FOCE Intermodule Communications System

by Roland Quiros

Advisor: Vladimir Prodanov

Senior Project

ELECTRICAL ENGINEERING DEPARTMENT

California Polytechnic State University

San Luis Obispo, CA

June 2012

Table of Contents

Index of Tables

Index of Figures

Acknowledgments

I want to take the time to thank my family, particularly my Aunt Lynn. Without your support, I could have never attended an institution as accomplished as Cal Poly. And, naturally, my parents, for pushing me to see things through to the end, despite the hurdles.

Of course, I'd like to thank Prof. Vladimir Prodanov and Chad Kecy of the Monterey Bay Research Institute for giving me the opportunity to work on this fascinating project. Everything described in these pages is a direct result of their continued support, and *infinite* patience.

1 Introduction

Recent years in aquatic research have seen a greater emphasis on autonomous instrument packages as a method of gathering data. By using the seafloor as a "natural laboratory," researchers have access to an immense range of data only a real-time, in situ environment could provide. In 2004, the Monterey Bay Research Institute (MBARI) conceived the Free Ocean Carbon Dioxide Enrichment (FOCE) system, taking advantage of these possibilities.

Since the late 1950s, scientists have observed an increasing carbon dioxide (CO_2) content penetrating the oceans, as a direct result of the atmospheric pollution caused by modern industries. With atmospheric CO_2 levels increasing by one million tons per hour, researchers have observed a sharp decrease in potential hydrogen (pH) and, consequently, the increasing acidification of the oceans. **[1]**

Curious in how this change would affect plant and animal life, and unsatisfied by the limited scope of isolated laboratory experiments, MBARI created the FOCE system. Consisting of a "long, rectangular flume with a series of baffles," the structure allows CO_2-enriched seawater to mix with the surrounding seawater while a sensor array observes the effects. **[2]**

At the heart of the FOCE is the Chassis system, a modular array of interactive, programmable boards which control the power and sensing capabilities of the FOCE apparatus. This project is focused on further developing this system, particularly the methods of communication between the different modules.

This particular project was the product of a collaboration between Cal Poly and MBARI, in an ongoing partnership developed by Professor Vladimir Prodanov and MBARI engineer and Cal Poly Alum Chad Kecy.

2 Background

While the FOCE project, as a whole, has been in development for years, the electronic systems controlling the experiment remain only at test stages. In this project, I further develop the software framework behind the system's various PIC24 microcontroller-based modules, particularly expanding and generalizing the means of communications between these modules. This will allow seamless integration between the components attached to the system, with enough leeway to allow additional functionality with minimal changes to the source code.

3 Requirements

While several of the boards were designed, constructed, and fabricated, the software controlling them have lacked a proper utility for communications between boards. The objective of this project is to create proper communications between the different boards.

Each board's PIC microcontroller is connected to the slotted backplane through their respective SPI connections, all slots following the second reserved for expansion daughter boards. These daughter cards are SPI slaves to a central control board, dubbed the CPU board, which receives commands from human users through a serial UART interface connected to a PC.

With this architecture in mind, Mr. Kecy presented the following steps to establishing board communication:

1. The CPU board must know what type of daughter board resides in each slot;
2. The CPU board must know the types of commands/responses for each board type. These will be stored in the CPU in the event any type of board is plugged in;
3. CPU board must be able to combine 1 and 2;
4. CPU board will cycle through the backplane slots, requesting and storing board type IDs. If after a set amount of time without a valid response, the board will timeout and the slot is declared as “open.”

As the functionality of future daughter board have yet to be finalized, the method of communication must also be as board-agnostic as possible. That is, the information provided in step 2 must be provided by the daughter board, itself, with as few assumptions on the CPU board's part as possible.

Furthermore, user commands come in the following flavors:

- **Debug Commands** These are board-specific commands, but are all written around a common struct. They include “GET,” “SET,” and “CLR” (Clear). These allow access to raw data and internal variables not available to the user. They are accessed through a dedicated RS-232 debug port.
- **User Commands** These are the highest level commands which access data that are being stored in the CPU. These include “READ” and “WRITE.” They are accessed through the Ethernet connection to the CPU.

  All data is stored locally in the CPU's PIC. In the event of multiple instances of the same type of board, i.e. the Low-Voltage Load Switcher, the naming of variables will continue onto the next board. e.g. “Volt_09” will refer to the monitored voltage of channel #1 on LVLS #2.
- **CPU BUS Commands** These are internal system commands only. These automatically retrieve/set variables to and from the CPU board to daughter boards. They are used to populate the CPU PIC's storage which is available to the outside world.

Additionally, as this project has been inherited by Cal Poly students in the past **[3]**, and will

continue to be inherited in the future, so particular care should be placed in writing clear, thoroughly documented code to make the transitions as painless as possible. Consequently, the code submitted should represent a model for later boards to develop on. Thus, to demonstrate coding best practices, the CPU board will fully communicate with one other implemented daughter board provided with the system: the Environment board.

4 Design

Since the scope of this project is mostly within the software realm, **all hardware was provided by Mr. Kecy and MBARI**. Detailed schematics can be found in Appendix B. This section will describe parts of the hardware relevant to the code's operation.

Hardware

The FOCE Chassis system consists of a slotted **backplane** that connects various boards with different functionalities. The first two slots are reserved for the **Power Supply Board**, and the **CPU Board**. The remaining 8, numbered from zero, are expansion slots for the daughter boards. Two of such were provided with the apparatus: the **Environment Board**, and the **Low-Voltage Load Switcher**.

A PIC24 microcontroller sits at the core of each board, with different models in the PIC24 family chosen according to specific needs. Additionally, each board has a serial UART RS-232 port for debugging.

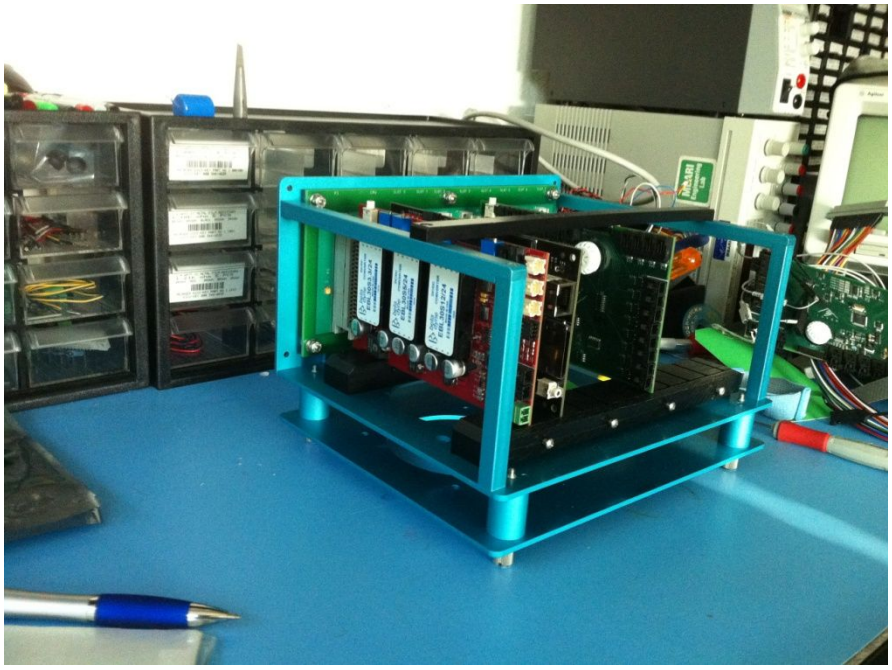


Figure 1: The entire FOCE Chassis Assembly

Backplane

The backplane holds the boards in plane while providing numerous ports that provide power and communication between modules.

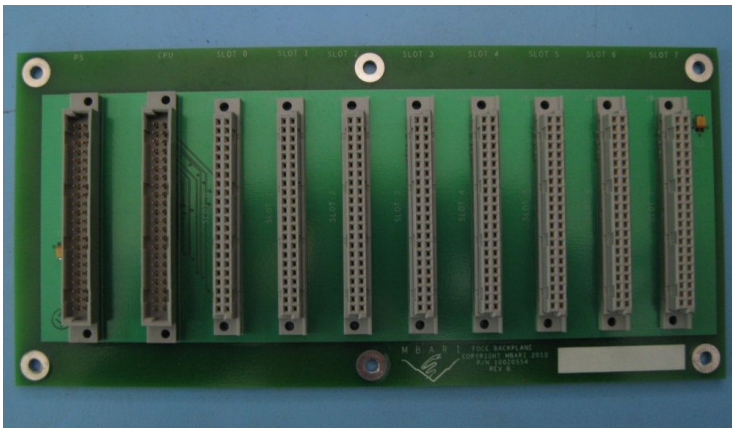


Figure 2: FOCE Chassis backplane

Figure 3 shows an example of one of the slots depicted in Figure 2, for the daughter boards. The slots for the Power Supply Board and CPU Board are slightly different, but only in terms of which pins are inputs or outputs.

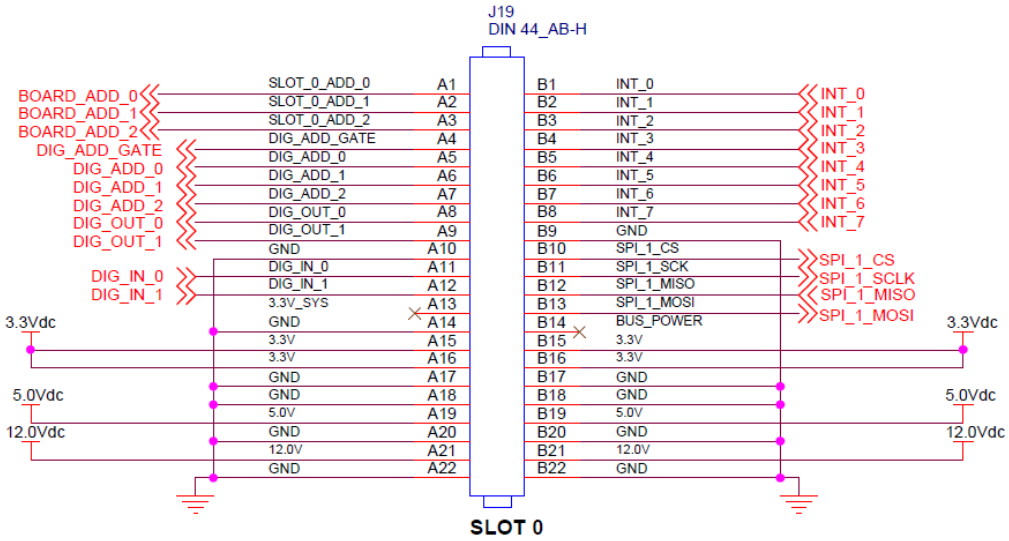


Figure 3: A backplane connector slot pinout diagram for a daughter board

The DIG_OUT_ADDR<2:1> pins and the DIG_OUT_ADDR_GATE pin are the most relevant parts of the backplane for this project. The DIG_OUT_ADDR pins control which daughter board is tied to the CPU Board's SPI lines, behaving similar to a three-input decoder, with DIG_OUT_ADDR_GATE acting as a chip enable line.

Power Supply Board

As its name suggests, the Power Supply Board delivers power to the rest of the device, the current iteration taking a 24V DC input, and stepping it down to 12V, 5V, and 3.3V for different applications. Its schematics can be found in Appendix B.2.

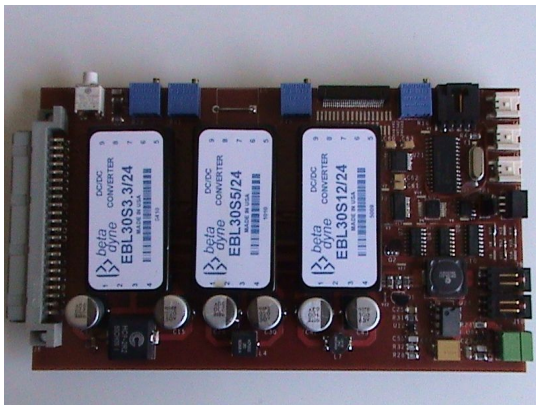


Figure 4: Power Supply Board

CPU Board

The CPU Board is where a majority of my work takes place. Its schematics are located in Appendix B.1.

This board acts as the interface between the human user and the daughter boards in the system. As said in the Requirements, this board takes user commands over the RS-232 port and propagates them to the daughter boards via SPI, which in turn provide the appropriate output to the CPU Board and the user.

The CPU Board also possesses a *Lantronix XPort* Ethernet port. As stated in the specifications, this is where the high-level user commands are transmitted. However, due to time constraints, this port remains unused for the moment. As of writing, all user commands are provided through the debug RS-232 port.

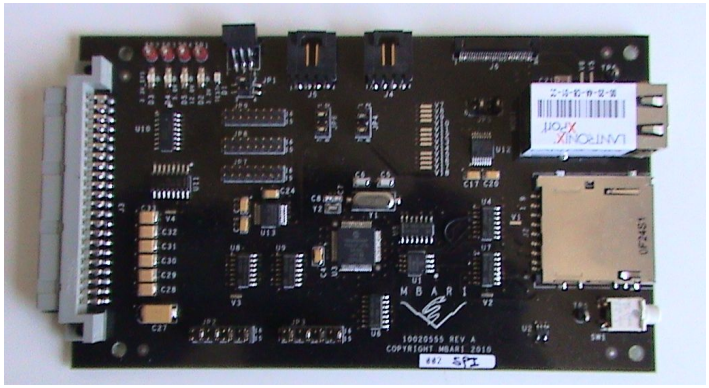


Figure 5: CPU Board

Environment Board

The Environment Board contains a large array of sensor ports, as well as a built-in humidity sensor, thermometer, pressure sensor, and compass. Each of these sensors is connected to an ADS8344 Analog-Digital converter, which in turn is connected to the PIC through SPI.

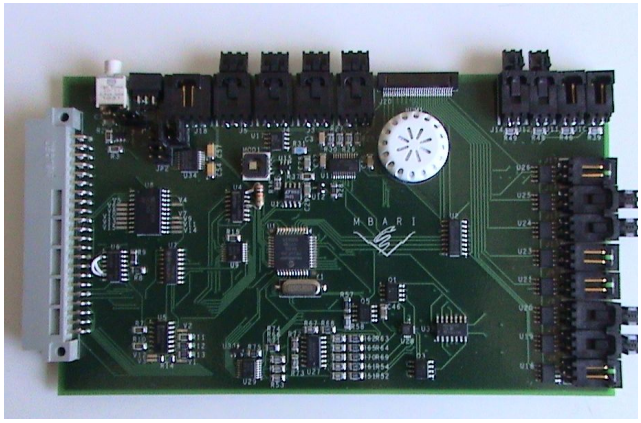


Figure 6: Environment Board

Through this project, the Environment board is fully communicable with the CPU Board, in order to demonstrate the added features.

Low-Voltage Load Switcher

This board is responsible for monitoring the voltage and currents of individual instruments. This board in particular, deals with load switching up to 30V and 2A.

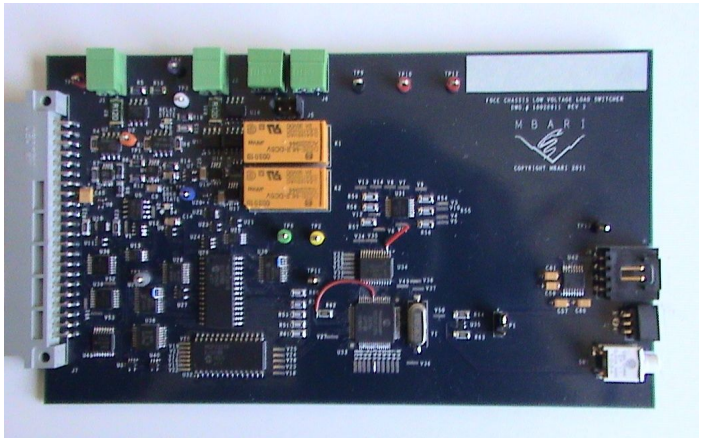


Figure 7: Low-Voltage Load Switcher

The Load Switcher does not have code associated with it as of writing, and was outside of this project's scope.

Software

Memory Models

According to the specifications, the CPU Board must store various quantities in a local table for the user to access. The user, then, must provide some identifier in order to retrieve the correct quantity—namely, a string of text. However, these identifiers may change depending on which daughter boards are connected to the system. Therefore, I needed to accomplish two things:

(1) Find a way to retrieve saved quantities from string data

(2) Allow the CPU Board to discover what kinds of identifiers a daughter board has available.

To address (1), I used a **hash table**, a data structure that performs precisely what (1) describes: stores information accessible through some arbitrary identifier data. When the user provides an identifier, or key, referring to a quantity (for example, "HUM01" for the Environment Board's built-in humidity sensor) a hash table will convert that key into a number. That number refers to an index in a table, which contains the data the user originally requested. The method used to obtain the index is called a **hash**.

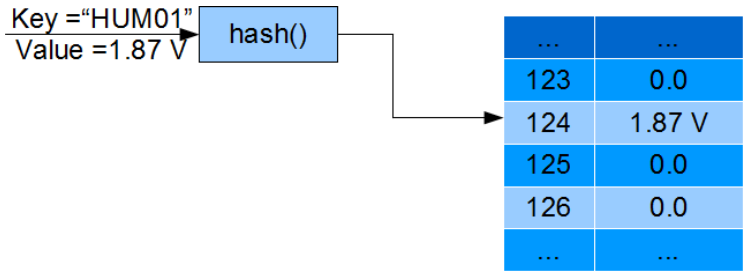


Figure 8: General operation of a hash table

Most decently fast hash methods require the storing table to be considerably larger than the intended data set to avoid **collisions**. Collisions occur when two different identifiers result in the same index number after undergoing the hash. One can prevent collisions by keeping the hash table under a certain **load factor**, expressed by n / s, where n is the number of entries and s is the size of the table. With most good hash functions, little to no collisions occur with load factors under about 0.7. However, with larger data sets the memory requirements will greatly increase, much of it remaining unused to maintain the load factor.

While most modern computer systems have ample memory to support large hash tables, embedded applications such as the FOCE Chassis have a much smaller pool of resources. Therefore, the hash method chosen must be both processor and memory-efficient. Fortunately, the **Pearson Hash** meets both those requirements.

For (2), I created a structure called **CardInfo,** designed to hold sufficient information about a daughter board to accomplish its requirements.

The Pearson Hash

Designed in 1990 and intended for fast execution on 8-bit processors, the Perason Hash capable

of generating unique numbers for an large variety of data sets, so long as the resulting table has a limited number of entries. It has since been used extensively in embedded applications, compilers, and networking for its efficiency and well-defined limits, evident in Listing 1, below.

Listing 1: A Pearson Hash, in BASIC **[4]**

```
h[0] := 0
for i in 1..n loop
      h[i] := -T[ h[i - 1] xor C[i] ];
end loop
return h[n]
```

Put simply, the hash loops through every byte in a string, `C`, submits it to an exclusive-or with the current hash number, `h`, then uses the result to return a number from a look-up table `T`. `T` is typically an array of 256 pseudorandom bytes, generated beforehand. So long as this table remains the same, the results of the hash will remain constant. Table 1 shows a possible permutation of `T`.

39	159	180	252	71	6	13	164	232	35	226	155	98	120	154	69
157	24	137	29	147	78	121	85	112	8	248	130	55	117	190	160
176	131	228	64	211	106	38	27	140	30	88	210	227	104	84	77
75	107	169	138	195	184	70	90	61	166	7	244	165	108	219	51
9	139	209	40	31	202	58	179	116	33	207	146	76	60	242	124
254	197	80	167	153	145	129	233	132	48	246	86	156	177	36	187
45	1	96	18	19	62	185	234	99	16	218	95	128	224	123	253
42	109	4	247	72	5	151	136	0	152	148	127	204	133	17	14
182	217	54	199	119	174	82	57	215	41	114	208	206	110	239	23
189	15	3	22	188	79	113	172	28	2	222	21	251	225	237	105
102	32	56	181	126	83	230	53	158	52	59	213	118	100	67	142
220	170	144	115	205	26	125	168	249	66	175	97	255	92	229	91
214	236	178	243	46	44	201	250	135	186	150	221	163	216	162	43
11	101	34	37	194	25	50	12	87	198	173	240	193	171	143	231
111	141	191	103	74	245	223	20	161	235	122	63	89	149	73	238
134	68	93	183	241	81	196	49	192	65	212	94	203	10	200	47

Table 1: A permutation of a Pearson lookup table

With a properly adjusted table, the Pearson Hash can attain **perfect hashing**, that is, no collisions, for a small, privileged set of entries. Without adjustment, collisions can be avoided, so long as the load factor is kept sufficiently small.

With a lookup table the size of Table 1, the Pearson hash table can only have 256 potential locations in memory. However, by expanding the table, this are can easily be increased.**[4]**

The CardInfo Structure

Each daughter board needed a way to store information about itself,. The CardInfo structure tightly packs this information into a single, organized unit for the CPU Board to readily retrieve over SPI.

Listing 2: The CardInfo structure, with private fields omitted

```
struct CardInfo
{
    unsigned char typeId;
    char          name[5];
    unsigned char varCount;
    char          varNames[256];
} __attribute__ ((__packed__));
```

Where `typeId` is the number associated with a daughter board. As of writing, the different board types with their associated numbers are:

Type ID	Board Type
01	**CPU**
02	**Power Supply**
03	Low Voltage Load Switcher
04	High Voltage Load Switcher
05	**Environmental Board**
06	Ground Fault Detection
07	Unassigned
08	Unassigned
09	Unassigned

Table 2: Daughter boards and their type numbers

`name` is a 5-or-fewer-character string holding an abbreviation of the daughter board's type name. For example, the Environment Board is abbreviated to “ENVR.”

`varCount` holds the number of variable identifiers stored in `varNames`. `varNames` holds the real meat of this structure: the variable names, their format, and their **I/O types**. The names are short text strings the user may use to request a quantity, and the format determines what number format the quantity should take (floating point, integer, etc.). The I/O type indicates whether a quantity is an input (specified by the user) or an output (read by the user). This information accomplishes two things: 1. Allows the CPU board to construct an appropriate hash table based on the variable names, and 2. provides the user information about what variables are available and their behavior.

Byte #:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	...	68	69	70	71	72	73
Value:	*H*	*U*	*M*	*0*	*1*	\0	OUT	*H*	*U*	*M*	*0*	2	\0	OUT	...	*A*	*D*	*C*	I	\0	IN

Table 3: Packing of variable names and input types in `varNames`

Now that we have a proper understanding of the memory structures, we can examine how the memory is organized. Figure 9, below, shows a high-level block diagram of device's memory model.

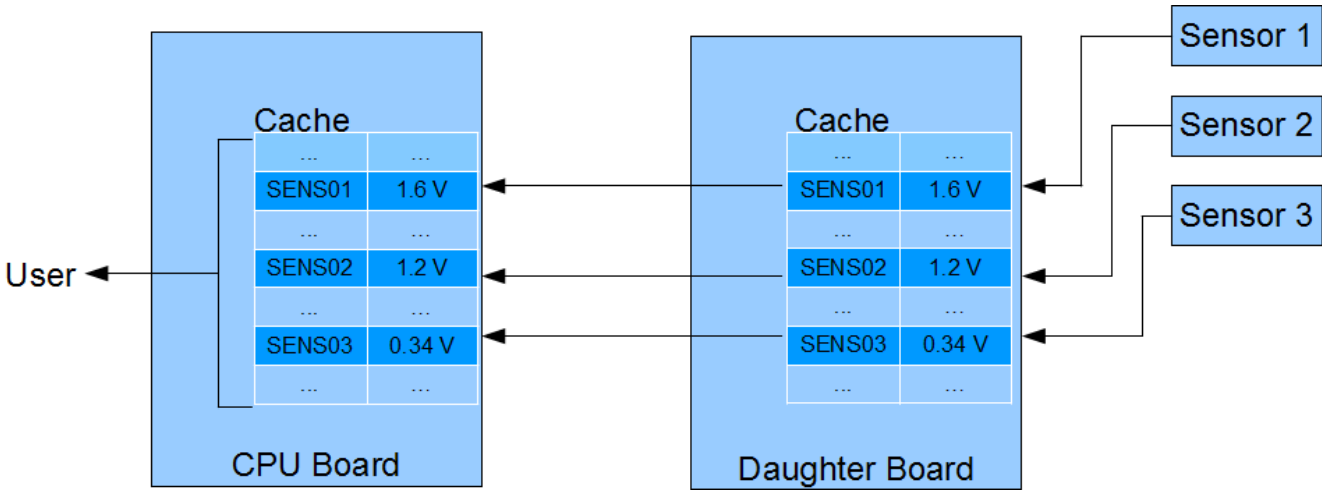


Figure 9: Block diagram of the software memory model

As we can see, each board maintains a local hash table of variables, called the **cache**, that holds useful information regarding itself or, in the CPU Board's case, other boards.

The cache in the daughter board will usually be populated by data relevant to that board. For example, the Environment Board's cache would be filled with analog conversion data, each variable keyed by a name descriptive of its originating sensor, i.e. “HUM01” or “TEMP02.”

Each variable in the CPU Board's cache maps to a variable in one of the daughter boards' caches, as indicated by the dotted lines. Changes in a variable located on the daughter board will illicit a change in the corresponding location on the CPU Board, if the variable in question is an output I/O type, and vise versa if it's an input.

The Requirements for User Commands state the need for duplicate daughter boards with identical variable identifiers. This means the system needs a sufficient method of regularly dealing with collisions. Due to the low clustering capabilities of the Pearson Hash, this project simply uses **linear probing** to separate identical hash values. That is, if two identifiers result in the same hash (such as two “VOLT01” variables on two Load Switchers), the second will simply occupy the next available space in the cache.

In order to track changes between the daughter boards, as well as construct the memory model supporting them, the CPU Board must send read and write requests to the daughter board over SPI.

Serial Peripheral Interface

The **Serial Peripheral Interface** is a four-wire, synchronous bus type with a very simple method of message delivery. Given one unit specified as a master, and another as a slave, the master unit provides a clock cycle and chip select signals to the slave, prompting the slave to receive data as

the master sends them. And, with every bit shifted into the slave through the Master-Out, Slave-In (MOSI) line, the slave can return a corresponding reply on the Master-In, Slave-Out (MISO) line.

The data transferred is unbound by any sort of protocol, allowing users to send an arbitrary amount of data. The simplicity of SPI's delivery system makes it ideal for short-distance, inter-microcontroller communication. **[5]**

In the FOCE Chassis system, all data exchanges between the CPU Board and the daughter boards occur on the SPI1 lines of each PIC, where each packet of information is a 16-bit word. This word is divided into two components: the **command type** and the **payload**, residing in the most significant byte and least significant byte, respectively.

The command type tells the daughter board what kind of behavior the CPU board is expecting from the transaction. Currently there are five command types: **INFO**, **GET**, **SET**, **BUFF_IN**, and **BUFF_OUT**. The payload of each word provides some argument for each command, typically a memory index.

INFO: Daughter Board Identification

The INFO command lets the CPU Board read from the daughter board's CardInfo structure, one byte at a time. This is essential to creating the CPU's local cache, especially when multiple daughter boards of the same type are present.

Initially, the CPU Board will cycle through each of the slots, setting the DIG_OUT_ADDR* pins to connect the SPI1 lines, and sending successive INFO commands until the daughter board's CardInfo is fully parsed.

Each variable name in the received CardInfo `varName` field is then hashed and assigned to a location in the CPU Board's cache. Depending on its I/O type, the hash code will also be added to the **polling list**, a set of GET or SET commands transmitted to the daughter board over a user-defined interval.

GET, SET: CPU BUS Commands

As alluded in the Requirements, the CPU Board's cache is updated by issuing CPU BUS Commands to the daughter cards. These commands are stored in the previously mentioned polling list.

CPU BUS Commands directly perform read and write operations on the daughter board's memory. Naturally, GET performs reads and SET performs writes. Writing a value to an input or inout variable will often prompt a change in one of the output variables, giving the user control over the daughter board's behavior.

CPU BUS Commands, as stated by the Requirements, are only used internally. However, if the user requires more internal information, particularly for debugging purposes, he or she can issue a **Buffered Command**.

BUFF_IN, BUFF_OUT: Buffered Commands

Again referring to the Requirements, the user can submit **Debug Commands** through the RS-232

port on each of the cards. When connected to a computer with a properly configured terminal emulator, this allows the user to perform certain operations and, usually, receiving a detailed print out on screen. This is done by sending string data captured from the serial port over SPI, with BUFF_IN as the packet's command type and a single byte of character data in the payload. The daughter board will then add the payload to a buffer for processing, hence the name **Buffered Commands**. Once finished, the daughter board will write the output text to another buffer, which the CPU Board will then query, one byte at a time, with BUFF_OUT commands.

Before, these Debug Commands were unavailable to the user if he or she was only connected to the CPU Board's RS-232 port, as they were local to a daughter board. However, the user is now capable of sending entire command strings to the daughter boards, as if he or she were connected to them directly. This added functionality should prove useful when the FOCE Chassis is deployed, allowing users to locate operational problems not typically visible from User Commands.

Because entire text strings are sent over SPI, rather than single-word transmissions, the transaction times for Buffered Commands are much longer than CPU BUS Commands. Consequently, while Buffered Commands can be more flexible than CPU BUS Commands, they are significantly slower and are more prone to transmission errors. Thus, they should not regularly be used to gather vital environmental or time-sensitive information.

5 Analysis

One of the larger challenges in this project was inheriting the reasonably large pre-existing code base, and adopting its structural practices. Much of the code for dealing with the board's internal functionality already existed, such as the RS-232 interfacing and text parsing. With so many features already in place, adding new ones without breaking the old is a balancing act of trial and error.

The best way to add new features to an existing code base is to do it slowly and iteratively, with an extensive test plan. **[6]**

RealTerm

The serial UART on each board provided an immensely useful way to debug and test changes. I could easily write debug statements directly to my computer's screen, as if it were a command prompt.

To capture and transmit the serial data I used RealTerm, an open-source program designed to do exactly that.

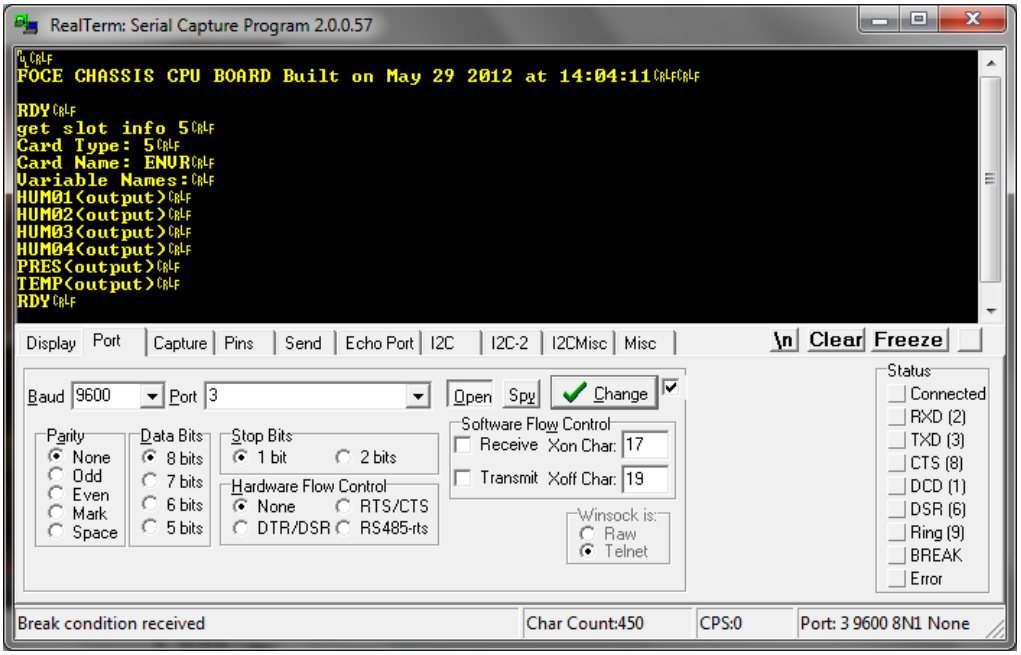


Figure 10: Example of a RealTerm session, interfaced with the CPU Board

The Environment Board

In order to properly test inter-board communications, I would inherently needed a second board to receive the CPU Board's requests. Fortunately, the Environment Board's code base was just as developed as the CPU Board's at the time, making it the best candidate for testing.

Besides the memory structures described in the Design, the Environment Board would also need some source of data to populate its cache. Better yet if that data could be meaningful. As the board is essentially a collection of analog inputs routed through an ADC, gathering the data and compiling it in the PIC would (supposedly) be a straightforward task.

The ADC in question was the Burr-Brown ADS8344, a 16-bit, 8-channel package using an SPI digital interface to control its conversion settings. More on its theory of operation can be found in the package's datasheet. **[7]**

Testing and Development Procedure

Using RealTerm, adding features iteratively, and testing the CPU Board alongside the Environment Board, the development process roughly went as followed:

1. Go through each file in the existing code base and test each function and structure, extensive printfs injected at various points. This, understandably, took the longest amount of time.
2. Understand the existing SPI routines—the initialization code and the test transmission code. Understand how bytes are sent and received.
    - Here, I found the existing code already had a template for what would become the cache and CardInfo structures. The Environment Board would simply read and write from arrays, given an index.
3. Add Buffered Commands.

    3.1. Create a Debug Command to send string data over SPI.

    a) Send string data from CPU Board.

    b) Check if data correctly received by the Environment Board. If incorrect, change code and return to step (a).

    c) Attempt to send copy of the string back to the CPU Board.

    d) Check if data correctly received by CPU Board. If incorrect, change code and return to step (c).

    3.2. Return string data from Environment Board to CPU Board

    a) On the Environment Board, instead of sending a copy of the received string back, forward it to the command parser.

    b) Send the resulting printout back to CPU Board. Repeat send/receive testing from 3.1.

    - This was probably the most time-consuming test cycle, since bugs would occurred in the most innocuous places due to how time-dependent SPI transfers are. Also, the time spent switching between boards and reprogramming the PICs quickly added up.
4. Implement the Pearson Hash on a PC, first. Verify its operation by feeding it entire

Wikipedia pages and locating collisions.

5. Transfer Pearson Hash to CPU and Environment Boards.

   5.1. Perform smaller-scale test with a variety of possible variable names. Since it was already verified on the computer, this stage wasn't too intensive.

   5.2. Create User Commands that read and write data addressed by variable name, rather than index. Use this name to create the hash number, then use this hash number to replace the index mentioned in step 2.

6. Environment Board ADC

   6.1. Test if the on-board sensors are working. i.e., breathe on the humidity sensor and look for a change in the corresponding test point voltage.

   6.2. Create a Debug Command on the Environment board that receives data from the ADC.

      a) Compare received data to measured voltage.

6 Conclusion and Recommendations

This project's goal was to establish a means of communication between the different modules in the FOCE Chassis system. In anticipation for further development, these features were designed to be reasonably expandable, requiring few changes, if any, to the existing code if a new board were to be developed.

While I believe this objective was clearly reached, and was successful, there are still many features that should have been completed. The Ethernet port on the CPU Board remains untouched, despite Mr. Kecy suggesting I research and write some test code for it. Some code exists for the SD Card slot on the CPU Board, which would let users to log data onto a nonvolatile storage medium. Yet, I doubt this feature will be implemented in . Had I finished the primary requirements sooner, these features, and others, may have gotten at least some attention.

I simply hadn't anticipated how extensive and time-consuming some of the research and testing would be. While the requirements were conceptually simple, the finer details of their implementation, such as the timing on the Buffered Commands, required far more trial and error than I'd thought. These same assumptions also led be to de-prioritize the project at some points in favor of other (less vital-to-my-graduation) coursework. This led to some gaps in communication between myself, Prof. Prodanov, and Mr. Kecy.

On a more technical note, while the solution works well, there is definitely room for improvement. For one, the daughter boards use a polling system to update their local caches. That is, like the CPU Board they update their caches based on some interval. Since these changes are only useful when displayed on the CPU Board, much computing power is potentially wasted in updating the cache. I would suggest a more event-driven approach, where the values only update when the CPU Board requests them, but this interferes with the simplicity of the cache. As a majority of the daughter board's SPI processing occurs in interrupts, where function calls are prohibited, this simplicity is as much a technical boon as a mental one.

Also, in order to minimize unintended collisions in the cache, the Pearson lookup table should be altered according to the variable name set. Since these names were not known during this project, I couldn't perform rigorous collision testing. As it is, the current table should be sufficient, provided the load factors remain within reason.

For the sake of simplicity, entries to the cache from identical daughter boards result in unavoidable collisions, which the CPU Board deals with using linear probing. Again, provided the load factor remains reasonable, and over-clustering doesn't occur, this shouldn't be a problem for some time. When that time comes, simply adjusting the lookup table should suffice.

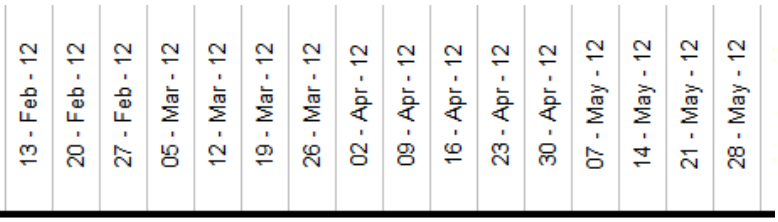
7 Bibliography

1. “The Future Through the Lens of FOCE.” Monterey Bay Aquarium Research Institute. Annual Report 9. 2008. Web. May 2012. <http://www.mbari.org/news/publications/ar/chapters/08_foce.pdf>.
2. Brewer, Peter. “Free-ocean carbon dioxide enrichment (FOCE) experiment.” Monterey Bay Aquarium Research Institute. The MARS Ocean Observatory Testbed. 2010. Web. May 2012. <http://www.mbari.org/mars/science/foce.html>.
3. Nelson, Eric and Sean Tolibas. “FOCE Medium Voltage Load Switcher.” California Polytechnic University, San Luis Obispo. 2011. Web. May 2012. <http://digitalcommons.calpoly.edu/eesp/121/>.
4. Pearson, Peter. "Fast hashing of variable-length text strings."*Communications of the ACM*. 33.6 (1990): 677-680. Web. 24 May. 2012. <http://dl.acm.org/citation.cfm?id=78978>.
5. ."DHC Load Balancing Algorithm." *Internet Engineering Task Force*. Network Working Group, Feb 2001. Web. May 2012. <http://www.apps.ietf.org/rfc/rfc3074.html>.
6. “ADS8344 16-bit, 8-Channel Serial Output Sampling Analog-to-Digital Converter.” Texas Instruments. Web. May 2012. <http://www.ti.com/product/ads8344>
7. Feathers, Michael. *Working Effectively With Legacy Code*. Prentice Hall, 2004. Print.

Appendix A: Schedule

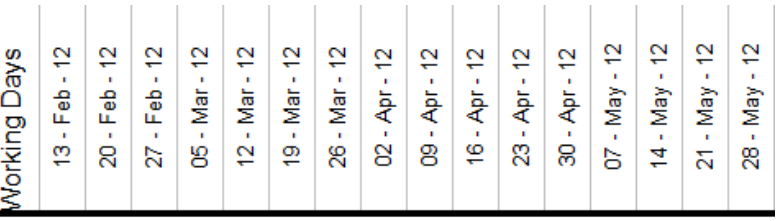
Projected Schedule

WBS	Tasks	Start	End	Duration (Days)
1	**Preparation**	2/16/12	2/24/12	8
1.1	Retrieve FOCE Chassis	2/16/12	2/17/12	1
1.2	Gather Equipment	2/17/12	2/24/12	7
2	**Research**	2/17/12	4/15/12	58
2.1	Existing SPI hardware	2/17/12	3/18/12	30
2.2	Legacy Code	2/17/12	3/18/12	30
2.3	Ethernet	3/18/12	4/01/12	14
2.4	EEPROM (SD Card)	4/01/12	4/15/12	14
3	**Development**	3/18/12	5/15/12	58
3.1	SPI Communication	3/18/12	4/01/12	14
3.2	Ethernet Communication	4/01/12	4/15/12	14
3.3	EEPROM	4/15/12	4/29/12	14
3.4	Report / Documentation	5/01/12	5/15/12	14



Actual Schedule

WBS	Tasks	Start	End	Duration (Days)	Working Days
1	**Preparation**	2/16/12	2/24/12	8	7
1.1	Retrieve FOCE Chassis	2/16/12	2/17/12	1	2
1.2	Gather Equipment	2/17/12	2/24/12	7	6
2	**Research**	2/17/12	4/30/12	73	52
2.1	Existing SPI hardware	2/17/12	3/18/12	30	21
2.2	Legacy Code	3/31/12	4/30/12	30	21
3	**Development**	3/31/12	5/29/12	59	42
3.1	SPI Communication	3/31/12	5/20/12	50	35
3.4	Report / Documentation	5/20/12	5/29/12	9	7



Appendix B: Tools and Equipment

Hardware

- **Microchip MPLAB ICD 2 In-Circuit Debugger**
  Debugger and programmer for the FOCE Chassis' multiple PIC24-based circuits.
- **Hewlett Packard 6202B DC Power Supply**
  Used to power the FOCE Chassis apparatus, which required a DC input of at least 24V to function.
- **EXTech Instruments True RMS Multimeter 430**
  High quality digital multimeter used for various node tests; continuity checks, etc.

Software

- **Microchip MPLAB IDE 8.83**
  Latest development environment compatible with the ICD 2 Debugger. All code for the PIC24 MCUs was developed in this IDE.
- **MPLAB C30 C Compiler, Evaluation Version**
  C compiler for PIC24 MCUs, free version with limited optimization levels.
- **RealTerm Serial Terminal**
  Open source serial data capture program used to communicate to the FOCE Chassis' RS-232 port.
- **Mercurial**
  Version control tool used to manage changes in code.

Appendix C: User Commands

Listed here are descriptions of the top-level commands the user can enter through the CPU Board's RS-232 port, via a serial terminal application (i.e. RealTerm).

READ

READ [VARIABLE NAME] [INSTANCE] [FORMAT]

Reads a value, named [VARIABLE NAME], from the CPU Board's local cache.

If more than one daughter board of the same type is attached to the apparatus, each duplicate board's variables can be accessed by specifying an [INSTANCE]. Instances are numbered according to the slot order of their originating daughter cards. That is, a Load Switcher in slot 3 with a variable named VOLT01 can be accessed with [INSTANCE] = 0, and a Load Switcher in slot 4 with the same variable can be accessed with [INSTANCE] = 1. [INSTANCE] = 0 by default.

To output the requested value in a certain number format, [FORMAT] can be specified as one of the following:

- **INT** signed integer
- **UINT** unsigned integer
- **CHAR** character
- **FLOAT** floating (demical) point number

READ will fail if the specified variable name or instance doesn't exist in the cache.

WRITE

WRITE [VALUE] [VARIABLE NAME] [INSTANCE] [FORMAT]

Writes a value, [VALUE], to the CPU Board's local cache. All other fields serve the same purpose as in READ.

If the user writes to an input variable, the changes will be propagated to the daughter board on the next update interval.

WRITE will fail if the specified variable name or instance doesn't exist in the cache.

INTERVAL

INTERVAL [TIME]

The CPU Board's local cache will update all its values every user-defined interval. This command sets the length of that interval, in clock ticks.

GET SLOT INFO

GET SLOT INFO [SLOT]

Prints information about the daughter board plugged into slot [SLOT] in the backplane. Will display type ID number, name, and a list of all accessible variables the board provides.

GET SLOTS

GET SLOTS

Displays the daughter board types currently attached to the FOCE Chassis system.

SLOT RESCAN

SLOT RESCAN

Clears the CPU Board's local cache and the required behavior operation from each attached daughter board. This is run automatically on start-up, and should only be called if the card configurations changed during runtime.

SEND

SEND [SLOT] [STRING DATA]

Sends raw string data to the specified slot. Allows the user access to debug routines only present on the daughter boards. Should only be used for diagnostic purposes.

Appendix D: Code

Due to the large amount of legacy code, this section only lists files that were created for this project. This does not include files that were extensively modified, for space reasons. A comprehensive listing, with continuously updating documentation, can be found at https://bitbucket.org/rolandmquiros/mbari-foce .

cpu_brd/sys_defs.h

```
/*****************************************************************************/
/* Copyright 2010 MBARI.                                                     */
/* MBARI Proprietary Information. All rights reserved.                       */
/*****************************************************************************/
#ifndef SYS_DEFS_H
#define SYS_DEFS_H

/* general defs */
#define TRUE    1
#define FALSE   0

#define HI      1
#define LO      0

/* serial I/O defs */
#define SER_CONSOLE     2
#define SER_SPARE       2

/* Daughterboard Type defs */
#define TYPE_OPEN    15
#define TYPE_CPU     1
#define TYPE_PS      2
#define TYPE_LVLS    3
#define TYPE_HVLS    4
#define TYPE_ENVIRON 5
#define TYPE_GFDET   6
#define TYPE_UNASS1  7
#define TYPE_UNASS2  8
#define TYPE_UNASS3  9

...
#endif
```

cpu_brd/cache.h

```
/*****************************************************************************/
/* Copyright 2012 MBARI.                                                     */
/* MBARI Proprietary Information. All rights reserved.                       */
/*****************************************************************************/
#ifndef CACHE_H
#define CACHE_H

#define CACHE_IO_OUT   0
#define CACHE_IO_IN    1

#define CACHE_VALLEN 256
extern unsigned int cacheValues[CACHE_VALLEN];

void cacheInit();
```

```
void cacheSetEntry(unsigned char index, unsigned int value,
    unsigned char source, unsigned char iotype);
void cacheGetEntry(unsigned char index, unsigned int* value,
    unsigned char* source, unsigned char* iotype);

#endif
```

cpu_brd/cache.c

```
/*****************************************************************************/
/* Copyright 2012 MBARI.                                                     */
/* MBARI Proprietary Information. All rights reserved.                       */
/*****************************************************************************/
#include <string.h>
#include <stdio.h>
#include "cache.h"
#include "serial.h"
#include "sys_defs.h"

#define CACHE_SRCLEN 128
#define CACHE_IOLEN  32

unsigned int cacheValues[CACHE_VALLEN] = { 0 };
unsigned char cacheIOTypes[CACHE_IOLEN] = { 0 };
unsigned char cacheSources[CACHE_SRCLEN];

void cacheInit()
{
    memset(cacheSources, 0xFF, CACHE_SRCLEN);
}

void cacheSetEntry(unsigned char index, unsigned int value,
    unsigned char source, unsigned char iotype)
{
    unsigned int ioIndex = index / 8;
    unsigned int ioShift = 7 - (index - 8 * ioIndex);
    unsigned int srcIndex = index / 2;

    cacheValues[index] = value;
    cacheSources[srcIndex] = (index % 2 == 0) ?
        (cacheSources[srcIndex] & 0x0F) + ((source << 4) & 0xF0) :
        (cacheSources[srcIndex] & 0xF0) + (source & 0x0F);

    switch (iotype)
    {
        case CACHE_IO_IN: case CACHE_IO_OUT:
            cacheIOTypes[ioIndex] ^= iotype << ioShift;
        break;
    }
}

void cacheGetEntry(unsigned char index, unsigned int* value,
    unsigned char* source, unsigned char* iotype)
{
    unsigned int ioIndex = index / 8;
    unsigned int ioShift = 7 - (index - 8 * ioIndex);
    unsigned int srcIndex = index / 2;

    if (value != NULL)
    {
        *value = cacheValues[index];
    }
```

```
    if (source != NULL)
    {
        *source = cacheSources[srcIndex];
        *source = (index % 2 == 0) ? (*source & 0xF0) >> 4 : *source & 0x0F;
    }

    if (iotype != NULL)
    {
        *iotype = (cacheIOTypes[ioIndex] >> ioShift) & 1;
    }
}
```

cpu_brd/slot.h

```
/*****************************************************************************/
/* Copyright 2012 MBARI.                                                     */
/* MBARI Proprietary Information. All rights reserved.                       */
/*****************************************************************************/

#ifndef SLOT_H
#define SLOT_H

#include <p24fxxxx.h>

#define NUM_SLOTS 8

void slotInit();
void slotSet(unsigned char slot);
void slotSelect(unsigned char slot);
void slotDeselect();
unsigned char slotGetType(unsigned char slot);
const char* slotGetName(unsigned char slot);
unsigned char slotTypeFromName(const char* name);
const char* slotDisplayName(unsigned char type);
void slotDumpInfo(unsigned char slot);
void slotUpdate();

#endif
```

cpu_brd/slot.c

```
#include <string.h>
#include "cache.h"
#include "cardinfo.h"
#include "dig_io.h"
#include "hash.h"
#include "slot.h"
#include "spi.h"
#include "sys_defs.h"

#define SLOT_ID_MSG 0x40
#define SLOT_NAME_LEN 4
#define SLOT_TIMEOUT 1000

CardInfo slotInfo[NUM_SLOTS];

char msg_buff[64];

int slotSendRecv(unsigned int addr, unsigned int* recv)
{
    unsigned int i, rd = 0;

    if (recv == NULL)
```

```
    {
        return 2;
    }

    for (i = 0; i < SLOT_TIMEOUT && rd == 0; i++)
    {
        SPI1STATbits.SPIROV = 0;
        rd = spiSlaveRead(addr, INFO_CMD);
    }
    *recv = rd;

    if (i >= SLOT_TIMEOUT)
    {
        return 1;
    }

    return 0;
}

void slotGetInfo(unsigned char slot)
{
    unsigned int i, recv, offset;

    char* nameBuf;
    unsigned char len;
    unsigned char inout;
    unsigned char hash;
    unsigned char colSlot;

    int err;

    slotSelect(slot);

    /* Read in CardInfo struct */
    for (i = 0; i < sizeof(CardInfo) - 1; i++)
    {
        if ((err = slotSendRecv(i, &recv)))
        {
            memset(&slotInfo[slot], 0, sizeof(CardInfo));
            slotInfo[slot].typeId = TYPE_OPEN;
            break;
        }

        *((unsigned char*)&slotInfo[slot] + i) = recv & ARG_MASK;
    }

    /* Apply variables to cache */
    offset = 0;
    for (i = 0; i < slotInfo[slot].varCount; i++)
    {
        nameBuf = slotInfo[slot].varNames + offset;
        len = strlen(nameBuf);
        hash = hashCode(nameBuf, len);

        offset += len;
        inout = slotInfo[slot].varNames[++offset];
        offset++;

        /* Check for collisions in the cache */
        do {
            cacheGetEntry(hash, 0, &colSlot, 0);
            if (colSlot != TYPE_OPEN)
            {
```

```
                /* Insert into the next available bucket */
                hash++;
            }
        }
        while (colSlot != TYPE_OPEN);

        /* Apply to cache */
        cacheSetEntry(hash, 0xFFFF, slot, inout);
    }
}

void slotInit()
{
    unsigned char i;

    /* Retrieve daughtercard information */
    for (i = 0; i < NUM_SLOTS; i++)
    {
        slotGetInfo(i);
    }
}

void slotSet(unsigned char slot)
{
    switch(slot)
       {
              case 0:
              {
                     digSetOutBit(0,1);  // Gate
                     digSetOutBit(1,0);  // Add 0
                     digSetOutBit(2,0);  // Add 1
                     digSetOutBit(3,0);  // Add 2
                     break;
              }
              case 1:
              {
                     digSetOutBit(0,1);  // Gate
                     digSetOutBit(1,1);  // Add 0
                     digSetOutBit(2,0);  // Add 1
                     digSetOutBit(3,0);  // Add 2
                     break;
              }

              case 2:
              {
                     digSetOutBit(0,1);  // Gate
                     digSetOutBit(1,0);  // Add 0
                     digSetOutBit(2,1);  // Add 1
                     digSetOutBit(3,0);  // Add 2
                     break;
              }

              case 3:
              {
                     digSetOutBit(0,1);  // Gate
                     digSetOutBit(1,1);  // Add 0
                     digSetOutBit(2,1);  // Add 1
                     digSetOutBit(3,0);  // Add 2
                     break;
              }
              case 4:
              {
                     digSetOutBit(0,1);  // Gate
```

```
                    digSetOutBit(1,0);  // Add 0
                    digSetOutBit(2,0);  // Add 1
                    digSetOutBit(3,1);  // Add 2
                    break;
             }

             case 5:
             {
                    digSetOutBit(0,1);  // Gate
                    digSetOutBit(1,1);  // Add 0
                    digSetOutBit(2,0);  // Add 1
                    digSetOutBit(3,1);  // Add 2
                    break;
             }
             case 6:
             {
                    digSetOutBit(0,1);  // Gate
                    digSetOutBit(1,0);  // Add 0
                    digSetOutBit(2,1);  // Add 1
                    digSetOutBit(3,1);  // Add 2
                    break;
             }
             case 7:
             {
                    digSetOutBit(0,1);  // Gate
                    digSetOutBit(1,1);  // Add 0
                    digSetOutBit(2,1);  // Add 1
                    digSetOutBit(3,1);  // Add 2
                    break;
             }
             default:
             {
                    digSetOutBit(0,0);  // Gate
                    digSetOutBit(1,0);  // Add 0
                    digSetOutBit(2,0);  // Add 1
                    digSetOutBit(3,0);  // Add 2
                    break;
             }
      }
}

void slotSelect(unsigned char slot)
{
    slotSet(slot);
    spiCS(1);
}

void slotDeselect()
{
    slotSet(-1);
    spiCS(0);
}

unsigned char slotGetType(unsigned char slot)
{
    if (slot < NUM_SLOTS) {
        return slotInfo[slot].typeId;
    }
    return 0;
}

const char* slotGetName(unsigned char slot)
{
```

```
    if (slot < NUM_SLOTS)
    {
        const char* ret = slotInfo[slot].name;
        return ret;
    }

    return 0;
}

const char* slotDisplayName(unsigned char type)
{
    switch (type)
    {
        case TYPE_OPEN:
            return "Open";
        case TYPE_CPU:
            return "CPU";
        case TYPE_PS:
            return "Power Supply";
        case TYPE_LVLS:
            return "LVLS";
        case TYPE_HVLS:
            return "HVLS";
        case TYPE_ENVIRON:
            return "Environment";
        case TYPE_GFDET:
            return "GFDET";
        case TYPE_UNASS1: case TYPE_UNASS2: case TYPE_UNASS3:
            return "Unassigned";
        default:
            return "Unknown";
    }

    return 0;
}

void slotDumpInfo(unsigned char slot)
{
    if (slot < NUM_SLOTS)
    {
        cardInfoDump(&slotInfo[slot]);
    }
}

void slotCardUpdate(CardInfo *card)
{
    unsigned int i;
    unsigned int len;
    unsigned char hash;
    unsigned int offset = 0;

    unsigned int dat;

    /* Iterate through variable names */
    for (i = 0; i < card->varCount; i++)
    {
        len = strlen(card->varNames + offset);
        hash = hashCode(card->varNames + offset, len);
        offset += len + 2;

        /* Update value */
        dat = spiSlaveRead(hash, READ_CMD);
        cacheValues[hash] = dat;
```

```
    }
}

void slotUpdate()
{
    unsigned int i;
    for (i = 0; i < NUM_SLOTS; i++)
    {
        if (slotInfo[i].typeId != TYPE_OPEN)
        {
            slotSelect(i);
            slotCardUpdate(&slotInfo[i]);
            slotSelect(99);
        }
    }
}
```

common/cardinfo.h

```
/*****************************************************************************/
/* Copyright 2012 MBARI.                                                     */
/* MBARI Proprietary Information. All rights reserved.                       */
/*****************************************************************************/
#ifndef CARDINFO_H
#define CARDINFO_H

#define CARDINFO_NAME_LEN 5
#define CARDINFO_VAR_LEN  128

struct CardInfo
{
    unsigned char typeId;
    char          name[CARDINFO_NAME_LEN];
    unsigned char varCount;
    char          varNames[CARDINFO_VAR_LEN];
    unsigned char vnoff;
} __attribute__ ((__packed__));

typedef struct CardInfo CardInfo;

int cardInfoAddVar(const char* name, unsigned char inout, CardInfo* card);
void cardInfoDump(const CardInfo* card);

#endif
```

common/cardinfo.c

```
/*****************************************************************************/
/* Copyright 2012 MBARI.                                                     */
/* MBARI Proprietary Information. All rights reserved.                       */
/*****************************************************************************/
#include <string.h>
#include <stdio.h>
#include "cardinfo.h"
#include "errors.h"
#include "serial.h"
#include "sys_defs.h"

#define CARD_VAR_OUT   0
#define CARD_VAR_IN    1

#define INOUT_SHIFT 5
```

```
static char msg_buf[64];

int cardInfoAddVar(const char* name, unsigned char inout, CardInfo* card)
{
    unsigned int len;

    if (name == NULL || card == NULL) {
        return ERR_FAILURE;
    }

    len = strlen(name);

    /* Make sure there's enough room */
    if (len + 2 >= CARDINFO_VAR_LEN)
    {
        return ERR_FAILURE;
    }

    strcpy(card->varNames + card->vnoff, name);
    card->vnoff += len;
    *(card->varNames + card->vnoff) = 0;
    card->vnoff++;

    *(card->varNames + card->vnoff) = inout;
    card->vnoff++;

    return ERR_SUCCESS;
}

void cardInfoDump(const CardInfo* card)
{
    unsigned int i;
    unsigned int offset = 0;
    unsigned int inout;
    char* nameBuf;
    char* varPtr;

    if (card == NULL)
    {
        return;
    }

    sprintf(msg_buf, "Card Type: %d\r\n", card->typeId);
    serPutString(SER_CONSOLE, msg_buf);

    sprintf(msg_buf, "Card Name: %s\r\n", card->name);
    serPutString(SER_CONSOLE, msg_buf);

    serPutString(SER_CONSOLE, "Variable Names:\r\n");
    for (i = 0; i < card->varCount; i++)
    {
        nameBuf = card->varNames + offset;
        serPutByte(SER_CONSOLE, '\t');
        serPutString(SER_CONSOLE, nameBuf);

        offset += strlen(nameBuf);
        inout = card->varNames[++offset];
        offset++;

        switch (inout)
        {
            case CARD_VAR_IN:
                serPutString(SER_CONSOLE, "(input)\r\n");
```

```
                break;
            case CARD_VAR_OUT:
                serPutString(SER_CONSOLE, "(output)\r\n");
                break;
            default:
                serPutString(SER_CONSOLE, "\r\n");
        }
    }
}
```

common/hash.h

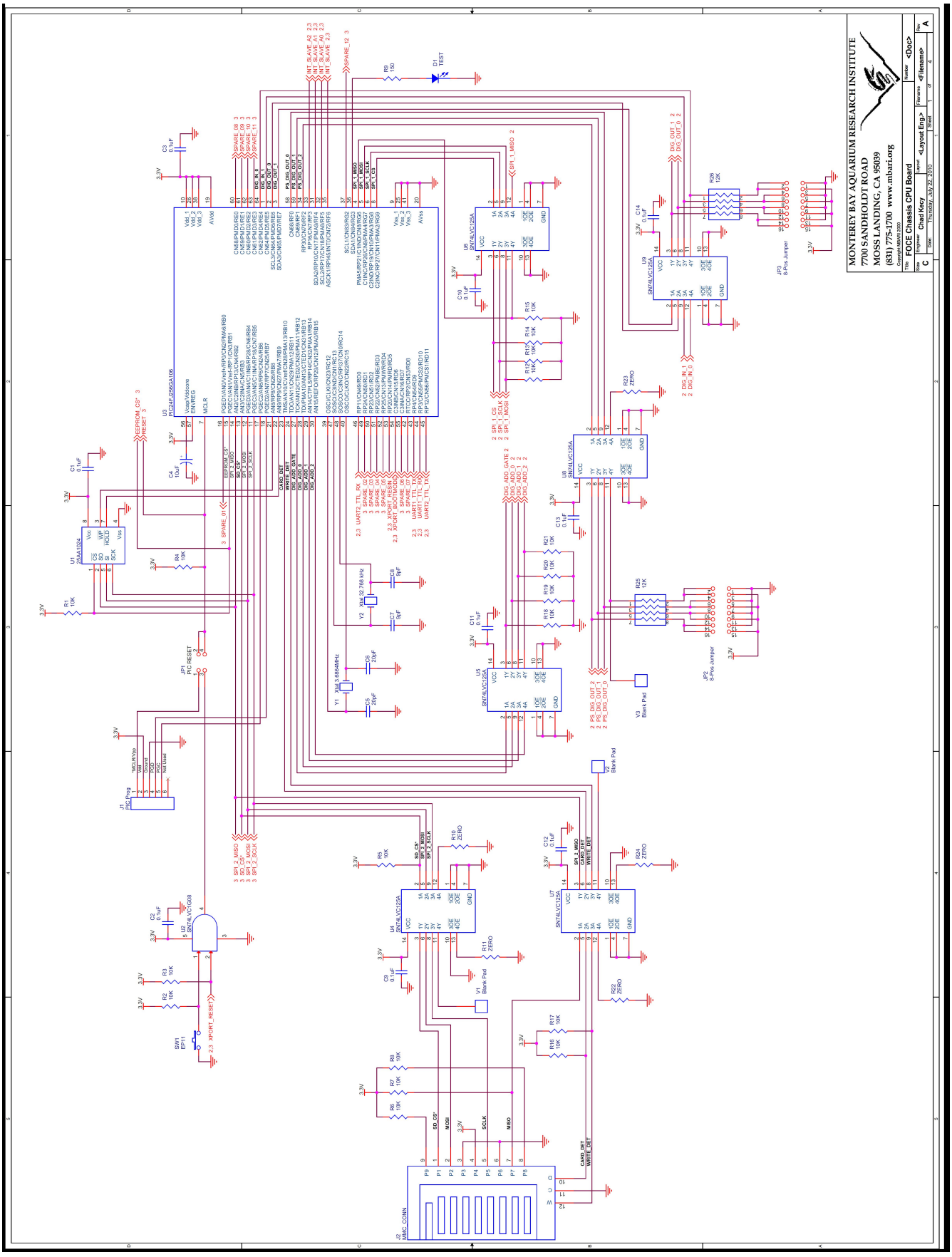
```
/*****************************************************************************/
/* Copyright 2012 MBARI.                                                     */
/* MBARI Proprietary Information. All rights reserved.                       */
/*****************************************************************************/
#ifndef HASH_H
#define HASH_H

#define HASH_MAX 256

unsigned char hashCode(const char* dat, unsigned int len);

#endif
```

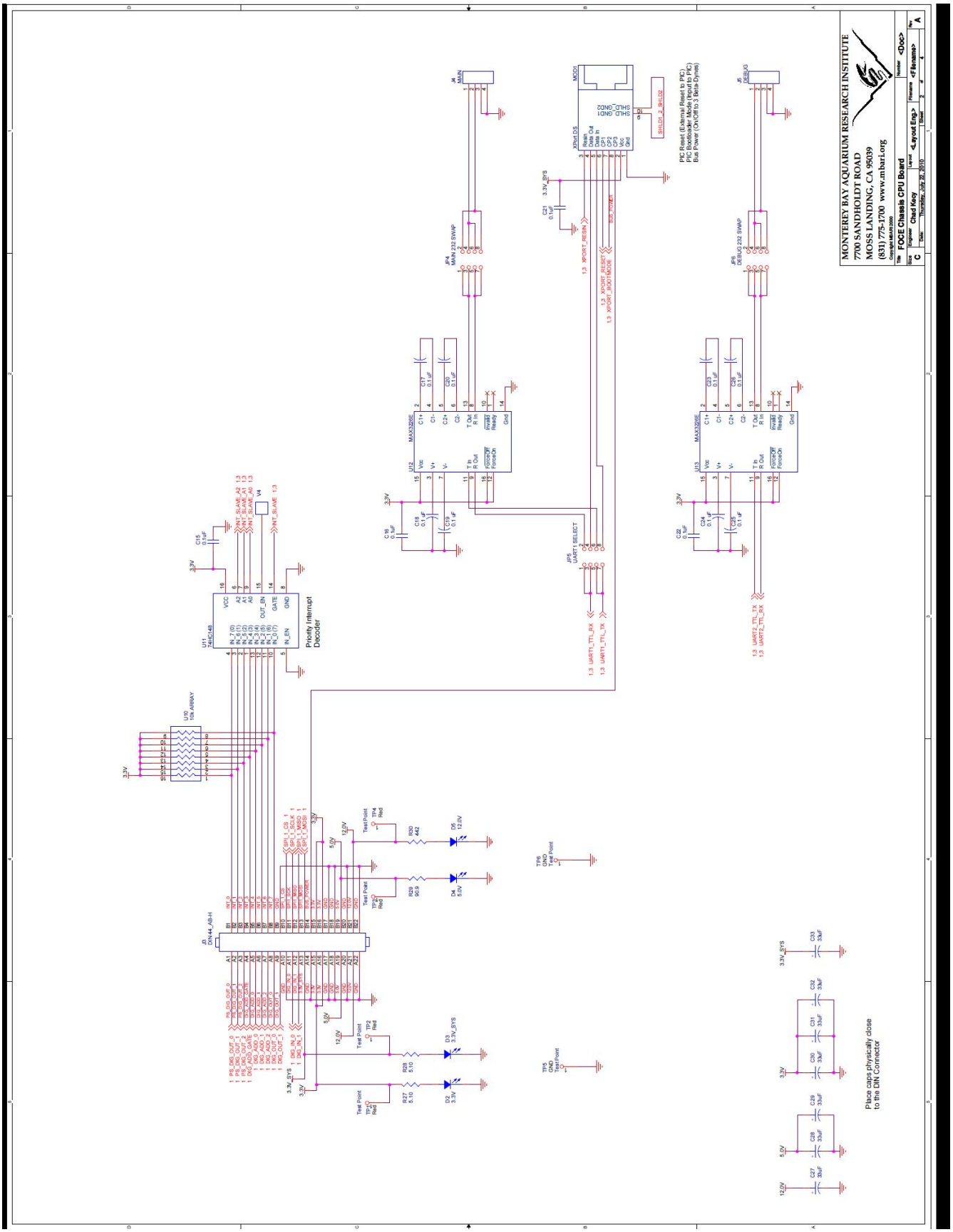
common/hash.c

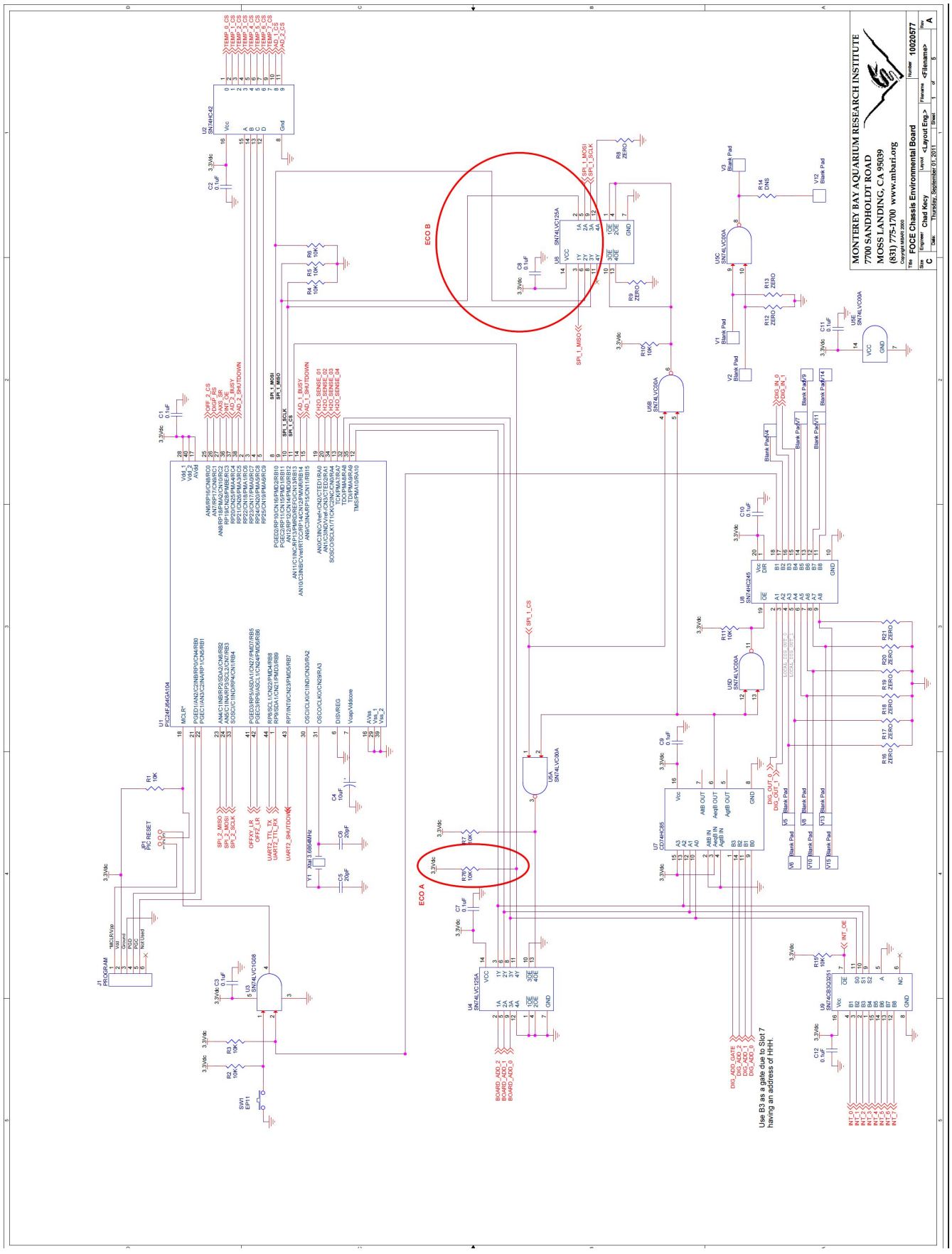
```
#include "hash.h"

static unsigned char pearsonTable[HASH_MAX] =
{
    41, 179, 9, 40, 93, 224, 24, 94, 39, 110, 70, 85, 196, 129, 80,
    11, 127, 42, 87, 8, 243, 96, 174, 153, 61, 192, 17, 20, 235, 7, 47,
    222, 53, 216, 101, 112, 44, 139, 109, 233, 108, 57, 240, 229, 160, 219, 251,
    15, 221, 245, 239, 104, 182, 33, 138, 59, 166, 247, 146, 190, 31, 125, 188,
    114, 97, 86, 157, 106, 3, 142, 66, 69, 152, 177, 116, 50, 149, 242, 144,
    28, 29, 95, 37, 2, 117, 246, 65, 183, 206, 241, 230, 131, 13, 145, 63,
    98, 200, 176, 226, 5, 158, 189, 49, 18, 54, 30, 120, 227, 68, 48, 91,
    89, 252, 210, 185, 203, 140, 171, 155, 56, 162, 249, 201, 134, 244, 67, 197,
    148, 248, 82, 100, 34, 90, 236, 51, 167, 178, 207, 193, 164, 225, 218, 147,
    79, 71, 217, 151, 38, 198, 215, 212, 123, 156, 76, 121, 113, 253, 55, 255,
    23, 25, 231, 187, 21, 72, 209, 103, 81, 35, 12, 6, 122, 78, 32, 60,
    92, 14, 202, 83, 128, 195, 135, 223, 211, 181, 84, 0, 119, 173, 165, 234,
    45, 208, 107, 136, 199, 126, 141, 163, 180, 99, 172, 220, 75, 115, 232, 111,
    228, 204, 237, 73, 254, 62, 105, 132, 22, 36, 118, 143, 161, 133, 169, 10,
    52, 186, 184, 170, 77, 205, 137, 124, 238, 16, 159, 250, 191, 74, 27, 130,
    4, 194, 1, 46, 58, 102, 64, 168, 150, 88, 214, 175, 213, 154, 26, 43
};

unsigned char hashCode(const char* dat, unsigned int len)
{
    unsigned char h = 0;
    unsigned int i;

    for (i = 0; i < len; i++)
    {
        h = pearsonTable[h ^ dat[i]];
    }

    return h;
}
```

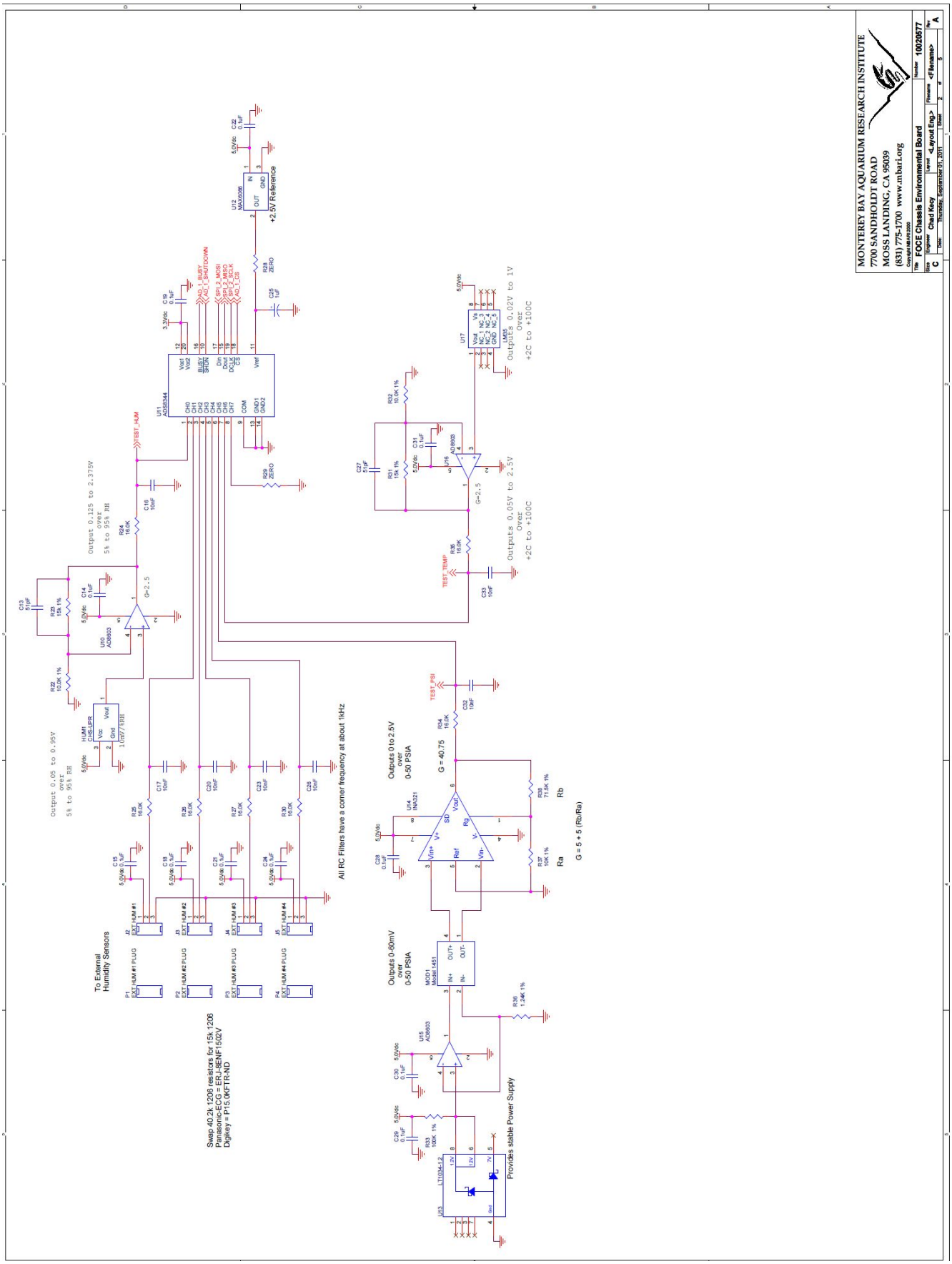
Appendix E: Schematics

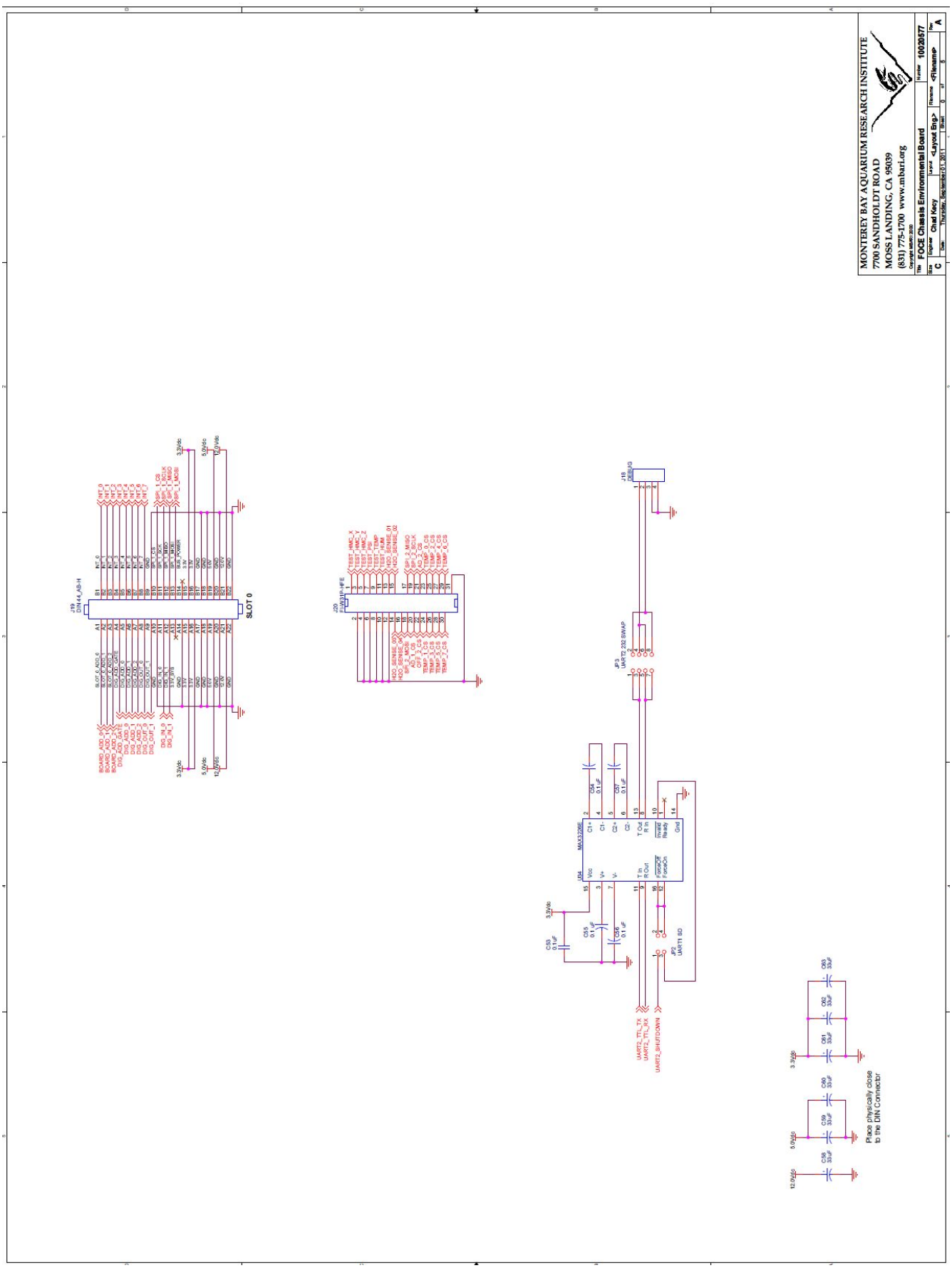
As all hardware was previously designed and provided at the onset of the project, none of the schematics attached represent my work performed. They are provided for the sake of completion.


MONTEREY BAY AQUARIUM RESEARCH INSTITUTE
7700 SANDHOLDT ROAD
MOSS LANDING, CA 95039
(831) 775-1700 www.mbari.org
Copyright MBARI 2000
Title FOCE Chassis CPU Board
Number <Doc>
Size C
Engineer Chad Kecy
Layout Engineer
Filename <Filename>
Date Thursday, June 22, 2000
Sheet 1 of 1
Rev A
U1 25AA1024
U2 SN74LVC1G08
U3 PIC24FJ256GA106
U4 SN74LVC125A
U5 SN74LVC125A
U6 SN74LVC125A
U7 SN74LVC125A
U8 SN74LVC125A
U9 SN74LVC125A
J1 PIC Prog
J2 MMC_CONN
JP1 PIC RESET
JP2 6-Pos Jumper
JP3 8-Pos Jumper
SW1 EVQ11
D1 TEST
Y1 Xtal 3.6864MHz
Y2 Xtal 32.768 kHz
V1 Blank Pad
V2 Blank Pad
V3 Blank Pad
3.3V
SD_CS*
MOSI
SCLK
MISO
CARD_DET
WRITE_DET


MONTEREY BAY AQUARIUM RESEARCH INSTITUTE
7700 SANDHOLDT ROAD
MOSS LANDING, CA 95039
(831) 775-1700 www.mbari.org
FOCE Chassis CPU Board
Chad Kecy
Priority Interrupt Decoder
Place caps physically close to the DIN Connector


MONTEREY BAY AQUARIUM RESEARCH INSTITUTE
7700 SANDHOLDT ROAD
MOSS LANDING, CA 95039
(831) 775-1700 www.mbari.org
FOCE Chassis Environmental Board
Chad Kecy
Thursday, September 01, 2011
10020577
ECO A
ECO B
Use R3 as a gate due to Slot 7 having an address of HHH.


MONTEREY BAY AQUARIUM RESEARCH INSTITUTE
7700 SANDHOLDT ROAD
MOSS LANDING, CA 95039
(831) 775-1700 www.mbari.org
FOCE Chassis Environmental Board
Chad Key
<Layout Eng>
Thursday, September 01, 2011
+2.5V Reference
Output 0.125 to 2.375V over 5% to 95% RH
Output 0.05 to 0.95V over 5% to 95% RH
Outputs 0.02V to 1V Over +2C to +100C
Outputs 0.05V to 2.5V Over +2C to +100C
Outputs 0to2.5V over 0-50 PSIA
Outputs 0-50mV over 0-50 PSIA
G = 40.75
To External Humidity Sensors
All RC Filters have a corner frequency at about 1kHz
Swap 40.2k 1206 resistors for 15k 1206
Panasonic-ECG = ERJ-8ENF1502V
Digikey = P15.0KFTR-ND
Provides stable Power Supply
Rb
Ra
G = 5 + 5 (Rb/Ra)


MONTEREY BAY AQUARIUM RESEARCH INSTITUTE
7700 SANDHOLDT ROAD
MOSS LANDING, CA 95039
(831) 775-1700 www.mbari.org
FOCE Chassis Environmental Board
Chief Kecy
Thursday, September 01, 2011
SLOT 0
Place physically close to the DIN Connector