

MBARI Carbon WG System Guide

Contents

Topic	Description	Links
Synopsis	brief description of system feature	Synopsis
Tour	hardware and software basics	Tour
Quick Start	learn how to load and run a script	Quick Start
System Overview	topics	System Overview
	Navigating	Navigating
	IO Control	IO Control
	Accessing Information	Accessing Information
	Making Measurements	Making Measurements
	Data Inspection	Data Inspection
	Debug Features	Debug Features
Scripting Guide	Scripting information	Scripting Guide
Integration Guide	Developing clients	Integration Guide
Appendices	Reference information	Appendices
	Data Type IDs	Data Type IDs
	Script Examples	Script Examples
	Output Format Examples	Output Format Examples
	Common Issues	Common Issues
Resources	useful resources	Resources

Entries in each topic are arranged alphabetically.

Synopsis

Feature summary

The MBARI carbon wave-glider payload implements a DIC and pCO₂ measurement application using a general purpose instrument controller and a LI-COR gas analyzer. The controller provides fluid handling and measurement logic to make DIC/pCO₂ observations remotely and produce real-time telemetry.

This document describes the software architecture and design for the carbon wave-glider payload application.

Features

- Script-driven applications - change controller applications without changing firmware
- Single command set used for console and script engine
- Integrated application commands and macros simplify scripting
- Hosted or autonomous operation
- Simple cron-style scheduling
- Stores raw LI-COR data and data summaries
- File pager with search (e.g. less)
- Binary log inspection/formatting utilities (onboard + Mac/Linux/Cygwin)
- Full control of digital IO (input/output)
- Integrated user interface help
- Serial boot-loader for firmware updates
- Serial ASCII file upload
- System log
- Debugging and development features
- serial tranceiver sleep disable
- ADC sleep modes
- automatic sleep timer

Specifications

Parameter	Specification	notes
Input Voltage	12-16 VDC	[1]
Active Current	10 mA	
Sleep Current	2 mA	
digital IO	22	
digital IO voltage	12VDC	
digital IO current	2.5A	
ADC Channels	3	[2]
Storage	SD Card	
File System	FatFS	
RTC	Y	

Parameter	Specification	notes
Timer resolution	10 ms	
UARTs	3	[3]
Processor	PIC24FJ1024GA606	
Clock frequency	32MHz	
Flash RAM	32K	
Flash Storage	700K	[4]

[1] processor only

[2] 2 application, 1 battery monitor

[3] console, host, application

[4] using large data model

[Contents](#)

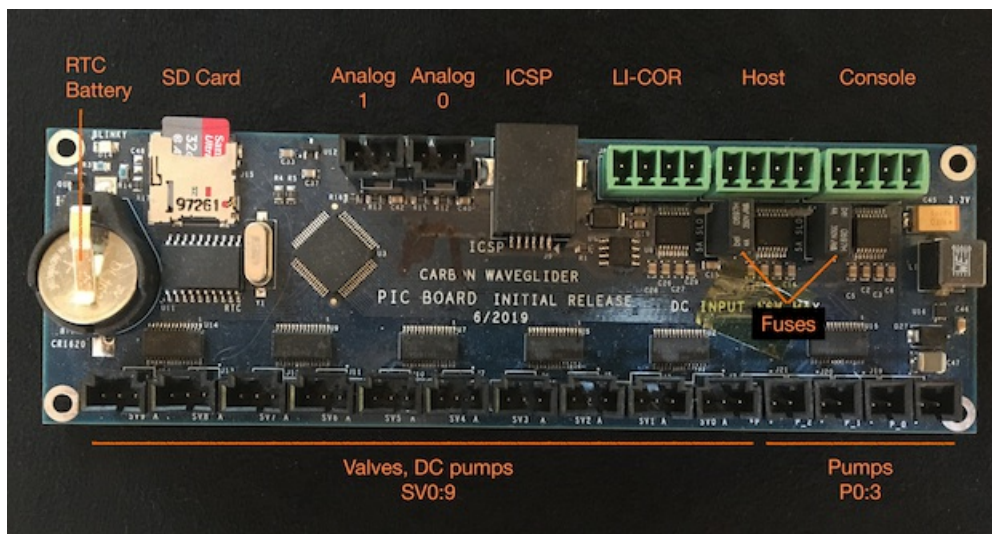
Tour

System orientation

Learn about system components and terminology

Controller Board

Connection	Label	Description	Spec
Serial IO			RS-232
<i>Host Port</i>	<i>HOST</i>	Host computer	
<i>Console Port</i>	<i>CONSOLE</i>	User terminal	
<i>Instrument (LI-COR) Port</i>	<i>LICOR</i>	LI-COR data/control	
Digital IO			12VDC
<i>Valve Ports</i>	<i>SV0:9</i>	Valves, air/H2O pumps	
<i>Pump Ports</i>	<i>P0:3</i>	acid/sample pumps	
Analog IO			0-5V
<i>ADC Ports</i>	<i>Analog0:1</i>	User analog IO	
<i>Battery Voltage Monitor</i>	<i>Analog2</i>	Input voltage	0-16V
Other IO			
<i>ICSP Port</i>	<i>ICSP</i>	Programming port	
Serviceable Components			
<i>SD Card</i>		Data, scripts, cfg	UHS-1 FatFS
<i>RTC battery</i>	<i>BT1</i>	Clock backup battery	CR1620
<i>Fuses</i>	<i>_5A SLO__</i>		BK/PCS 1-5A



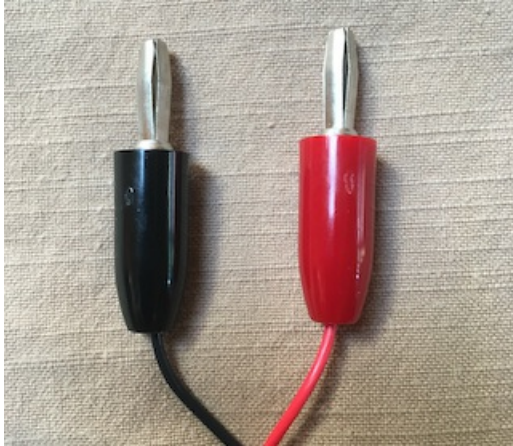
Controller Components

Cables

Cable	Description	Connectors
<i>Console/Host Cable</i>	Controller to host/console	Phoenix 1826995 / DB9M
<i>Licor Cable</i>	Controller to LI-COR	Phoenix 1826995 / DB9F
<i>Pump Test Cable</i>	Test pump port output	C-Grid / Whip SHELL 0050579202 FEMALE PIN 0016020103
<i>Valve Test Cable</i>	Test valve port output	C-Grid / Whip SHELL 0050579203 FEMALE PIN 0016020103

Cable pinouts are documented in the [schematics](#)

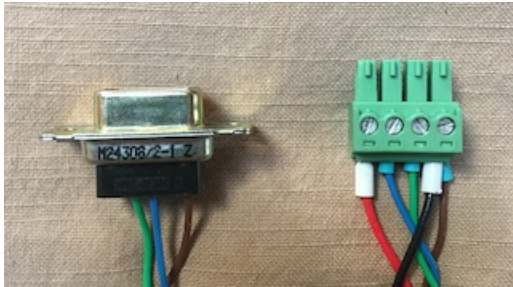
Connector Identification



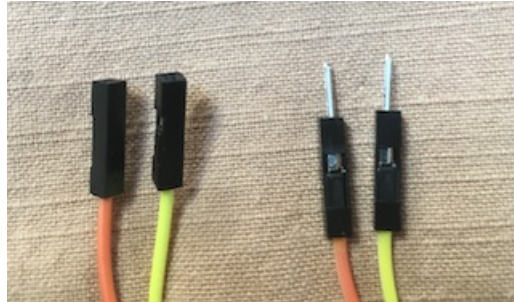
Power Connector (Banana)



Pump/Valve (C-Grid 2/3 pin)]



Serial (DB9/Pheonix)



Test Jumpers (M/F)

Software Components

Component	Description	Notes
<i>User Interface (UI)</i>	Console-based system interface	
<i>Script Engine</i>	Interprets and executes scripts	
<i>Data Queues</i>	Store data for current measurement cycle	
<i>Scheduler</i>	Run scripts at specified rates/times	
<i>Logger</i>	Store data and system events	
<i>Bootloader</i>	Update firmware via console	
<i>Terminal Emulator</i>	Console interface application	e.g. minicom, hyperterm
<i>Utilities</i>	Applications supporting data workflows	Mac/Linux/Cygwin

Files

File	Description	Format
<i>CONFIG.TXT</i>	System configuration file	ASCII
<i>SYSBOOT.SCR</i>	System initialization script	ASCII
<i>DATyymm.nnn</i>	Measurement data log	Binary
<i>RAWyymm.nnn</i>	Raw LI-COR data log	ASCII
<i>SYSyymm.nnn</i>	System event log	ASCII
<i>CRONTAB.SYS</i>	System crontab	ASCII
<i>CRONTAB.USR</i>	User crontab	ASCII
<i>HELP.MD</i>	On-board text	ASCII/Markdown
<i>Scripts</i>	User scripts	ASCII

Quick Start

Basic system operation

Load and run a simple script. In this section, you'll learn to:

- connect controller hardware
 - establish a terminal session to the user interface
 - upload a script from the host computer to the controller
 - examine a file
 - load, inspect and run a script
-

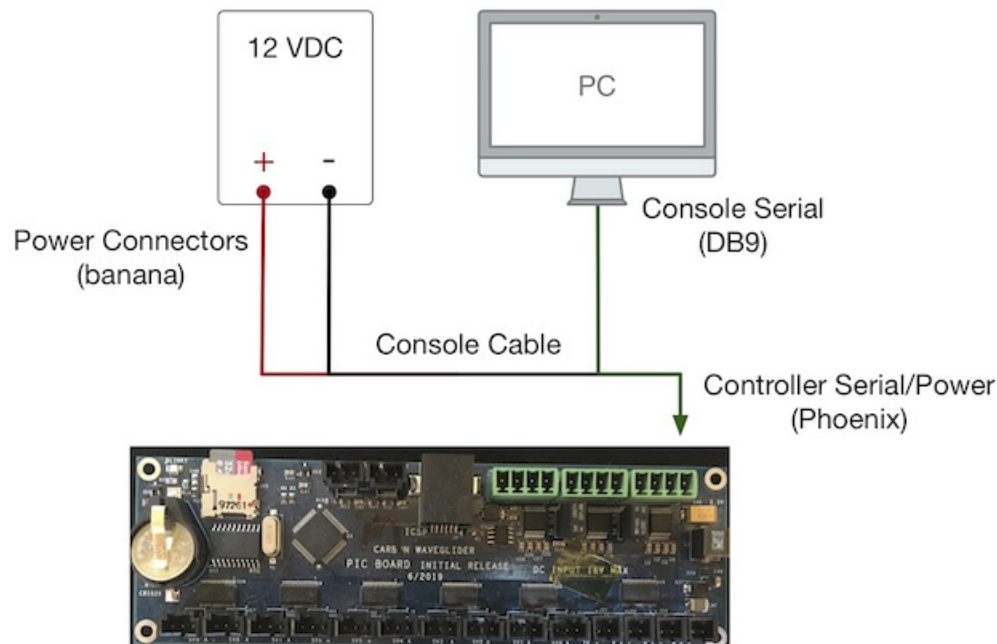
Materials

- Controller board (with firmware installed)
- Console cable
- 12V/1A DC power supply
- Computer with serial port or USB serial adapter
- Anti-static surface (recommended)
- Terminal emulator (e.g. minicom, Hyperterm)

Connect Hardware

If the controller hardware isn't mounted in an enclosure, it is recommended to operate it on a static-safe surface and observe good hardware handling practices.

Connect the controller hardware to the power supply and computer using console cable, as shown in the following diagram.



Hardware Connections

Start Terminal Session

Open a console session to the controller using a terminal program with serial communications settings

Setting	Value
speed	9600 bps
parity	none
data bits	8
stop bits	1
flow control	none
local echo	disabled
line wrap	enabled

It may also be necessary to adjust the following parameters; they may be expressed differently (or not exist), depending on the application.

Option	Value
terminal emulation	xterm, VT102, VT100
add carriage return	disabled
add line feed	disabled
backspace key	send CTRL-H, destructive backspace

When you enable power to the controller, the terminal should output a start up message similar to this:

```
serial IO [OK]
sdcard mount [OK]
log init [OK]
Loading config...cfg load [OK]
running boot script...
00:00:03
bootstrap [OK]

MBARI CO2/DIC
firmware[ cwg_nonboot v0.0.0 tag 2.0.0b0]
build [ Oct 15 2020 - 21:53:04 ]
cfg_sn [ C024 ]
sys_sn [ FFFF ]

>>
```

The '>>' characters are referred to as the command prompt. These may be shown for reference in this document, but they should not be entered by the user.

The user interface is case-insensitive: user input is converted to upper-case before processing. However, there are a few commands with interactive modes that support case-sensitive searches (e.g. less).

Upload Script

The script linked to below is a simple script that

- prints messages to the console
- shows firmware version information
- shows elapsed time since system start

The code may be found in the documentation package [here](#)

```
# print a message to the console
println "\r\nhello world"

# show version information
println "\r\nversion:"
ver

#show the system uptime
println "\r\nUptime:"
uptime

END
```

The details will be described further in the scripting guide.

It may be uploaded directly from the examples directory or you may first copy the script to a text file. The file transfer to the controller is initiated by issuing the upload command at the console.

```
upload hello.scr
```

The name given to the upload command is the name of the file written on the controller. It does not have to match the file name on the host computer. File names on the controller use the DOS 8-dot-3 format: <name>.<extension>, where the name length is limited to 8 characters, and extension to 3 characters.

Also, script may use any valid filename; by convention, the .SCR extension is recommended.

When controller responds with

```
Waiting for file path[FOO.TXT] tmout[15000]
Interrupt/end using CTRL-D
```

use the terminal program to send the text file; this is typically done using a menu option to send a file, which varies per application.

In minicom, for example, CTRL-A Y opens a navigation menu, where you navigate to the file and select it.

From the hyperterm menu bar, use Transfer >> Send Text File... then find and select the script file.

With some terminal programs, you may simply copy and paste the script text directly into the terminal window.

The upload command times out after 15s if the transfer ends or hasn't started. If that happens, use the N option to set the upload timeout. For example, to set a 30s timeout, use:

```
upload N30 hello.scr
```

Each line of the file is displayed on the console as the file is uploaded. Note that if you type other characters into the console during the upload session, they will be put into the file. When the script upload is finished, wait for the timeout to expire or end the upload session typing CTRL-D (press and hold CONTROL then D keys) on the console.

The CAT command may be used to examine the contents of the uploaded script:

```
cat hello.scr
```

It should look like the contents shown above.

Run Script

The SCR (“script”) command is used together with its sub-commands to manage and operate scripts. In order to be run, scripts must first be loaded into a “slot”, which is a data structure that keeps track of the script’s state. There are currently four script slots, numbered 0:3. These slot numbers are also referred to as handles.

In the context of script commands, scripts may be identified by their name or handle. Since handles are shorter, they are convenient to use; both are used in this document.

Load the script using:

```
scr load hello.scr
```

To inspect the script slots and find a script handle, use the `show` directive (sometimes referred to as sub-commands):

```
scr show
```

which responds with something similar to

```
SCR[0]
[self      580a]
[ctx       580c]
[name      HELLO.SCR]
[fp        581c]
[state     x0001]
[modes     x0000]
[events    x0000]
[error     0]
[tmr[0]    {N,0,0}]
[tmr[1]    {N,0,0}]
[period    0]
[line      0]

SCR[1]
[self      5862]
[ctx       0]

SCR[2]
[self      58ba]
[ctx       0]

SCR[3]
[self      5912]
[ctx       0]
```

We can see that SCR[0] (handle 0) contains HELLO.SCR, and that the other 3 slots are empty. The other state/status information will be covered later.

To run hello.scr, use the `run` directive (i.e. sub-command); here, we’ll use the name, but you could also use the handle (0):

```
scr run hello.scr
```

with corresponding output similar to:

```
HELLO WORLD
```

```
VERSION:
```

```
MBARI CO2/DIC  
firmware[ cwg_nonboot v0.0.0 tag 2.0.0b0]  
build   [ Oct 15 2020 - 21:53:04 ]  
cfg_sn  [ C024 ]  
sys_sn  [ FFFF ]
```

```
UPTIME:  
00:52:03
```

Optionally, you may unload the script as with the `unload` script directive; here, we'll use the handle:

```
script unload 0
```

Having successfully

- set up the system
- familiarized yourself with the user interface
- installed and run a script

see the next section for a deeper dive into system features and operation.

[Contents](#)

System Overview

Getting things done

This section takes a closer look at system operation. After learning how to get around the user interface and an overview of the controller command set, explore fundamental operations that enable workflows in the lab and deployment contexts. Along the way, pick up what you need to know about the underlying system architecture and concepts.

Fundamental Operations

Navigating	Commands and how to use them
IO control	Actuating valves and pumps
Accessing information	Inspecting state and status
Making measurements	Measurement workflow fundamentals
Inspecting data	

Navigating

Interactions with the system take place through the serial console (terminal). Users or host computers send text commands to the controller, which typically returns a response. Commands are the primary means of getting things done, exposing all of the system’s capabilities to the user or host computer.

Command Structure

Commands consist of a command, which may be followed by a directive (sub-command), zero or more options specifying command behavior, and/or zero or more arguments:

```
command [directive] [options...] [arguments...]
```

for example

```
set bits v3 v4a pr
```

where *set* is a command, *bits* is a directive and *v3 v4a pr* are arguments. A directive may be thought of as an argument that is also a command, with zero or more arguments. Directive arguments lists use a special syntax to distinguish itself from other command arguments. Directives may also have arguments that are lists that are distinct from other arguments:

```
command <directive>:<li>[...],<dir_arg>[,<dir_arg>...] <cmd_arg>
for example:
mdio ioh:ul/li/pw,20000 vex
```

It is useful to think of the commands in functional groups:

Functions	Description	Commands	Directives (sub-commands)
-----------	-------------	----------	---------------------------

Functions	Description	Commands	Directives (sub-commands)
<i>Application</i>			
IO Control	measurement	MDIO	IOH,IOL SHL,SLH PHL,PLH
Measurement Control	measurement	MMEAS	CYC USRn LOAD TM OUTR,OUTM,OUTn SHOW
LI-COR	LI-COR control	LI	CAL INIT RATE STAT
Logging	Log inspection	LSHOW LLESS LINFO LCAT LOG LSEQ LSEG	
Queue	Measurement queues	QCOUNT QCLEAR QPEEK QPOP QSHOW	
Scripting	Script mgmt	SCR	LOAD UNLOAD SHOW RUN KILL SCH SYNC RESET JOG END DELAY SDEBUG
		CRON	
<i>IO Control</i>			

Functions	Description	Commands	Directives (sub-commands)
Get	State info	GET	BITS AD VAR TIME NEXT PRINT SER
Set	Change IO,settings	SET	BITS TIME VAR PRINT SER BAUD
Clr	Change IO	CLR	BITS
Pulse	Cycle IO		BITS NBITS
<i>Logic, Variables</i>			
Test	Logical IO Tests	TEST	BITS
Register	Built-in variables	CREG DREG FREG	
<i>System</i>			
File System	Manage, view files	LS DIR CAT LESS CP MV RM REN UPLOAD DSTAT LC CFGEX	
System	System utilities	HELP ? HIST ! RESET SLEEP UPTIME	
Diagnostic	Diagnostic, debug	CON PMEM PRINT PRINTLN HELLO SWEEP	

The groups are sometimes referred to as APIs (Application Programming Interface). All of the commands are fully documented in the controller help guide.

Commands, options and arguments that make up the user interface are designed with several guiding principles in mind:

- simplicity (easy to type and understand)
- human- and machine-friendly
- balance minimal command length with readability
- structural and behavioral consistency
- one good way to do anything : full, fine-grained functional exposure with minimal redundancy
- economic with system resources

To that end, these conventions are used:

- `<foo>` indicates an option named foo, which has yet to be defined
- In the context of a command line use note, arguments in square brackets e.g. `[ <foo> <bar> ]` are optional but may contain required elements. For example `[op <var>]` indicates that you may use `'op <var>'` but not just `'op'` or `'<var>'`
- In the context of command notes, options in square brackets may indicate set of literal values `[cXn|\\*]` or `[=,-,++,+=,-.=,*=,/=,%=]`
- Options with ellipses `'...'` indicates that the preceding option may be repeated e.g., `[<foo>...]` means one or more `<foo>` options
- Options don't use dashes (H not -H)
- Except where it is ambiguous, there is no space between option arguments and values e.g. `x1.23, F003`
- A colon is used to disambiguate options and arguments; e.g. a command has options S and S2 with numeric values - `S:123 S2:1.34`
- A colon is used to delimit multi-valued options (i.e. a list), which are comma-delimited e.g. `out:cr,pdh`
- If an option list contains multi-valued variables (sub-list), the sub-list is delimited using `'/'`, e.g. `sh1:v1a/v3,5000`
- `*` is a wildcard, signifying all/any

Some commands allow a variable number of arguments and options, enabling multiple operations to be specified on a single line. Command lines are limited in length to 127 characters (including spaces); this constraint is imposed by the size of the text buffer used to capture user input.

History Buffer

The user interface includes a short (12 line) history buffer. Pressing the up/down cursor keys scrolls up and down through the history buffer. Examine the history buffer by using the HIST command (or shortcut !).


```
>>hist
00 GET VAR
01 GET VAR LICOR_MODEL
02 GET VAR LICOR_MODEL ECHO
03 GET VAR LI_SPANC
04 SET VAR LI_SPANC 1.234
05 GET VAR LI_SPANC
06 CFGEX MYCONF.TXT
07 CAT MYCONF.TXT
08 LS
09 DIR
10 LC SYS2009.000
11 HIST
12 CLR BITS V* P*
13 GET BITS
14 LD
15 LS
```

The remaining sections focus on commands and workflows that enable lab work and deployments.

Contents

IO control

For the DIC/PCO2 application, the primary functions of the controller are configuring the flow of fluids and gases through a manifold, then configuring the LI-COR gas analyzer and recording its measurement (observation) output under various conditions.

Controlling IO is done by actuating various types of pumps and valves using specified sequences and timing. The physical devices involved use different methods of actuation, but at a fundamental level, the controller is largely devoted to switching IO bits on and off at the right time.

This section describes the various types of IO and actuation used in this application.

IO Types

The controller has both digital and analog IO (input/output).

The system's digital IO represent binary logic values that have 2 states: HIGH and LOW (true/false).

HIGH represents a logically asserted (TRUE) state; you may see it referred to using 'HI','H','1','Y','TRUE','T'.

LOW represents a logical un-asserted (FALSE) state; you may see it referred to using 'LO','L','0','N','FALSE','F'.

By convention, we typically use H/L, HI/LO or 0/1.

Electrically, the logical states are represented by DC voltage signal levels, which may be 3.3, 5 or 12 VDC for HI, and 0 VDC for LO.

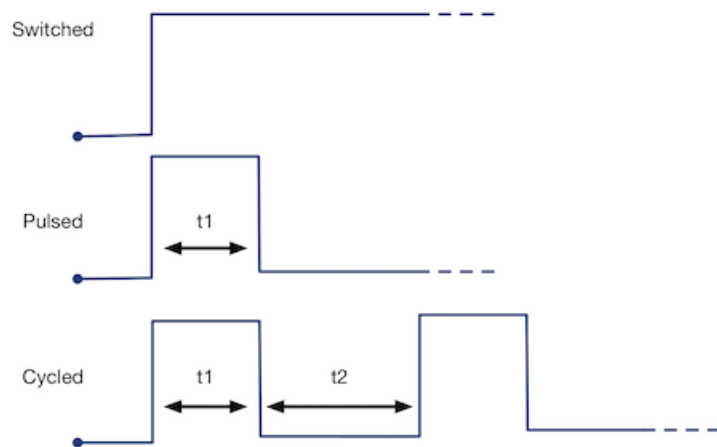
Each of the digital IO bits in the system are named (and indexed), and represent a dedicated physical processor output pin:

Name	Bit	Function	signal level
SV0A	00	Valve port 0/A	12VDC
SV0B	01	Valve port 0/A	12VDC
SV1A	02	Valve port 1/A	12VDC
SV1B	03	Valve port 1/A	12VDC
SV2A	04	Valve port 2/A	12VDC
SV2B	05	Valve port 2/A	12VDC
SV3A	06	Valve port 3/A	12VDC
SV3B	07	Valve port 3/A	12VDC
SV4A	08	Valve port 4/A	12VDC
SV4B	09	Valve port 4/A	12VDC
SV5A	10	Valve port 5/A	12VDC
SV5B	11	Valve port 5/A	12VDC
SV6A	12	Valve port 6/A	12VDC
SV6B	13	Valve port 6/A	12VDC
SV7A	14	Valve port 7/A	12VDC
SV7B	15	Valve port 7/A	12VDC
SV8A	16	Valve port 8/A	12VDC
SV8B	17	Valve port 8/A	12VDC
SV9A	18	Valve port 9/A	12VDC
SV9B	19	Valve port 9/A	12VDC
PUMP0_DC	20	Pump port 0	12VDC
PUMP1_DC	21	Pump port 0	12VDC
PUMP2_DC	22	Pump port 0	12VDC
PUMP3_DC	23	Pump port 0	12VDC
LICOR_DC	24	LI-COR DC power	12 VDC
ADC0	25	ADC channel 0 enable	3.3VDC
ADC1	26	ADC channel 0 enable	3.3VDC
ADC2	27	ADC channel 0 enable	3.3VDC
XCVR_HOST	28	Host transceiver enable	3.3VDC
XCVR_LICOR	29	LI-COR transceiver enable	3.3VDC
XCVR_CONS	30	Console transceiver enable	3.3VDC

The controller has three methods of actuating digital IO: switched, pulsed and cycled. The different methods reflect the ways that the physical devices attached to them work, and other design constraints. The latching relays used to operate the valves, for example, can reduce power consumption.

The actuation types are described and represented graphically below.

Type	Description	Use
Switched	static, set HI or LO	air pump seawater pump LI-COR power ADC and tranceiver enable
Pulsed	single pulse with specified width (t1): H,t1,L	valves (latching relays)
Cycled	multiple pulses using specified duty cycle H,t1,L,t2...	acid pump sample pump



Actuation methods

The relevant user interface command sets for operating digital IO include MDIO, SET, CLR, PULSE.

MDIO is typically recommended for general purpose use. In addition to exposing all of the IO (analog and digital) and actuation types, it exposes named IO presets (macros), that perform common application-specific IO operations with the required timing. The IO presets have corresponding measurement presets, making measurements easy to script. For example, the command sequence

```
mdio sample
mmeas sample
mdio mterm
```

- configures relevant manifold valves and air flow for a sample measurement and turns on the LI-COR.
- after a delay, acquires and logs a series of measurements using a set of pre-defined measurement parameters (which may be overridden)
- turns off the LI-COR and air pumps after the measurement cycle is complete

The SET, CLR, PULSE commands may also be used, and are especially useful for composing operations not covered by existing macros, and also for general manual operation of the system.

The MDIO and SET/CLR/PULSE command sets use common syntax for representing IO. For example, SV0A is v0a, and the valve pair SV3A and SV3B are referred to as v3. The pump ports may be referred to functional mnemonics as well, e.g. pa is the acid pump, pw is the seawater pump.

While these valve and pump mnemonics are the recommended way to reference IO, they may be referenced by index (p0, p1, etc.) and by logical bit indices using decimal or hex notations. All of these are described in detail in the controller help guide.

MDIO and MMEAS are covered in more depth in the measurement section, and in the controller help guide.

Contents

Accessing information

Having visibility into the state and status of the system is critical for operating the system. This section provides a brief overview of commands for inspecting various aspects of the system, including

- Digital and analog IO

- Configuration values
- System logs and other files
- Date and Time
- Register variables

The commands used for inspecting data logs, measurement queues and scripts are covered in the related sections on those topics.

Digital and analog IO

The GET command includes a number of sub-commands for inspecting the state of digital and analog IO.

GET BITS may be used to inspect the current state of all digital IO. The following command sequence demonstrates CLR/SET/GET BITS:

```
# clear all pump and valve IO bits (switched)
>>clr bits v* p*

# confirm all pump and valve bits 0 (LO)
>>get bits
bit      SVn      (A/B)
[ 0/ 1] SV0       0/0
[ 2/ 3] SV1       0/0
[ 4/ 5] SV2       0/0
[ 6/ 7] SV3       0/0
[ 8/ 9] SV4       0/0
[10/11] SV5       0/0
[12/13] SV6       0/0
[14/15] SV7       0/0
[16/17] SV8       0/0
[18/19] SV9       0/0
[ 20] Pump0 DC    0
[ 21] Pump1 DC    0
[ 22] Pump2 DC    0
[ 23] Pump3 DC    0
[ 24] Licor DC    0
[ 25] ADC0        0
[ 26] ADC1        0
[ 27] ADC2        0
[ 28] HOST  XCVR  1
[ 29] LICOR XCVR  1
[ 30] CONS  XCVR  1

# set bits v3a/b, v5a, pump0 and seawater pump (pump1)
>>set bits v3 v5a p0 pw

# use GET BITS to confirm
>>get bits
bit      SVn      (A/B)
[ 0/ 1] SV0       1/0
[ 2/ 3] SV1       0/0
[ 4/ 5] SV2       0/0
[ 6/ 7] SV3       1/1
[ 8/ 9] SV4       0/0
[10/11] SV5       1/0
[12/13] SV6       0/0
[14/15] SV7       0/0
[16/17] SV8       0/0
[18/19] SV9       0/0
[ 20] Pump0 DC    1
[ 21] Pump1 DC    1
[ 22] Pump2 DC    0
[ 23] Pump3 DC    0
[ 24] Licor DC    0
[ 25] ADC0        0
[ 26] ADC1        0
[ 27] ADC2        0
[ 28] HOST  XCVR  1
[ 29] LICOR XCVR  1
[ 30] CONS  XCVR  1
```

GET AD is used to look at the current value of one or more ADC channels:

```
>>GET AD 0 1 2
ADC0 : 0.0000 V
ADC1 : 0.0000 V
ADC2 : 11.2559 V
```

Here, only ADC2, the battery monitor channel, is indicating input; and is calibrated for the battery voltage (indicating 11.26V). ADC channels 0 and one are calibrated for 0-5V input.

Configuration values

System configuration values are read from a file (CONFIG.TXT) and stored in memory. GET and SET include the VAR directive for inspecting and changing configuration values.

```
# get two specific values
get var licor_model echo
    LICOR_MODEL =      850
    ECHO =            1

# get all values
>>get var
[ self      5ada]
[ key      VERBOSE]
[ val      0]
[ key      ECHO]
[ val      1]
[ key      LOG_RAW]
[ val      1]
[ key      AUTO_OUT]
[ val      0]
[ key      LICOR_MODEL]
[ val      850]
[ key      LI_SPANC]
[ val      3.14]
[ key      LI_CFLAGS]
[ val      X000003FF]
[ key      LI_OFLAGS]
[ val      X001F7FFF]
[ key      SERIAL_N]
[ val      XC024]
[ key      VALVE_TYPE]
[ val      X004D]
[ key      S0]
[ val      ]
[ key      S1]
[ val      ]
[ key      S2]
[ val      ]
[ key      S3]
[ val      ]
[ key      FOO_BYTE]
[ val      X58]
[ key      FOO_S]
[ val      Ersatz String]
[ key      FOO_U32]
[ val      XCAFEDEAD]
[ key      FOO_I32]
[ val      1234567]
[ key      FOO_DBL]
[ val      1.23457e+07]
```

By the same token, SET may be used to set values:

```
>>get var li_spanc
    LI_SPANC =      3.14

>>set var li_spanc 1.234

>>get var li_spanc
    LI_SPANC =      1.234
```

Values set in a console session are made in memory, but will not persist across system restarts unless they are written (exported) to the configuration file. The CFGEX command is used to write the current configuration in memory to CONFIG.TXT or another file (shown here):

```
>>cfgex myconf.txt

>>cat myconf.txt
VERBOSE 0
ECHO 1
LOG_RAW 1
AUTO_OUT 0
LICOR_MODEL 850
LI_SPANC 3.14
LI_CFLAGS X000003FF
LI_OFLAGS X001F7FFF
SERIAL_N XC024
VALVE_TYPE X004D
S0
S1
S2
S3
FOO_BYTE X58
FOO_S Ersatz String
FOO_U32 XCAFEDEAD
FOO_I32 1234567
FOO_DBL 1.23457e+07
```

Configuration variables are strongly typed, meaning that they are stored using a specific data primitive. When using SET VAR to change the value of a configuration variable, the value must be formatted appropriately for its native data type.

SET VAR parses values using C printf formats; it will typically try several formats; hex values should be prepended with 'X' disambiguate them from decimal values.

Type	Internal Representation	Default Format	Example
bool	boolean	X%02X	set var FooBool X07
uint16	unsigned 16-bit integer	X%04X	set var FooU16 X0A27
int16	signed 16-bit integer	%d	set var FooI16 42
uint32	unsigned 32-bit integer	X%08lX	set var FooU32 X1234C02
int32	signed 32-bit integer	%ld	set var FooI32 -8675309
uint64	unsigned 64-bit integer	X%016llX	set var FooU64 X123DEADBEEF
int64	signed 64-bit integer	%lld	set var FooI64 10000000000
double	32-bit double	%ld	set var FooI32 -3.14159
byte	unsigned char	%c	set var FooByte Z
string	NULL terminated ASCII string	%s	set var FooStr "foo bar"

String variables have a maximum length of 64 bytes (including terminating NULL)

System logs and other files

As mentioned in the Tour section, there are a number of files maintained on the SD Card file system. Many of the files are ASCII text files (UTF-8 character set); data summary logs (DATYYMM.NNN) are stored in a binary format.

The LS command (and alternative DIR) are used to show a summary listing of the files on the SD Card:

```
>>ls
----A 2020/08/28 04:10      242 CONFIG.TXT
----A 2020/10/06 05:16    43843 HELP.MD
----A 2019/09/18 10:52     240 PMEL.SCR
----A 2020/10/07 05:18        4 DATSEQ
----A 2020/10/05 22:22     546 DICM1V1.SCR
----A 2020/10/07 05:18     4200 DAT2010.000
----A 2020/10/18 00:35    29691 SYS2010.000
----A 2020/10/07 05:18   648978 RAW2010.000
----A 2020/10/10 04:15       80 CRONTAB.SYS
----A 2020/10/10 05:23       45 CRONTAB.USR
----A 2020/10/17 00:15      171 SYSBOOT.SCR
----A 2020/10/17 05:49      179 HELLO.SCR
----A 2020/10/18 00:38      243 MYCONF.TXT

46791.26 KiB in 13 files 1 directories
1716064 KiB free
```

sharp-eyed readers will note that the numbers in the example output don't add up. This example was edited for brevity).

The CAT and LESS commands are provided to examine text file contents; they are not compatible with binary files; commands for binary files are described in later sections.

By default, CAT simply outputs the contents of a file to the console. This is useful for short files. CAT includes advanced options for displaying portions of a file by defining start and end conditions, based on line numbers, bytes and timestamps (for system logs). In this way, it's possible to look at the last few lines of a file, or find events in the system log corresponding to a certain time.

It's companion, LESS, works much like it's desktop counterpart: it provides a simple interactive pager that enables users to jump forward and backwards, and even search the file for matching text. LESS provides a help summary when called:

```
>>less sys2009.000

      Command Summary
Move  :   Up [u U]   Down [d D CR]
Jump  : First [g <] Last [G >]
Search: Find [f /]  Next [n N]
Session: Help [h H ?] Quit [q Q x X]

2020-09-18T03:11:37Z,sys reboot rcon[x20C3],
2020-09-18T03:17:38Z,sys reboot rcon[x0044],
2020-09-18T03:35:07Z,sys reboot rcon[x20C3],
```

The LC command counts the number of lines in the specified file and also returns the time it took to count them. LC counts ~1k lines/sec, so it can take a long time for long files. It may be interrupted by any user input.

Date and time

The GET/SET TIME sub-commands are also used to read and set the real-time clock (RTC).

```
>>get time
2020-10-18T03:40:35Z

>>set time 2020 10 18 03 50 00
new time is:2020-10-18T03:50:00Z

>>get time
2020-10-18T03:50:03Z
```

Register variables

The controller implements a set of built-in variables that are accessible to users in the console and script contexts; these are referred to as register variables.

There are three types of register variables: counter registers (32-bit unsigned values), double registers (32-bit floating

point values) and flag registers (single boolean bits). There are 8 counter and double registers and 32 flag registers.

They are intended to be used in scripting context to implement loops, conditional branching, or inter-script communication. Some commands (GET AD, e.g.) allow their output to be stored in a register value:

```
>>get ad 2:dx0
ADC2 : 11.2588 V DX0
>>dreg dx0
DX0 = 1.125879e+01
```

MDIO also allows delays to be set using a register variable. The PRINT and PRINTLN commands support printing register variables, which is useful for format conversion when debugging.

Note that variable substitution is only supported in specific contexts rather than across all commands.

Setting and inspecting registers is done using commands CREG, DREG and FREG.

These commands include options for displaying one or more register values, and simple operator expressions to modify them. The expressions may be chained on to perform multiple operations on a single line:

```
#>>creg cx0=0 cx0 cx0-- cx0 cx0+=2 cx0 cx0\*=4 cx0
CX0 = 0
CX0 = 4294967295
CX0 = 1
CX0 = 4
```

Because they are globally accessible, care must be taken when modifying register variables or sharing them between scripts, as there are no protections against concurrent modification. This could potentially lead to unexpected behavior in scripts, for example.

Contents

Making measurements

As previously noted, the typical measurement cycle consists primarily of

- Changing IO states to move fluid and gas through a manifold to produce different measurement conditions (air, zero, standard, sample, etc.)
- Using the LI-COR gas analyzer to make a series of observations (measurements) under each condition

The exact steps vary depending on context, but the fundamental operations are similar in other contexts as well, such as performing a calibration or developing measurement protocols or data workflows.

In this section, we'll look more closely at the commands for making measurements. Though it will be shown how to log and display measurements, details about data infrastructure are presented in a later section.

Measurement by example: composing a measurement cycle using MDIO and MMEAS

The MDIO command has a number of sub-commands that enable fine control over IO operations. It accepts multiple arguments and options, making it possible to complete several actions with a single line. In addition, a number of IO presets (macros) are defined, short-cuts for actuating sets of commonly used valves, pumps, etc.

MDIO and MMEAS options are fully documented in the controller help guide; here, we'll take a look at how these are used the context of a typical measurement cycle.

In this section, we'll construct a complete measurement cycle step by step, building on the information presented so far,

and adding new material as we go.

Step 1: Power and Valve Setup

We'll start with the MDIO command and IOH directive to enable the LI-COR UART transceiver, and run the VEX macro, which exercises the valves by opening and closing them in a specified sequence.

```
# enable licor transceiver, exercise valves
mdio ioh:ul vex
```

Sidebar: command and directive syntax

Controller commands are executed in the order written (left to right). Here, the IOH operation will happen before the VEX macro. While not the case in this line, order matters.

Also, note the structure of the IOH directive. As noted earlier, directives use a colon to delimit its argument list. Here, IOH only had a single argument (UL, the IO bits to switch), but it could also accept additional IO bits, as well as a post-switching delay.

If we wanted to turn on the air and seawater pumps at the same time and then wait for 20 seconds (20000 msec), we could use

```
mdio ioh:pa/pw,20000
```

As detailed in the Navigating section, IOH is an example of directive syntax; directives are arguments are commands, and IOH's arguments are comma-delimited and include an argument that is a list (delimited using '/').

Step 2: Initialize Measurement Buffers

On the next line, the MMEAS CYC directive is used to initialize the measurement data buffers at the start of the cycle:

```
# reset cycle buffer, set record type
mmeas cyc:rst,rtz0,rmsample
```

Here, the RST argument resets the measurement cycle buffer; more about data buffering shortly. The RTZ0 argument sets the record type for the measurement cycle. Record type identifies the data structure version; there is currently only one type, RTZ0.

RMSAMPLE sets the record measurement ID, which identifies the general context for the measurements it contains (each of which has its own identifier).

RMSAMPLE indicates a sample cycle; other types include RMZCAL (zero-calibration), RMSCAL (Standard Calibration), and RMX (a user-defined measurement ID).

To summarize, this step has initiated a new sample cycle and set the record identifiers for the measurements that will follow. Details on sampling and data buffering and structures will be provided later.

Step 3: Perform a ZCAL measurement

The measurements begin with a zero-calibration:

```
# set up manifold for zero calibration
mdio zcal

# take ZCAL measurement, and for debugging, sent measurement to the console
mmeas tm:zcal outm:qc,pdh
```

The MDIO/MMEAS pair is a typical pattern of use: perform an IO set-up sequence with an MDIO macro, followed by a corresponding set of measurements using an MMEAS macro.

The MDIO ZCAL macro performs a number of steps:

- switches a set of valves on the manifold
- turns on the air pump
- waits for a specified period
- turns the air pump power off and LI-COR power on

Next, the MMEAS first uses the TM (take measurement) directive to

- configure the licor output period
- take a specified number of samples, logging raw source data and building a data summary in real time during sampling
- place the data summary measurement in the cycle buffer and setting the measurement ID to MMZCAL

Here, the TM directive uses only the ZCAL macro; it is possible to override various macro parameters. For example,

```
MMEAS TM:zcal,B2000
```

overrides the default ZCAL blanking interval (1 sec, time that the LI-COR is running before starting measurements) with 2000 msec (2 sec).

Other TM overrides include:

P<n>	set LI-COR sample period (decimal sec, 0.0 or >=0.5) use: p (e.g. p0.5)
T<n>	set measurement time use: t<n> {e.g. t30000}
N<n>	set measurement sample count use: n<n> {e.g. n16}
B<n>	set blanking period (msec) use: b<n> {e.g. b2000}
W<n>	enable/disable calibration value write to instrument use: w[+-] (e.g. w+)
D<n>	delay for specified time (immediate, msec) use: d<n> {e.g. d12500}

it is possible to express multiple macro names (with optional overrides) in the same TM command:

```
MMEAS TM:zcal,scal
```

However, this is not typical, since different IO set up is generally needed for each measurement.

Finally, for debugging and demonstration, the OUTM (output measurement) directive places the measurement summary into a data queue, and also to the console. OUTM would not typically be used in this way in a deployment context.

The basic structure for the OUTM directive is shown below.

```
outm:[CHLQ],[PBRCTXHDV]

where

[CHLQ] - destination flags (one or more)
C : console
H : host
L : log
Q : queue

[PBRCTXHDV] - format flags (zero or more, for console only)

P : Pretty
B : Base85 (ASCII encoding, more efficient than uuencode)
C : CSV
R : raw
X : hex
XX : PrettyHex
X* : hex+PrettyHex
H : header
D : data
V : enable verbose content (if any)
```

So outm:qc,pdh sends the last measurement to the console and the queue. For console output, it uses pretty data format and includes both header and data, as shown below.

```
rtid 1
rmid 1
mmid 12
blank_ms 1000
meas_n 8
meas_ms 4000
period_s 0.500
to_msec 4800
cal_wr 43
op_id 12
self 124e
sync 216681474
type_id 0001
meas_id 0001
timestamp 1603082297000/2020-10-19T04:38:17.000Z
seq_n 0
meas_count 1
len 256
# Measurement
self 1268
timestamp[0] 1603082293500/2020-10-19T04:38:13.500Z
timestamp[1] 1603082297000/2020-10-19T04:38:17.000Z
meas_id 000C
stat_n 8
period 0.50
# min/max/M/S
co2 +1.048e+00 +1.467e+00 +1.325e+00 +1.272e-01
co2abs +3.140e-02 +4.390e-02 +3.966e-02 +3.800e-03
h2o +1.139e+01 +1.411e+01 +1.285e+01 +8.585e-01
h2oabs +8.964e+00 +1.218e+01 +1.073e+01 +1.019e+00
h2odewpoint +8.665e-02 +1.005e-01 +9.418e-02 +4.395e-03
celltemp +2.528e+01 +2.530e+01 +2.529e+01 +8.093e-03
cellpres +1.009e+02 +1.009e+02 +1.009e+02 +2.127e-03
ivolt +1.114e+01 +1.115e+01 +1.114e+01 +2.349e-03
flowrate +0.000e+00 +0.000e+00 +0.000e+00 +0.000e+00
raw_co2 +4.174e+06 +4.193e+06 +4.187e+06 +4.877e+03
raw_co2ref +4.777e+06 +4.811e+06 +4.801e+06 +8.618e+03
raw_h2o +2.056e+06 +2.059e+06 +2.058e+06 +1.062e+03
raw_h2oref +3.587e+06 +3.596e+06 +3.592e+06 +2.419e+03
# Metrics
co2-ep0 +1.467e+00
co2-ep1 +1.337e+00
co2-area +4.670e+00
MX +4.000e+00
SX +2.800e+01
SXY -1.636e-01
# CO2 Trend
co2_m -5.844e-03
co2_b +1.348e+00
co2_R -9.192e-02
co2_R^2 +8.450e-03
```

Step 4: ZCAL cleanup and SCAL measurement

The ZCAL measurement leaves the LI-COR running. The MDIO macro MTERM turns off power to the LI-COR and air pump; it is frequently used to turn these off between measurements.

We'll add MTERM to an MDIO/MMEAS command pair to do a standard calibration SCAL measurement:

```
# turn off LI-COR, air pump, set up manifold for SCAL
mdio mterm scal

# take SCAL measurement, and for debugging, send the measurement to the console
mmeas tm:scal outm:qc,pdh
```

Step 5: Making the seawater sample measurement

The next part of the cycle is the sample measurement. The setup for the sample measurement requires a number of steps; MDIO macros are used, and split between two MDIO calls for readability.

In the first MDIO call, the PURGE and ZERO macros empty the sample chamber and strips the CO₂. In the second, FILL, ACID and SAMPLE fill the chamber with seawater, inject ACID, then configure the manifold for measurement (which includes starting air flow and turning the LI-COR on).

The MMEAS TM directive is used to read and log LI-COR data, this time with SAMPLE measurement profile.

```
# purge, fill, prepare sample
mdio purge zero
mdio fill acid sample

# sample meas
mmeas tm:sample outm:qc,pdh
```

Step 6: Air measurement, record logging

The final measurement in this cycle is a measurement of ambient air. This step resembles the sample measurement step:

```
# air sample, log record
mdio mterm purge air
mmeas tm:air outm:qc,pdh outr:ql
```

After wrapping up the previous measurement by turning off the LI-COR and air pump, the MDIO PURGE and AIR macros set up the air measurement.

There is a new OTR (output record) directive on the MMEAS line. In this critical step, the OTR directive causes the set of measurements in the cycle buffer to be written to the binary summary log as a single record (using the metadata we set up at the beginning of the cycle).

The OTR directive here uses the Q and L options to indicate that the record be placed in the cycle queue (which contains the last completed data record) and also in the data log. Placing the record in the queue is done so that a host computer can retrieve it, or so that we may inspect it for debugging (more about this in the section on accessing data).

Step 7: Ending the cycle

At the end of the cycle, the manifold should be put into a state that will keep it functioning reliably. To do this, the fluids and gases need to be purged, devices powered down to conserve energy, and valves put into a sealed resting state.

This is accomplished with an MDIO call:

```
# zero, rest, disable LI-COR tranceiver
mdio mterm zero rest iol:ul
```

The REST directive sets the manifold valves in a safe state between cycles. The IOL directive, a counterpart to IOH for switching IO to a LO state, turns off the LI-COR UART transceiver.

Putting it all together

The complete cycle appears below, with the scripting END directive at the end. This could be placed into a text file, uploaded and used to run a complete measurement cycle in the lab.

```
# enable licor tranceiver, exercise valves
mdio ioh:ul vex

# reset cycle buffer, set record type
mmeas cyc:rst,rtz0,rmsample

# get zcal meas
mdio zcal
mmeas tm:zcal outm:qc,pdh

# get scal meas
mdio mterm scal
mmeas tm:scal outm:qc,pdh

# purge, fill, prepare sample
mdio purge zero
mdio fill acid sample

# sample meas
mmeas tm:sample outm:qc,pdh

# air sample, log record
mdio mterm purge air
mmeas tm:air outm:qc,pdh outr:ql

# zero, rest, disable LI-COR tranceiver
mdio mterm zero rest iol:ul

END
```

In the next section, learn more about measurement data, where it is stored, and how to access it.

Contents

Data processing and inspection

In the previous section, some data was generated and logged in the context of a measurement cycle. In this section, data processing and logging details are presented, together with some methods for inspecting data via the user interface.

Where to Find Data

There are several places where measurement data may be found on the controller, summarized in the following table:

Location	Description	Access
Measurement cycle buffer	Circular buffer used by MMEAS	N/A
Measurement and record queues	Circular queues for debug and host	QUEUE
Summary data log	Measurement summary records	LLESS, LCAT, LINFO
Raw data log	Raw LI-COR XML records	CAT, LESS

These will be described in the sections that follow. To make sense of where and how data are stored, it would be helpful to

establish the underlying concepts surrounding data and structures used in this application.

Controller Data Model

The data pipeline for the DIC/pCO₂ controller application starts with the LI-COR gas analyzer. When it's in measurement mode, the LI-COR emits a stream of XML records at a maximum rate of 2 Hz (typically used for this application). The XML record contents represent one multi-variate data vector at a point in time, and are somewhat configurable. For this application, we capture all of the ~13 parameters that the LI-COR produces.

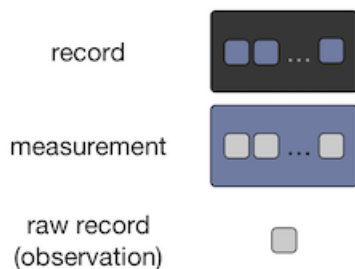
In the application lexicon, these are referred to as *raw records*, or *LI-COR records*. They look something like this:

```
<li850><data><celltemp>2.67288e1</celltemp><cellpres>1.00757e2</cellpres><co2>3.84294</co2><co2abs>8.9066209e-2</co2abs><h2o>1.58870e1</h2o><h2oabs>1.0847693e-1</h2oabs><h2odewpoint>1.39659e1</h2odewpoint><ivolt>1.1140673e1</ivolt><raw><co2>3964586</co2><co2ref>4778925</co2ref><h2o>2001193</h2o><h2oref>3576335</h2oref></raw><flowrate>0</flowrate></data></li850>
```

Again, these raw records are a single observation representing a set of *parameters* (aka *fields*) at a single point in time. For these to be meaningful, there must be more than one, and the context in which they are taken must be understood.

In our data model, a sequence of observations taken at the same time in the same context (set of measurement conditions) are referred to as a *measurement*. For example, we may collect a number of records over a period of several seconds while ambient air is being moved through the LI-COR, comprising an air measurement.

Along the same lines, it is useful to aggregate a set of related measurements taken at approximately the same time; here, we refer to such a measurement group as a *record*, and the process (set of operations and timing) for periodically gathering these measurements is referred to as a *measurement cycle*.



Data Hierarchy

To be accurate, the measurements that comprise a record don't simply contain raw records. The size of the raw records and the number required to make a meaningful measurement makes it impossible to fit them into the available memory. Even if it were possible, it may make telemetry impractical over expensive and/or limited bandwidth communication links, e.g. satellite or SMS (text messaging).

To address these constraints, measurements instead contain a condensed summary of the raw record data. When measurements are made, the min, max, mean and standard deviation of each parameter in the raw records is computed as each record arrives. Additional derivative calculations are also made, including;

- trend line parameters (slope, intercept) and correlation value for the CO₂ concentration
- initial and final CO₂ concentration values (end points)
- area under the CO₂ signal (parameter) curve
- CO₂ variance (SXY)
- time axis mean, standard deviation (MX SX)

These measurement summary data include additional metadata to fully capture the measurement context:

- timestamp
- number of raw samples used
- nominal period of the raw samples (LI-COR output period)
- a measurement ID (mmid) identifies the measurement conditions (MMAIR, MMZCAL, MMSCAL, MMSAMPLE, MMZERO, etc.)

Likewise, the enclosing records contain similar metadata to establish context:

- start and end timestamps
- record type ID (rtid, a record structure version identifier)
- record measurement ID (rmid) identifies the group of measurements (e.g. RMSAMPLE RMZCAL, etc.)
- sequence number
- number of measurements enclosed
- data size (bytes)
- a sync pattern (marks the start of a record, which can help to recover data should a data file become corrupted)

Measurement data

Primary measurement data structure and contents

timestamps

measurement start and end times

meas_id

specifies measurement condition, e.g. baseline, zero, span

stat_n

number of observations used for computation

period

licor output period (s) for this measurement

Measurement Data

One or more processed licor records. All measurements use this format

timestamps[2]
meas_id
stat_n
period
M[13]
S[13]
min[13]
max[13]
X[6]

CO2 trend line parameters may be calculated from mdata

slope
offset
R²
CO2 trendline slope
CO2 trendline offset
CO2 trendline correlation

min, max, M, S

Licor parameter minimum, maximum, mean (M) and standard deviation (S) values

co2
co2abs
h2o
h2odewpol
nt
h2oabs
celltemp
cellpres
ivolt
flowrate
rawco2
rawco2ref
rawh2o
rawh2oref
co2 concentration
absolute CO2
H2O concentration
dewpoint
absolute H2O
measurement cell
temperature
measurement cell pressure
voltage
gas flow rate
CO2 counts
CO2 reference counts
H2O counts
H2O reference counts

X

Calculated parameters

co2_area
co2_ep0
co2_ep1
MX
SX
SXY
area under CO2 curve
initial CO2 value
final CO2 value
x-axis (time) mean
x-axis (time) standard deviation
CO2 variance

Record Data Contents

The physical structure of this data on disk and in memory is described in subsequent sections. With an understanding of the data model, we can look more closely at how it is collected and where it is stored.

Cycle Buffer

Each time MMEAS TM directive is invoked, it takes one or more specified measurements. These measurements are placed in a circular buffer referred to as the *cycle buffer*. The cycle buffer contains a record that may include up to six measurements. The buffer depth is a compile-time option, LICOR_MAX_CYCLE_MEASUREMENTS, in cwg-shared.h (currently set to 6).

Each measurement entry uses ~256 bytes, and a full record uses a little over 1 kByte, or ~4% of the available memory (RAM). Though 4% may not seem like a lot, the available RAM is fairly fully allocated. It is used for a number of critical purposes, and must be balanced thoughtfully; use care when changing any memory allocations.

Since it is a circular buffer, the entries are overwritten when it is full, starting with the oldest.

As mentioned earlier, in a typical measurement cycle, we set the record metadata using the MMEAS CYC directive, and reset the buffer so that new measurements are directed to the start of the buffer. Then a series of measurements are taken using MDIO and MMEAS. Each measurement is placed in the cycle buffer record with its associated metadata. At the end of the cycle, the MMEAS OUTR is used to send the record, now filled with measurement summary data, to the summary data log and/or cycle queue (and/or console).

The cycle buffer itself is not directly accessible by users. Instead, MMEAS OUTM and OUTR directives are used to place measurements and records into queues that may be accessed by users and host computers.

These queues and the summary data log are described in subsequent sections.

Measurement and Cycle Queues

As mentioned above, data may be placed into the measurement and record queues as it is collected.

These queues are circular buffers that are accessible to users and host computers, using the QUEUE command set.

The measurement queue holds individual measurements; they include measurement metadata, but not record metadata. The measurement queue depth is set using compile-time option QUEUE_MEAS_DEPTH in log.h (currently 12, i.e. two record cycles).

The cycle queue holds a single complete measurement cycle record; i.e. it has a depth of 1. It's depth is set using QUEUE_CYCLE_DEPTH in log.h.

The QUEUE API provides commands for managing data in the queues. Conceptually, the queues implement a stack, or last-in-first-out (LIFO) buffer. The commands are summarized below, and fully documented in the controller help guide.

Command	Description
QCOUNT	return the number of entries currently in the queue
QPEEK	display (but don't delete) one or more queue entries in a variety of formats
QPOP	delete one or more queue entries, optionally log or display them first
QCLEAR	delete all queue entries
QSHOW	show a summary of the queue and it's entries

These commands accept an index argument ($I < n > | M | C$) to indicate which queue to operate on. The measurement queue is index 0, the cycle queue uses index 1.

Data output format examples are provided in the appendices.

General Log Concepts

The controller maintains a number of log files, including the summary data log, the raw data log and the system log.

Log naming Log files all use the same naming convention: XXXYYMM.NNN where

xxx	3-character name
yy	year
mm	month
nnn	segment

the summary, raw and system logs use the names DAT, RAW and SYS, respectively.

Log segmentation Logs may be segmented, i.e. distributed across multiple files on the SD card. The log segments may be defined in terms of time and/or size. That is, a new file may be automatically generated when certain conditions are met:

- if it does not exist when the controller is started
- at the start of each month
- when the file exceeds a specified size and size segmentation is enabled

Segmentation by size may be used for several reasons: First, if a file becomes too long, it can affect the sampling process as the time to traverse the file for reading or writing exceeds a critical threshold. Secondly, putting data into multiple files may prevent data loss or corruption in the event of system or user errors. Finally, smaller files are simpler and/or more convenient to transfer and share.

Currently, only the raw data log is enabled segmentation based on size using a limit of 20 MB. All log files are created at startup if they don't exist, and are also segmented by month.

Log formats Their contents use one of two general formats: ASCII (i.e. text) or binary.

As you'd expect, the size and growth rate of the logs is determined by the details of the measurement cycle and the frequency of measurements.

Inspection tools

LCAT and LLESS The LCAT and LLESS commands may be used to inspect the contents of the (binary) summary data log.

LCAT is intended for use by users and/or software clients; it is non-interactive, and enables one to specify a sub-set of records and format the output in human- and machine-friendly formats. For example, a host computer could use LCAT to request the last several records. (if it only wanted the most recent, it could also use QPEEK instead).

LLESS is an interactive tool; as such, it is not ideally suited for machine-to-machine transactions.

LCAT and LLESS options and examples are documented in the controller help guide.

Source code for LCAT and LLESS is provided in the repository, enabling them to run on Mac, or Windows (Cygwin required) computers.

CAT and LESS

The commands CAT and LESS may be used to examine the contents of text files. .

CAT is intended for use by users and/or software clients; it is non-interactive, and enables one to filter output using time

(for raw and system logs e.g., or lines, for example.

LESS is an interactive tool; as such, it is not ideally suited for machine-to-machine transactions.

So, CAT and LESS are the tools of choice for inspecting text files, i.e. the raw and system logs.

CAT and LESS options and examples are documented in the controller help guide.

CAT and LCAT both have an H option to view a help message:

```
cat h
```

LESS and LLESS both display help summaries at startup.

Summary Data Log_

Info	
Naming	DATYYMM.NNN
Format	Binary
Segment size	0 (unlimited)
Inspection	LCAT, LLESS

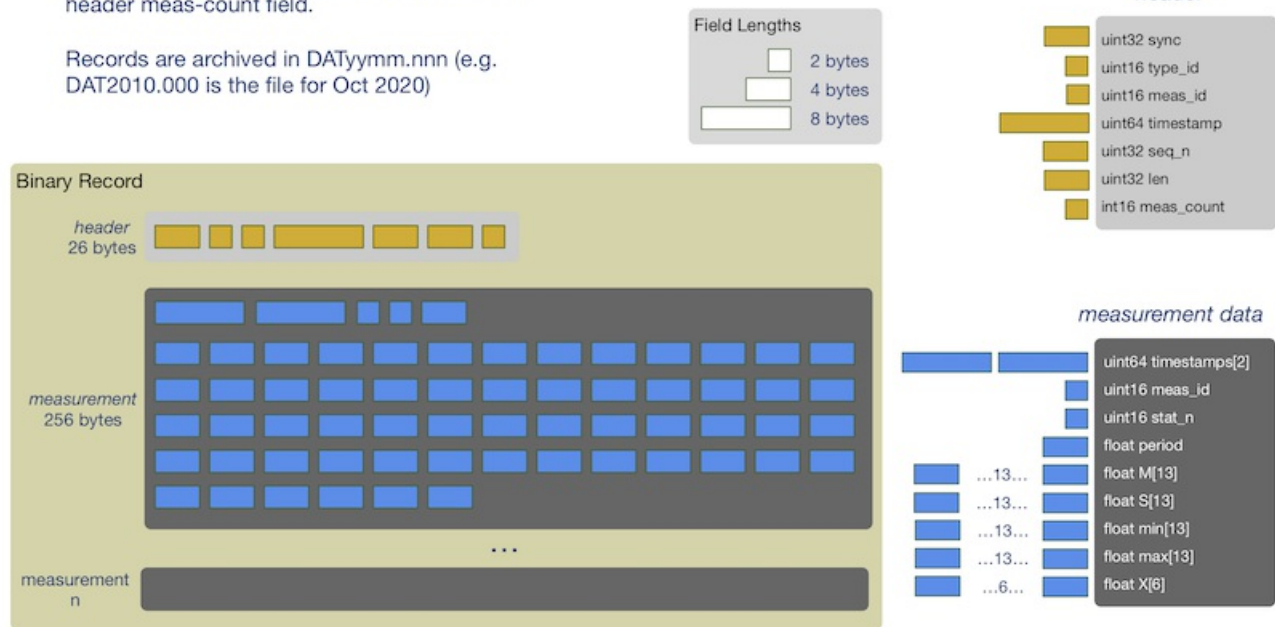
The summary data log is where the measurement cycle records are archived using the MMEAS OUTR directive (using using the QPOP OUT directive).

The contents of the summary data log are maintained in a binary format for performance reasons. The structure of the records is shown in the graphic below.

Binary record structure

Each binary record consists of a header and n (typically up to 6) measurements, indicated by header meas-count field.

Records are archived in DATymm.nnn (e.g. DAT2010.000 is the file for Oct 2020)



Binary Record Structure

The records may be of variable length, since they may contain different number of records, depending upon how your measurement cycle(s) are defined. As noted earlier, the records contain a sync pattern - a specified sequence of bytes - to delimit the start of a record, and also contain size metadata in the (fixed-size) header.

If you are writing software to read the file, the basic algorithm involves:

- find the sync pattern
- read the header (fixed size)
- validate and use the header information to determine the data length
- read the measurement data

If the file is intact, the next sync pattern should always follow the last byte of the previous record. If corruption has occurred,

- advance byte by byte until the sync pattern is located or until the end of the file.

A Matlab script, `read_log.m`, may be used to load summary data log files into Matlab. It is available in the repository.

Raw data log

Info

Naming	RAWYYMM.NNN
Format	ASCII
Segment size	20 MB
Inspection	CAT, LESS

The raw data log contains the archive of raw LI-COR data corresponding to the summary data records. The raw data records are formatted as comma-delimited ASCII, using:

```
<timestamp>,<id>,<xml>CRLF
```

where

Field	About	Example
timestamp	ISO8601 timestamp	2020-10-16T12:43:50Z
id	16-bit ASCII Hex: upper byte:RMID lower byte:MMID	0201
xml	LI-COR XML record	<li850>...</li850>

System Log

Info

Naming	SYSYYMM.NNN
Format	ASCII
Segment size	0 (unlimited)
Inspection	CAT, LESS

The system log records system events and errors. System log records are formatted as comma-delimited ASCII, using:

```
<timestamp>,<id>,<msg>CRLF
```

where

Field	About	Example
timestamp	ISO8601 timestamp	2020-10-16T12:43:50Z
id	16-bit ASCII Hex: upper byte:RMID lower byte:MMID	0201
msg	log message	

The LOG command may be used to append a timestamped message to the system log. This feature can be useful to commit

special contextual data to the data archive:

```
>>cat r:-,3 sys2010.000
2020-10-19T21:23:54Z,set [BITS UL],
2020-10-19T21:24:00Z,set [BITS UL LI],
2020-10-19T22:58:28Z,clr [BITS UL LI],

>>log "'noted a lot of growth near the intake'"

>>cat r:-,3 sys2010.000
2020-10-19T21:24:00Z,set [BITS UL LI],
2020-10-19T22:58:28Z,clr [BITS UL LI],
2020-10-20T01:29:39Z,'NOTED A LOT OF GROWTH NEAR THE INTAKE',
```

[Contents](#)

Debug features

There are a few features that are useful for debugging and/or just getting additional visibility into what the system is doing.

LED

Considering it's simplicity, the LED ("blinky") is very expressive. It is more than just a blinking light: it is a context-sensitive output that can provide insight into the software state should faults occur.

Under normal operating conditions the LED is used to indicate activity state:

Blinking at 1 Hz	Awake, receiving input
ON (continuous)	Executing command
OFF	Sleeping

When a bootloader is installed, the LED also conveys the bootloader state at boot time:

ON 2 sec	Entering bootloader
Blink 1x 250 ms	Received bootloader enable sequence byte
ON (continuous)	Bootloader waiting for firmware upload
Blink 3x 250 ms	Booting to application
Blink fast, dim	Loading firmware image

In the unusual event that a serious unrecoverable software or hardware fault occurs, the processor is forced to jump to special routines called error traps. These routines have been customized to blink the LED in a pattern indicating the type of fault that has occurred:

Blink 1x	Oscillator Failure: possible hardware issue
Blink 2x	Stack Error: possible stack overflow or corruption (possible bug, e.g. mismatched or invalid arguments passed to function)
Blink 3x	Address Error: Illegal memory access (possible bug, e.g. array overrun, invalid pointer)
Blink 4x	Math Error: illegal math operation (e.g. divide by zero, math function input range violation)

Once in an error trap, the pattern repeats until the system is restarted.

Startup Status Messages

When the system boots, it displays a series of messages, the presence or absence of which indicates something about errors that may have occurred during the boot process.

```

serial IO [OK]
sdcc mount [OK]
log_init [OK]
Loading config...cfg load [OK]
running boot script...
00:00:03
bootstrap [OK]

MBARI CO2/DIC
firmware[ cwg_nonboot v0.0.0 tag 2.0.0b0 ]
build [ Oct 19 2020 - 21:01:28 ]
status [ 2083 ]
cfg_sn [ C024 ]
sys_sn [ FFFF ]

```

If no messages appear on the console, an error has occurred early in the boot process, i.e. before the serial ports are up and running.

The firmware reads the RCON (restart control) register on start-up. The RCON value indicates the cause(s) of the last processor restart. The RCON value is reported as a 16-bit hex number in the “status” line of the system start message. The following table is lifted from the PIC24FJGA606 data sheet (section 7.1, resets)

Flag Bit	Setting Event	Clearing Event
TRAPR (RCON[15])	Trap Conflict Event	POR
IOPUWR (RCON[14])	Illegal Opcode or Uninitialized W Register Access	POR
CM (RCON[9])	Configuration Mismatch Reset	POR
EXTR (RCON[7])	MCLR Reset	POR
SWR (RCON[6])	RESET Instruction	POR
WDTO (RCON[4])	WDT Time-out	CLRWDT, PWRSV Instruction, POR
SLEEP (RCON[3])	PWRSV #0 Instruction	POR
IDLE (RCON[2])	PWRSV #1 Instruction	POR
BOR (RCON[1])	POR, BOR	—
POR (RCON[0])	POR	—

Debug Output

There are a couple mechanisms implementing debug messages in the firmware that may be dynamically enabled or disabled. There are per-module debug messages, output using macros (DPRINT/VPRINT), and per script debug output.

Module Debug Output

The DEBUG command sets (and gets) debug flags that enable debug output on a per module basis.

```

use      : debug [<op><mask>]
brief    : show/set per-module debug output flags
          MOD_ACT =0x0001 // cli_impl
          MOD_TMR =0x0002 // timers
          MOD_SCR =0x0004 // script
          MOD_LIC =0x0008 // LI-COR
          MOD_SER =0x0010 // iosecial
          MOD_CFG =0x0020 // config
          MOD_MAIN=0x1000 // main
          MOD_TOKS=0x2000 // command token

input    : op - operator [+:set -:clear =:assign]
input    : mask - hex bit field [0-FFFFFFFF]
return   : debug flags
examples :
# disable debug output
debug =0
SYS_MDFLAGS=x00000000
# enable scr module debug output
debug +4
SYS_MDFLAGS=x00000004
# enable LI-COR module debug output
debug +8
SYS_MDFLAGS=x0000000c
# disable LI-COR module debug output
debug -8
SYS_MDFLAGS=x00000004
# show debug flags
debug
SYS_MDFLAGS=x00000004

```

Verbose and Debug Print Output

The SET PRINT directive configures the VPRINT and DPRINT macros in the code. These can be especially useful when prototyping new code. The DPRINT macro is enabled if non-zero. The VPRINT macro supports “volume” levels; VPRINT messages will be output if their configured level is less than or equal to the VPRINT level.

```

use      : set print [D<n>] [V<n>]
brief    : set value of debug and verbose print variables
          used to configure DPRINT and V*PRINT macros
          DPRINT is enabled if !=0
          VPRINT is enabled if !=0, and supports multiple levels (>0)
return   : none
examples :
# enable debug printing and verbose level 2
set print dl v2

```

Existing statements implemented in this way can be found by searching the source code.

Script Debug

The SCR SDEBUG directive enables script debug output on a per-script basis. This emits messages about the script engine state.

```

use      : scr sdebug +|- <id>
brief    : enable/disable debug output for specified script
input    : id - script name or handle
return   : none
examples :
# enable script debugging for script ID 1
scr sdebug + 1

```

See the Scripting Guide section for notes on the use of PRINT and PRINTLN for producing debug output in the script context.

[Contents](#)

Scripting Guide

Guide to developing scripts

Scripting is among the controller's core features. Scripting enables users to capture command sequences in text files, which may be played back at system startup. This architecture enables the controller hardware to implement different applications simply by switching scripts: it can make DIC measurements one day and controlling a camera the next.

By the same token, it is easy to update and change applications on the fly.

The scripting capability extends the existing user APIs. In fact, the script engine uses the user interface code to interpret scripts. All of the commands are the same in script context as they are on the console.

The controller and scripting engine don't offer robust protection from script interactions, invalid constructs, exception handling, etc. In the absence of such safety nets, testing is critical to ensure that your scripts behave as expected.

In this section, learn more about the SCR command, which introduce directives for managing scripts, as well as some pointers for writing and using scripts.

Script Basics

Scripts are simply a set of controller commands that have been written into a text (ASCII) file. The commands may be played back by using the SCR RUN directive, and may be scheduled for periodic execution using the SCR SCH directive or CRON command. Scripts may even run and schedule other scripts.

In this section, a few elements that are unique to the scripting context are described.

File Format

Script files are expected to be ASCII text.

Text files used on the controller must terminate lines using CRLF ("", i.e. DOS style). However, line endings are automatically converted to CRLF when using the UPLOAD command, so it's OK to use newline ("", Unix style) in your source text.

If you copy files directly to the SD Card, be sure to make the necessary line ending conversions.

Code lines have a maximum length of 127 characters (including comments and whitespace) Empty lines and leading/trailing space characters (' ', ' ') are allowed. Lines that exceed the maximum length are ignored; an error message is issued if sdebug is enabled.

The practical limit for script length is imposed by the maximum file size (4 GB for FatFS). If scripts become very long (10s of MBytes), access times may become significant. However, typical scripts are less than a few kBytes.

Comments and Space

Comments are supported in script files. All text following "#" or "//" is interpreted as a comment. Comments may appear at the end of lines (but count against the line length).

```
# this is a comment
// and this is a comment

print "there is a comment after this code\r\n" // this is code with a comment


# comments that are too
loooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
g are ignored
print "code lines that are too
loooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
g are ignored\r\n"
print "how bad # can this // be?\r\n" # is this trouble?


    print "it's OK to include leading space, but why would you?\r\n"
```

END command

Scripts should be terminated with the END command; the END line should also be terminated with CRLF. A warning will be emitted if script parsing reaches the end of the file without finding the END statement.

Branching

Sometimes it is useful to conditionally execute different branches of code. A very basic branching mechanism is implemented using an IF command with some simple logical test expression syntax; a GOTO and labels are implemented as well. Combined with register variables, these may be used to implement more advanced constructs like loops and sub-routines.

GOTO, Labels

The GOTO command and labels are a simple construct that causes execution to jump to a new location.

Then basic syntax for GOTO is:

```
goto <label>
```

where \<label\> uses the form:

```
:<name>
```

Valid label names must be unique (one occurrence per file), and may use

- letters (A-Z, case insensitive)
- digits (0-9)
- other printable characters in the range 33-126 dec/0x21-0xFE hex

```
println "you WILL see this"
goto foo
println "you WILL NOT see this"
:foo
println "you WILL ALSO see this"
:bar
println "unused labels? no problem"
END

outputs:

you WILL see this
you WILL ALSO see this
unused labels? no problem
```

The real usefulness of GOTO and labels is in combining them with the IF construct.

IF

The IF command may be used to conditionally execute a command. The basic syntax is

```
IF [ <expression> ] <statement>
```

If <expression> evaluates to TRUE, <statement> is executed; otherwise execution continues following the IF statement.

Here, <statement> may be any valid controller command. GOTO is often used to implement IF/ELSE constructs as described below.

Expressions

Expressions are boolean statements that evaluate to TRUE (non-zero) or FALSE (zero). Expressions have the general form(s)

```
# numeric comparison
<num> <op> <num>
where
<op> is one of : == != > >= < <=
<num> is a register variable or configuration variable

register values are compared as floating point numbers (32-bit double)
Comparing very large or very small numbers may produce invalid results.

# logical comparison
[!] <bool> [<op> [!] <bool>]
where
<op> is one of : && || ==
<num> is a register variable or configuration variable (zero: FALSE, TRUE otherwise)
```

The TEST BITS directive may be used to get IO bit states into register variables for comparisons.

```
Example: Using test bits for script flow control

# clear all IO bits (except for transceivers)
clr bits p* v* a* 1
# set ADC bit
set bits a*
# store current IO state in CX
# store ADC bit state in CX1
test bits a* MCX0 OCX1
# compare bit state vs mask
if [ CX0 == CX1 ] goto pass
# test failed
println "set bits a [ERR / $CX0 / $CX1]"
goto cont
:pass
# test passed
println "set bits a* [OK]"
:cont
```

Similarly, GET AD may be used to store ADC channel values in a DX register variable.

IF/ELSE

The example above contains an example of using IF, GOTO and labels to construct an IF/THEN/ELSE logic block:

```
# Example: IF/THEN/ELSE

# Test condition (branch to PASS if true)
if [ CX0 == CX1 ] goto pass

# ELSE action
println "set bits a [ERR / $CX0 / $CX1]"
# Jump over THEN to CONT
goto cont

# THEN action
:pass
println "set bits a* [OK]"

# Keep going
:cont
```

Looping

The IF and GOTO commands can be used with variables to create loops.

```
# initialize a counter (out side of loop)
creg CX0=3

# loop start label
:LOOP

# test for an exit condition
if [ CX0 <= 0 ] goto QUIT

# do something...
if [ CX0 == 2 ] freg FX0=1
println "looping CX0=$CX0 FX0=$FX0"

# update loop counter
creg CX0--

# next iteration
goto LOOP

# loop resumes here
:QUIT
println "done"
END
```

The same pattern may be extended to nested loops as well:

```
# nested loop demo

# init output loop counter
creg CX0=4

:level_0
creg CX1=3
println "L0 action"

:level_1
creg CX2=2
println "L1 action"

:level_2
println "L2 action"

creg CX2--
if [ CX2 != 0 ] goto level_2
creg CX1--
if [ CX1 != 0 ] goto level_1
creg CX0--
if [ CX0 != 0 ] goto level_0
println "done"
end
```

Output

The examples above include script output using PRINT and PRINTLN commands. Here are a few notes on PRINT and PRINTLN.

PRINT, PRINTLN

PRINT and PRINTLN, as their names suggest, print formatted output messages to the console.

PRINT and PRINTLN are mostly interchangeable; the only difference is that PRINTLN automatically appends CRLF to output.

For many messages, PRINTLN is the go-to tool.

However, in a loop, or in a conditional construct where you want several values on the same line, but won't determine until run-time which lines will add values to the line, PRINT is the way to go.

The PRINT commands use the same syntax:

```
print <fmt> [<var>...]  
where  
<fmt> is a C-style print format string  
<var> may be  
- register variable names  
- configuration variable names
```

Format strings (<fmt>) may contain - text - C escape characters sequences (e.g. ',,') - C print specifiers (e.g. %ld, %04x, %3.4lf) must match <var> elements in number and use appropriate data types - variable and register references (e.g. \$CX0, \$LOG_RAW) which substitutes their value using their default format.

This is useful for converting between value formats (hex <-> decimal). Again, note that formats used must use appropriate types for the values they represent.

For example, since integers on this processor are 16-bit, and long integers are 32-bits, CX registers or UINT32 configuration variable correspond to %ld and %lx, rather than %d and %x.

Here are a few examples:

```
println " var-sub CX0 dfl[$CX0]"  
println "reformat CX0 hex[%4lx]" CX0  
println "reformat CX0 dec[%ld]" CX0  
println "DREG/variable LI_SPANC[%3.2lf] DX0[%.4e]" LI_SPANC DX0
```

Advanced Techniques

Functions

Functions are script segments than can be invoked from within a script to perform frequently-used command sequences. Using functions has a number of benefits:

- reduces errors and maintenance by reusing code (instead of cutting and pasting)
- reduces overall code size
- improves reliability by reducing potential points of failure
- simplifies debugging

In true programming languages, functions generally accept input arguments and may return output values. A compiler or interpreter typically validates their use and flags errors.

In this application, the GOTO and label features, together with (register and configuration) variables may be used to implement a crude approximation of a function. Given the limited support for variables and expressions, this implementation is more like macros than functions: it captures a reusable sequence of operations.

The function specifies

- an entry point label
- registers to pass in and return values (if any)
- a string variable (e.g, S0) to set the return label

Of course, the calling script and function must agree upon the registers and variables used.

In the following example, a function cycles selected IO with a delay. Though it could be done more efficiently with existing commands, it demonstrates the basic function pattern. The following inputs and outputs are used :

in/out	Var	Description
in	CX0	cycles
in	CX1	IO mask
in	CX2	delay msec
in	S0	return label
out	CX3	status 0:done

```
set var S0 fret
creg cx0=5 cx1=x800000 cx2=3000

println "cycles[%lu] bits[%08lx] delay[%ld] ret[$S0]" CX0 CX1 CX2
goto cycle
:fret

if [ CX3 == 0 ] goto success
println "ERR stat[%ld]" CX3
goto quit
:success
println "OK stat[%ld]" CX3
# goto exit
goto quit

# cycle function
:cycle
# set bits (CX1), delay (CX2 msec), clr bits
mdio IOH:CX1 DCX2 IOL:CX1

# decrement counter
creg cx0--

# update status
creg cx3=cx0

# check whether done
if [ cx0 == 0 ] goto end_cycle
# start next cycle
goto cycle

:end_cycle
println "returning to $S0"
goto $S0

println "should not be here"

# script exit
:quit
println "done"
END
```

Signaling

Scripts can use variables to signal each other; for example, a primary script may trigger one of several secondary scripts that are pending on some flag, based on some condition (a configuration value, e.g.).

See the [Script Examples](#).

Installation

UPLOAD

As demonstrated in the quick start section, scripts may either be installed directly to the SD card: - power off the controller - remove the SD card from its socket - put it into an appropriate card reader and connect it to a computer - mount the SD Card - copy files to the SD card and unmount - if needed, convert the files on the SD card to use Windows line endings (CRLF) - replace the SD card in the controller socket

Alternatively, use the UPLOAD command:

```
upload <name>
```

The name given to the upload command is the name of the file written on the controller. It does not have to match the file name on the host computer. File names on the controller use the DOS 8-dot-3 format: <name>.<extension>, where the name length is limited to 8 characters, and extension to 3 characters.

Also, script may use any valid filename; by convention, the .SCR extension is recommended.

When controller responds with

```
Waiting for file path[FOO.TXT] timeout[15000]  
Interrupt/end using CTRL-D
```

use the terminal program to send the text file; this is typically done using a menu option to send a file, which varies per application.

In minicom, for example, CTRL-A Y opens a navigation menu, where you navigate to the file and select it.

From the hyperterm menu bar, use Transfer >> Send Text File... then find and select the script file.

With some terminal programs, you may simply copy and paste the script text directly into the terminal window.

The upload command times out after 15s if the transfer ends or hasn't started. If that happens, use the N option to set the upload timeout. For example, to set a 30s timeout, use:

```
upload N30 hello.scr
```

Each line of the file is displayed on the console as the file is uploaded. Note that if you type other characters into the console during the upload session, they will be put into the file. When the script upload is finished, wait for the timeout to expire or end the upload session typing CTRL-D (press and hold CONTROL then D keys) on the console.

Running scripts

Starting

LOAD/UNLOAD

In order to be run, scripts must first be loaded into a "slot", which is a data structure that keeps track of the script's state. There are currently four script slots, numbered 0:3. These slot numbers are also referred to as handles.

In the context of script commands, scripts may be identified by their name or handle. Since handles are shorter, they are convenient to use; both are used in this document.

Load the script using the SCR LOAD directive:

```
scr load hello.scr
```

An error message will be issued if the script is already loaded. To force reload/overwrite, use the 'o+' option:

```
scr load o+ hello.scr
```

To inspect the script slots or determine a script handle, use the SCR SHOW directive:

```
scr show  
or  
scr show <id>
```

To remove a script from its slot, use the SCR UNLOAD directive, which may be used with a name or handle, e.g.:

```
scr unload hello.scr  
or  
scr unload 0
```

RUN

To run hello.scr (handle 0, for example), use the SCR RUN directive; for example:

```
scr run hello.scr  
or  
scr run 0
```

The controller firmware is a single thread of execution. When scripts are running, the script engine is called once during the main execution loop. The script engine iterates over the script slots (starting with slot zero), invoking a state machine update for each script. So, scripts advance one step per execution cycle, which is effectively one line of execution. Essentially, it is a form of cooperative multi-tasking with round-robin scheduling. Each line is atomic, i.e. it will run to completion before other script lines run.

Console and host port serial IO is serviced before script updates in the main execution loop. If a script performs operations that take a very long time, response to user/host IO may be slow. Commands with delays (MDIO, MMEAS) do not yield while they are delaying.

The SCR DELAY directive may be used to suspend a script's execution for a specified interval.

Host-side clients may need to suspend a/the running script(s) and/or use appropriate logic and timeouts.

See the Scheduling section below for information about running scripts with SCR SYNC.

JOG

It may be useful to execute scripts one instruction at a time, e.g. for debugging. To single step a script, use the JOG directive to put it in manual mode. It may be useful to enable script debugging as well with the SCR SDEBUG directive


```
# load hello.scr (in this example, assume handle 0)
scr load hello.scr

# enable script Debugging
scr sdebug + 0

# put handle 0 in manual mode (advances first step)
scr jog 0

# advance script handle 0 one step
scr jog 0

# repeat until done...
```

SCR JOG may also be used on a running script.

To resume running a script in manual mode, use SCR RUN. To return to the beginning (reset) use SCR KILL.

Suspending and stopping

DELAY

The SCR DELAY directive may be used to temporarily suspend execution of a running script.

```
scr delay <time_ms> <id>
```

This is useful for getting exclusive use of the console for a time, while ensuring that the script will resume if the connection is dropped.

The script's run timer (period SCH timer) continues to run while delayed and resumes when delay expires.

To resume running before the delay has expired, RUN or SYNC may be used, or by using DELAY with a period of 0.

A delay period <0 will pause the script indefinitely.

RESET

The SCR RESET directive resets a script's delay timer and execution pointer. It does NOT reset the run timer, so a script scheduled with SCH will continue to execute.

To stop a script, use SCR KILL or SCR SCH using a period of 0 and 'N' option (no sync)

KILL

SCR KILL stops a running one or more scripts.

```
# stop hello.scr and script handle 1
scr kill hello.scr 1
```

Scheduling

SCH

The SCR SCH directive is used to start or end periodic execution of a script.

```
scr sch P<period> D<sync_delay> <id>

where
  period    : execution interval (msec)
  sync_now  : start running now (Y: start schedule, but don't start)
             id : script name or handle

Example:

# run hello.scr every 20 min, starting now
scr sch P1200000 D hello.scr

# run hello.scr every 20 min, start in 5 min
scr sch P1200000 D300000 hello.scr

# set hello.scr schedule, but don't start
# set a.scr schedule and start in 15 sec
scr sch P60000 hello.scr P120000 D15000 a.scr
```

SCR SCH is a relative scheduling mechanism; it is possible to delay the start of the period using SCH SYNC (below); in this way, it is possible to align the script period with, for example, the start of the hour. The alignment has limited precision, however.

The CRON command implements a scheduler that can execute scripts more precisely at specific dates/days/times.

SYNC

The SCR SYNC directive may also be used to start scripts after a delay. It is typically used to start a periodically scheduled script (scheduled with SCR SCH) to begin at a certain time, i.e. shift it's schedule to a new start time.

```
# start hello.scr 10 s from now
# (if already schedule, shift schedule start to 10s from now)
scr sync D10000 hello.scr

# run a.scr now (even if unscheduled)
scr sync a.scr
```

CRON

version >= 3.1.0b0

The CRON command exposes a scheduling feature similar to that provided by the cron utility that is common on many Unix variants.

```
use      : cron [options]
brief    : manage cron schedules
options  :
+|-      enable/disable cron
X        show time to next scheduled event (msec)
          - disabled tabs excluded
          - indicator (+|-) shows cron enable status

L:USR|SYS|*      list tab file contents
S:USR|SYS|*      show tab status
C:USR|SYS,<flags> configure tab flags
               +|-e: enable/disable tab
               +|-x: enable/disable exclusive mode
F:USR|SYS,<file> set tab file
T:USR|SYS,<i>,<sch>[,<scr>] configure task schedule and script (optional)

return   : varies, status info etc.

notes    :

[1] changing schedules or tab files unloads scripts immediately, with no changes to IO.
[2] enabling exclusive mode takes effect in the next execution cycle
[3] disabling cron or crontab does not change files or schedules. In concurrent mode,
    currently running scripts will run to completion.

[4] schedule syntax: schedules are space-delimited strings with 6 fields:

    mon mday h m s wday

    mon : month      values: 1-12 JAN(1) FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC
    mday: month day  values: 1-31
    h:   hour        values: 0-23
    m:   minute       values: 0-59
    s:   second       values: 0-59
    wday: day of week values: 0-6 SUN MON TUE WED THU FRI SAT
```

```

m: minute      values: 0-59
s: second      values: 0-59
wday: weekday  values: 0-6  SU(0) MO TU WE TH FR SA

```

Fields are comma-delimited specifiers which may include

```

individual fields val
ranges             val-val
wildcard           *
modulus            */n
range w/ modulus   val-val/n

```

example:

```

# mon: every third month,
# mday: every other day
# hour: every other hour between 0000-0600 and every hour 1800-2000
# every 20 min and half past the hour
# wday: MO,TH,FR,SA
JAN-DEC/3 */2 0-6/2,18-20 */20,30 MO,TH-SA

```

[5] crontab syntax: crontabs (tables) are text files containing task specifiers consisting of a schedule and a task (script) name:

```
"schedule" = task
```

```

schedule : schedule specifier (see description, above)
task      : script name

```

- comments are allowed
- everything following "#" or "/" is a comments
- comments may appear at the end of a line
- leading/trailing whitespace is allowed (and surrounding '=')
- schedule specifier must be in single or double quotes

crontabs may contain a configuration line, which may be used to set flag values:

```
flags=<flags>
```

where flag values are as described above. The exclusive mode flag indicates that scripts should be run from start to finish without yielding. If unset, scripts are executed as other scripts would be, yielding between states. Currently, the exclusive flag applies to all tasks defined in the crontab.

Scripts are unloaded on completion or error.

There are two crontabs in the system: CRONTAB.SYS and CRONTAB.USR. Each crontab may hold up to 4 schedule specifiers. However, there are a total of 4 script slots for the system.

example:

```

# user cron tab
flags = +x+e
"* * */2 */10 0 MO-FR" = utime.scr

```

examples :

```

# show USR crontab contents
cron l:usr
# show cron status and time to next event
cron s:* x
# set usr crontab to exclusive mode and enable
cron c:usr,+x,+e
# set usr crontab task ID 0 schedule (every other day, every 20 min between 0600-18:00)
cron t:usr,0,"* */2 6-18 */20 0 SU-SA/2"
# set sys crontab file to testtab.sys
cron f:sys,testtab.sys

```

Debugging

SHOW

To inspect the script slots and find a script handle, use the SCR SHOW directive (sometimes referred to as sub-commands):

```

scr show
or
scr show <id>

```

which responds with something similar to

```

SCR[0]
  self          5ac2
  ctx           5ac4
  name          UTIME.SCR
  fp           5ad4
  state         x0003
  modes         x0000
  events        x0082
  error         0
  tmr[0]        {Y, 4652, 20801}
  tmr[1]        {Y, 56806, 20801}
  period        60000
  line          0

SCR[1]
  self          5b1a
  ctx           0

SCR[2]
  self          5b72
  ctx           0

SCR[3]
  self          5bca
  ctx           0

```

Here, SCR[0] (handle 0) contains HELLO.SCR, and that the other 3 slots are empty. The other state/status information will be covered later.

Debugging features

As noted in previous sections, these features are useful for debugging scripts:

<i>SCR JOG</i>	Manual script execution mode
<i>SCR SDEBUG</i>	Enable per-script debug output
<i>PRINT, PRINTLN</i>	Formatted print, supports variable substitution
<i>DEBUG</i>	Enable per-module debug output
<i>SET PRINT</i>	Enable extra DPRINT/VPRINT macro output
<i>CREG, DREG, FREG</i>	set/inspect register variables
<i>GET/SET BITS</i>	set/inspect register variables
<i>PMEM</i>	inspect dynamic memory contents

[Contents](#)

Integration Guide

Guide to developing clients

section pending...

Client-topic

Client section header

[Contents](#)

Appendices

additional and reference information

Contents:

- Data Type IDs
 - Output Format Examples
 - Script Examples
 - Script Context
 - Common Issues
-

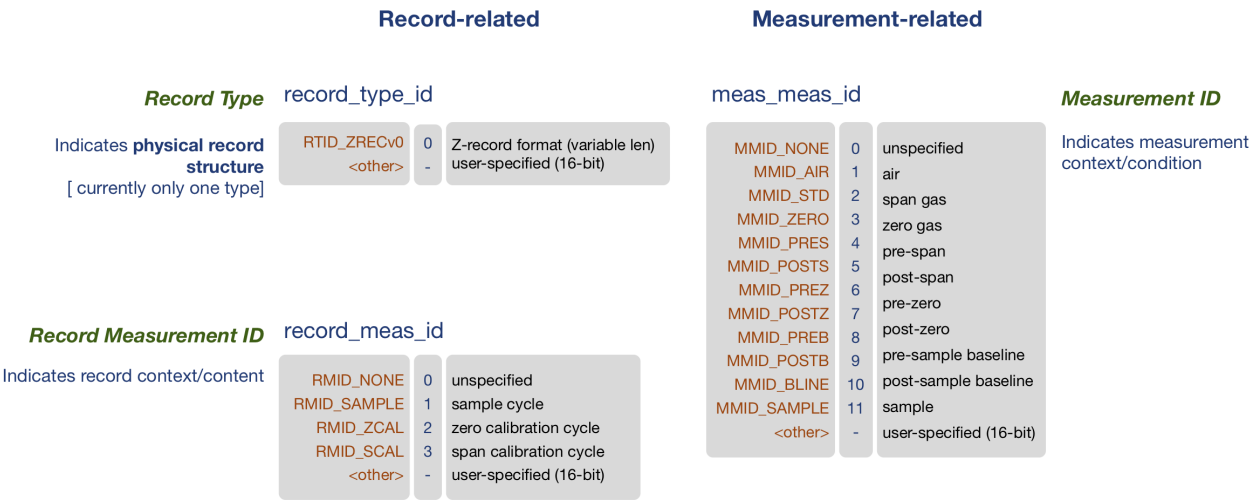
Data Type IDs

The graphic below summarizes various ID metadata used in measurements and records

- Record Type IDs (RTID)
- Record Measurement IDs (RMID)
- Measurement IDs (MMID)

Data Type IDs

Constants used to define
record and measurement
types



Data Type IDs

Output Format Examples

The graphic below provides examples of data output formats

Output formats

The contents of the cycle buffer and queues may be inspected using several output formats.

Output formats are provided as arguments to mmeas and the queue commands. Commands syntax for displaying the latest measurement on the console are in various formats are shown here.

qpeek and qpop use similar syntax ("out:" instead of "out<src>:"). Output is sent to the console using Pretty format by default.

```
qpeek I0 out:c,xx
```

Human-readable formats (partial output shown)

Pretty - Formatted table

```
mmeas outm:c,p

self      1238
sync      0CBA1C02
timestamp 1582838414000
type_id   0052
meas_id   0061
seq_n     0
len       280
meas_count 1
measurement 0
self      1254
timestamp[0] 1582842029500
timestamp[1] 1582842034000
meas_id    0001
stat_n     10
min/max/K/S
co2 +1.064e+01 +1.090e+01 +1.074e+01 +7.230e-02
co2abs +1.155e-02 +1.182e-02 +1.166e-02 +7.737e-05
h2o +8.784e+00 +8.786e+00 +8.785e+00 +5.663e-04
h2oabs +5.345e+00 +5.347e+00 +5.346e+00 +8.040e-04
_
```

PrettyHex - Formatted hexadecimal

```
mmeas outm:c,xx

00000000: EF BE AD DE 02 00 00 00 F8 54 36 88 70 01 00 00
00000010: 00 00 00 00 01 00 7C 00 00 00 2A 08 E4 31 6D 88
00000020: 70 01 00 00 78 43 4D 88 70 01 00 00 00 00 0A 00
00000030: A4 C0 E9 40 A3 E1 00 3C F6 B2 18 41 4F 91 D1 40
00000040: 6D 14 9E 3D B8 BC CF 41 E7 5A CC 42 9D 7C 31 41
00000050: 00 00 00 00 A2 FA 84 4A 10 B8 93 4A EB 6C 01 4A
_
```

Machine-Machine formats (partial output shown)

Base85 - Efficient ASCII encoding

```
mmeas outm:c,b

[4L{X0SSi2{/6ecz#/)s000000rr@q00$h<s$Icz#/...
```

Hex - Unformatted hexadecimal

```
mmeas outm:c,x

EFBDDE02000000B83B358870010000000000000000...
```

CSV - Comma-delimited ASCII

```
mmeas outm:c,c

1582830433000,0042,004D,0,1512,6,...
```

Raw - Just the bytes

```
mmeas outm:c,r
```

Output Format examples

Script Examples

File	Contents
expr.scr	Expressions
fmt.scr	Script formatting
func.scr	Function example
hello.scr	Hello World
jog.scr	Manual Mode
nest.scr	Nested loop
a.scr	Signaling (part a)
b.scr	Signaling (part b)

Script Context

The following table summarizes the script context fields displayed when using the SCR SHOW directive.

field	description	example
self	script instance address	5ac2
ctx	context pointer address	5ac4
name	file name	UTIME.SCR
fp	file pointer address	5ad4
state	state enumeration	x0003
modes	mode flags	x0000
events	event flags	x0082
error	error flags	0
tmr[0]	run timer	{Y, 4652, 20801}
tmr[1]	delay timer	{Y, 56806, 20801}
period	execution period (msec)	60000
line	line number	0

ctx::timer

A script context has two timers: the run timer (for periodic scheduling), and the delay timer (for execution delays). Three fields e.g. {Y, 4652, 20801}, are displayed by SCR SHOW:

Field	Description
status	Y: active N: inactive
remaining	ticks, 0.01s
counter	ticks, 0.01s

ctx::state

The state field is an enumeration value

Value	Name	description
-1	SS_INVALID	Non-valid state (used as error value)
0	SS_IDLE	Initialized, not ready to run
1	SS_READY	Ready to run
2	SS_RUN	Running
3	SS_DELAY	Delayed (pending delay timer)

Value	Name	description
4	SS_PAUSE	Paused (pending flag)
5	SS_ERR	Error state

ctx::modes

Value	Name	description
0x0001	SM_AUTOLOOP	automatically restart
0x0002	SM_MAN	manual mode indicator
0x0004	SM_NOSLEEP	do not sleep
0x0800	SM_DEBUG	debug output enabled
0x1000	SM_ERROR	an error occurred
0x8000	SM_INVALID	in invalid state

ctx::events

Value	Name	description
0x0001	SE_END	stop execution
0x0002	SE_PLAY	run/continue
0x0004	SE_PAUSE	pause execution
0x0008	SE_RESUME	resume execution
0x0010	SE_TMREXP	timer expired
0x0020	SE_FWD	jog forward
0x0040	SE_REV	jog backward
0x0080	SE_DELAY	delay execution
0x0100	SE_JMP	jump to location
0x1000	SE_ERROR	error occurred
0x8000	SE_INVALID	invalid state

ctx::errors

Value	Name	description
0	EOK	OK, no error
1	EEXEC	execution error
2	ECMD	command failed
3	ERead	read error
4	EINVAL	invalid argument/state

Common Issues

Not booting - fuses - power - cables, connectors

No connection

- loopback test
- cables and connectors
- RX/TX swap
- comms settings
- local echo

Extra line endings

- terminal software line ending settings

Random characters

- baud rate, parity
- cables, connectors
- floating ground

[Contents](#)

Resources

Useful links

Find drawings and links to additional information

Drawings

[Schematic](#)

[Pneumatic Diagram](#)

MBARI

[MBARI](#)

<http://www.mbari.org>

Microchip

[MPLAB IDE](#)

<https://www.microchip.com/mplab/mplab-x-ide>

[PIC24FJ1024GA606 Datasheet](#)

<http://ww1.microchip.com/downloads/en/DeviceDoc/PIC24FJ1024GA610-GB610-Family-Data-Sheet-DS30010074G.pdf>

[XC16 Downloads and Docs](#)

<https://www.microchip.com/en-us/development-tools-tools-and-software/mplab-xc-compilers>

Software

[ASCII Table](#)

<http://www.asciitable.com>

[printf Manpage](#)

<https://linux.die.net/man/3/printf>

[FatFS](#)

http://elm-chan.org/fsw/ff/00index_e.html

[Contents](#)

