HawkEye 360's CI/CD Approach to Automatic On-Orbit Firmware Updates

Eric Haengel, Lorin Metzger
HawkEye 360
485 Spring Park Place STE 450, Herndon VA 20170; 1-734-358-2827
eric@he360.com

ABSTRACT

HawkEye 360 is the world leader in RF signal monitoring and analytics from space. Its constellation of more than two dozen small satellites, each with a GPS time-synchronized on-board Software Defined Radio, enables it to record and monitor terrestrial RF signals emitted from anywhere on Earth, multiple times a day. The satellites fly in clusters of three each, in a tight orbital formation, enabling trilateration of emitters by observed time and frequency difference of arrival. RF signals come in such a wide variety in their applications and content, that from early on the spacecraft design was structured with flexibility in mind. All processing elements, e.g. microprocessors, FPGAs, embedded Linux computers, in the payload can be re-programmed safely on-orbit, by leaning on bootloaders or two-string redundancy, should an issue occur. Furthermore, TCP/IP and UDP/IP support on the payload communications link enables a flexible workflow with standard Linux network tools, for uplinking new firmware binaries during communications passes. In the past year, HawkEye 360 implemented a Continuous Integration (CI) system with Gitlab CI tools to facilitate automated Software in the Loop (SIL) and Hardware in the Loop (HIL) testing of these binaries, on the equipment in its flatsat lab of equivalent payload hardware. A Continuous Deployment (CD) system based on Mender, a software framework for over-the-air firmware updates developed initially for IoT devices, was adopted to facilitate automated synchronization and installation of new firmware binaries on-orbit. This talk and accompanying paper will describe the development and implementation of this CI/CD system and the new on-orbit functionality it has unlocked for the company.

OVERVIEW

HawkEye 360's spacecraft are capable of monitoring RF signals over the range of roughly 70 MHz to 18 GHz. Not only that, but these spacecraft can also listen on multiple antennas and record/process the signals from those multiple antennas simultaneously. Each satellite can simply record the RF signals acquired, as timestamped I/Q sample files, and they can also process and derive analytics about the signal at the edge. Edge processing is enabled by the payload computer, which is currently a quad-core ARM processor and FPGA System on a Chip. The flexibility of the payload to perform a wide variety of tasks is intentional, and it's a feature that customers take advantage of.

Behind all of the hardware capability is an expansive ecosystem of embedded firmware and ground-segment software to support the range of tasks available on the spacecraft. Often, a customer asks the company to perform a specific task for which software is not yet implemented to support it. The company also has internal initiatives to develop new products and capabilities, requiring new software developments. As such, we are updating the embedded firmware on orbit, on all of the satellites, along with the ground software to support it, on a daily basis.

Challenges for On-Orbit updates

There are three categories of challenges for on-orbit spacecraft firmware updates. The first category is, the spacecraft's communication link must support the bandwidth needed to regularly uplink new firmware binaries to the vehicle, so that they can be then installed. This can be considered a hardware design requirement, but there is also a software consideration. For example, if a developer or operator uploads new firmware for the communications application that runs on the spacecraft, and if there is a bug in that new firmware that causes the spacecraft to no longer contact ground stations, that's a potential failure mode that also needs to be mitigated against.

The second category is that for many processors, any corruption of the boot firmware can lead that processor to become totally inoperable. Therefore, it's important to structure boot firmware for each processor in a way such that at least one of the following is true:

Case 1. The boot firmware is stored in a way that makes it generally unalterable and passively stable for the space environment.

Case 2. Multiple redundant copies of the boot firmware can be selected at boot time, so if one image is corrupted or has a bug, it can be repaired by first booting from another working image and then accessing the corrupted image.

Case 3. It's possible for an external processor to access and modify the boot firmware storage. In this case the external processor can be used to repair/upgrade the boot firmware if needed.

An example of a Case 1 solution is to store the boot firmware on a non-volatile storage medium known to operate effectively and retain its state in the space environment for as long as the mission requires. For chip components operating in space, Total Ionization Dosage (TID) and Single Event Effects (SEE) can cause both faults and state changes. There are some non-volatile storage architectures that have proven to be more resilient than others on this front. One such example is some Ferroelectric Random Access Memory (FRAM) chipsets [1], with a few others.

The third category of the challenge is, even if the spacecraft and all its embedded processors are in healthy operable condition, if a firmware update introduces a bug that causes a temporary mission outage, that can lead to lost revenue and even customers. The considerations for this are completely different from the first category, and are specific to each business, its customers, and the tolerance for risk for each party. Generally, the best scenario is to catch bugs on the ground with ground-based software and hardware testing. Beyond that, two approaches that can be effective are: the progressive roll-out of upgrades across the fleet, and the implementation of automated alerts and monitors for outages.

Communications System Architecture

HawkEye 360 implemente a modest 1-2 megabit per second payload uplink to complement its payload downlink, which is much faster. When both uplink and downlink are in active use over a ground station, the onboard payload computer supports transmission of Internet Protocol packets over the radio link. This enables developers and operators at the company to access the payload computer via standard Linux networking tools such as SSH and RSYNC. It also enables an entire ecosystem of open source tools for managing remote devices, as most such software tools ultimately expect to be able to communicate with the remote devices over sockets.

In addition to the payload communications system, the spacecraft bus implements a lower data rate command and control transceiver, which is always available for back-up access to the platform.

Processing Elements on HawkEye 360's Payload

Each satellite payload has two Software Defined Radios (SDRs), developed internally by the company. These SDRs are built around a quad-core ARM processor paired with an on-chip FPGA, sometimes called a System on a Chip (SoC). The reason for two-string redundancy on the SDR front is that it enables the company to rapidly iterate and develop state-of-the-art SDRs into the future. By always flying one known, flight-proven device plus a newly developed device, the company can mitigate the risk of flying an unproven device for a mission-critical application.

The embedded firmware considerations for the SoC are largely divided between the boot firmware and the application firmware. This device runs Open Embedded Linux, with u-Boot and the Xilinx First Stage bootloader (FSBL) managing the start-up sequence. There is also a boot FPGA image that is loaded at start-up. The combination of these piece parts can be considered a single boot image. Copies of these boot images reside on two redundant QSPI flash memory chips. Each chip is partitioned in half, for up to four redundant copies of the boot firmware for the SoC. The capacity for redundant boot images in QSPI flash memory is an important hardware consideration to enable safe reprogramming of the FSBL image, u-Boot image, boot FPGA image, and the Linux image itself.

The other hardware consideration that enables safe on-orbit reprogramming of the SoC boot image, is the implementation of a flash memory-based FPGA on the same circuit board. This secondary FPGA monitors the SoC boot sequence for reboot loops and failed boots, and it will automatically switch the SoC to boot from a different copy of the boot firmware should an issue occur. The secondary FPGA, together with the redundant QSPI flash memory structure for boot images, enables safe on-orbit re-programmability for the SoC boot firmware.

For non-boot firmware, the situation is a lot simpler. Some application firmware is baked into the Linux image itself, but much of it resides on a secondary non-volatile storage medium. For that, there are two redundant eMMC chips per SDR and, more recently a Solid State Drive per SDR. The general approach for application firmware, is to have multiple copies of the firmware images located on the secondary storage as a mount point in Linux. Symbolic links select which application image to use operationally, for each type of application run on the SDR.

In addition to the SoC, each payload implements either one or two 8051 series microprocessors, to command and control the payload hardware, e.g., the RF front-end and Low Noise Amplifier circuitry. This microprocessor implements a light-weight custom bootloader, that can be entered into by external command. This bootloader enables reprogramming of the 8051's application firmware. It can be entered into even if the application firmware is for some reason corrupted, enabling recovery against a bad application firmware load. This processor's program memory is stored on internal FRAM.

CONTINUOUS INTEGRATION FOR SMALL SATELLITES

Continuous Integration (CI) refers to regularly integrating source code changes, often from multiple developers working in parallel, to the main branch of a source code repository. It is one of the more common developer workflows in place today for terrestrial software development. Some published works indicate that other New Space companies, like HawkEye 360, have adopted a CI approach to on-orbit firmware updates [2]. That said, it appears to be a relatively new approach for the industry.

Our CI development workflow is essentially as follows:

1. Requirements for a new firmware capability are defined based on new hardware to be flown on an upcoming mission or on a new product initiative, and developers are assigned to the project.
2. The new firmware is generally developed on a branch of an existing source code repository and tested first by the developers on payload hardware expressly set aside for testing an development. These spacecraft payloads for test and development are colloquially referred as "flatsats."
3. When ready, the firmware is requested to be merged into the main branch of its source code repository. At this point, the firmware is required to pass many automated tests. These tests are all defined/constructed as jobs and pipelines utilizing Gitlab CI tools. Some of them occur automatically on a schedule, and some are initiated as needed by developers.
4. Some of these tests are software tests. For example, firmware that is ultimately meant for an ARM processor may be compiled first for x86 and run through a number of software-only tests on an x86 server.
5. Many of the tests are Hardware-in-the-Loop (HIL) tests. For these tests, the firmware is installed on the flatsats along with all other boot and application firmware for the spacecraft, and each flatsat is then tasked to perform a variety of standard operations. This is done to check that the new firmware has not negatively impacted legacy capabilities.
6. Should all software and hardware tests pass, the new firmware is merged into the main branch of its source code repository.
7. Depending on the urgency of the new feature, one or more new features may go through the same process before a release is officially tagged for that source code repository. Once tagged, the application firmware is packaged and placed in a centralized firmware artifact storage and is staged for release to the constellation.

AUTOMATING ON-ORBIT UPDATES

The other side of developing new software for the spacecraft, is deploying that new software to the constellation. We follow the principle that by deploying small software improvements more frequently it's easier to isolate bugs or issues to specific new developments, than if a larger and more substantial firmware update is rolled out all at once. As such, we follow a Continuous Deployment (CD) strategy to complement our CI approach. Firmware updates for the satellites are rolled out several times a day.

A CD process will look different for every company and their spacecraft's unique set of firmware. For this constellation, there are essentially three categories of firmware updates performed on the spacecraft regularly, and each has its own considerations:

1. Application software updates. This is the most straightforward type of update performed on the spacecraft. Generally, new images are uploaded over ground station communication passes, and placed in non-volatile storage on a Linux accessible mountpoint. A symbolic link is updated to point to the new release package, for a given application.
2. Linux updates. Occasionally, the bootloader (u-Boot/FSBL), the FPGA boot image, or the entire Linux root filesystem needs to be reinstalled. Generally, that whole collection of firmware artifacts is uplinked all at once, and installed in one shot. The installation process is to boot a different QSPI flash memory image than the one intended for upgrade. Once booted to Linux, the QSPI memory partition intended for upgrade is mounted, and the files are placed where they need to go.
3. 8051 microprocessor updates. One or two such devices per payload are accessible to the payload computer via Controller Area Network (CAN). It's also possible to direct CAN packets at this device directly from the command and

control radio. In normal conditions, the payload computers have a routine they can execute to upload a new application image to the 8051. The command and control radio serves as a backup means to program this device.

Earlier on in the company, when there were only a handful of spacecraft in orbit, it was possible to perform many of these updates by hand during a ground station payload communications pass. A developer or operator would log into the ground station, upload the firmware to the spacecraft with RSYNC, remotely connect to the payload computer over SSH, and then perform the update with a series of Linux shell commands or scripts. As the constellation has grown, this is no longer sustainable, and automation has been implemented to streamline the roll-out of updates.

We have adopted the Open Source Software (OSS) tool Mender for automation purposes. This software, initially meant for Internet of Things (IoT) devices, is essentially a general-purpose remote software update management tool. With Mender, one can specify firmware images to deploy, schedule those deployments, and receive feedback from the remote device about executing that installation. On the payload computer side, the Mender client calls an update script specified by the user, to execute the update. The protocol for the Mender server to communicate with the Mender client on the device is built on HTTPS. Like with any OSS, as a software tool, it's not completely without its issues, but it has been a successful tool for our needs to date.

SUMMARY

Our spacecraft payload can perform a wide variety of tasks in acquiring RF signals and analyzing them at the edge. As such, we are constantly innovating on the software on the vehicle, to support the evolving needs of our customers. As our constellation has grown, we have implemented a CI/CD approach to developing and managing the firmware on our spacecraft. Our CI approach is based on an automated system of Gitlab CI jobs and pipelines that perform software and HIL testing with our flatsats. Our CD approach utilizes Mender, an OSS tool for managing firmware on remote devices, to automate the deployment and execution of firmware upgrades.

This is made possible by a design architecture that enables the safe remote reprogramming of all processing elements on the payload, and the implementation of a standard Internet Protocol stack on top of our payload communication radio link.

ACKNOWLEDGMENTS

The authors would like to acknowledge all of the efforts by the space team at HawkEye 360 in fielding a constellation of highly capable spacecraft and implementing the system described in this paper. It truly has been a team effort that's been years in the making. The authors would also like to extend their gratitude to the company's customers and investors for their continued support over the years.

REFERENCES

1. Dahl, B. A., et al. "Radiation evaluation of ferroelectric random access memory embedded in 180nm cmos technology." *2015 IEEE Radiation Effects Data Workshop (REDW)*. IEEE, 2015.
2. Badshah, Akash, Natalie Morris, and Matthew Monson. "Over-The-Vacuum Update–Starlink's Approach for Reliably Upgrading Software on Thousands of Satellites." (2023).