Implementing A Robust Data Storage Software System For CP9

Author: Stuart Weickgenant

Advisor:
Dr. John Bellardo

Senior Project

California Polytechnic State University

Computer Engineering Department

June 2014

Abstract

This project focuses on the continuation of the CP9 CubeSat project, specifically the software which will be running on the satellite when it goes into space. This project mostly goes into designing a robust system which stores the data collected from the sensors on board CP9, whose purpose is to collect vibrations data from its launch vehicle during its ascent into space, into an easy to analyze system once the data is downlinked to PolySat's ground station after launch. One other thing this system does is to prevent CP9 from collecting unnecessary data after launch. Testing was done on this system to verify that this portion of the software for CP9 works as expected and to specifications.

Acknowledgements

I would like to thank my senior project advisor, Dr. John Bellardo, for all of his support and guidance throughout the development of the CP9 Software and also for responding to my late night emails when I had questions.

I would also like to thank everyone on the CP9 Software, Electrical and Mechanical teams for all of the hard work they put into this project before and during the time when I worked on this project, and also for all of their support during this entire process. Without them CP9 would not have been a reality.

Finally, I would like to thank my parents who supported me during this project and also during my entire time at Cal Poly.

Table of Contents

List of Figures

List of Tables

Introduction

PolySat and CubeSat

A CubeSat is a 10x10x10 cm cube-shaped nanosatellite with a mass of up to 1.33kg [1]. These satellites are widely used in universities and are also started to be developed by government and private entities, because of their relatively low cost and also their ability to test and verify a proof-of-concept of a particular component or software system that they use as the CubeSat's payload.

PolySat and CubeSat were both founded and still currently operate at Cal Poly, with the help of Dr. Jordi Puig-Suari, a cofounder of the CubeSat standard. There is one main difference between PolySat and CubeSat. CubeSat is in charge of the CubeSat standard and also builds the Poly-PicoSatellite Orbital Deployer (P-POD), which is what most CubeSats are launched and deployed from [2]. These P-PODs are attached, as secondary payload, to the inside of whatever launch vehicle they are manifested on. PolySat, on the other hand, is one of the the groups which builds the actual CubeSats which are placed inside of the P-POD.

CP9 Mission

The CP9 mission is a collaboration between PolySat and Merritt Island High School, which began in 2011, and is sponsored by NASA Launch Services Program (LSP) [3]. PolySat is developing CP9, which is a 2U CubeSat, while Merritt Island High School is developing StangSat, a 1U CubeSat.

The CP9 mission has two main parts. The first part is to measure and record data from its P-POD during launch at specified intervals after launch. These intervals are chosen by NASA

LSP. Specifically CP9 will record thermal data with a thermocouple as well as acceleration forces felt during launch with two, three-axis accelerometers. The second part is to demonstrate wireless communications with StangSat, which violates the existing CubeSat specification which does not allow RF communication inside the P-POD [3]. During launch, StangSat will record and stream its own launch data, namely accelerometer data, to CP9, in real-time, via a wireless communications module, which will subsequently be recorded and stored on board CP9. After CP9 has deployed from its P-POD, once the launch is completed, and PolySat has a established a RF link with CP9, both CP9's and Merritt Island High School's launch data will be downlinked to PolySat's ground station, via a RF radio link, for analysis.

Requirements and Specifications

Requirements

There are a number of requirements that the software for CP9 will need to follow in order to be successful and in order to meet the mission requirements.

Software Architecture

One of the main requirements is that the software follows the PolySat Software Architecture, which was first designed by Greg Manyak in his Cal Poly Masters' Thesis [4] and illustrated below in **Figure 1 [5].**

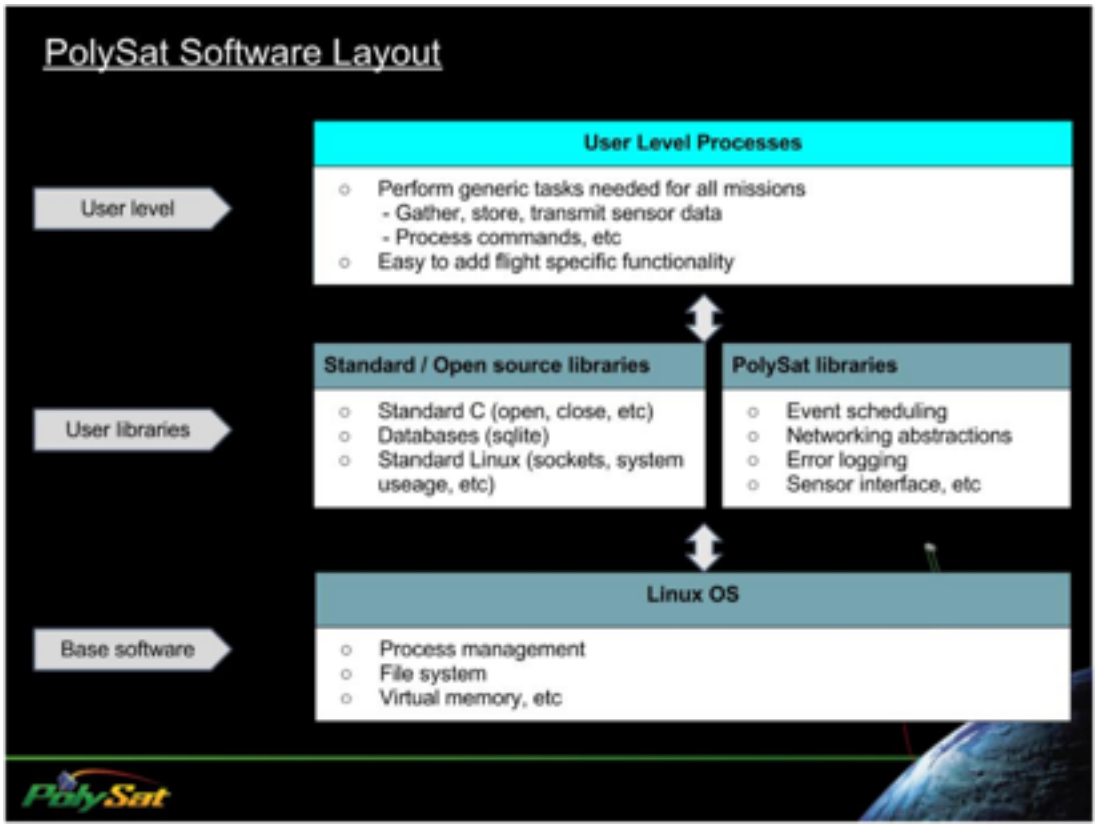

Figure 1: PolySat Software Architecture Design

In this design, the underlying OS for the software is Linux, which provides all of the memory and process management elements, as well as the file system, which will be needed for storing all of the launch data that is collected. Built on top of that are the standard Linux and C libraries as well as the custom PolySat libraries, which handle things such as event scheduling, interprocess communication, keeping time on the satellite, and sensor drivers. Above that, in the user level, are generic processes, which are common to all PolySat satellites, including communicating with the satellite via a ground station, as well as a payload process which handles mission specific requirements.

Thus, CP9's software will need to be modular enough to fit this software architecture, which means that the code can be reused with little to no modifications. The main differences between the software on CP9 and the software on future satellites should be the sensors which are used in the system, the mission-specific payload process, and the flight specific calibrations. All other parts of the software should be from preexisting libraries.

Reliability

There are two main reasons that CP9's software needs to be reliable. The first reason has to with CP9's mission. Because the entire purpose of CP9's mission is to collect launch data from its own sensors as well as to collect data from StangSat the software for CP9 will need to be designed with that in mind. Since the launch will only occur once this means that the launch data can only be recorded once, which means that the software needs to always work correctly as well.

The second reason, has to do with post-launch operations. As CP9 is orbiting the earth, it will have all of the data that it collected from the launch, which is ready to be downlinked to the ground station. Downlinking the data will take time, because of two factors. The first factor is that the RF link will be inherently slow, because of the distance the it has to travel from the satellite to the ground station. The second factor is that the satellite will only be in communications range of the ground station for a limited amount of time per day, since the ground station is on a fixed point on Earth, while CP9 will be orbiting around Earth.

Because of the long amount of time it will take to downlink of the data, the software will need to be robust enough to protect the data against unexpected factors. One unexpected thing that could occur is that the software reboots on CP9. This means that the data will need to be stored in non-volatile memory, so that the reboot doesn't erase the data. Another thing that needs to be considered, if the software unexpectedly reboots, is that the data isn't overwritten for any reason. One possible way this could occur is that the payload process restarts and begins collecting data from the sensors again, even though the launch has already taken place. Safeguards will need to be put in place to ensure that data isn't collected again so the launch data isn't overwritten.

Timing Requirements

As described in the CP9 mission section above, CP9 will need to be able to record and store data at specific intervals after launch. This means that CP9 will need to be able start and stop recording data almost instantaneously, so as to not lose any of the data. CP9 will also need

to be able to differentiate the data which is recorded at the different intervals. As a result, CP9's software will need to be both fast and smart to meet this requirement.

Specifications

Based on the requirements defined in the previous section, specifications for the software can be defined below in **Table 1**:

Table 1: Specifications For CP9 Software

Engineering Specfications	Justification(s)
1. No Memory Leaks	Memory leaks can cause CP9 to reboot, because of the lack of memory to run processes
2. Non-Volatile Data Storage	If CP9 reboots, the data will not be erased
3. Payload process will not run for a second time if CP9 reboots again after the launch has happened	Running the process a second time could cause data to be overwritten. Also the process would consume unnecessary power to read from the sensors
4. Needs to be modular	Code needs to follow the PolySat Software Architecture, where each process can be a standalone piece.
5. Needs to be able to record data at specific intervals of time after launch	Defined in CP9 Mission Requirements
6. Needs to be able to start and stop recording data based on start and end of specific intervals of time.	Defined in CP9 Mission Requirements and also not to use up unnecessary data storage space
7. Protect against overwriting data	Critical for mission success. Data collected during launch will not be lost

Design and Implementation

System Overview

One of the main things, that was considered when designing CP9's software system, was code reuse. A large amount of code was able to be completely reused or slightly modified from past PolySat missions and also from previous people who also worked on this mission. This code included RF communication with PolySat's ground station, which includes the ability to transfer files, the Linux operating system and file system, the ability to schedule code to run at specific time, process management code, code which polls a barometer to check if the launch vehicle that CP9 was placed in, has launched, and code which interfaces and reads the specific sensors from the hardware, including the accelerometers and thermocouple. One other thing that was able to be utilized, which was provided on the hardware side, was a NAND chip, which provides a non-volatile storage area, to store the launch data that was recorded.

The main things that needed to be designed and implemented were the ability to record data at specific intervals of time, including the ability to start and stop data recording, the ability to prevent the payload process from running from a second time if CP9 has already launched and recorded launch data, and the ability to differentiate between the data recorded at the different intervals of time.

Recording Data at Specific Intervals of Time

One of the mission requirements, is to be able to record data at specific intervals of time, which would include a way to be able to start and stop recording data, This section describes a design to achieve that.

All options included a structure which holds the start and stop time of an interval (in seconds after the system boots up), which is shown below in **Figure 2**:

```
typedef struct interval{
    int startTime, stopTime;
}interval;
```

Figure 2: Definition of Interval Struct

Because there could possibly be more than one interval to store data for, a secondary structure, shown below in **Figure 3**, was created which held a list of all the interval needed. This structure is also shown below with example intervals that are intervals for when to store data.

```
interval intervalList[] {
   {0,10},
   {12,20},
   {30,60}
};
```

Figure 3: Definition of IntervalList Struct

For the actual algorithm design, there were a number of different options considered. One option was to have a function, that was passed in a pointer to the intervalList struct described above that wouldn't exit until the last interval had run. The function would save the reading from the sensors if the difference between the current time and the time difference between the time when the system booted up, was the start time for the first interval. It would continue to run

record the data for this interval until the time difference equaled the ending time of the interval. After that it would move onto the next interval and repeat the process. This would continue until there were no more intervals to run. The flow diagram for this function is shown below in **Figure 4**.

Figure 4: Original Algorithm for Saving Data During Specific Intervals

This option, however, wasn't chosen. It wasn't chosen because CP9 would be spin waiting for a large amount of time and would waste all of the process resources checking the time every time around the while loop, finding the time difference, and then saving the data only

if the time difference was within the start and stop time of the interval. Using all of the system resources also would prevent the CP9 process from doing any other tasks. This is not acceptable as it does not meet the PolySat Software Architecture guidelines for being able to run tasks concurrently [4]. Another main reason this algorithm wasn't chosen had to do with checking the current system time. The system time is checked exactly once at the beginning of the loop. After the time is checked it takes some non-zero amount of time to complete the rest of the loop and then be able to check the time again, which means that the time difference could be just past either the start or stop of an interval which would cause a missed transition between saving and not saving data or vice versa. In order to improve this, a range of times could be checked instead of just a single time, however that would require calibrating this range to the time that the entire loop would take to run once.

In order to meet the PolySat Software Architecture guidelines and to eliminate the spin waiting within the system the previous option was modified to make use of the event scheduling functionality already built into the PolySat software library, in which a callback function is scheduled to run at a user specified time. The modified algorithm still makes use of the save data flag from the previous option, which says whether to save data or not. One change however is that all of the callback functions are scheduled at the beginning for the start and stop value of each interval, and then when each callback function the save data flag is toggled. A summary of the algorithm is described below in **Figure 5**.

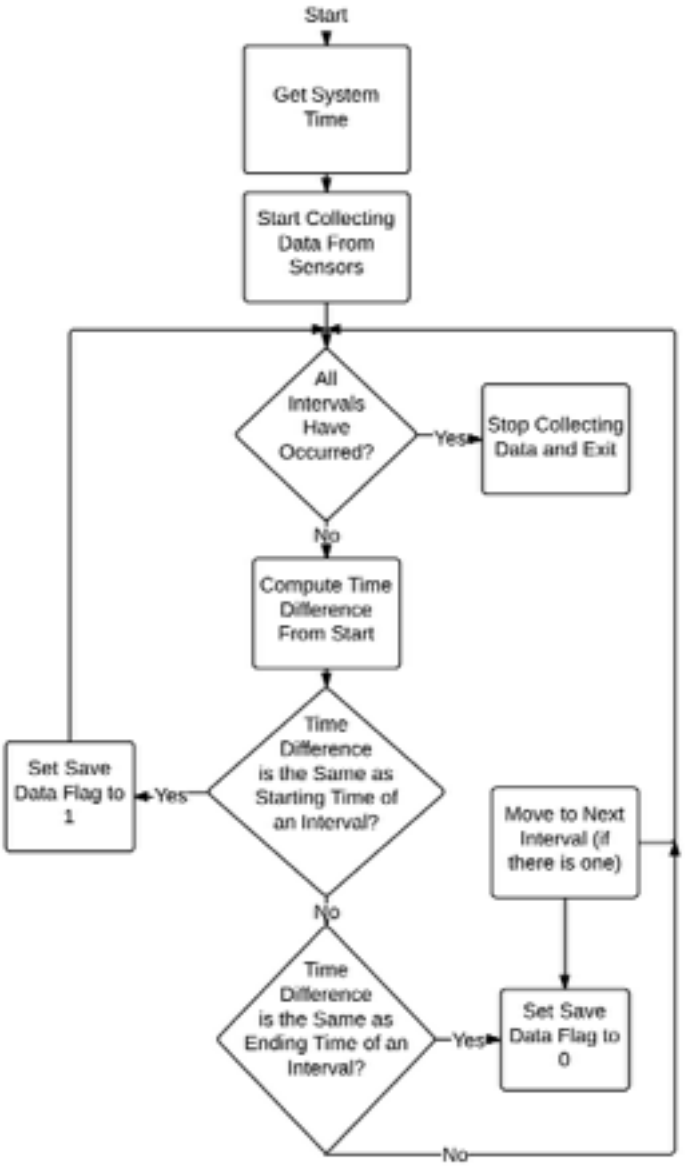

Figure 5: Algorithm for Saving Data During Specific Intervals

The advantage of this approach is that each event only consumes CPU time when its callback runs, which eliminates the spin wait problem of wasting system resources on one task. This also allows the payload to do other tasks in the mean time. Another advantage of this approach is that it eliminates having to check the system time inside of the callback function and

instead passes that responsibility to the scheduler of the Linux OS, which handles calling the callback functions.

Differentiating Between Data Recorded at Different Intervals

It is necessary to be able to differentiate between data recorded at the different intervals during the mission to make it easier to analyze the data when it is downlinked to the ground station and also to be able to better organize the data. This section will describe a method which achieves both of these goals.

For each interval of time to record data, a unique filename will be chosen, so that it is easy to differentiate between them. The filename will be chosen to contain useful information about an interval. The information includes the system time of when the interval begins, which will be taken when an interval begins. The system time, which is in milliseconds since epoch will be converted to a date and time (in the 24-hour UTC time zone format), using a standard C library function, to eliminate having to figuring it out once the file is downlinked.

In addition to the data and time, the file name would also include the number in the intervalList that the interval is, e.g Interval #1 would correspond to the first interval in the intervalList, while Interval #3 would correspond to the third interval in the intervalList to record data at. This number would be stored in a temporary counter variable which would be incremented when a new interval begins. Including the interval number in addition to the timestamp is also advantageous for being to distinguish the different intervals of recording data in case the RTC on the system loses its time. As soon as an interval ends the file would be closed

and the system would wait for the next interval to begin. An example filename is shown below in **Figure 6.**

2000-01-01-14:23:14-Interval#1.txt

Figure 6: Example Filename for an Interval

Inhibiting the Payload Process From Running a Second Time

One of the challenges with CP9, as opposed to other PolySat satellites is that the entirety of its mission takes place during the launch period instead of in orbit. Because of that the sensors do not need to be used after CP9 reaches orbit, so the ability to read from them is turned off by the software. If the satellite turn off, however and reboots, the payload process will turn on the sensors again if no precautions are put in place.

To prevent the satellite from running the payload a second time after launch, as soon as CP9 detects a launch and then has finished its last interval of time that its needs to collect data in, the system time of this event is recorded and then saved to a file in non-volatile storage on the satellite. If the satellite reboots, or the payload process restarts, before any of the sensors are initialized, the process will check to see if this file exists. If does exists it will compare the time that is in the file to the current system time. If the current system time is later than the time in the file, then this means that the satellite has already launched. As a result none of the sensors are initialized and no data is recorded. On the other hand if the file does not exist or if the system time is earlier than the time in the file then the system boots as normal.

The reason that the system time checking was used over another method, such as writing a random value to a file, is that it is very reliable, namely that the system time is based solely off

the value from a hardware RTC that the OS reads instead of a particular variable or function. This prevents software from being able to modify the system time. The only downside to this method is that if the RTC loses power in any way then the current time stored would be lost [6] . However, the chance of the RTC losing power is small.

Results and Verification

Inhibiting the Payload Process

Since this code directly affects whether or not data is collected not it is important to make sure that this code will always work correct. Testing this code will include looking at all three possible outcomes that could occur with this code running and making sure that the code will handle each case correctly. The three possible outcomes are:

1. The file that holds the timestamp does not exist
2. The file that holds the timestamp exists and the current system time is later than the time in the file.
3. The file that holds the timestamp exists and the current system time is earlier than the time in the file.

Table 2 shows these three possible outcomes, the expected results, and what each result would correspond to during the mission if it occurred.

Table 2: Outcomes and Expected Results For Payload Process

Outcomes	Expected Result	Meaning
Timestamp File Does Not Exist	Payload Process Collects Sensor Data	Launch Has Not Happened
Timestamp File Exists And Current System Time Is Earlier	Payload Process Collects Sensor Data	Launch Has Not Happened
Timestamp File Exists And Current System Time Is Later	Payload Process Does Not Collect Sensor Data	Launch Has Happened

Test Setup and Procedure

The test setup included simulating all three outcomes. Each outcome was tested 10 times to verify that it worked correctly. For each test there were two debug statements within the code, which were placed just before the code which determines if the sensors will be initialized. One debug statement said “Sensor Data Being Collected”, if the sensor data was going to be collected, and the other debug statement, said “Sensor Data Not Being Collected”, if the sensor data will not be collected. The debug statement will be output in: the /var/log/messages file. In addition to the debug statements, there will be one interval placed in the intervalList which has a range of 0 to 10 seconds. For these tests no data was actually collected, however an empty file which would hold the data was, if data was supposed to be collected. This file was placed in the sensor data storage folder.

For Test #1 where the timestamp file does not exist, the setup for each iteration of the test required removing the timestamp file (if one exists), and clearing the folder where the sensor data would be stored. Then the satellite was rebooted. After 10 seconds, which is the end of the

interval, the test was complete, and both /var/log/messages and the folder, where the sensor data was stored, was checked. The test was considered a success if the debug message “Sensor Data Being Collected” appeared in /var/log/message and also if there was a file inside of the sensor data storage folder with the correct filename which was defined in a previous section. If either of these weren’t present then the test failed.

For Test #2, where the timestamp file exists, and the current system time is earlier than the time in the file, a timestamp file was created on CP9 using the Linux system utility date which was output in milliseconds since epoch. This output was then redirected to a file with the correct timestamp filename. The full command to do this was: “date +%s > TIMESTAMPFILE, where TIMESTAMPFILE is the correct name of the timestamp file. After this file was created the time in the file was increased by 100,000 milliseconds. This was done so that the time in the file would be greater than the current system time and also so the test could be run multiple times without having to regenerate the timestamp file again and again. Both of these steps were done once before the first iteration of the test. Then for each iteration the same process, including verification, was followed, as in Test #1, except the timestamp file was not removed.

Test #3, which where the timestamp file exists, and the current system time is greater than the time in the file, the same procedure in Test #2 was followed, except that instead of increasing the time in the file by 100,000 milliseconds, the time in the file was decreased by 100,000 milliseconds. The test was considered a success if the debug message “Sensor Data Not Being Collected” appeared in /var/log/message and also if the sensor data storage folder was empty. If either of these weren't present the test failed. After all of the tests, the results were compiled and are shown below in **Table 3**.

Table 3: Test Results For All Payload Process Tests

Test Number	Test Simulated	Expected Result	Number of Tests Passed	Number of Tests Failed	Percentage of Tests Passed
1	Timestamp File Does Not Exist	Empty Data File is Created in Sensor Data Folder and Debug Message “Sensor Data Being Collected” appears in /var/log/messages	10	0	100%
2	Timestamp File Exists And Current System Time Is Earlier	Empty Data File is Created in Sensor Data Folder and Debug Message “Sensor Data Being Collected” appears in /var/log/messages	10	0	100%
3	Timestamp File Exists And Current System Time Is Later	Sensor Data Folder is Empty and Debug Message “Sensor Data Being Collected” appears in /var/log/messages	10	0	100%

Based on the data the modification to the payload process will work as expected.

Data Collection At Specific Intervals

This test combined being able to record data at specific intervals of time and also being able to differentiate between the data recorded since they could both be verified at once by looking at both the filename and making sure that data was recorded.

To verify that both of these features worked correctly three tests were designed with multiple intervals. The difference between each test was the spacing of how far apart the end time of one interval was from the start time of the next interval. For the first test there was a total of three intervals which were spaced 20 seconds apart from each other. For the second test there were three intervals again and each interval was spaced 5 seconds apart from each other. For the final test there was three intervals again but each interval was back to back, which meant the end time of the first interval was the start time of the second interval, and the end time of the second interval was the start time of the third interval. For all of these tests each interval lasted 30 seconds and the first interval began at 0 seconds. The purpose of this spacing was to determine if the software could start and stop recording data at any point of time. The other part of this test was to verify that the filenames were unique and met the design which was previously shown in **Figure 6**. **Tables 4, 5**, and **6** show the results of each test including expected and actual filenames.

Table 4: Test Results For Interval Spacing of 20 Seconds

Interval Number In List	Expected Time Difference Between End Of First Interval and Start of Second Interval	Expected Total Time Difference Between First Interval and Current Interval (sec)	Expected File Name	Actual File Name
1	20	0	2000_01_02_00_23_14-Interval#1.txt	2000_01_02_00_23_14-Interval#1.txt
2	20	50	2000_01_02_00_24_04-Interval#2.txt	2000_01_02_00_24_04-Interval#2.txt
3	20	100	2000_01_02_00_24_54-Interval#3.txt	2000_01_02_00_24_54-Interval#3.txt

Table 5: Test Results For Interval Spacing of 5 Seconds

Interval Number In List	Expected Time Difference Between End Of First Interval and Start of Second Interval	Expected Total Time Difference Between First Interval and Current Interval (sec)	Expected File Name	Actual File Name
1	5	0	2000_01_01_23_22_50-Interval#1.txt	2000_01_01_23_22_50-Interval#1.txt
2	5	35	2000_01_01_23_23_25-Interval#2.txt	2000_01_01_23_23_25-Interval#2.txt
3	5	70	2000_01_01_23_23_60-Interval#3.txt	2000_01_01_23_23_60-Interval#3.txt

Table 6: Test Results For Interval Spacing of 0 Seconds

Interval Number In List	Expected Time Difference Between End Of First Interval and Start of Second Interval	Expected Total Time Difference Between First Interval and Current Interval (sec)	Expected File Name	Actual File Name
1	0	0	2000_01_02_00_07_30-Interval#1.txt	2000_01_02_00_07_30-Interval#1.txt
2	0	30	2000_01_02_00_07_60-Interval#2.txt	2000_01_02_00_07_60-Interval#2.txt
3	0	60	2000_01_02_00_08_30-Interval#3.txt	2000_01_02_00_08_60-Interval#3.txt

The data for each of the filenames are match the expected values. For Test#1, the time between each interval was the expected difference of 50 seconds. For Test#1, the time between each interval was the expected difference of 35 seconds. Finally for Test #3, the time between each interval was the expected difference of 30 seconds.

Conclusions

Design Changes

Based on the design and results of this project, there are a couple of things that I would have added or changed to have made the software better.

One thing I would have done is added some more fields to the interval struct. Because each interval corresponds to a specific amount of time, it would have been useful to add a field, which could store a string that describes what is supposed to be happening during this period of the launch. For example, if the first interval started at 0 seconds after boot and ended at 5 seconds after boot, then a string could be added that says “Initial_launch_vehicle_take_off”. This string could then be appended to the filename, so instead of the filename in **Figure 6** being named 2000-01-01-14:23:14-Interval#1.txt, it could instead be named: 2000-01-01-14:23:14-Interval#1-Initial_launch_vehicle_take_off.txt. This would make the filename more descriptive and also would eliminate having to look up what each interval corresponds to.

Another changes I would have made is making the code which detects if the launch has already occurred more robust. As described earlier, the timestamp will only be written to a file, after the last interval of time to record data has ended. One problem this could bring up is that, if for some reason CP9 reboots during launch without the timestamp being written, then CP9 would start collecting data again at the first interval of time. If the timestamp was instead recorded right after a launch is detected, then if CP9 reboots right after that, then it would lose any future data from that point on, because CP9 will have seen that the timestamp has already been written to the file. One change could be using a two timestamp where both of these

situations are combined: one timestamp is recorded immediately after the launch starts and another timestamp is recorded as soon as the last interval ends. If CP9 reboots then the software would check that both of those files exist, and also if the current system time is greater than that the timestamp taken when the last interval was taken. If all of those conditions are true then the launch has already occurred. If not then it would start taking data again.

Future Work

Besides the things that were listed in the design changes section there are no more major structural changes that need to be made to CP9's software. The only thing left to do now is to change specific flight calibrations values in the software once CP9 is assigned to a launch vehicle and PolySat knows more about the orbit that it will be in. Such flight calibrations would include knowing exactly what specific intervals CP9 would be recording data at during launch, and also things not specifically included as part of this senior project such as what frequency CP9 would be communicating at with the ground station or when to turn on CP9 antenna, or the specific beacon format which will be used to track CP9.

Specific flight testing will also need to be performed in a vacuum chamber and on a vibrations tables to simulate actual data that could be taken from the sensors. This testing will also be used to further verify the code which polls the barometer to see if a launch occurred, which was briefly mentioned in the System Overview section.

Conclusion

Based on the design, implementation, and testing of the software for CP9, CP9 will be able to detect when a launch has occurred, through the use of a timestamp which is stored to a file in non-volatile memory after the last interval has ended after the launch has started. This was tested by writing a timestamp to a file which was earlier than the current system time and also by writing a timestamp to a file which was later than the current system time. The only way that this system would fail is if the RTC, which CP9 gets it time from, loses power. Interval testing was verified by testing intervals which started one after another and also by testing intervals which were far apart from each other. One way this system could be improved is by adding a string to the interval struct, which would describe what is going on when the interval is happening. The software, after the flight specific calibrations are made, will work as expected when CP9 is launched.

References

[1] "About Us." *CubeSat*. CubeSat, 2014. Web. 22 May 2014. <http://www.cubesat.org/index.php/about-us>.

[2] *Educational Payload on the Vega Maiden Flight Call For CubeSat Proposals* 1 (2008): n. pag. European Space Agency, 25 Feb. 2008. Web. 22 May 2014. <http://esamultimedia.esa.int/docs/LEX-EC/CubeSat_CFP_issue_1_rev_1.pdf>.

[3] "CP9." *PolySat*. PolySat, n.d. Web. 22 May 2014. <http://polysat.calpoly.edu/in-development/cp9/>.

[4] G. Manyak, “Fault Tolerant and Flexible CubeSat Software Architecture,” Master’s thesis, CalPoly - California Polytechnic State University, 2011. [Online]. Available: http://digitalcommons.calpoly.edu/theses/550/

[5] Bertolino, Dominic, “Implementing a MATLAB Based Attitude Determination Algorithm in C within the PolySat Software Architecture” Senior project, California Polytechnic State University San Luis Obispo, 2013. DigitalCommons@Calpoly. Web. 22 May. 2014.

[6] "HowStuffWorks "Why Does My Computer Need a Battery?"" *HowStuffWorks*. HowStuffWorks, n.d. Web. 24 May 2014. <http://computer.howstuffworks.com/question319.htm>.

Appendix A

ABET Senior Project Analysis

Summary of Functional Requirements

The software designed for this project will be used on CP9, a CubeSat designed and developed by PolySat at Cal Poly. The software will be able to collect and record launch data at specific intervals of time, which will be determined by NASA. The software will also be able to detect if a launch has already occurred, and if so, turn off the data collection features on CP9. Finally, after launch, CP9 will be able to downlink the data that has been collected to PolySat's ground station.

Primary Constraints

The primary constraint with this project was the time needed to write, develop, and test. Because the software needs to work correctly, as it is going to be going into space, a lot of time was spent testing and debugging the software to make sure there were no bugs. Other constraints included making the software fit with the PolySat Software Architecture, which was described in the report, and being able to run specific code at specific times during the mission.

Economic

There is no direct cost associated with the software for CP9, because C, which is the programming language for this software, and its compiler, gcc, are both free.

However, there is a cost to write the software. Assuming that the programmer is paid the current California minimum wage rate ($8 USD), and the software takes 200 hours to research, write, test, and debug, then the cost will be $1600 USD.

If Manufactured on a Commercial Basis

The software specifically written for CP9 is meant to only be used for this mission, however parts of it can be modified and reused, but only by members of PolySat. Therefore this software will not be commercially sold.

Environmental

There is no environmental impact for this software except the amount of time on a computer that was spent writing it. This is because a computer consumes electricity which may have come from a non-renewable energy source such as a fossil-fuel power plant.

Manufacturability

Software is not manufactured, therefore there are no issues associated with this.

Sustainability

The only challenge with maintaining this system is inserting the flight-specific calibration values for certain parts of the software. If that is done then the software image will need to be rebuilt and programmed onto CP9.

The only resource that the software consumes is electrical power, because of the processor that the software is programmed on to. If the energy used to power the processor is taken solely from sustainable sources such as the Sun, then the software would be completely

sustainable. However, since batteries are used to help power the system there are resources used to make and charge the batteries.

Ethical

Because the data that is taken during CP9's mission will be shared only with members of NASA and PolySat, steps must be taken to ensure that this data isn't accidentally shared with any others. These steps includes security usernames and passwords so that only members of PolySat are able to log onto the satellite or download the data from the satellite.

Health and Safety

The only health consideration to take into account is the well-being of the programmer and making sure that he or she doesn't get any injuries, such as carpal tunnel or cramps, when writing this software.

Social and Political

The software for CP9 is ITAR controlled. ITAR stands for International Traffic in Arms Regulations, and is a set of U.S government regulations which says that information related to weapons (CubeSats are considered weapons) may not be shared with any non U.S citizen. Therefore the specific details about CP9's software will have to be carefully controlled so that it isn't shared with non U.S citizens.

Development

During this project I learned how to use an open source build manager called Buildroot, which PolySat uses to configure and build its satellite software images. I also learned how to

use Git, which was used to backup code that I developed. Finally, I learned how to use event-driven programming, which is programming that is driven by specific actions that the satellite will take.