/usr/sbin/clksync

LEO Nano – Satellite Clock Synchronization Software

By Matt Zimmerer

Senior Project

COMPUTER ENGINEERING DEPARTMENT

California Polytechnic State University
San Luis Obispo

June 2013

Table of Contents

List of Figures/Tables

Acknowledgements

I would like to thank my friends and family for supporting me during the development and testing of this and related works with Polysat. I give many thanks to the guidance and help granted by Dr. Bellardo in the conception and development of this software. I also wish to thank all those that spent countless nights debugging a system event only to uncover clksync as the culprit, my condolences. I also want to thank Dr. Benson for helping me archive this work into the CalPoly senior project archive.

Abstract

A CalPoly Student Based Organization called Polysat takes mission contracts regarding the development and production of nano-satellites designed to fulfill specific mission requirements. Nano-satellites themselves are very complex and dense electronic devices. Polysat Satellites have recently switched to a new version of the main system board. This system board hosts an ARM AT91SAM92G0 processor, and enough hardware to comfortably run an embedded version of the Linux kernel. Each mission requires specific configuration for the system image, and specialized software. I've spent much of the last two years developing software, and recently debugging both my software, and the software system as it runs on as a whole. One particular piece of software I wrote is called 'clksync'. Clksync is clock synchronization software that allows a ground station to synchronize a fast moving low earth orbit satellite to a ground clock, correcting for both clock offset and clock drift.

Introduction

Synchronizing an orbiting satellite's clock can be broken into 2 distinct problems. The first problem is how to synchronize the clock in space to a clock on earth. The second problem is how to correct for clock drift caused by the massive temperature derivatives found when the satellite moves in and out of view of the sun. Having synchronized orbiting clocks does have great benefits. The most notable benefit is that a synchronized clock can be a replacement for a power hungry GPS unit. The correct time on-board a satellite can be fed into an orbital model, which will yield the correct position (+- error of the clock). Secondly, timestamps on telemetry points require no post processing on the ground, if they were correct in the first place.

Synchronizing clocks on earth is easy. Network Time Protocol (NTP), has a few well trusted time servers that broadcast time packets, and as they do so, the average time latency on any given physical layer is remembered, and added into the time synchronization algorithms (a distributed algorithm). The problem becomes quite a bit harder when that time delay is changing too fast for NTP to accommodate for. A nano-satellite typically flies in low earth orbit (LEO), and at a velocity on the order of 7.8Km/s. At this velocity, the propagation delay to the satellite is changing so rapidly, that the only way to really know the prorogation delay is to calculate it beforehand using a mathematic model of the satellites trajectory. Knowing this model, adding in the propagation delay becomes a simple matter of having a correct orbital model.

The second problem that must be solved is how to account for the massive clock drifts as the satellite moves in and out of the sun. Changes in heat will change the oscillation frequency of the system processor, which will in-turn change the system clock tick rate. Fortunately, our Linux Kernel allows the conversion from electronic clock tick to kernel clock tick to use a variable transform. That is where we can change the amount of clock ticks it takes for one microsecond of system time. The proposed solution was to use an on-board real-time clock with high thermal stability as a baseline for the system clock. This would grant the unstable system clock the stability of the real time clock without any loss of stability.

Background

NTP – Network Time Protocol. This is the standard Internet protocol used to synchronize time across nodes in a network. This was the baseline for my initial research, and some of the low level software interfaces I used exist only for NTP.

LEO – Low earth orbit. Nano-satellites typically operate in LEO, at a velocity on the order of 7.8Km/s

Linux – An open source operating system, and the OS of choice for Polysat Nano-satellites.

Kernel – The core of an operating system. Performs all the low level maintenance required to give the end user or user level software a stable operating platform.

RTC – Real time clock. A clock designed to hold 'real time', such as the current year/month/day/second. RTCs typically only have seconds resolution.

Virtual RTC (vRTC) – A virtual real time clock written specifically for this software. It is a mathematical extension of the RTC, allowing frequency and offset transformations as well as providing micro second resolution.. The ground to earth drift and offset are applied here.

System clock – A kernels clock, which is seeded from the CPUs internal oscillator. A clock maintained by its operating systems kernel. These are usually set to the on-board RTC on system boot.

Clock Drift – The amount by which the offset between two clocks grow over time. Typically measured in parts per million (ppm), or in other words, microseconds per seconds.

Clock Offset – The instantaneous offset between two clocks.

Packet – A grouping of data transmitted across a network.

EMA – Exponential moving average, a simple data smoothing algorithm.

Outliers – Data points that are sufficiently far from the data set's average.

Propagation Delay – The delay it takes a physical signal to travel across a physical medium.

Phase Change Memory (PCM) – A special type of non-volatile memory that is radiation hardened.

Requirements

1. The system clock must not drift from the RTC. Any substantial drift will make the system clock time unreliable, and other system components may begin to fail.
2. The system clock offset to the master ground-clock must be reasonably small. If after synchronization, the offset is still larger than 100ms, the system has failed at synchronizing the clocks.
3. The drift rate of the satellite clock must match the drift rate of the ground clock after synchronization. If after a substantial period the satellite clock's offset has grown beyond a reasonable measure, the system has failed at synchronizing the clocks.

Specifications

1. The clksync process must not use a significant amount of CPU time. Since clksync operates as a 'real time process' with maximum priorities, using too much CPU time will prevent other system components from functioning.
2. Clksync is capable of rebooting the system. Clksync must ONLY reboot the system after a ground-station has synchronized the target satellite, and a reboot is necessary to set the system clock.
3. The system clock can NEVER be set (except at boot), only speed up then slowed down to account for small offset corrections. Setting the system clock directly will cause other software components to fail.
4. Clksync must always be able to respond to watchdog status requests. Failing to do so at any time will cause the satellite system to unexpectedly reboot.
5. If the board has a broken hardware component, the software must not fail, but proceed in a clean and deterministic fashion.
6. Software state must be saved to PCM (phase change memory) when the system is rebooting, or about to reboot.
7. Sampling of the RTC must occur on the seconds roll over event. Polling the RTC will cost too much, and there are no available interrupts, so a binary search must be implemented.

Design

Virtual real time clock

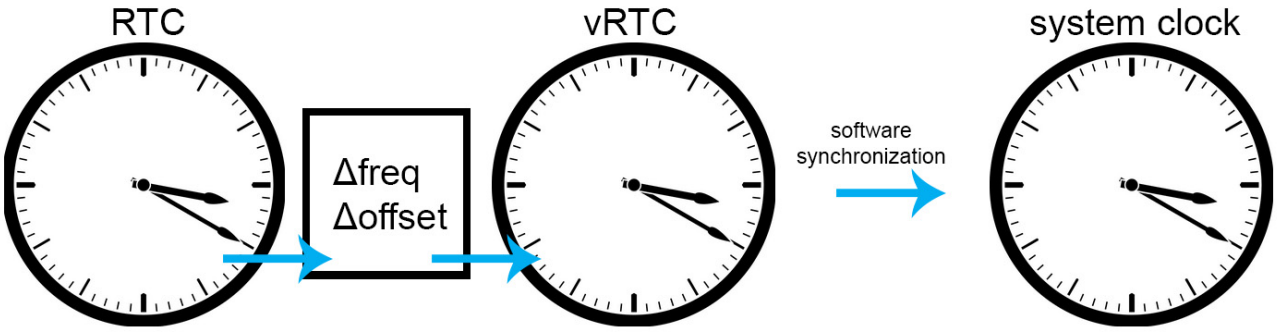

Figure 1- Conceived clock system for clksync

The virtual real time clock is implemented as a wrapper over reads and writes to the real time clock. There is a frequency and offset transformation applied, allowing the user to virtually speed up/slow down/ or offset the actual hardware. (see figure 1).

```
struct vRTC {
        int rtc_fd;
        int32_t freq;
        int32_t offset;
        int32_t rtc_last;
        int32_t rtc_adjustment;
        int32_t offset_leftover;
        uint8_t state;
};
```

Struct 1- virtual real time clock state struct

The virtual real time clock has a few memory fields used to maintain its state. The frequency and offset are the actual transformation variables input by the user. The rtc_last field is set to the RTC reading each time the virtual real time clock is read. This allows the offset to grow over time based on the frequency by extrapolating a delta from the last RTC reading. The rtc_adjustment field is a 0 biased signed integer containing any pending additions or subtractions to the RTC. For instance, if the frequency is set, and over time the vRTC drifts from the RTC substantially, the vRTC object may decide to actually set the RTC by a second, and do the inverse to the virtual clock fields. The offset_leftover field remembers small fractions of a microsecond at each vRTC reading that are left over from the frequency adjustment extrapolation. This is necessary to make the possible frequency shift values truly continuous, and promote system precision. The final struct member state, is unused, and seems to have escaped my code cleaning iterations.

Synchronizing the system clock to the Real time clock.

One of the important aspects of this project was ensuring a thermally stable clock. To accomplish this, I implemented a software state machine that synchronizes the kernel's system clock to a thermally stable real time clock. To accomplish this, the vRTC's second rollover event needed to be captured, and measurements needed to be made in order to make the appropriate adjustments with the kernel. The virtual real time clock was used since it just a mathematical extension of the RTC, and thus has the same thermal stability of the hardware.

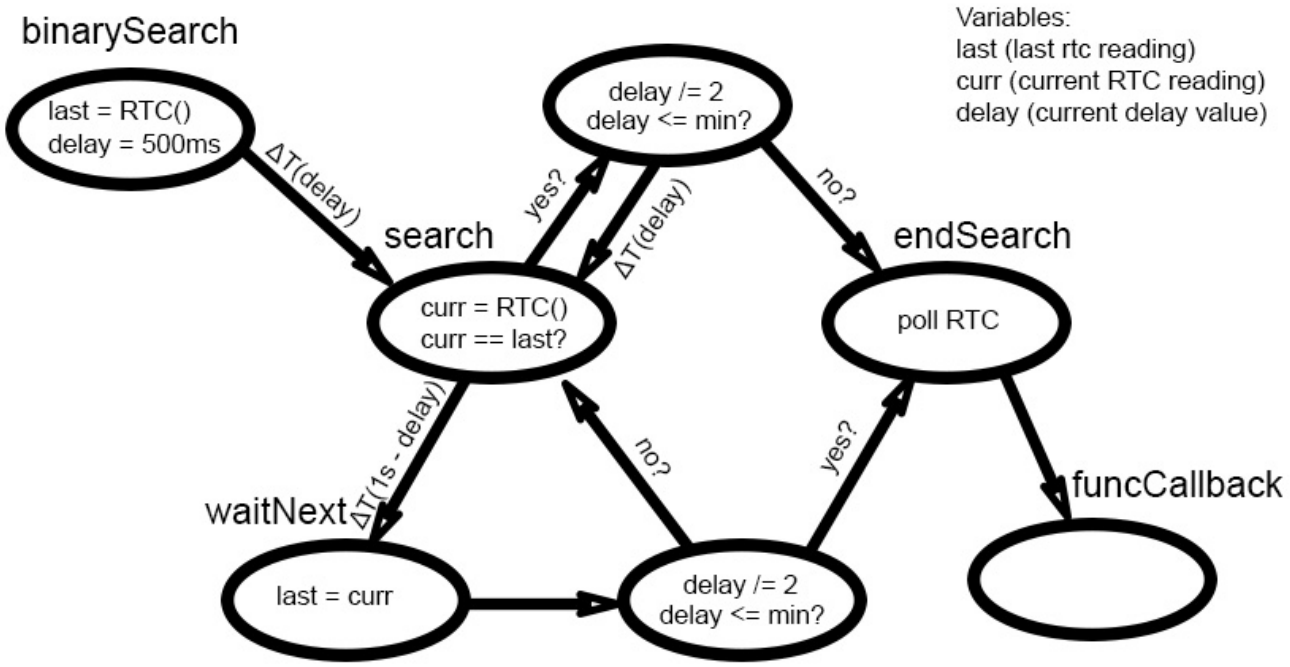

Figure 2- Binary search software flow diagram

To capture the RTC seconds edge, yet another state machine was devised (see figure 2). Since polling the RTC would use too much CPU resources, and there was no capture-able interrupt event, a binary search was proposed. The binary search algorithm is a simple state machine implemented using a Polysat proprietary process event system. When a second rollover event is captured, a function callback is made, allowing this state machine to be used for different purposes. The binary search function starts the event chain. It records the RTC reading, and waits 500ms. Then another reading is taken, if the seconds field has changed, it waits the rest of the second (1 – delay). If the RTC value has not changed in the first search call, it keeps waiting less by a factor of two until the delay meets a minimum threshold. In any case when the delay meets a minimum threshold, the binary search ends, and the RTC is then polled a minimal amount. Once the seconds field changes and the rollover event is captured, a

function call back is made to perform whatever action was intended.

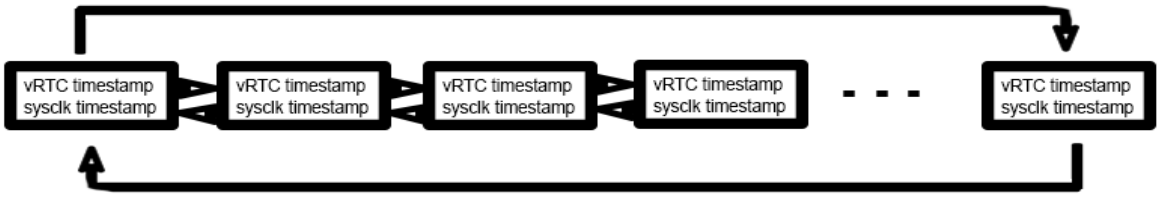

Figure 3- Doubly linked list of vRTC and system clock samples

To measure vRTC to system offset and drift, a linked list data structure with data smoothing algorithms was implemented (see figure 3). At each RTC seconds rollover event two samples are taken: one sample of the vRTC and one sample of the system clock. These samples are inserted into a doubly linked list, which grows at each sample. Once the linked list grows sufficiently long to measure drift rates and offset measurements, they are taken and fed into an exponential moving average filter. Once at least 5 entries have been added into the filter, offset measurements are made to determine if the kernel needs a frequency or offset adjustment. If the vRTC to system clock offset exceeds a set amount, both the frequency and offset are negated by calling adjtimex(), which is used by NTP to discipline the kernel clock.

System calls can take variable amounts of time to complete. This variable amount of time can break the stability of clksync by invalidating measurements causing the software to incorrectly adjust the system clock. To improve this aspect of the software, the priority of this process is set at run time to the Linux 'real time priority'. This has the advantage of expediting all system calls and improving performance, but has the disadvantage of allowing clksync to reduce the performance of all other running processes. Since clksync is running as a real time process, special care was made to make the implementation efficient. This was accomplished by favoring fixed point operations over floating point operations, preventing any polling behavior, and using the -O3 compiler optimization flag.

Synchronizing the virtual real time clock to a ground clock

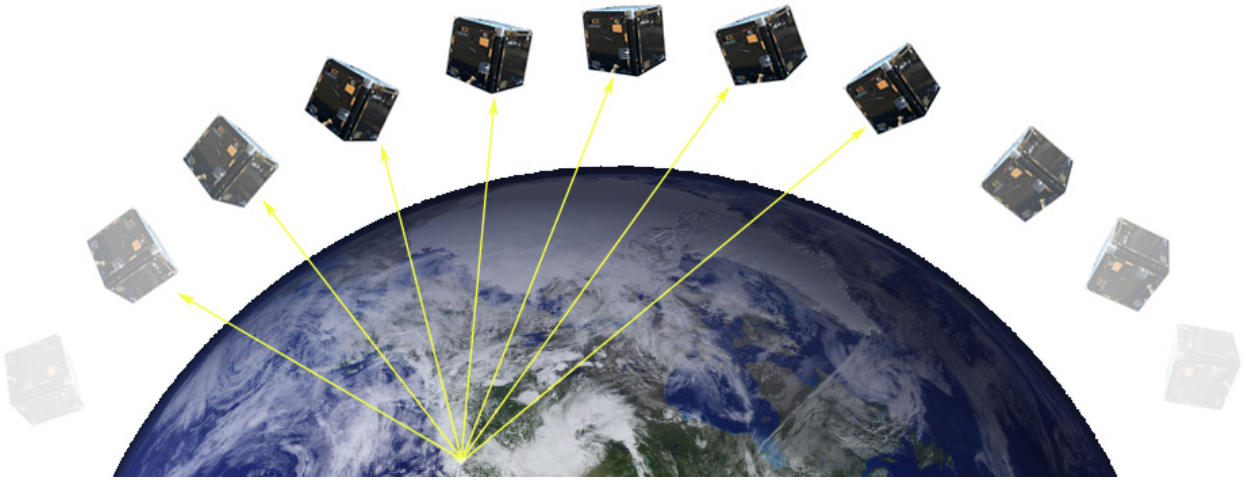

Figure 4 - Visual representation of variable propagation delay with time packets

This aspect of synchronization was designed with 'time packets'. A time packet is a small packet structure that contains the expected time of arrival as a timeval (32 bits for seconds since epoch, and 32 bits for microsecond offset). A ground station sends 15 time packets toward an orbiting satellite, and when the satellite receives a packet, it only needs to synchronize to the 'expected time' received. This allows much of the processing to be done on ground. (see figure 4).

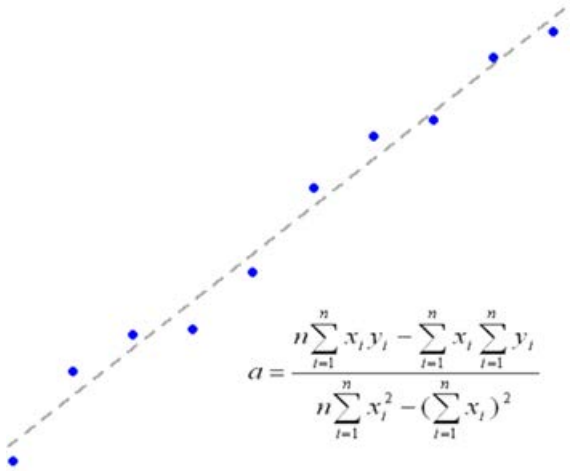

Figure 5- Linear regression slope calculation

To measure drift and offset, a simple linear regression was implemented (see figure 5). Figure 5 is a general representation of how to calculate the slope component of a linear regression. The clock

synchronization software for instance, had as an x-axis, the real time the packet was received (in seconds), and the y-axis as a calculated signed difference between ground and earth timestamps (in microseconds). The slope therefore has units of usec/sec, or parts per million (ppm), which is the standard unit for measuring clock drift. It was difficult for me to prevent overflows with fixed point 32bit numbers, so I opted for a floating point solution. Lastyle, the offset is taken from the last received packet and applied to the satellite's clock system.

Data outliers proved to be a real problem in preventing the linear regression from properly synchronizing the clocks. A single outlier might significantly throw off the drift calculation, and cause the satellite to drift excessively. To remove outliers, a simple solution was used. The time-packet list is sorted by greatest offset, and then the N largest outliers are removed. Such a simple fix made the software much more reliable.

Synchronization would in some cases necessitate a reboot. If the vRTC to system clock offset was too large, simply setting the system clock was not an option, as it would cause serious system level problems. First, the software state was saved to phase change memory, then the satellite was rebooted. Once the satellite comes live, before any other processes are started, clksync sets the system clock to the vRTC. Then, the rest of the system processes are loaded and the satellite resumes proper functionality.

Testing/Verification

This process underwent many waves of testing, and is currently at a production level in terms of stability. The first testing phase consisted of micro testing during the development. The second testing phase was a complete software level test, aimed at rooting out some of the larger bugs. The third testing phase was a complete systems level test, aimed at tooting out any problems between clksync and the hardware environment it runs on. The final testing phase was a long term endurance test; this final test is what made some of the sneaky system stability breaking bugs more apparent.

The initial testing phase was broken into two parts: testing the development of the vRTC to system clock synchronization, and the ground clock to system clock synchronization. For both parts, I created a simple logging feature that printed out .csv lines to a file containing the clock values at each clock reading.

```
946685406.000000,  946685406.003369, 40
946685408.000000,  946685408.003384, 41
946685410.000000,  946685410.003290, 42
946685412.000000,  946685412.003303, 43
946685414.000000,  946685414.003393, 44
946685416.000000,  946685416.003569, 45
946685441.000000,  946685441.000038, 1
946685443.000000,  946685443.000038, 2
946685445.000000,  946685444.999979, 3
946685447.000000,  946685446.999949, 4
946685449.000000,  946685449.000205, 5
946685453.000000,  946685452.999949, 6
946685455.000000,  946685455.000117, 7
946685457.000000,  946685457.000192, 8
946685459.000000,  946685458.999968, 9
946685461.000000,  946685460.999933, 10
946685463.000000,  946685462.999930, 11
946685465.000000,  946685464.999962, 12
946685467.000000,  946685466.999981, 13
946685469.000000,  946685469.000213, 14
```

Table 1- Sample vRTC to system clock debugging data

The left column is vRTC readings in seconds, the middle column is system clock readings in seconds, and the right column is the double linked sample list depth count (see table 1). This output made it easy to generate graphs using excel or open office, and thus very easy to see what was going on under the hood. At various iterations of the development, the results were visualized, and all requirements were checked against.

Once the system's independent parts were functioning properly, they were combined into the first full version of the program. All log files were turned on, and a complete software test was started. The testing procedure was roughly as follows:

1. Clear all system state and log file data.
2. Boot the satellite, and observe that the system clock is properly set to the RTC.
3. Watch the vRTC to system clock readout file to make sure the system remained stable after a sufficient time period had elapsed (~5 minutes).
4. Synchronize the satellite to a desktop by using the time-packet functionality.
5. Observe the effects. The expected behavior is as follows:
   a. If the offset was > 2seconds, a reboot should occur. Otherwise the system should slowly synchronize.
   b. Post reboot, the system should come live with the system clock set to the desktops time.
   c. The vRTC to system clock synchronization should remain stable.
6. Let the system run for an extended period (~24 hours), and observe the collected data, verifying that the software meets requirements.

At a certain point, system level testing commenced. Initially, clksync upset the satellites software configuration because of unforeseen factors. The first problem that occurred was that the clock synchronization software would enter a high CPU utilization mode if the on-board RTC was missing or malfunctioning. The solution to this problem was to, upon opening the file descriptor of a broken RTC, and upon error, enter into an idle loop using sleep(60). This however caused more problems. There is a software watchdog process that periodically sends status request packets to all registered processes. While clksync was executing its sleep(60), it was unable to respond to status requests, and would cause the software watchdog to panic, rebooting the system. To fix this, I switched to using event based delays built on top of the select system call. This was very easy to implement since we already have a dynamic library that does this, and is shared among all Polysat flight software processes.

The last big problem that occurred was a mysterious reboot. At first I did not see a possible way that clksync could be rebooting our satellite, but the swarm of E-mails I was getting seemed to not agree. I eventually found an event where another on-board process, took a large chunk of CPU utilization to do some image processing. This was creating a few latent system calls, which fed into a frequency calculation between the vRTC and the system clock. The result was a bad frequency adjustment, which made the offset between the vRTC and the system clock sufficiently large for a reboot to occur. The fix to this was required to be minimal and 100% full proof since we were under large pressure from JPL to stabilize our software, and no more frightful events would be welcomed. I added a small switch that only allows reboots to occur after time-packets have been received from ground. This effectively reduces automation, but improves stability of the entire satellite.

Conclusion

This project greatly expanded my expertise with linux software development, improved my awareness of system stability concerns, and introduced me to processing real world data. This software is intended to be used in a future mission called EXOCUBE, where we will be implementing an attitude determination and control system that at its core requires a correct system clock. EXOCUBE will be using magnetometers and sun sensors to determine its position and orientation around earth. If the system clock becomes unsynchronized with real time, the satellite will think it’s in another location around earth, expect a different magnetic field or sun vector than what is observed, then likely spin out of control. That being said, development on clksync will not be finished until EXOCUBE leaves earth, and then possibly after that on future satellites.

Bibliography

[1] LXR community. (since 1995). Linux source code cross reference. *lxr.**linux**.no*.
http://lxr.linux.no/#linux+v2.6.30.2/

[2] Public domain (2013) NTP: The network time protocol.
http://www.ntp.org/.

[3] Wikipedia: The free encyclopedia (2013, June 7) Network time protocol
http://en.wikipedia.org/wiki/Network_Time_Protocol
vhj

[4] Dallas Semiconductors, “Extremely Accurate I2C-integrated RTC (DS3231)” datasheet, 2013.

[5] Polysat, (2013) Polysat homepage
http://polysat.calpoly.edu/

[6] Tyvak, (2013) Tyvak Nano-Satellite Systems LLC.
http://tyvak.com/

Appendices

Senior Project Analysis

Nano Satellite clock synchronization software

Student: Matt Zimmerer

Advisor: Bridget Benson

Summary of functional requirments:

This software is capable of synchronizing an orbiting satellite to a ground clock by using a radio link. The claimed synchronization precision is +- 100ms after synchronization has occurred. In addiction to synchronizing a distant clock, this software also disciplines a temperature intolerant system clock by referencing a thermally stable RTC.

Primary Constraints:

This software can not set the system clock, so alternate approaches to correcting offsets were sought after. In addition, this software needed to use a minimal amount of CPU resources, due to its high priority with the linux scheduler. This software also had the ability to reboot the system, but had do only do so deterministically, since system stability is of paramount concern. Real time data carried with it random variation, and this random variation needed to be dealt with for precise measurements. Lastly, this software operates on a satellite, so it must work the first time, everytime. It is very hard to reflash an orbiting satellite with bug fixes.

Economic:

Original estimated cost:

This software did not have any associated costs. It in fact lowered the cost of the host system by removing the need for a GPS unit.

Original development time:

This project was expected to take 1-2 months to complete. (started in March 2012)

Actual development time:

This project took 6 months to develop, and underwent additional testing (albeit sparse) for an additional 8 months.

If manufactured on a commercial base:

The software itself has no intention for commercial sale, and is Polysat proprietary property. Tyvak Nano-Satellites LLC currently is marketing this softwares host hardware system. Cost per flight board is $6,500, and a 1U cube sat containing the flight board is $45,000. This does not include the cost of launching a cubesat into space, where the cost / Kg averages at about 8,000.

Environmental

The only environmental concern is associated with this softwares host system. Orbital debre is growing in space, and will cause problems for future space exploration / infrastructure addition.

Manufacturability

The current cost of manufacturing the host hardware platform is high, but unknown to me. The system board is a multi layer board, with small trace sizes, and a number of ball grid array components that require expensive pick and placer machines to manufacture.

Sustainability

Sustaining this system is a periodic task. A ground crew must operate a groundstation (or autonomous software must be developed), and periodiodcally, the satellite must be sent synchronization packets. If this is not done, the satellites clock may drift from its high precision mark, but may still be useful if only coarse synchronization is required.

The design can be upgraded by spending more time improving the code. Currently, I believe the timepacket outlier filter is naïve, and can be vastly improved. Also, the synchronization system expects a number of packets within short duration of eachother. This can be improved by designing software that makes no assumption about when timepackets will be sent (no burst mode). This would reduce the amount of work the ground crew would have to do while maintaining the satellite.

Ethical

This software has no associated ethical concerns. It is intended to operate on an automomous platform, interfaces only with clocks, and has an extremely low network profile.

Social and political

Usually, a GPS would be used to get both location and time. This software replaces the current trend, and is causeing a few people within the NASA community to be a bit uneasy. This uneasiness can however, be corrected with a proper amount of testing.

Development

In developing this software, I learned many techiniques.

- Monitoring processor tick count to get the highest resolution timing data (early testing)
- Using a custom log file system to aid in the debugging of large software projects
- Using the Polysat process library event system API
- SVN software versioning software
- Lab switched to GIT, learned GIT
- Raw data filtering
- Fixed point overflow analysis