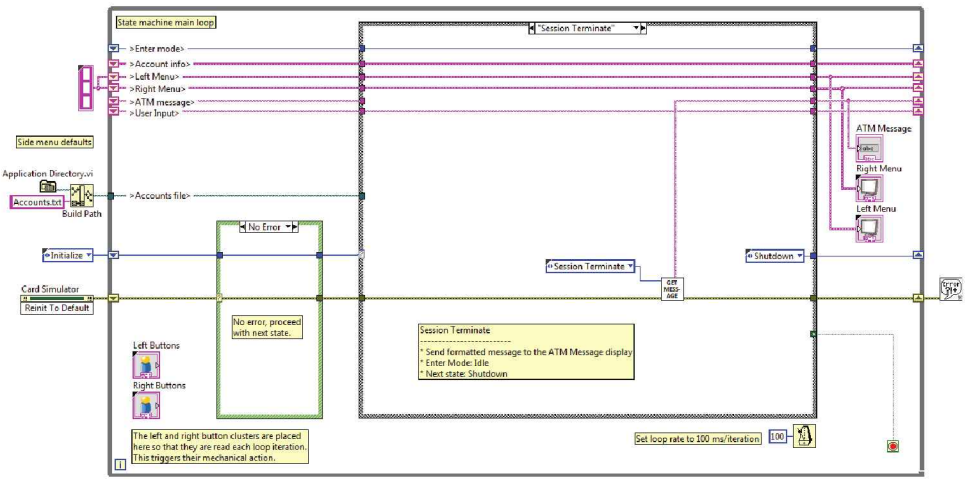


Figure M.25: The *Session Terminate* state of the ATM controller

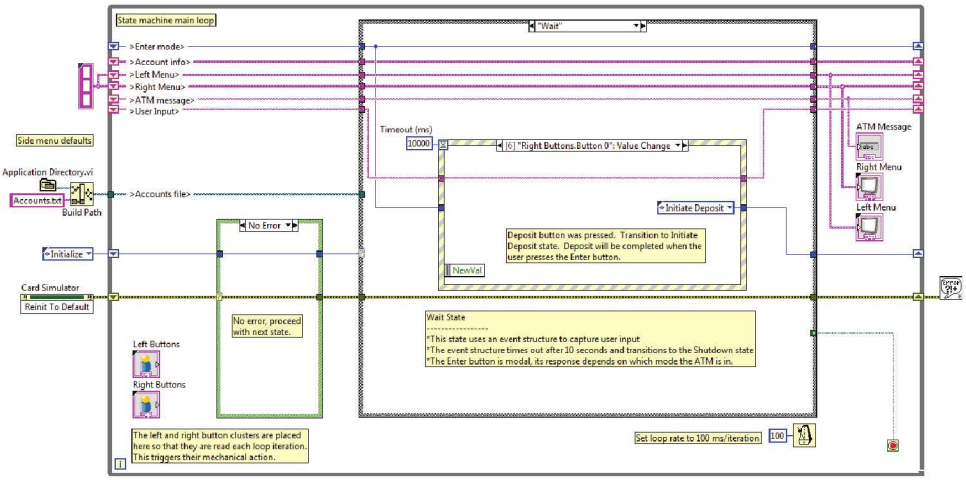


Figure M.26: The event case handling the value change of the DEPOSIT button

Deposit state on the state shift register as shown in Figure M.27. The *Initiate Deposit* state puts the key focus on the USER INPUT field and displays the *Deposit* message. The mode of the ENTER button is changed to *Deposit*, and the *Wait* state is placed on the shift register.

The *Wait* state waits for the user to press the ENTER button to signal that the amount of the deposit has been entered in the USER INPUT field. When the *Enter* button is pressed (see Figure M.17), the *Deposit* mode triggers a transition to the *Complete Deposit* state shown in Figure M.28. This state executes the deposit using the `Deposit` sub-VI shown in Figure M.8. The *Deposit Complete* message is displayed on the ATM

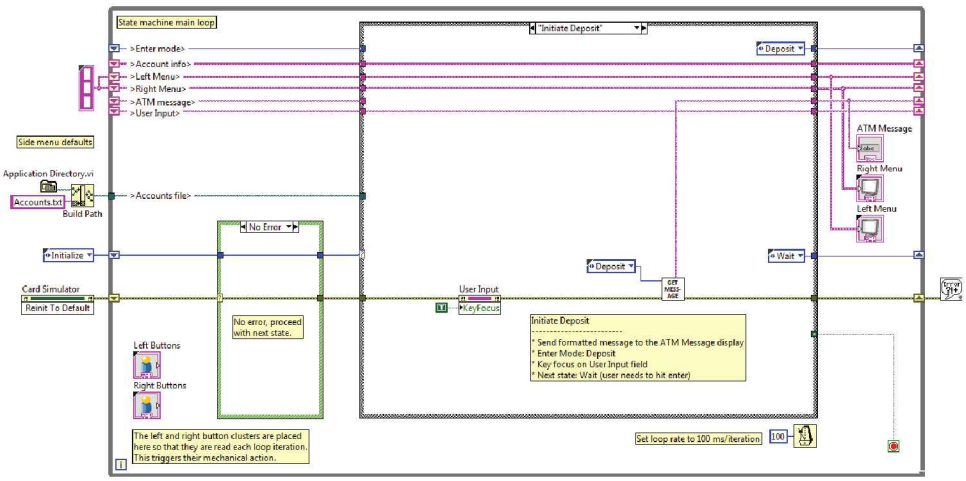


Figure M.27: The *Initiate Deposit* state of the ATM controller

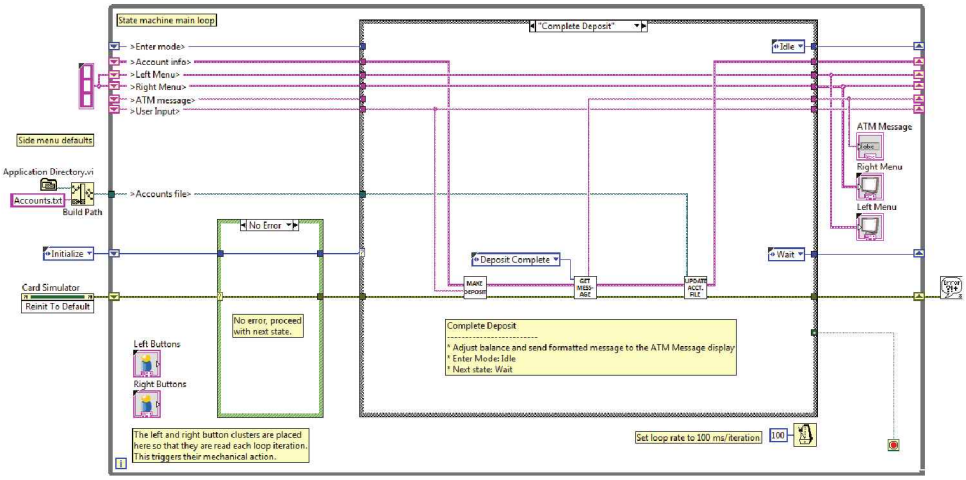


Figure M.28: The *Complete Deposit* state of the ATM controller

central display. The account file is then updated using the `Update Account File` sub-VI shown in Figure M.7. The mode of the ENTER button is then set to *Idle*, and the state machine returns to the *Wait* state.

What would have happened if the user had pressed some other button other than the ENTER button after pressing the DEPOSIT button? In that case, the event structure would have responded to the value change of the button that was pressed. The intent would be to discard the deposit request and to service the new user request instead. This is why we had to set the mode of ENTER button to *Idle* in the *Balance Inquiry* state. Suppose the user presses the DEPOSIT button, but then decides not to complete the operation and presses the BALANCE INQUIRY button to check the account balance. The

deposit operation has now been abandoned, so the mode of the ENTER button needs to be reset to *Idle*.

The event case handling the value change of the WITHDRAW button is shown in Figure M.29. This state places the *Initiate Withdrawal* state on the state shift register shown in Figure M.30. The *Initiate Withdrawal* state puts the key focus on the USER INPUT field and displays the *Withdrawal* message. The mode of the ENTER button is changed to *Withdraw* and the *Wait* state is placed on the shift register.

The *Wait* state waits for the user to press the ENTER button to signal that the amount of the withdrawal has been entered in the USER INPUT field. When the *Enter* button is pressed (see Figure M.17), the *Withdraw* mode triggers a transition to the *Complete Withdrawal* state. Figure M.31 shows the *Complete Withdrawal* state for the

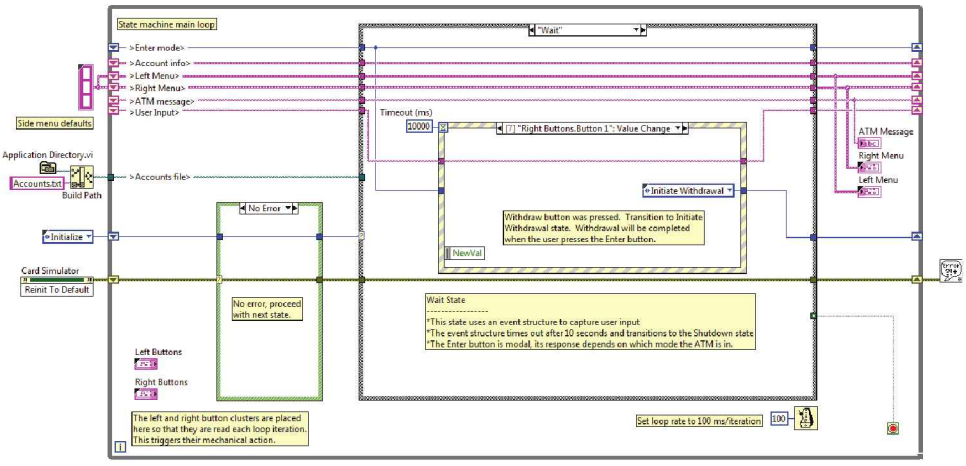


Figure M.29: The event case handling the value change of the WITHDRAW button

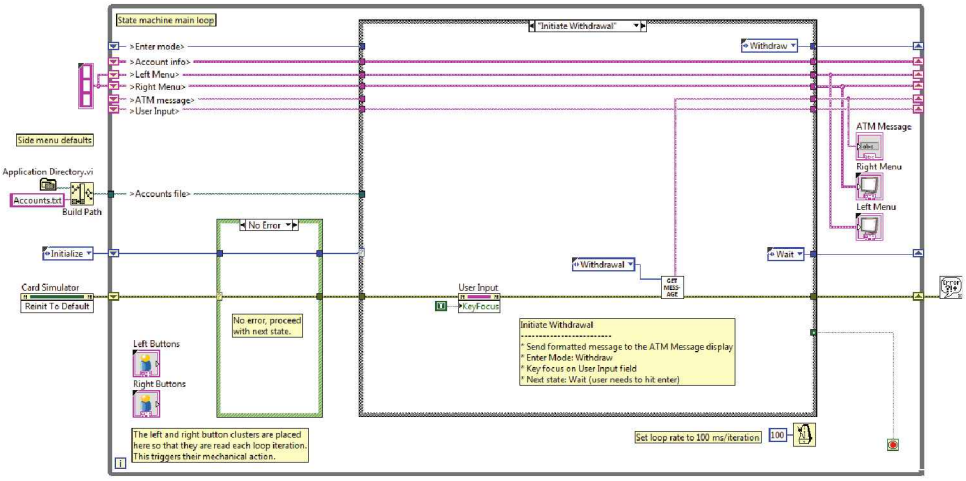


Figure M.30: The *Initiate Withdrawal* state of the ATM controller

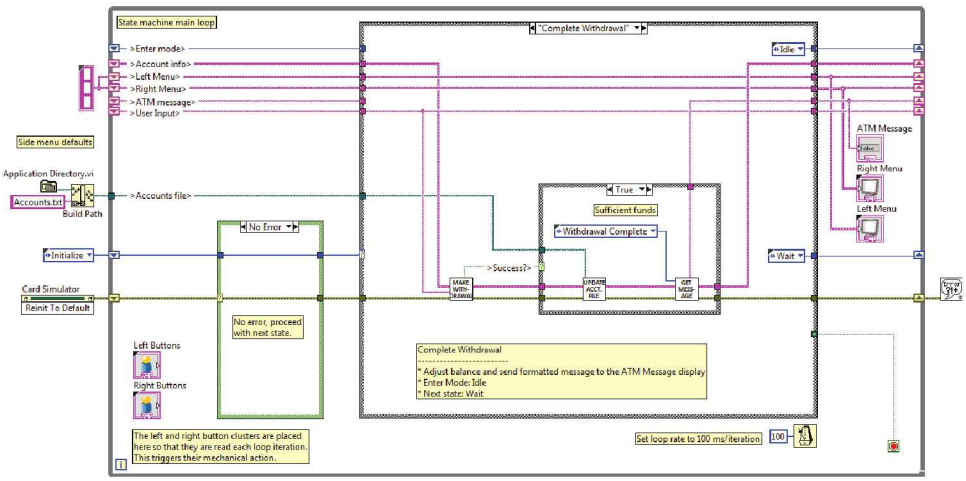


Figure M.31: The *Complete Withdrawal* state of the ATM controller showing the sufficient funds case

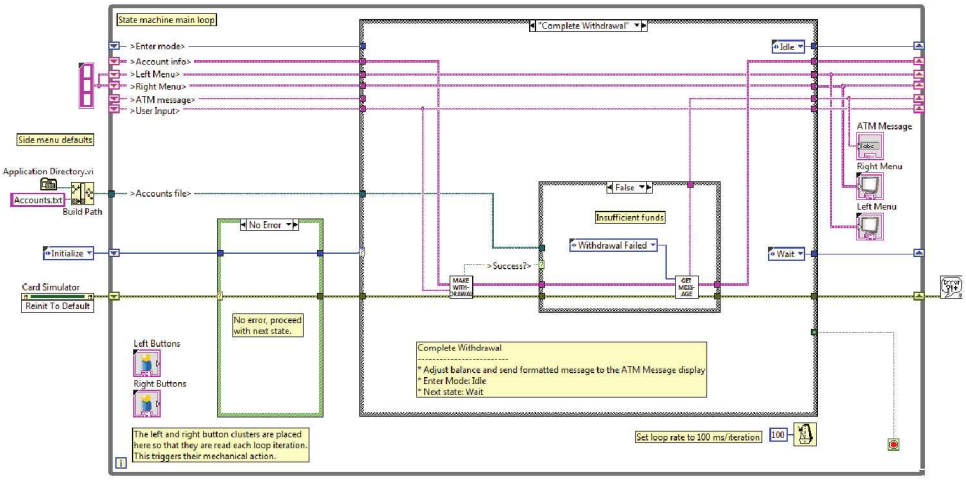


Figure M.32: The *Complete Withdrawal* state of the ATM controller showing the insufficient funds case

case where there are sufficient funds to cover the withdrawal. This state executes the withdrawal using the `Withdrawal` sub-VI shown in Figure M.9 and Figure M.10. The *Withdrawal Complete* message is displayed on the ATM central display if there are sufficient funds for the withdrawal; otherwise, the *Withdrawal Failed* message is displayed. If the withdrawal is successful, the account file is updated using the `Update Account File` sub-VI shown in Figure M.7. The mode of the ENTER button is then set to *Idle* and the state machine returns to the *Wait* state. Figure M.32 shows the *Complete Withdrawal* state when the funds are not sufficient for the requested withdrawal.

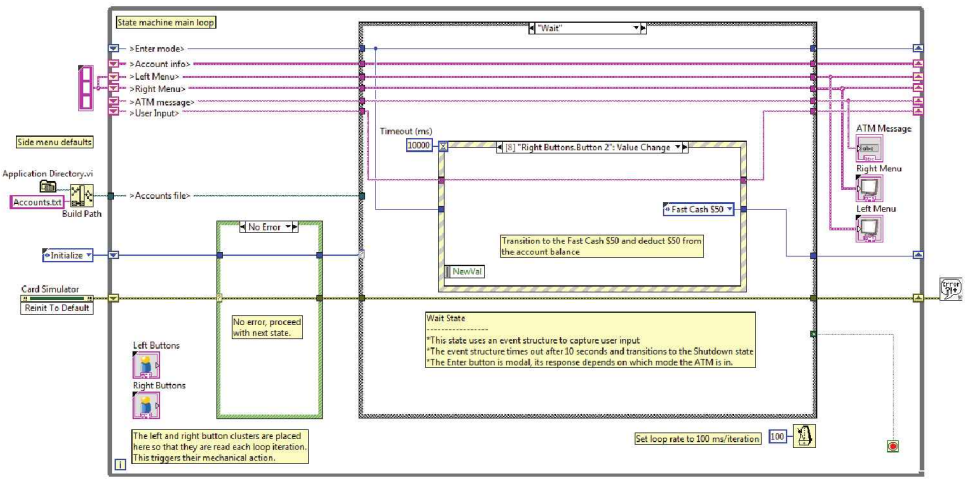


Figure M.33: The event case handling the value change of the FAST CASH $50 button

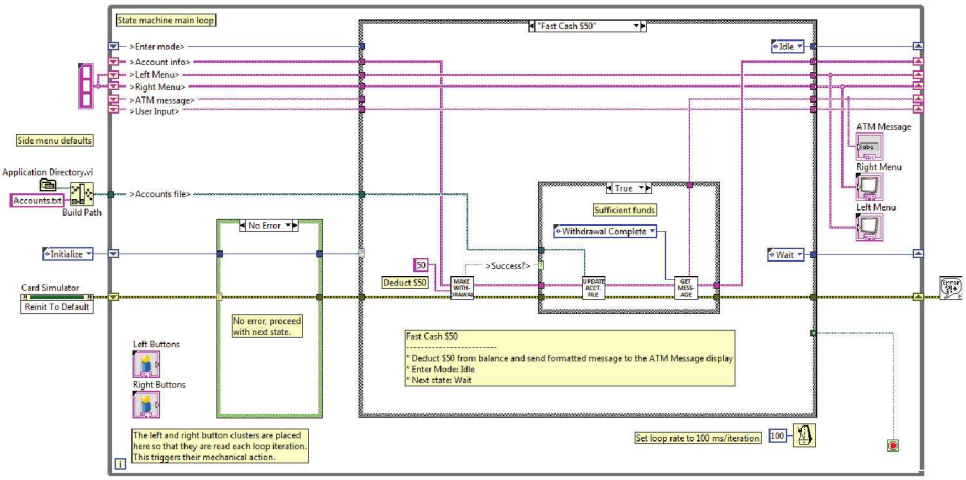


Figure M.34: The *Fast Cash $50* state of the ATM controller showing the sufficient funds case

The final button is the FAST CASH $50 button. The event case handling the value change of this button is shown in Figure M.33. This state places the *Fast Cash $50* state on the state shift register, shown in Figure M.34. This state is a one-step withdrawal of $50 with no need to monitor the ENTER button. If there are sufficient funds, the account file is updated and the *Withdrawal Complete* message is displayed on the ATM central indicator. If there are insufficient funds, as shown in Figure M.35, the account file is not updated and the *Withdrawal Failed* message is displayed. This state resets the mode of the ENTER button to *Idle* and places the *Wait* state on the state shift register.

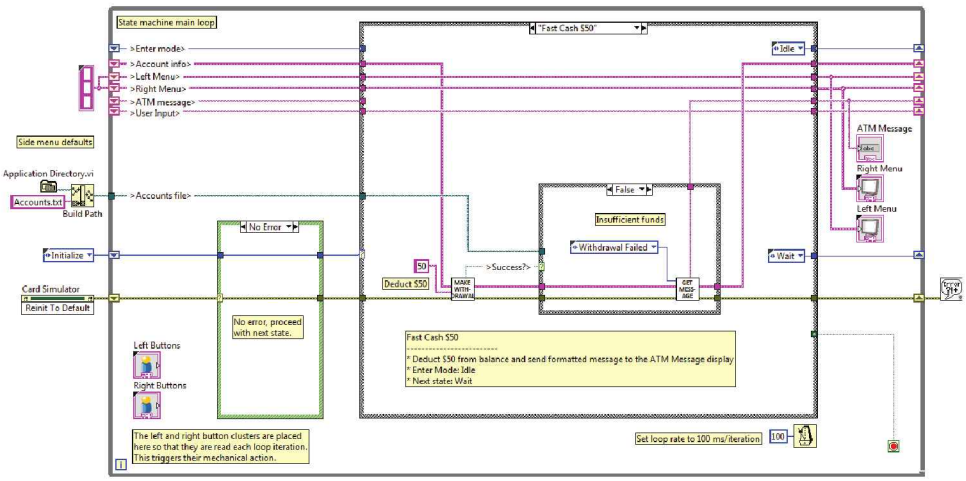


Figure M.35: The *Fast Cash $50* state of the ATM controller showing the insufficient funds case

This concludes the development of the ATM controller. The main development effort was in developing the state transition diagram and the support sub-VIs. The main VI itself was simply an event-driven classic state machine. Developing the individual states of the state machine is straightforward when given a good state transition diagram as a starting point.

The ATM controller is a good example of how to code an event-driven classic state machine. The event structure serves as a user-driven switching station. The event structure exists simply to route the state machine to the correct state for a given user input.



Problem Set 8 Solutions

Exercise 8.1

The event case handling the value change of the WRITE CONFIGURATION button of the string-based DAQ state machine is shown in Figure N.1. The string-based queue is formed in the input area of the main loop. The `Split and Queue String` sub-VI is then used to place the first three states on the queue. This improves readability and conserves screen space. Compare this figure to Figure 15.14. In this version of the DAQ state machine only the *Initialize* state was enqueued in the input area of the loop. The *Build DAQ Task* state and the *Wait* state were then enqueued within the *Initialize* state in order to save space. Since these three states are always executed in this order when the program starts, it improves readability when they are enqueued in the input area of the loop.

Notice that readability is also improved in the event case shown in Figure N.1. The `Split and Queue String` sub-VI is again used to enqueue multiple states using a single string constant. Compare this with Figure K.30. Placing multiple **`Enqueue Element`** functions in a single event case can be awkward. The string-based implementation is cleaner and easier to read. The other event cases are similar and will not be shown here.

One disadvantage of the string-based state machine is that a *Default* state needs to be defined to catch undefined or misspelled states. The *Default* state of the string-based DAQ state machine is shown in Figure N.2

Exercise 8.2

Using clusters for arguments instead of strings is convenient because you don't need to develop a state/argument convention and the associate parsing sub-VIs. However, the cluster-based approach can take up much more screen real estate than strings. They are more awkward to use and edit, and the arguments must always be used even for states where they are not needed. Also, since every cluster is a unique data type it is difficult to reuse code written for cluster-based arguments.

The overall architecture and the *Initialize* state of the cluster-based implementation of the car wash controller with arguments is shown in Figure N.3. The queue is

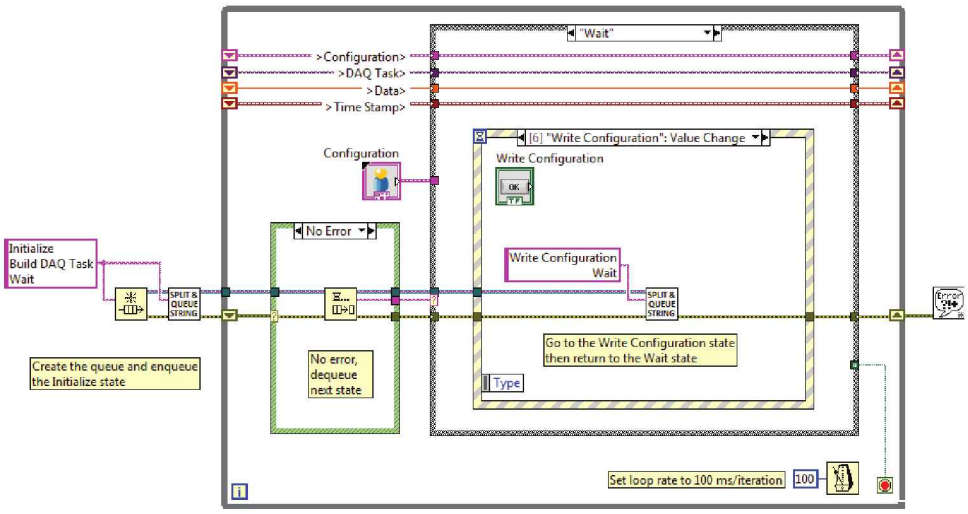


Figure N.1: The event case handling the value change of the WRITE CONFIGURATION button of the string-based DAQ state machine

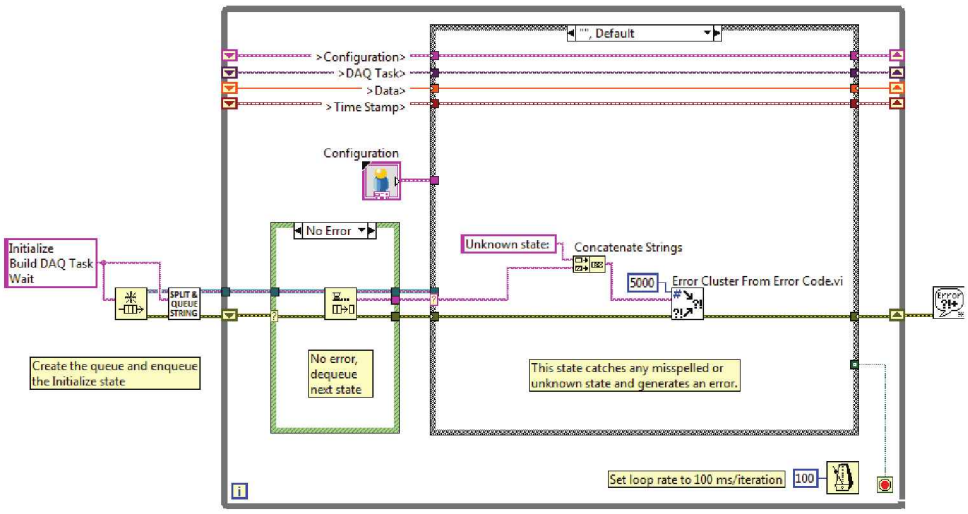


Figure N.2: The *Default* state of the string-based DAQ state machine

created in the input area of the loop. Notice that the arguments need to be included in the cluster even though they are not needed. It is also not as easy to queue up multiple states in the input area of the loop. This would require either a string of **`Enqueue Element`** functions or it would require an array of clusters and a **`For`** loop in the input area.

Both of these options require much more screen real estate than using the multiline string constant as shown in Figure 21.13. In the string-based version, we could

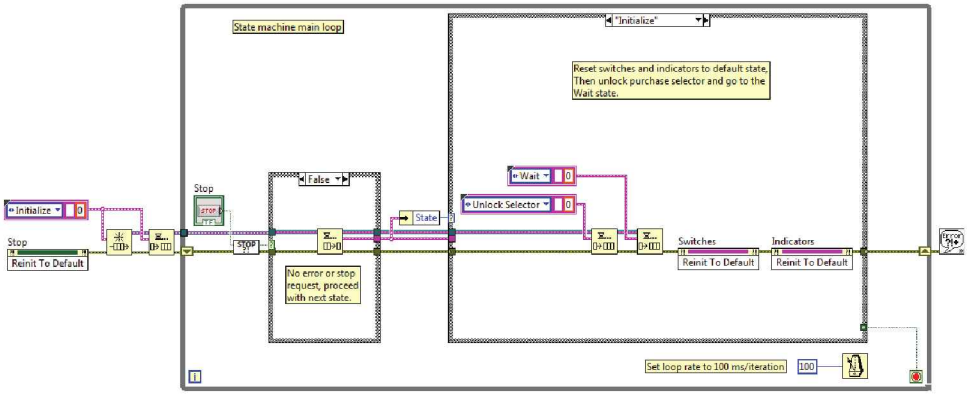


Figure N.3: The *Initialize* state of the car wash controller with cluster-based arguments

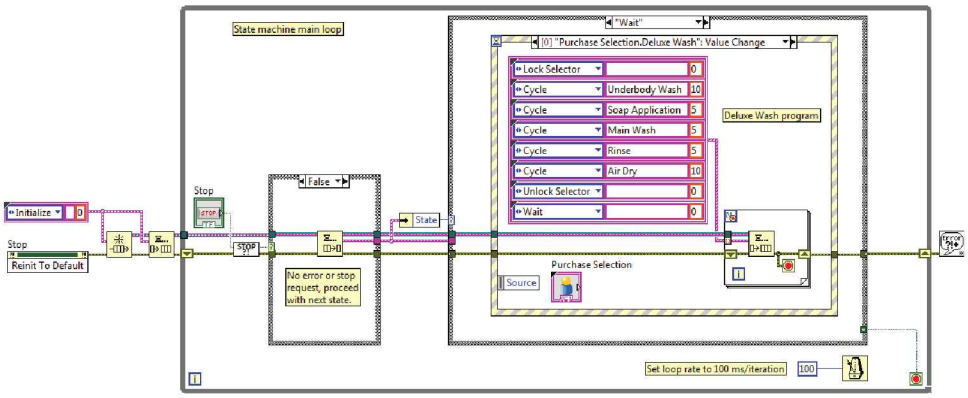


Figure N.4: The event case handling the value change of the DELUXE WASH button for the car wash controller with cluster-based arguments

use the general-purpose `Split and Queue String` sub-VI. This sub-VI will work with any string-based queue since all string-based queues have the same data type. Since it is too awkward to queue up the *Unlock Selector* and *Wait* states in the input area of the cluster-based state machine, they are queued up in the *Initialize* state instead. This makes the flow of the code harder to read.

The contrast in readability between the string-based and cluster-based codes is especially evident in the event case handling the value change of the DELUXE WASH button as seen in Figure N.4. This state uses an array constant of clusters with arguments and a **`For`** loop to queue up the states and arguments needed for a Deluxe wash. This array constant is large and awkward to edit. Each cluster requires the use of arguments even though many of the states need no arguments. The constant is not easy to read, and has many thick border lines of different colors distracting the reader and decreasing readability. It is difficult to add or subtract elements, particularly in

the middle of the array. This decreases maintainability. The same issues plague the event case handling the value change of the ECONOMY WASH button, seen in Figure N.5. Even the event case handling the value change of the STOP button has the same issues, as seen in Figure N.6. The *Shutdown* state doesn't need arguments, but the cluster containing this state needs to include the arguments. Compare these figures with the corresponding string-based controller states shown in Figures 21.17, 21.18, and 21.16.

The *Lock Selector* and *Unlock Selector* states don't use the arguments and are very similar to the corresponding states in the string-based car wash controller with arguments and so they will not be reproduced here. The *Cycle* state is also similar to the string-based version as can be seen in Figure N.7. The main difference is that the car wash cycle sub-VI needs to be modified to accept the cluster based arguments. The block diagram of the `Car Wash Cycle Cluster Based` sub-VI is shown

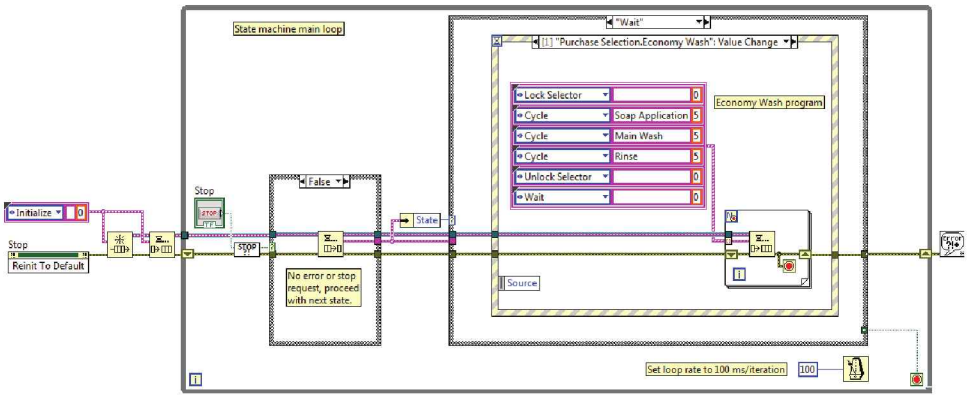


Figure N.5: The event case handling the value change of the ECONOMY WASH button for the car wash controller with cluster-based arguments

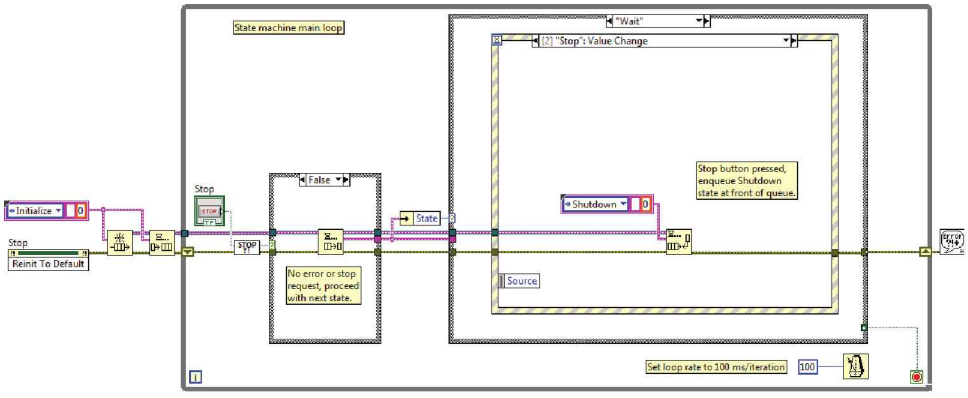


Figure N.6: The event case handling the value change of the STOP button for the car wash controller with cluster-based arguments

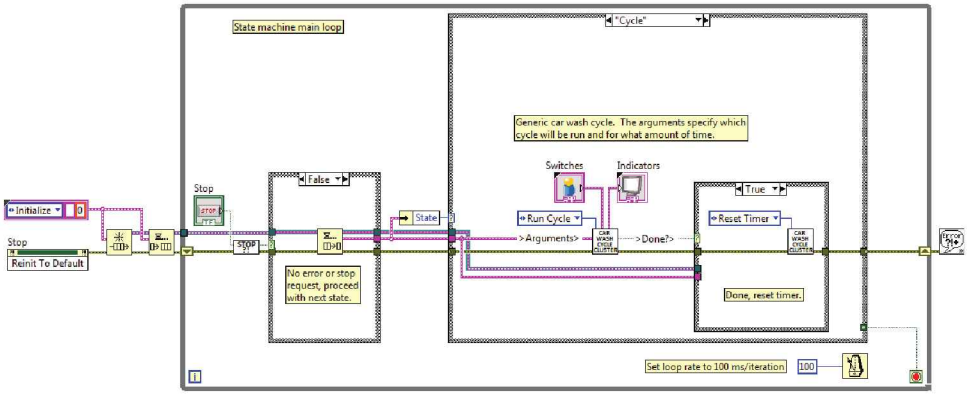


Figure N.7: The *Cycle* state of the car wash controller with cluster-based arguments

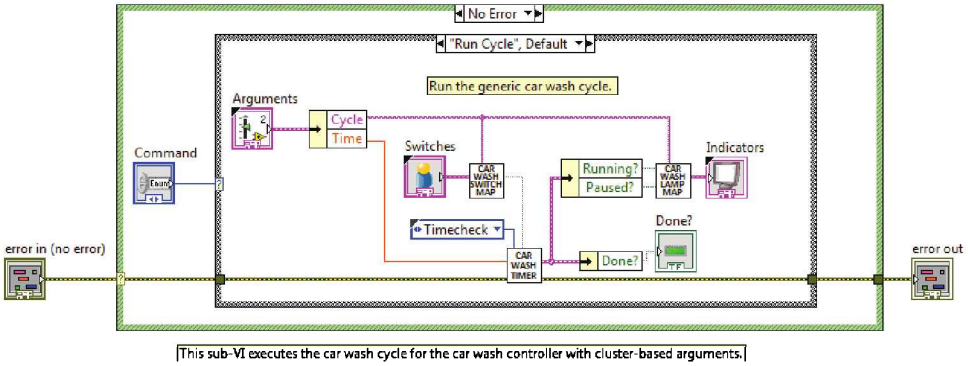


Figure N.8: The cluster-based car wash cycle action engine

in Figure N.8. Compare this with the string-based version developed in Figure 21.12. Both versions use the same car wash timer action engine and the same mapping sub-VIs. The cluster-based version unbundles the cycle name and time target from the argument cluster, while the string-based version parses the cycle name and time target from the argument string. This concludes the development of the cluster-based car wash controller with arguments.

Exercise 8.3

The queued-string car wash can be modified to read the Deluxe and Economy program strings from text files by reading the text files in the input area of the loop. Since the programs are constant while the program is running, it makes sense to put them in the input area of the loop. Persistent storage of the programs can be achieved by wiring the program strings as inputs to the main loop. Since the program strings never change, there is no need for a data highway. This is demonstrated in Figure N.9. The absolute path to the Deluxe wash text file is created using the **`Application`**

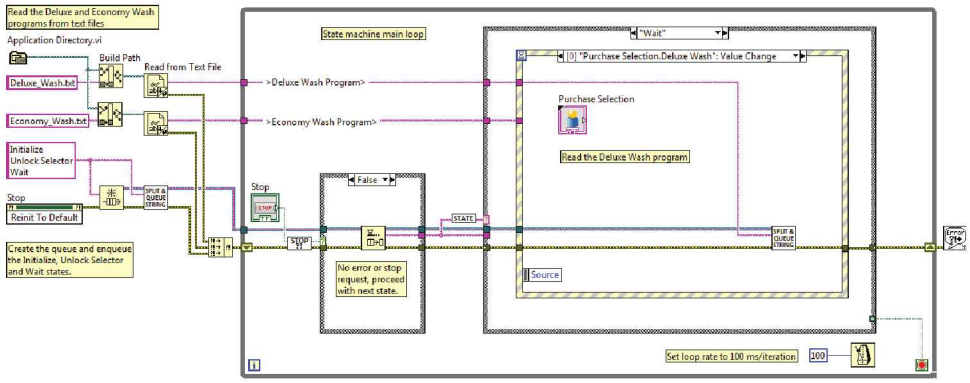


Figure N.9: The event case for handling the value change of the DELUXE WASH button for the revised string-based car wash controller

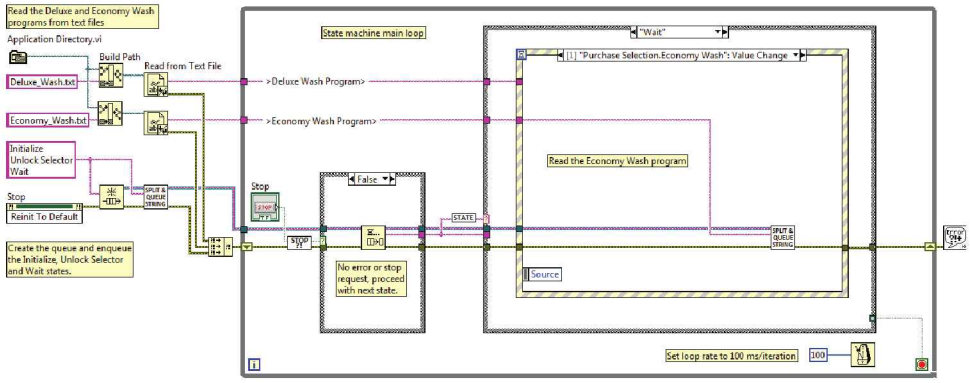


Figure N.10: The event case for handling the value change of the ECONOMY WASH button for the revised string-based car wash controller

`Directory` function (see the help file in Figure 1.62) and the **`Build Path`** function (see the help file in Figure 1.63). The path is then wired to the **`Read from Text File`** function to read the Deluxe Wash instructions from the text file. The resulting multiline string is then wired as an input to the main loop and through the lobby to the main case structure. From there, the Deluxe Wash instruction string is wired to the event structure in the *Wait* state. When the DELUXE WASH button is pressed, the Deluxe wash instruction string is split and placed on the queue by the `Split and Queue String` sub-VI developed in Figure 20.27.

The same process is followed for the Economy Wash instruction string as shown in Figure N.10. The absolute path to the text file containing the string is built in the input area of the main loop. The string is then read and is wired as an input to the main loop, the main case structure, and finally to the event structure in the *Wait* state. When the ECONOMY WASH button is pressed, the Economy wash instruction string is

split and placed on the queue by the `Split and Queue String` sub-VI. The rest of the program is indentical to the queued string car wash controller developed in Chapter 21. This concludes the development of the revision of the queued-string car wash controller.

Exercise 8.4

The block diagram for the `Signal Map` sub-VI is shown in Figure N.11. This figure shows the *N-S Green* case. This sub-VI accepts the name of a signal, the duration of green/red signals and the duration of yellow signals as inputs. The output of the sub-VI is the correct signal pattern cluster and the appropriate signal duration. This is accomplished by wiring the name string to the selector of a case structure. A default value cluster constant of the traffic signal cluster is passed to the central input of a **`Bundle By Name`** function. The appropriate lights are turned on by bundling a Boolean True for the lights that are on. The resulting cluster and the appropriate duration are then passed to the output of the function. The seven other signal states are similar and are not shown here.

The block diagram for the `Next Traffic State String` sub-VI is shown in Figure N.12. This sub-VI accepts a signal name and the vehicle sensor cluster as inputs and it outputs the name of the next state. Many of the traffic states have fixed transitions, such as the *N-S Green* state shown in Figure N.12. Other states require a decision based on the vehicle sensor cluster, such as the *N-S Yellow* state shown in Figure N.13.

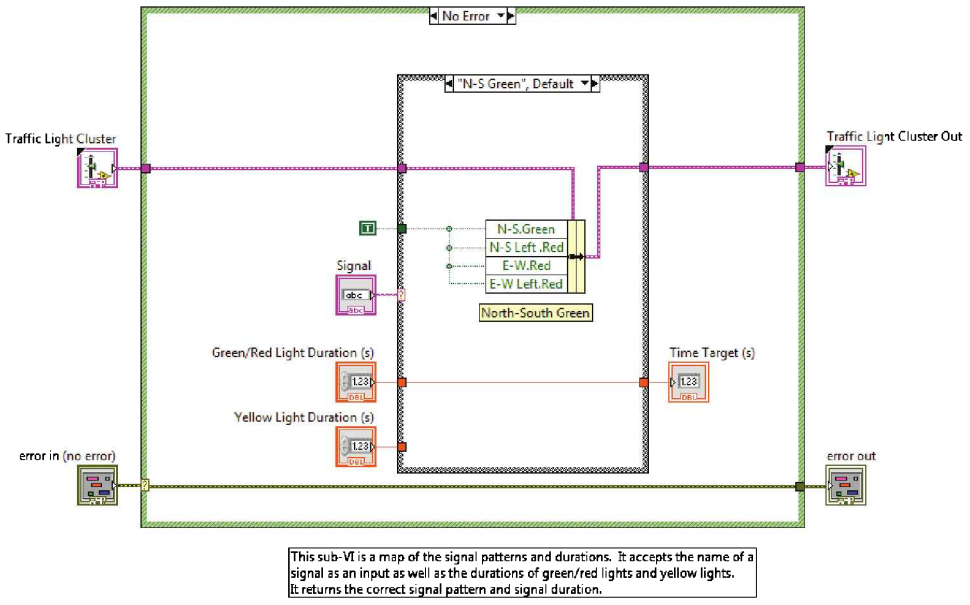


Figure N.11: The block diagram of the `Signal Map` sub-VI

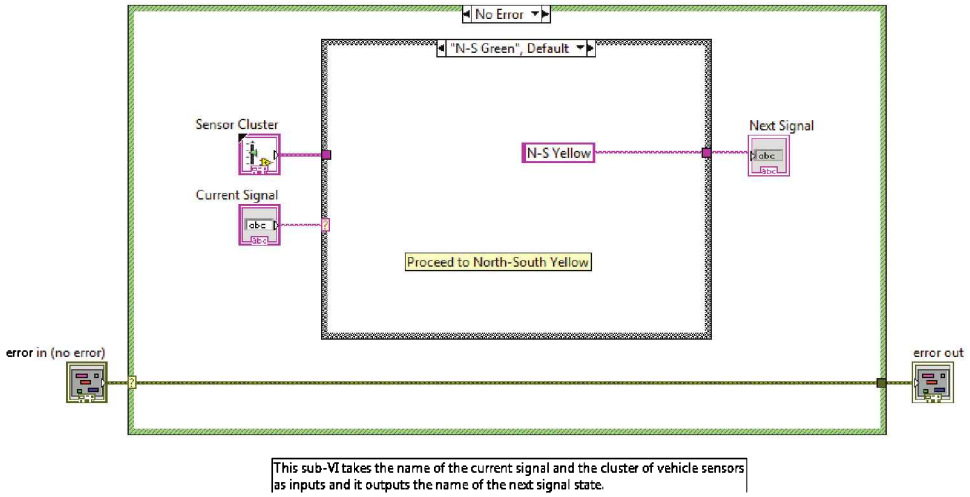


Figure N.12: The *N-S Green* of the `Next Traffic State String` sub-VI

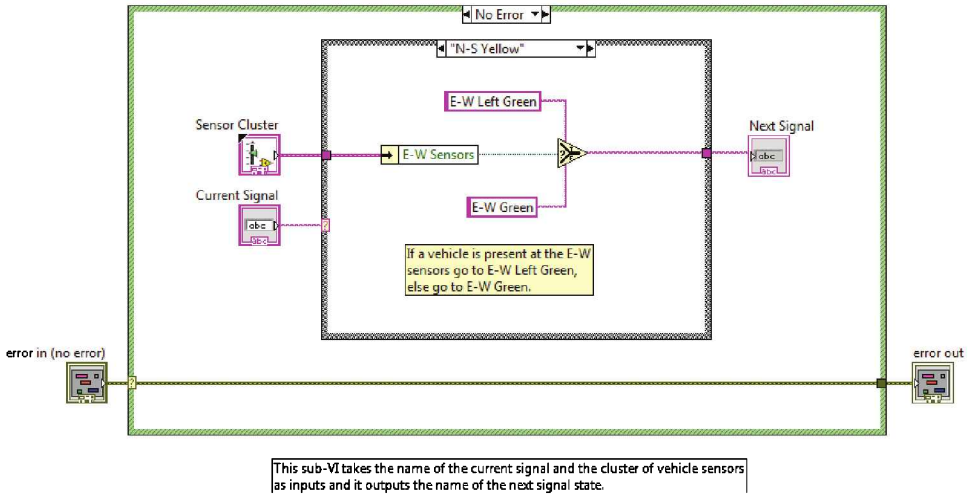


Figure N.13: The *N-S Yellow* of the `Next Traffic State String` sub-VI

Exercise 8.5

The overall structure and *Initialize* state of the traffic light controller with arguments is shown in Figure N.14. The state enumerated constant and a string are bundled into a cluster and this cluster is placed on the state shift register of the main loop. If the STOP button has not been pressed and no error is detected, the cluster is passed through the guard clause and the state enumerated constant is unbundled and passed to the selector of the main case structure. The *Initialize* state then resets the front panel sensors, manually resets the auto-reset timer, and places the *Signal* state with the `N-S`

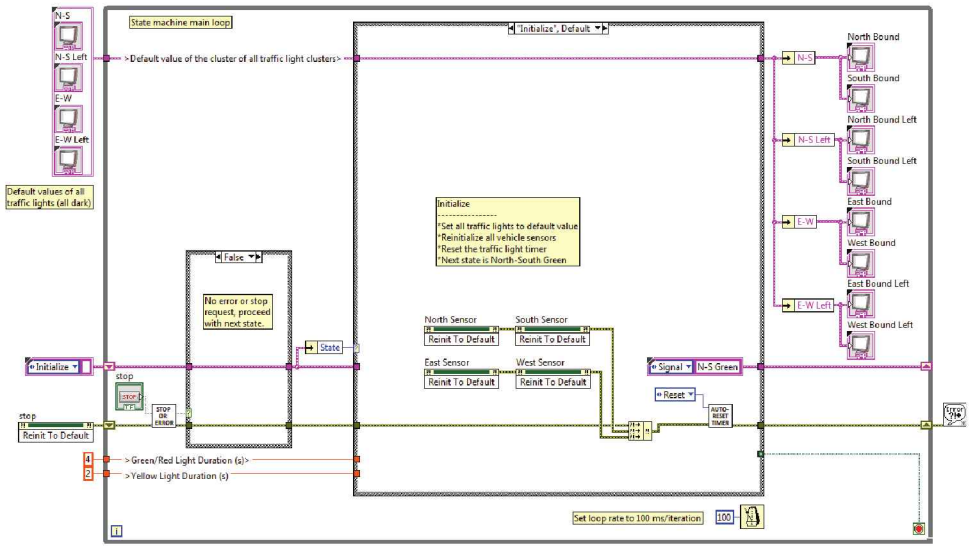


Figure N.14: The *Initialize* state of the traffic light controller with arguments

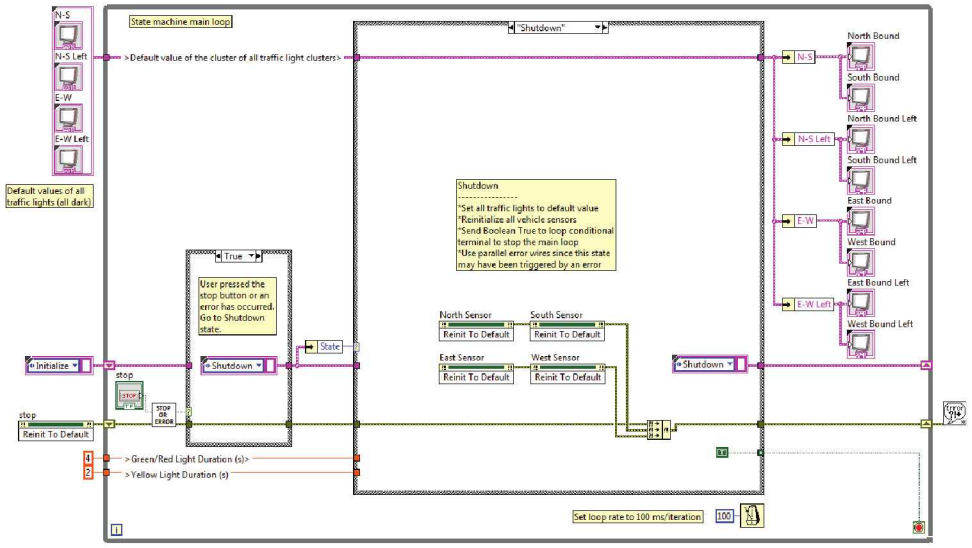


Figure N.15: The *Shutdown* state of the traffic light controller with arguments

`Green` string argument on the shift register. If the user has pressed the STOP button or if an error is detected, the guard clause sends the controller directly to the *Shutdown* state as seen in Figure N.15.

The controller spends most of its time in the *Signal* state as seen in Figure N.16. The `Signal Lights and Time Target` sub-VI with the icon shown in Figure N.17 takes the state and signal name cluster as an input and returns the correct signal pattern

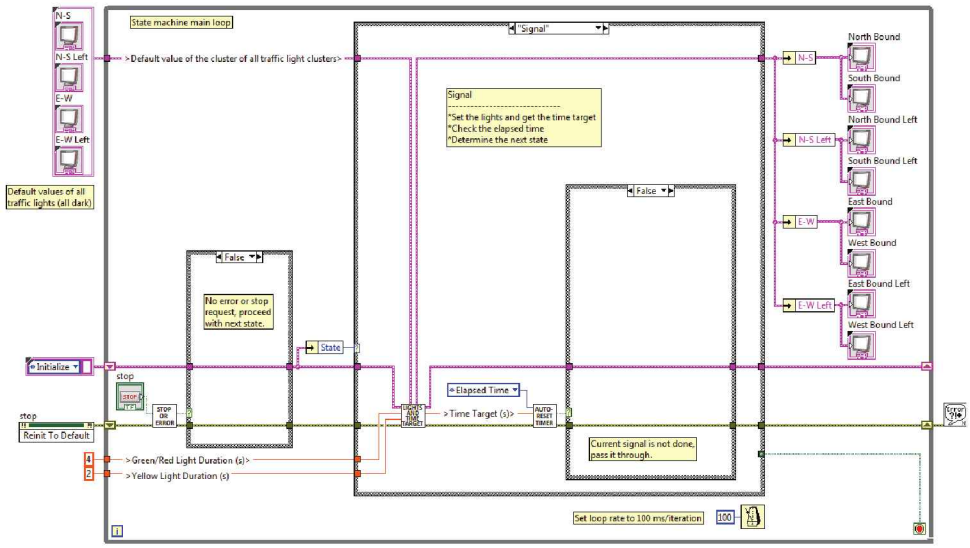


Figure N.16: The *Signal* state of the traffic light controller with arguments, signal not done

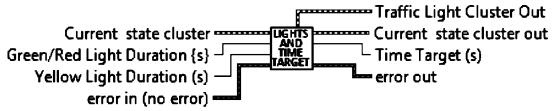


Figure N.17: The icon of the `Signal Lights and Time Target` sub-VI

and signal duration. It does this using the `Signal Map` sub-VI developed in Exercise 8.4. The block diagram of the `Signal Lights and Time Target` is shown in Figure N.18. This sub-VI is a wrapper for the `Signal Map` sub-VI. The signal name is unbundled from the input cluster and is passed to the `Signal Map` sub-VI along with the light duration inputs.

If the current signal is not done, the cluster of the state constant and the signal name is returned to the shift register. If the current signal is done, the controller reacts as shown in Figure N.19. The vehicle sensor cluster and the current state cluster are passed to the `Get Next Signal Cluster` sub-VI whose icon is shown in Figure N.20. This sub-VI, with the block diagram shown in Figure N.21, is a wrapper of the `Next Traffic State String` developed in Exercise 8.4. The signal name is unbundled from the state cluster and is passed to the `Next Traffic State String` along with the vehicle sensor cluster. The next signal name is then bundled into state cluster, overwriting the name of the old traffic signal. This concludes the traffic light controller with arguments.

Exercise 8.6

The *Initialize* state of the security system controller with arguments is shown in Figure N.22. A type-defined cluster consisting of the state enumerated constant and

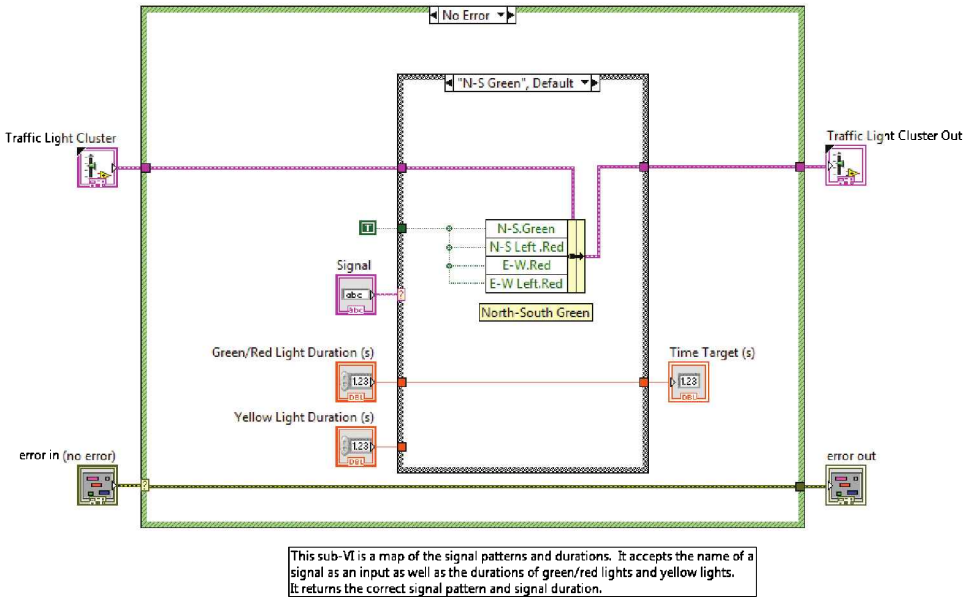


Figure N.18: The block diagram of the `Signal Lights and Time Target` sub-VI

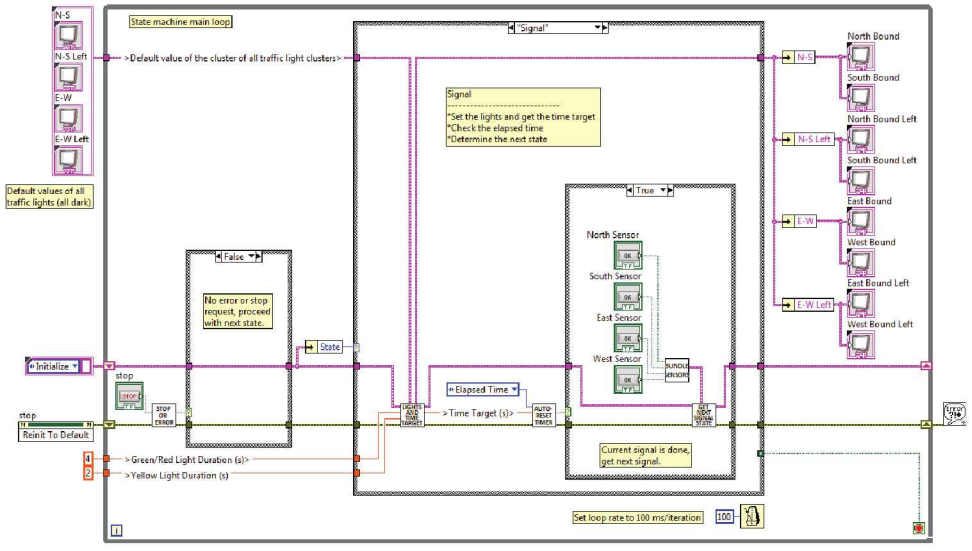


Figure N.19: The *Signal* state of the traffic light controller with arguments, signal done

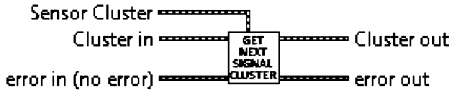


Figure N.20: The icon of the `Get Next Signal Cluster` sub-VI

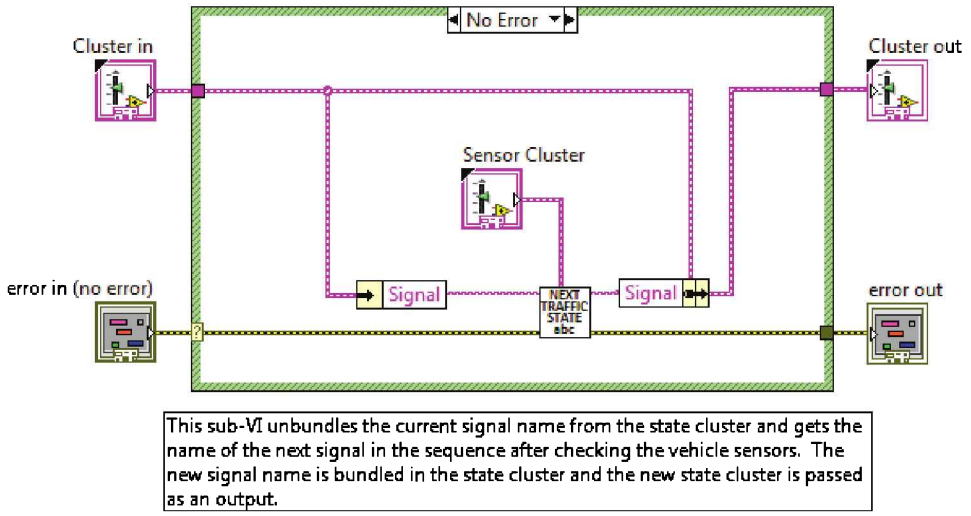


Figure N.21: The block diagram of the `Get Next Signal Cluster` sub-VI

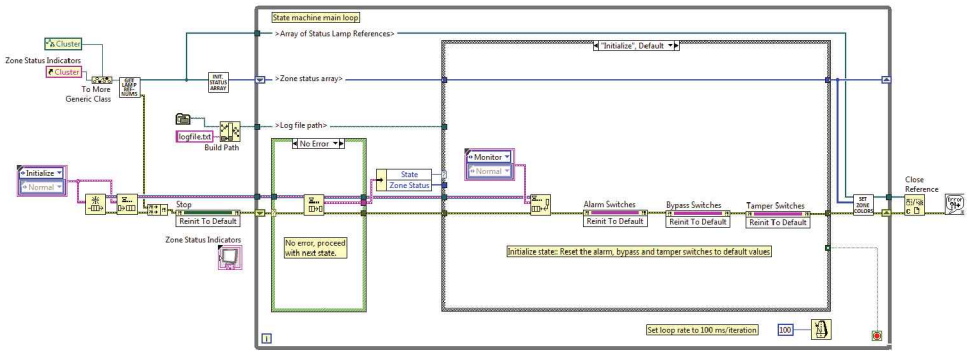


Figure N.22: The *Initialize* state of the security system controller with arguments

the zone indicator array is used to create the queue. The zone indicator cluster is shown without its index display to reduce its screen footprint. The zone status array element is grayed-out to indicate that this array currently has zero-length. The data type of the element has been defined by the array in the cluster has no elements as yet.

If no error is present, the guard clause dequeues the next cluster. The **`Unbundle By Name`** function is then used between the guard clause and the main case structure to unbundle the state enumerated constant and the zone status array. The state enumerated constant is passed to the selector of the main case structure and the zone status cluster is passed as an input to the main case structure. In the *Initialize* state, the front panel switches are reset, and a cluster containing the *Monitor* state and the zone status array are placed on the queue.

If an error is detected, the guard clause bypasses the queue entirely, and a cluster containing the *Shutdown* state is passed to the **`Unbundle By Name`** function, as seen in Figure N.23.

The security system spends most of its time in the *Monitor* state, which contains the event structure monitoring the user interface. The *Shutdown* state is enqueued if the STOP button is pressed while the controller is in this state, as seen in Figure N.24.

The event case handling the value change of the front panel switches is shown in Figure N.25. The old zone states are pulled from the data highway and are bundled into a cluster with the *Update Log* state. The cluster is then placed on the queue. The new zone status array is calculated by the `Get Status Of All Zones` sub-VI and is then placed on the data highway.

The *Update Log* state is shown in Figure N.26. The `Update Log` sub-VI pulls the new zone status from the data highway. The old zone status is passed to it as an argument. The log file is updated if any of the zones have changed status. The *Monitor* state is then placed back on the queue. This concludes the security system controller with arguments.

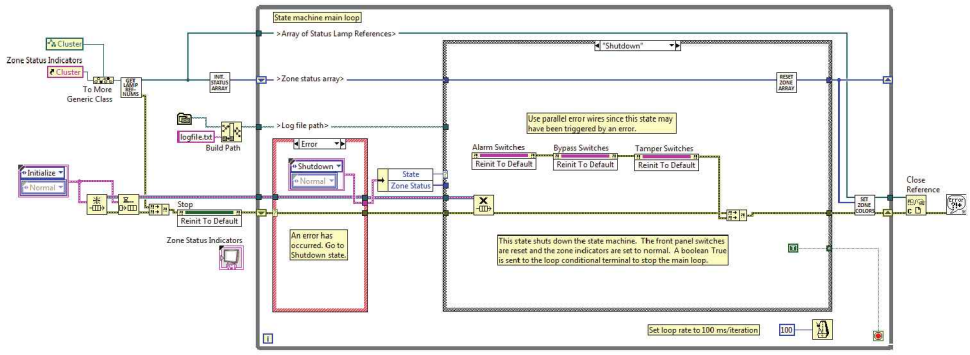


Figure N.23: The *Shutdown* state of the security system controller with arguments

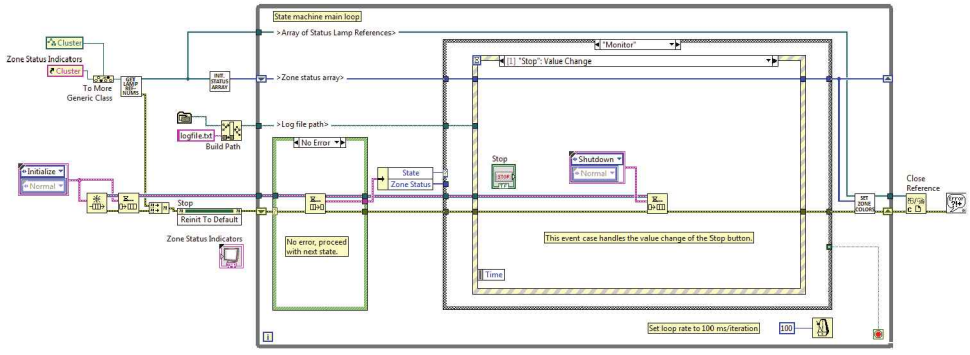


Figure N.24: The event case handling the value change of the STOP button of the security system controller with arguments

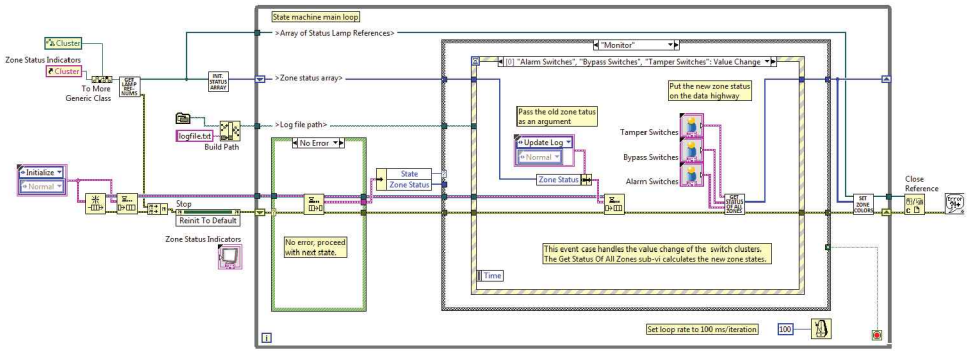


Figure N.25: The event case handling the value change of the switches of the security system controller with arguments

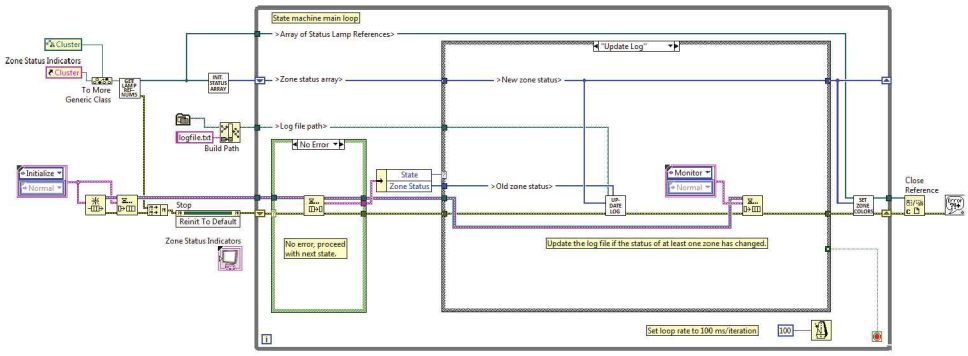


Figure N.26: The *Update Log* state of the security system controller with arguments

Exercise 8.7

The security system controller with reversed arguments is identical to the security system controller developed in Exercise 8.6 except for differences in two states. The event case handling the value change of the front panel switches is shown in Figure N.27. The new zone status array is bundled into the queue cluster and is passed to the *Update Log* state. The old zone status array remains on the data highway.

The *Update Log* state is shown in Figure N.28. The `Update Log` sub-VI pulls the old zone status array from the data highway and is passed the new zone status array as an argument. The log file is updated if at least one zone has changed status. The old zone status on the data highway is then overwritten with the new zone status.

Exercise 8.8

There is one slight functional difference between the security system controllers developed in Exercise 8.6 and Exercise 8.7. Examine the event case handling the value changes of the switches shown in Figure N.25. The new zone status is written to the

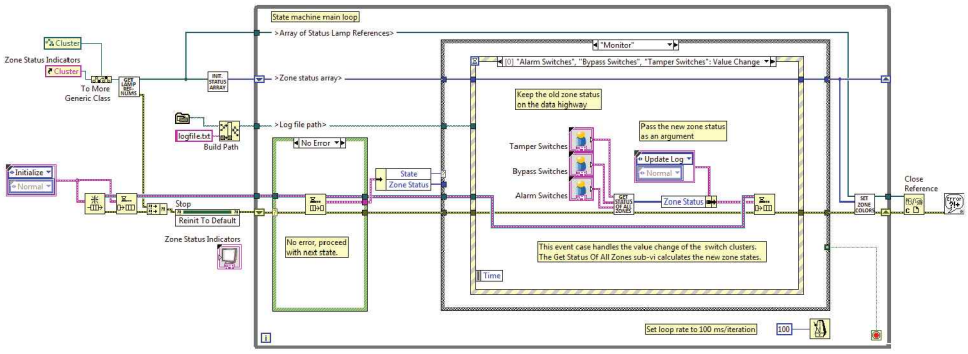


Figure N.27: The event case handling the value change of the switches of the security system controller with arguments reversed

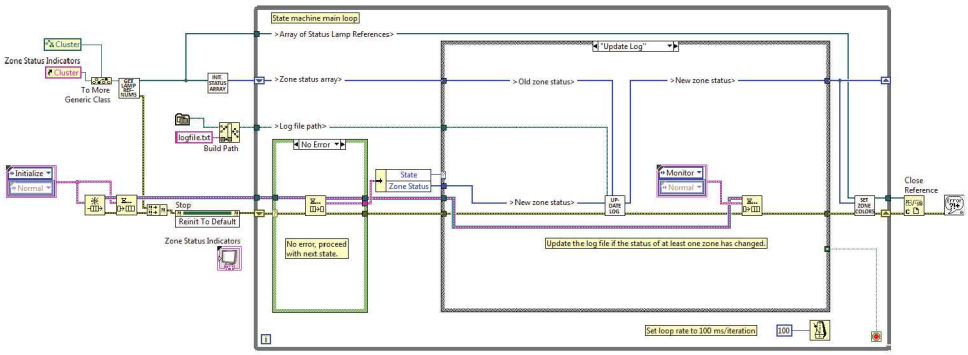


Figure N.28: The *Update Log* state of the security system controller with arguments reversed

data highway in this state, so the zone status indicators will be updated with the new zone status at the end of that loop iteration. Now examine Figure N.27. Since the old zone status array is kept on the data highway, the old zone status will be displayed at the end of the loop iteration that processes the switches value change. The new zone status is not displayed until the next loop iteration. Because of this, it is preferable to use the arguments scheme proposed in Exercise 8.6.

But in reality, there is no compelling reason to use arguments in this case. They do not increase the readability, scalability or maintainability of the security controller compared to the version developed in Exercise 5.2. But we will see that there is a compelling reason for using arguments when we explore the producer-consumer parallel loop architecture in Chapter 27. In this scenario, we will have one loop constantly monitoring the user interface. This loop will generate the new zone status array when a front panel switch is pressed. A second, parallel loop will process the new zone status array and will update the log file if necessary while the first loop continues to monitor the user interface. In this situation, a queue can be used to communicate between the two loops and the new zone status can be passed from the user interface

loop to the processing loop as an argument in the queue cluster. In this case, the argument scheme proposed by Exercise 8.7 is more useful, since it is the new zone status array that needs to be passed from one loop to the other via the queue.

Exercise 8.9

The queued-string boiler controller is very similar to the queued state machine implementation as can be seen in Figure N.29. This figure shows the *Initialize* state of the boiler controller. The queue is created in the input area of the loop using a string. The queue is initialized with a multiline string containing the *Initialize* state and the *Load Sequence* state. This string is split into its individual lines and is enqueued using the `Split and Queue String` sub-VI developed in Figure 20.27. The ability to easily enqueue multiple states with a single string in the input area of the loop is an advantage of the string-based approach. The queue is then dequeued in the lobby of the state machine by the guard clause, assuming that the STOP button has not been pressed and no error has been detected.

The *Initialize* state resets the front panel controls and ensures that the PILOT button is not blinking. The *Load Sequence* state shown in Figure N.30 enqueues the entire boiler sequence using a multiline string constant and the `Split and Queue String` sub-VI. Notice that the string constant ends with the same two states enqueued in the input area of the loop, the *Initialize* state, and the *Load Sequence* state. This allows the user to restart the boiler after a normal shutdown.

The string constant is much easier to create, read and edit than an array of enumerated constants. The disadvantage of using strings is that a default state needs to be created to catch spelling errors in the state names. The default state is shown in Figure N.31. This state generates a custom error if the state string is not recognized.

The *Application Run* state is shown in Figure N.32. If the state is not done, the current state is placed back on the front of the queue using the **`Enqueue Element At Opposite End`** function. If the state is finished, then the current state is allowed to expire and a new state is dequeued in the guard clause at the next loop iteration as

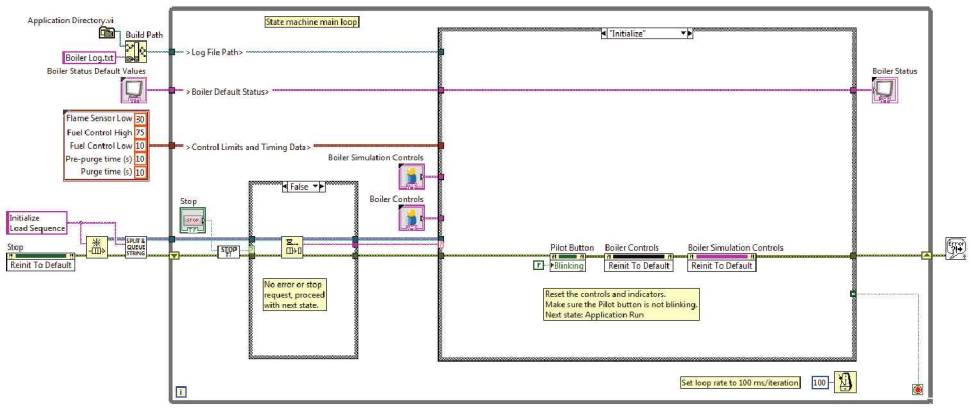


Figure N.29: The *Initialize* state of the queued-string boiler controller

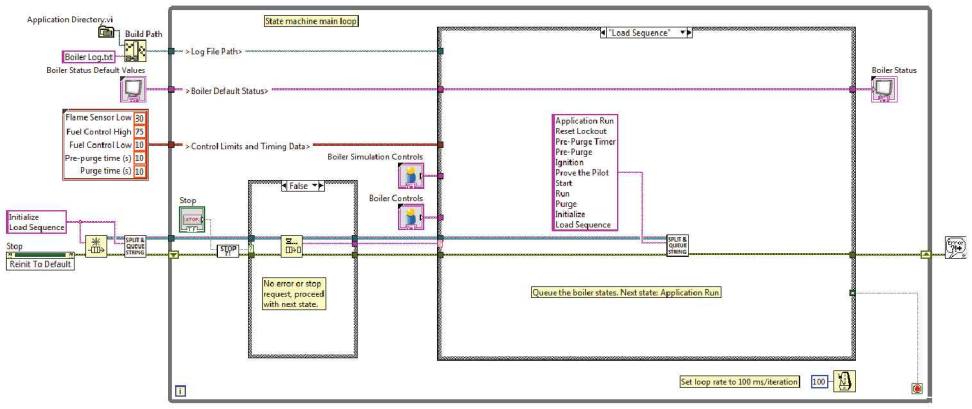


Figure N.30: The *Load Sequence* state of the queued-string boiler controller

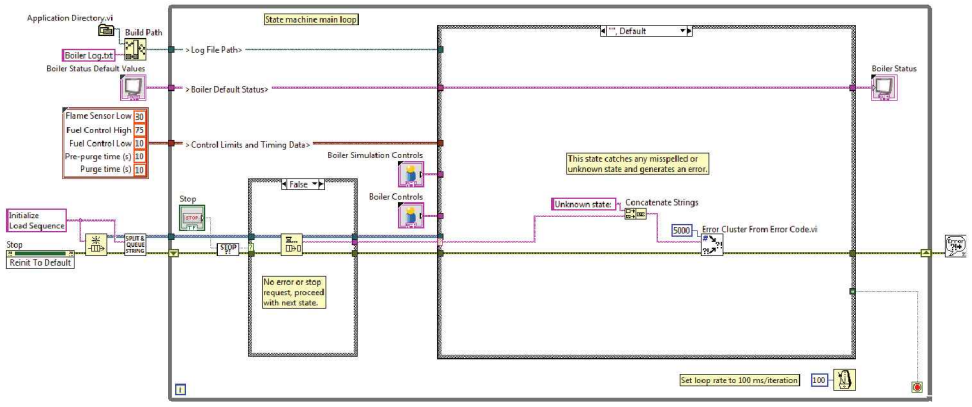


Figure N.31: The *Default* state of the queued-string boiler controller

seen in Figure N.33. The other boiler process states are similar and will not be shown here.

If an error is detected or the user presses the STOP button, the guard clause bypasses the queue and sends the state machine directly to the *Shutdown* state as shown in Figure N.34. The queue is released using the **`Release Queue`** function. This function executes even if an error is present at its error input terminal so it does not need to be placed on a parallel error wire. This concludes the queued version of the boiler controller.

Notice that queues do not add functionality in this case. The series of states in the boiler controller is fixed, so the flexibiity added by queues is not needed. The queue does decouple each state from the other states, though. The individual states do not need to know what state comes next. Also, readability is enhanced because the entire sequence of states is visible in the *Initialize* state. This makes it easier for the reader to visualize the entire boiler process.

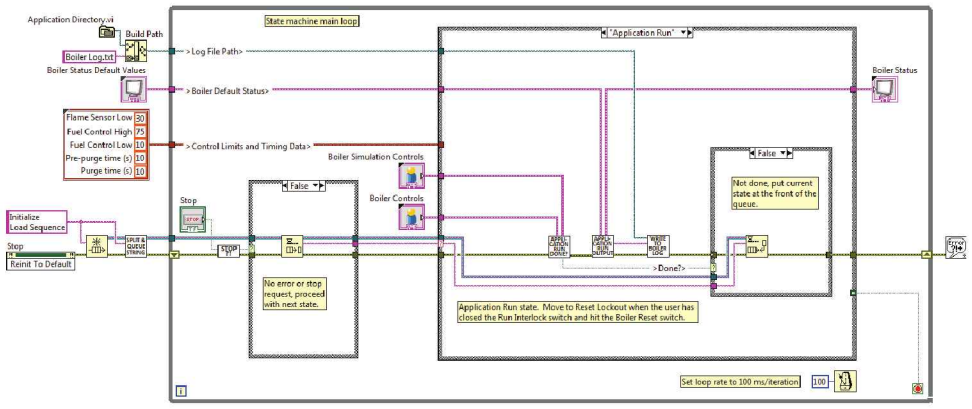


Figure N.32: The *Application Run* state of the queued-string boiler controller, state not done

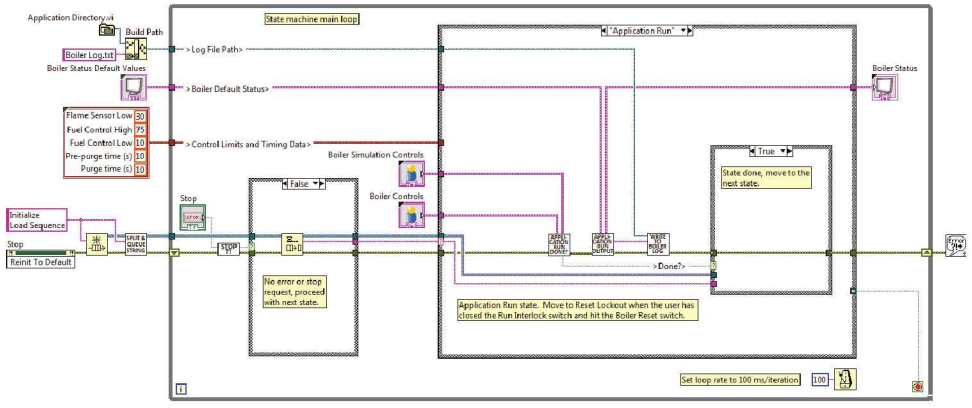


Figure N.33: The *Application Run* state of the queued-string boiler controller, state done

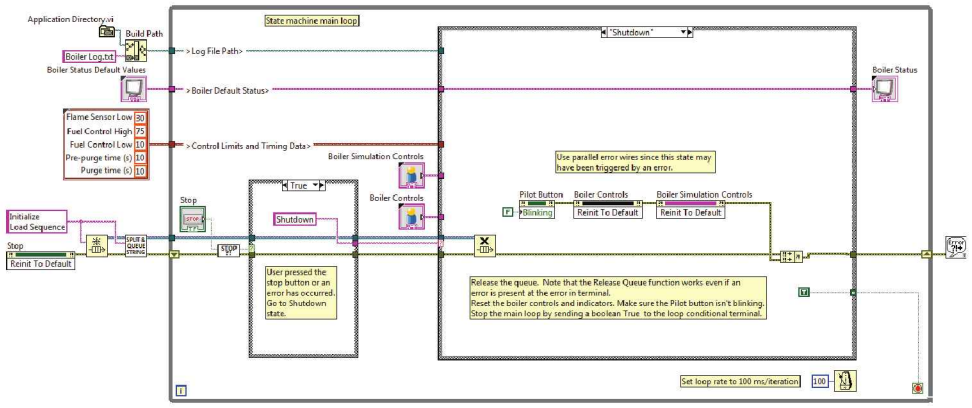


Figure N.34: The *Shutdown* state of the queued-string boiler controller

Exercise 8.10

The block diagram of one possible implementation of the `Parse State and Argument Array` sub-VI is shown in Figure N.35. The input string is placed on a shift register of a **`While`** loop. The string is parsed using the `Parse State and Argument` sub-VI, and the argument string minus the state is put back on the shift register. The argument string is parsed over and over with the "state" portion of each string being collected on an auto-indexing output tunnel of the loop. Each loop iteration the argument string is checked to see if it is empty. If it is empty, the loop ends. The first element of the output array is then deleted using the **`Delete From Array`** function. This function returns the deleted element, which is the state portion of the original string. It also returns the rest of the array, which is the array of arguments from the original string.

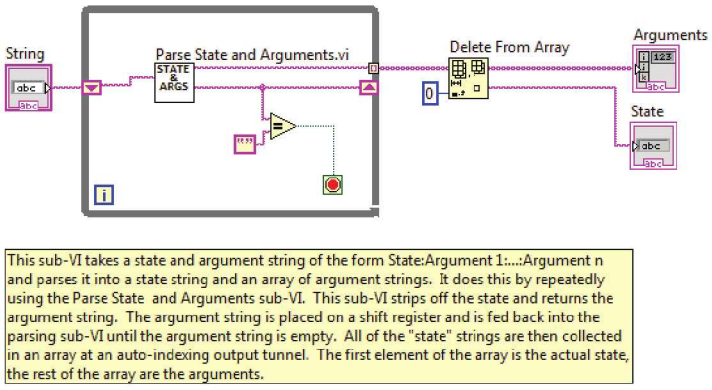


Figure N.35: The block diagram of the `Parse State and Argument Array` sub-VI

Exercise 8.11

The block diagram of one possible implementation of the `Get DBL Arguments` is shown in Figure N.36. This sub-VI takes advantage of the fact that the **`Fract/Exp String To Number`** function is polymorphic and can convert the entire string array in one function call. Once the array has been converted to double-precision floating point numbers the first two elements are returned using the **`Index Array`** function.

Exercise 8.12

One possible implementation of the `Pre-Purge Timer Done Arguments` sub-VI is shown in Figure N.37. The argument array is generated by the `Parse State and Argument Array` sub-VI, and the arguments are converted to double-precision numbers using the `Get DBL Arguments` function. The rest of the sub-VI is similar to that of the `Pre-Purge Timer Done` sub-VI shown in Figure G.20.

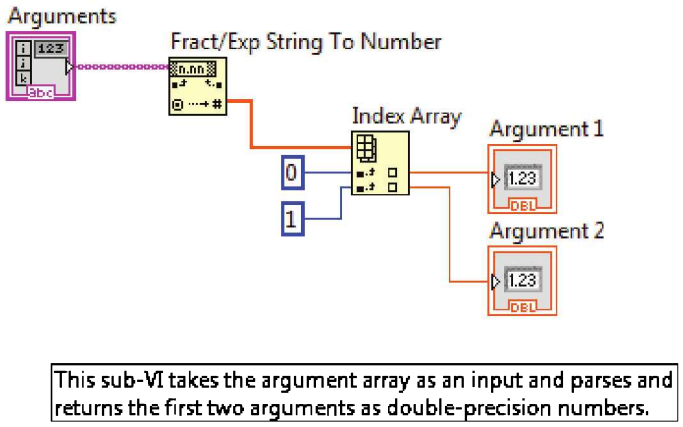


Figure N.36: The block diagram of the `Get DBL Arguments` sub-VI

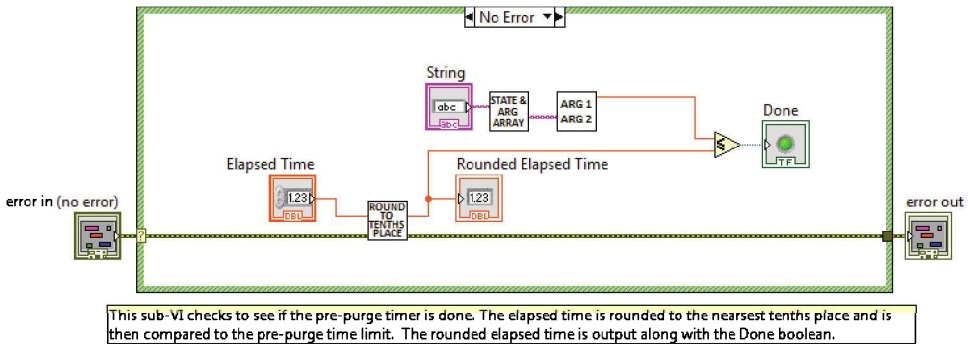


Figure N.37: The block diagram of the `Pre-Purge Timer Done Arguments` sub-VI

Exercise 8.13

One possible implementation of the `Ignition Done Arguments` sub-VI is shown in Figure N.38. The argument array is generated by the `Parse State and Argument Array` sub-VI, and the arguments are converted to double-precision numbers using the `Get DBL Arguments` function. The rest of the sub-VI is similar to that of the `Ignition Done` sub-VI shown in Figure G.24.

Exercise 8.14

One possible implementation of the `Start Done Arguments` sub-VI is shown in Figure N.39. The argument array is generated by the `Parse State and Argument Array` sub-VI, and the arguments are converted to double-precision numbers using the `Get DBL Arguments` function. The **`Max & Min`** function is used to select the high and low fuel control values from the two arguments. These limits are then passed

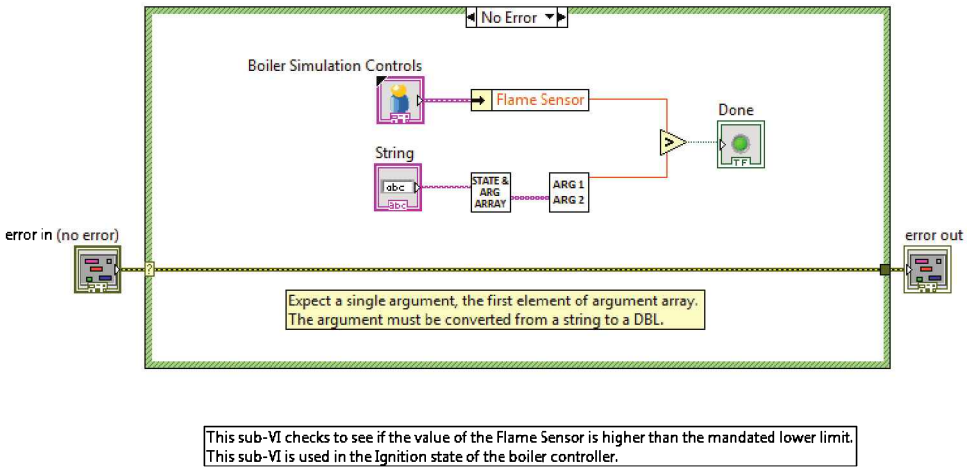


Figure N.38: The block diagram of the `Ignition Done Arguments` sub-VI

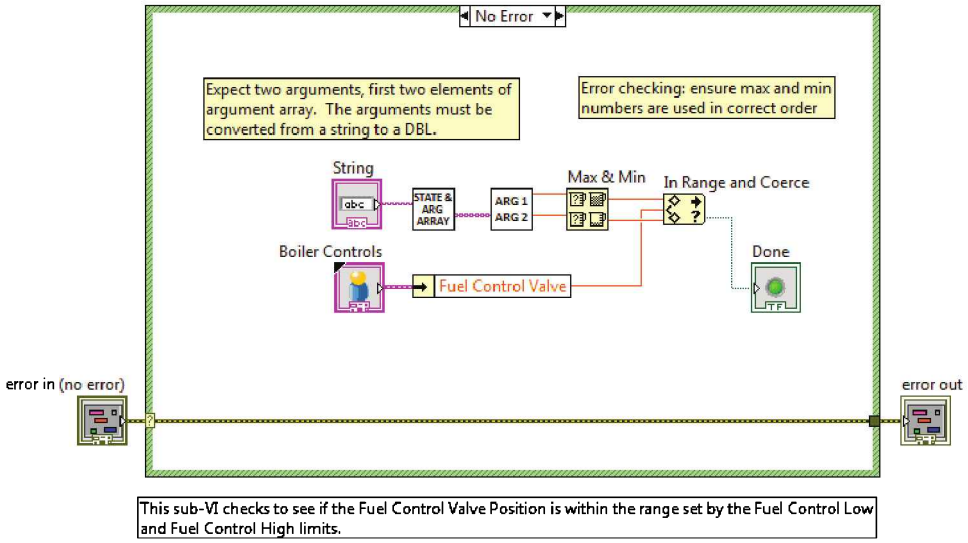


Figure N.39: The block diagram of the `Start Done Arguments` sub-VI

to the **`In Range and Coerce`** function to determine whether the FUEL CONTROL VALVE is within the specified range.

Exercise 8.15

One possible implementation of the `Run Done Arguments` sub-VI is shown in Figure N.40. The argument array is generated by the `Parse State and Argument Array` sub-VI, and the arguments are converted to double-precision numbers

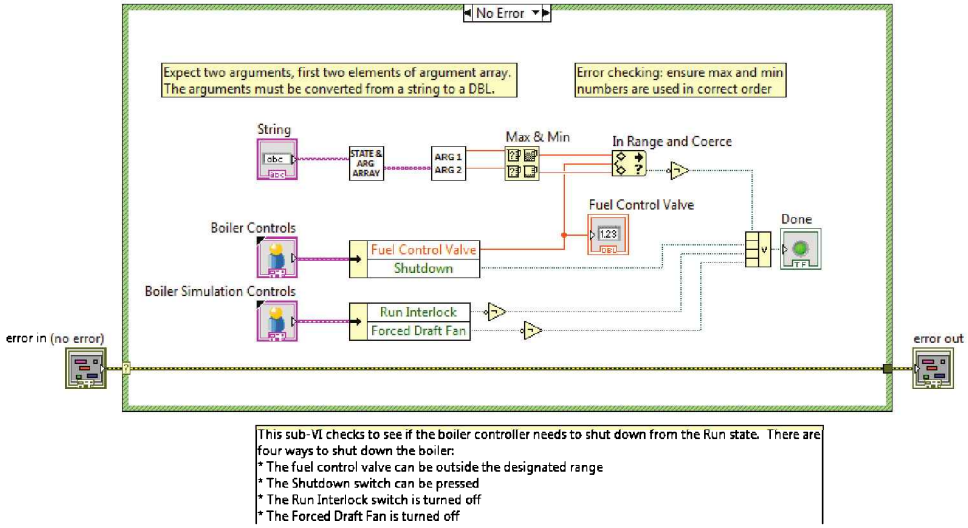


Figure N.40: The block diagram of the `Run Done Arguments` sub-VI

using the `Get DBL Arguments` function. The **`Max & Min`** function is used to select the high and low fuel control values from the two arguments. These limits are then passed to the **`In Range and Coerce`** function to determine whether the FUEL CONTROL VALVE is within the specified range.

Exercise 8.16

One possible implementation of the `Purge Done Arguments` sub-VI is shown in Figure N.41. The argument array is generated by the `Parse State and Argument Array` sub-VI, and the arguments are converted to double-precision numbers using the `Get DBL Arguments` function. The rest of the sub-VI is similar to that of the `Purge Done` sub-VI shown in Figure G.32.

Exercise 8.17

The *Load Sequence* state of the boiler controller with arguments is shown in Figure N.42. The LIMITS cluster has been eliminated and its contents have now been distributed in the program string to the states that need them. The rest of the state machine structure remains the same except for the addition of the `Parse State` sub-VI between the guard clause and the main case structure. In the queued-string version of the boiler controller, it was not necessary to parse the state from the queued string since the queued string had no arguments, but it is necessary here.

The *Pre-Purge Timer* state is shown in Figure N.43. The `Pre-Purge Timer Done Arguments` state pulls the arguments from the queued string instead of the LIMITS cluster. The rest of the modified states are all similar and will not be shown here. This concludes the development of the boiler controller with arguments.

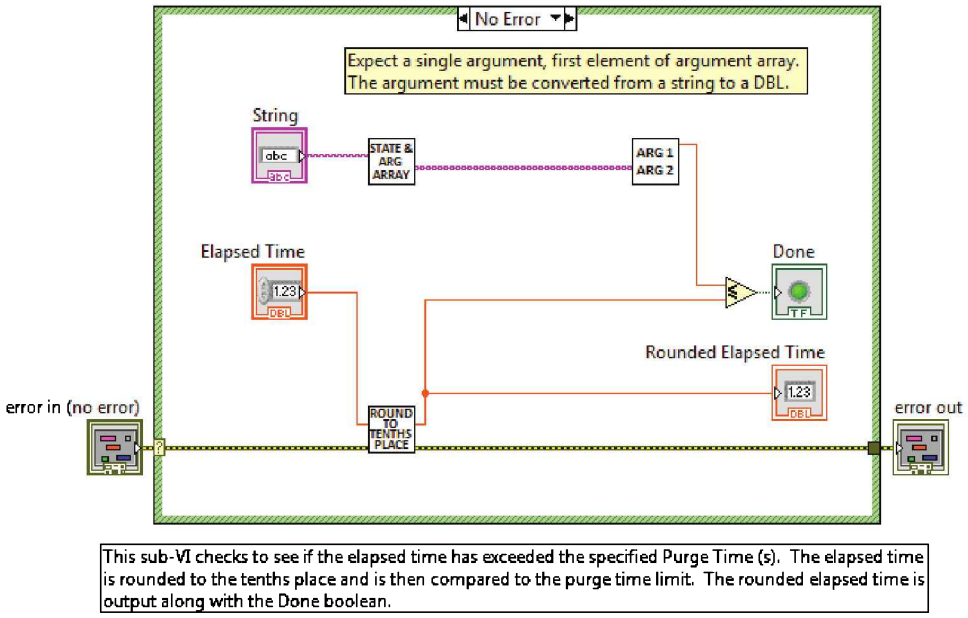


Figure N.41: The block diagram of the `Purge Done Arguments` sub-VI

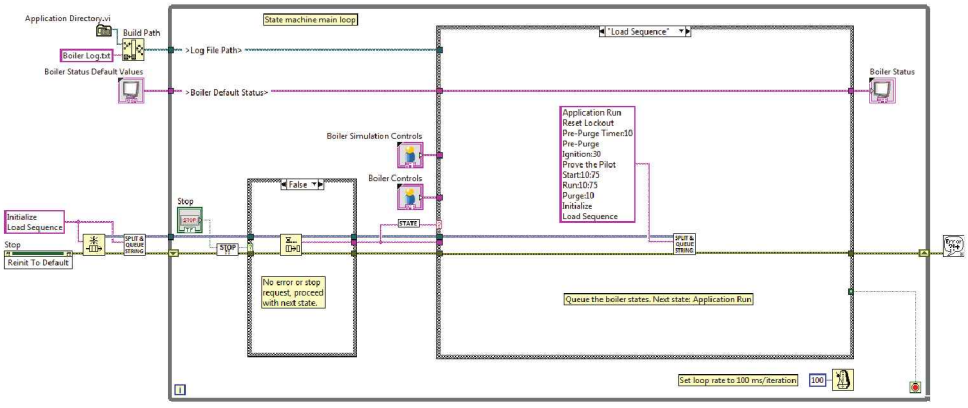


Figure N.42: The *Load Sequence* state of the boiler controller with arguments

Exercise 8.18

One possible implementation of the `State And Time Target` sub-VI is shown in Figure N.44. The **`Match Pattern`** function is used to split the input string at the first colon found. The portion of the string preceding the colon is the state name. The portion following the colon is a string containing the time target. This string is converted to a double-precision floating point number using the **`Fract/Exp String To Number`** function.

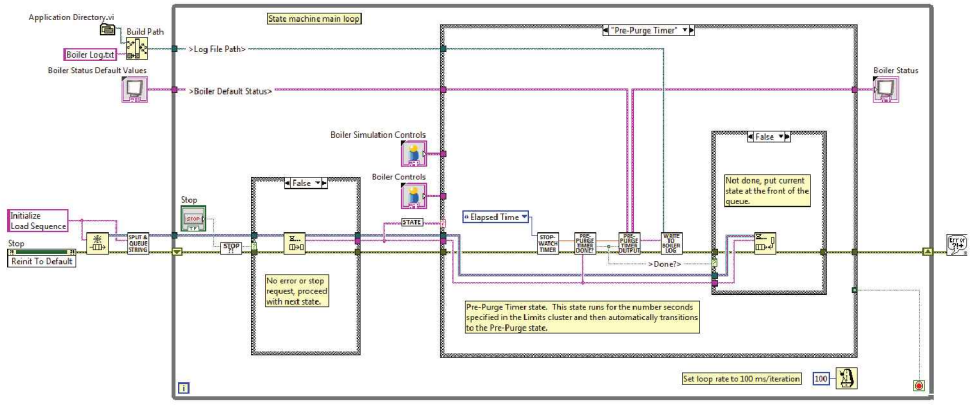


Figure N.43: The *Pre-Purge Timer* state of the boiler controller with arguments

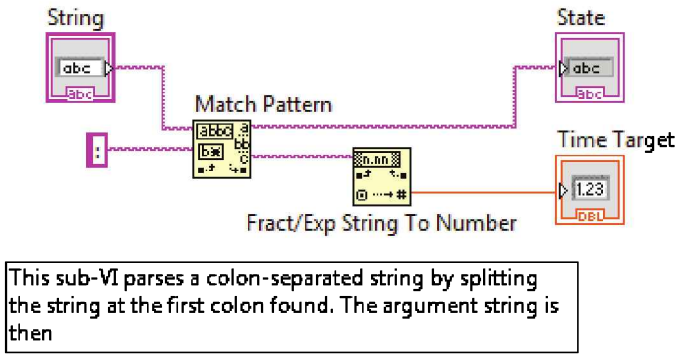


Figure N.44: The block diagram of the `State And Time Target` sub-VI

Exercise 8.19

One possible implementation of the `Build Option String Array` is shown in Figure N.45. The `String to Array` shown in Figure 20.26 is used to convert the multiline string to a string array. The rest of the logic of this sub-VI is identical to that of the `Build Wash Array` sub-VI developed in Figure 5.6 with the exception that a string array is output rather than an array of enumerated constants.

Exercise 8.20

The main structure and the *Initialize* state of the string-based Car Wash 2 controller with time target arguments is shown in Figure N.46. The queue is formed in the input area of the loop using a string element rather than an enumerated constant. The `State And Time Target` sub-VI is placed between the guard clause and the main case structure to parse the strings as they are dequeued. The state string is wired to

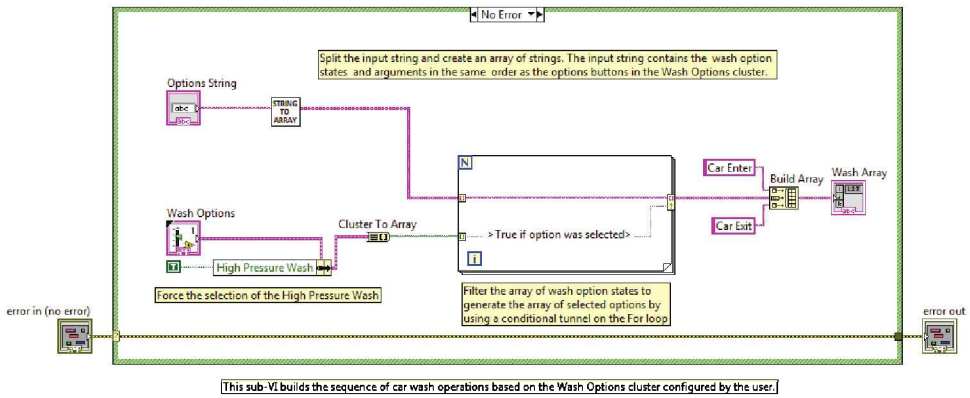


Figure N.45: The block diagram of the `Build Option String Array` sub-VI

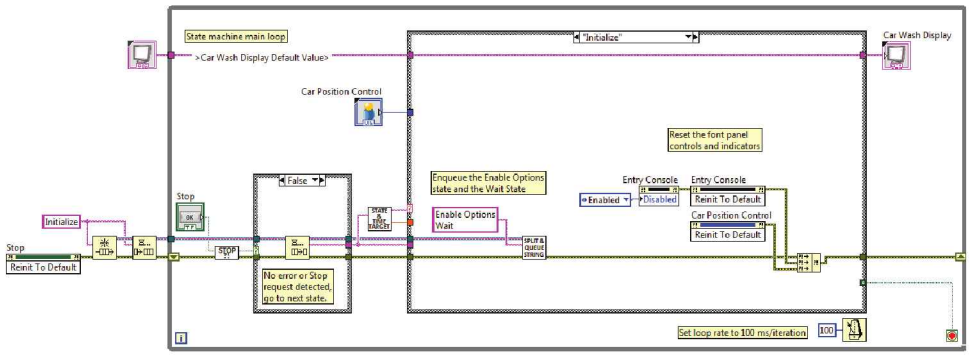


Figure N.46: The *Initialize* state of the string-based Car Wash 2 controller with time target arguments

the selector of the case structure, and the time target argument is wired as an input to the case structure.

Figure N.47 shows how the time target argument is used by the *Under Body Wash* state. The hard-coded time target has been removed from this state. Instead, the time target argument is wired to the `Car Wash 2 Timer` sub-VI.

Returning to Figure N.46, the *Initialize* state uses the `Split and Queue String` sub-VI shown in Figure 20.27 to queue up all of the states needed to take the controller to the *Queue Wash Sequence* state. This state is shown in Figure N.48. This state queues up the *Car Enter* state and all of the selected option states and the *Car Exit* state using the `Build Option String Array` sub-VI developed in Exercise 8.19. The *Car Exit* state returns to the *Initialize* state when the vehicle exits the car wash, as shown in Figure N.49.

When the user presses the STOP button or if an error is detected, the guard clause bypasses the queue and sends the controller directly to the *Shutdown* state as shown in Figure N.50. Since this is a string-based state machine, a default clause needs to be

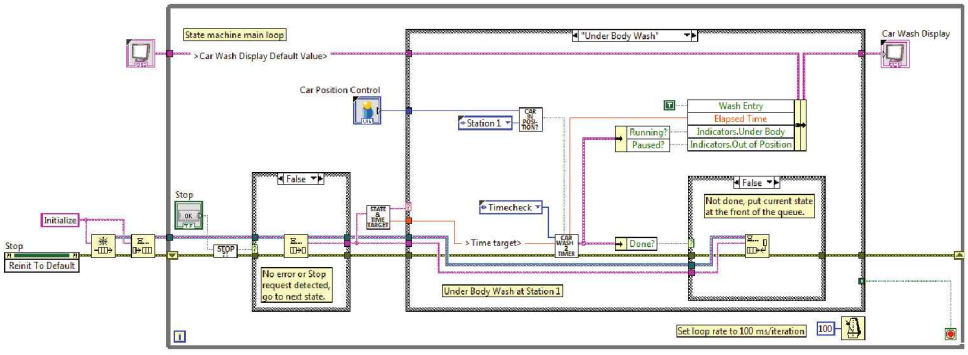


Figure N.47: The *Under Body Wash* state of the string-based Car Wash 2 controller with time target arguments

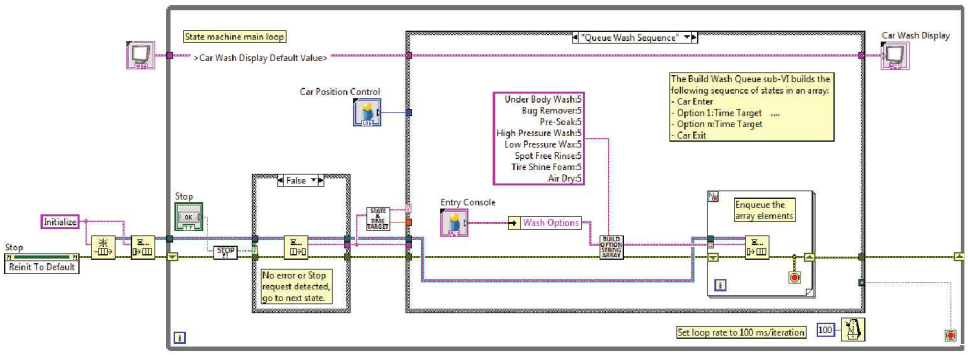


Figure N.48: The *Queue Wash Sequence* state of the string-based Car Wash 2 controller with time target arguments

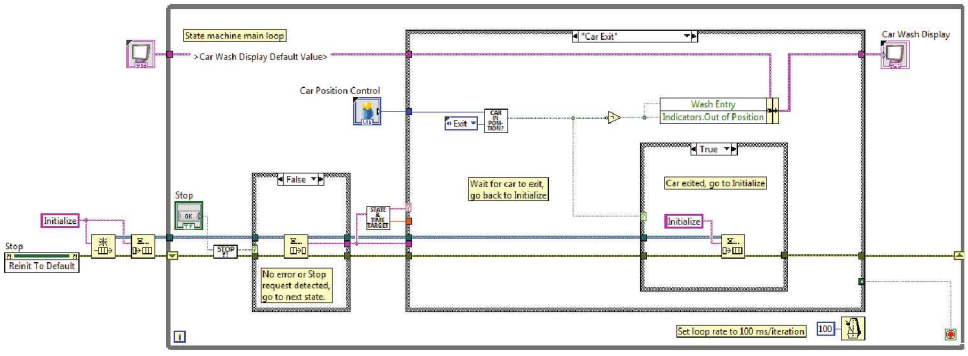


Figure N.49: The *Car Exit* state of the string-based Car Wash 2 controller with time target arguments where the vehicle has exited

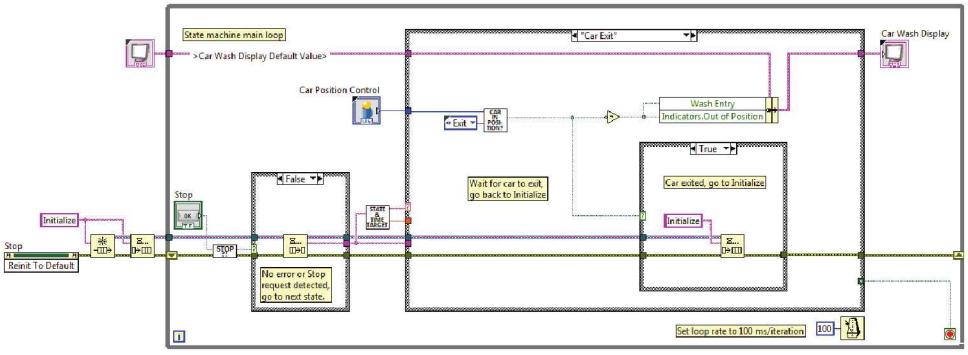


Figure N.50: The *Shutdown* state of the string-based Car Wash 2 controller with time target arguments

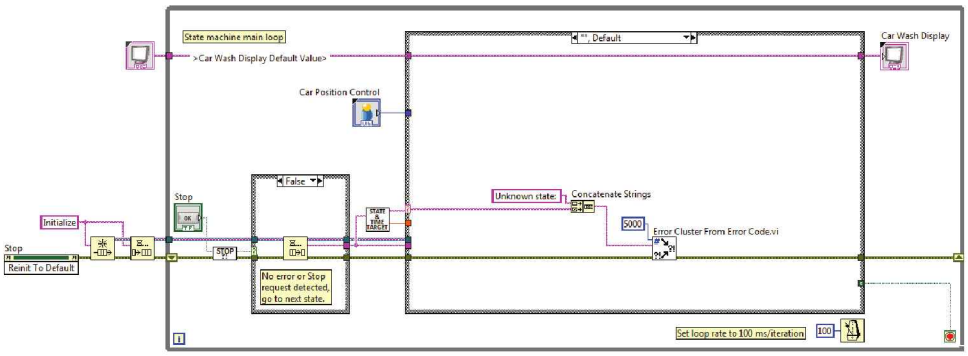


Figure N.51: The *Default* state of the string-based Car Wash 2 controller with time target arguments

provided to catch unknown states or spelling errors. This state is shown in Figure N.51. This concludes the development of the string-based Car Wash 2 controller with time target arguments.

Exercise 8.21

One possible implementation of the `Get Option Station` sub-VI is shown in Figure N.52. This sub-VI uses a case structure to map the input string to a specific value of the car wash position enumerated constant. The implementation of the `Set Option Display` sub-VI shown in Figure N.53 is similar. The output cluster is mapped to the input string using a case structure. The inputs are bundled into a default value cluster constant of the type-defined CAR WASH DISPLAY cluster. The cluster is shown as an icon to improve readability.

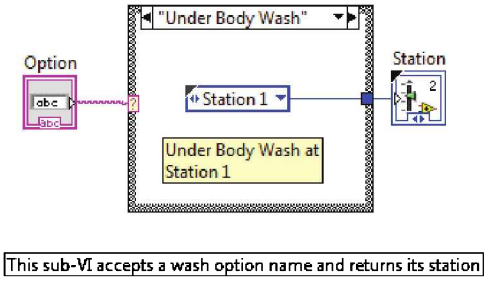


Figure N.52: The block diagram of the `Get Option Station` sub-VI

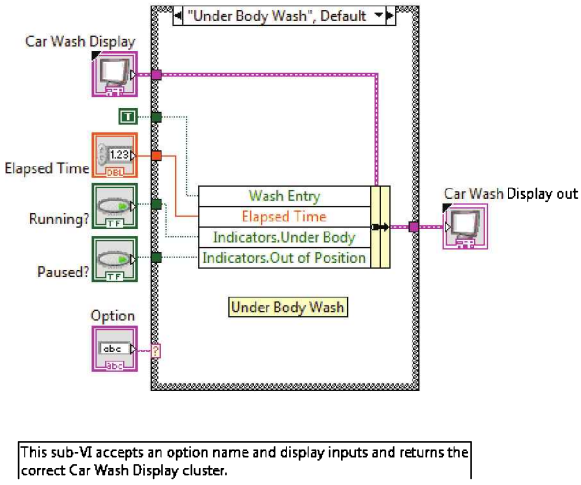


Figure N.53: The block diagram of the `Set Option Display` sub-VI

Exercise 8.22

An implementation of the `Run Car Wash Option` sub-VI modeled closely on Figure 22.17 is shown in Figure N.54. The option name and time target are parsed from the queued string using the `Parse Name and Time Target` sub-VI. The option name is passed to the `Get Option Station` sub-VI developed in Exercise 8.21. The resulting station is then passed to the `Car In Position` sub-VI to see if the car is at the proper station for the named option. The position switch output of the `Car In Position` sub-VI is then passed to the `Car Wash 2 Timer` sub-VI along with the time target for a timecheck. The timer outputs the DONE? Boolean. It also outputs the RUNNING? and PAUSED? Booleans which are passed to the `Set Option Display`

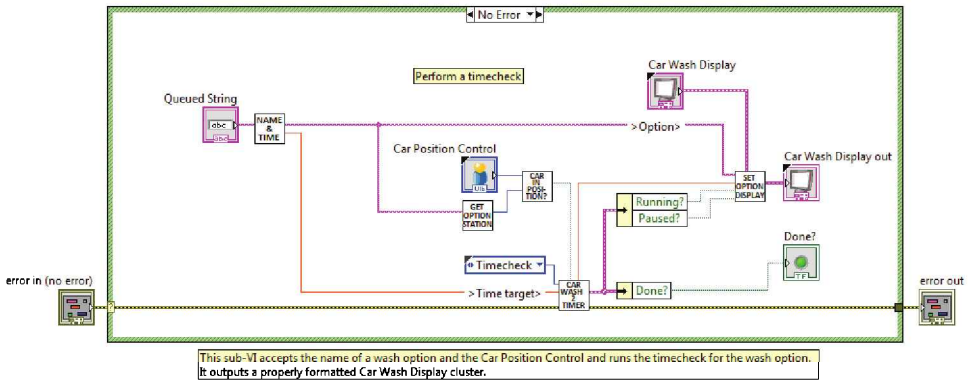


Figure N.54: The block diagram of the `Run Car Wash Option` sub-VI

sub-VI also developed in Exercise 8.21. This sub-VI outputs the cluster that will be sent to the CAR WASH DISPLAY in the main application.

Exercise 8.23

The overall structure of the queued-string Car Wash 2 controller with option arguments is shown in Figure N.55. The general structure of the state machine is almost identical to that shown in Figure N.48 except now only the state is parsed between the guard clause and the main case structure using the `Parse State` sub-VI. The *Queue Wash Sequence* state in Figure N.55 is also very similar to the one shown in Figure N.48, except now the lines of the string constant are now in the `State:Name:Time Target` format.

The entirely new *Option* state is shown in Figure N.56. This state uses the `Run Car Wash Option` sub-VI developed in Exercise 8.22 to run the selected option. This sub-VI parses the option name and time target from the input string and runs the timecheck for the given option. It also outputs a correctly formatted cluster for the CAR

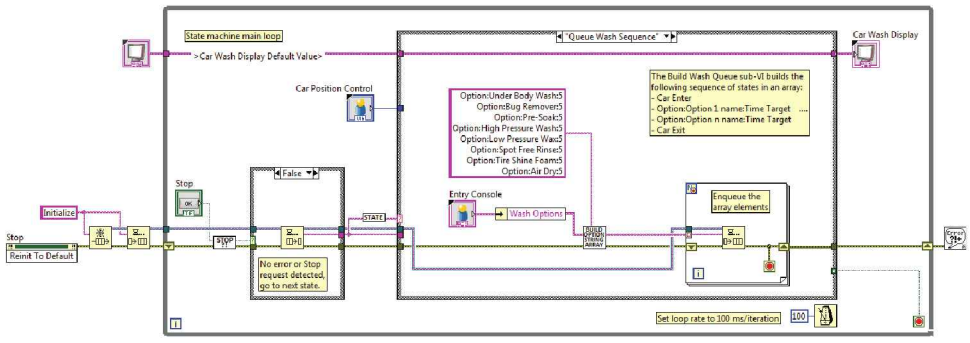


Figure N.55: The *Queue Wash Sequence* state of the Car Wash 2 controller with option arguments

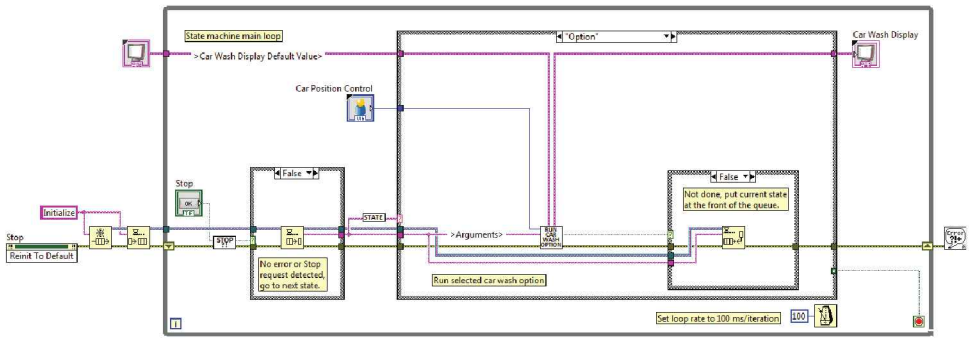


Figure N.56: The *Option* state of the Car Wash 2 controller with option arguments, option not done

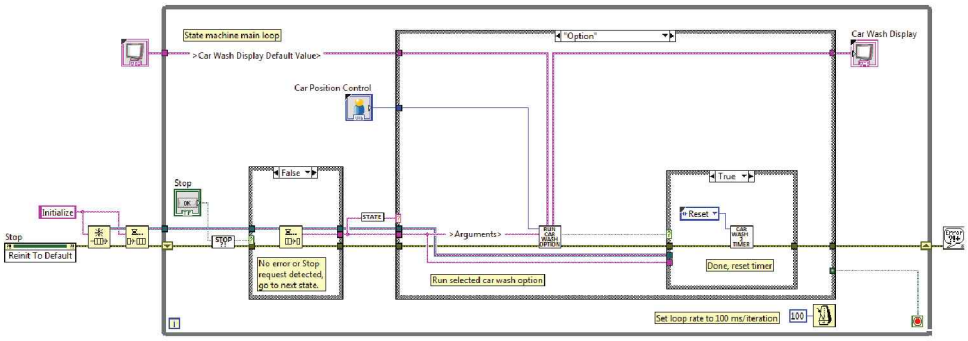


Figure N.57: The *Option* state of the Car Wash 2 controller with option arguments, option done

WASH DISPLAY indicator. If the option is not done, the queued string is replaced at the front of the queue using the **`Enqueue At Opposite End`** function. If the option is done, the timer is reset for the next option state, as shown in Figure N.57. This *Option* state replaces eight individual option states and is a huge improvement in the scalability and maintainability of the Car Wash 2 controller. The rest of the states are identical to those found in the Car Wash 2 controller with time arguments developed in Exercise 8.20. This concludes the development of the Car Wash 2 controller with option arguments.



Problem Set 9 Solutions

Exercise 9.1

There are a number of ways we can leverage the existing `Timer` class to create an auto-reset timer object. One method would be to create a child class of the timer used in the car wash controller, with the auto-reset time check overriding the time and proximity time check of the car wash timer. Another approach is to simply add an auto-reset time check method to the existing `Timer` class. This is the approach that will be taken here.

The icon of the `Auto Reset Time Check` method of the `Timer` class is shown in Figure O.1. This method takes a `Timer` object and a time target as arguments. The block diagram of the method is shown in Figure O.2. The elapsed time and last timestamp are unbundled from the private data of the `Timer` object and are passed to the `Auto Reset Time Check Kernel` sub-VI developed in Figure H.10 and Figure H.11. The updated elapsed time and timestamp are then bundled into the private data of the timer object, which is passed as an output. A Boolean indicator is also passed as an output indicating if the timer is finished or not.



Figure O.1: The icon of the `Auto Reset Time Check` method of the `Timer` class

Exercise 9.2

The overall structure of the traffic light controller with the `Timer` class and the *Initialize* state is shown in Figure O.3. A `Timer` object is placed on the data highway, otherwise the structure is nearly identical to the traffic light controller with arguments developed in Exercise 8.5. The *Initialize* state initializes the `Timer` object using the `Initialize Timer` method of the `Timer` class. If an error is detected or if the user presses the STOP button, the *Shutdown* state is executed as shown in Figure O.4.

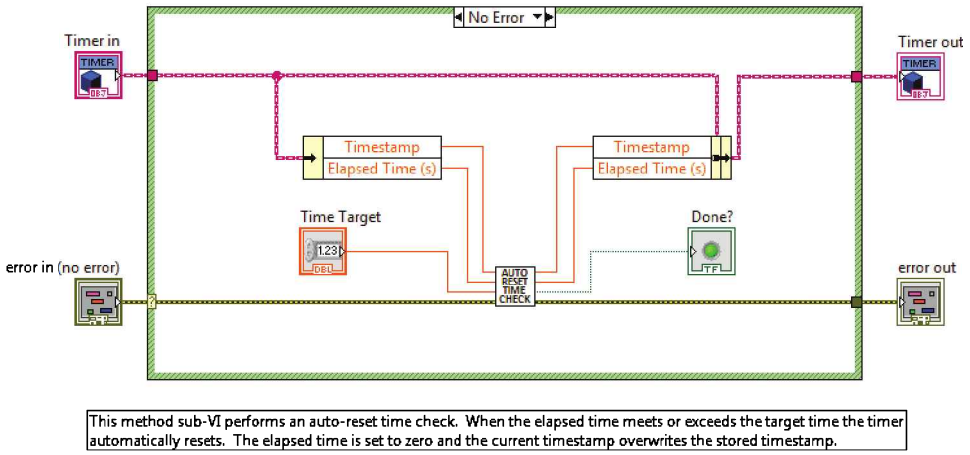


Figure O.2: The block diagram of the `Auto Reset Time Check` method of the `Timer` class

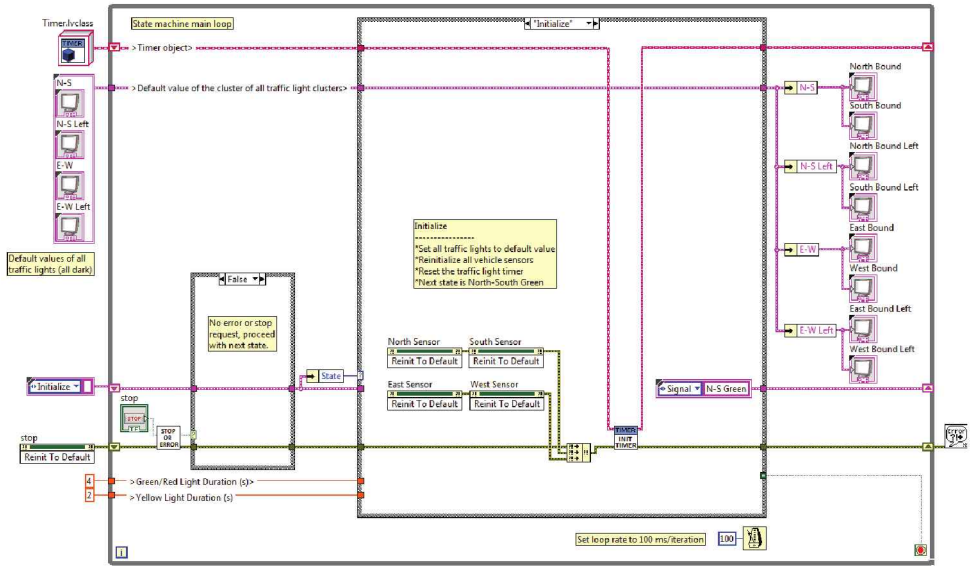


Figure O.3: The *Initialize* state of the traffic light controller with the `Timer` class

The traffic light controller spends most of its time in the *Signal* state as shown in Figure O.5. The `Timer` object is pulled down from the data highway and the auto-reset time check method is performed upon it. The altered object is then placed back on the data highway. Figure O.5 shows the case where the current signal is not done, Figure O.6 shows the case where the current signal is done. This concludes the traffic controller with a `Timer` object.

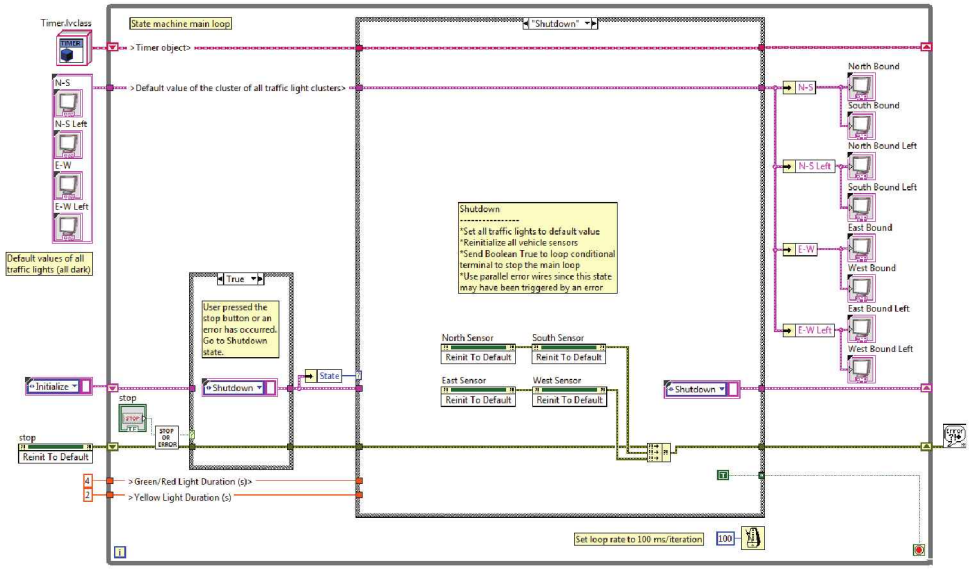


Figure O.4: The *Shutdown* state of the traffic light controller with the `Timer` class

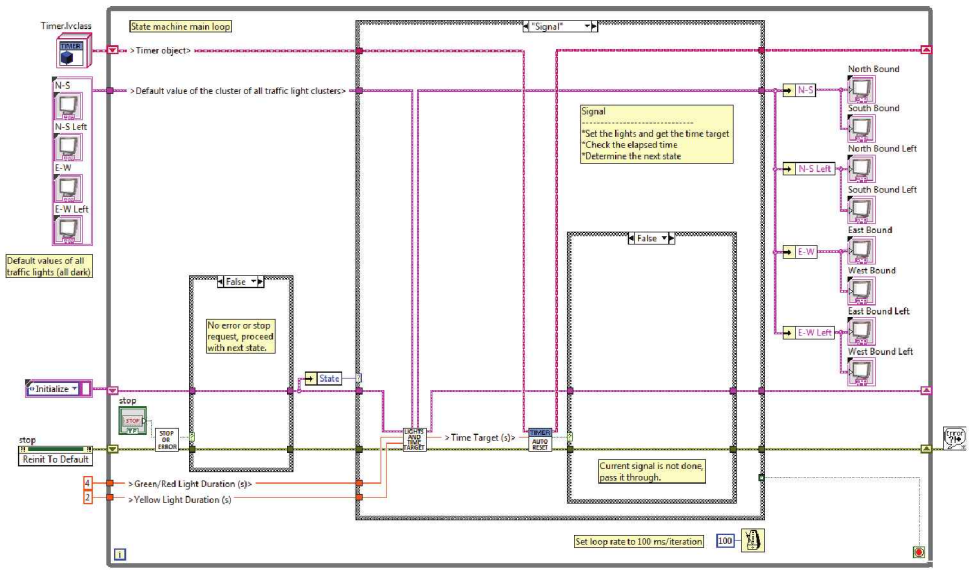


Figure O.5: The *Signal* state of the traffic light controller with the `Timer` class with the signal not done

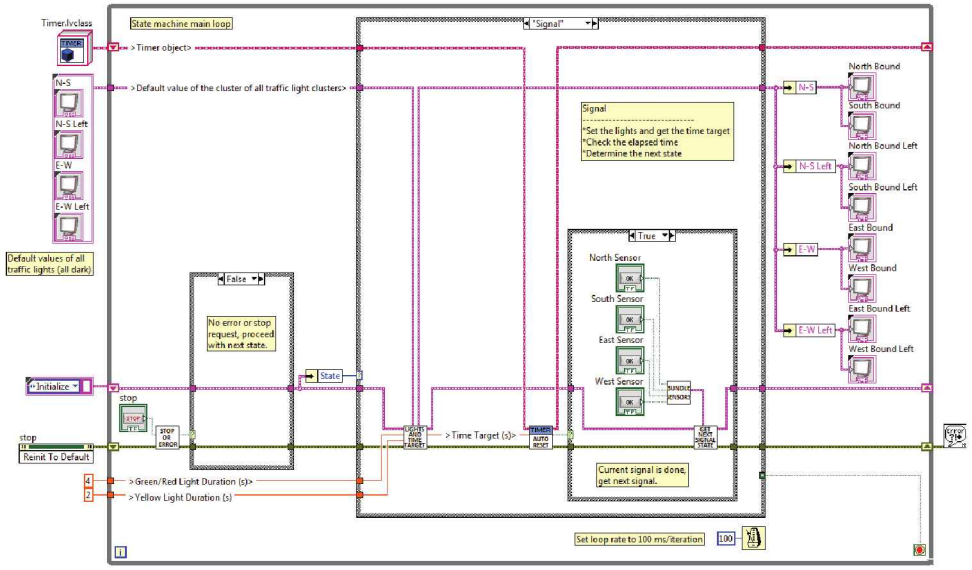


Figure O.6: The *Signal* state of the traffic light controller with the `Timer` class with the signal done

Exercise 9.3

One possible implementation of the `String Changed Kernel` sub-VI is shown in Figure O.7. This sub-VI accepts the old string as a state variable and compares it to the new string. The new string then overwrites the old string and is output as the new state variable value.

This sub-VI can now be used in a revised version of the `Write to Boiler Log File` that uses the data highway for state variable storage. One possible implementation of the `Write to Log Data Highway` sub-VI is shown in Figure O.8. The STORED STRING control accepts the state variable string and compares it to the the

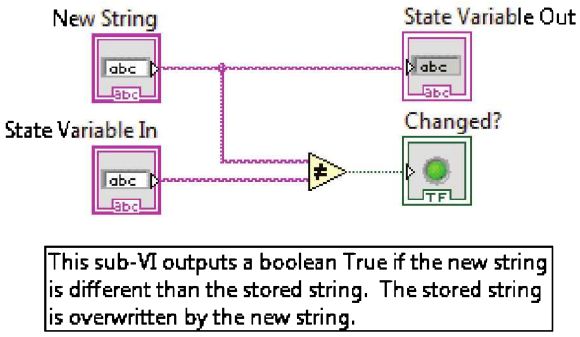


Figure O.7: The block diagram of the `String Changed Kernel` sub-VI

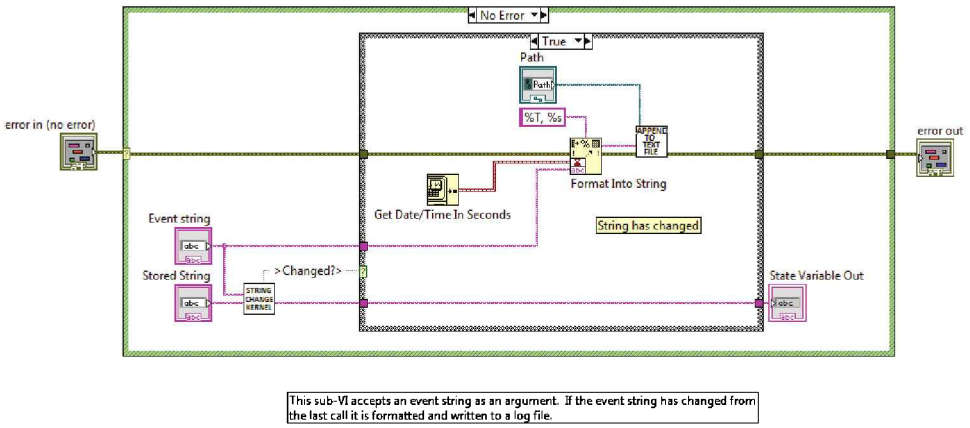


Figure O.8: The block diagram of the `Write To Log Data Highway` sub-VI with the string changed

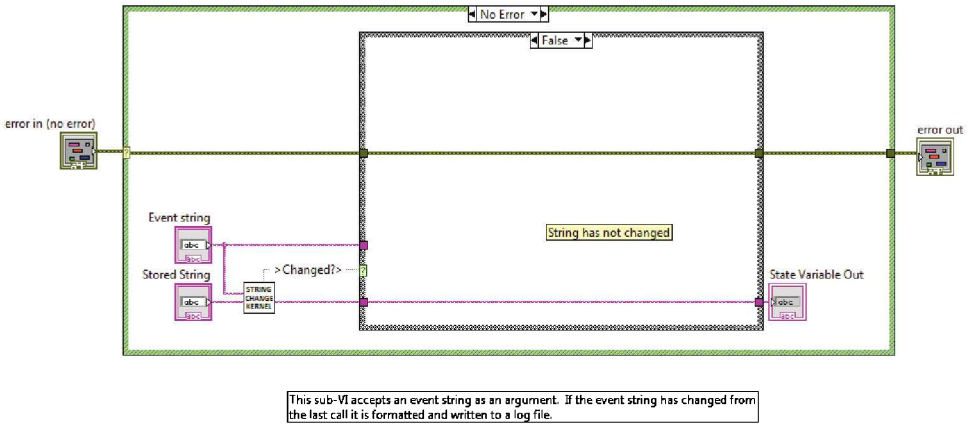


Figure O.9: The block diagram of the `Write To Log Data Highway` sub-VI with the string not changed

new string using the `String Changed Kernel` sub-VI. If the string has changed, a timestamp is added to it, and the string is written to the log file. If the string has not changed, no action is taken, as seen in Figure O.9. In either case, the new state variable string is passed as an output so that it can be stored on the data highway of the main VI.

Exercise 9.4

Upgrading the boiler controller to use the `Write to Log Data Highway` sub-VI is straightforward, as can be seen in Figure O.10. A data highway has been added to the boiler controller to store the log string state variable. The data highway shift register is

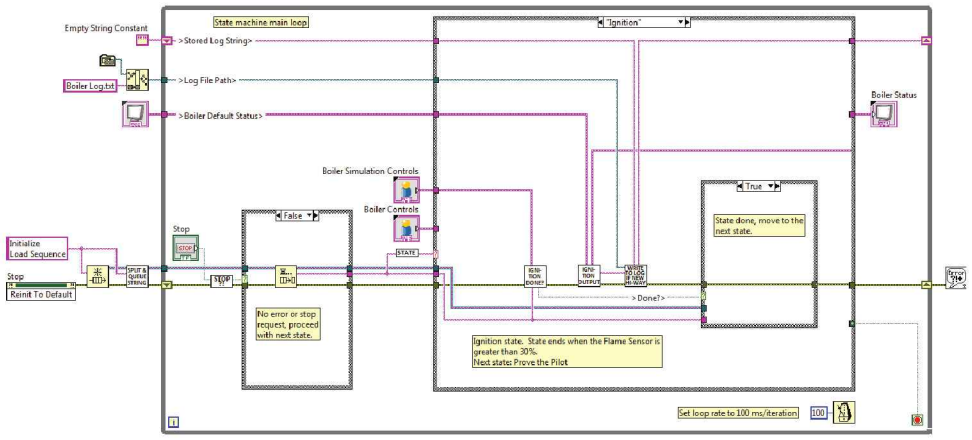


Figure O.10: The *Ignition* state of the boiler controller with data highway string storage

initialized with an empty string constant to ensure that the first log string placed on the data highway is different from the stored string. The `Write to Boiler Log File` sub-VI has been replaced with the `Write to Log Data Highway` sub-VI. This sub-VI pulls the stored log string down from the data highway and compares it to the new log string. The new log string is then passed as an output and is placed back on the data highway, overwriting the old log string. Figure O.10 shows the *Ignition* state. Every other state that writes to the log file implements a similar change, replacing the `Write to Boiler Log File` sub-VI with the `Write to Log Data Highway` sub-VI. States that do not write to the log file simply pass the value on the data highway through to the output shift register. These other states are not shown here. This concludes the implementation of the boiler controller that stores the log string on the data highway.

Exercise 9.5

The stopwatch timer is simpler than the other timers discussed in this book because it does not use a time target; it simply reports the elapsed time since it was last reset or initialized. This requires the same stored data used by the `Timer` class: the current elapsed time and the timestamp of the last call. Since the stored data is the same as the private data of the `Timer` class, we do not need to create a new class. Instead we can add a `Stopwatch` method to the existing `Timer` class.

The icon for the `Stopwatch Method Kernel` sub-VI is shown in Figure O.11, and the block diagram is shown in Figure O.12. The block diagram logic is identical

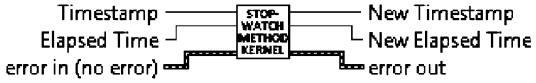


Figure O.11: The icon of the `Stopwatch Method Kernel` sub-VI

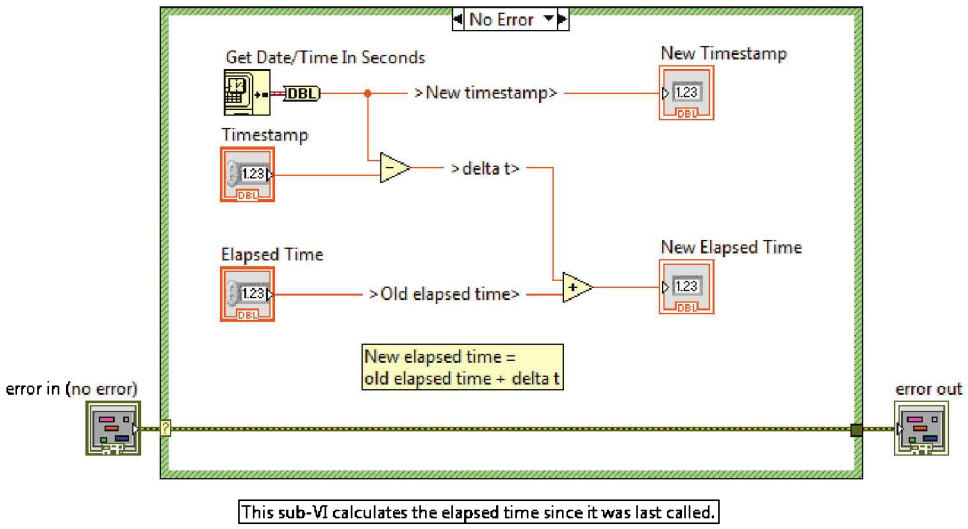


Figure O.12: The block diagram of the `Stopwatch Method Kernel` sub-VI

to that used by the boiler timer action engine seen in Figure H.6. The kernel obtains a current timestamp, calculates the elapsed time since it was last called, and the calculates the new elapsed time. The new elapsed time and new timestamp are passed as outputs.

The stopwatch timer will also need to be reset, but a new method does not have to be created for this. The `Initialize Timer` developed in Figure 23.21 can be used to reset or initialize the stopwatch timer.

Exercise 9.6

The *Reset Lockout* state of the revised boiler controller is shown in Figure O.13. A `Timer` object has been placed in the input area of the loop. The `Timer` object needs to be stored persistently from one loop iteration to the next, so a data highway was created for the `Timer` object. When the *Reset Lockout* state is completed, the timer needs to be reset in preparation for the next state, which is the *Pre-Purge Timer* state. The timer is reset in the *Reset Lockout* state by pulling the `Timer` object from the data highway and passing it to the `Initialize Timer` method of the `Timer` class. The initialized `Timer` object is then placed back on the data highway for use in the next loop iteration.

The *Pre-Purge Timer* state is shown in Figure O.14. Once again, the `Timer` object is pulled down from the data highway, but this time it is passed to the `Stopwatch` method of the `Timer` class to perform a time check. It is then returned to the data highway.

The same pattern repeats itself for the *Run* state and the *Purge* state. The `Timer` object is initialized using the the `Initialize Timer` method in the *Run* state, as seen in Figure O.15. The time check is performed using the `Stopwatch` method in

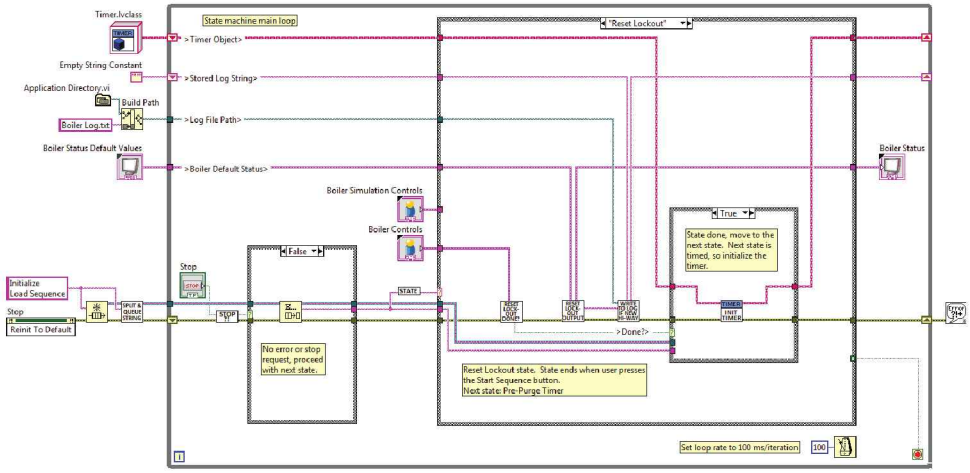


Figure O.13: The *Reset Lockout* state of the revised boiler controller

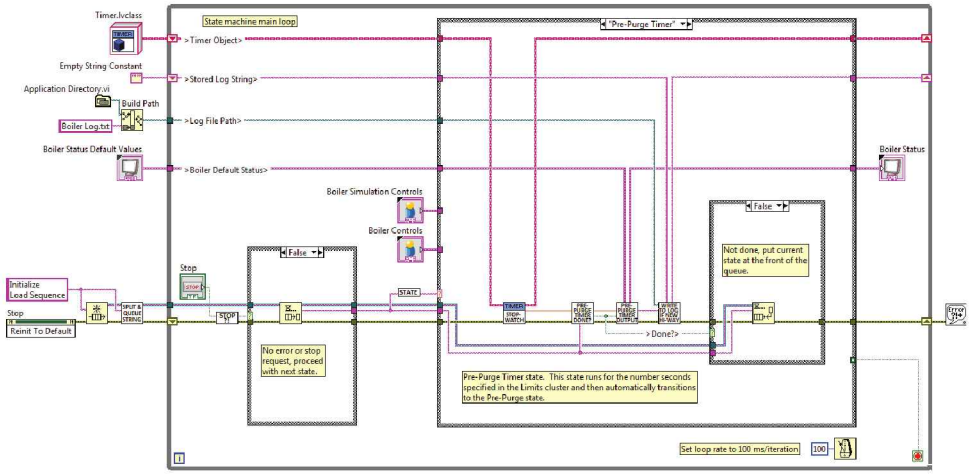


Figure O.14: The *Pre-Purge Timer* state of the revised boiler controller

the *Purge* state, as seen in Figure O.16. The other states of the boiler controller do not use the timer and are not affected by the addition of the `Timer` class. This concludes the development of the data-encapsulated boiler controller.

Exercise 9.7

An implementation of the `Car Wash 2 Time Check` method sub-VI is shown in Figure O.17. This implementation is identical to the one shown in Figure 23.18, except for the addition of an indicator to output the elapsed time.

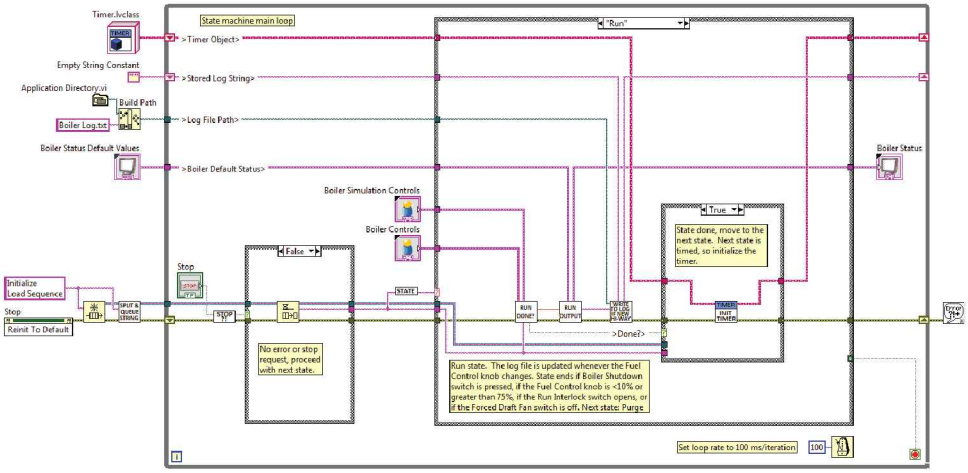


Figure O.15: The *Run* state of the boiler controller with `Timer` class

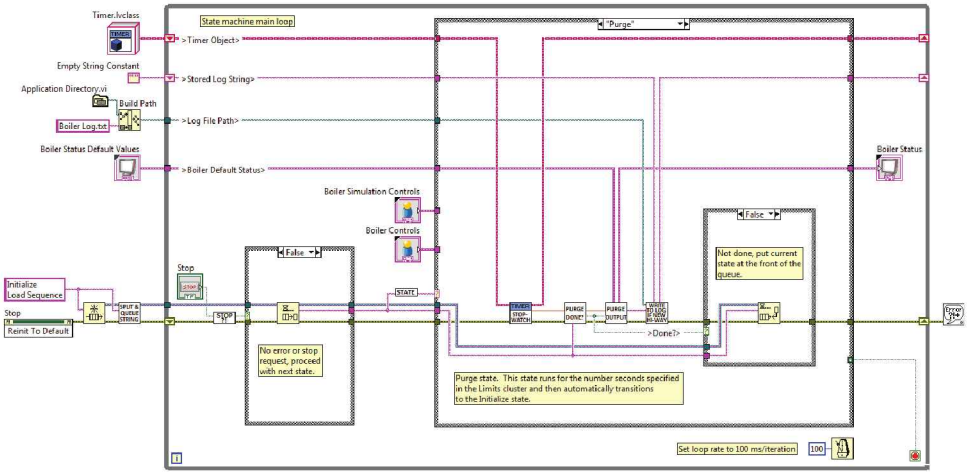


Figure O.16: The *Purge* state of the boiler controller with `Timer` class

Exercise 9.8

One implementation of the `Run Car Wash Option Timer` based on the VI developed in Exercise 8.22 is shown in Figure O.18. This sub-VI is identical to that shown in Figure N.54, except for the fact that it uses the `Car Wash 2 Time Check Method` of the `Timer` class developed in Exercise 9.7 rather than an action engine to perform the time check.

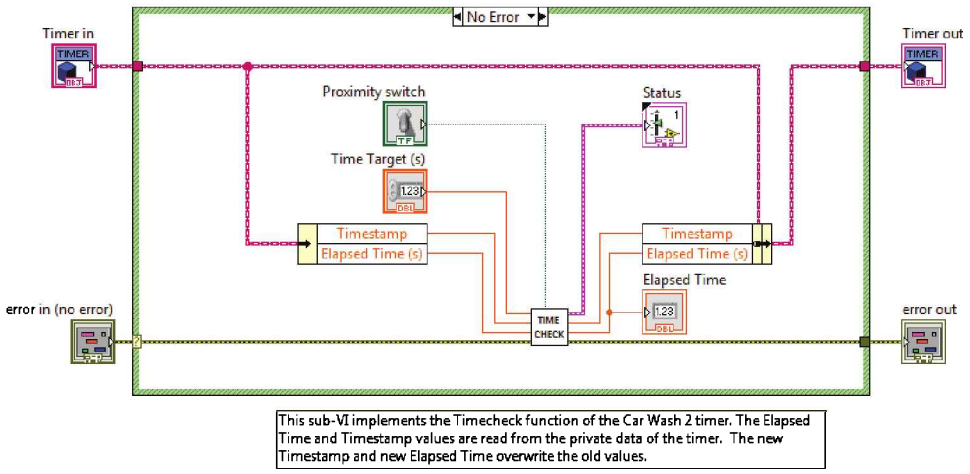


Figure O.17: The block diagram of the `Car Wash 2 Time Check` sub-VI

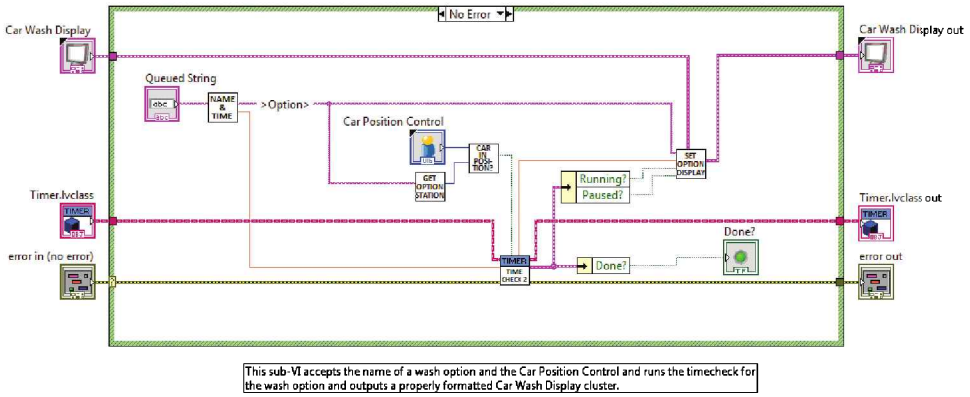


Figure O.18: The block diagram of the `Run Car Wash Option Timer lvclass` sub-VI

Exercise 9.9

The structure of the Car Wash 2 controller with `Timer` object is shown in Figure O.19. It is identical to the Car Wash 2 controller with option arguments with the exception of a `Timer` object on the data highway. Figure O.19 shows the *Car Enter* state while waiting for a car to enter. The `Timer` object stays on the data highway untouched until a car enters the car wash, as seen in Figure O.20. The timer is then reset using the `Initialize Timer` method of the `Timer` class. The initialized `Timer` object is then placed back on the data highway so that it can be used in the first wash option state.

The *Option* state is shown in Figure O.21. The `Timer` object is passed as an input to the `Run Car Wash Option Timer lvclass` sub-VI developed in Exercise 9.8. This sub-VI uses the `Car Wash 2 Time Check` method of the `Timer` class

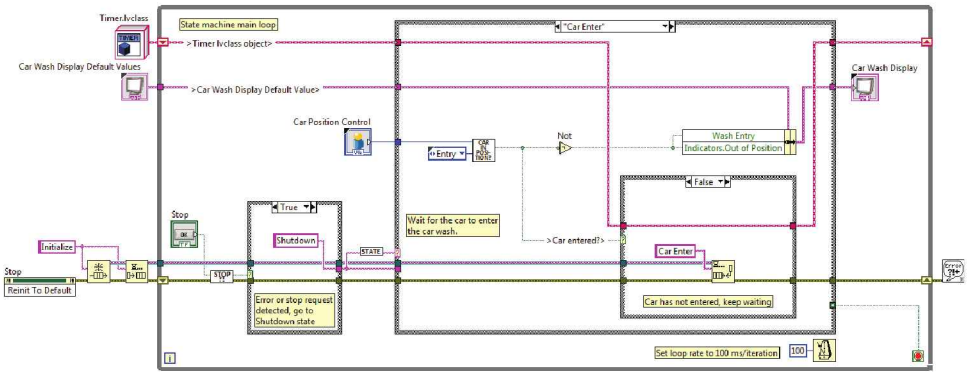


Figure O.19: The *Car Enter* state of the Car Wash 2 controller with `Timer` object with the car not entered

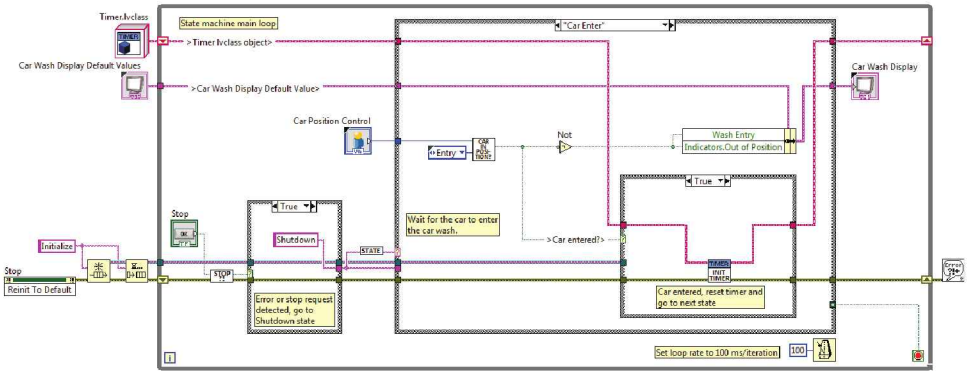


Figure O.20: The *Car Enter* state of the Car Wash 2 controller with `Timer` object with the car entered

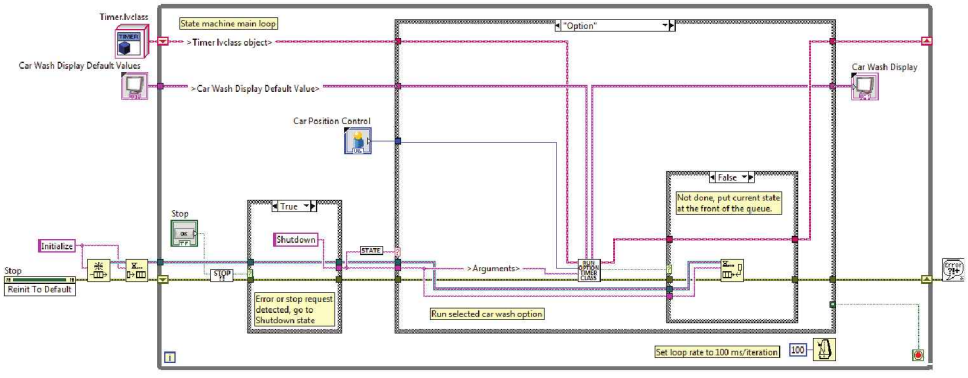


Figure O.21: The *Option* state of the Car Wash 2 controller with `Timer` object with the option not done

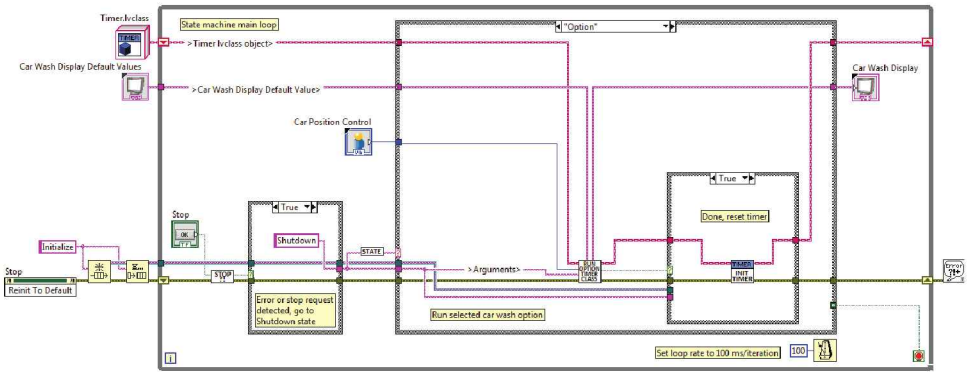


Figure O.22: The *Option* state of the Car Wash 2 controller with `Timer` object with the option done

developed in Exercise 9.7 to perform the time check and output the elapsed time. If the wash option has not met the time target, the queued string is placed back at the front of the queue using the **`Enqueue Element At Opposite End`** function. If the wash option state has met the time target, the timer is reset, as shown in Figure O.22 using the `Initialize Timer` method of the `Timer` class. The timer is then placed back on the data highway so that it can be used by the next wash option state. The other states of the controller do not access the timer and are identical to those in the Car Wash 2 controller with option arguments developed in Exercise 8.23. This concludes the development of the data-encaspusated Car Wash 2 controller with `Timer` object.

P

Problem Set 10 Solutions

Exercise 10.1

The `Option` class is very similar in structure to the `Cycle` class. Not only do they have identical private data, but the class libraries are also very similar. Both classes have read and write data accessors for the DONE Boolean. Both classes have dynamic dispatch methods for determining the value of the proximity switch used in the time check and for setting the appropriate value of the output display cluster. Both classes have static dispatch methods for initializing the private data, for performing the time check, and for running the wash cycle or wash option. In both classes, the static dispatch method for performing the time check is private in scope.

Exercise 10.2

The block diagram of the `Read Done` method is shown in Figure P.1. The value of the DONE Boolean is unbundled from the private data and sent to the output indicator.

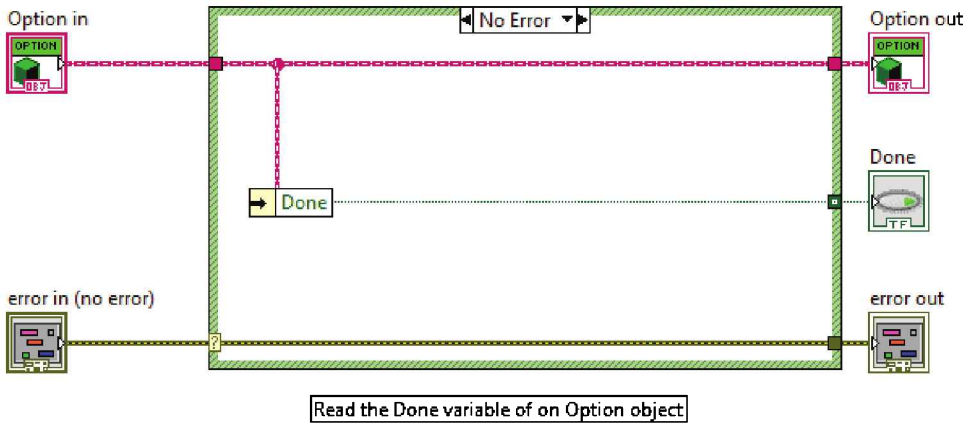


Figure P.1: The block diagram of the `Read Done` method of the `Option` class

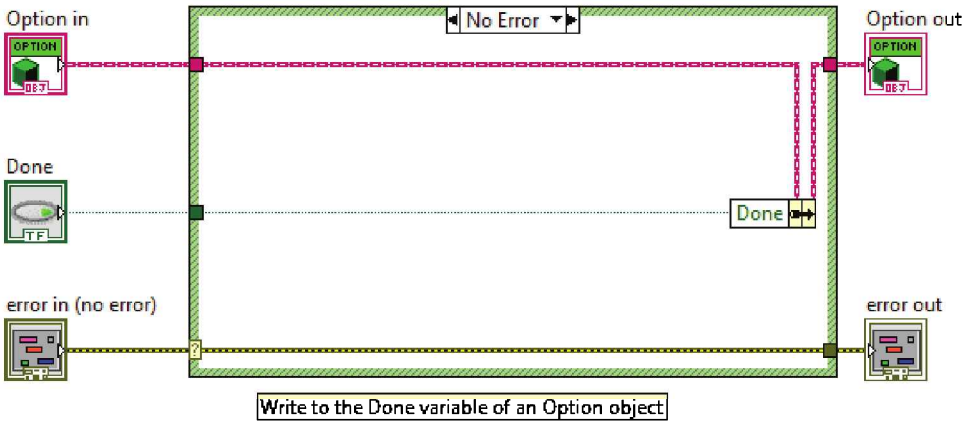


Figure P.2: The block diagram of the `Write Done` method of the `Option` class

The block diagram of the `Write Done` method is shown in Figure P.2. The new value of the DONE Boolean is bundled into the private data.

Exercise 10.3

The block diagram of the `Get Position` sub-VI is shown in Figure P.3. Although this sub-VI will be overridden by the child classes, the sub-VI is fully developed here to demonstrate its function. The `Car In Position` VI developed in Exercise 1.39 is used to compare the current value of the car position slider to the appropriate value of the car position enumerated constant.

The block diagram of the `Set Display` sub-VI is shown in Figure P.4. Although this sub-VI will be overridden by the child classes, the sub-VI is fully developed here to demonstrate its function. A default constant of the display indicator is wired to the

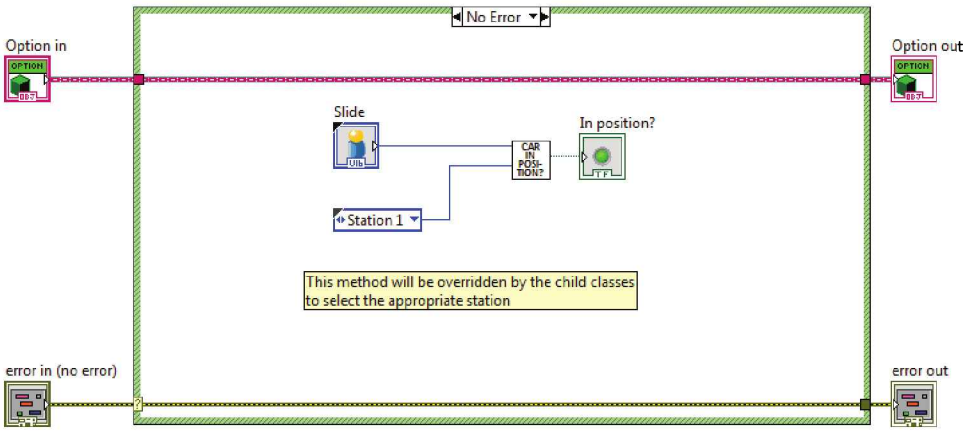


Figure P.3: The block diagram of the `Get Position` method of the `Option` class

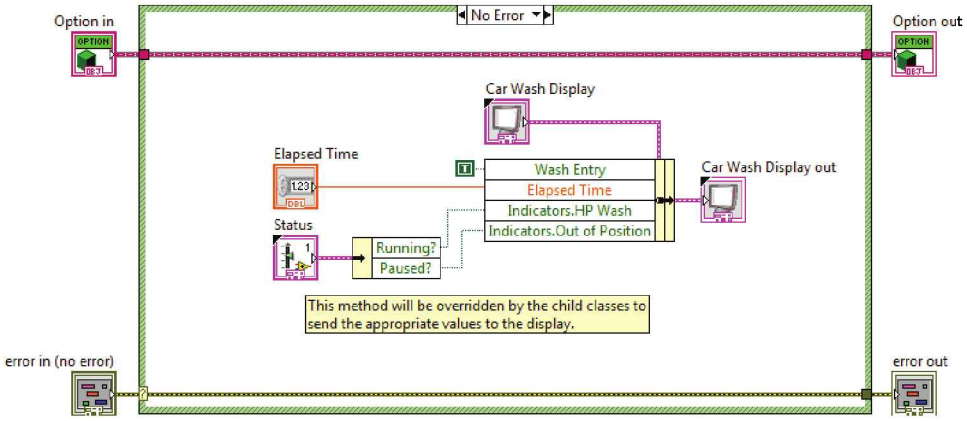


Figure P.4: The block diagram of the `Set Display` method of the `Option` class

central input of a **`Bundle By Name`** function. A Boolean True is wired to the *Wash Entry* field of the cluster, and the value of the ELAPSED TIME control is written to the *Elapsed Time* field of the cluster. The *Running?* Boolean of the status cluster is wired to the appropriate display indicator, and the *Paused?* Boolean of the status cluster is written to the out-of-position indicator.

Exercise 10.4

The block diagram of the `Initialize Option` method is shown in Figure P.5. This sub-VI writes the time target to the private data, sets the *Done* Boolean to True, and initializes the `Timer` object. The block diagram of the `Option Time Check` method sub-VI is shown in Figure P.6. This method unbundles the `Timer` object from

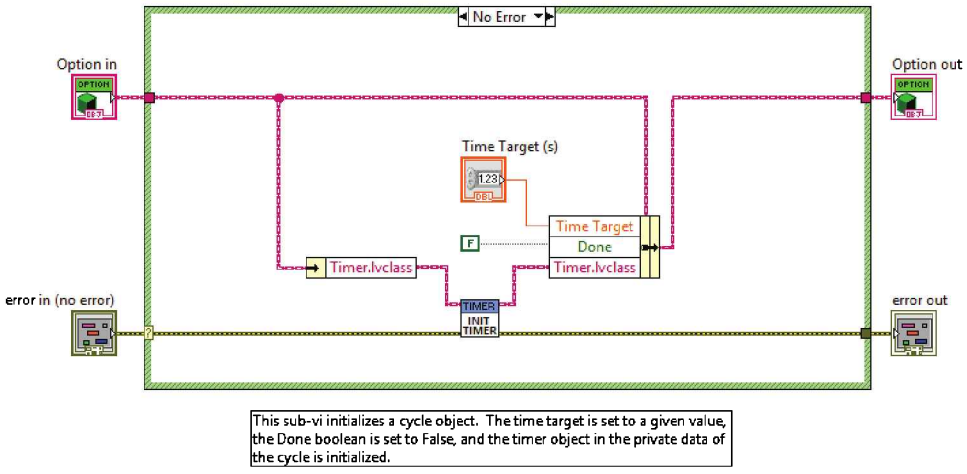


Figure P.5: The block diagram of the `Initialize Option` method of the `Option` class

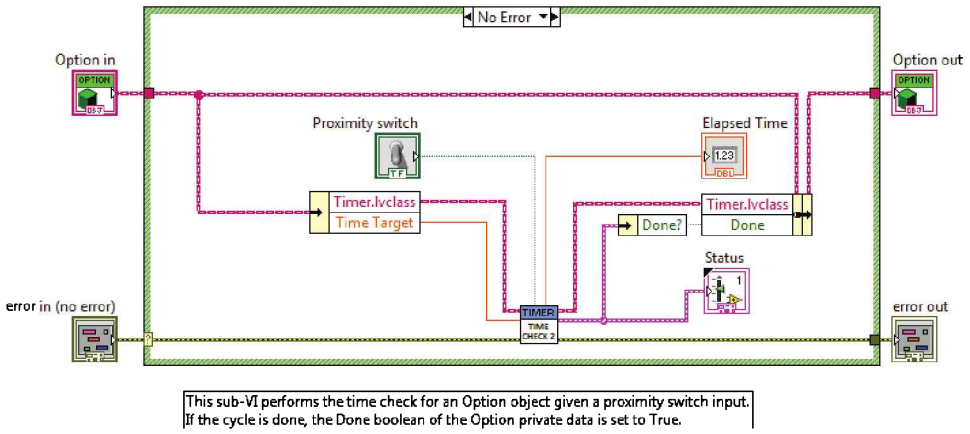


Figure P.6: The block diagram of the `Option Time Check` method of the `Option` class

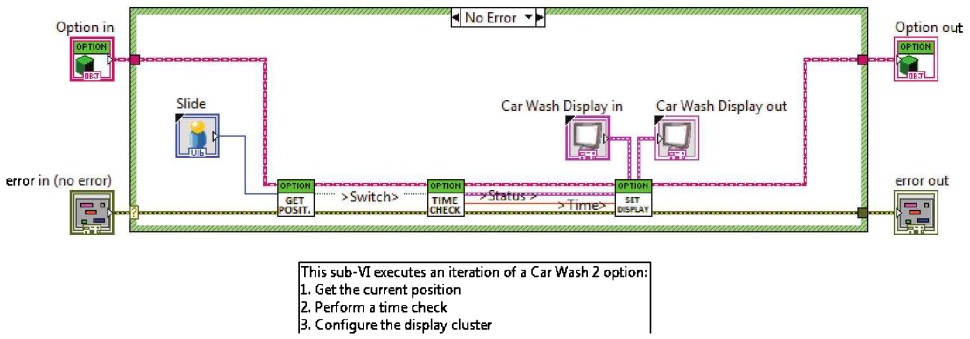


Figure P.7: The block diagram of the `Run Option` method of the `Option` class

the private data and uses the `Car Wash 2 Time Check Method` developed in Exercise 9.7 to perform the time check using a given value of the proximity switch.

The block diagram of the `Run Option` static dispatch method is shown in Figure P.7. This sub-VI is the heart of the `Option` class. This sub-VI accepts the car position slider as an input, outputs the value of the proximity switch using the dynamically dispatched `Get Position` method sub-VI, runs the time check using the statically dispatched `Option Time Check` sub-VI, and then generates the display cluster using the dynamically dispatched `Set Display` sub-VI.

Exercise 10.5

The eight child classes are nearly identical. The `Bug Remover` class will be shown here as an example. The `Bug Remover` class library and private data are shown in Figure P.8. The private data for this class is empty; no new items are added to the private data of the parent `Option` class.

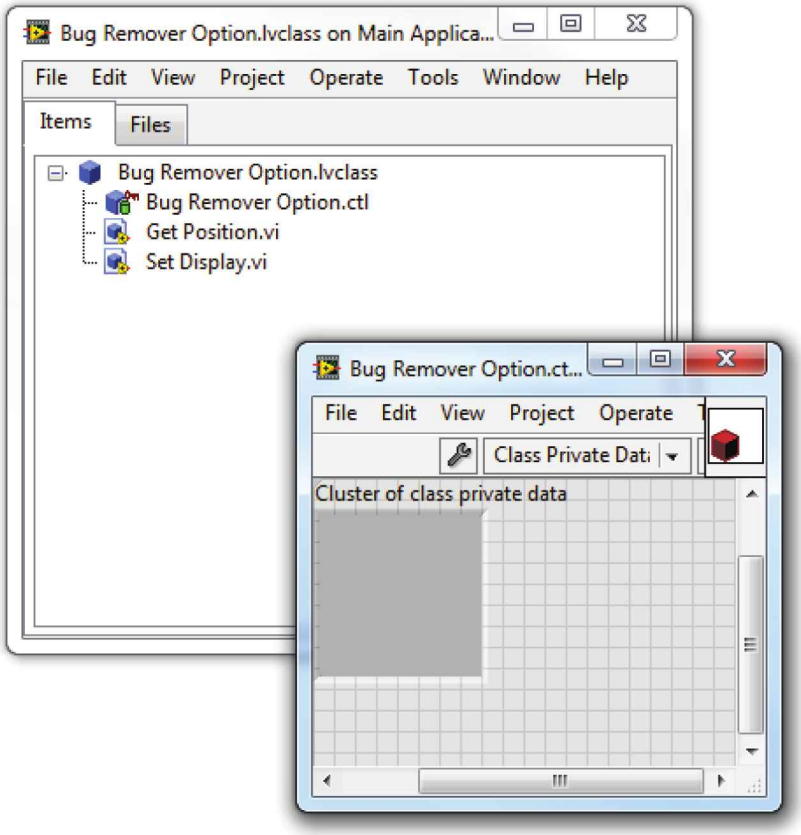


Figure P.8: The library and private data of the `Bug Remover` class



Figure P.9: The icon of the `Get Position` method of the `Bug Remover` class

Each child class overrides the two dynamic dispatch methods of the `Option` class. Using the `Bug Remover` class as an example, the icon for the `Get Position` override method is shown in Figure P.9, and the block diagram is shown in Figure P.10. The icon for the `Set Display` override method is shown in Figure P.11, and the block diagram is shown in Figure P.12.

Exercise 10.6

The `Get Option Object` sub-VI uses a case structure to produce the object whose name is wired to the selector terminal. As an example, the *Bug Remover* case of the case structure is shown in Figure P.13. The other cases are similar and are not shown here. Since a string is wired to the selector terminal, a default case must be defined

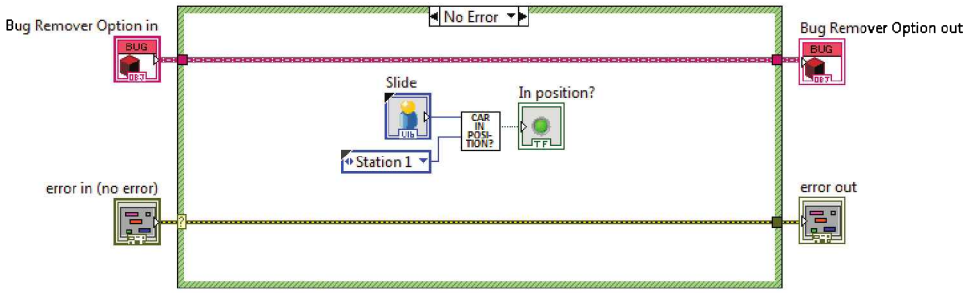


Figure P.10: The block diagram of the `Get Position` method of the `Bug Remover` class

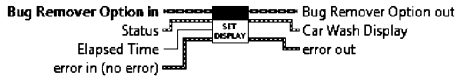


Figure P.11: The icon of the `Set Display` method of the `Bug Remover` class

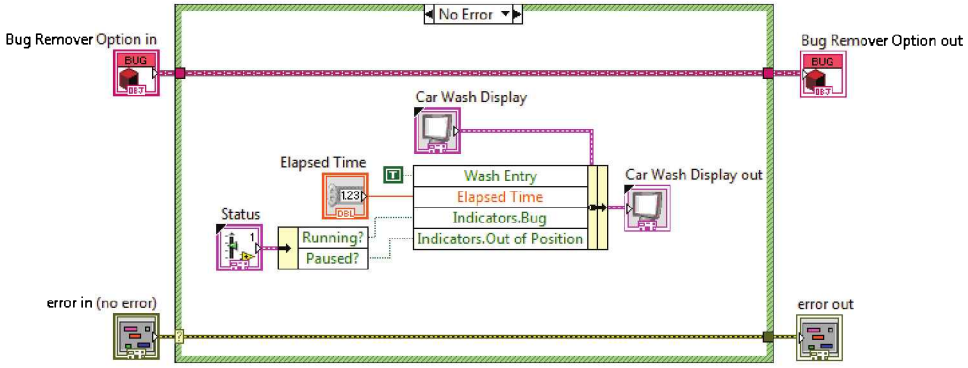


Figure P.12: The block diagram of the `Set Display` method of the `Bug Remover` class

as shown in Figure P.14. If the input string is `Option` or is not recognized, a generic `Option` object is produced.

Exercise 10.7

The block diagram of the `Option Factory` sub-VI is shown in Figure P.15. This figure shows the code executed when the input object is done. The `Parse Name and Time Target` sub-VI developed in Figure 21.5 is used to parse the option name and the time target from the input string. The option name is passed to the `Get Option Object` sub-VI developed in Exercise 10.6 to generate the desired object. The new object and the time target are then passed to the `Initialize Option` method of the `Option` class to properly initialize the new option object. Figure P.16 shows the code that executes if the input object is not done. The object is passed through to the output unchanged.

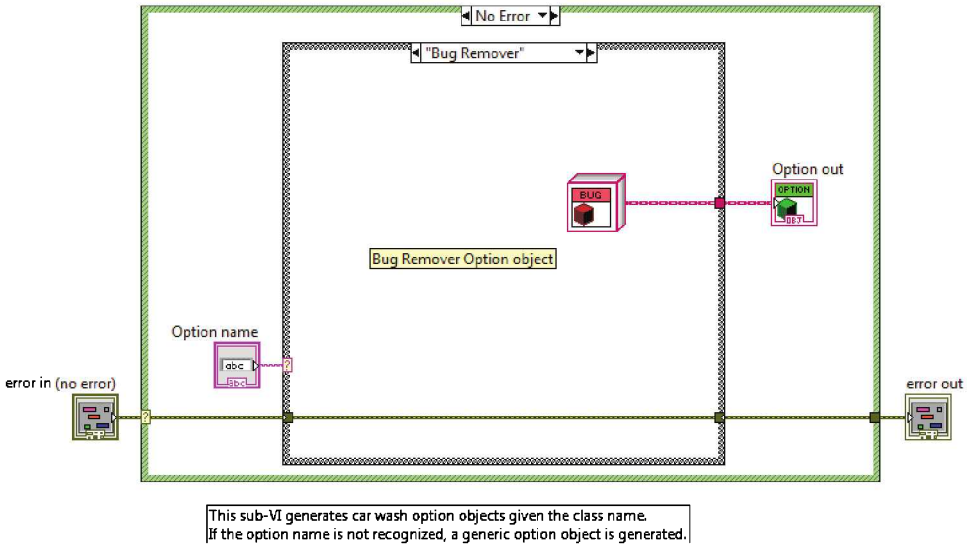


Figure P.13: The *Bug Remover* case of the `Get Option Object` sub-VI

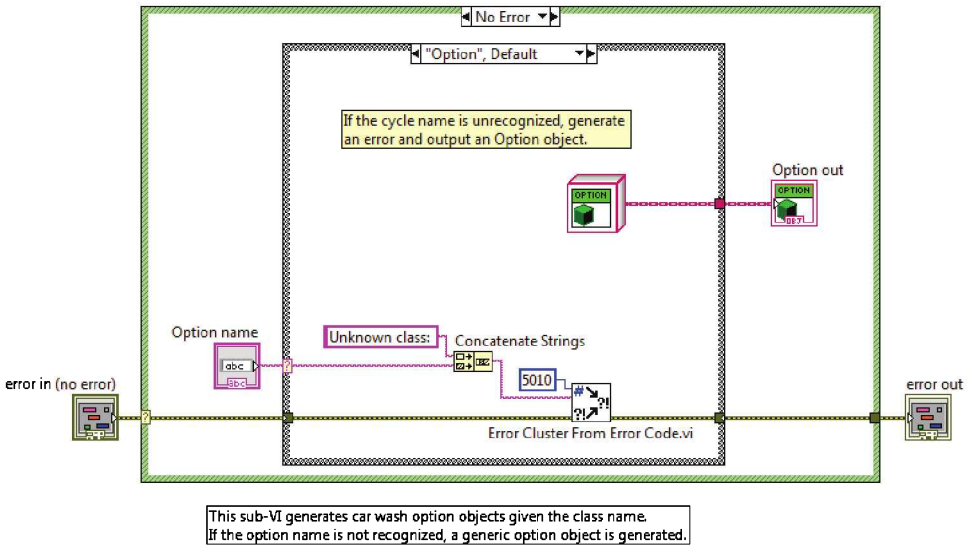


Figure P.14: The default case of the `Get Option Object` sub-VI

Exercise 10.8

The *Option* state of the Car Wash 2 controller using the `Option` class is shown in Figure P.17. This figure is remarkably similar to the corresponding *Cycle* state of the original car wash controller shown in Figure 25.1.

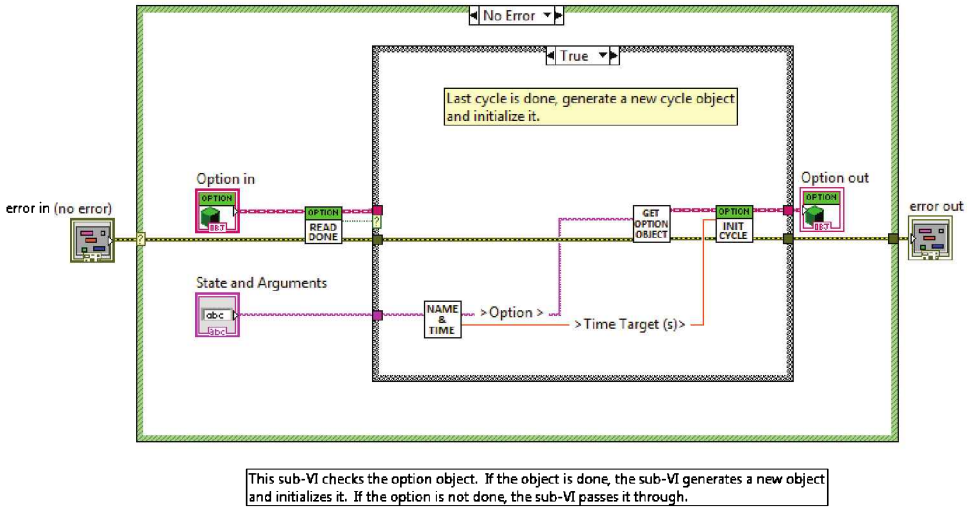


Figure P.15: The `Option Factory` sub-VI with the input object done

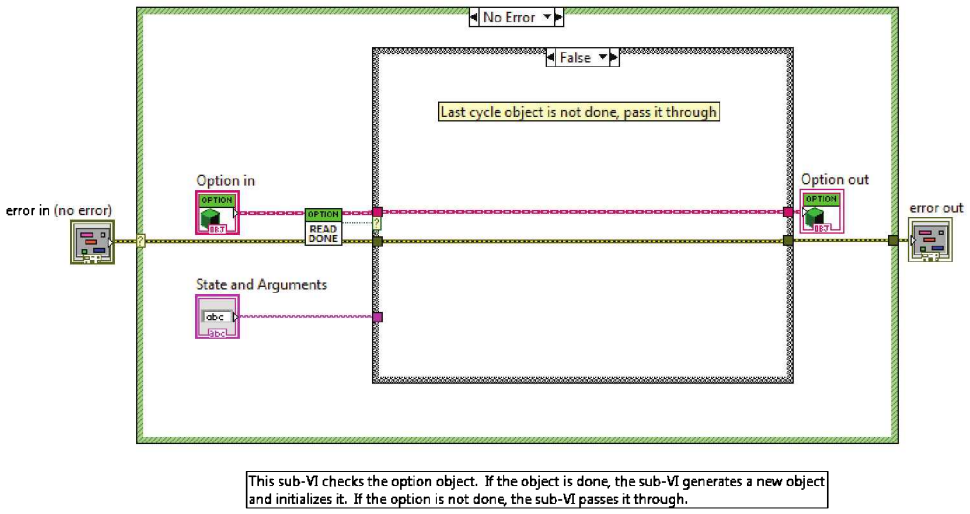


Figure P.16: The `Option Factory` sub-VI with the input object not done

An `Option` object is placed in the input area of the main loop. The DONE Boolean in the private data of the object is set to True using the `Write Done` method VI, and the resulting object is then used to initialize a shift register on the main loop. When the *Option* state is executed, this object is pulled down from the data highway and is fed as an input to the `Option Factory` sub-VI developed in Exercise 10.7. Since the object is completed, it is discarded, and the first object of the car wash program is generated and initialized instead. This object is then passed on to the `Run Option`

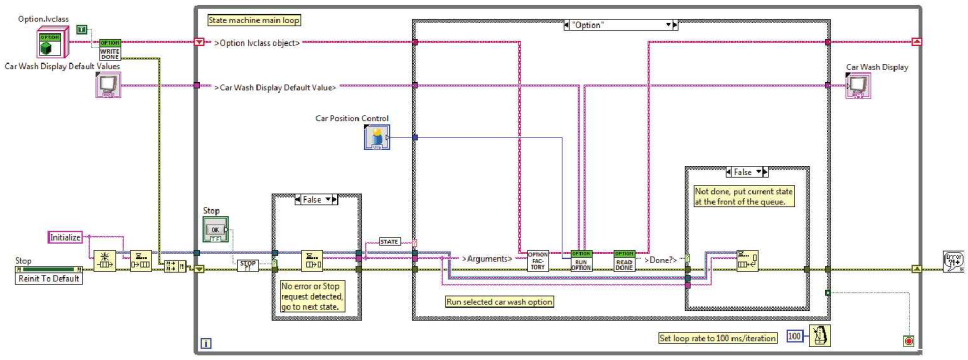


Figure P.17: The *Option* state of the Car Wash 2 controller using the `Option` method sub-VI developed in Exercise 10.4. This method sub-VI selects the appropriate proximity switch, runs the time check, and formats the output display. The `Read Done` method sub-VI is then used to check if the current option is finished. If it is not finished, the current state and argument string is placed at the front of the queue using the **`Enqueue At Opposite End`** function. If the current object is done, the string is not requeued. The object is placed back on the data highway and the process begins again during the next loop iteration.

Notice that this is nearly identical to the *Cycle* state in the original car wash controller. In both cases, the main VI is written entirely in terms of the abstract parent class (the `Cycle` class or the `Option` class). The abstract class is never actually used; the child classes travel on the wires defined by parent class instead. This improves the readability, maintainability, and scalability of the main VI.

Exercise 10.9

The private data and class library of the `Signal` class are shown in Figure P.18. The private data consists of a `Timer` object and a numeric control for the time target. The library shows two public static dispatch methods (`Initialize Signal` and `Time Check`), one private static dispatch method (`Set Time Target`), and one dynamic dispatch method (`Set Lights`).

The icon of the dynamically-dispatched `Set Lights` is shown in Figure P.19 and the block diagram is shown in Figure P.20. A default value of the signal lights cluster constant is passed to a **`Bundle By Name`** function, and the appropriate lights are lit by bundling a Boolean True to the relevant indicators.

This method sub-VI will be overridden by the child classes. The structure of the override methods will be identical, but different indicators will be selected in the **`Bundle By Name`** function.

The `Set Lights` method can be used to determine whether a given signal object is a green/red signal or a yellow signal in order to set the correct time target. The icon of the `Set Time Target` method sub-VI is shown in Figure P.21, and the block diagram is shown in Figure P.22. The `Set Lights` method is used to generate the light pattern for a given `Signal` object. The values of the yellow lights are unbundled

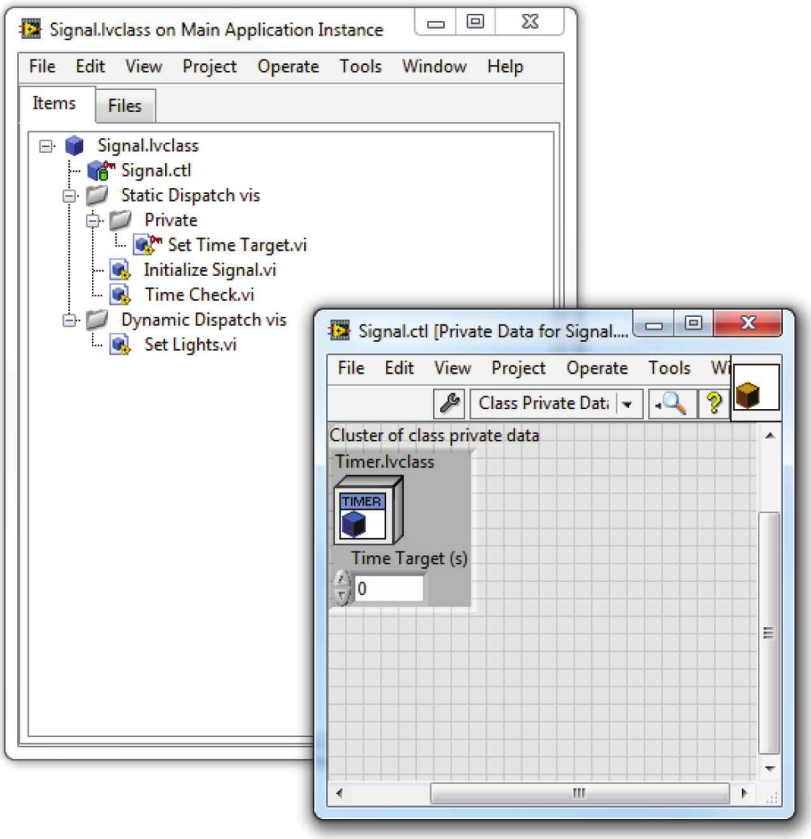


Figure P.18: The private data and class library of the `Signal` class

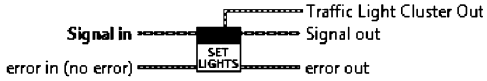


Figure P.19: The icon of the `Set Lights` method of the `Signal` class

and the logical `Or` is taken of the set. If the result is true, the object is given the duration of a yellow light. Otherwise, it is given the duration of a green/red light.

The `Set Time Target` method sub-VI can then be used by the `Initialize Signal` method sub-VI. The icon of this sub-VI is shown in Figure P.23, and the block diagram is shown in Figure P.24. This method unbundles the `Timer` object from the private data of the `Signal` object and initializes it using the `Initialize Timer` method of the `Timer` class developed in Figure 23.21. The `Timer` object is then rebundled into the private data, and the `Signal` object is then passed to the `Set Time Target` method of the `Signal` class to initialize the time target of the object.

The final method for the `Signal` class is the `Time Check` method. The icon for this method is shown in Figure P.25, and the block diagram is shown in Figure P.26.

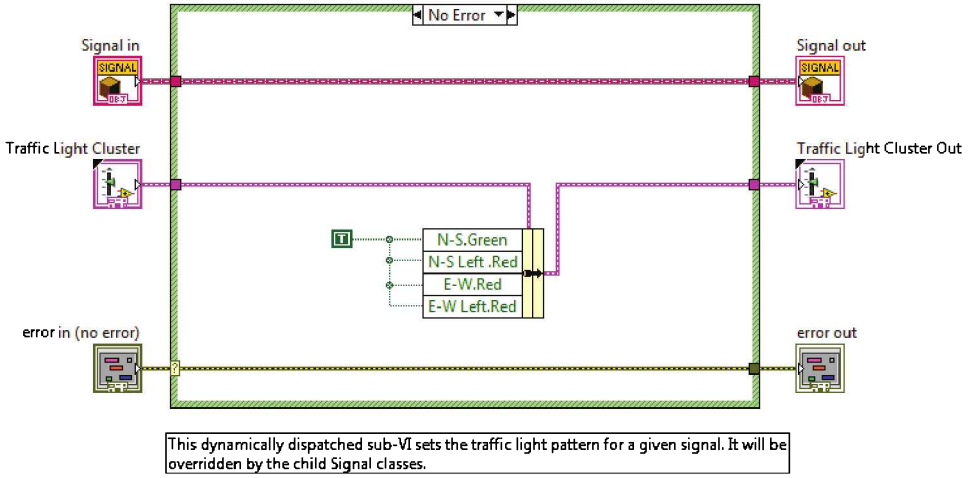


Figure P.20: The block diagram of the `Set Lights` method of the `Signal` class

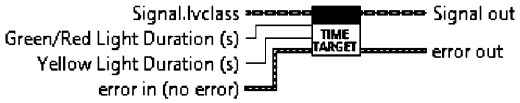


Figure P.21: The icon of the `Set Time Target` method of the `Signal` class

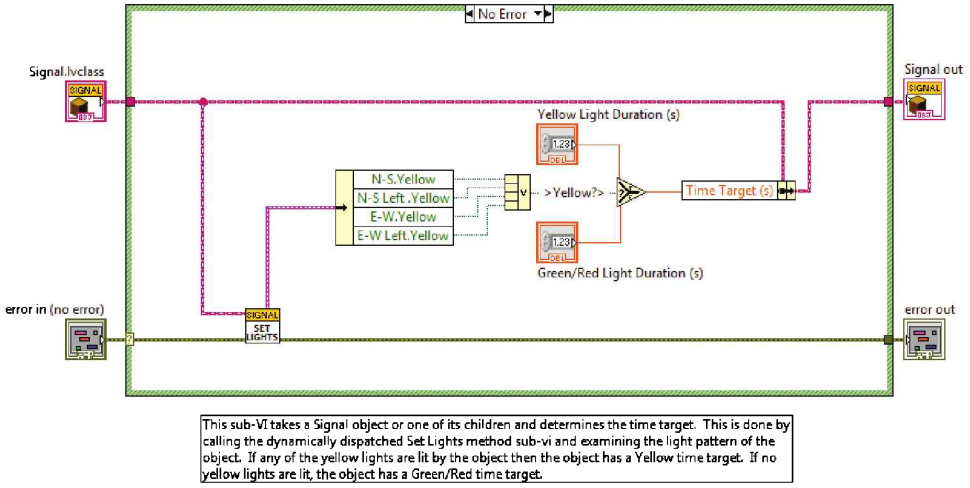


Figure P.22: The block diagram of the `Set Time Target` method of the `Signal` class

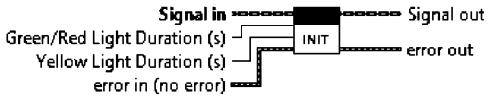


Figure P.23: The icon of the `Initialize Signal` method of the `Signal` class

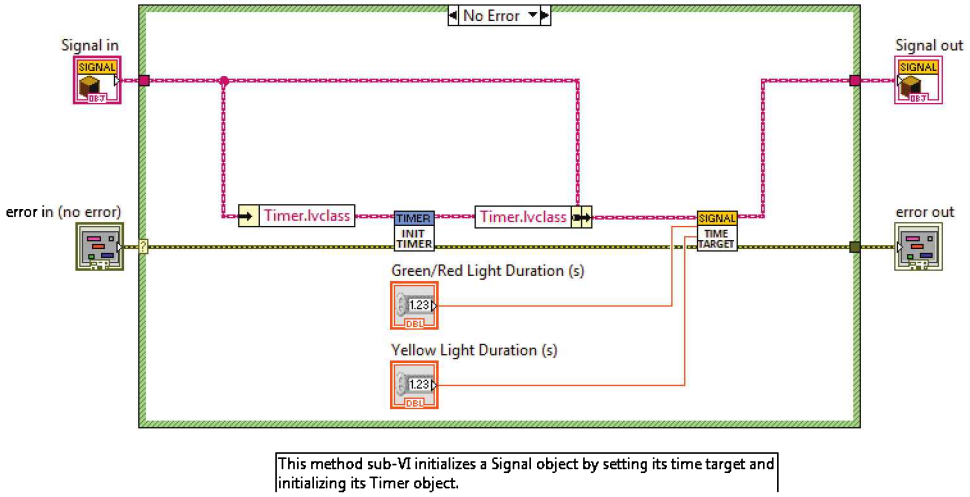


Figure P.24: The block diagram of the `Initialize Signal` method of the `Signal` class

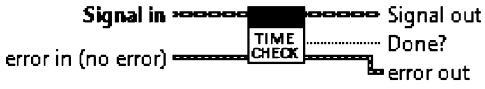


Figure P.25: The icon of the `Time Check` method of the `Signal` class

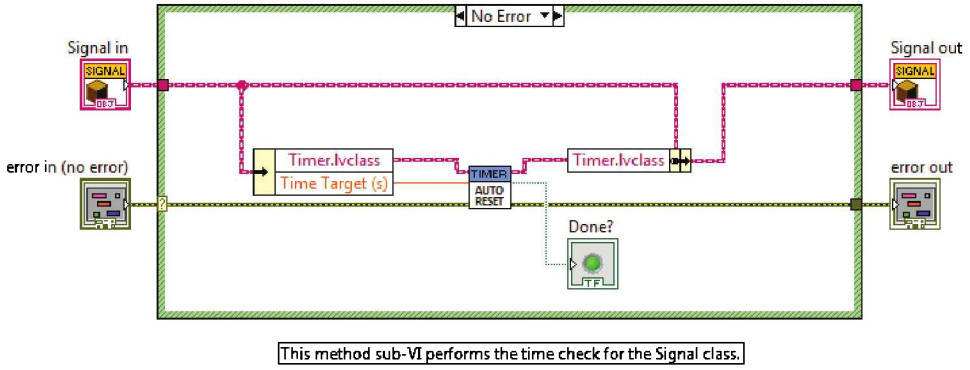


Figure P.26: The block diagram of the `Time Check` method of the `Signal` class

The `Timer` object and the time target are unbundled from the private data of the `Signal` object, and the `Auto Reset Time Check` method of the `Timer` class (developed in Figure O.2) is then executed on the `Timer` object. The `Timer` object is then bundled back into the private data of the `Signal` object.

Exercise 10.10

The private data and class library of the `E-W Green` class are shown in Figure P.27. The private data is empty since the child class needs no data other than that found in the parent class. The class library contains only the dynamic dispatch override of the `Set Lights` method whose icon is shown in Figure P.28 and whose block diagram is shown in Figure P.29. The block diagram is nearly identical to that shown in Figure P.20, where the only difference is in which lights are lit. The class libraries for the other seven signals are similar and will not be reproduced here.

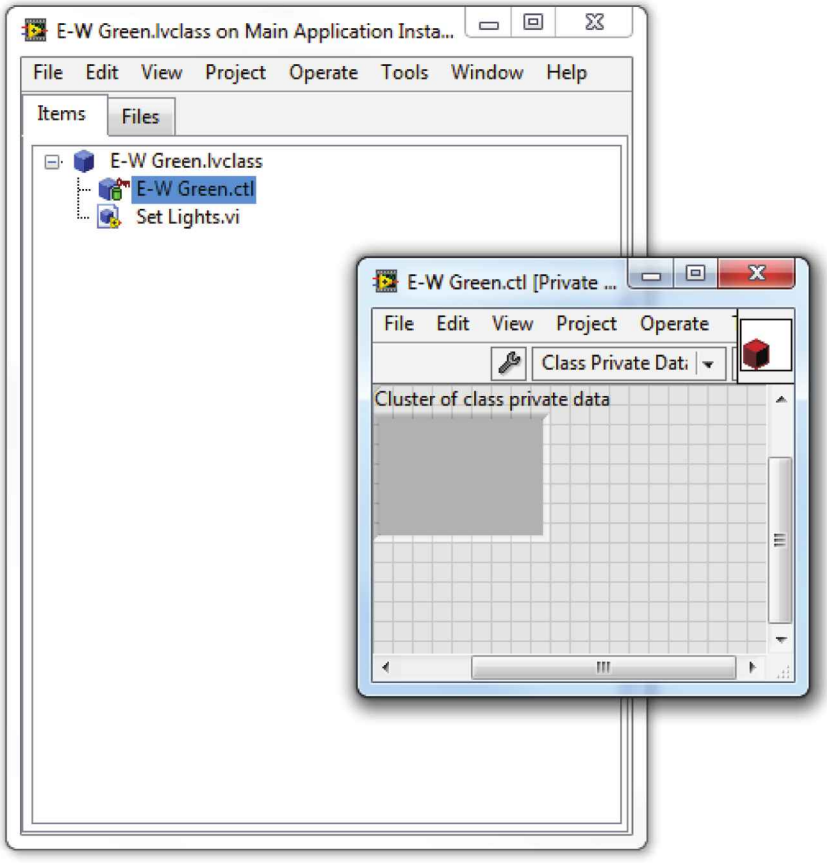


Figure P.27: The private data and class library of the `E-W Green` class

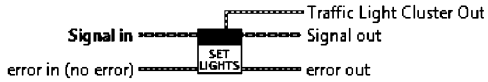


Figure P.28: The icon of the `Set Lights` override method of the `E-W Green` class

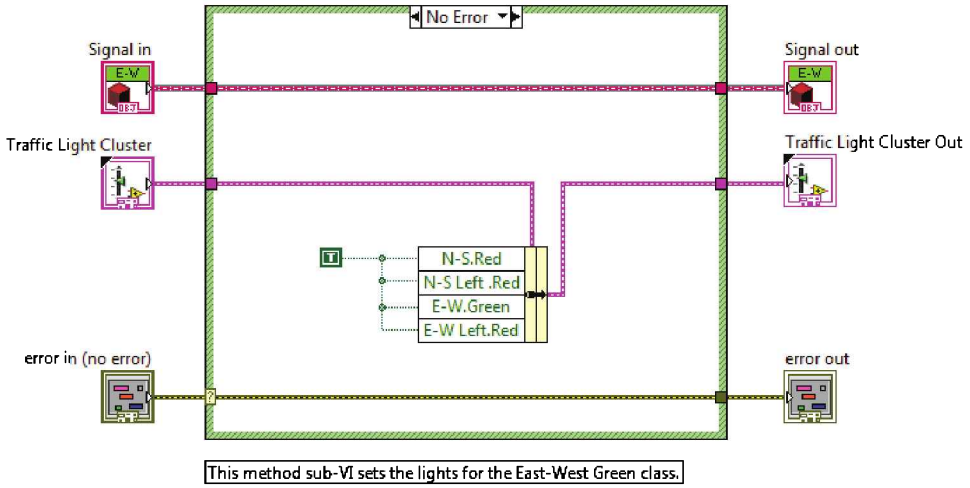


Figure P.29: The block diagram of the `Set Lights` override method of the `E-W Green` class

Exercise 10.11

The `Get Signal Name` sub-VI accepts a `Signal` object or one of its children as an input and outputs a string containing the name of the object. Figure P.30 shows one way to do this. First, the path to the object is found using the **`Get LV Class Path`** function. This function expects a LabVIEW Object as an input, so the `Signal` object is upcast to the LabVIEW Object using the **`To More Generic Class`** function to avoid a coercion dot at the input terminal of the **`Get LV Class Path`** function. The path to the class is then parsed using the **`Get File Extension`** function. This function returns the name of the class without the `*.lvclass` extension. This is the name of the object.

Exercise 10.12

The `Next Signal Object` sub-VI uses the factory pattern to generate the next signal object when the current signal has been completed. It takes the completed signal and the cluster of vehicle sensors as inputs. Figure P.31 shows the block diagram for the `N-S Green` case. This signal always transitions to the `N-S Yellow` signal.

First, the name of the current completed signal is extracted using the `Get Signal Name` sub-VI developed in Exercise 10.11. The name string is then used as an input to a case structure. When the proper next signal object has been generated, it is initialized using the `Initialize Signal` method of the `Signal` class.

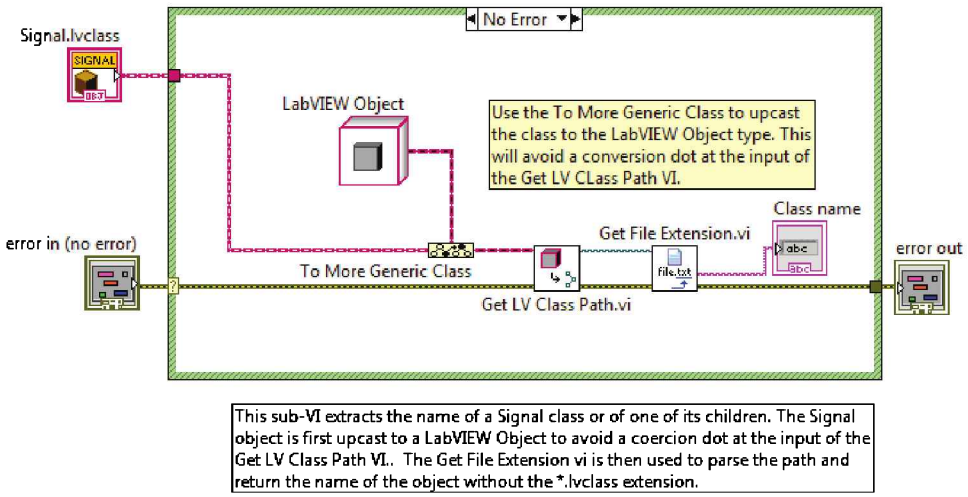


Figure P.30: The block diagram of the `Get Signal Name` sub-VI

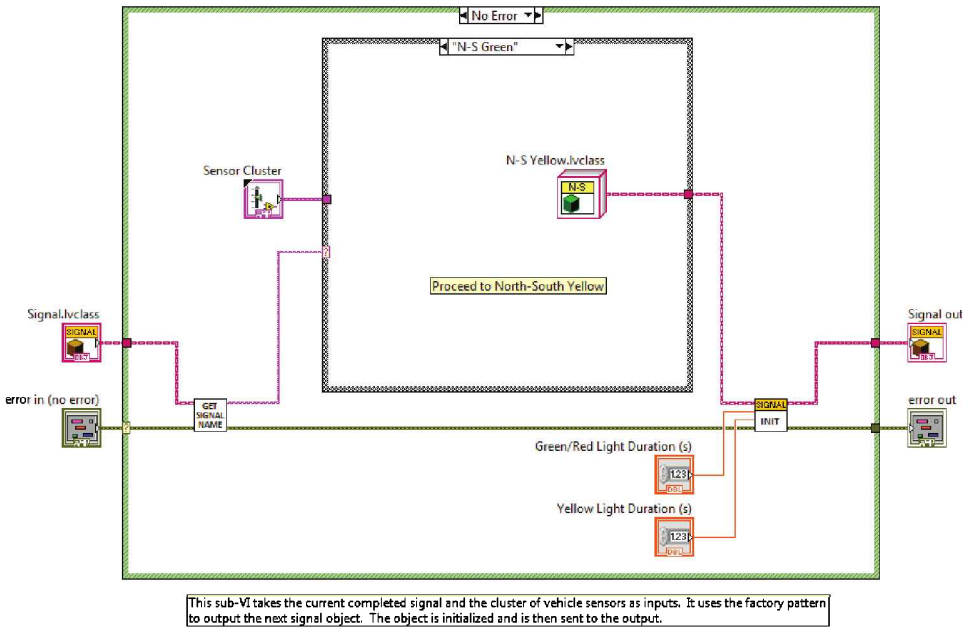


Figure P.31: The `N-S Green` case of the `Next Signal Object` sub-VI

Figure P.32 shows the block diagram for the `N-S Yellow` case. The vehicle sensors need to be consulted to determine the next signal in this case. The figure shows the case when no vehicle is present in the turn lane, and Figure P.33 shows the case when a vehicle is present in the turn lane.

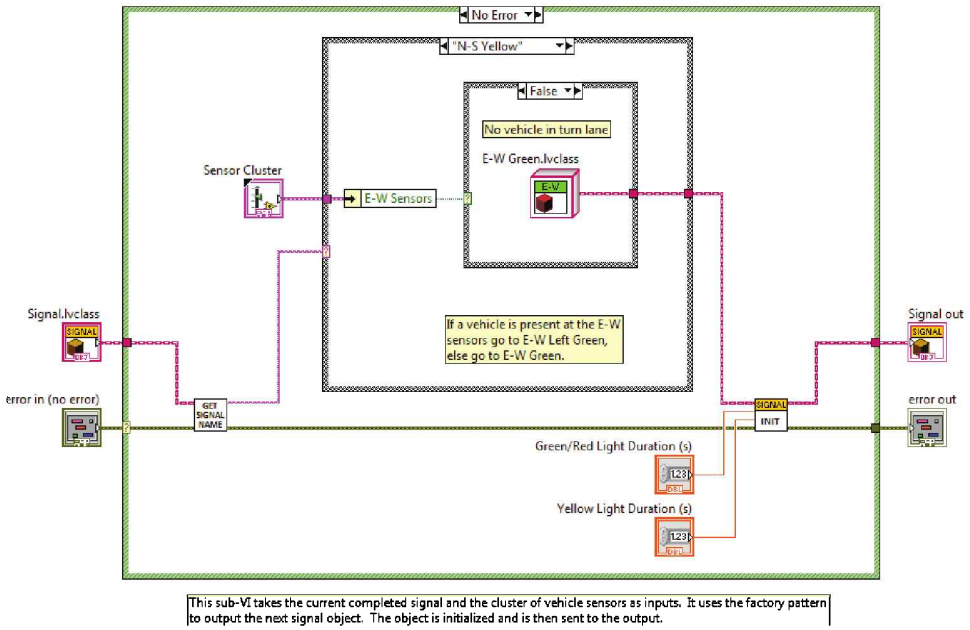


Figure P.32: The `N-S Yellow` case of the `Next Signal Object` sub-VI when no vehicle is present in the turn lane

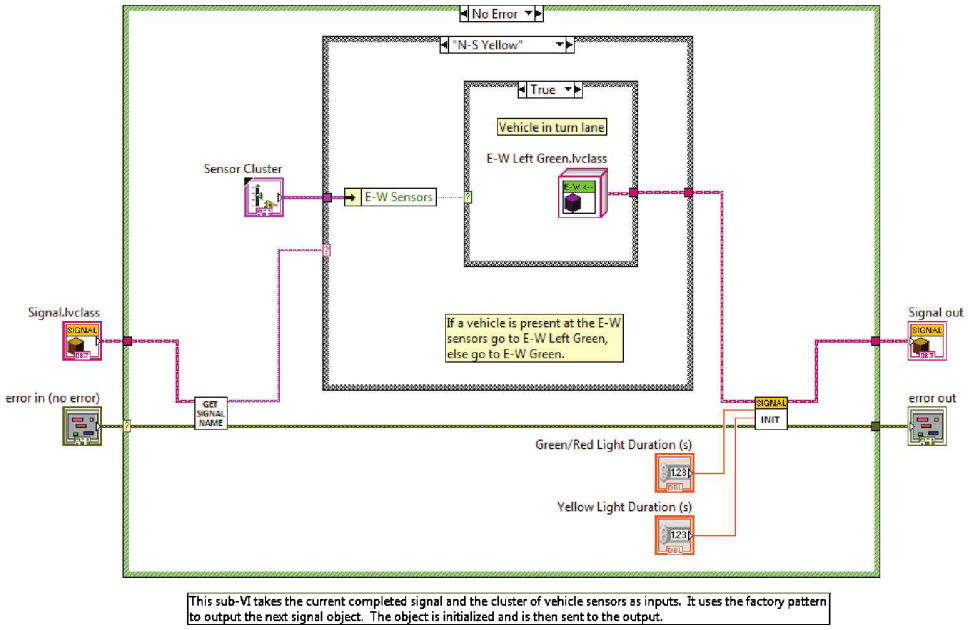


Figure P.33: The `N-S Yellow` case of the `Next Signal Object` sub-VI when a vehicle is present in the turn lane

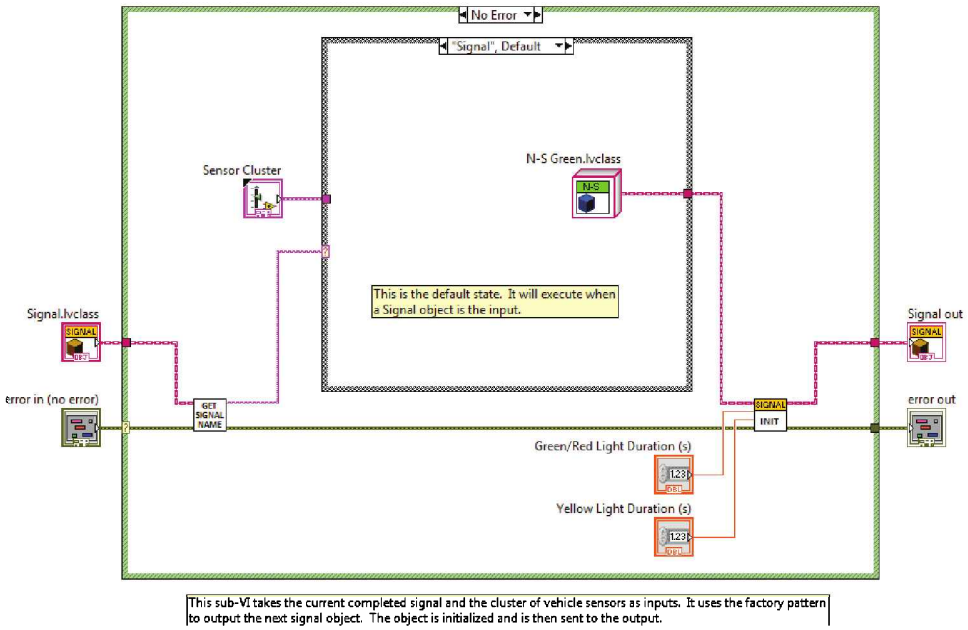


Figure P.34: The default case of the `Next Signal Object` sub-VI

Since a string is wired to the selector of the case structure, a default case is necessary. This is shown in Figure P.34. The factory will return a `N-S Green` object if the input object is not recognized.

Exercise 10.13

The overall structure and *Initialize* state of the traffic light controller with the `Signal` class ish shown in Figure P.35. A `Signal` object is used to initialize the shift register on the data highway, allowing the data highway to carry any of the child classes. The `Signal` object is then passed to the `Next Signal Object` factory sub-VI. Recall that, when a `Signal` object is passed to this sub-VI, it generates a `N-S Green` object, which is the first signal in the traffic light sequence.

The controller then transitions to the *Signal* state shown in Figure P.36. The dynamically dispatched `Set Lights` method creates the proper light pattern for the given `Signal` object. A time check is performed using the statically dispateched `Auto Reset Time Check` method. If the current signal is not completed, the `Signal` object is placed back on the data highway. If the current signal is completed, as shown in Figure P.37, the vehicle sensors are bundled into a cluster and are passed to the `Next Signal Object` along with the current signal to generate the next child `Signal` object.

When the user presses the STOP button or if an error is detected, the state machine transitions to the *Shutdown* state shown in Figure P.38. This concludes the traffic light controller with the `Signal` state.

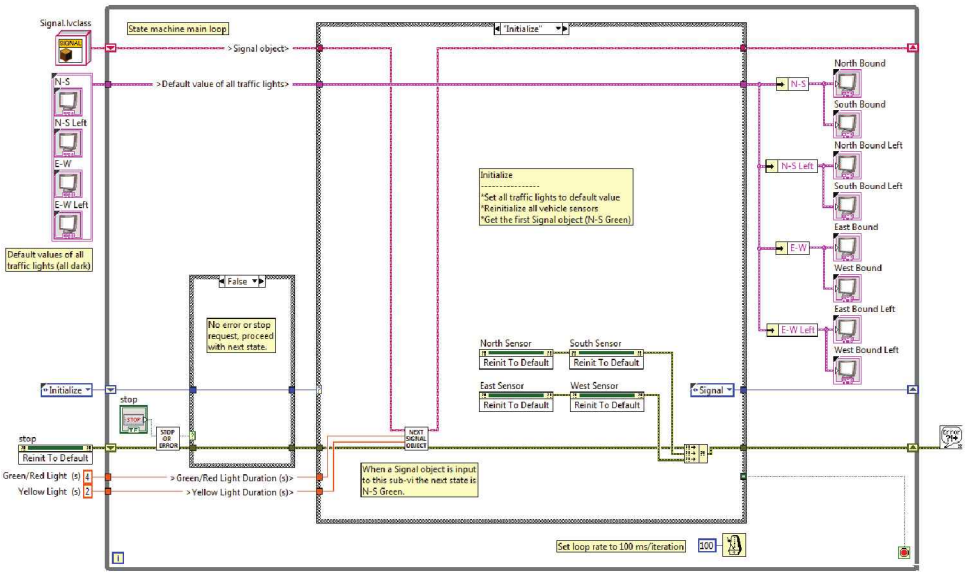


Figure P.35: The *Initialize* state of the traffic light controller with the `Signal` class

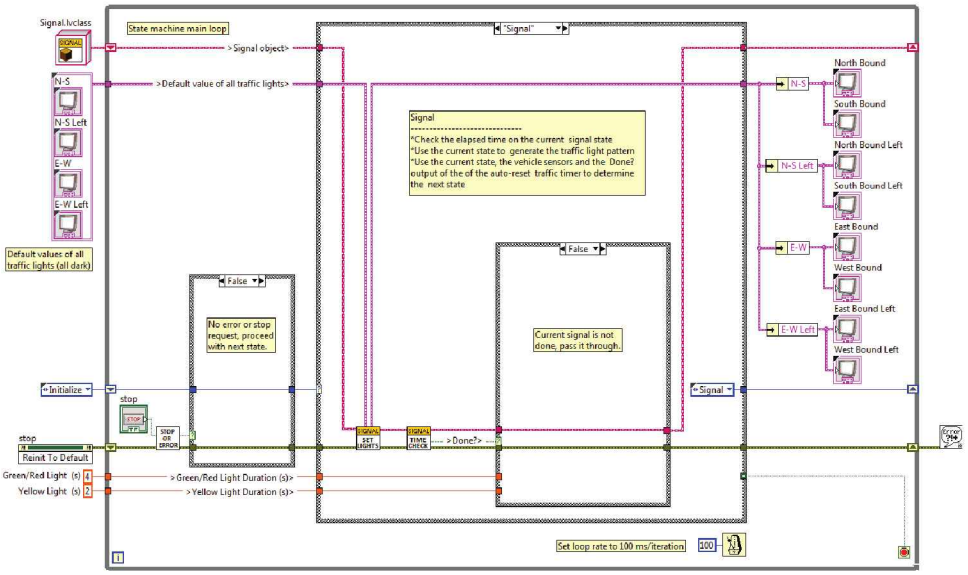


Figure P.36: The *Signal* state of the traffic light controller with the `Signal` class, signal not done

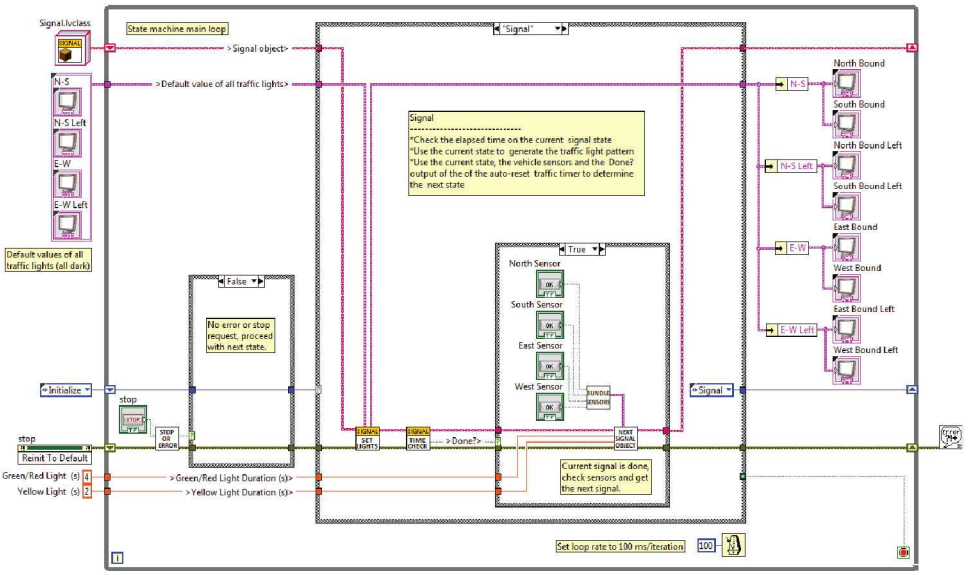


Figure P.37: The *Signal* state of the traffic light controller with the `Signal` class, signal done

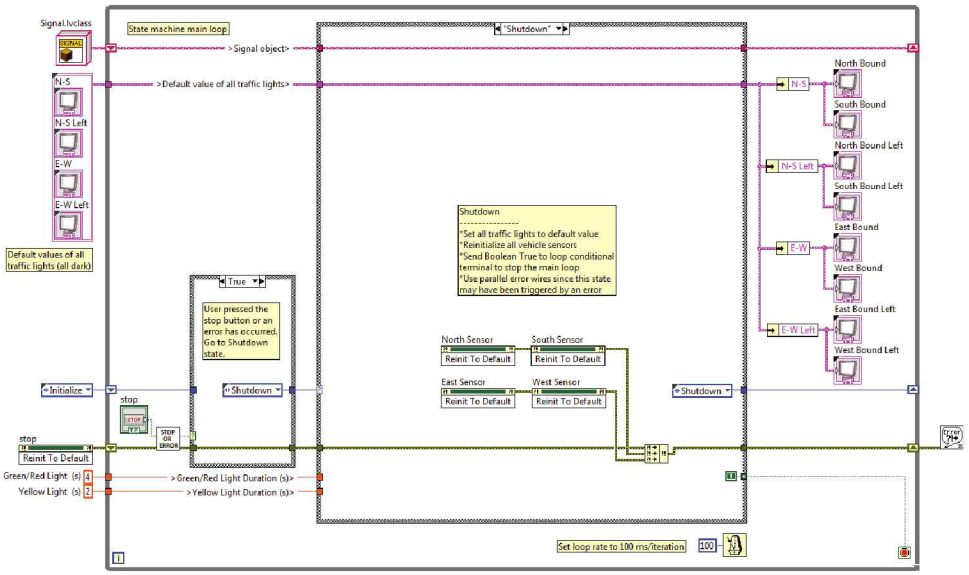


Figure P.38: The *Shutdown* state of the traffic light controller with the `Signal` class

Exercise 10.14

The block diagram of the `Read Done` method is shown in Figure P.39, and the block diagram of the `Write Done` method is shown in Figure P.40. Both methods use the static dispatch template. The read method unbundles the DONE Boolean from the

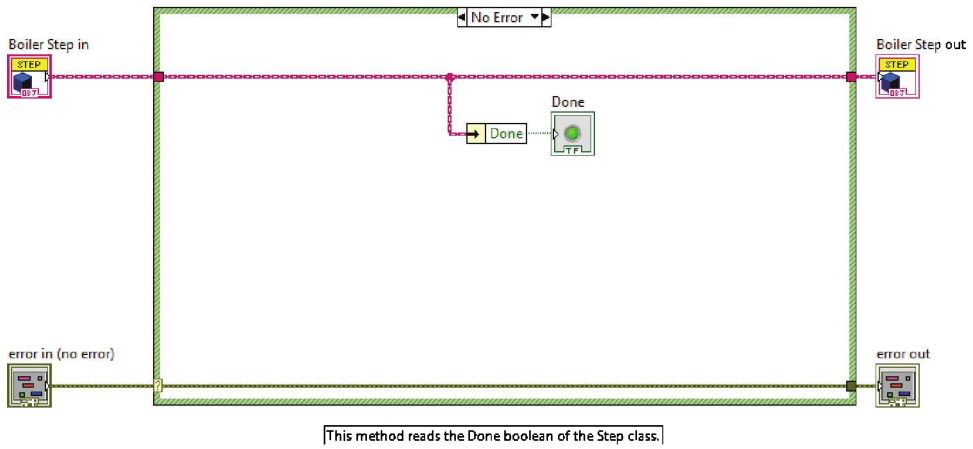


Figure P.39: The block diagram of the `Read Done` method of the `Step` class

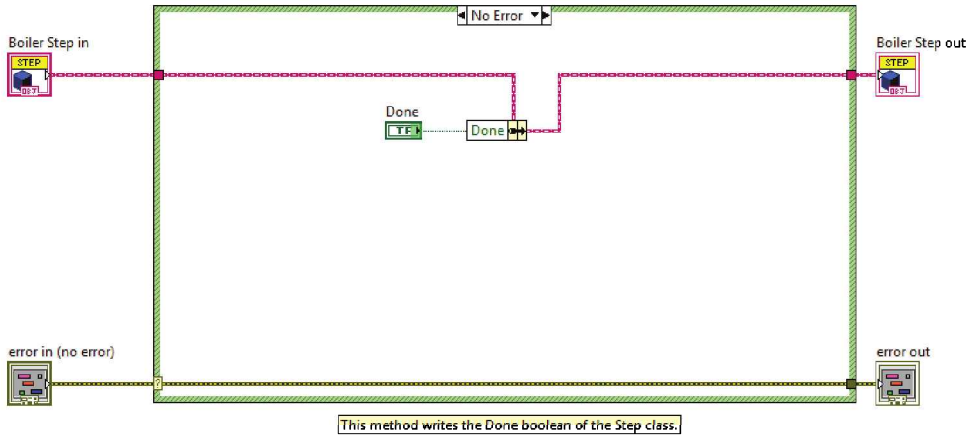


Figure P.40: The block diagram of the `Write Done` method of the `Step` class

private data and displays it, while the write method bundles the new value of the Boolean into the private data.

Exercise 10.15

The dynamic dispatch methods of the `Step` class are merely placeholders with the correct icon and connector pane but no code (also known as "stubs"). The block diagram of the `Step Initialize` method is shown in Figure P.41. This method accepts a `Step` object as an input. It also accepts the log file path, the reference to the PILOT button, and the array of arguments parsed from the queued string. The method returns an initialized `Step` object.

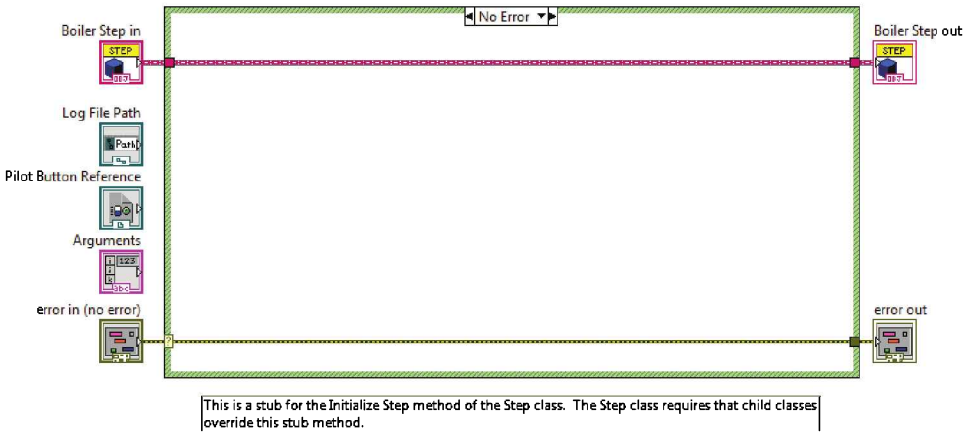


Figure P.41: The block diagram of the `Initialize Step` method of the `Step` class

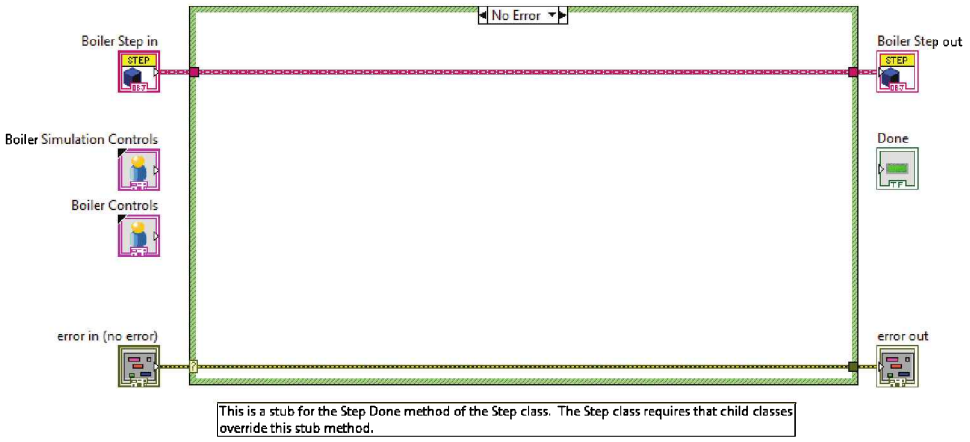


Figure P.42: The block diagram of the `Step Done` method of the `Step` class

Wiring Tip: The front panel control for the PILOT button references can be found in the *Refnum...Control Refnum* sub-palette. Since the *Blinking* property is available for all controls, we do not need to cast the control specifically to the Boolean type.

The block diagram of the `Step Done` method is shown in Figure P.42. This method accepts a `Step` object as an input. It also accepts all of the volatile inputs that change as the program runs: the BOILER CONTROLS cluster and the BOILER SIMULATION CONTROLS cluster. This method decides if the current step is finished and sets the DONE Boolean output appropriately. The method also outputs the modified `Step` object.

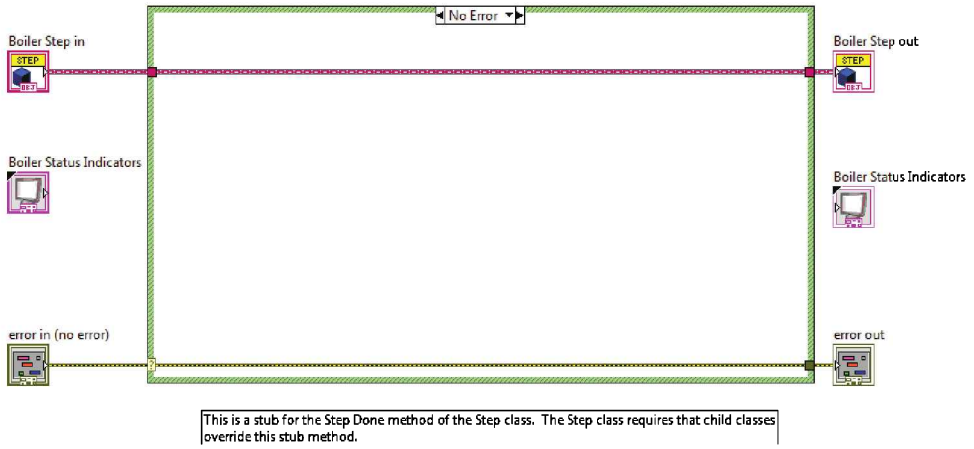


Figure P.43: The block diagram of the `Step Output` method of the `Step` class

The block diagram of the `Step Output` method is shown in Figure P.43. This method accepts a `Step` object as an input. It creates and outputs the status cluster for display by the main application. It will also write to the log file if necessary.

These three methods are stubs with no code. The child classes are expected to override these stubs with methods of their own. In order to force the child classes to do this, right-click on the *Step.lvclass* entry in the class library and bring up the *Properties* menu. Navigate to the *Item Settings* menu, and select one of the dynamic dispatch methods. Click on the `Require descendant classes to override this dynamic dispatch VI` checkbox to force the child classes to override this method as shown in Figure P.44. Repeat this process for the other two dynamic dispatch methods in the class.

Exercise 10.16

The block diagram of the `Conditional Write To Log File` sub-VI is shown in Figure P.45. If the STRING CHANGED? Boolean is True, the log string is formatted using the **`Format Into String`** function and is then written to the boiler log file using the `Append To Text File` sub-VI shown in Figure 1.45.

Exercise 10.17

The icon of the `Initialize Step` override method of the `Application Run` class is shown in Figure P.46, and the block diagram is shown in Figure P.47. This method VI writes the log file path and an empty log string to the private data of an `Application Run` object. The DONE Boolean of the parent `Step` class is then set to False using the `Write Done` method of the `Step` class.

The icon of the `Step Done` override method of the `Application Run` class is shown in Figure P.48 and the block diagram is shown in Figure P.49. The logic on the block diagram is the same as that shown in Figure G.14, but it has an

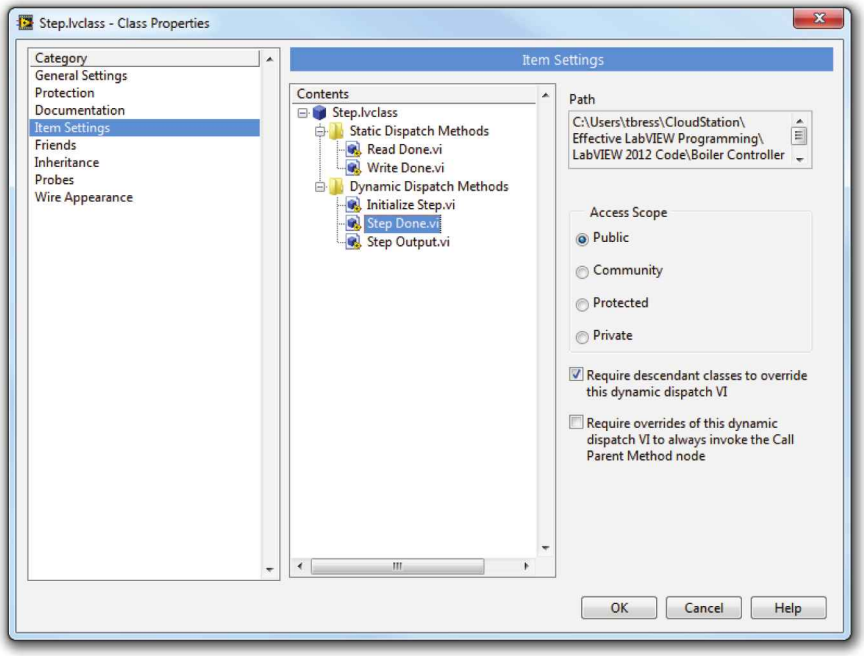


Figure P.44: The descendants can be forced to override the dynamically dispatched methods

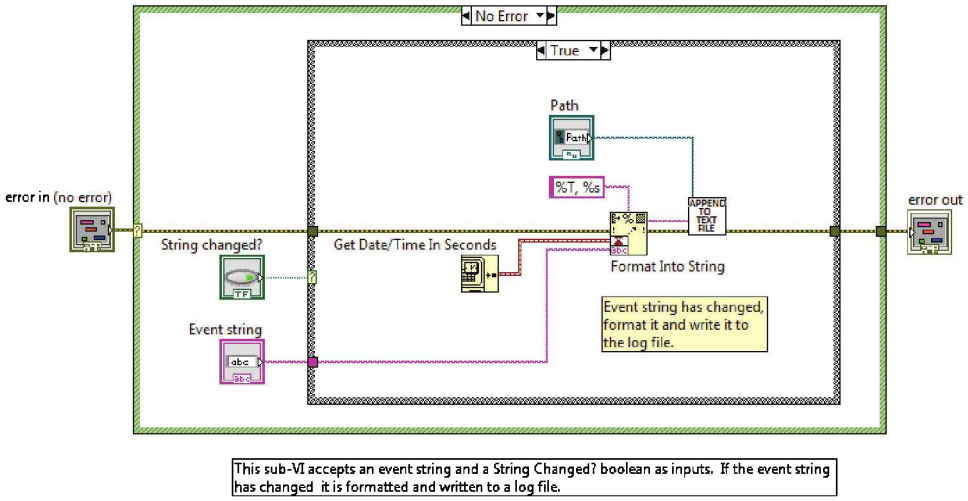


Figure P.45: The block diagram of the `Conditional Write To Log File` sub-VI

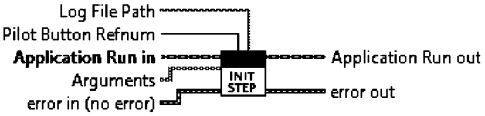


Figure P.46: The icon of the `Initialize Step` method of the `Application Run` class

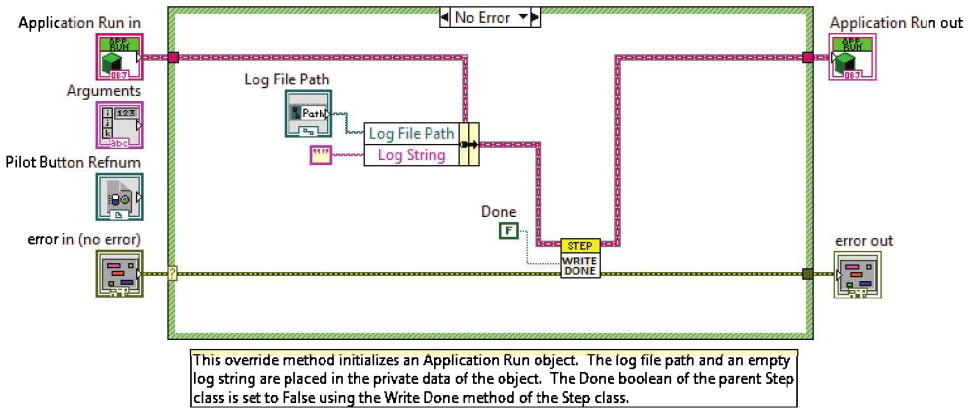


Figure P.47: The block diagram of the `Initialize Step` method of the `Application Run` class

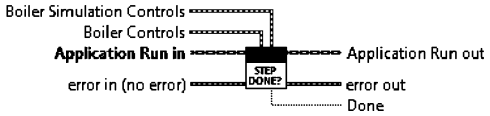


Figure P.48: The icon of the `Step Done` method of the `Application Run` class

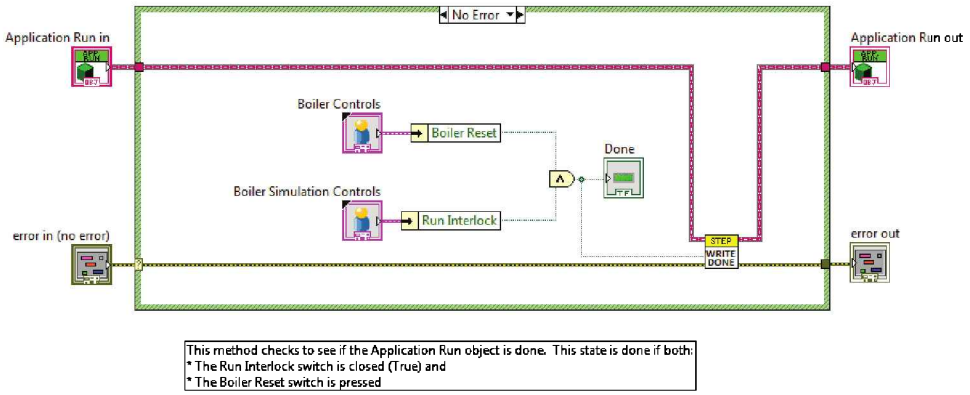


Figure P.49: The block diagram of the `Step Done` method of the `Application Run` class

important addition. The value of the DONE Boolean is written to the private data of the parent `Step` class using the `Write Done` method of the `Step` class.

The icon of the `Step Output` override method of the `Application Run` class is shown in Figure P.50, and the block diagram is shown in Figure P.51. The old log string is unbundled from the private data of the `Application Run` object and is

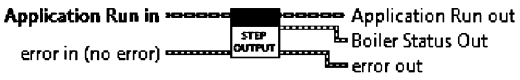


Figure P.50: The icon of the `Step Output` method of the `Application Run` class

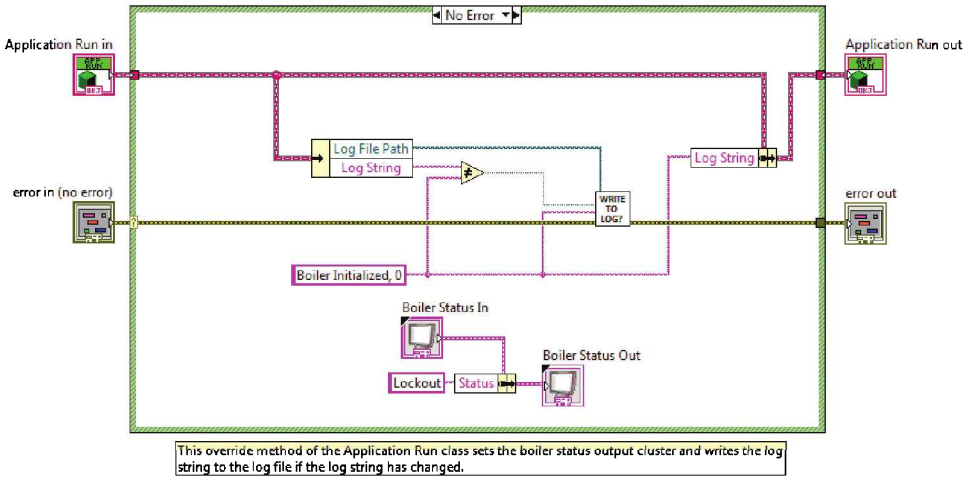


Figure P.51: The block diagram of the `Step Output` method of the `Application Run` class

compared with the new log string. The result of the comparison and the new log string are both passed to the `Conditional Write To Log File` sub-VI developed in Exercise 10.16. The path to the log file is also unbundled from the private data of the object and is then passed to the `Conditional Write To Log File` sub-VI as an input. The new log file is then bundled into the private data of the object, overwriting the old log string. This method also bundles a new status string into a default value cluster of the BOILER STATUS indicator and outputs it for display by the main application.

Exercise 10.18

The icon of the `Initialize Step` override method of the `Reset Lockout` class is shown in Figure P.52, and the block diagram is shown in Figure P.53. This method VI writes the log file path and an empty log string to the private data of an `Reset Lockout` object. The DONE Boolean of the parent `Step` class is then set to False using the `Write Done` method of the `Step` class.

The icon of the `Step Done` override method of the `Reset Lockout` class is shown in Figure P.54, and the block diagram is shown in Figure P.55. The logic on the block diagram is the same as that shown in Figure G.16. The value of the DONE Boolean is written to the private data of the parent `Step` class using the `Write Done` method of the `Step` class.

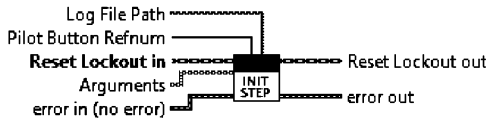


Figure P.52: The icon of the `Initialize Step` method of the `Reset Lockout` class

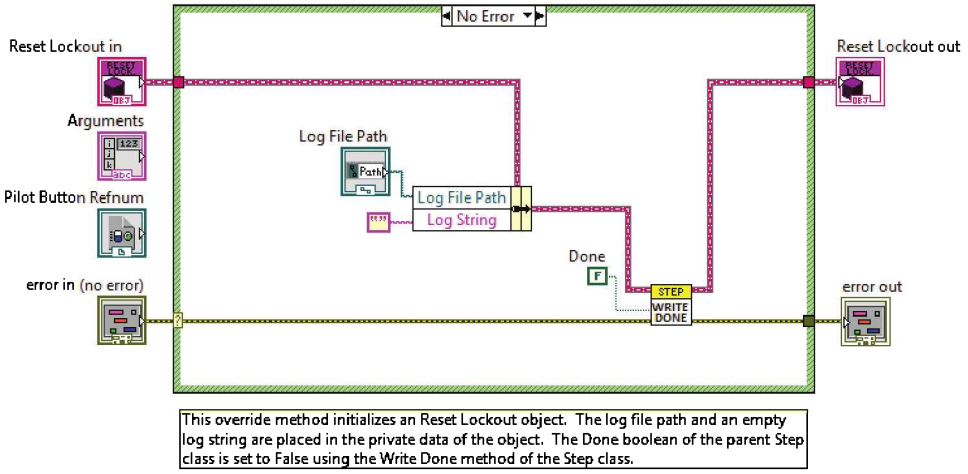


Figure P.53: The block diagram of the `Initialize Step` method of the `Reset Lockout` class

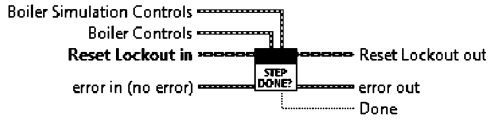


Figure P.54: The icon of the `Step Done` method of the `Reset Lockout` class

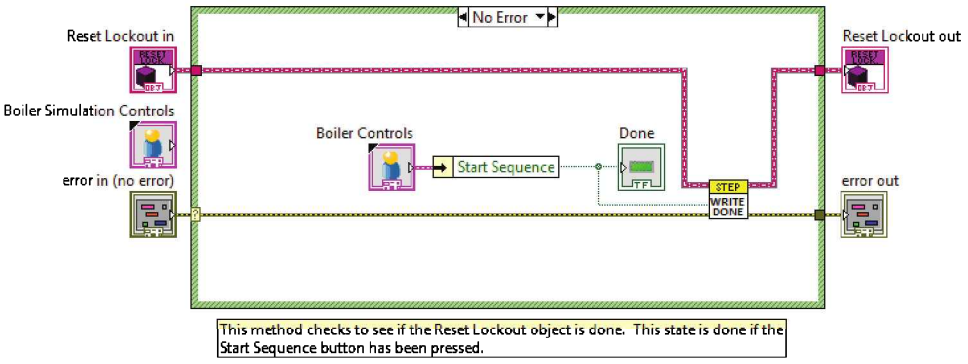


Figure P.55: The block diagram of the `Step Done` method of the `Reset Lockout` class

The icon of the `Step Output` override method of the `Reset Lockout` class is shown in Figure P.56, and the block diagram is shown in Figure P.57. This override method is nearly identical to that of the `Application Run` class. The only difference is in the contents of the log string and the status string.



Figure P.56: The icon of the `Step Output` method of the `Reset Lockout` class

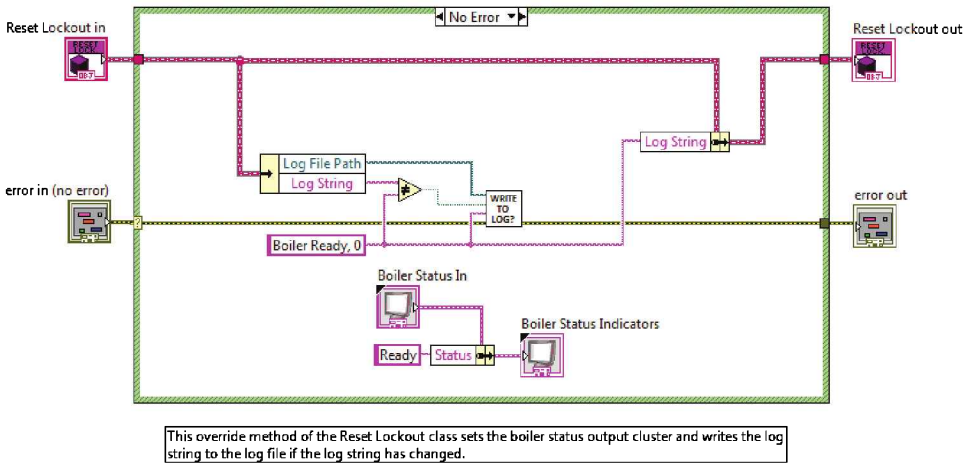


Figure P.57: The block diagram of the `Step Output` method of the `Reset Lockout` class

Exercise 10.19

The icon of the `Initialize Step` override method of the `Pre-Purge Timer` class is shown in Figure P.58, and the block diagram is shown in Figure P.59. This method VI writes the log file path to the private data of an `Pre-Purge Timer` object. The log file string will always be unique in this step because it contains the elapsed time so there is no need to store the old log string to see if it has changed.

The *Pre-Purge time (s)* constant is parsed from the array of arguments and is bundled to the private data of the object. The `Timer` object is unbundled from the private data of the object and is initialized using the `Initialize Timer` method of the `Timer` class. The initialized timer is then bundled back into the private data of the

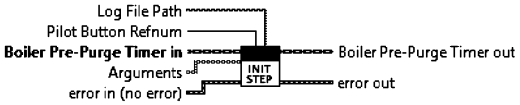


Figure P.58: The icon of the `Initialize Step` method of the `Pre-Purge Timer` class

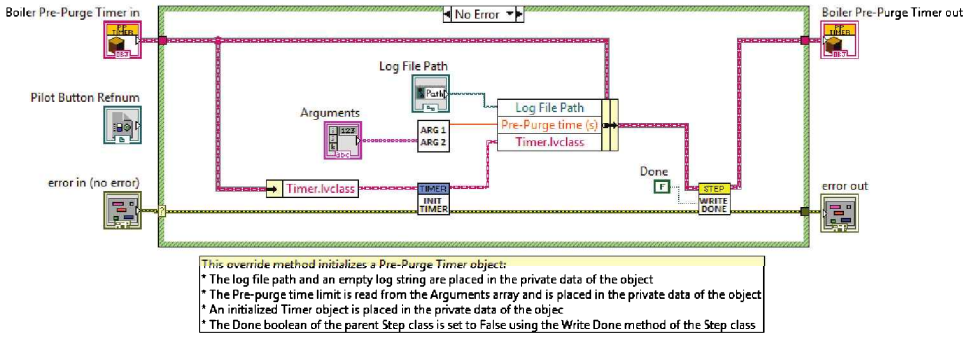


Figure P.59: The block diagram of the `Initialize Step` method of the `Pre-Purge Timer` class

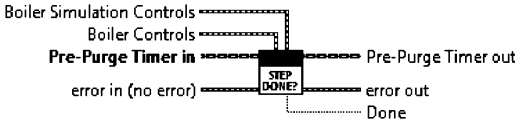


Figure P.60: The icon of the `Step Done` method of the `Pre-Purge Timer` class

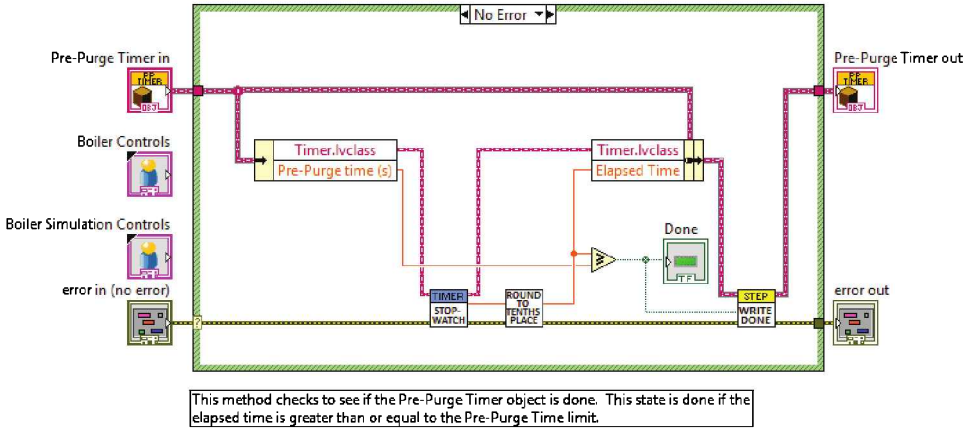


Figure P.61: The block diagram of the `Step Done` method of the `Pre-Purge Timer` class

object. The DONE Boolean of the parent `Step` class is then set to False using the `Write Done` method of the `Step` class.

The icon of the `Step Done` override method of the `Pre-Purge Timer` class is shown in Figure P.60, and the block diagram is shown in Figure P.61. The logic on the block diagram is the same as that shown in Figure G.19. Note that the elapsed time is written to the private data of the object for use in the `Step Output` method. The value of the DONE Boolean is written to the private data of the parent `Step` class using the `Write Done` method of the `Step` class.

The icon of the `Step Output` override method of the `Pre-Purge Timer` class is shown in Figure P.62, and the block diagram is shown in Figure P.63. The status cluster is updated using the elapsed time as an input. The log file string is then constructed. The log string changes depending on whether the current step is done or not, so the DONE Boolean of the parent class is read using the `Read Done` method of the `Step` class. The rounded elapsed time is also incorporated into the log string using the **`Format Into String`** function. Incorporating the elapsed time ensures that the log string is always unique, so no uniqueness check needs to be made. A Boolean True constant is wired to the *String Changed?* input of the `Conditional Write To Log File` sub-VI to ensure that the string is written to the log file.



Figure P.62: The icon of the `Step Output` method of the `Pre-Purge Timer` class

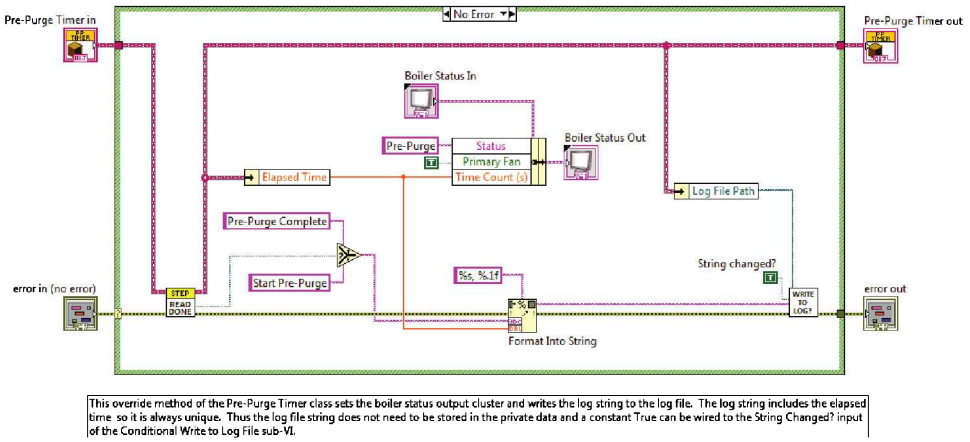


Figure P.63: The block diagram of the `Step Output` method of the `Pre-Purge Timer` class

Exercise 10.20

The icon of the `Initialize Step` override method of the `Pre-Purge` class is shown in Figure P.64, and the block diagram is shown in Figure P.65. This method VI writes the reference to the PILOT button to the private data of an `Pre-Purge` object. The

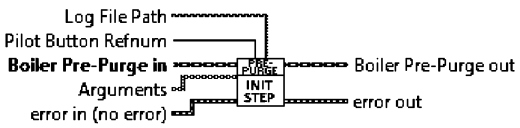


Figure P.64: The icon of the `Initialize Step` method of the `Pre-Purge` class

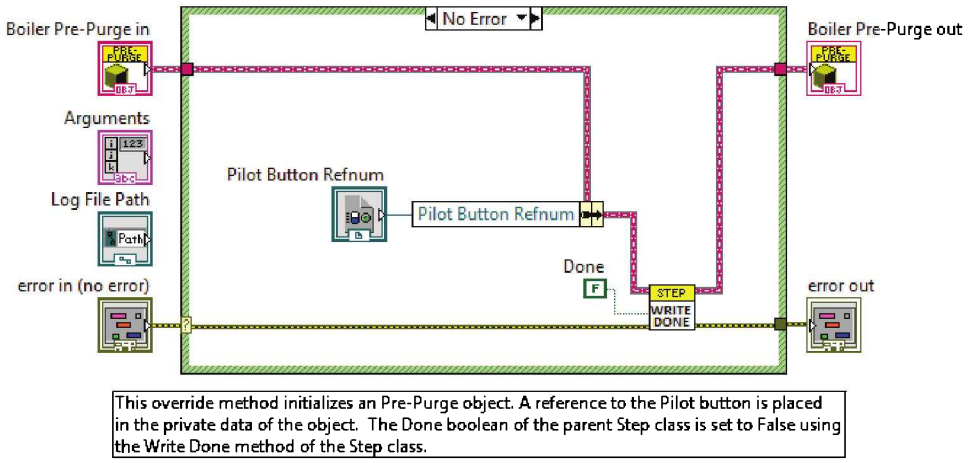


Figure P.65: The block diagram of the `Initialize Step` method of the `Pre-Purge` class

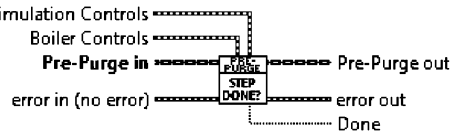


Figure P.66: The icon of the `Step Done` method of the `Pre-Purge` class

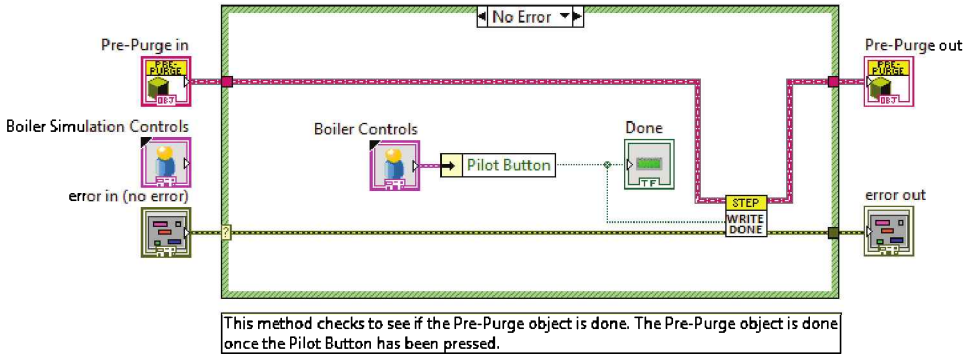


Figure P.67: The block diagram of the `Step Done` method of the `Pre-Purge` class

DONE Boolean of the parent class is also set to False using the `Write Done` method of the `Step` class.

The icon of the `Step Done` override method of the `Pre-Purge` class is shown in Figure P.66, and the block diagram is shown in Figure P.67. The logic on the block diagram is the same as that shown in Figure G.21. The value of the DONE Boolean is written to the private data of the parent `Step` class using the `Write Done` method of the `Step` class.



Figure P.68: The icon of the `Step Output` method of the `Pre-Purge` class

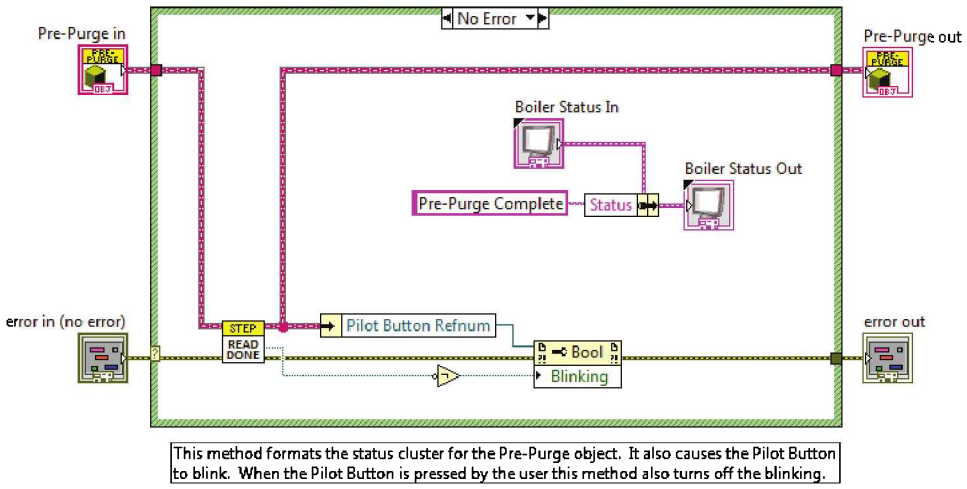


Figure P.69: The block diagram of the `Step Output` method of the `Pre-Purge` class

The icon of the `Step Output` override method of the `Pre-Purge` class is shown in Figure P.68, and the block diagram is shown in Figure P.69. The status cluster is updated in this method. The reference to the PILOT button is unbundled from the private data and is used to make the PILOT button blink using the *Blinking* property node. The value of the DONE Boolean is read using the `Read Done` method of the parent class, and the logical negation of this value is written to the *Blinking* property node. In this way, the button blinks while its value is False (button not pressed), and it stops blinking when the button is True (button is pressed).

The *Blinking* property node was placed in the `Step Output` method, because this method already interacts with the user interface by setting the value of the status cluster. Making the button blink has nothing to do with determining whether the step is done or not, so it makes more sense to include this functionality in the `Step Output` method. Putting the functionality in the `Step Done` method would decrease the cohesion of that method.

Exercise 10.21

The icon of the `Initialize Step` override method of the `Ignition` class is shown in Figure P.70, and the block diagram is shown in Figure P.71. This method VI writes the log file path and an empty log string to the private data of an `Ignition` object. The *Flame Sensor Low* constant is parsed from the array of arguments and is placed in

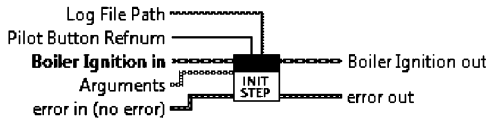


Figure P.70: The icon of the `Initialize Step` method of the `Ignition` class

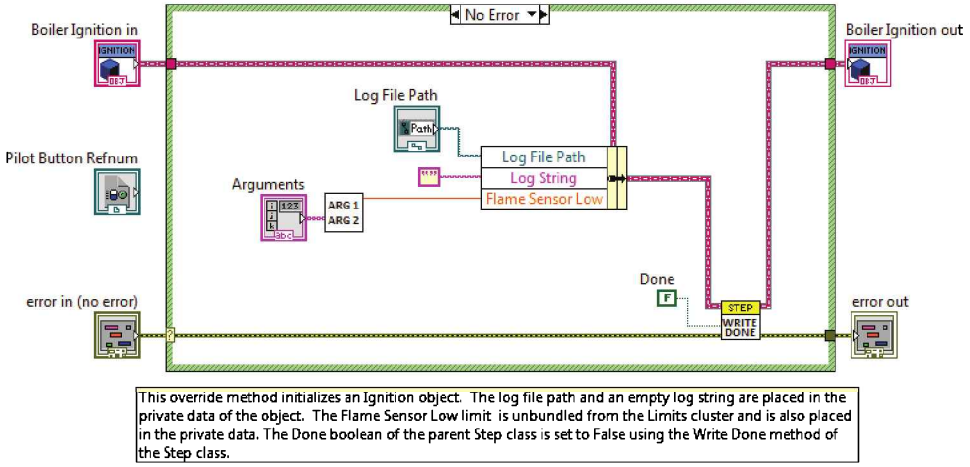


Figure P.71: The block diagram of the `Initialize Step` method of the `Ignition` class

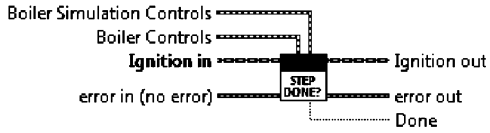


Figure P.72: The icon of the `Step Done` method of the `Ignition` class

the private data. The DONE Boolean of the parent `Step` class is then set to False using the `Write Done` method of the `Step` class.

The icon of the `Step Done` override method of the `Ignition` class is shown in Figure P.72, and the block diagram is shown in Figure P.73. The logic on the block diagram is the same as that shown in Figure G.23. The value of the DONE Boolean is written to the private data of the parent `Step` class using the `Write Done` method of the `Step` class.

The icon of the `Step Output` override method of the `Ignition` class is shown in Figure P.74, and the block diagram is shown in Figure P.75. This override method is nearly identical to that of the `Application Run` class. The only difference is in the contents of the log string and the values written to the status cluster.

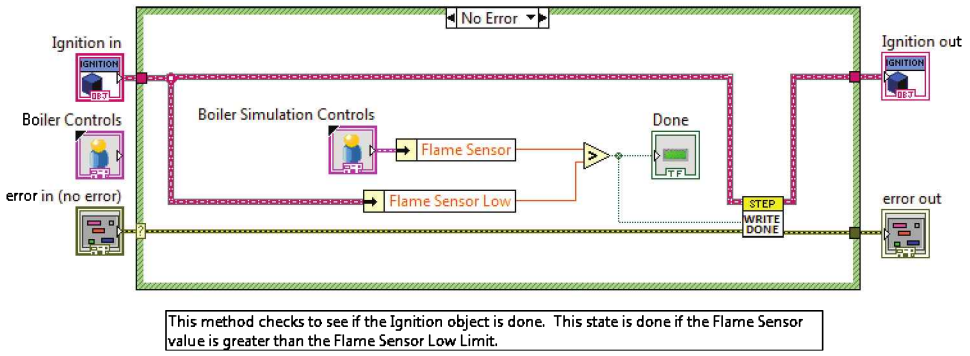


Figure P.73: The block diagram of the `Step Done` method of the `Ignition` class



Figure P.74: The icon of the `Step Output` method of the `Ignition` class

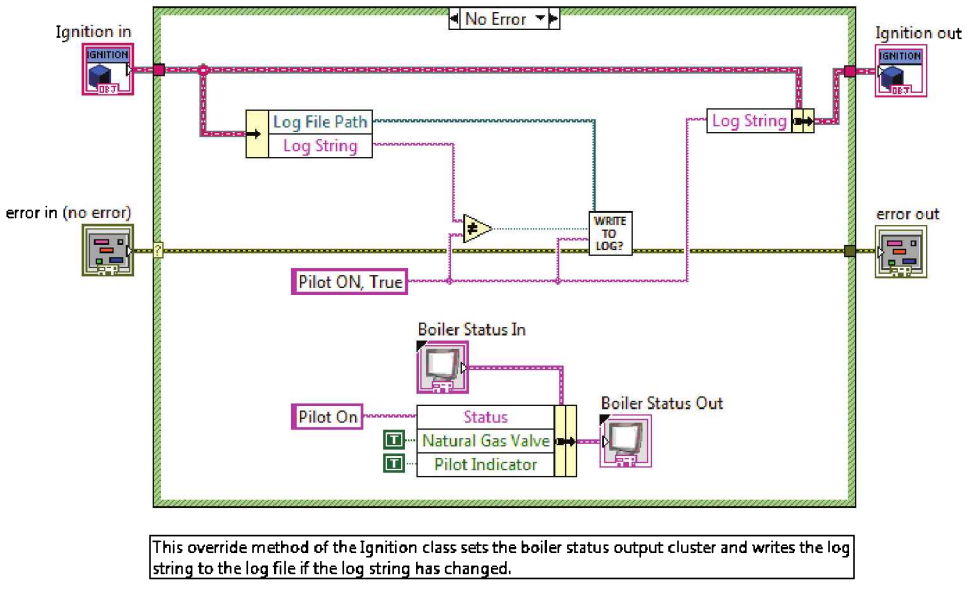


Figure P.75: The block diagram of the `Step Output` method of the `Ignition` class

Exercise 10.22

The icon of the `Initialize Step` override method of the `Prove The Pilot` class is shown in Figure P.76, and the block diagram is shown in Figure P.77. This method VI writes the log file path and an empty log string to the private data of an `Prove`

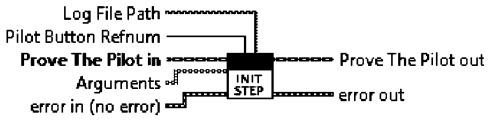


Figure P.76: The icon of the `Initialize Step` method of the `Prove The Pilot` class

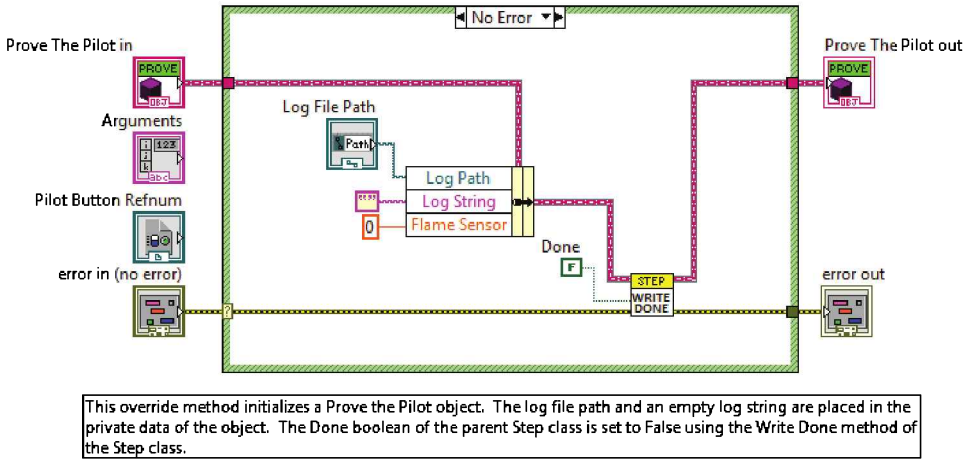


Figure P.77: The block diagram of the `Initialize Step` method of the `Prove The Pilot` class

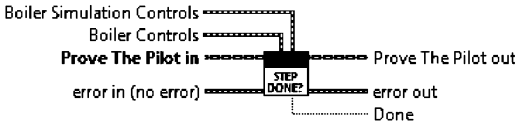


Figure P.78: The icon of the `Step Done` method of the `Prove The Pilot` class

`The Pilot` object. The value of the stored value of the flame sensor is also set to zero. The DONE Boolean of the parent `Step` class is then set to False using the `Write Done` method of the `Step` class.

The icon of the `Step Done` override method of the `Prove The Pilot` class is shown in Figure P.78, and the block diagram is shown in Figure P.79. The logic on the block diagram is the same as that shown in Figure G.25. The current value of the FLAME SENSOR is bundled into the private data. The value of the DONE Boolean is written to the private data of the parent `Step` class using the `Write Done` method of the `Step` class.

The icon of the `Step Output` override method of the `Prove The Pilot` class is shown in Figure P.80, and the block diagram is shown in Figure P.81. This override method is nearly identical to that of the `Application Run` class. The only difference is in the contents of the log string and the values written to the status cluster.

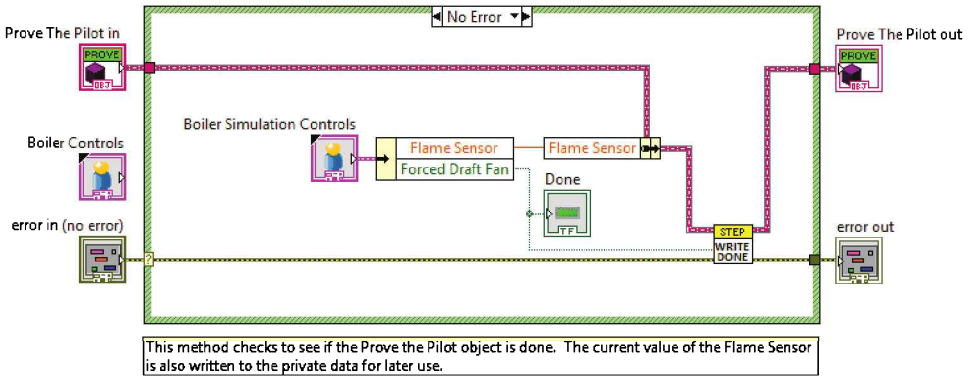


Figure P.79: The block diagram of the `Step Done` method of the `Prove The Pilot` class

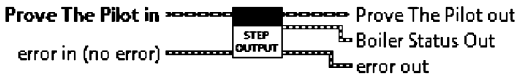


Figure P.80: The icon of the `Step Output` method of the `Prove The Pilot` class

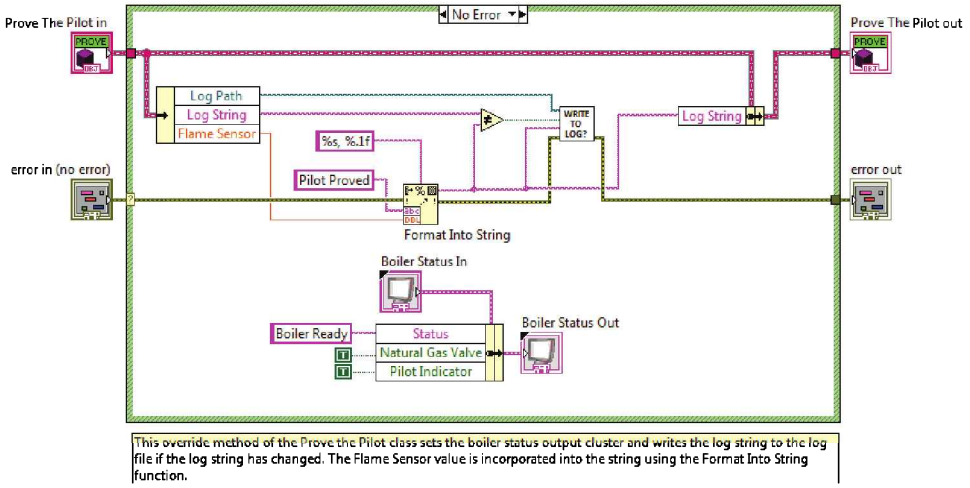


Figure P.81: The block diagram of the `Step Output` method of the `Prove The Pilot` class

Exercise 10.23

The icon of the `Initialize Step` override method of the `Start` class is shown in Figure P.82, and the block diagram is shown in Figure P.83. This method VI writes the log file path and an empty log string to the private data of an `Start` object. The fuel control limits are parsed from the array of arguments and are also placed in the

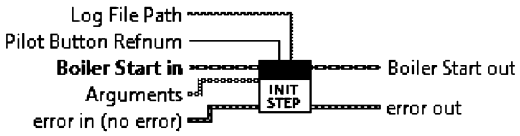


Figure P.82: The icon of the `Initialize Step` method of the `Start` class

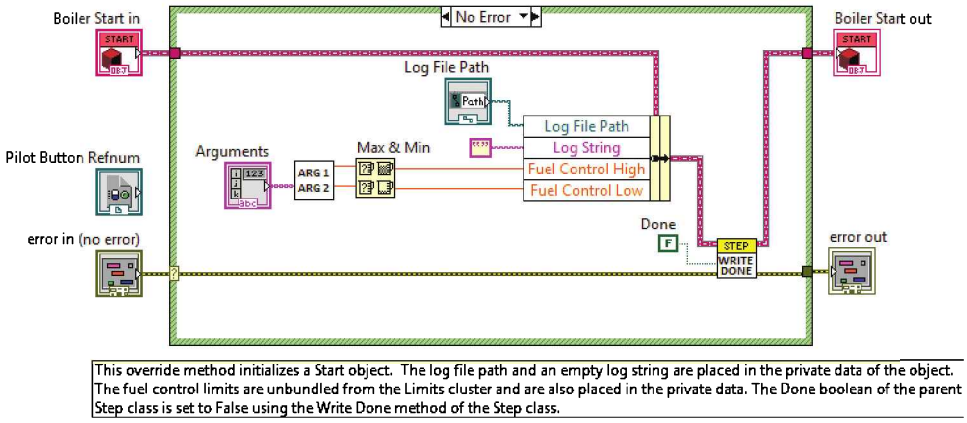


Figure P.83: The block diagram of the `Initialize Step` method of the `Start` class

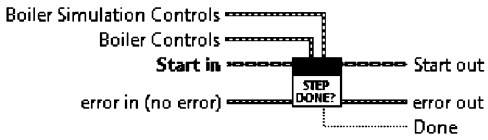


Figure P.84: The icon of the `Step Done` method of the `Start` class

private data. The DONE Boolean of the parent `Step` class is then set to False using the `Write Done` method of the `Step` class.

The icon of the `Step Done` override method of the `Start` class is shown in Figure P.84, and the block diagram is shown in Figure P.85. The logic on the block diagram is the same as that shown in Figure G.27. The value of the DONE Boolean is written to the private data of the parent `Step` class using the `Write Done` method of the `Step` class.

The icon of the `Step Output` override method of the `Start` class is shown in Figure P.86, and the block diagram is shown in Figure P.87. This override method is nearly identical to that of the `Application Run` class. The only difference is in the contents of the log string and the values written to the status cluster.

Exercise 10.24

The icon of the `Initialize Step` override method of the `Run` class is shown in Figure P.88, and the block diagram is shown in Figure P.89. This method VI writes

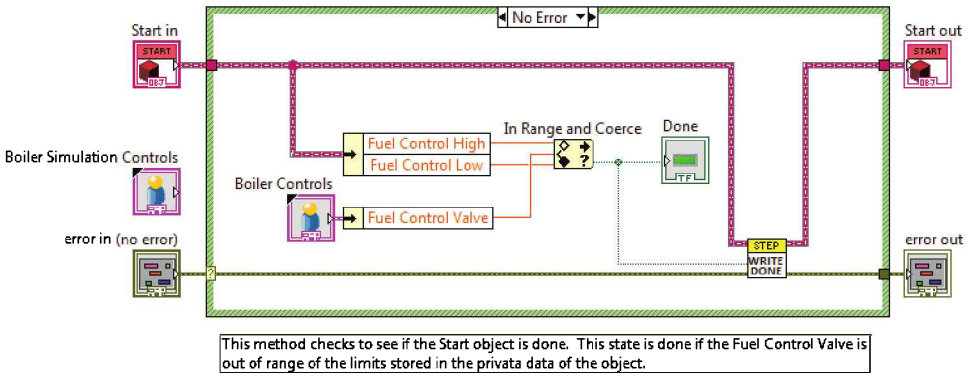


Figure P.85: The block diagram of the `Step Done` method of the `Start` class



Figure P.86: The icon of the `Step Output` method of the `Start` class

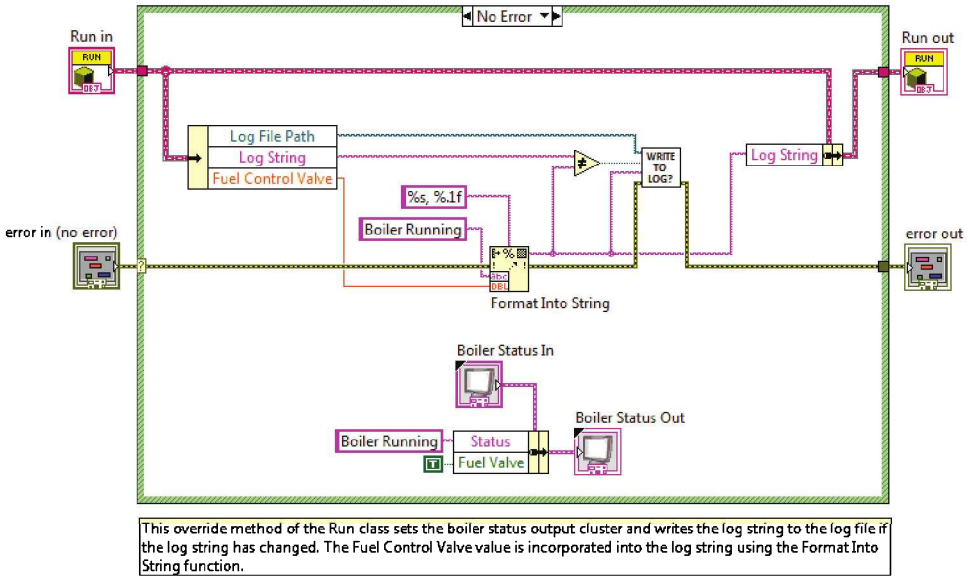


Figure P.87: The block diagram of the `Step Output` method of the `Start` class

the log file path and an empty log string to the private data of an `Run` object. The fuel-control limits are parsed from the array of arguments and are also placed in the private data. The DONE Boolean of the parent `Step` class is then set to False using the `Write Done` method of the `Step` class.

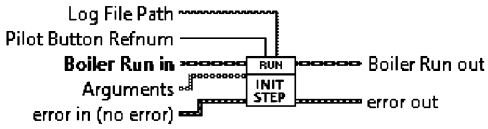


Figure P.88: The icon of the `Initialize Step` method of the `Run` class

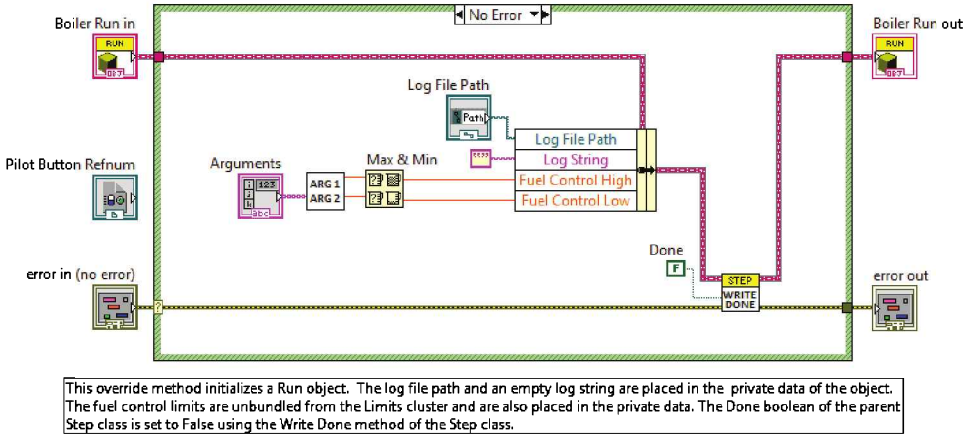


Figure P.89: The block diagram of the `Initialize Step` method of the `Run` class

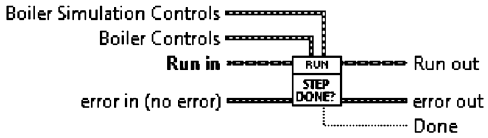


Figure P.90: The icon of the `Step Done` method of the `Run` class

The icon of the `Step Done` override method of the `Run` class is shown in Figure P.90, and the block diagram is shown in Figure P.91. The logic on the block diagram is the same as that shown in Figure G.29. The value of the DONE Boolean is written to the private data of the parent `Step` class using the `Write Done` method of the `Step` class.

The icon of the `Step Output` override method of the `Run` class is shown in Figure P.92, and the block diagram is shown in Figure P.93. The **`Format Into String`** function is used to incorporate the current value of the FUEL CONTROL VALVE into the log string. The log string is compared with the stored log string, and it is written to the log file if it has changed. The status cluster is also updated in this method.

Exercise 10.25

The icon of the `Initialize Step` override method of the `Purge` class is shown in Figure P.94, and the block diagram is shown in Figure P.95. This method VI writes the

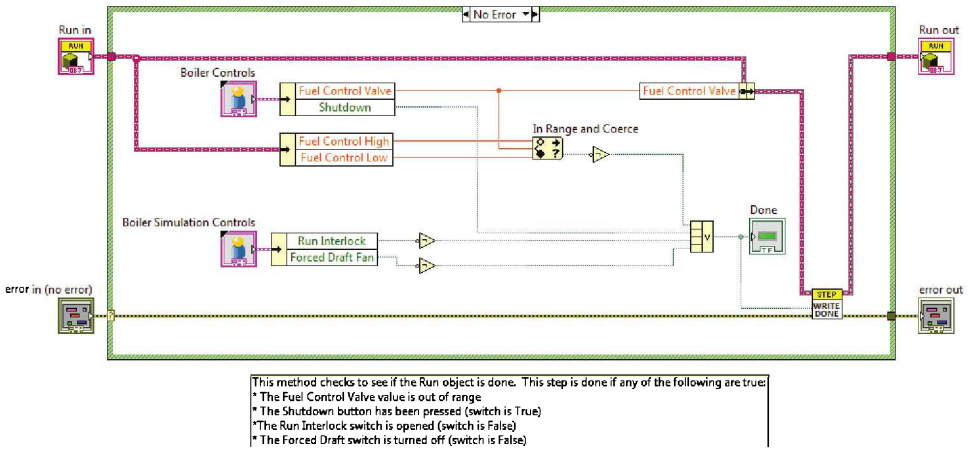


Figure P.91: The block diagram of the `Step Done` method of the `Run` class



Figure P.92: The icon of the `Step Output` method of the `Run` class

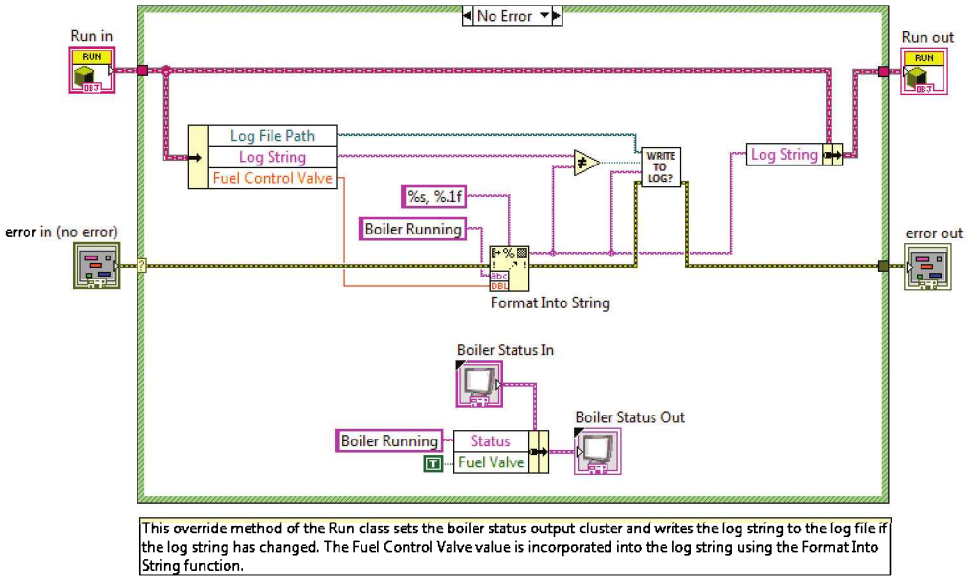


Figure P.93: The block diagram of the `Step Output` method of the `Run` class

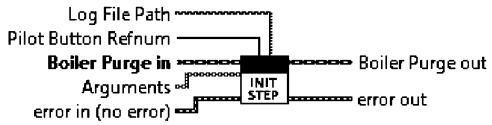


Figure P.94: The icon of the `Initialize Step` method of the `Purge` class

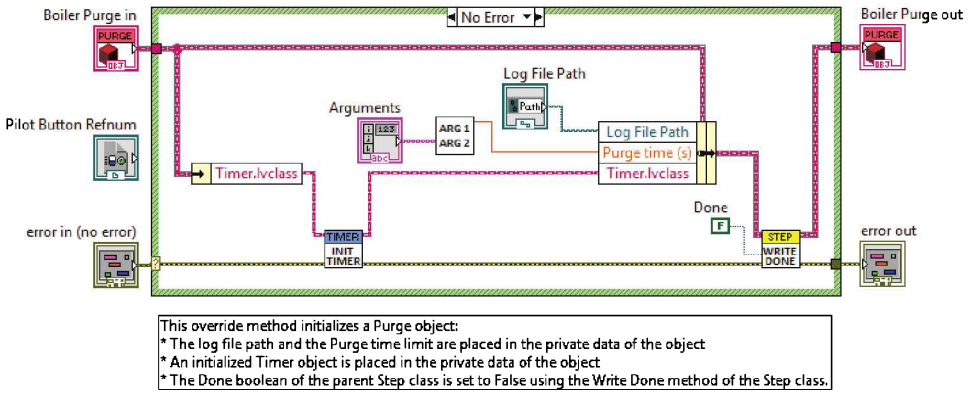


Figure P.95: The block diagram of the `Initialize Step` method of the `Purge` class

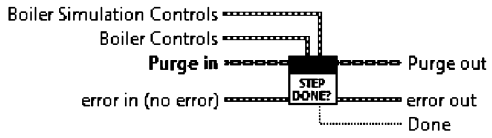


Figure P.96: The icon of the `Step Done` method of the `Purge` class

log file path to the private data of an `Purge` object. The purge time limit is parsed from the array of arguments and is placed in the private data. The `Timer` object is unbundled from the private data of the object and is initialized using the `Initialize Timer` method of the `Timer` class. The initialized timer is then bundled back into the private data of the object. The DONE Boolean of the parent `Step` class is then set to False using the `Write Done` method of the `Step` class.

The icon of the `Step Done` override method of the `Purge` class is shown in Figure P.96, and the block diagram is shown in Figure P.97. The logic on the block diagram is the same as that shown in Figure G.31. The value of the DONE Boolean is written to the private data of the parent `Step` class using the `Write Done` method of the `Step` class.

The icon of the `Step Output` override method of the `Purge` class is shown in Figure P.98, and the block diagram is shown in Figure P.99. The **`Format Into String`** function is used to incorporate the rounded elapsed time into the log string. Incorporating the elapsed time ensures that the log string is always unique, so no

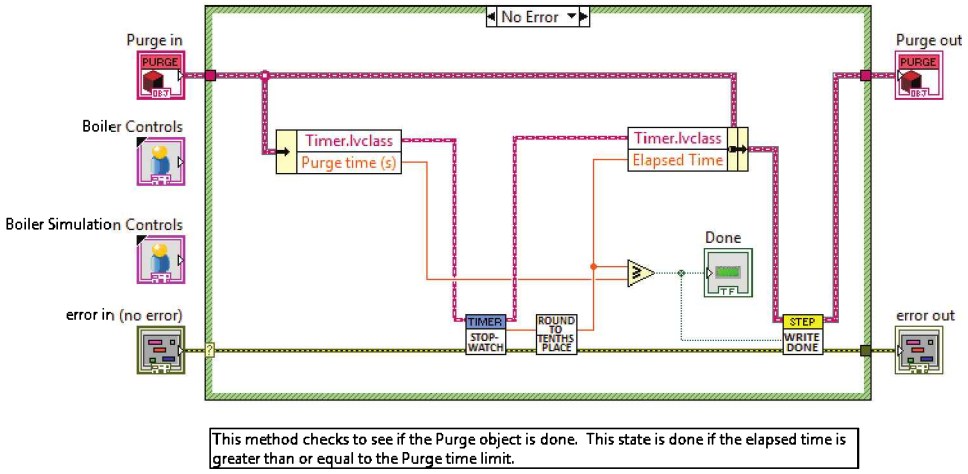


Figure P.97: The block diagram of the `Step Done` method of the `Purge` class



Figure P.98: The icon of the `Step Output` method of the `Purge` class

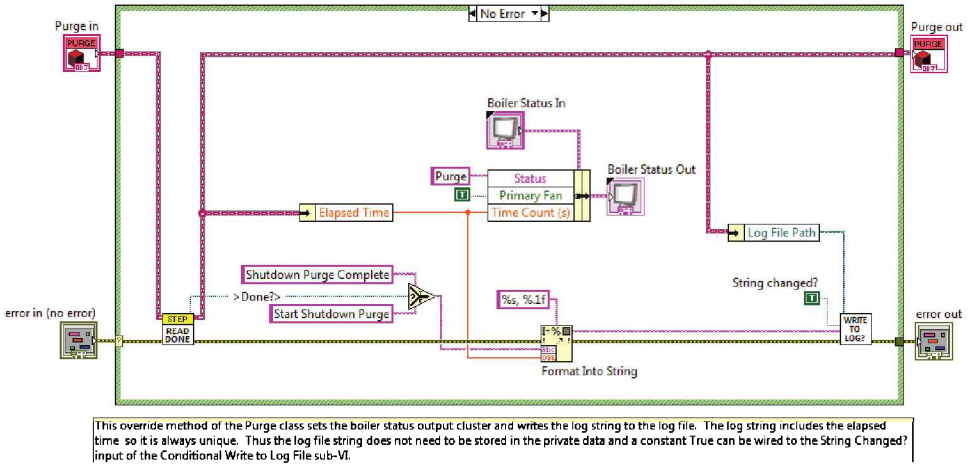


Figure P.99: The block diagram of the `Step Output` method of the `Purge` class

uniqueness check needs to be made. A Boolean True constant is wired to the *String Changed?* input of the `Conditional Write To Log File` sub-VI to ensure that the string is written to the log file. The status cluster is also updated in this method.

Exercise 10.26

The icon of the `Get Step Object` sub-VI is shown in Figure 26.33, and the block diagram showing the *Run* case is shown in Figure P.100. Each case of the case structure represents a different child object. If an unknown string is passed to the case structure, the default state produces a `Step` object and generates a custom error, as seen in Figure P.101.

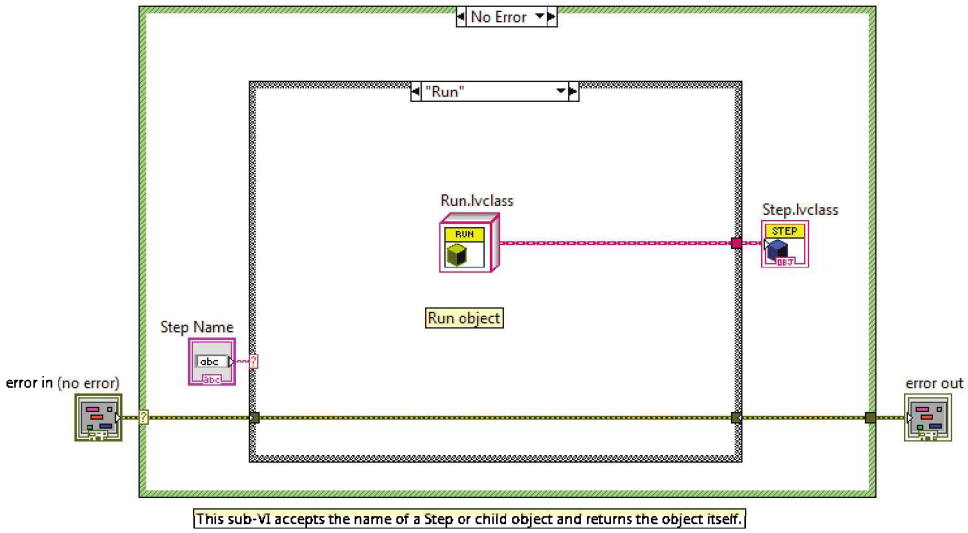


Figure P.100: The block diagram of the `Get Step Object` sub-VI

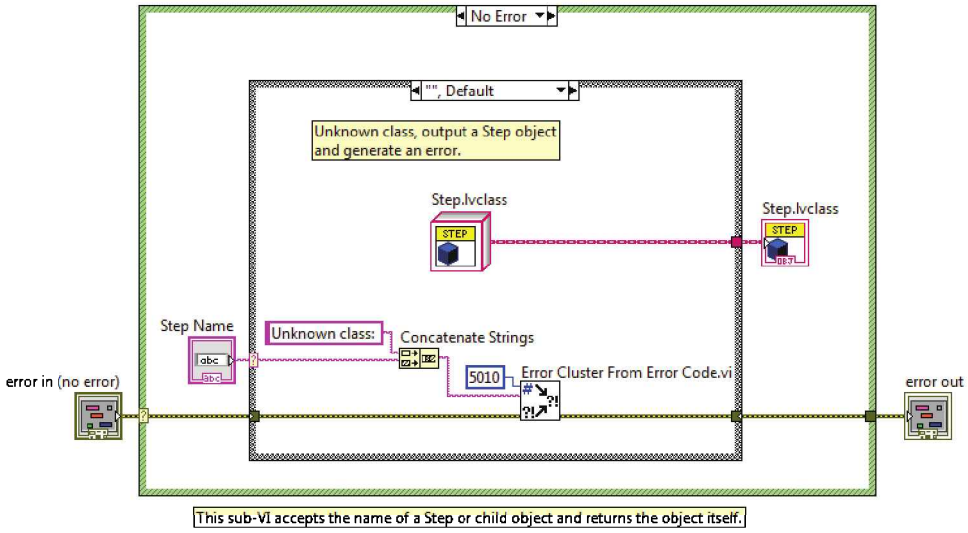


Figure P.101: The block diagram of the `Get Step Object` sub-VI

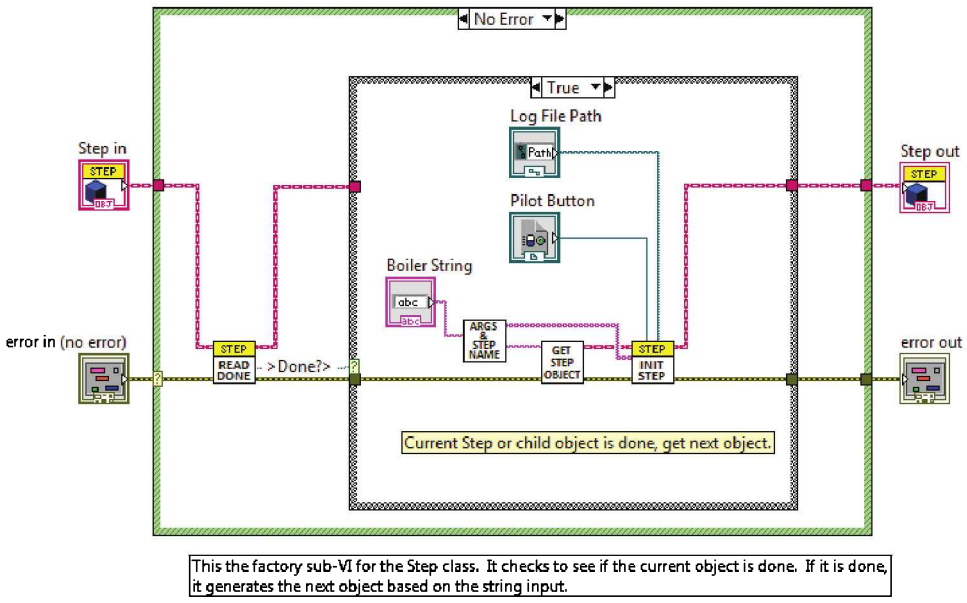


Figure P.102: The block diagram of the `Step Factory` sub-VI, current object done

Exercise 10.27

The icon of the `Step Factory` sub-VI is shown in Figure 26.34, and the block diagram is shown in Figure P.102. The DONE Boolean of the current `Step` object or child object is read using the `Read Done` method of the `Step` class. If the current object is completed, the input string is parsed using the `Parse State and Arguments` shown in Figure 20.12. The argument string is then passed to the `Get Step Object` sub-VI developed in Exercise 10.26. The resulting object is then initialized using the `Initialize Step` method of the `Step` class. If the current object is not completed, it is passed directly to the output, as shown in Figure P.103.

Exercise 10.28

Is it worth it? The answer to this question depends on the context. Yes, developing a class-based version of the boiler controller is much more labor intensive than the straightforward state machine implementations developed earlier in the book. The class-based method has one obvious advantage, it has many fewer states. The classic state machine implementation had eleven states while the class-based version has only four with only one of these states doing the real work of the state machine.

Another advantage of the class-based method is scalability. New steps can be added without having to modify the main application. Testability is improved as well. The individual classes can be tested independently of the main application. A team of developers can easily develop the classes in parallel, since they are all independent units.

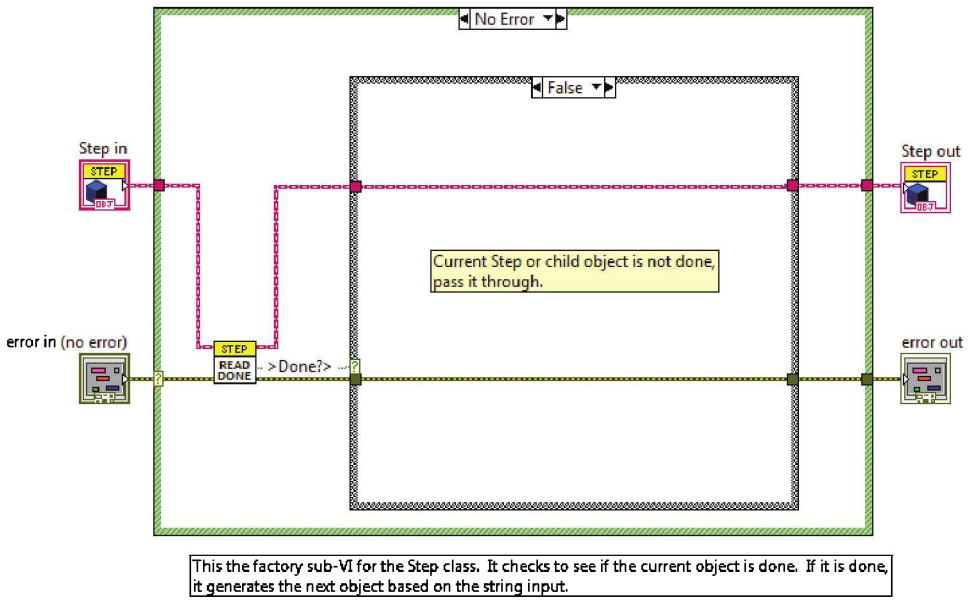
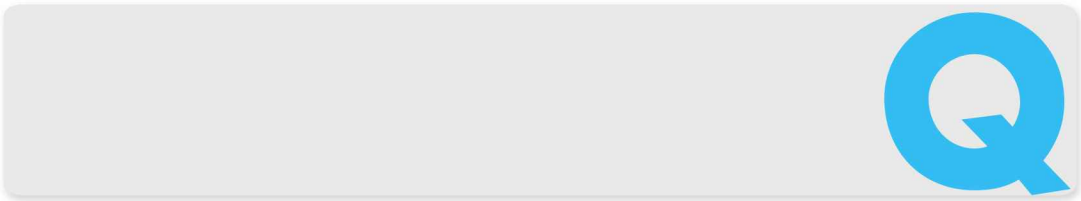


Figure P.103: The block diagram of the `Step Factory` sub-VI, current object not done

Code reuse is also improved. The basic framework developed here can be applied to many other multi-step processes. The only thing necessary is for a process to be made up of individual states that need to be monitored and that produce output of some sort. The ideas and code developed here easily can be adapted to other multi-step processes.

These advantages are all dependent on the size of the project. For smaller projects, these advantages likely will not justify the extra effort needed to develop the necessary classes. The effort will be justified for large projects. Medium-sized projects may or may not be justified in using classes. It depends on the experience and preferences of the developer.



Problem Set 11 Solutions

Exercise 11.1

The block diagram of the `Update Log Slowly` sub-VI is shown in Figure Q.1. This sub-VI is exactly like the `Update Log` sub-VI shown in Figure H.14, but it has the addition of the **`Wait (ms)`** function inside the case structure. This function is configured to add a five-second (5000 millisecond) wait to the block diagram whenever the log file is updated.

If you replace the `Update Log` sub-VI with the `Update Log Slowly` sub-VI in any of the single-loop security system controllers, you will see the need for a parallel-loop implementation. The front panel becomes unresponsive to user input when the log file is updating. This is because a single-loop state machine can only process one state at a time. The `Update Log Slowly` sub-VI forces the state machine

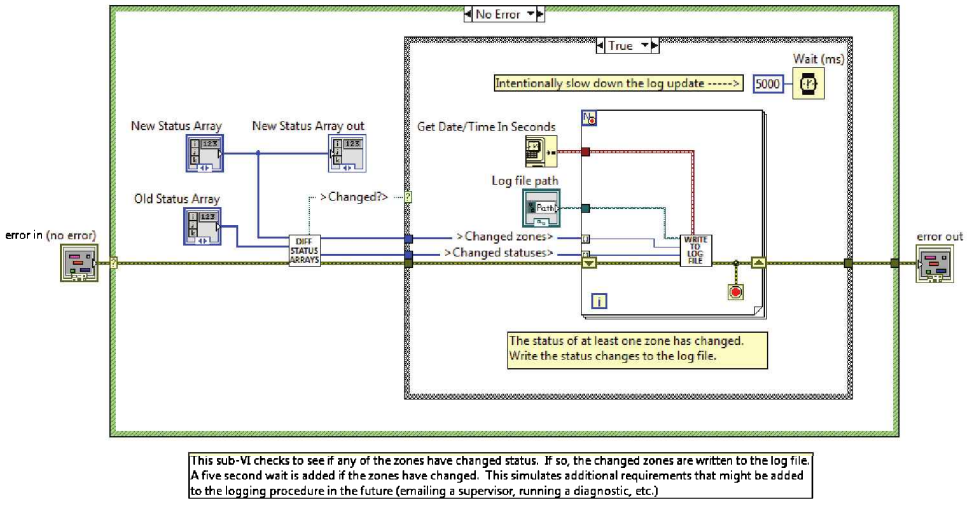


Figure Q.1: The block diagram of the `Update Log Slowly` sub-VI

to stay in the *Update Log* state for five seconds. The *Wait* state is not executing during those five seconds, so the front panel is unmonitored and unresponsive during that time. This is undesirable behavior for a security system. If a parallel loop security system controller is developed, one loop of the controller can monitor the front panel continuously while the other loop takes its time updating the log file. The front panel would remain responsive while the log is being updated. The log updates will still be slow, but since the update requests would be stored on a lossless queue, all log updates eventually would be processed while the front panel remains responsive to user input.

Exercise 11.2

The overall structure and normal operation of the producer-consumer version of the security system controller is shown in Figure Q.2. The queue is formed in the input area to both loops, and the queue reference wire is split and passed to both loops. The queue element is a cluster consisting of a state enumerated constant and a zone status array constant. The state enumerated constant lists the three possible states for the consumer loop: *Initialize*, *Update Log*, and *Shutdown*. Notice that there is no *Monitor* state in the consumer loop. The producer loop continuously monitors the user interface. Since the producer loop is responsible for monitoring the user interface, the code to change the zone indicator colors is also placed in the producer loop.

The array of zone indicator lamp references is also generated from a reference to a zone switches cluster. The zone status array is preallocated using the polymorphic `Initialize Status Array` sub-VI. This sub-VI generates a default zone status array with a size equal to the number of zones. This array is used to initialize the data highway shift register on the consumer loop. This array is also wired to the `Set Zone Colors` sub-VI in the input area of the loop to initialize the zone status

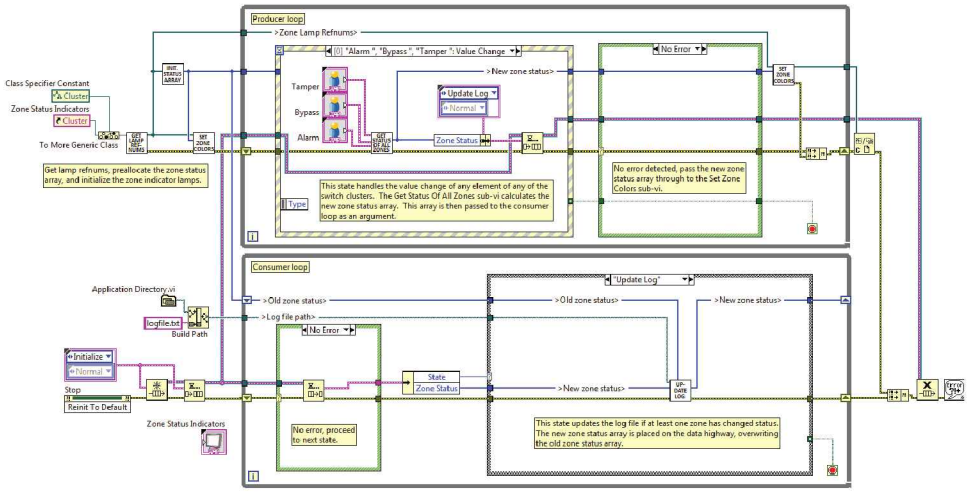


Figure Q.2: The normal operation of the producer-consumer security system controller

indicator lamps to green before the loops start. The default zone status array is also passed to the producer loop as a constant input for later use by the event case handling the value change of the STOP button.

The producer loop contains the event structure that monitors the user interface. Figure Q.2 shows the event case handling the value change of the switches. The new zone status cluster is calculated by the `Get Status Of All Zones` sub-VI. The new zone status array is then bundled with the *Update Log* state and the resulting cluster is placed on the queue. If no error is detected by the producer loop guard clause, the new zone status array is passed to the `Set Zone Colors` sub-VI in the alley of the producer loop. The producer loop then iterates and awaits user input.

Meanwhile, the consumer loop dequeues the state and argument cluster and unbundles its values. The state is wired to the selector of the main case structure, and the zone status array is wired as an input to the main case structure. Figure Q.2 shows the *Update Log* state in the consumer loop. The `Update Log` sub-VI pulls the old zone status array from the data highway and compares it to the new zone status array passed to it from the producer loop. The log file is updated if the status of at least one zone has changed. The new zone status array is then placed on the data highway, overwriting the old zone status array. The consumer loop then iterates. If a new cluster is waiting on the queue, it is dequeued and processed as outlined here. If the queue is empty, the `Dequeue Element` function waits, and the consumer loop sleeps until another state and argument cluster is made available. Notice that the consumer loop is not explicitly timed. Timing the loop in unnecessary since the loop spends most of its time sleeping while waiting for a queue element to appear. When a queue element does appear, it is processed by a single state, and then the consumer loop goes back to sleep while waiting for the next element. Thus, there is no benefit to timing the loop with the **`Wait Until Next ms Multiple`** function.

The *Update Log* state does not dictate a transition to the next state. In the single loop version of the security system with arguments, it was necessary to enqueue a return to the *Monitor* state. But in the producer-consumer version of the security system controller, the producer loop *is* the *Monitor* state, so it is always active.

When the security system controller starts, it executes the *Initialize* state shown in Figure Q.3. This state reinitializes the front panel switches. Again, there is no need to enqueue a state transition from this state, because the producer loop continuously monitors the front panel. After executing the *Initialize* state, the consumer loop will sleep until a new zone status array is available for processing.

While there is only one way to start the security system controller, there are three ways to shut it down. If the user presses the STOP button, the system response is as shown in Figure Q.4. A cluster containing the *Shutdown* state is placed on the queue in the producer loop. The cluster is placed at the back of the queue to allow any log requests on the queue to be processed before the program closes. The STOP button is placed in this event case—even though it is unwired to ensure that it is read and that its mechanical action is triggered. A Boolean True is wired to the producer loop stop terminal, stopping the producer loop. The default zone status array is passed through the event structure and the guard clause, causing the `Set Zone Colors` sub-VI to reinitialize the zone indicator lamps before the producer loop shuts down. When the producer loop stops, the array of indicator lamp references are then passed to the

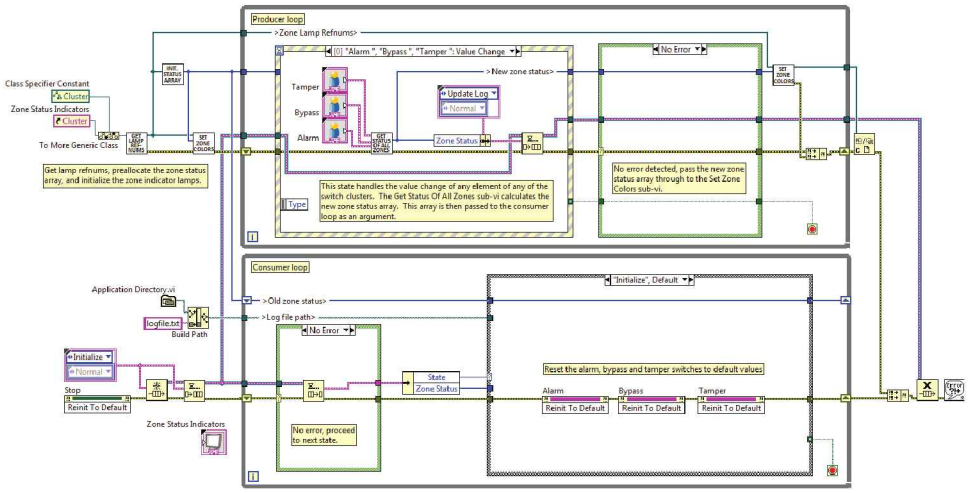


Figure Q.3: The *Initialize* state of the producer-consumer security system controller

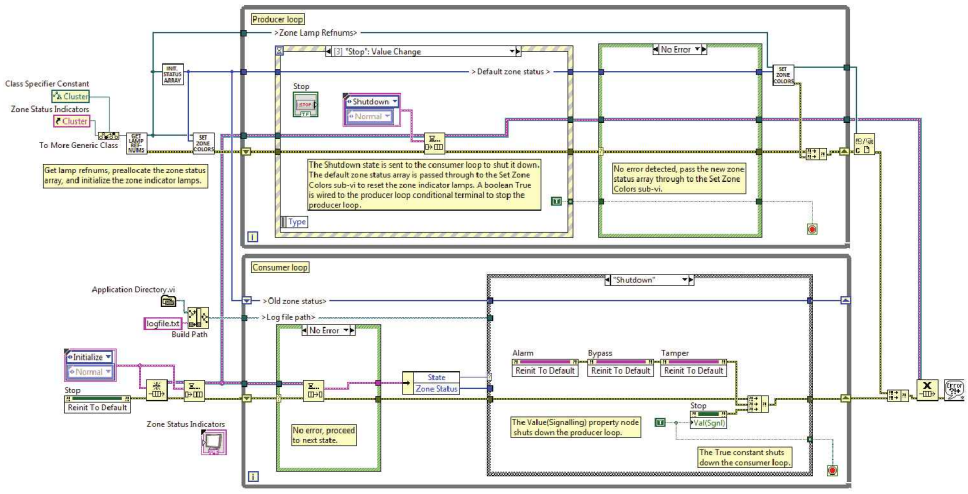


Figure Q.4: The *Shutdown* state of the producer-consumer security system controller

`Close Reference` function, closing the references before the program exits. The state and argument cluster is then dequeued by the consumer loop, and the *Shutdown* state is executed. A Boolean True is wired to the VALUE(SIGNALLING) property node of the STOP button, but since the producer loop has already shut down, this has no effect. The STOP button is also reinitialized in this state. The mechanical action of the STOP button is set to *Switch When Pressed*, since latching mechanical actions create difficulties with the VALUE(SIGNALLING) property node. The Boolean True is also wired to the consumer loop stop terminal, stopping the consumer loop. The error wires from

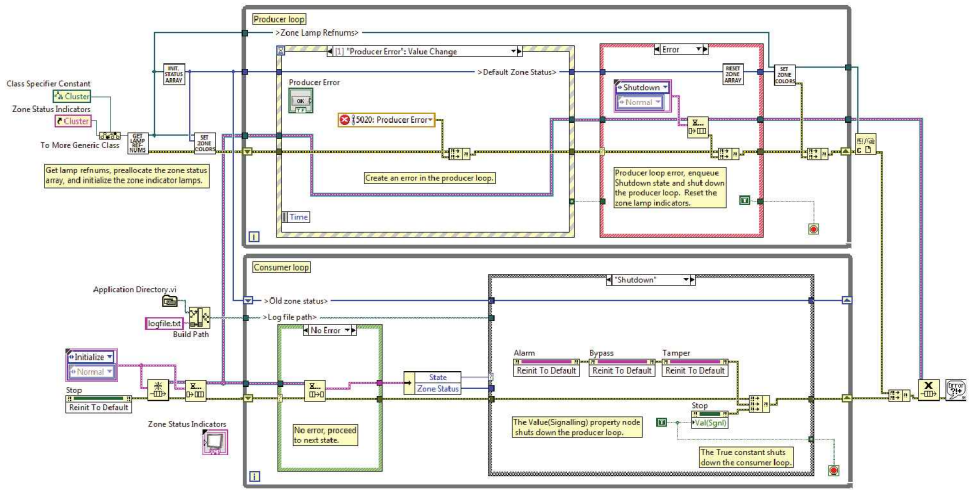


Figure Q.5: The producer-consumer security system controller shuts down if a producer loop error is detected

the two loops are then merged, and the queue is released. Merging the errors ensures that both loops have shut down before the queue is released.

If an error occurs in the producer loop, the security system response is as shown in Figure Q.5. This figure shows the response of the controller to a simulated error in the producer loop. The event case that handles the value change of the PRODUCER ERROR button produces a custom error using the **`Error Cluster From Error Code`** function. The guard clause following the event structure detects the error and enqueues a cluster containing the *Shutdown* state at the front of the queue for immediate execution. The **`Enqueue Element`** function is placed on a parallel error wire to ensure that it executes—even though there is an error present on the main error wire. The error on the main error wire is given priority in the merge. Notice that the *Shutdown* state is enqueued normally and not placed at the front of the queue by using the **`Enqueue Element At Opposite End`** function. This is done to ensure that any *Update Log* requests remaining on the queue are processed before the program exits. The `Reset Zone Status` sub-VI in the producer loop guard clause resets the zone status array to its default value. A Boolean True is wired to the producer loop stop terminal.

The zone indicator lamps are then reset to their default state by the `Set Zone Colors` sub-VI, and the producer loop shuts down. The `Set Zone Colors` sub-VI is placed on a parallel error wire to ensure that the zone lamps will be reset—even if an error is present on the producer main error wire. The consumer loop then dequeues and executes the *Shutdown* state, shutting down the consumer loop. Both loops shut down if an error occurs in the producer loop.

If the user generates an error in the consumer loop by pressing the CONSUMER ERROR button, the security system responds as shown in Figure Q.6. The event case handling the value change of the CONSUMER ERROR button enqueues the *Consumer*

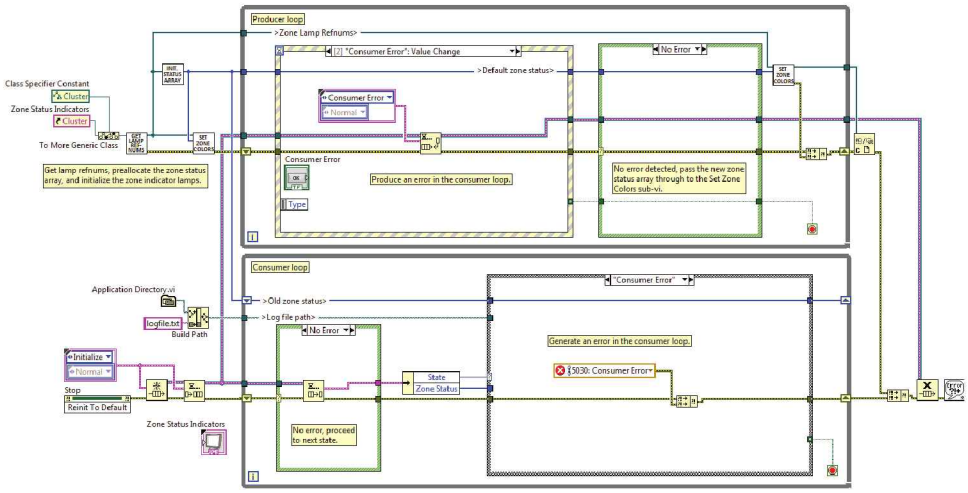


Figure Q.6: Generating an error in the consumer loop of the producer-consumer security system controller

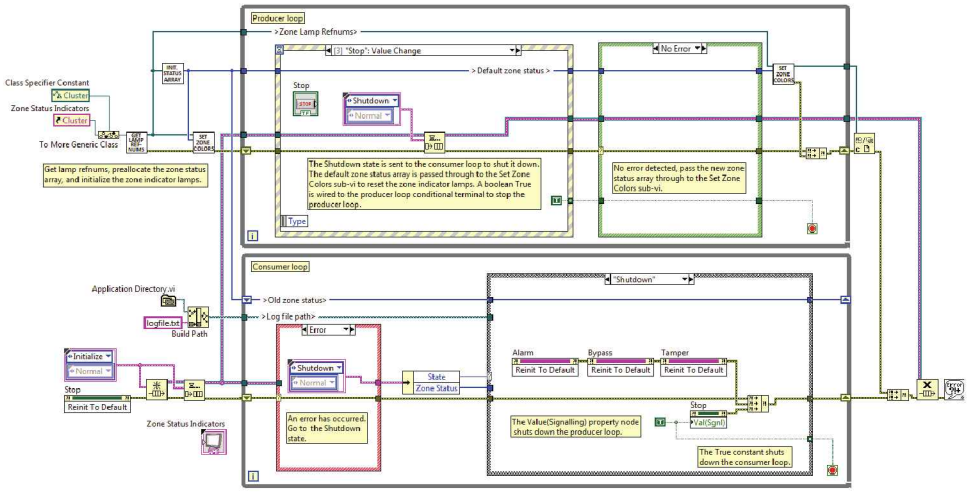


Figure Q.7: The producer-consumer security system controller shuts down if a consumer loop error is detected

Error state. This state generates a custom error in the consumer loop using the **`Error Cluster From Error Code`** function.

The error is detected by the consumer loop guard clause, as shown in Figure Q.7. The guard clause bypasses the queue and sends the consumer loop directly to the *Shutdown* state. This shuts down the consumer loop. In the *Shutdown* state, the VALUE(SIGNALLING) property node of the STOP button executes, triggering the event case in the producer loop that handles the value change of the STOP button.

The VALUE(SIGNALLING) property node is placed on a parallel error wire to ensure that it executes—even if a consumer loop error has occurred. The event structure in the producer loop detects the STOP button value change signal sent by the consumer loop and shuts down. Thus, both loops shut down if an error occurs in the consumer loop. This concludes the producer-consumer version of the security system controller.

If the user simulates an error with either the PRODUCER ERROR or CONSUMER ERROR buttons, the state machine will shut down gracefully and report the correct error. The state machine is also scalable. If the number of zones on the front panel is changed, the block diagram will handle the changed number of zones correctly without needing to be modified.

Exercise 11.3

The front panel will remain responsive if the `Update Log Slowly` sub-VI is used in place of the `Update Log` sub-VI. This demonstrates the utility of parallel loop architectures. If the STOP button is pressed while there are still log update requests on the queue, there will be a delay before the program ends as the log updates are processed. This behavior is due to the fact that the producer loop enqueues the *Shutdown* state normally when the STOP button is pressed. All states on the queue ahead of the shutdown request are processed before the state machine shuts down. Notice that if the user simulates a producer loop error, the producer loop shuts down immediately while the consumer loop still runs and processes any pre-existing log update requests.

Exercise 11.4

It is critically important to merge the producer and consumer loop error wires before releasing the queue. Merging the error wires uses dataflow to ensure that both loops have shut down before releasing the shared queue. If the queue is released before merging the error wires, there is a possibility that the loop that did not release the queue might attempt to access the queue, causing an error. This is illustrated in Figure Q.8. Notice that the `Update Log Slowly` sub-VI is being used to update the log. Imagine that a number of log update requests are on the queue and that the user presses the PRODUCER ERROR button. In Figure Q.8, the producer loop releases the queue before the error wires are merged, so there is no guarantee that the consumer loop has ended. In fact, since there are a number of slow log update requests on the queue, the producer loop is guaranteed to release the queue while the consumer loop is processing the log update requests. When the consumer loop attempts to dequeue the next log update request, it generates an error, since it is attempting to access a non-existent queue. The program will terminate with an error due to trying to access the queue rather than terminating due to the simulated producer loop error.

Curiously enough, the block diagram shown in Figure Q.9 works correctly—even though the queue is released before the error wires are merged. In this case, the **`Release Queue`** function receives the queue reference from the producer loop and the error wire from the consumer loop. Basic dataflow principles stipulate that a function does not execute until it receives data at all of its input terminals. So the queue cannot be released until the producer loop terminates and passes on the queue

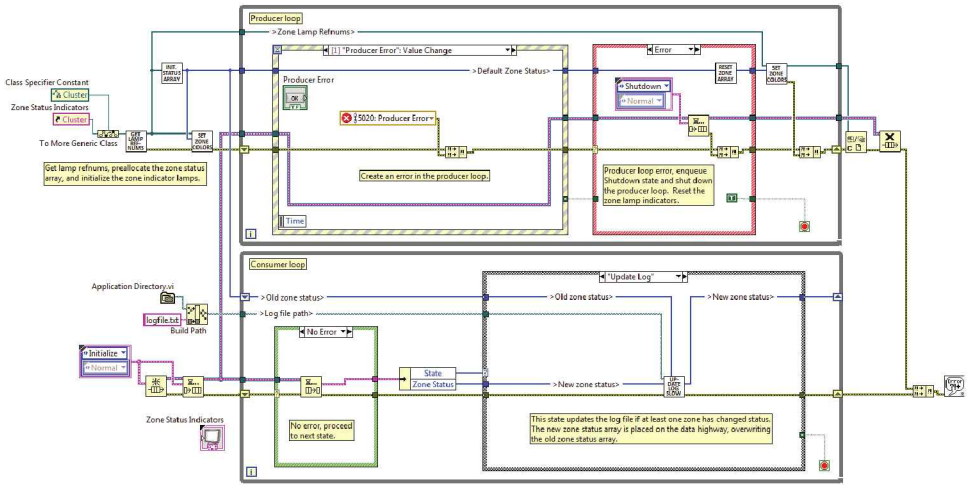


Figure Q.8: Releasing the queue before merging the error wires can cause problems

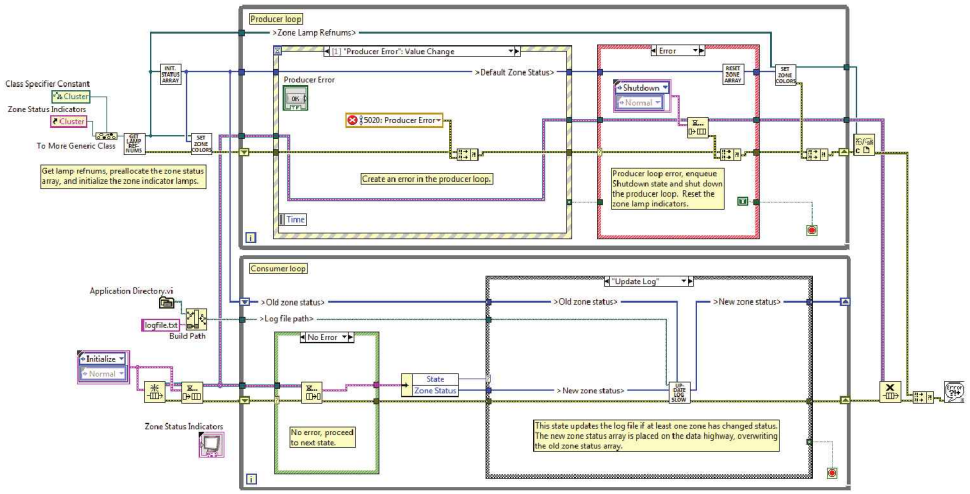


Figure Q.9: Releasing the queue before merging the error wires works in this case

reference and the consumer loop terminates and passes on the error information. So the queue cannot be released until both the producer and the consumer loop terminate. However, merging the error wires before the queue is released results in more readable code, so it is preferred to the implementation shown in Figure Q.9.

Exercise 11.5

The block diagram of the `Create State and Argument String` sub-VI is shown in Figure Q.10. The zone status array is flattened to a string using the native **`Flatten`**

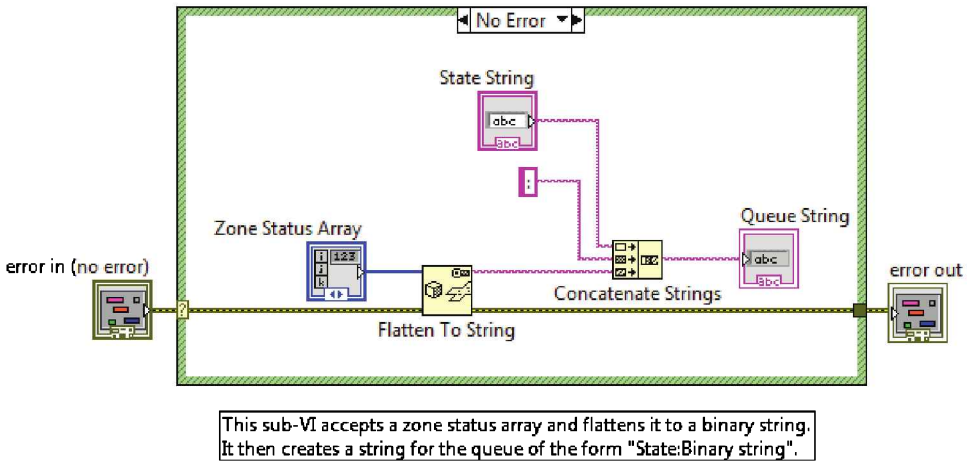


Figure Q.10: The block diagram of the `Create State and Argument String` sub-VI

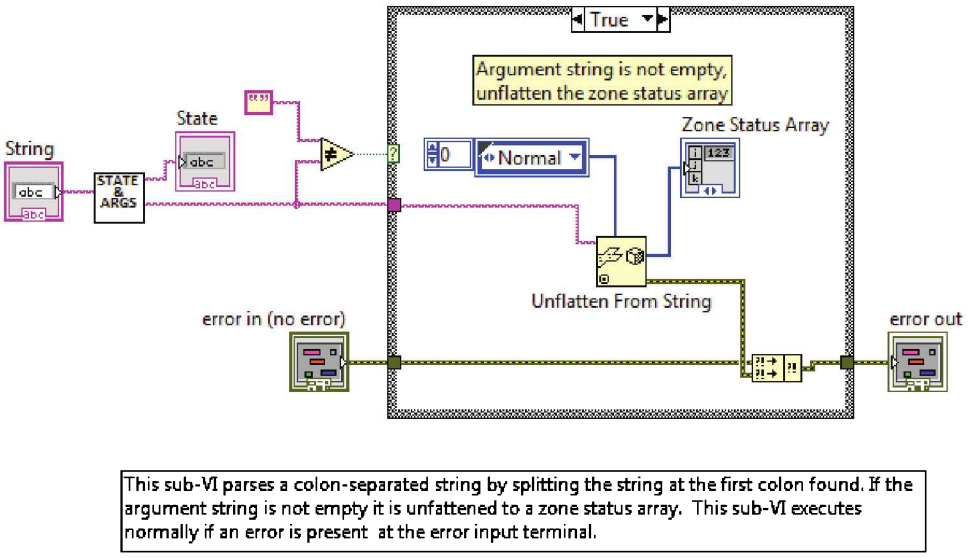


Figure Q.11: The block diagram of the `Parse State and Zone Status Array` sub-VI

`To String` function. The resulting string is then concatenated with the input state string to form a string of the form `State:Argument`, where the argument is the flattened string.

The block diagram of the `Parse State and Zone Status Array` is shown in Figure Q.11. The state and arguments are extracted from the input string using the `Parse State and Arguments` sub-VI developed in Figure 20.12. The argument string is then compared to the empty string to determine if any arguments are present.

If the argument string is not empty, the argument string is unflattened using the **`Unflatten From String`** function. This function is placed on a parallel error wire to ensure that it executes—even if an error is present on the main error wire. This is very important. Suppose that the consumer loop develops an error. The consumer loop guard clause detects the error and enqueues the *Shutdown* state string. The *Shutdown* state string then is passed to the `Parse State and Zone Status Array` sub-VI. This sub-VI needs to function properly and pass the *Shutdown* state to the main case structure—even though a consumer loop error is present. If it does not function correctly and simply passes the error on, the state output of this function will be the default value for a string, namely the empty string. When the empty string is passed to the main case structure, it triggers the default state rather than the *Shutdown* state. The default state generates another error due to the unrecognized state. The new error again causes the guard clause to queue up the *Shutdown* state, but the error again causes the `Parse State and Zone Status Array` to forego execution and output the empty string to the selector of the main case structure. An infinite loop is generated, and the program will hang.

Exercise 11.6

The block diagram of the string-base version of the producer-consumer security system is shown in Figure Q.12. This figure shows the cases where the queue is accessed in the producer loop and in the consumer loop. The sub-VIs developed in Exercise 11.5 are used to flatten and unflatten the zone status array using native LabVIEW functions. Notice how the block diagram is more readable and maintainable than the block diagram developed in Exercise 11.2. The argument string is not seen in this version of the code. State strings without arguments are queued up everywhere except in

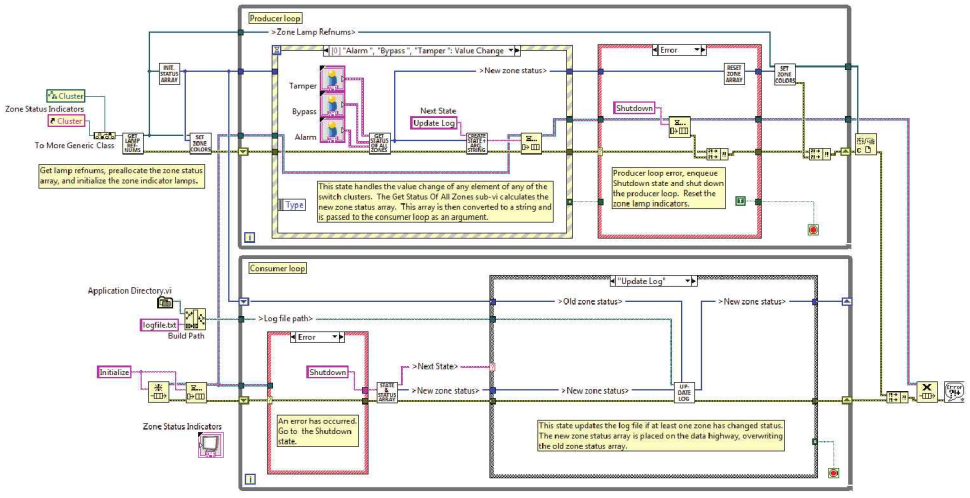


Figure Q.12: The string-based producer-consumer security system controller

the event case handling the value change of the switches clusters. This event case uses the `Create State and Argument String` sub-VI to create the combined state and argument string. The `Parse State and Zone Status Array` sub-VI is then used to convert the argument string to a zone status array between the consumer loop guard clause and main case structure. The queue string is much more readable than the queue cluster because the argument is not included when it is not need.

The string-based version of the security system controller is just as scalable and just as error-tolerant as the cluster-based version. The only difference between the two versions of the code is that the zone status array is flattened to a string before being passed to the consumer loop and is unflattened when read in the consumer loop. The native **`Flatten To String`** and **`Unflatten From String`** functions do not impact the scalability, parallelism, and maintainability of the cluster-based code.

Exercise 11.7

The flattening process will be aided by the `Character Array To String` sub-VI with the block diagram shown in Figure Q.13. This sub-VI uses a shift register to accomplish its task. The shift register is initialized with the empty string. This is important because uninitialized shift registers retain their value from one call to the next. This would be undesirable, because each call of this sub-VI is independent of other calls and an uninitialized shift register would contaminate the results of one call with results from the previous call.

The empty string becomes the first element of the concatenated string. The string array elements are then passed into a **`For`** loop using an auto-indexed input tunnel. The first array element is concatenated with the empty string, the result of which is simply the first array element. This string is then placed on the shift register. On the next loop iteration, the second array element is concatenated to the first, and the result is returned to the shift register. This process continues until the array is exhausted, creating a single string with all of the array elements concatenated in their array order.

The `Character Array To String` sub-VI can then be used to develop the `Zone Status Array To String` sub-VI shown in Figure Q.14. This sub-VI

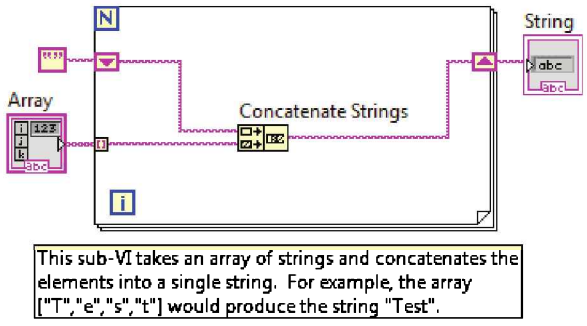


Figure Q.13: The block diagram of the `Character Array To String` sub-VI

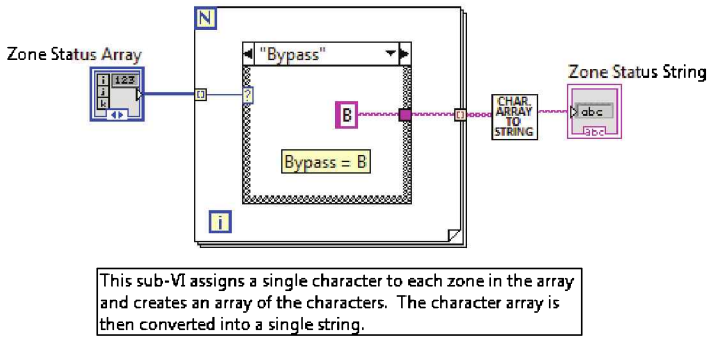


Figure Q.14: The block diagram of the `Zone Status Array To String` sub-VI

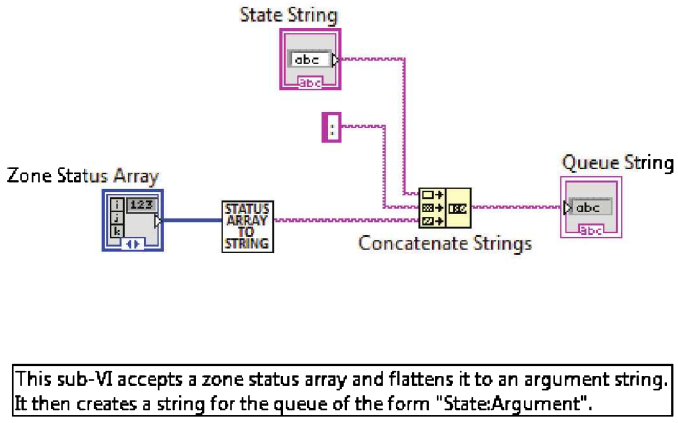


Figure Q.15: The block diagram of the `Create State and Argument String Custom Flattening` sub-VI

flattens the zone status array to a string containing the initial letters of each zone status. The zone status array is wired to the auto-indexed input of a **`For`** loop. The auto-indexed input passes each zone status to a case structure keyed to the zone status. Each zone status generates a string character corresponding to the first character of the zone status. The string characters are collected by an auto-indexed output tunnel and are converted to an array. The array is then converted to a string using the `Character Array To String` sub-VI developed here.

The custom flattening scheme can now be implemented in the `Create State and Argument String Custom Flattening` sub-VI shown in Figure Q.15. The zone status array is flattened to a string using the `Zone Status Array To String` sub-VI developed previously. The resulting string is then concatenated with the state string using the `State:Argument` convention. Since the custom flattening sub-VI has no error terminals, this sub-VI has no error terminals either.

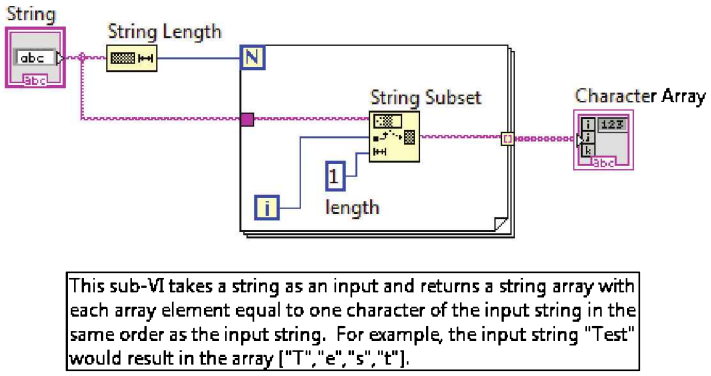


Figure Q.16: The block diagram of the `String To Character Array` sub-VI

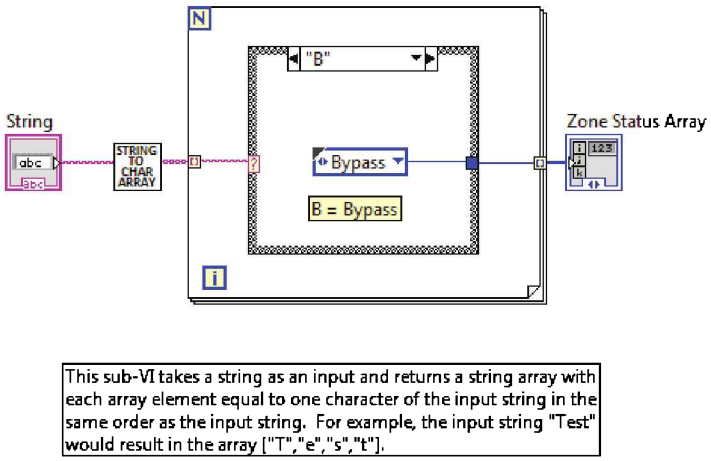


Figure Q.17: The block diagram of the `String To Zone Status Array` sub-VI

The unflattening process will be aided by the `String To Character Array` sub-VI with the block diagram shown in Figure Q.16. This sub-VI accepts a string and creates an array of its individual characters in their original order. It does this by first determining the length of the string using the **`String Length`** function. The length is wired to the count terminal of a **`For`** loop, causing the loop to run once per character in the input string. Inside the loop, the **`String Subset`** function is used to isolate each individual character in turn. This is done by wiring the loop iteration counter to the *offset* input of the **`String Subset`** function and by wiring a `1` to the *length* input. The individual characters are then sent to an auto-indexed output tunnel where they are collected into the desired character array.

The `String To Character Array` sub-VI can then be used to develop the `String To Zone Status Array` sub-VI shown in Figure Q.17. The input string

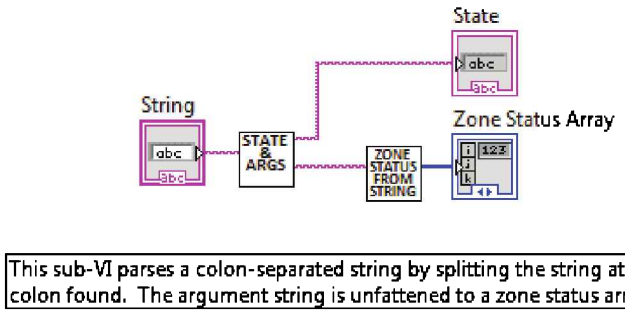


Figure Q.18: The block diagram of the `Parse State and Zone Status Array Custom Unflattening` sub-VI

is parsed into a character array using the `String To Character Array` sub-VI developed previously. The character array is then wired to an auto-indexed input tunnel of a **`For`** loop, and each character element is wired to a case structure. The case structure delivers the appropriate zone-status enumerated constant corresponding to the character string, and these enumerated constants are collected into a zone-status array by an auto-indexed output tunnel.

The custom unflattening scheme can now be implemented in the `Parse State and Zone Status Array Custom Unflattening` sub-VI wth the block diagram shown in Figure Q.18. The queued string is parsed into the state and argument strings using the `Parse State and Arguments` sub-VI shown in Figure 20.12. The argument string is then unflattened to a zone status array using the `String To Zone Status Array` sub-VI developed previously. Notice how much simpler this sub-VI is then the one shown in Figure Q.11. The `Parse State and Zone Status Array` sub-VI needed to check to see if the agrument string was empty before proceeding with the unflattening to avoid generating an error. This is not necessary with the custom unflattening scheme. The Figure Q.17 checks the string length and wires it to the count terminal of a **`For`** loop. If the length is zero, the **`For`** loop does not execute, so the unflattening process does not take place.

Note the extensive use of auto-indexing in these sub-VIs. This ensures that the sub-VIs will work for an arbitrary number of zones. The use of the **`String Length`** function in the `String To Character Array` serves the same purpose. Since the number of characters in the string equals the number of zones, reading the length of the string allows the sub-VI read the number of zones from the input string without making any prior assumptions about the number of zones serviced.

Figure Q.19 shows the string-based producer–consumer security system controller with the custom flattening and unflattening routines in place. The main difference between this top-level application and the one shown in Figure Q.12 is that the custom flattening and unflattening sub-VIs do not have error terminals and are not connected to the main error wires of the producer and consumer loops. This is an advantage of the custom scheme, since there are fewer chances for

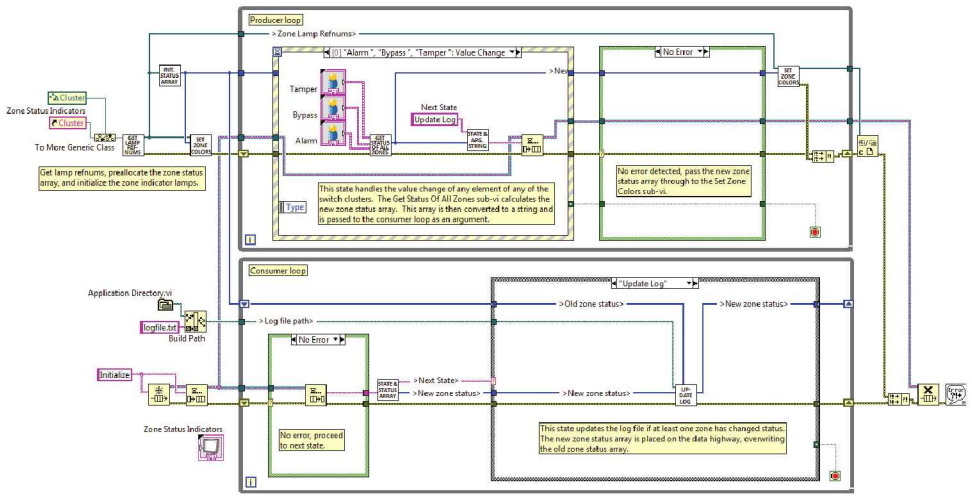


Figure Q.19: The string-based producer–consumer security system controller with custom flattening and unflattening

generating errors. It also allows us to avoid the situation in the consumer loop where the unflattening sub-VI had to be specially constructed to operate normally—even if an error was present at its error input terminals. The custom scheme has other advantages as well. Since the flattened string is short and human-readable, it easily can be probed or displayed (if need be). Perhaps the primary advantage is that the format of the flattened string is known and is totally under the control of the developer. The disadvantages of the custom scheme are that it is likely not reusable in other codes and that it required extra programming effort to implement.

The revised version of the string-based security system controller is just as scalable as its predecessors. As noted, the custom flattening and unflattening sub-VIs make extensive use of auto-indexing to ensure that they will function for an arbitrary number of zones.

Exercise 11.8

The overall structure of the producer–consumer Car Wash 2 controller with switched queue is shown in Figure Q.20. This figure shows the normal method of stopping the controller. The switched queue is created and initialized in the input area of the main loop. The event case handling the value change of the STOP button in the producer loop uses the `Enqueue At Opposite End and Set Switch` sub-VI shown in Figure 29.23 to place the *Shutdown* state at the front of the queue and to simultaneously disable the requeuing process by setting the queue switch to True. A Boolean True is used to shut down the producer loop.

The requeuing process in the *Option* state in the consumer loop uses the `Switched Enqueue At Opposite End` function shown in Figure 29.25 to check the queue

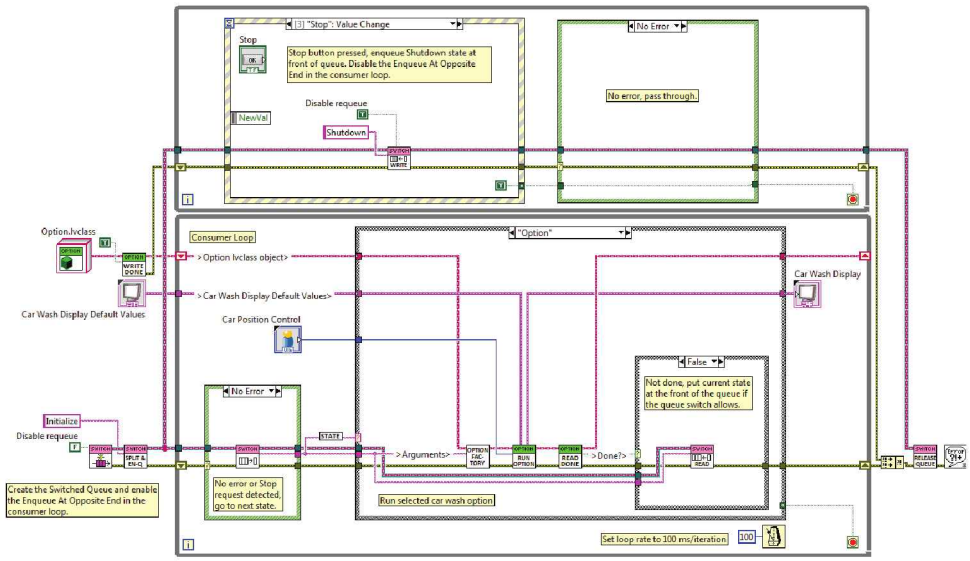


Figure Q.20: The producer-consumer Car Wash 2 controller with switched queue

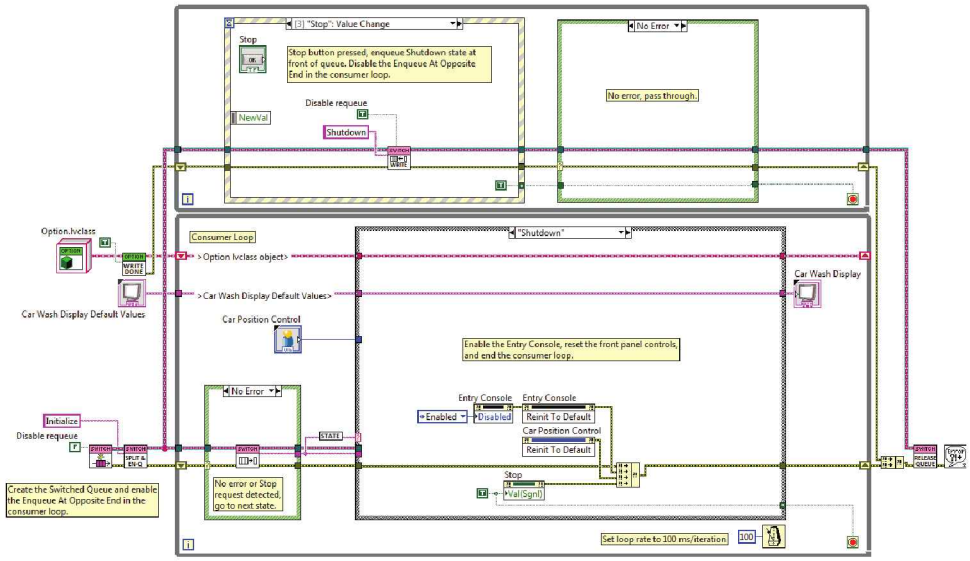


Figure Q.21: Stopping the producer-consumer Car Wash 2 controller with switched queue

switch before performing the requeue. This prevents the current option state from being queued on top of the *Shutdown* state, preventing a race condition.

The *Shutdown* state is shown in Figure Q.21. A *Value(Signaling)* property node is used to trigger the event structure, but this is of no consequence, since the producer

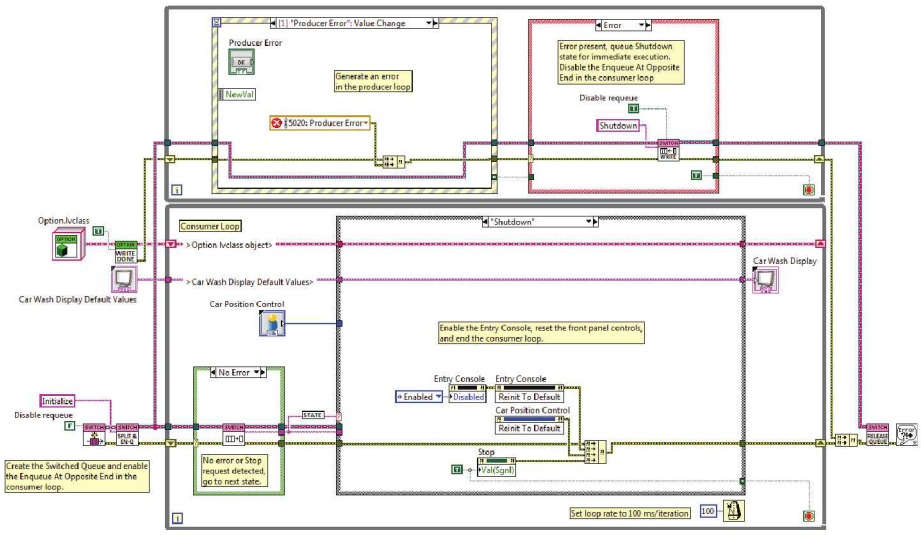


Figure Q.22: The producer-consumer Car Wash 2 controller with switched queue, error in the producer loop

loop has already shut down. A Boolean True is used in the *Shutdown* state to shut down the consumer loop. After both loops have shut down, their error wires are merged and the switched queue is released. This error merge ensures that both loops shut down before the switched queue is released.

The Car Wash 2 controller will also shut down if an error is detected in the producer loop, as shown in Figure Q.22. This figure shows the event case handling the value change of the PRODUCER ERROR button. A custom error is generated in the event case. The error is detected by the guard clause in the producer loop. The *Shutdown* state is placed at the front of the queue, and the requeue operation is disabled by setting the queue switch to True. A Boolean True is wired to the conditional terminal of the producer loop, shutting it down. When the *Shutdown* state is dequeued in the consumer loop, the consumer loop will also shut down. Both loops terminate when an error is detected in the producer loop. Recall that the `Enqueue At Opposite End and Set Switch` sub-VI was designed to operate normally—even if an error is present at the error input terminal. This function does not need to be placed on a parallel error wire.

The Car Wash 2 controller will also shut down if an error is detected in the consumer loop, as shown in Figure Q.23. This figure shows the event case handling the value change of the CONSUMER ERROR button. The *Consumer Error* state is placed at the front of the queue and the queue switch is thrown to protect this state from being covered by a requeue operation. The *Consumer Error* state in the consumer loop creates a custom error in the consumer loop. This is detected during the next consumer loop iteration by the guard clause, which sends the consumer loop to the *Shutdown* state shown in Figure Q.21. The *Shutdown* state shuts down the consumer loop and

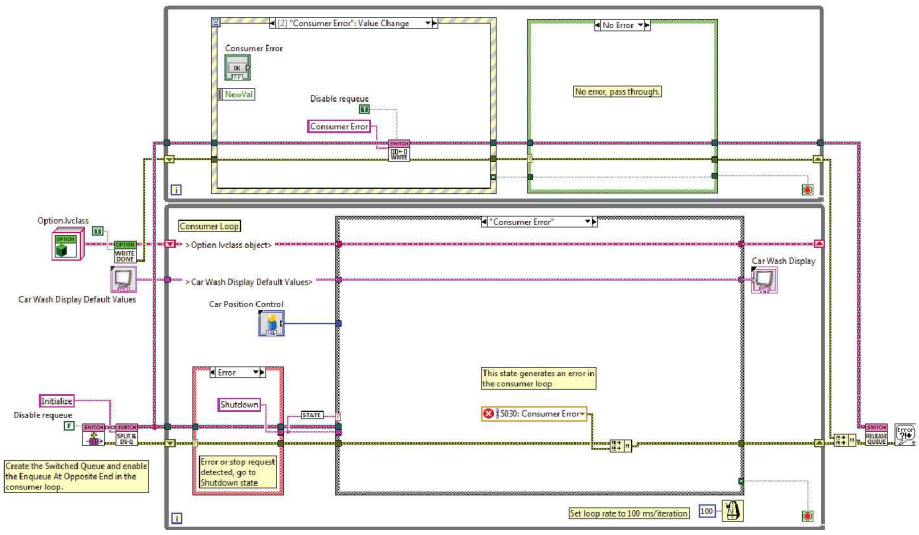


Figure Q.23: The producer-consumer Car Wash 2 controller with switched queue, error in the consumer loop

it also uses a *Value(Signalling)* property node of the STOP button to trigger the event case in the producer loop handling the value change of the STOP button. This in turn shuts down the producer loop. Both loops terminate if an error is detected in the consumer loop.