Datalogger Sequence Execution Engine (DSQEE)

Edmund Yingxiang Yee

April 2012

Completed under the supervision of John Bellardo PH.D of the Computer Science Department at California Polytechnic State University, San Luis Obispo for the PolySat Research Group.

Abstract

The PolySat Research Group accepts projects from several organizations that wish to use a CubeSat for some experiment. One of the projects called Intelligent Payload Experiment, or IPEX, needs software to interact with a payload in addition to our system avionics. One process from our system avionics is datalogger, which will be augmented from its original datalogging scheme to support sequentially execution of commands/algorithms that our client, Jet Propulsion Laboratory, or JPL, need. My part of the project explains the software design and implementation decisions behind datalogger.

Acknowledgments

Thanks to Dr. John Bellardo for providing assistance in the new design of datalogger, Dr. Phillip L. Nico and Dr. Kevin O'Gorman for providing the source code for the lex parser, Mike Tran for providing the original source code of datalogger, JPL for providing funding for the entire IPEX project, and finally the PolySat Research Group for providing the environment and resources to make this project possible.

Contents

List of Figures

1 Introduction

Datalogger was originally designed to retrieve the output of a process. Datalogger would read a configuration file containing information regarding database storage, process, and scheduling . The output would be parsed by "key=value" pairs, with each pair separated by a newline. Each of these "key=value" pairs would be mapped to a group, location, and sensor ID which would subsequently store the values into a SQLite database.

1.1 A Brief Description of PolySat

The CubeSat standard was developed in 1999 by Dr. Jordi Puig-Suari, an aerospace engineering professor from California Polytechnic State University, San Luis Obispo, and Dr. Robert Twiggs of Stanford University. A CubeSat was subsequently defined as a 10 cm x 10 cm x 10 cm for a 1U. The size of a CubeSat can be expanded to a length of 20 cm or 30 cm, 2U and 3U respectively. Some of the many benefits CubeSats supply are being cost-effective satellites that are easy to design and build for various space missions. Inside each of these CubeSats would be some experiment with educational or scientific merit. Generally these experiments are designed so they can fit inside a CubeSat but the results can apply to larger and more expensive satellites. A total of 3U's can fit inside a Poly Pico Orbital Deployer (PPOD), which attaches to a launch vehicle and deploys the CubeSats into outer space to begin their mission when it is deemed necessary to do so.

1.2 Previous System Avionics

The previous system uses a custom embedded operating system to run their software. The skills needed to develop on these systems required a lot of experience in embedded development. Therefore, a push to move to a custom Unix operating system, customized with a tool called BusyBox, was made to make development faster and coding easier. The programming skills required by PolySat programmers was now targeted at sophomores and juniors rather than soon to be graduating seniors. By abstracting away all the embedded knowledge required and using a familiar environment, the time needed to bring up new recruits was faster and easier. All the newer satellites developed by PolySat are planned to use this new system for future missions, which include IPEX.

1.3 An Introduction to IPEX

Intelligent Payload Experiment, or IPEX, is an experiment funded by Jet Propulsion Laboratory (JPL). The IPEX CubeSat payload (Space Cube Mini) includes a Field-Programmable Gate Array (FPGA) processor that, due to power budget limitations, will be on for a limited amount of time. It will be processing images from the 4 cell phone cameras attached to the CubeSat. Images taken can also be processed on PolySat's system board for flexibility and redundancy reasons. The actual hardware for the Space Cube Mini is being developed by NASA-Goddard and will be given to PolySat for integration when it is complete. JPL will be providing the software to determine when a camera takes a picture and which pictures it wants to process. JPL's Continuous Activity Scheduling Planning Execution and Replanning (CASPER) will be modified to work for the IPEX mission. It will determine which image processing algorithms (provided by JPL) it wants to run on either the Space Cube Mini or PolySat's system board. The general purpose of this experiment is for JPL to test on board processing of images while in flight and sending down only relevant images of the Earth. The secondary purpose is to test their hardware and see the effects space has on it.

1.4 PolySat's Role in IPEX

JPL requires methods to communicate with PolySat's system avionics to be able to achieve their mission. Therefore, some software may need to be modified to accomplish certain mission specific objectives. One of JPL's objectives was to schedule and run the commands in their script files sequentially and know the status of each script while it is running. Due datalogger's close resemblance to this objective, it was chosen to be augmented to support the functionality needed by JPL.

1.5 Datalogger Sequence Execution Engine Introduction

This new feature set would be aptly named Datalogger Sequence Execution Engine (DSQEE), pronounced desk-ee, and would support accepting commands from JPL's CASPER. Basically DSQEE will act as the support for JPL's software to interact with PolySat's scheduling software. It will execute the commands CASPER gives it sequentially after a certain time specified by the scheduled event.

2 Datalogger Sequence Execution Engine Requirements

As the client of PolySat, JPL had several requirements for the execution engine they needed for CASPER.

Functional Requirements	Justification
Support the ability to execute commands sequentially.	The script files commands expect the previous command to have succeeded before running the next one.
Able to support remote connection to the payload Space Cube Mini.	CASPER needs to be able to control the Space Cube Mini.

Be able to accept and support UDP connections.	All commands sent to datalogger will be UDP packets with a command byte detailing what function should be executed.
Be able to parse script files and be able to recognize comments.	The script files contains commands separated by newlines and comments. Datalogger must be able to distinguish between all of these and perform the appropriate action according to each line.
Be able to support redirection symbols '<' and '>'.	DSQEE's purpose is to emulate the command shell when it runs commands. Therefore it must support things like redirection.
Be able to support piping '|'.	DSQEE's purpose is to emulate the command shell when it runs commands. Therefore it must support things like pipes.
Be able to change the parameters in a script file on demand.	The commands need to be flexible enough to support a variety of different values for the parameters.
Be able to know the current progress of a sequence and see if it is still going, completed, or failed.	If an error occurs or the finishes, this information is needed to know how to deal with these situations.
Be able to handle errors like missing script file, missing executable, and invalid configuration for starting a sequence.	Datalogger should be able to handle errors elegantly and simply report the error and handle the error in an appropriate way.
Be generic enough to be usable on future PolySat missions and still fit the needs of IPEX.	Future PolySat missions may require a sequential execution engine and making it generic solves this problem.
Be able to have commands depend on the previous command's output/success.	The goal of this new feature is to sequentially run commands. Otherwise datalogger's current features already support this.

3 Original Datalogger Design

The original datalogger was developed by Mike Tran for the PolySat group. It's purpose was to execute processes, read their output, and stored the key-value pairs the process outputs into a SQLite database. It parsed two configuration files: a process/event and database configuration file to schedule the event and construct the entries in the database respectively. It then waits for the next event to occur in the event queue. After an event occurs, datalogger forks off the process and reads the output of the process. It expects output formatted in ”key=value” pairs with each entry separated by new lines. It parses this output and stores this information into the database.

The code was later restructured to be more flexible to include some additional features. The configuration file was changed so the database and process/event configuration file would be in one file. Each configuration file is expected to have at least one event or subprocess entry. An event entry corresponds to a subprocess and has information like start time, scheduled time, and whether or not it is reoccurring. A subprocess entry has a name, process path, and parameters field with each parameter entry separated by a white space. Each subprocess can have an optional one or more sensor subfield that includes the sensor key, name, location, and groups field for storing the subprocess output into the database.

Some new features added to datalogger was to make it more dynamic in adding/removing events. To remove or add an event while datalogger is running, datalogger accepts an UDP packet with information to either add/remove an event. For adding an event, the UDP packet contains the location for the configuration file. For removing an event, the UDP packet contains a list of event names separated by white space. To make the code more robust, a few implementation methods in parsing and setting up the events were changed. The overall purpose to execute processes and store its output inside the SQLite database remained the same.

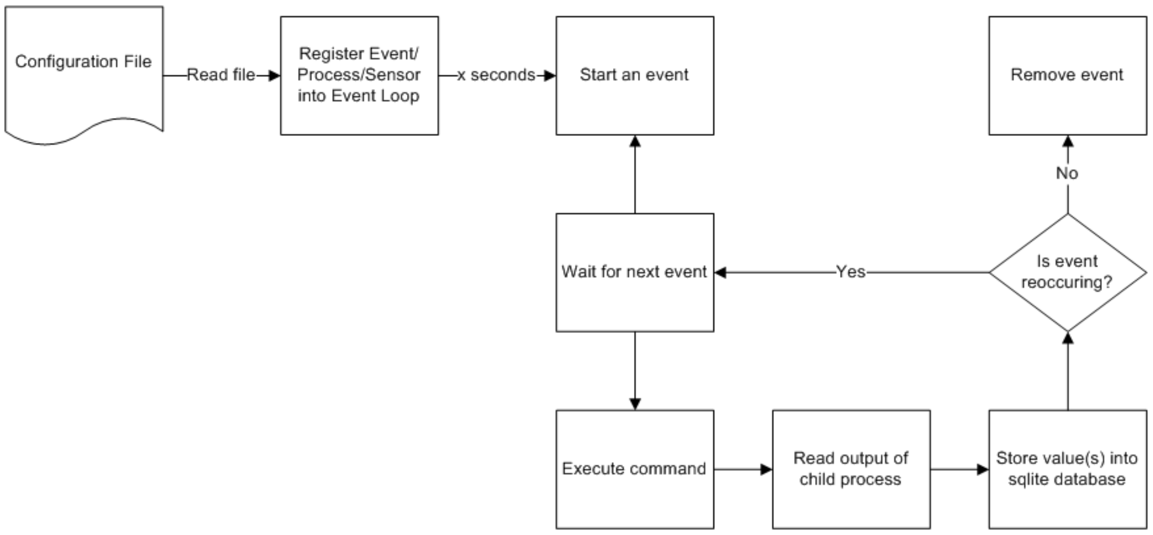

Figure 1: High level flow diagram depicting original datalogger functionality.

4 Design of Datalogger Sequence Execution Engine

For reference, this paper will call each command line argument in the script file a command. The command can contain the process it is executing, redirection symbols, or the piping symbols. A set of commands will be called a sequence.

To meet the needs of JPL, a process was needed to read a script file and execute the commands inside it sequentially. The process would need to be able to communicate with the Space Cube Mini payload system and remotely control it. It would need to be able to interface with JPL's CASPER and work with PolySat's current system avionics. The process chosen was datalogger because it already forks processes and already had PolySat's system avionics code integrated into it. It was also decided that this new functionality should be generic enough to be used on future PolySat missions as well as serve the needs of JPL.

4.1 An Overview of What Datalogger Sequence Execution Engine Will Do

The command will be in the form of an UDP packet that contains all the relevant information like unique ID, time to execute, continue, script file location, and key-value pairs for string replacement in the script file. DSQEE parses each of these fields and stores them appropriately. It then waits a certain amount of seconds for the event to trigger to actually being executing the sequence. After the event occurs, it parses the script file for commands and runs the string replacement algorithm on them. It proceeds to execute each command sequentially. It uses the unique ID as the name of the file to store the current progress of the execution engine as well as the exit status of each command. Upon failure of any of the commands, the continue byte would be queries to determine if the sequence should continue executing or stop and write the exit progress as an error.

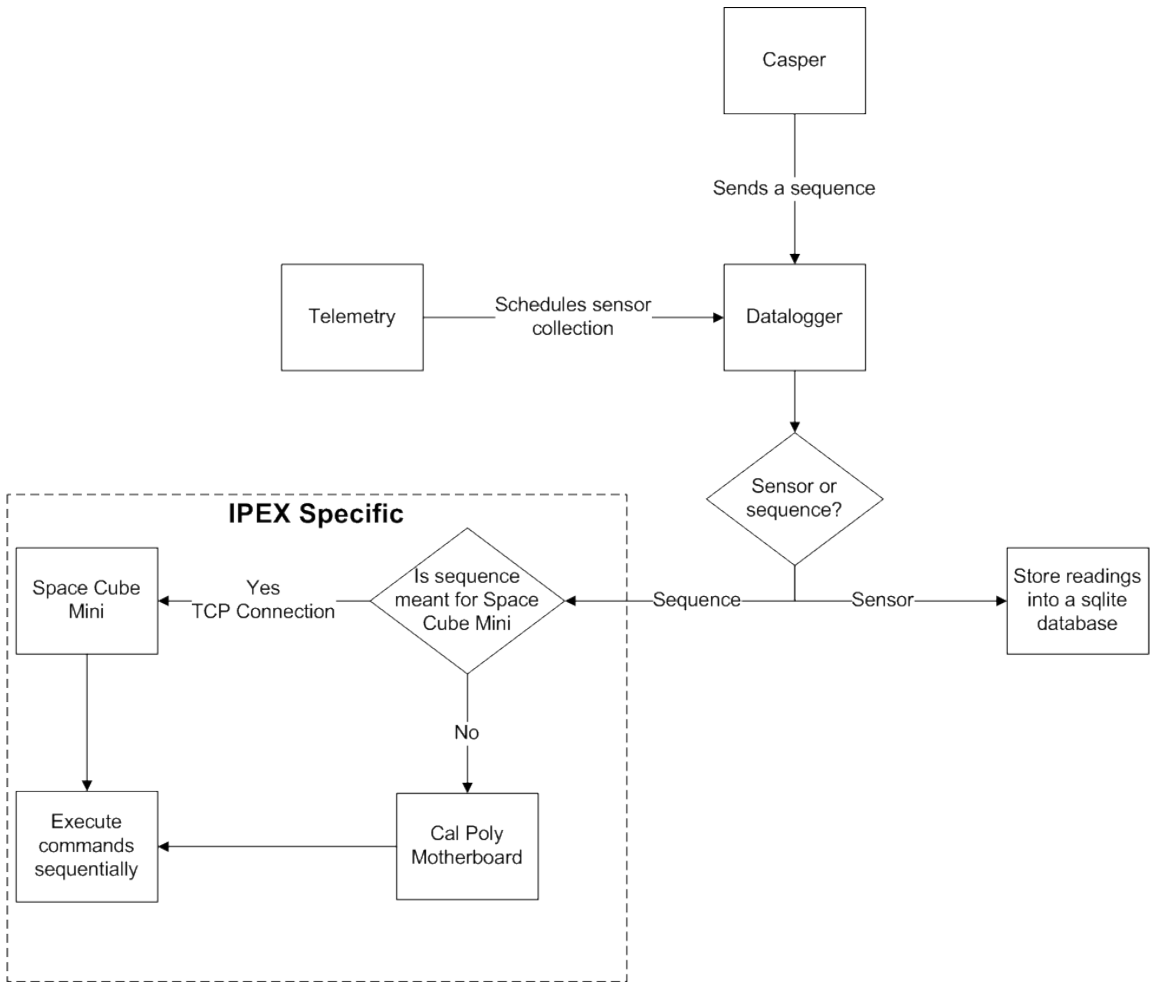

Figure 2: High level flow diagram depicting IPEX datalogger functionality.

4.2 Accepting a Sequence UDP packet

Datalogger inherently uses PolySat's process library base for UDP connections. The process library calls the function datalogger set according to the command byte it received, which is the first byte in the data portion of the packet. Datalogger accepts two different types of UDP packets regarding sequences, adding and removing a sequence. Every string field is required to be null terminated for datalogger to correctly parse it. If a string is not null terminated, datalogger will only read up to the length of the packet passed in and may not function correctly.

4.2.1 Adding a Sequence

When a process decides to add a sequence to run, datalogger accepts an UDP packet containing:

- Command byte
  The command byte is the 0x07 header byte that PolySat's system avionics code requires at the beginning of every UDP packet. It is used to distinguish which function to call when the packet arrives. This part is only used by the process library and is actually removed when datalogger reads the packet.

- Time stamp
  The time stamp is when the event needs to be run. It is the time in seconds from the epoch time. If the time is below the current time, the sequence will run right away. Otherwise datalogger will calculate how many seconds there is left to run from the current time and wait.

- Continue Byte
  The original requirements from JPL required To make datalogger flexible, this continue byte gives the option of continuing running the commands in a sequence if one of the previous commands failed.

- Unique ID
  The unique ID is a string that should not be replicated. It is used to identify each sequence datalogger currently in its event queue and the output file name. Datalogger uses the unique ID to remove a sequence that has not run yet from its event queue. Overlapping unique ID names can also result in the output file for a previous sequence to be overwritten.

- Path to the script file
  This path to script file indicates where on the system the script file is located. Datalogger verifies this path exists before accepting it.

- A list of optional key value pairs
  This field is entirely optional and is included to make the script files more flexible. The list of key value pairs will be used for string replacement after the script file is parsed. This allows a smaller set of script files to be used and makes it easier to use the them for different scenarios.

After receiving this packet, datalogger will send an ACK packet back to the sender to confirm it got the packet. It contains the type of action (register/unregister), the result code, and the unique ID it registered.

4.2.2 Removing a Sequence

If a sequence is not suppose to run anymore, a process can send datalogger a UDP packet containing:

- Command Byte
  The command byte is the 0x09 header byte.
- A list of unique ID's to remove
  This is a list of unique ID's that have not run yet. Datalogger looks each of these unique ID's that were passed in and only removes the ones that have not run yet. Sequences that have never been registered or have already run are ignored.

After receiving this packet, datalogger will send an ACK packet back to the sender to confirm it got the packet. It contains the type of action (register/unregister), the result code, and the unique IDs it removed.

4.3 Parsing a Script file

For the IPEX mission, JPL will be providing these script files containing command to run on their images. These scripts contains commands separated by newlines with comments represented as '#'. Datalogger would need some way to take the text inside and organize it in a way such that datalogger can distinguish what's the executable, the arguments, input/output file name, and etc.

4.3.1 Type of Parser Used

One of the past projects students had to solve was a program called MUSH (Minimally Useful Shell). This program mimics the shell and has some functionality similar to datalogger. MUSH takes in a string and executes it as it would on the shell. Professor Nico and Professor O'Gorman, the professors that teach Systems Programming, gave their students the actual code to parse the command line and returned everything their students would need to execute the string as a command line. The code was already implemented and used by many students, so it was fairly well tested. The code did need some tweaking in the regular expressions to decide which text was important, but other than that the code remained the same.

In essence, it is a text parser called lexer (lexical analyzers). It purpose is to to separate the command line into values datalogger could easily execute and understand. The lexer code takes in a string and separates them using regular expressions (regex). A flex (fast lexical analyzer generator) compiler is used to actually generate the lexer (lexical analyzers) into C source code that can be integrated with datalogger. A bison parser file is used to take the values the lexer outputs and creates the structure that separates all the command line information. A bison parser generator is used with the bison parser file to generate the C source code.

4.3.2 How the Parser Works

Per their terminology, a stage is a single executable and multiple stages occur when pipes are present. Their code interacts with the lexer code to generate a structure called a pipeline that contains the original command line, the amount of stages, and the first stage. A command can contain multiple stages, but each stage contains information regarding the input and output name (redirection symbols '<' or '>'), the number of arguments, and an array of arguments themselves. The parser will ignore comment blocks which begin when a '#' is detected. To generate this structure for each line, datalogger reads each line in the file and applies the parser code to it.

4.3.3 Changing Parameters On Demand

To use a small set of script files for multiple different scenarios, JPL wanted some way to dynamically change the parameter values in their script files. To support this, datalogger performs a string replacement on the parameters. One of the fields in the UDP packet is an optional list of key-value pairs. If an uneven amount of pairs was found, datalogger reports a parsing error and disregards the last key-value pair. These key-value pairs will be used in the string replacement algorithm to handle parameter changes in the script file. These key-value pairs are parsed and stored inside a hash table. When datalogger creates the pipeline structure for each command, datalogger checks the arguments of every single command and looks for arguments preceded by a '$'. If datalogger finds one, it removes the leading '$' character and looks up the argument in the key-value hash table. If it finds a match, it replaces the whole argument with the value found in the hash table and continues. If no match is found, it is ignored.

4.3.4 Continuing or Stopping Per Command

Another feature to make commands more flexible was having a per command continue byte. This would override the global continue byte received in the original UDP packet to continue or stop on failure. The key word datalogger watches for are the '-' and the '+' signs in front of the commands in the script file. During the parsing, every command will be checked to see whether a '+' or '-' appears in front of it. If it does, datalogger stores this and will use it in the callback when the command finishes. If the command ends up failing, datalogger first checks the continue byte for per command and sees if it's been set.

If it's been set, datalogger follows whatever value it is to continue or stop. If it hasn't been set, datalogger defaults to whatever the global continue byte is.

4.4 Knowing the Progress of a Sequence

A useful feature that CASPER wanted was to know the current progress of a sequence that was running. To accommodate this, datalogger creates an output file according to the unique ID field that was passed in from the UDP packet. The file created has read/write permissions for user, group, and other. If the file doesn't exist, datalogger creates it. If it existed previously, datalogger clears the information in the file (why it is important that the unique ID is not used again). The contents of the output file are 4 bytes for the error status and exit status. The upper 2 bytes are the exit status. The exit status is an unsigned integer value that indicates whether the sequence is complete, currently running, complete with error, running with error, or incomplete with error. Possible values range from 0-65535, though only 0-4 are currently used. Datalogger will keep the sequence in one of these known states by checking this progress status every time a command and updating it according to the current progress status.

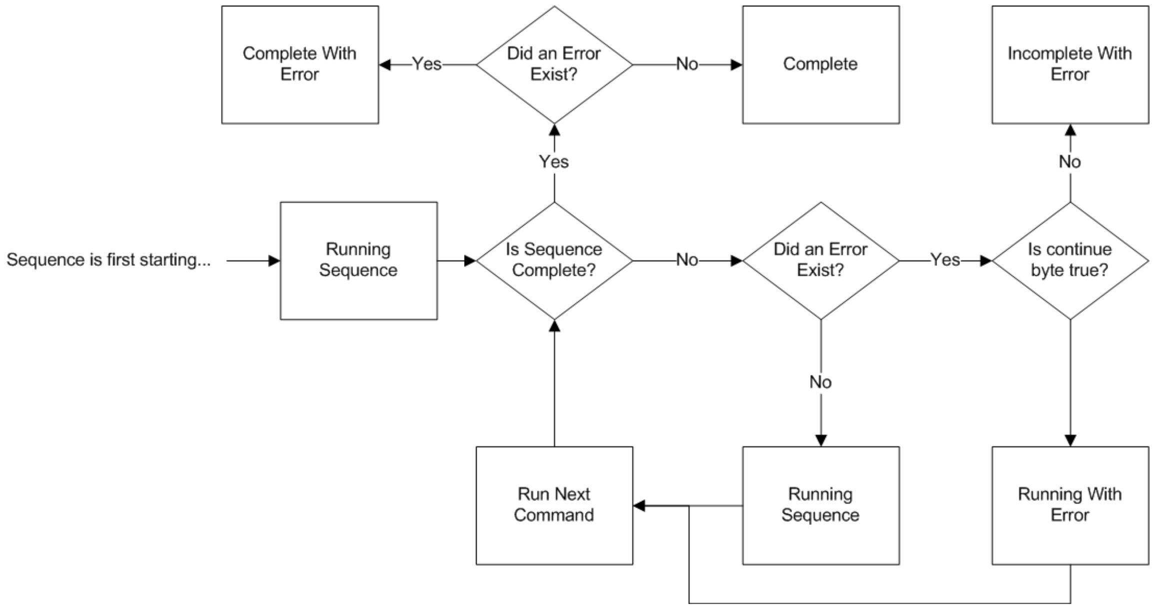

Figure 3: State Diagram showing possible sequence states.

The lower 2 bytes are the error status. The error status is a bit field indicating the type of error that occurred. Currently only 3 bits are used to indicate errors but there is possibility for 13 more error messages.

Error	Bit field
No error	0x0000
Parsing error occurred in script file	0x0001
Uneven key-value pairs in packet	0x0002

Figure 4: Table showing the current error statuses supported by datalogger.

4.5 Executing Commands Sequentially

One of the main requirements was not to start a command until the previous command had finished executing, in other words sequential execution. To achieve this, datalogger takes every single command it parsed in and creates a node in the linked list for that command. Nodes are continually added until all the commands are in the linked list. The last node added will be a node that contains the sequence's data, the length, and the file descriptor that will be associated with the output file. This is needed so datalogger can reference values like the continue or the file descriptor on later executions. Once the list has been set up, datalogger waits for the event to actually occur before any actual parsing is done. An ACK packet is sent back to the caller to verify it got the sequence to create.

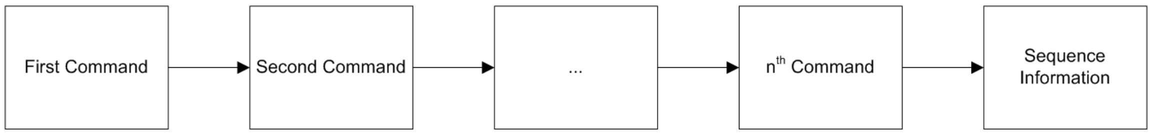

Figure 5: Diagram showing the contents of the linked list.

When the event finally occurs, datalogger removes the first node and tries to fork it. If an error occurs anywhere (fork fails, process that was forked fails, and etc.), datalogger checks the continue byte to see if it should keep trying with the failed sequence. If the continue byte is false, it cleans up the list and writes an error message to the output file. If the continue byte is true, datalogger writes the exit status as a failure and moves on to the next command. This is accomplished by using callbacks, functionality which is provided by PolySat's process library. The Datalogger will the recursively call the execution function until all the nodes are finished executing. The progress of the sequence will also indicate from here on that there was a failure on the current or previous command. The exit status in the output file will reveal which ones failed. Once the length of the linked list reaches one, datalogger will know that the last element should have been the node containing the sequence information. It stops execution and writes to the output file that is is finished. Depending on whether or not there were errors, a different value will be written to indicate completion.

4.6 Remotely Accessing the Space Cube Mini

One of the main concerns was how to remotely control the Space Cube Mini. One of the primary mission goals was to verify the commands that ran on the Space Cube Mini are the same as the ones that ran on PolySat's Atmel processor for redundancy. The two Unix commands remote shell (rsh) and remote synchronization (rsysnc) was considered much more effective and efficient than trying to implement them. These commands have been around for several years (rsh in 1983 and rsync in 1996) and have been thoroughly used and tested. Their reliability can be trusted more than an implementation done in 6 months by someone with little experience in it. Even though they were not by default enabled on PolySat's system kernel, it was relatively simple to enable them and make them available for use.

5 Results and Testing

During the implementation phase, datalogger's new functionality DSQEE was incrementally tested. Test programs were run against it to schedule sequences to verify that it correctly performed the new features. Afterwards, a select amount of script files meant to test specific functionality were created for overall testing. These test scripts ranged from correct/incorrect sequences, feature specific, empty files, and comments. These script files were run thousands of times with different parameters to verify on how datalogger behaved in different application environments.

There were some initial bugs that were discovered through running these tests, but these bugs have so far been fixed. Bugs aside, the test scripts proved that datalogger was robust and would perform to the specifications.

6 Alternative Design Choices

During the course of this senior project, I considered several other designs on how to complete this project. The factors I considered in choosing a solution were efficiency, difficulty in implementation, and code upkeep.

6.1 Sequential Execution

One of the first requirements that came up was to allow for sequential execution. Originally, JPL wanted a design that would allow the commands to depend on each other and perform another action dependent on the outcome of the previous command. To model this, dataflow programming was considered. In dataflow programming, a program is modeled as a directed graph of data flowing between operations. It focuses on how things connect rather than how things happen. This made it ideal to model commands as a state and have different commands execute depending on what happened.

However, JPL then decided a dataflow model was not needed and it was only necessary to try to run an entire script of commands. CASPER was decided it would be intelligent enough to do something meaningful if the sequence failed. Therefore, this feature was no longer needed and a much simpler design of linked lists to keep track of the commands was decided.

6.2 Using system() to Execute

The function system(const char *command) is a C standard library function that takes the command parameter and executes "/bin/sh -c command". This would effectively execute an entire command without datalogger having to do any parsing. It returns the exit status of the child or -1 on error. The functionality is almost exactly what datalogger does, but with one small difference. system() blocks SIGCHLD and ignores SIGINT and SIGQUIT. This is a potential problem because PolySat's process library assumes that it is the only one who can block and watch for these signals. This makes it incompatible with our software and could not be used. Datalogger would be better off doing the forking, piping, and redirection itself to be a non-blocking process.

6.3 Parsing the Script File

One of the first ideas to parse the script file was to manually parse the file in C. However, C does not inherently support strings. This makes it significantly more difficult to do any string manipulation. The code to do it in straight C was deemed to complex and messy, so the lexer was deemed the better choice.

6.4 Changing Parameter On Demand

JPL wanted a convenient way to be able to change commands in the script file. They did not want to have several different script files when only a couple parameters would change. They wanted a convenient and easy way to change the parameters. One of the solutions was to use the system's environment variables to know what value to use. The C standard library function char *getenv(const char *name) searches the environment list and returns a pointer to the value inside the environment variable. The complexity arises because these same script files need to run on the Space Cube Mini. An entirely different system is run on there and the environment variable would have to be somehow modified on there as well to keep things consistent. This increases the complexity more than necessary and was therefore scrapped.

7 Possible Future Changes

Datalogger's current build is fairly stable and robust. It is fairly easy to understand the design, thus making code upkeep much easier. There are a couple changes I would have liked to have done.

7.1 String Replacement Implementation

Some improvements to datalogger could be on how it performs its string replacement algorithm. The current weakness is the inability to support replacement for redirection and piping instructions. Any special characters cannot be used because datalogger currently does the parsing through the lexer before it performs the string replacement. The current design model would have to be tweaked slightly so that the string replacement would be performed on the string that's passed into the parser. The current string replacement function actually expects the pipeline structure to be passed in to perform the replacement.

7.2 On Demand Parameter Replacement

Something a little more flexible than string replacement would have been convenient as well. The string replacement algorithm I proposed was good enough for JPL, but I felt that better and more elegant solutions could've been found to perform the same exact functionality. Given that C is also not well suited at handling strings, a better solution could probably be found with some more research.

7.3 Packet Format

The packet format datalogger accepts/sends for sequences is also difficult to create/parse in C. Given that the strings are all null terminated, the total length that the sender passed has to be trusted. Otherwise, it would be difficult to know when to stop. If the sender for some reason passes in a malformed packet because setting up the packet is trickier, datalogger could potentially fail unintentionally.

8 Conclusion

The feature set added by the new DSQEE can be summarized as sequential execution of script files and storing the results of these sequences into output files by unique ID's. DSQEE's design and implementation phase has been a learning process to further improve my programming capabilities. I was able to work with JPL's programmer to sort out what was required and verify that my solutions to their requirements was sufficient for them.

Feedback from JPL regarding DSQEE have so far been positive. Their initial test runs have seen that DSQEE does what it was specified to do and so far they seem to be satisfied with the current implementation.

A Specification

- Sequences will have the ability to run concurrently.
- A sequence is a list of commands that CASPER wants to run for a particular image. They will run in the order they are provided with each command separated by a newline.
  - A command will not start until the previous one has been completed.
  - If the output of a command needs to be stored somewhere, CASPER can specify its output with the redirection symbols ">".
  - If CASPER wants to use a file as the input for a command, use "<".
  - If CASPER wants to override the global continue byte, any command preceded by a "+" or "-" will ignore the continue byte. The "+" will be used for continue on failure. The "-" will be used for do not continue on failure.
  - The output of each exit command (the return value) will be stored in a file specified by the unique ID name.
    - The return values will be stored in binary format
    - The output will be stored in /data/output.
  - Use remote shell (rsh) login and rsync (both uses TCP connections) to communicate with the Space Cube Mini (scMini). The scMini will be given a static IP address.
    - Use rsh to specify the desire to run a command remotely
    - Use rysnc to copy files between the mainboard and the scMini.
- CASPER will send UDP packets to inform the execution engine of which sequences it wants to run.
  - A sequence will be defined by a file in the file system. Datalogger will open this file and parse it for commands to run.
  - A string replacement will be performed on the arguments. Each special argument in the file should be preceded by a $.
  - Datalogger parses the script file and breaks up the pipes/redirections before it sets up the structure. Therefore it is not possible to change the redirection/pipes without directly editing the script files.
  - A command for any sequence will not be reoccurring; each command only happens once (unless it's in the sequence two or more times).
- When an UDP packet is received by datalogger, CASPER will receive an ACK back regardless of failure (unless datalogger never got the UDP packet).

- Any error that happened in the sequence will be stored in the output file.
- The current working directory for datalogger will be in "/data".

B Expected Datalogger Output

- Datalogger stores the exit status of each command into a binary file. Each exit status will be 4 bytes.
- The name of the output will the specified by the unique ID located in the UDP packet sent to datalogger for creating a sequence.
  - Datalogger will truncate the file in "/data/output/[unique ID name]", so it is important that any sequence ID that ran should not be reused.
  - Datalogger will store all the output files by default in "/data/output".
- The current progress of the sequence will be indicated in the first 4 bytes of the output file. The upper 2 bytes represents the current status of the sequence. The lower 2 bytes represents the error status of the sequence.
  Progress Status
  - 0x0000: completion of the sequence
  - 0x0001: sequence is still running
  - 0x0002: sequence completed but with errors
  - 0x0003: sequence is still running but with errors
  - 0x0004: sequence is incomplete because of errors

  Error Status

  The error status is represented as a bit field. Each bit represents its own error. For example, an error of parsing and uneven key-value pairs would be in binary 0000 0000 0000 0011, or 0x3.

  - 0x0000: no error
  - 0x0001: parsing error occurred for script file
  - 0x0002: an uneven amount of key-value pairs was detected
- Subsequent 4-byte values will represent the exit statuses of the commands.
- To find out what the status of the exit command was, use the macros defined in <sys/wait.h>. Taken from the man pages for wait, some useful ones might be:

- WIFEXITED(status)
  returns true if the child terminated normally, that is, by calling exit(3) or _exit(2), or by returning from main().
- WEXITSTATUS(status)
  returns the exit status of the child. This consists of the least significant 8 bits of the status argument that the child specified in a call to exit(3) or _exit(2) or as the argument for a return statement in main(). This macro should only be employed if WIFEXITED returned true.
- WIFSIGNALED(status)
  returns true if the child process was terminated by a signal.

Format of output file:

Bytes	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	Error Status		Progress Status		First Exit Status				Second Exit Status				...			

C Datalogger Command Format

Datalogger can expect two different types of packets. They are creating a sequence and removing a sequence.

C.1 Creating a Sequence

Datalogger expects the payload in the UDP packet to be in the specified order:

1. A 1-byte command byte. To create a sequence, use 0x07.
2. A 4-byte time stamp. This will the be standard time from the epoch time. Convert this to network byte order before sending.
3. A 1-byte value continue if failed. This byte signifies that if the command failed, should Datalogger continue executing at the n+1 command or give up.
   - Datalogger will use the UNIX convention for exit statuses. Any non-zero value will be considered a failure. A exit status of 0 is success.
4. An unique id (string) specifying the sequence name. This will be used as the name to create the binary output file.
   (a) Will be used if CASPER wishes to terminate a sequence before it runs.
   (b) Must be null terminated.

5. A path to the sequence file. This must be null terminated.

6. (Optional) A list of key-value pairs (strings) specifying the keywords in the script file to be replaced with “value”.

   (a) If a ’$’ was actually intended as a character in the string, use ’$$’ before the string. This does not apply to words where a key-value pair does not exist as no replacement will be made.

   (b) Each item (key and value are considered separate) must be null terminated.

   (c) Partial string replacement will not be supported (i.e. when a parameter is replaced, it cannot be mid string).

   (d) The key word in its entirety will be replaced with the value. The words in the script file preceded by a ’$’ will be replaced if a key-value pair exists. Otherwise no replacement will occur. The packet should not need to include the ’$’ unless it was intentional.

Remember to convert to network order using htons or htonl for relevant fields.
An example payload packet to create a sequence is:

Bytes	0	1	2	3	4	5	5	6
	Source Port		Destination Port		Length		Checksum	
Data portion of packet begins now...								
Bytes	7	8	9	10	11	12		
	Command (0x07)	Time Stamp				Continue		
Bytes	13	14	...	16	17	...		
	Unique ID			Script File Path				
Bytes	18	19	...					
	(Optional) Key-Value Pairs							

* RFC 768 definition for UDP Packets

Note: The length in the UDP header is used to define the total packet size.

C.2 Removing a sequence

Datalogger expects the payload in the UDP packet to be in the specified order:

1. Command byte of 0x09 to signify removing a sequence.

2. One or more unique IDS that are null terminated.

An example payload packet to remove a sequence is:

Bytes	0	1	2	3	4	5	5	6
	Source Port		Destination Port		Length		Checksum	
Data portion of packet begins now...								
Bytes	7	8	9	...				
	Command (0x09)	Unique ID to remove						

* RFC 768 definition for UDP Packets

Note: The length in the UDP header is used to define the total packet size.
Remember to convert to network order using htons or htonl for relevant fields.

D Creating a UDP packet to send to Datalogger

The process library that Datalogger uses the functions to send/receive data:

- ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);
- ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen);

These are standard UNIX functions. More information can be found by reading the man pages.

D.1 Creating a packet to send to Datalogger

Below is a snippet on how to initialize the socket to write to Datalogger. Datalogger uses the port number **50006**. An alternative approach and more correct method is to use:

- struct servent *getservbyname(const char *name, const char *proto);

This is a standard UNIX function and more information can be found by reading the man pages.
Below is the code from the PolySat process library on how we initiate a socket connection.

```
// gets socket by port number
int socket_init(int port)
{
   struct sockaddr_in addr;
   int fd;

   // Make sure we can create the socket file
   //descriptor before configuring
   if ((fd=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1) {
      ERRNO_WARN("Failed to open socket\n");
   } else {
      // Configure socket to be non-blocking
      ERR_WARN(fcntl(fd, F_SETFL, O_NONBLOCK),
            "Failed to configure socket to be non-blocking\n");

      // Set up the socket address structure
      memset((char *) &addr, 0, sizeof(addr));
      // Select IPv4 Socket type
      addr.sin_family = AF_INET;
      // Convert the port from host ("readable") order to network
      addr.sin_port = htons(port);
      // Allow socket to receive messages from any IP address
      addr.sin_addr.s_addr = htonl(INADDR_ANY);

      ERR_WARN(bind(fd, (const struct sockaddr *)&addr, sizeof(addr)),
            "Failed to bind socket on port %d\n", port);
   }

   return fd;
}
```

E Creating/Removing a sequence from Datalogger

The procedure for creating and removing a sequence from Datalogger are similar. The only differences are:

- Creating a sequence requires the command byte to be **0x07**.

- Removing a sequence requires the command byte to be **0x09**.

One thing to note is that Datalogger will not stop a sequence from running if it has already run.

F Sending a packet to Datalogger

This code assumes creating a sequence packet to Datalogger.
Below is slightly modified code taken from PolySat's process library code base where we send data to our processes.

```
int socket_send_data(int fd)
{
   /*
    * Code to generate the sequence information
    * to send to datalogger. Includes information like
    * unique id, path name, etc...
    */

   // Allocate space for the command byte
   char *data = (char*)malloc(dataLen+1);

   // Triggers datalogger's adding sequence code, use 0x09 for
   // removing a sequence
   data[0] = 0x07;
   // Copy over all old data into the new data
   memcpy(data+1, originalData, dataLen);

   // new length is total data + 1 byte for the command
   dataLen = dataLen + 1;

   struct sockaddr_in dest;

   // configure the socket address with the looked up port
   memset((char*)&dest, 0, sizeof(dest));
   // Select IPv4 Socket type
   dest.sin_family = AF_INET;
   // Set destination port to datalogger
   dest.sin_port = htons(5006);
```

```
    // Use localhost ip as source of message
    dest.sin_addr.s_addr = htonl(INADDR_LOOPBACK);

    // Send the completed packet to datalogger
    size = sendto(fd, data, dataLen, 0, (const struct sockaddr *)dest,
       sizeof(struct sockaddr_in)) ;

    return size;
 }
```

Note: The dataLen is the length of the payload. "sendto" will do the math to ensure the total length of the packet is correct provided dataLen is correct.

G Receiving an ACK from Datalogger

Datalogger will respond with an ACK to the caller when it successfully receives a packet. The format of the ACK will be:

- The first byte is the command byte which for an ACK from datalogger is **0x0A**.
- The second byte represents what action Datalogger received.
  - 1: registered a sequence
  - 2: unregistered a sequence
- The next byte is the result code. This will be 0 on success and 1 on failure.
- For removing a sequence, the result code will be 0 if all unique IDs were found, otherwise it will be considered a failure if only part of the unique IDs were found.
- The unique ID that was requested to be registered/unregistered. For creating a sequence, this will be the name of the sequence created. For removing a sequence, it will be a list of successfully removed sequences.

Bytes	0	1	2	3	4	...
	Command Byte	Action	Result Code	Unique ID		

H Example Expected Input/Output

Example Payload Packet

```
#Packet Information
(UDP packet information...) 1331946059 0  <- cont.
        cont.-> TakePictureStandard.001\0 /data/TakePictureStandard.sh
        cont.-> \0width\0800\0height\0600\0 <-cont.
        cont.-> location\0/data/foo.png\0

#Script file
#Some other commands...
/usr/bin/takePicture $width $height
/usr/bin/parsePicture $location

#Datalogger will rune these commands as....
/usr/bin/takePicture 800 600
/usr/bin/parsePicture /data/foo.png
```

Example Sequence File

```
#TakePictureStandard.sh
#Command Parameters Redirection (Optional)
/data/takePicture $width $height > $location
/usr/bin/parsePicture $location
rsync /data/foo.png scMini@129.65.116.245:/data/foo.png
rsh scMini@129.65.116.245 /usr/bin/parsePicture $location
rsync scMini@129.65.116.245:/data/foo.png /data/foo.scMini.png
diff $location /data/foo.scMini.png > foo.diff
```

Example Location of Output file
This example assumes that there were other sequences that ran. The .001 ... .003 after each sequence name is defined by the UDP packet field unique ID.

```
edmund@edmund-desktop:/data/output$ ls -l
total 20
-rw-r--r-- 1 root root  5 2011-12-25 11:38 isPictureCloudy.001
-rw-r--r-- 1 root root  4 2011-12-25 11:38 isPictureCloudy.002
-rw-r--r-- 1 root root 10 2011-12-25 11:38 TakePictureStandard.001
-rw-r--r-- 1 root root  6 2011-12-25 11:38 TakePictureStandard.002
-rw-r--r-- 1 root root  3 2011-12-25 11:38 TakePictureStandard.003
```

Example Output file
A hexdump of the output file. In this example, the diff failed, so the sequence has failed to complete since it's not allowed to "complete" if the continue byte is set to 0. For this scenario, it actually did complete since only the last one failed, but it is still considered "incomplete".

```
edmund@edmund-Lenovo:executionEngine$ hexdump TakePictureStandard.001
0000000 0004 0000 0000 0000 0000 0000 0000 0000
0000010 0000 0000 0000 0000 0001 0000
000001c
```

Note: Done on a little endian machine.