MOSES Flight Software
A Guide to the Source

Reginald M Mead

09 July 2005

Contents

1 Introduction

This manual is not intended as a Software User's Guide. It is instead offered as a guide to help the maintainer of the MOSES Flight Software to understand the software at the source code level. However, this guide is also not meant to replace the code as being the main source of information on the software. An in-depth understanding of this software can only be achieved by scouring the source code. This might come as a harsh reality to some, but this guide attempts to act as a road map for navigating around this potentially intimidating reality.

It should also be noted that this manual is bound to contain errors. Again, the source should be used as the final arbiter of what is correct and what is incorrect.

2 **V**irtual **T**i**M**er and **U**plink(VTMU) Deck

The VTMUDeck is the interface through which timers and uplinks gain entry into the system. In essence, this program is nothing more than a loop which poles several banks of digital I/O pins looking for state changes and taking the appropriate actions when these changes occur. Currently, there are five predefined timers, nine uplinks, and one somewhat unrelated shutter signal. These signals are read in on DIO ports A & B using the DIOBoard class. Changes in the pins are exposed by taking the exclusive or of the new pins and the old pins. Any bits that have changed become ones while those that have stayed the same become zeros. Currently, the software only supports having a maximum of one bit change on any given iteration. Fortunately, due to the nature of the Timers and Uplinks, having concurrent signals is very unlikely, and would most likely be the result of a very unusual situation. Also, the software has roughly one millisecond timing resolution, which should be sufficient for its target environment.

Pin position must be calculated to discern the meaning of the input. As the input is read in as an unsigned integral 8 bit value, the position of the changed bit can be calculated by the log_2 of the value. The initial result which is of type double must be converted to an int. Due to the internal representation of doubles, it is possible for a value such as 15 to be stored as 14.99999... This poses a problem because a conversion to int will merely truncate anything past the decimal point. In the previous example, log_2 128 becomes 14.99999 which is converted to 14! To avoid this, we take the value multiplied by 1000 plus one, then we divide by 1000 to get the final result. So, 14.99999 becomes 14999.99 = 14999 + 1 = 15000 / 1000 = 15. This accomplished by the following code:

```
temp = (board.readPins(DIOBoard::PORTA) << 8 |
        board.readPins(DIOBoard::PORTB) );
diff = temp ^ pins;
double dindex = log10(double(diff)) / log10(2.0);
int index = (int(dindex*1000)+1) / 1000;
```

Where pins was the value of pins after the previous read.

2.1 Timers

As previously mentioned, there are five predefined timer signals. These include *Data Start, Data Stop, Dark 2, Dark 4,* and *Sleep*. All but the *Sleep* timer are used to initiate exposure sequences. The difference between the Dark Sequences and the Data Sequence is that only the Data Sequence opens the shutter during an exposure. It should also be noted that the *Data Stop* timer serves as a dual purpose timer. It not only instructs the Data Sequence to stop, but it also initiates the Dark 3 Sequence. When any of these signals are detected, the VTMUDeck uses MIOPipe to convey the signal to the MDAQ Program. Also, the IOSPipe is used to inform the IOServer of the signals arrival.

The *Sleep* timer is used to put the computer to sleep while it is coming back down to Earth. This is mostly concerned with conserving power so that there is enough left to extract the data once the payload is recovered. The sleep command first uses the IOSPipe to inform the IOServer of the signals arrival. Next it forks to create a new process. This process is then overwritten with the 'apmsleep' command by using the execlp system call. When given the '–standby' parameter, the 'apmsleep' command proceeds to put the computer into standby mode until it either receives an interrupt or exceeds its timeout limit. This limit is given as a parameter in the form "+hh:mm". In this program the limit is set as 20 minutes. The command as a whole appears as follows:

```
if(fork() == 0)
  execlp("apmsleep","apmsleep","--standby","+00:20",(char *)0);
```

2.2 Uplinks

There are currently nine uplinks that the VTMUDeck is programmed to respond to. Many of these only serve as an added layer of redundancy and perform virtually the same task as one of the predefined timers. The benefit to having the uplinks is that they can be issued at any time and can be issued multiple times, should an earlier uplink / timer fail or need to be repeated. Besides the *Data Start, Data Stop, Dark 2, Dark 4,* and *Sleep* uplinks, which perform obvious functions, there are *Dark 1, Dark 3, Wake,* and *Test* uplinks as well. There might be some curiosity at this point as to why *Dark 1* does not have its own timer. The reason is simple. As it is currently planned, the Dark 1 Sequence will be taken while the payload is still on the rail. Therefore, there is no need, and in fact it would make no sense, to have a pretimed timer. The *Dark 3*timer should be self explanatory.

This brings us to *Wake* and *Test* . To be honest *Wake* does not wake the system as you might naturally assume. In order to execute the *Wake* code, the computer must allready be awake. Therefore, the *Wake* code only serves to inform the IOServer that the computer has been awoken. In some sense the *Test* uplink is very similar to *Wake* . It too only serves to inform the IOServer of its arrival. This might make more sense for *Test* though, because its purpose is to test the responsiveness of the computer and in particular the IOServer.

2.3 Shutter

Finally we come to the shutter signal. This signal indicates whether the shutter is open or closed. In this sense, it is the only pin that we would also like to know when it goes off as well as when it comes on. Therefore, there are two nearly identicle pieces of code, one for when the pin goes high and one for when it goes low. In both cases a highly precise timing is acquired after this signal has been received. This timing is then sent to the IOServer along with a specification as to whether this is a shutter open or close signal, and all of this information is formated and sent down the H/K Downlink. It can be used later to determine the acuracy of our exposure durations.

It might be helpfull to note that high resolution timing is accomplished through the unix gettimeofday() function. This function is particularily usefull because it can achieve microsecond resolution. It is the basis for all high resolution timing in this software.

3 Power Server

The PowerServer is responsible for turning on and off power to subsystems as well as keeping track of the current power configuration. Changes and queries can be made during the experiment by communicating with the PowerServer through the PowerPipe class. This class uses a system FIFO to send commands to the PowerServer. The PowerServer can also be used in conjunction with the startpower script and the powersys.conf configuration file to power on any necessary systems in a predefined order at startup. These options will be covered later.

3.1 Internal Structure

Internally, the PowerServer is a continuous loop which reads command from a PowerPipe object and uses a PowerSystem object to fulfill the commands. The PowerSystem class is merely a PowerServer oriented wrapper around the DIOBoard class. So when you tell it to turn on a subsystem, you are really just telling it to assert a DIO pin. The PowerSystem class uses DIO ports C and D.

There are four different types of commands. There are *Up* commands that tell the PowerServer to power up a system, *Down* commands that tell the PowerServer to power down a system, *Query* commands which ask for the status of a system, and there is also an *Exit* command. When the commands come out of the PowerPipe, they are in the form of a three character string. The first character is either 'U' for up, 'D' for down, 'Q' for query, or 'E' for exit. The remaining characters are either "XT" in the case of the exit command, or they are numbers specifying the pin / system that is of interest. For the *Up* and *Down* commands, the remaining characters can also be "AL" specifying that all systems should be either powered up or down.

3.2 Automatic Powerup at Startup

As mentioned earlier, the startpower script can be used in conjunction with the powersys.conf configuration file to automatically power up any desired systems at computer boot time. The script can either interactively prompt the user for every system included in powersys.conf or it can be made to automatically start all systems through the use of the -a flag. The powersys.conf file uses a very simple structure. Each subsystem must have a name and number. The number is basically the system's DIO pin number. The name is only used for display purposes in order to let the user know what is happening. The name and number must be separated by an equals sign and the whole line needs to end with a colon. The systems are started in the order in which they appear in the file

As an example, Lets assume that we have device1, device2, and device3. They need to be started in that order and they have been assigned DIO pins 5, 6, and 7 respectively. The powersys.conf file would look as follows:

```
#powersys.conf

device1=05:
device2=06:
device3=07:
```

These devices will be powered up at startup if the 'startpower -a' script is run after the PowerServer has been started.

4 Mission Data AcQuisition(MDAQ)

The Mission Data Acquisition or MDAQ program is probably the most essential piece of software in the entire MOSES collection. It is responsible for orchestrating the part of the experiment that collects actual science data. This involves multiple steps, first there must be a predefined exposure sequence that dictates the number of exposures and the durations of each individual exposure in the sequence. When an exposure sequence is initiated, the MDAQ program must operate the shutter correctly, command the Read Out Electronics(ROE) to read out the images on the CCD cameras, acquire the data with a sustained rate of 4MB/s, and finally it needs to both write the data to disk in an acceptable format and send as much data as it can down the high-speed data telemetry link. Besides this main task, the program must also be able to communicate with the IOServer and alter its behavior according to any such requests received. This process requires the combined efforts of many individual pieces. The rest of this section will focus on these pieces and how they work together.

4.1 Internal Structure

The MDAQ program can logically be divided into 4 different sections. In general, these sections correspond with the different threads, however one section is actually composed of five threads with each doing virtually the same thing. These sections include the main thread, which initializes all of the variables, starts the threads, and then waits for the threads to finish, the telemetry thread, which streams science down down the telemetry link as it becomes available after acquisition, the interactive I/O thread, which allows interprocess communication between the MDAQ program and the other pieces of software, and finally the fourth section is the data capture section, which executes exposure sequences, acquires data, and write the data to the disk. Due to some clashes between system signals and threads, each signal must have its own thread that waits for the signal to appear and then starts the data acquisition process. Therefore, all five signal threads get lumped into one section from a logical standpoint. An in-depth discussion on each of these sections will follow.

4.1.1 Main Thread

As previously mentioned, the main thread is mostly responsible for initializing variables, starting all of the threads and then waiting for the threads to finish. This is no doubt one of the simplest sections, but it still performs several critical tasks. To begin with, there are several attributes and settings that can be specified by a user before beginning the program. These settings are stored in a file, usually mdaq.conf, and they must be extracted and parsed at the beginning of the program. This process is facilitated through the use of an InitAttr object. A discussion of the InitAttr class will be postponed until later in this document. At this point it will suffice to understand that these attributes are retrieved at this point in the code. Arguably the most important piece of information that is retrieved in this way is a set of sequence file names. These file names define which sequence file is associated with any given signal. This is particularly useful, because it means that a user can have any number of possible sequences stored in separate files and can specify in the mdaq.conf file which sequences to use just before a run.

Next, Sequence objects are created by passing the sequence filenames as arguments to the Sequence constructor. It is useful to encapsulate sequence information in an object because it makes performing operations on the sequence very convenient. They can also keep track of the current position in a sequence while it is being executed. All of these features are discussed in detail later. At the same time that the Sequence objects are being created, they are also being associated with a signal number through the use of an associative array. After this point, it becomes trivial to determine which sequence to execute after receiving a signal.

The Main thread also uses this initial setup time to ensure that the Read Out Electronics(ROE) is ready to take images in a desired manner. It does so through use of a ReadOutElectronics object, covered later. The ReadOutElectronics object is instructed to exit default mode, reset, and then exit default mode again. The cause for this seeming redundancy is that commanding the ROE to exit default mode while in command mode has no effect. However, commanding the ROE to reset while it is in default mode causes a software error. Therefore, we exit default mode once to make sure that the ROE is commandable, tell it to reset so that we know what configuration it is in, and then finally exit default mode again to make it commandable in a known configuration.

MDAQ then configures the program for the proper use of signals and starts all of the threads. At this point the main thread has nothing more to do, so it waits for an exit signal. Upon receipt of this signal, the other threads are instructed to finish, killed if necessary, and resources are destroyed and given back to the system. This concludes the duties given to the main thread.

4.1.2 Telemetry Thread

Telemetry in MDAQ simply involves sending acquired data(images) out of the telemetry port as they become available. Unfortunately, this process isn't as easy as writing the data to disk. The reason behind this, is that the telemetry link operates at a much slower data rate than the storage disk or data acquisition hardware. In fact, it is going to be impossible to send all of the data that we receive down the telemetry link. Therefore, the telemetry process is merely concerned with the sustained transmission of as much data as possible.

The driver design of the telemetry card makes writing to the telemetry port virtually as simple as opening and writing to a file. The telemetry port is specified as /dev/sfc0. Accordingly this file/device is opened for writing at the beginning of this thread. The only non file-like operation that is performed is an ioctl to the device that flushes the transmission buffer. From this point on, the user should be able to treat this device as a file, in theory.

Filenames of files that need to be written to the telemetry port are obtained in the telemetry thread from a queue of file names. These file names are written to the queue in a data acquisition thread after the files have been written to disk. In essence, the telemetry thread is basically a big loop that waits for the queue to become not empty and then systematically opens and writes the files to the port until the queue becomes empty again. In the actual experiment, it is very likely that the queue will remain empty during the beginning of the experiment, but remain populated once it begins to receive file names. When this loop is instructed to terminate by the main thread, it expunges the names of any untransmitted files from the queue.

As a final note on the telemetry thread, it also formats the data in such a way that on the other end, an observer can tell where one file ends and another begins. This is accomplished by adding a header and footer to every file. The header and footer contain bitpatterns that are extremely unlikely if not impossible to appear in actual data. The code should be consulted to learn the actual bitpatterns used.

4.1.3 Interactive I/O Thread

Communication with the MDAQ program from the outside world is accomplished through the interactive I/O thread. This thread serves as an interface for querying and altering the behavior of the MDAQ program while it is running. Commands to the MDAQ program are acquired through the use of an MIOPipe object, which utilizes a system FIFO to facilitate interprocess communication. It is recommended that other programs whishing to communicate with the MDAQ program use the MIOPipe class to ensure convenience and compatibility.

At this point all of the different interactive I/O options will be listed, but it is highly recommended that the MIOPipe class be used / consulted to understand how these options are used:

Set the sequence name associated with a signal

Set the output name used as a stub for file names

Get the Process ID for a signal / thread

Get the sequence name associated with a signal

Get the sequence information(exposure durations) for a signal

Get the output name used for stubs

Get the name of the currently executing sequence, if any

Get the length of the current exposure

Get the index of the current exposure in its exposure sequence

Get the status(on/off) of selftest mode

Get the status(on/off) of stims mode

Get the status(on/off) of telemetry output

Get the status(on/off) of producing channel 0 data

Get the status(on/off) of producing only positive channel data

Scale the current sequence by a specified ratio

Translate the current sequence by a specified delta value

Find and replace all occurrences of one exposure duration with another for the current sequence

Jump to a specified index in the current sequence

Save the current sequence configuration to a specified file

Find an exposure duration in the current sequence and jump to it

Begin / Resume the current sequence

Pause / Stop(if received twice) the current sequence

Stop the current exposure immediately.

Turn telemetry output on/off

Turn Channel 0 data production on/off

Turn Positive channel only data production on/off

Open the Shutter

Close the Shutter

Get the value of an ROE House Keeping parameter

Reset the ROE

Command the ROE to exit default mode

Command the ROE to enter selftest mode

Command the ROE to enter/exit stims mode

Get the Analogue Electronics parameters for the ROE

Set the Analogue Electronics parameters for the ROE

Again, to see how these options are used and implemented, it is recommended that the MIOPipe and MDAQ source be consulted.

4.1.4 Data Acquisition

Each signal has its own thread which waits for the signal to arrive. Once the signal arrives, the capture_data function is called and this starts the data acquisition process. Inside the capture_data function, the process begins by loading the correct sequence. This is accomplished by setting the current_sequence pointer to point to the sequence associated with the received signal number. Next, the function enters a for-loop that steps through the exposure sequence using the Sequence.getNextExposure() method and performing Sequence.getExposureCount() iterations.

Data Acquisition inside the for-loop involves several steps. First the shutter must be opened and closed, remaining open for the desired exposure length. This is accomplished with the takeExposure() function. The actual signals to the shutter are generated by a DIOBoard object, and the timing is achieved with the gettimeofday() system call.

Next, a DmaDaq object is told to expect data and the ReadOutElectronics object is told to read out the data. At this point the ROE should be streaming data to the FPGA, which will in turn be using Direct Memory Access(DMA) to place the data into main memory. Because DMA does not require the use of the processor, the DmaDaq object is able to simultaneously sort the previous data as new data is coming in. The data needs to be sorted because the data from all four channels is interlaced as it comes down the stream. Thus it is necessary to place all of the channel zero data into its own buffer and so on. After the call to DmaDaq.finishAndSort() has completed, the program should be left with four buffers full of data.

The final step is to write the acquired data to disk. This is easily accomplished with a call to write_data(). Calling write_data() requires the program to send in a pointer to an array of four buffers with data, a pointer to an array of four ints representing the sizes, in pixels, of the four channels, and a Data_Frame_Info structure that contains relevant information about the exposure and sequence. Inside the write_data() function, an instance of the RoeImage class is created to automate the both the formatting and writing of the data. This function also adds the new filename to the telemetry queue so that it can be transmitted if time allows. Both the DmaDaq and RoeImage classes are described in future sections.

This process occurs for every exposure. After the last exposure has been taken, the thread exits the capture_data function and once again begins listening for the appropriate signal to arrive.

4.2 Helper Classes

A number of non-standard helper classes are utilized to make the MDAQ program easier to write, easier to understand, and easier to maintain. Each of these classes are described in the following subsections.

4.2.1 InitAttr

The InitAttr class provides a convenient way to extract attribute information from a file. Specifically, it extracts space delimited key-value pairs, where the key always appears first. As the file is parsed in the constructor, the pairs are stored in an associative array, so that the key can be easily used to fetch the value.

If the type of data is for a key is already known, the InitAttr class also has methods that automatically cast and convert the ascii value to the desired type. The following is an example of what a file might look like followed by a typical program that might use the InitAttr class with this file.

```
**********************************
  #attr.conf

  val1 10
  val2 2.5

  printVals true

**********************************
 //initReader.cpp
 #include"initattr.h"
 ...
 InitAttr attr("attr.conf");

 int v1 = attr.getIntParam("val1");
 double v2 = attr.getDoubleParam("val2");

 if(attr.getBoolParam("printVals") == true)
     out << "v1 * v2 =" << (double(v1) * v2) <<endl;

**********************************
```

Output:

```
  v1 * v2 = 25.0

**********************************
```

4.2.2 Sequence

The Sequence class provides an object oriented representation of an exposure sequence. Sequences can either be created by passing exposure information to the constructor, or the sequence can be parsed from a file by passing a file name to the constructor. Sequence files end in .asq, which stands for ascii sequence. The structure of an asq file is relatively simple and an example will appear at the end of this section. For now, it will suffice to know that the file contains the name of the sequence, the exposure count, and a space delimited list of exposure durations.

A Sequence object can keep track of the current position in an exposure sequence while the sequence is being executed. It also contains many methods for manipulating the sequence in different ways. The following is a list of possible sequence operations:

Get the sequence name

Get the file name

Get the exposure count

Get an exposure at a specified index

Get the current exposure

Get the next exposure

Find out if there are more exposures left

Get the current index

Find and replace one exposure duration with another

Scale all exposures by a specified ratio

Translate all exposures by a specified delta

Jump to a specified index

Find an exposure and jump to it

Save the sequence and any changes made to a specified file

Thus, the Sequence class makes working sequences very convenient. It is suggested that the source for the sequence class be consulted to determine specific method names and argument types. The following is an example of a sequence file:

```
SEQUENCE:
        NAME: datadefault
        COUNT: 5
        BEGIN:
                .5 1.0 2.0 1.0 .5
        END:
```

4.2.3 ReadOutElectronics

The ReadOutElectronics class is meant to provide an easy and reliable way of communicating with the ROE. Upon creation, it attempts to establish a connection with the ROE over a serial port that must be pointed to by /dev/roe. The ReadOutElectronics class takes care of configuring the serial port to ensure correct communication.

To begin using the ROE directly after it has been powered up, it is necessary to use the ReadOutElectronics.exitDefault() method which takes the ROE out of its default mode, that being one which is not commandable and reads out an image every eight seconds, and puts it into command mode.

In command mode, there are a number of different options that can be selected to alter the way the ROE behaves. This may come as a surprise, considering that the ROE's primary and virtually only job is to read out the data on the CCD cameras, but there are a number of different ways that this data can be read out. For instance, if the ROE is placed in selftest mode through the use of the selftestMode() method, the ROE will not read out the CCD's at all, but will instead read out a predefined sequence of vertical bars. This can be useful for testing the flight computers data acquisition abilities. Once the ROE is in selftest mode, it must be reset via the reset() method in order to exit selftest mode. One would also need to exit default mode again after the reset.

Another useful tool is the ROE's Stims mode. This is another diagnostic tool for checking the correct functioning of the ROE. In this mode, a predefined pattern is generated in much the same way as the selftest pattern. The difference is that in stims mode the pattern is fed through the CCD readout circuits and thus any extraneous noise and other anomalies show up in the pattern. Also, unlike selftest mode, stims mode can be exited by using the stimOff() method.

Yet another tool that is provided by the ReadOutElectronics class, is the ability to easily query the ROE for house-keeping data. The ROE has many voltages, currents, and temperatures that each have a unique H/K id number. When this id number is passed to the getHK() method, the raw value for the H/K parameter is returned. The types and id's for the ROE's H/K parameters are defined in the roehk.h file.

This finally brings us to the bread and butter of the ROE, the flush() and readOut() methods. Flush is used to clear any accumulated data on the cameras and readOut is used to get the data from the cameras. The biggest difference between the two, is that flushing the ROE is faster and doesn't send any data down the data link. It is recommended that the ROE be flushed five times in succession before every exposure. Readout does just what one would expect, it reads out the cameras. It is suggested that the ROE be allowed four seconds for reading out data between exposures. The only notable point of confusion for this method is that it also requires a block id, which is basically just a byte that defines how the ROE needs to be read out. There are unique id's for reading out the ROE normally, reading out while in selftest mode, reading out while in stims mode, and there are even some undocumented id's which were used for testing during the construction of the ROE. All of these block id's are documented in the roe.h file.

As a last point of interest, there are also methods for getting and setting the analogue electronics parameters inside the ROE. These methods normally go unused and are usually only useful during testing. These parameters consist of 8 bytes which control the ROE's behavior. In fact, it is by setting these values that the ROE is placed into selftest and stims mode. In those methods these parameters are written automatically. It is suggested that these parameters only be changed if an experienced ROE operator / technician knows what they are doing.

4.2.4 DmaDaq

The DmaDaq class is mostly just an object oriented wrapper around the FPGA API defined in pcifio.h. As such, the pcifio api provides in-depth coverage on most of the functions called in this class. The methods without procedural equivalents include the constructor, the finishAndSort() method, and the flush() method. These methods will be covered here.

Most of the constructors activities involve initializing the pcifio board and fifo. After this initialization has completed, the constructor goes on to setup proper DMA operation. When the computer is booted up, the boot loader sets away a portion of memory towards the top for DMA. The constructor retrieves both a pointer to this memory, as well as a logical address that is passed to the DMA functions. The constructor takes parameters specifying the buffer count and the buffer size. Basically, these values are used to partition the DMA buffer into several smaller sections. These smaller sections are then used in a see-saw manner. While one buffer is being filled up with data another is being readout. In the moses software, the number of buffers is set to 4. The size is set to be one fourth of the data coming in one exposure. The reason for this is slightly complicated. In the pcifio api, there is a maximum amount of data that can be read with one call to the card. This maximum value just happens to be equal to one fourth of our data. Therefore, the function must be called four times. We use four buffers, so that every call to the card can have its own buffer and when the calls are finished the data lines up in one contiguous block.

As previously mentioned, the read function must be called four times. Because the FPGA card needs to be listening for data before the ROE starts reading out, one of these calls is made, the ROE is commanded to readout, and then the other three calls must be made. These other three calls have been combined into the finishAndSort() method. The nice thing about this method is that it also sorts the data while new data is being acquired. As previously mentioned in this document, the data from the ROE comes from four different channels and is interleaved. The sorting that takes place in this method untangles these channels and stores the data from each channel into and array of four pixel buffers which is passed as a parameter. The method also updates and array of four index which is also passed as a parameter. When this method finishes, these four indexes can be used to tell the sizes in pixels of the four data buffers.

The last method that needs to be covered is also the simplest. The flush() method simply reads and discards any extraneous data that is left in the physical fifo after previous read methods have read out the real data from the ROE. The information left in the fifo is often the byproduct of certain workarounds that had to used in the FPGA design.

4.2.5 RoeImage

Images from the ROE are written to disk through the use of the RoeImage class. At first glance this class might seem unnecessary because the data from the ROE is saved in such a simple format. How Simple? Well to be honest, The first channel is written if it is present, followed by the second channel, third, and fourth. This format was chosen because software already existed for reading such a file into an image / data analysis software suite. The reason that we can't just write the data to disk and be done with it is that there is a lot of important information about an image that shouldn't be lost. This is where the RoeImage class comes in. It implements an effective and efficient way of storing this information.

Here is how it works. After the data portion of the image has been set, a number of other methods are used to set the relevant information. This information includes the name of the image(which is usually something like darktest1), setting the number of bits per pixel, setting the width and the height, setting the exposure duration, setting the number of channels included, setting the date and time, and setting the origin, instrument, observer, and object for the image. It now becomes obvious that this is a lot of useful information. To store this information it is important that the image have a unique filename. This is easily achieved by using the date and time. The reason that the name needs to be unique is that the RoeImage class saves all of the image information in a common XML file. The file name is used as the index into this file. This XML file can then be parsed by different software later or it can even be parsed by the RoeImage class to read an image and its information from disk.

5 Input Output(IO) Server

The IOServer acts as the software's interface to the outside world. While it is true that the software has been designed to run autonomously, a need was stated early in the project to allow the observation and alteration of virtually every aspect of the software during pre and mid experiment. Physically, the communication between the ground support hardware and the flight computer is achieved through a set of telemetry links, one uplink and one downlink. The uplink operates at 1200 baud and the downlink operates at 9600 baud. Requests and responses are transmitted in integrity validated packets. Once the input packets are read in through a serial port into the IOServer, they are deciphered and then the IOServer is responsible for querying and commanding the other programs. There are also several tasks that the IOServer is solely responsible for. These include providing a virtual shell that functions similar to a standard unix shell, and providing streaming house-keeping information from the computer. The entirety of the IOServers responsibilities is implemented using four threads. Each of these threads will be explained in the following section.

5.1 Internal Structure

The IOServer program is divided into four threads. These include the main thread, which initializes the variables and threads and also responds to input packets, the shell output thread, which listens for output from the virtual shell and sends it down the downlink as it arrives, the vtmudeck input thread, which listens for message from the vtmudeck indicating the arrival of different timers, uplinks, and signals, and finally the streaming house-keeping thread which sends streaming data about onboard voltages and temperatures down the downlink. Each of these threads will be discussed in detail in the following sections.

5.1.1 Main Thread

The main thread contains the bulk of the IOServer code. This is because it not only handles the initialization of the variables and threads, but it is also responsible for reading in packets and then responding the requests therein. This culminates in a large loop that contains an equally large number of If statements.

Initialization of the majority of the variables and threads should be self explanatory when looking at the code. The important bits include the MIOPipe and PowerPipe as well as the VSHell objects. The MIOPipe and PowerPipe do what one would expect, they provide a convenient interface for communicating with both the MDAQ program and the PowerServer program. It is recommended that the source for these classes be consulted to understand how they function. The VShell object is a virtual shell that allows commands to be executed as if they were typed in a unix shell. This class will be described in detail later. The important piece to understand now is the meaning of the parameter passed to the constructor, in this case "/dev/SOUTPIPE". This string specifies the destination to which the shell's output will be sent. This program uses /dev/SOUTPIPE which is a fifo. By doing this, the fifo can be read later and the output can be extracted.

Next, the main thread enters the main loop and it waits to receive the first packet. Packets encapsulate requests and responses and are read and written with Input and Output Packetstreams. Each of these classes is described in detail later. The packets read by the flight computer can have one of four types. These types include SHELL which contains a command to be executed in the virtual shell, MDAQ_RQS which is a request for some action by the MDAQ program, PWR which is either a request to turn on or off a powersystem or is a query to request the status of a powersystem, or finally it can be of type HK_RQS which is a request for a particular house-keeping value. Each of these types in turn contain a number of possible subtypes. These subtypes are to numerous to list here, so it is recommended that the source as well as **H** ouse Keeping **L** ink **P** rotocol(HLP) documentation be consulted to find out about specific options and their use.

The main program also checks for the validity of the packets. It is possible that the packets could become corrupted during transmission. Therefore, parity checking ensures that the integrity of a packet is known. Upon receipt of good packets, Good Acknowledgements are sent. Likewise, upon receipt of bad packets, Bad Acknowledgements are sent.

When the main thread exits its main loop, it waits for the other threads to end before ending the program.

5.1.2 Shell Output Thread

The shell output thread is fairly simple. Just as the name implies, this thread waits for output from the virtual shell to become available and then sends it in shell packets down the downlink. In this program the output from the shell is sent to /dev/SOUTPIPE, so this thread opens /dev/SOUTPIPE and continuously reads it. Because packet data can have a maximum length of 255 bytes, it is possible for a shell command to produce more data than can fit into one packet. In this case, multiple shell output packets are sent down the downlink.

5.1.3 VTMUDeck Input Thread

The VTMUDeck Input Thread listens for input from the VTMUDeck indicating the arrival of a Timer, Uplink, or Shutter signal. Communication with the VTMUDeck is accomplished with an IOSPipe object. This class is similar to the MIOPipe and PowerPipe classes, and as such, the source should be consulted to see how these classes are used.

After a message has been read indicating that a Timer, Uplink, or Shutter signal has been received, the vtmudeck input thread's only required action is to format a message with the relevant information and send it down the downlink to ground support.

5.1.4 Streaming HouseKeeping Thread

The streaming housekeeping thread uses the LMSensors Project's sensors library to read voltage and temperature information from the motherboard. Using their library involves opening a config file and using it to initialize their library. After this, a sensors_chip_name structure can be passed to the sensors_get_feature() function to extract information from the appropriate chip. On the Hercules EBX motherboard, this chip is the via686a. Relevant information includes a 2.0V reading, 2.5V reading, 3.3V reading, 5V reading, 12V reading, and several temperature readings.

These values are extracted once every two seconds and sent down the downlink.

5.2 Helper Classes

Several Helper classes are utilized to give the IOServer its full functionality. These classes include the Packet class, InputPacketStream class, OutputPacketStream class, and the VShell class. Each of these classes will be described in detail in the following sections.

5.2.1 Packet

The Packet class is used to encapsulate the transmission packet. Data, which is almost always in the form or requests or responses, is sent between the flight computer and ground support in discrete packets. Each packet contains a type field, subtype field, data length field, data field, timestamp field, and parity byte. These fields are capped with start and stop characters, parity encoded, and sent as a byte stream across the up and down links. With the Packet class, each of these processes is taken care of automatically. A packet can be created by passing it the necessary information and then sent down an output stream using the sendPacket() method. On the other end of the physical link, the packet can be read from an input stream using the readPacket() method. This method also automatically calculates the packets parity byte and then checks this against the parity byte transmitted to check for packet integrity. It is recommended that the source be consulted to find the exact syntax of the other accessor and mutator methods. It is also recommended that the HLP documentation be consulted to learn about the specifics of the HLP Packet specification.

5.2.2 InputPacketStream & OutputPacketStream

The InputPacketStream and OutputPacketStream classes are merely wrapper classes that are designed to increase the ease of using packets. After the object has been created using the correct stream object, packets can be read and written by simply calling readPacket() or writePacket().

5.2.3 **V**irtual(V) Shell

The VShell class is used to provide access to a unix shell. In this case, the shell is actually a slightly restricted bash shell. The VShell class achieves this by creating a new bash process. This process has its standard input, standard output, and standard error streams redirected in such a way that it can be controlled by this shell.

A VShell object is created by passing a device or file where the output should be sent. This is usually a pipe that can be read from a program. After the object has been created, commands are executing by passing them to the execute() method. For example vshell.execute(”ls”) would list all of the files in the current directory. When the VShell is finished being used, the shell should be closed by using the exit() method.

As previously mentioned, this is a somewhat restricted bash shell. Although it should behave very similarly to a regular bash shell, because it is one, the nature of the shell's creation dictates job control may be unavailable. It is recommended that the software be thoroughly tested with a wide range of scenarios to determine what is possible and what isn't.

6 Appendix A - Digital I/O Pinouts

DIO Bank Pinouts					
Bank	*Pin*	*Purpose*	*Used By*	*Input*	*Output*
A	0	Data Start Timer	VTMUDeck	X	
	1	Data Stop Timer			
	2	Dark 2 Timer			
	3	Dark 4 Timer			
	4	Sleep Timer			
	5	Data Start Uplink			
	6	Data Stop Uplink			
	7	Dark 1 Uplink			
B	0	Dark 2 Uplink	VTMUDeck	X	
	1	Dark 3 Uplink			
	2	Dark 4 Uplink			
	3	Sleep Uplink			
	4	Wake Uplink			
	5	Test Uplink			
	6	Shutter Signal			
	7	Unassigned	N/A		
C	0	Dynamically Assigned In powersys.conf	PowerServer		X
	1				
	2				
	3				
	4				
	5				
	6				
	7				
D	0	Dynamically Assigned In powersys.conf	PowerServer		X
	1				
	2				
	3				
	4				
	5				
	6				
	7				

DIO Bank Pinouts					
Bank	*Pin*	*Purpose*	*Used By*	*Input*	*Output*
E	0	Unassigned	N/A		X
	1				
	2				
	3				
	4				
	5	Clear Virtual Fifo	MDAQ		
	6	Open Shutter			
	7	Close Shutter			

7 Appendix B - Notes on Software Usage

Compiling this software should be as easy as using the Makefiles in the respective directories. However, there are a number of libraries that must be present in order to compile the software. These include libdscud5.a, pcifio_api.a, and libsensors.so. These libraries must also reside in the correct directories. The Makefiles should be consulted to assist in placing these libraries in the correct directories.

After the moses_install script is executed to ensure that all of the correct device files and directories have been created, the software can be started, most likely at startup. It is important to note that there is a sequence in which the programs should be started and tends to produce the best results. In order these go:

```
/moses/bin/vtmudeck &
/moses/bin/powerserver &
/moses/bin/mdaq &
/moses/bin/ioserver &
```

After this, communication with the system can be achieved through the IOServer.