A Lightweight Open Source Command and Control Center and its Interface to Cubesat Flight Software

Patrick H. Stakem

Capitol Technology University, phstakem@captechu.edu

Johns Hopkins University, pstakem1@jhu.edu

Loyola University in Maryland, pstakem@loyola.edu

Guilherme Korol

guilhermekorol@hotmail.com

Pontifical University Catholic of Rio Grande do Sul,

Porte Alegre, Brazil.

Gabriel Augusto Gomes

gabrielaugustodm@gmail.com

University of Colorado, Colorado Springs.

Sao Paulo State Technical College (FATEC-BS)

Introduction

During a 2015 summer workshop for Cubesat Design, Engineering, and Operations at Capitol Technology University (Laurel, MD), a requirement arose for a lightweight command and control system for the Cubesats. This would implement all of the usual control center functions such as telemetry ingest, commanding, archiving, telemetry format conversion, trending, and alerts. The Cubesat onboard computer was a Raspberry Pi Model B. This was running a standard Linux image, and incorporated portions of GSFC Flight Software's cFS (Core Flight System). It interfaced with a variety of sensors, and was command-able. It was decided early on that the total desired solution would be open source. This was implemented using the COSMOS software from Ball Aerospace under Ubuntu Linux. The Apache webserver was also added. Using features of the COSMOS system, formatted and limit-checked telemetry were passed to the web server. This allowed the end user terminal to be laptops, tablets, or smartphones. In addition to its control center role, the software package allows for integration testing of the Cubesat Flatsat. An advantage of this approach is that the control center hardware and software is validated early. Different features of the same software is used in the two environments.

The system is planned to be used to support Capitol's 2018 Cubesat mission. Further additions will include a secure remote commanding capability, limits-violations text messages, and a “smart” limit checker, that acts as a virtual system engineer on console, deciding when human intervention is needed.

The Cubesat Architecture

A Cubesat is defined by a set of standards. The 1U cubesat is a cube, 10 cm on a side, with a mass less than 1.33 kilograms. For this project, we were working with non-flight units. The structure was 3-D printed in our lab. The configuration included a computer and several sensors, along with a battery. Wireless ethernet was used in testing, in conjunction with the lab infrastructure.

The flight computer was the Raspberry Pi 2 Model B+, using an ARM7 architecture. The 8.3 x 5.7 cm board hosts a 900 MHz 32-bit quad-core cpu, plus one gigabyte of SRAM, and various I/O interfaces. A microSD flash card is used for non-volatile storage. This can be 8 gigabytes or more. The unit weighs under 60 grams. About the only shortcoming of the architecture is the lack of a real-time clock, but this can be added with a small external board. Out configuration of the Pi had an external light sensor and a temperature sensor connected via the i^2c bus.

Although the Raspberry Pi is not designed to be Rad hard, it showed a surprisingly good radiation tolerance in tests (Ref. 3). It continued to operate through a dose of 150 krad(Si), with only the loss of USB connectivity.

The Pi runs linux. Usually this is a variation of the Debian distribution. The Pi can also support real time operating systems such as RTEMS, FreeRTOS, CentOS, and others, with the edition of the real-time clock.

Besides the standard Debian-based linux, our Pi had modules from NASA/GSFC's Core Flight Software installed. CFS is an Open Source product of NASA/GSFC Code 582, the Flight Software Branch. The Core Flight Software consists of a series of modules for various onboard functions. We implemented the TLMSystem.py, EventMessage.py, and CmdUtil.c. One of the authors (Korol) wrote a module Sensors.py to tie in the sensors.

Sensors.py is the heart of the onboard software. It is in charge of communication with the cFS tools, reads the sensors, sends data as telemetry, and interprets incoming commands from the Control Center. In order to send telemetry, Sensors.py uses cmdUtil.c, which is a cFS tool. Telemetry is constructed by receiving packets through another pair of cFS tools, TlmSystem.py and EventMessage.py to our module, Sensors.

The main script reads ambient light and temperature sensors connected to an external ADC chip, and an additional accelerometer sensor. All these data can be retrieved using commands such as ‘ALL’, which returns data from all sensors, ‘TMP’, returning only temperature data, ‘LGT’ for light, among others. More importantly, the script is structured in a way that makes the implementation of new commands and sensors easy. Some minor changes were made to the EventMessage script in

order to get it interfaced to sensors. The Core Flight Executive is a portable, platform independent embedded system framework developed by NASA Goddard Space Flight Center. This framework is used as the basis for the flight software for satellite data systems and instruments, but can be used on other general embedded systems. We did not implement the CFE, but only some applications from the cFS, core Flight Software.

CFE and cFS are released as open source under the NASA Open Source Agreement 1.3 (NOSA-1.3).

The Open Source Approach

All software both flight and ground used on this project was required to be open source, by self-imposed restrictions. It is not a technical topic, but concerns the right to use (and/or own, modify) software. It's about those software licenses you click to agree with, and never read. That's what the intellectual property lawyers are betting on.

Software and software tools are available in proprietary and open source versions. Open source software is free and widely available, and may be incorporated into your system. It is available under license, which generally says that you can use it, but derivative products must be made available under the same license. This presents a problem if it is mixed with purchased, licensed commercial software, or a level of exclusivity is required.

Adapting a commercial or open source operating system to a particular problem domain can be tricky. Usually, the commercial operating systems need to be used "as-is" and the source code is not available. The software can usually be configured between well-defined limits, but there will be no visibility of the internal workings. For the open source situation, there will be a multitude of source code modules and libraries that can be configured and customized, but the process is complex. The user can also write their own modules in this case.

Many Federal agencies have developed Open Source policies. NASA has created an open source license, the NASA Open Source Agreement (NOSA), to address these issues. The Open Source Initiative (www.opensource.org) maintains the definition of Open Source, and certifies licenses such as the NOSA.

The GNU General Public License (GPL) is the most widely used free software license. It guarantees end users the freedoms to use, study, share, copy, and modify the software. Software that ensures that these rights are retained is called free software. The license was originally written by Richard Stallman of the Free Software Foundation (FSF) for the GNU project in 1989. The GPL is a *copyleft* license, which means that derived works can only be distributed under the same license terms. This is in distinction to permissive free software licenses, of which the BSD licenses are the standard examples. Copyleft is in counterpoint to traditional copyright. Mixing proprietary software "poisons" free software, and cannot be included or integrated with it, without abandoned the GPL. The GPL covers the GNU/linux operating systems and most of the GNU/linux-based applications.

Open Source describes a collaborative environment for development and testing. Use of Open Source code carries with it an implied responsibility to "pay back" to the community. Open Source is not necessarily free.

A Vendor's software tools and operating system or application code is usually proprietary intellectual property. It is unusual to get the source code to examine, at least without binding legal documents and additional costs. Along with this, you do get vendor support.

The Open Source philosophy is sometimes at odds with the rigidized procedures evolved to ensure software performance and reliability. Offsetting this is the increased visibility into the internals of the software packages, and control over the entire software package. Application code can be open source. The programming language Python is open source. The popular web server Apache is also open source.

The Open Source software that we used included linux, the Core Flight System, and the Python language on the Pi. The support computer included Ubuntu linux, COSMOS, and the Apache web server.

ITAR

Systems that provide "satellite control software" are included under the *International Trafficking in Arms* (ITAR) regulation, as the software is defined as "munitions" subject to export control. The Department of State interprets and enforces ITAR regulations. It applies to items that might be transferred to non-US citizens, even citizens of friendly nations or NATO Partners. Even items received from Allies may not necessarily be provided back to them. Software and embedded systems related to launch vehicles and satellites are given particular scrutiny. The ITAR regulations date from the period of the Cold War with the Soviet Union. Increased enforcement of ITAR regulations recently have resulted in American market share in satellite technology declining. A license is required to export controlled technology. This includes passing technical information to a foreign national within the United States. Penalties of up to $100 million have been imposed for violations of the ITAR Regulations, and imprisonment is also possible. Something as simple as carrying ITAR information on a laptop or storage medium outside the US is considered a violation. ITAR regulations are complex, and need to be understood when working in areas of possible application. ITAR regulations apply to the hardware, software, and Intellectual Property assets, as well as test data and documentation. We verified with Ball Aerospace that the COSMOS product was not subject to ITAR regulations.

The Control Center Architecture

Our satellite control center was implemented in Capitol's Space Operations Institute (SOI). This facility has a high-speed direct link to NASA-GSFC, some 5 miles away. The facility is used to train student interns to be satellite operators at GSFC. It was established in 2002 with a NASA Grant, and has supported six on-orbit missions. It can serve as a backup or fall-back control center. It is based on networked pc architectures, as traditional consoles.

In the SOI, we implemented the COSMOS software on one pc. At the time, COSMOS was not available on linux, so the Windows-7 version was used. The COSMOS software remains open source.

Ball Aerospace did not yet have a Linux version of COSMOS, but indicated it should be feasible, as they had implemented it on the linux-based RTOS, CentOS. At this point, one of the author's (Stakem) ex-students from JHU, currently at NASA, volunteered to take this task on. He implemented the linux version of COSMOS, using the source code from Ball's website, over a weekend. Then, he posted the linux version back to their website. That's how open source is supposed to work.

We switched to the linux-hosted version, and implemented it on one of the Author's (Stakem) personal laptops. We also implemented the Apache web server.

Onboard, telemetry was generated by a module of CFS running on the Pi. We did not implement the full CCSDS protocol for this, due to time constraints. Rather, it was TCP/IP format, to keep things simple. A hardwired or wireless (wifi) connection could be used. Later, we can implement the CCSDS protocols, and encapsulate them in IP packets.

Telemetry & Command as a service.

Telemetry received by COSMOS is logged as binary, and goes through the Telemetry Extractor process, which produces a text file. It can then be viewed as packets, as extracted telemetry, engineering units, or graphed. There is also a limit-checking process that can be invoked on selected telemetry. This involves setting yellow and red limits for each selected telemetry point.

The link between COSMOS and Apache was solved by taking the Telemetry Extractor's file and converting it from text to HTML. These new files were accessible to the Apache web server. The data was put up on the lweb, and could be viewed by laptop, tablet, or smartphone.

Commanding of the Cubesat's onboard processor was also implemented. This used COSMOS's Command Sender process, to send a CCSDS-configured packet to the Pi's CFS Module command receiver. Here it was parsed and passed along to the Sensors.py routine. The commands were simple – enable/disable telemetry, select individual sensors, or “all.”

The actual control center machine was still located in the SOI room, which was now closed and dark.

COSMOS

The COSMOS software package consists of a series of applications that can control and monitor a wide range of embedded systems, including space flight systems. The reader is directed to the Ball website for more in depth information. Here, we will discuss those applications we used in this project. We specifically used the command sender, the packet viewer, the telemetry viewer, the telemetry grapher, the telemetry extractor, and Limit Checker.

The Command and Telemetry Server is the real-time hub of the COSMOS

system. It logs commands and telemetry, and provides the API for other tools to send commands, receive telemetry and other tasks. It can display the raw binary of the command and telemetry packets. It also performs limits monitoring of received telemetry. Interaction with the running COSMOS can be scripted, or realtime. Configuration files, pre-defined, provide the telemetry and command characteristics of the selected "target" system.

The Packet Viewer allows for viewing of selected formatted telemetry items in engineering units. The telemetry grapher provides realtime or off-line graphing of selected telemetry points. The data viewer provides a text-based data visualization function for such items as log files and memory dumps.

The Telemetry Extractor converts telemetry log files into (comma separated value) CSV files, to be fed to other tools such as relational data bases or analysis tools such as MatLab or Excel. The Script Runner functions allows the user to develop and execute test or operational procedures. The Limits Monitor shows graphically any and all telemetry points that exceed predefined yellow or red limits.

Another nice feature of COSMOS is the Handbook creator, which reads the COSMOS configuration files about a specific target, and produces HTML or PDF Command & Telemetry handbooks. Again, this automation removes several error sources in the production of this product.

COSMOS is implemented in Ruby, an open source, object-oriented language. COSMOS refers to the system it is communicating with as the "target." Thus, we defined the command and telemetry files for the target "Pi." We could support multiple targets with unique identifiers, but this feature was not implemented due to time constraints.

We found COSMOS to be well-documented and easy to use. When we did contact Ball for clarifications, we were provided prompt attention. Remember, this is for a "free" product.

This is an ongoing collaborative distributed project. We would be pleased to work with any interested individual or organization, and pass along our code.

Next steps include:

1. Implement limits-exceeded texting, directly to the appropriate individual's smartphone.
2. Incorporate Mission Team Group Meeting software.
3. Increase the capability of the Limits Monitor towards a "smart automated controller"
4. Implement automated learning, trending, and aging monitoring.
5. Develop additional cFS aps to merge with COSMOS.
6. Explore the cpu loading of simultaneous multi-targets.
7. Implement the control center as a virtual machine, and in the "Cloud."
8. Explore Control Center as a service.
9. Explore Android version of Cosmos, allowing for a Tablet host.

Download the code:

The COSMOS source code:
https://github.com/BallAerospace/COSMOS

The cFS source code:
http://sourceforge.net/projects/coreflightexec/

The CFE source code:
http://sourceforge.net/projects/coreflightexec/

The flight software image:

https://drive.google.com/file/d/0B_IgfVkdSVrNVENBczAtd08wME0/view

Main web pages:

CFS - https://cfs.gsfc.nasa.gov/

CFE - http://opensource.gsfc.nasa.gov/projects/cfe/index.php

COSMOS - www.cosmosrb.com

References:

1. Cudmore, Alan *NASA/GSFC's Flight Software Architecture: core Flight Executive and Core Flight System*, NASA/GSFC Code 582.
2. Fesq, Lorraine; Dvorak. Dan, "NASA's Software Architecture Review Board findings from the Review of GSFC's "core Flight Executive/Core Flight Software" (cFE/CFS), Fight Software Workshop, Nov 7-9, 2012, SWRI, San Antonio, TX.
3. Violette, Daniel P. "Arduino/Raspberry Pi: Hobbyist Hardware and Radiation Total Dose Degradation," 2014, presented at the EEE Parts for Small Missions Conference, NASA-GSFC, Greenbelt, MD, September 10-11, 2014.
4. Wilmot, Jonathan *Use of CCSDS File Delivery Protocol (CFDP) in NASA/GSFC's Flight Software Architecture: Core Flight Executive (CFE) and Core Flight System (cFS)*, NASA GSFC, Code 582.
5. *Core Flight System (cFS) Deployment Guide*, Ver 2.8, 9/30/2010, NASA/GSFC 582-2008-012.
6. Open Source Space, http://www.linuxjournal.com/content/open-source-space?page=0,0

Author's Biographies

Mr. Patrick H. Stakem

Mr. Stakem spent 44 years as a support contractor at every NASA Center, and specialized in flight software support. He teaches for Loyola University in Maryland, Graduate Department of Computer Science, Capitol Technology University, Electrical Engineering Department, and The Johns Hopkins University, Whiting School of Engineering, Graduate Engineering for Professionals Program. He has published numerous technical papers and books.

Mr. Guilherme Korol is a student at the Pontifical University Catholic of Rio Grande do Sul, Porte Alegre, Brazil. He received a scholarship from the Brazilian Scientific Mobility Program to attend the

University of Colorado at Denver, and the Summer 2015 Cubesat Engineering and Operations Program at Capital Technical University.

Mr. Gabriel Augusto Gomes is currently a Computer Science student at the University of Colorado at Colorado Springs. He has a scholarship from the Brazilian Scientific Mobility Program. He also received a scholarship from the same program to attend the Summer 2015 Cubesat Engineering and Operations Program at Capital Technical University. His home University is Sao Paulo State Technical College.