Refactoring existing MBARI CPF C# code to C/C++ Statement of Work (SOW)

2020 Dec 17	Initial draft
2020 Dec 22	Edited design guidelines section
2021 Jan 19	General edits
2021 Feb 06	Initial release

1. Introduction

MBARI has developed an autonomous oceanographic coastal profiling float (the CPF, more at https://www.mbari.org/coastal-profiling-float/) that has been working relatively well for the last several years. The existing software is written in C# using the .NET Micro Framework and the Visual Studio IDE. In order to move this technology to the greater oceanographic research community outside of MBARI and for a variety of other reasons, we are going to rewrite the code in C/C++. The goal for this work is for SysProgs to provide a well-engineered VisualGDB solution including a test framework focusing on the basic framework and peripheral drivers so MBARI engineers can focus on porting the existing CPF functionality to the new solution.

The basic plan for this work consists of a series of stand-alone work packages (WP) with defined deliverables. Each WP would have a (hopefully) clear statement of work, typically a set of unit and functional tests for the proposed functionality. Personally, I'd like to minimize the amount of separate documentation in favor of easy to understand, self-describing code.

It is worth stating up front that while we have a working Visual Studio C# solution and experience with C/C++ we are open to SysProgs suggestions in any area where improvements can be made. Furthermore, everything in the statement of work is up for discussion.

2. Background

2.1. Hardware

The target processor for the C/C++ app is a STM32F413ZH. The existing C# app runs on a GHI G400S SOM (https://docs.ghielectronics.com/hardware/netmf/g400s.html) with an Atmel SAM9X35 running at 400 MHz (which is probably much more processor than we really need). The G400S does have 128 MB of SRAM and 4MB of flash and we take advantage of a significant amount of this. We will be developing a custom version of the G400S SOM with the STM32F413ZH and will provide external SRAM and flash as needed.

The CPF currently has 6X UART connected devices, 2X spare UART ports, 3X core I2C devices and 9X secondary I2C devices. All 12X I2C devices are multiplexed onto one microcontroller I2C port. There is also a uSD card on a 4 bit wide SDIO port.

2.2. Software

The current C# app uses a non-blocking state machine with minimal interrupt service routines for asynchronous device communications. There are 3 basic types of classes: states, devices and peripherals. The basic states for the CPF are SMNotRunning, initializePlatform, initializeMission, preMissionDelay, recovery, initProfile, findNeutral, descend, park, anchor, startCPMode, deanchor, ascend, stopCPMode, surfaceOps and quit. Each state inherits basic functionality from StateBase. Each device is basically an external piece of hardware e.g. Iridium modem, motor controller, microSD

card or scientific instrument, that interfaces to the CPF app via a standard microcontroller peripheral e.g. UART, I2C and possibly SPI in the future. As an example, the most important scientific instrument is a Seabird 41 CTD (conductivity, depth, temperature). In the C# code, there is a singleton CTD class that provides the functionality required to control the CTD and this class inherits a UART class.

3. Work Package 1 (WP1): Review of existing code

I've been developing a small C/C++ prototype of our existing C# code with the hope of improving my knowledge of C/C++ and prototyping the critical parts of the existing code. I've used a STM32F413ZH Nucleo board and a STM32F413 Discovery board for this work. This first WP would review this prototype with the intent of 1) introducing SysProgs to the existing software design and 2) identify ways to improve the design. This work would be on an hourly basis with the hope of completing this task in something like 4-8 hours.

Deliverables:

a. List of changes to MBARI prototype. Hopefully, this list can be documented with comments in the prototype code.
b. Potentially an additional WP for SysProgs to implement some of the changes.
c. List of suggested unit and functional tests.

4. WP2: UART Drivers

Add 9X UART drivers to the existing prototype that can run simultaneously. Each UART driver should be capable of either DMA or interrupt driven modes with the baud rate, stop bits, start bits, etc. fixed at compile time. We can provide specific details defining which device uses which UART, baud rate, stop bits, parity, DMA or interrupt, etc. Theoretically, these details won't change but we all know theory and reality aren't always the same. How configurable these drivers are can be discussed in WP1.

Deliverables:

a. Unit Test: a simple loop back test for each driver that makes sure what gets sent out is received. The BER should be essentially zero.
b. Functional test: Run the prototype code with all 9 unit tests simultaneously at increasingly higher baud rates until the BER is non-zero.

Required Materials:

a. MBARI can provide a STM32F413ZH Nucleo board (https://www.st.com/en/evaluation-tools/nucleo-f413zh.html). These boards are inexpensive enough that it may just be easier for SysProgs to buy them and add it to our invoice3.

5. WP3: I2C Drivers

This is basically the same task as WP2 for the I2C peripherals. We can provide the details of the I2C devices at the beginning of the work.

6. WP4: Mass Storage

I have the STM provided FATFs working in the prototype code on the Discovery board with a built in uSD card but haven't gotten it working on the Nucleo board with a separate uSD protoboard. If we get this working before SysProgs starts great, otherwise we'd like to include it as a WP.

7. Automated code conversion tool

We need to talk about this

8. Software Engineering guidelines

The highest priority goal is to provide robust, well tested code that can run on an autonomous platform for years at a time. The second highest priority goal is to provide code that is easily understandable and maintainable by MBARI engineers upon delivery and also years down the road. We are very pragmatic in our approach to software engineering and development and are happy to do whatever makes sense but we've found that appropriate design guidelines can help. To achieve these goals we propose the following guidelines

8.1. Code should not require an expert understanding of C/C++.

8.2. Classes, methods and variable should use very descriptive names. Use verbs in method names to describe the action being performed e.g., checkForStuckPressure, logEngrMsg, isQEmpty.

8.3. RAII should be used where ever it makes sense.

8.4. Dynamic memory allocation e.g., new, should only be used during initialization

8.5. My personal order of preference is std::array, string, std::vector, c-style arrays

8.6. Encapsulation is important. For example, peripheral drivers shouldn't care what device they're connected to and visa-versa.