CP7 and Datalogger Software Design and Implementation

Hoang (Mike) Tran

14 June 2011

Contents

Introduction **1**

What is PolySat? 1

What is CP7? 2

Design **3**

CP7 PIC Code Modifications 3

CP7 Avionics Specifications 4

Datalogger Overview 7

Configuration Files 8

Datalogger Implementation **14**

Conclusion **21**

Bibliography **23**

List of Figures

1 I2C Message Block Structure 3

2 CP7 Payload Command Handling Flow Diagram 5

3 Example Event/Subprocess Configuration File 9

4 Datalogger High-Level Flow Diagram 14

5 Configuration Code Macros With Example Configuration File 16

6 Flow Diagram of init_datalogger_DB Function 19

Abstract

This paper reports on the work done for PolySat, CalPolys CubeSat development team, on their CP7 project. CP7, a satellite designed to test and characterize particle dampers in microgravity conditions, uses some old hardcare and software, but is also one of the first projects to utilize their new electronics and software system. Part of the system, a process known as the Datalogger, is responsible for collecting and storing data into the database. This paper explains the design and implementation of the Datalogger, as well as modifications and specifications to CP7 software.

Acknowledgements

I would like to take the time to thank Dr. John Bellardo, for whose guidance and help made this project possible. Sean Fitzsimmons, for his continual input and sanity checking. Greg Manyak, for his hard work in the design of the software architecture. Daniel Walker, for his input and help regarding his CP7 PIC code. John Abel, for his input and thorough knowledge of the CP7 hardware system. Greg Eddington, and Greg Manyak and Dr. John Bellardo again, for their implementation of the PolySat process library used by the Datalogger. Aaron Morelli, for his implementation of the database API. For Dr. Jordi Puig-Suari, who created this amazing opportunity for students and continues to lead us. For Austin Williams, who designed the new bus electronics board so that we could develop new code to run on it.

Introduction

What is PolySat?

Dr. Jordi Puig-Suari, an aerospace engineering professor at Cal Poly San Luis Obispo, and Dr. Robert Twiggs of Stanford University developed a standard for picosatellites, the CubeSat standard, in 1999. The standard describes a 10 cm. x 10 cm. x 10 cm. structure (1U) that can be placed inside the Poly Pico Orbital Deployer (PPOD). The PPOD is attached to a launch vehicle and is designed to ensure safety of this vehicle and its main payload during launch, as well as deploy CubeSats once safe to do so. The CubeSat standard, along with the PPOD, has given a relatively cheap method for universities, corporations, and government entities to design, build, and implement cost-effective satellites for various space experiments. PolySat, the CubeSat research and development team at Cal Poly San Luis Obispo, headed by Dr. Puig-Suari, has been working on CubeSats since 1999. Currently, there are three CubeSats in orbit with several more in development.

With the development of a new electronics board and bus, and the upgrade from a PIC microcontroller to a microprocessor with an Atmel ARM9 architecture running a custom distribution of Linux, the PolySat software team deemed it necessary to also upgrade the software system running on the CubeSats. Formerly, the basic communications and command and data handling (C&DH) software were reusable between missions, but modifications were needed on a per satellite basis. These changes would need to be recompiled each time and specific payload software was required to be written in the same programming language as the system software. However, the PolySat software team decided that more flexible and modular software was necessary to utilize many of the capabilities made possible with the switch to running on a Linux operating system.

What is CP7?

PolySat, over the past couple years, has been designing and testing a CubeSat, codenamed CP7, and is in its final implementation process. CP7, made possible via a Northrup Grumman grant, implements an experiment to test and characterize particle dampening effects in micro-gravity conditions. CP7 will be one of the last CubeSats developed by PolySat that uses a PIC microcontroller; it will, however, also incorporate the new bus and its accompanying software system. The payload, consisting of a set of three particle dampers, is controlled by the PIC and the communications and power of the satellite is controlled by the new PolySat avionics system. The particle dampers are three separate cantilever beams filled at 0%, 95%, and 99% of a very fine tungsten powder. Piezoelectric actuators are used to activate the beams for vibration and accelerometers are placed on the particle damper heads for measurement. The PIC microcontroller is responsible for receiving commands from the avionics system, locking, unlocking, and actuating the beams, and collecting and storing the data from experiments.

Design

CP7 PIC Code Modifications

For the CP7 project, a different member of the PolySat software team originally implemented the payload experiment software that is controlled by the PIC microcontroller. He contributed the logic and code base in order to carry out experiments, however, left a few things unfinished that are required to enable communication between the payload system and the avionics system. The communication between the payload and avionics systems occurs via the Inter Integrated Circuit (I2C) bus protocol. There were drivers already written for this interface for the PIC, also written by a previous PolySat member, however, there was no command handling implemented on the payload side to receive experiment commands from the new avionics bus. The message block sent across this I2C interface requires a very specific format to work correctly with the driver used, and this format has not changed due to a very sensitive I2C driver that should not be modified.

Figure 1: I2C Message Block Structure

As shown in the above figure, there are at least three bytes of data that are required in the message. The first byte must be a number to indicate the command value and should correspond to a given experiment number defined in the payload code. The second byte signifies the total length of the message,not including the first command byte or last byte of the message block, which is a CRC byte of the entire message block. Following the length byte are a variable number

of data, or parameter bytes if additional data should be included with the specific message that is sent to the payload process. This field could be empty or, in the case of CP7, contain the necessary parameter data to unlock a specific beam or to run a frequency sweep on a beam. The final byte in the message is an 8-bit CRC checksum. This byte is used for error-checking to ensure that none of the data sent through the I2C interface were corrupted; if this is the case, the message is ignored entirely by the payload. In addition to implementing a command handler for the payload process, some modifications to the code were also needed because it was decided that some of the previously static run-time parameters needed to be dynamic. This was a relatively simple change, but did require a bit of time to become more familiar with the previously written code and to identify where appropriate changes were necessary.

CP7 Avionics Specifications

There were also some hardware features that were included specifically for the purpose of simplifying the implementation of software, but werent fully utilized. There is a general purpose input-output (GPIO) line connecting the payload system to the avionics system that was intended to allow the avionics board to monitor whether or not the payload board was currently ignoring I2C messages. This is important because the avionics system needs a method to know whether it should attempt to send an I2C message to the payload board; it was discovered on the Zero-G NASA flight, prior to a full engineering build of CP7, that the payload PIC was unable to handle I2C messages without affecting the data while it was trying to perform experiments. The I2C protocol is entirely interrupt-driven, so any data written to any I2C slave device connected to the same bus as the payload system will potentially trigger unnecessary interrupts. This forces the system into the interrupt handler to determine whether it needs to handle interrupt; this interruption creates an issue because experiment data has a high risk of becoming corrupted. In order to address this issue, the GPIO is used in conjunction with disabling the I2C interrupt in the payload code. After exiting the I2C interrupt handler, the payload enters a command check

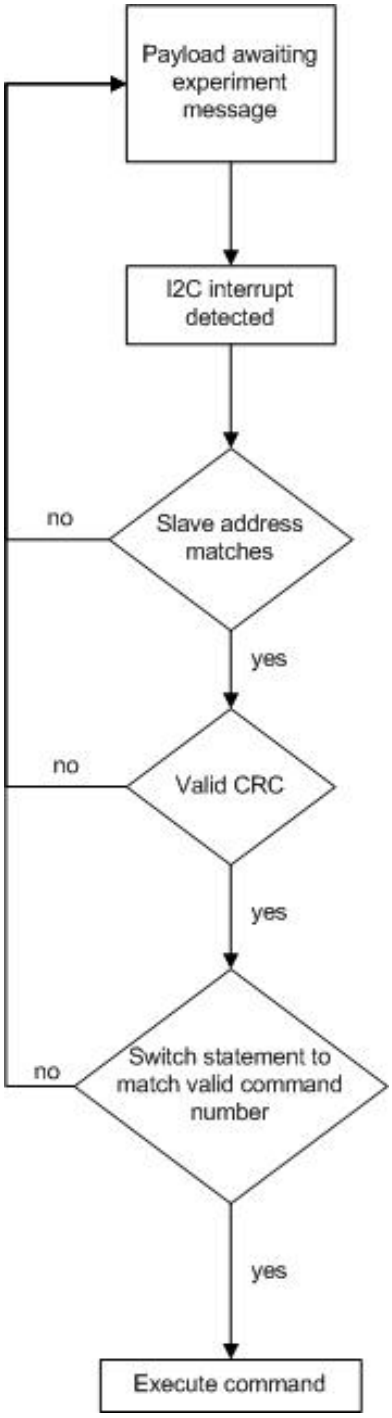

Figure 2: CP7 Payload Command Handling Flow Diagram

case; upon entering this check, the GPIO signifying that I2C communication has been disabled isset and I2C interrupts are disabled. The payload then checks the I2C message to see if the command matches any of the accepted commands if it does then the command is executed. Upon completion of the command, or if the message sent has an undefined command number, in which case nothing occurs, the system re-enables I2C interrupts and then pulls the GPIO low to signal the avionics system that it is ready to accept I2C messages again.

There is also an 8-bit GPIO expander that can be used to offer more information on the status of the currently running experiment, where six of these bits are used as a counter to indicate the status of the running experiment. After each successful completion of a frequency test, the counter will update itself. This allows a method of checking whether or not there is an issue with the payload process. If, for a predefined set amount of time, the counter does not change, then it can be assumed that there was an error executing the experiment. In this case, the avionics system has control to hard reset the payload system in order to recover from this error. It can then resend the command to restart the experiment.

One of the remaining bits of the GPIO expander is used by the payload process to signal completion of a sweep experiment. The last bit of the GPIO expander will be controlled only on the avionics side and read from the payload side. It is implemented this way because the payload board stores its data from the experiments onto external EEPROMs (located on the payload electronics). In previous PolySat missions, a data send-and-receive method was needed in order to get data from the EEPROM; however, another hardware enhancement allows both the avionics and payload systems to have access to the EEPROMs and the payload process controls which system has that control. At the beginning of a frequency sweep command, the payload will assume control of the EEPROM in order to allow it to write experiment data. At the end of a sweep, the payload process will transfer control to the avionics system to allow it to read data directly from it; this is simpler than implementing a data transfer protocol between the payload and avionics processors. It is the avionics process responsibility to first check whether

the payload is accepting I2C messages. If it does, then the avionics process must check whether the EEPROM busy bit in the GPIO expander is currently set. If it is, then it must not send a command to start an experiment. If the bit is low, signifying that no other process is currently reading from the EEPROM, then the command can be sent to execute an experiment. Once the command is sent, then the avionics process should monitor the 8-bit GPIO expander to ensure that there is not an issue that causes the experiment to stop on the payload side. The avionics process should also monitor the I2C_IGNORE line, which is the previously mentioned GPIO used to signal whether I2C messaging is disabled. The avionics process must also be sure to set the bit reserved for EEPROM busy status so that another process will not attempt to send a command to run an experiment. When the avionics process is done reading data from the EEPROM, the EEPROM busy bit should be set low to free it for future experiments and processes.

Datalogger Overview

The datalogger is one of the several static processes in the new bus software that is responsible for scheduling events that return data, which is ultimately input to systems database. Examples of such an event include sensor snapshots, beacon data, and satellite experiment-specific payload data. The datalogger must be as general as possible in order to maintain proper modularity and handle any possible data without prior knowledge of said data. It was also desirable for the datalogger to be reusable between experiments without needing any modifications to its code, aside from additional features being added. This led to an implementation that uses configuration files for all processes on the new bus, including one to initialize the datalogger with any necessary parameters needed for a particular mission. Datalogger also implements a method that can spawn additional processes to collect the desired data and stores this data into the database after receiving it from spawned processes.

Configuration Files

The configuration files offer a way to initialize the datalogger differently depending on which mission is being performed without directly changing the datalogger code. This allows the datalogger to only be compiled once, where different versions of the configuration files exist rather than different versions of the datalogger for various missions. The use of configuration files also allows for changes to be made to the datalogger settings during orbit if a change is needed. A new configuration file can be uplinked to the satellite and the datalogger process restarted, as opposed to the datalogger being recompiled on the ground and uplinking the entirely new process. The former would be safer and more reliable than the latter. The current design of datalogger uses three configuration files; one is for configuration of the events, another is for configuration of the database that datalogger is responsible for maintaining, and the last for configuration of commands this process can receive.

The event and subprocess configuration file supplies all necessary data for the datalogger to schedule events to occur. The maintenance of events is done with a priority queue that has two different types of events that can be scheduled. In both cases, a function is registered with an event as its call-back function to be called when the event occurs. One type of event, a file descriptor (FD) event, can be registered with any open file descriptor and will occur when a read or write, depending on the type of file descriptor, will not block. The other type of event, a timed event, is the type that the event and subprocess configuration file is responsible for. Timed events occur at specific moments in time, and optionally, at a repeating interval.

As seen in the above figure, the configuration file is formatted in a very specific manner for parsing purposes. Each object in the configuration file looks very similarly to an item in an XML file. The object begins with the <[object name]>tag and ends with the closing <[object name]>tag. Between the tags, the properties for that object are set. There are only two objects defined that will be parsed for the event and subprocess configuration file, which

```
<EVENT>
   NAME=current
   NUM=1
   START_TIME=P+20
   SCHED_TIME=30
   DURATION=0
   OPS=2
</EVENT>

<SUBPROCESS>
   NAME=current
   PROC_PATH=./current
   POWER=2
   POWER_PROC=./currentPower
   PARAM=
</SUBPROCESS>

<EVENT>
   NAME=temperature
   NUM=2
   START_TIME=P+30
   SCHED_TIME=20
   DURATION=0
   OPS=2
</EVENT>

<SUBPROCESS>
   NAME=temperature
   PROC_PATH=./temperature
   POWER=1
   POWER_PROC=./temperaturePower
   PARAM=
</SUBPROCESS>
```

Figure 3: Example Event/Subprocess Configuration File

are named EVENT and SUBPROCESS. Each EVENT object should have an accompanying SUBPROCESS object, otherwise, datalogger will not work correctly.

The EVENT object has several defined properties that describe an event from a higher-level view. The NAMEproperty is the main identifier and each event should have a different name. NUM is another identifying property and its use is optional; it is not currently being used, but was included for completeness and the plan is to include it in a future revision of datalogger. The START_TIME property is self-explanatory, however, the property itself can take on several different formats. The first two formats are variations that both represent Coordinated Universal Time (UTC) in the form of Hour:Minute:Second Month/Day/Year. The difference is in the interpretation of the year field. In one case, only two digits are expected and if the number is between 69 and 99, then the century is assumed to be in the 1900s; if the

number is between 00 and 68, then the century is assumed to be in the 2000s.

For the other UTC format, then the entire year, including the century, is expected. The hour field is represented using a 24-hour clock and the month should be represented in its number form(e.g., 1 for January, 3 for March). The UTC time format is not currently being used, but was included for future cases in which an event needs to occur at a very specific time. The next expected format is a representation with respect to mission time. This format is denoted by the leading characters, T+, followed by some integer number. The integer value following the T+ string is interpreted as the number of seconds after the mission starts that the event should occur. Mission time should be used when an event should execute at a specific time after the mission has started, but when the specific UTC time is unknown. The last acceptable START_TIME format is very similar to mission time, but is instead process time. It is denoted by the leading string, P+, and also followed by some integer number.The integer number following the P+ string is also interpreted as the number of seconds after the datalogger process starts at which the event should occur.

The SCHED_TIME property is used to specify whether the particular event should be rescheduled, and if so, at what rate it should be rescheduled. There is no special format for this property; the integer number supplied here is interpreted to signify the interval that the event should be rescheduled at following the first occurrence. The granularity of this property is in seconds. If it was desired for an event to repeat, but only for a certain amount of time, then the DURATION field can be used for this feature. The granularity of this property is in minutes and datalogger will continue to reschedule the event to occur at the specified SCHED_TIME until DURATION has been reached; it will then remove the event from the queue after at least the DURATION limit has been reached.

The OPS parameter is for specifying which mode of operation the event is allowed to occur in. The PolySat satellites generally have two modes of operation: pre-ops and normal ops. Most events, such as payload operations, occur during the normal operations mode of a satellite.

Beacons and sensor snapshots can occur in both pre-ops and normal ops. However, for some missions it may be beneficial to have events that only occur during pre-ops; since the configuration file for events and subprocesses is only read once and the events are queued at that time, the OPS property is used to keep track of this. This way, it became extremely simple to determine whether an event should occur or not on the satellite based on the satellites current mode of operation and the specified mode of operation for an event.

The SUBPROCESS object in the configuration file gives the necessary information that datalogger needs in order to spawn the subprocesses associated with an event, as well as additional information. SUBPROCNAME is the identifying property and this name should match the subprocess corresponding EVENT object. Each event should only be associated with one subprocess; the capability for an event to have multiple subprocesses has not been implemented, and there are no current plans to add that capability because there are currently no valid reasons to do so.

PROC_PATH is the property that links an event and subprocess to an actual process. This property should be a string of the complete path to the executable that should be called when it is time for the event to occur. The PARAMS property of the SUBPROCESS object is a string that lists any command-line parameters that need to be passed to the subprocess and a single white space character delimits each parameter. POWER is the expected power, in milliwatts, that the subprocess will consume in order to run. This parameter is important because on a satellite, power is limited and it must recharge whenever possible; the satellite should not attempt to spawn a subprocess if there is not enough power to execute the subprocess as it may cause the entire satellite to shutdown due to lack of power. Because the number in the POWER property is only an estimate, POWER_PROC links to an executable, in a similar fashion as the PROC_PATH property, which will calculate and return a more accurate power consumption value.

There is a second configuration file that datalogger uses to initialize and populate the

SQLITE database that stores and maintains sensor data. There are three properties associated with the database: locations, groups, and sensors. Sensor readings are the main items of the database; these are key-value pairs stored in the database, similar to a hash table. Each item for the database has a corresponding ID that is used as its unique key.This configuration file has three objects that it can recognize and parse. The GROUP and LOCATION objects both only have one property to them: an identifying name. Locations are used to describe where the sensor is located physically on the satellite. Similarly, groups are used to allow further organization of sensors. The SENSOR object has several more properties. Sensors also have an identifying name property to differentiate between them. Sensors generally have a location on the satellite and also can belong to multiple groups; it does not make sense for a sensor to have no location and to belong to no groups. For example, a temperature sensor could be located on the top panel and be part of the temperature group. The LOCATIONS property in the SENSOR object is a string that should match the NAME property of a LOCATION object in the configuration file for the database. The GROUPS property can list any number of groups where this sensor belongs. The names of all the groups where this sensor belongs should be delimited by one white space character and should also all match the groups NAME property. The last property defined for the sensor configuration is included for efficiency considerations and specifies how many groups are listed in the GROUPS property.

The last configuration file used by datalogger describes commands that can be sent to the datalogger. The format for this configuration file was defined and developed by another member of the PolySat software team, but it follows the same style syntax that the other two configuration files do: an XML-like look. The PROC field tells us which process the command is intended for. The NAME property is a description property for users to inform what this command will be used for (this is generally a one-word description). The FUNC property links this command to a specific function that is implemented in the source code for whichever process the command configuration file is for and it is case-sensitive. It should match the name of

the function character-by-character. NUM is an identifying property and used by the actual command-handler; the number of the command is included in the actual command sent and is matched with this field when figuring out which command should be called and then the function named by the FUNC field is executed. CMDOID, GRPOID and PROTECTION are included for security reasons. Currently, only a status command is mandatory; it is still a stub function that does not necessarily do anything because it has not been determined how the status should be handled.

Datalogger Implementation

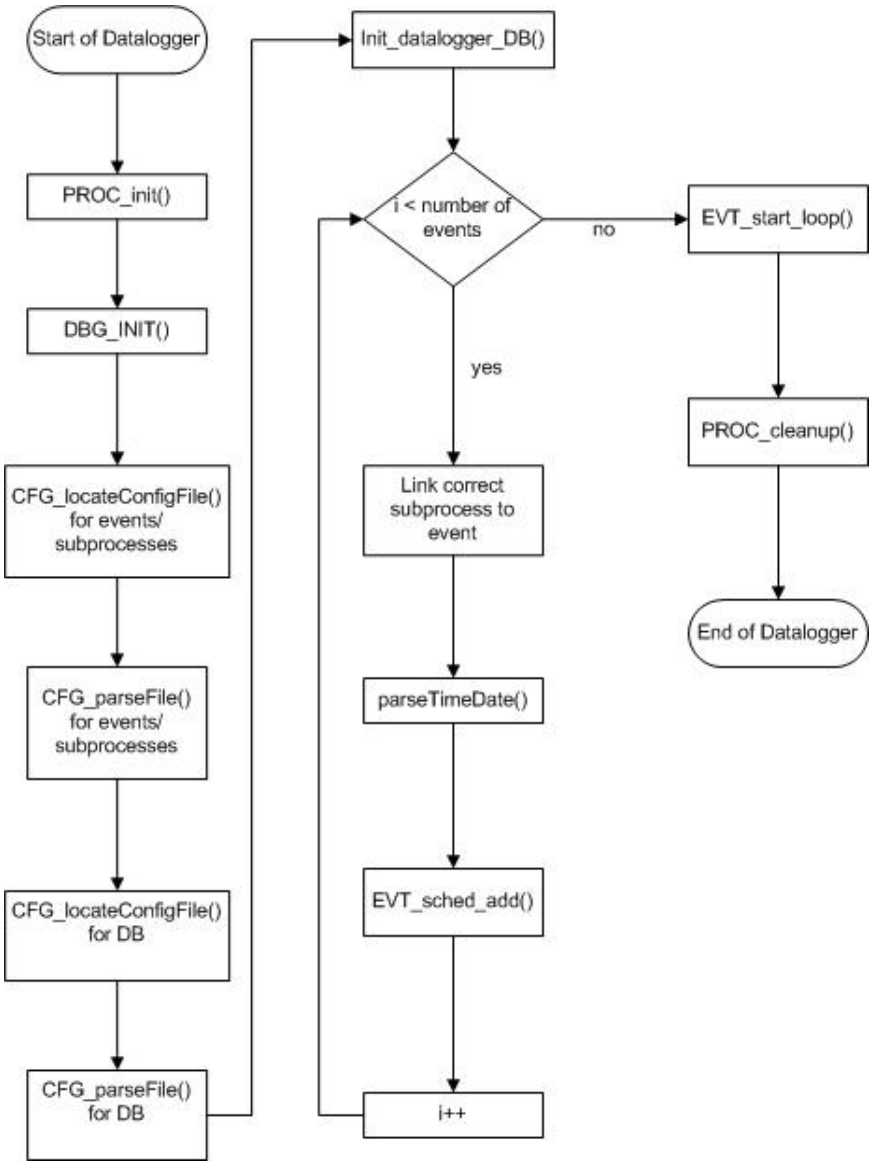

Figure 4: Datalogger High-Level Flow Diagram

The datalogger process begins by initializing itself for event handling. The PROC_init method accepts a string that denotes the name of the process and sets up the process for an event, as well as necessary initialization for command handling. The next step is to initialize the common debugging tools developed to standardize the debugging methods used by team members. The configuration files are then located and parsed. CFG_locateConfigFile accepts a parameter that is a string representing the configuration file's name. Then CFG_parseFile opens, reads, and parses the files for the objects. This method accepts a parameter that is a pointer to a defined C-typed struct in the configuration code. There are also user-defined structs that need to be constructed before configuration file parsing can be done. In this struct, one must declare the types and internal names to be used to access the properties declared for a configuration file object. Then, there are a set of macros to use to link the user-defined struct containing the elements and the object and its properties. CFG_NEWOBJ creates a new configuration file object, where the first parameter is an identifier, the next parameter is what to do upon initialization of this object, and the third is what to do upon clean-up of the object. The most common step upon initialization of an object is to use another macro which allocates memory for the structure type. Upon cleanup, there is not anything required to be done, so they are left as null. The following parameters to the CFG_NEWOBJ macro are optional based on how many properties are needed to be parsed from the configuration file. There are several macros that can be used depending on the type declared in the user-defined structure that is meant to hold the parsed information. Examples, such as CFG_STRDUP or CFG_UINT32, are used to link the struct to the object. Each macro takes three parameters of their own: a string to indicate the property name, the structure it should be associated with, and the field in that structure that should be linked to the property. An example of the source code required to link the EVENT object to its internal struct is shown in Figure 5.

```
<EVENT>
   NAME=current
   NUM=1
   START_TIME=P+20
   SCHED_TIME=30
   DURATION=0
   OPS=2
</EVENT>

<SUBPROCESS>
   NAME=current
   PROC_PATH=./current
   POWER=2
   POWER_PROC=./currentPower
   PARAM=
</SUBPROCESS>

<EVENT>
   NAME=temperature
   NUM=2
   START_TIME=P+30
   SCHED_TIME=20
   DURATION=0
   OPS=2
</EVENT>

<SUBPROCESS>
   NAME=temperature
   PROC_PATH=./temperature
   POWER=1
   POWER_PROC=./temperaturePower
   PARAM=
</SUBPROCESS>
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~/httran/seniorProject/httran/datalogger.cfg [+]                    15,9          All
```

```
struct CFG_EventDesc{
   char *evtName;
   uint32_t evtNum;
   char *startTime;
   uint32_t schedTime;
   uint32_t duration;
   uint32_t ops;
   struct CFG_SubProcDesc *subProc;
   uint32_t count;
};

/*
 * This macro ties the fields of the event in the config file to the corresponding field in the stru
 * This determines what the fields must be named in the config file.
 */
CFG_NEWOBJ(EventDesc,
      CFG_MALLOC(struct CFG_EventDesc),
      CFG_NULL,
      CFG_STRDUP("NAME", struct CFG_EventDesc, evtName),
      CFG_UINT32("NUM", struct CFG_EventDesc, evtNum),
      CFG_STRDUP("START_TIME", struct CFG_EventDesc, startTime),
      CFG_UINT32("SCHED_TIME", struct CFG_EventDesc, schedTime),
      CFG_UINT32("DURATION", struct CFG_EventDesc, duration),
      CFG_UINT32("OPS", struct CFG_EventDesc, ops)
);

/*
 * The structure used to hold the SubProcess data parsed from the config file.
 *
 * subProcName -
 * subProcPath -
 * power -
 * powerPath -
 */
struct CFG_SubProcDesc{
   char *subProcName;
   char *subProcPath;
   uint32_t power;
   char *powerPath;
   char *param;
};

CFG_NEWOBJ(SubProcDesc,
      CFG_MALLOC(struct CFG_SubProcDesc),
      CFG_NULL,
      CFG_STRDUP("NAME", struct CFG_SubProcDesc, subProcName),
      CFG_STRDUP("PROC_PATH", struct CFG_SubProcDesc, subProcPath),
      CFG_UINT32("POWER", struct CFG_SubProcDesc, power),
      CFG_STRDUP("POWER_PROC", struct CFG_SubProcDesc, powerPath),
      CFG_STRDUP("PARAM", struct CFG_SubProcDesc, param)
);
datalogger.h                                                    118,1          43%
```

Figure 5: Configuration Code Macros With Example Configuration File

After the configuration files are parsed, the database must be started and initialized. All of the data needed to populate the database are read from the database configuration file. Hash tables are used to offer a quick, efficient, and effective way of retrieving the database IDs. The key for each entry into the hash table is the name of the SENSOR, GROUP, or LOCATION (objects previously described); this must match the NAME used in the configuration file. The value tied to each key is the database ID number for the item. The database application program interface (API) was designed and implemented by a different member of the PolySat software team and is used to populate and maintain the database used onboard the satellite. First, the database must be opened with a call to open_psdb(). Next, the arrays of location, group, and sensor IDs are allocated memory space depending on how many of each is read from the configuration file. Using a Linux library (linked with ¡search.h¿) implementation of hash tables, the hash tables are initialized and created with a call to hcreate_r. The groups and locations are iterated and created in the database with create(), which is also included in the database API, and it accepts three parameters: a string symbolizing its name, the ID variable of type DBName_id, and the type of object it is creating, either GROUP or LOCATION. The string representing its name should be obtained from the configuration file, and the DBName_id can be retrieved from one of the ones in the arrays allocated previously. After the location or group has been created successfully, the DBName_id, which is set in the create function, is used as the value to the key, which is the name, to populate the hash table. Because sensors are associated with a location and a variable number of groups when created for the database, the first step of registering the sensors with their DBName_id requires a lookup in the hash table for the locations DBName_id that corresponds to the name for the LOCATION property in the configuration file. Since sensors can belong to multiple groups and the GROUPS property is a list delimited by a single white space character, the NUMGROUPS property is used to allocate the exact amount of memory for an array of DBName_id pointers for each group. Then, parsing the GROUPS property for each group name, a lookup into the hash table to acquire the correct DBName_id

is used and a space in the array of pointers is set to point to this value. Once the location and the group(s) for the sensor are set, then the sensor is created using the create_sensor() method. This method accepts as parameters, the name of the sensor, a pointer to the DBName_id of its location, an array of pointers to the DBName_ids of the groups the sensor belongs to, the number of groups it belongs to, and a pointer to a DBName_id for the sensor itself. All of the parameters, except for the last which was previously allocated and can arbitrarily be picked from any of the unused DBName_ids in the array for sensors, were parsed from the configuration file. After a successful creation of a sensor, the initialized DBName_id is used as the value, with the name of the sensor as the key, and the hash table is populated with this pair.

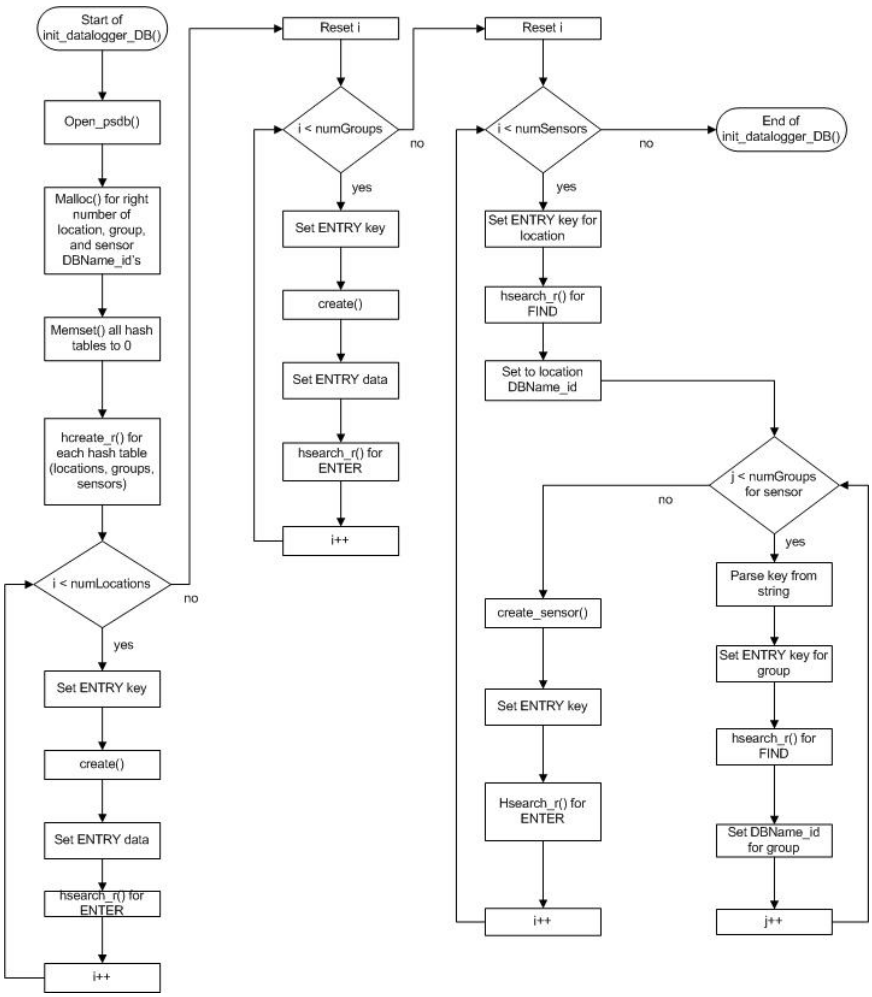

Figure 6: Flow Diagram of init_datalogger_DB Function

After the database has successfully been initialized and populated, the events and subprocesses that were parsed from their configuration file are actually linked together in a struct to be referenced more quickly than by searching through the arrays that are stored with the data when CFG_parseFile() was called. The start time of each event is parsed, using a call to parseTimeDate(), which returns the number of seconds between the time that it was called and the time the event should be scheduled to start. To be as general as possible in part of the implementation, all events use the same call-back function when they are scheduled, using EVT_sched_add(). The parsed start time is also used to signify when this particular event should occur and the structure containing the event data is passed into the call as its call-back data. Inside of the call-back function, the events property that specifies which mode of operation the satellite should be in for the event to occur is checked with the current mode of operation of the satellite. If they do not match, then the call-back finishes and nothing happens at all. If they do match, then datalogger should check if the satellite has enough power remaining in its batteries for it to execute the subprocess of this event. This is currently not yet implemented because it has not been determined the best way to do this, but the necessary information was included in the configuration file and half of the needed capabilities are already present. When it has been completely implemented, and if there is enough power remaining, then the subprocess associated with the particular event is started. To start an event, a child process needs to be started with a call to PROC_fork_child(). This method, developed for the PolySat process Library, handles fork()-ing a new process, exec()-ing the specified subprocess, and then pipe()-ing its file descriptors all in one method. After the child has been successfully started, a call-back is registered that listens on the stdout file descriptor of the child through a call to CHLD_stdout_reader(). Stdout is used for communication between the child and datalogger; the only constraint on subprocesses used by datalogger is that it must output the key, immediately followed by an equal sign, and then the value of any data the subprocess would like to return. Each key-value pair should be on separate lines to make for easier parsing. The

call-back function checks the buffer that is loaded with the data from the child process stdout for a newline. Once that newline character has been found, it passes the string starting from either the beginning of the buffer or starting from the first character following a newline into a function, parseChildOutput(), which extracts the key and the value pair from the supplied string. If the string supplied is in the expected format then a call to storeKeyValue() does a lookup in the hash table for the specified key, converts the string representation of value into a double type, and then uses the correct DBName_id to store the value into the database. Data can be retrieved from any subprocess spawned by datalogger so long as the data is supplied in the format just described. Currently, only a data-type of double is supported by the database API, however, datalogger has been implemented with the capabilities to convert the value into a type int as well. In future revisions, if different data-types are desired and supported, then datalogger will already have the means to store an int, a double, or a string into the database.

Conclusion

The datalogger implementation chosen is as general as any other possible implementation. The only two ways to generalize code in C to run different things depending on whats wanted is to use function pointers or to use separate processes. The use of function pointers requires the source code to be recompiled when a different function pointer needs to be used and limits the programming language that can be used, even if a different programming language makes more sense to do a specific task. Dataloggers implementation of spawning secondary processes allow these separate processes to be worked on concurrently and in any programming language; an implementation involving function pointers could result in conflicts.

If I had more time for my senior project, I would definitely want to test it more vigorously and extensively. There are a few issues which I felt were a bit rushed and I would have liked to spend more time implementing them for the best solution possible. An example of one

would be my method for passing in parameters to subprocesses. The configuration file having a property to list parameters works, but if given more time, I would like to investigate other, possibly more elegant, alternatives. One such alternative could be to pass the data through the subprocess stdin; the stdout file descriptor is already being used for the child to communicate with datalogger and an implementation writing to the stdin of the child may offer more flexibility.

Also, if presented with more time, I would have liked to refine the code being used for CP7. I spent a considerable amount of time trying to familiarize myself with the CP7 PIC code, some of it, mistakenly, was unnecessary. There are a few minor issues with the current PIC code that I would have liked to fix and implement and test to fully complete the payload system. I also spent a tremendous amount of time carefully planning the implementation of datalogger and would have liked more time to implement the CP7 subprocess more completely. There is a need for an 8-bit CRC generator for the CP7 project for its I2C message generation; they are currently being calculated before-hand and set in the configuration file.

Bibliography

- Manyak, Greg. Cal Poly Alumnus, Fault Tolerant and Flexible CubeSat Software Architecture, Thesis

- Walker, Daniel. Cal Poly Alumnus, "Development of a CubeSat Payload to Model Particle Dampening in Space: Design and Implementation of Software for CP7", Senior Project