Flight Software and Software-Driven Approaches to Small Satellite Networks

Robert Harvey

Contents

1 Introduction 296
2 Flight Software and Microsats 296
3 Role of the Flight Software Team 298
4 Flight Software Life Cycle 300
5 Flight Software and Processor Selection 303
6 Software Build System Architecture 307
  6.1 Build Process Inputs 307
  6.2 Build Process Artifacts 309
7 Flight Computer Software Design 312
8 Flight Software and System Design 315
  8.1 Satellite Safe Mode 316
  8.2 Atomic Configuration Updates 318
  8.3 On-Orbit Software Upgrades 319
  8.4 Accountability in Operations 320
  8.5 Security 322
9 Conclusion 328
10 Cross-References 328
References 329

Abstract

There is a growing market for satellites that fall into the “Microsat” and “Nanosat” classifications. Many of these satellites are designed and manufactured by small groups such as in academia, startups, or small incubator teams inside larger organizations. These environments tend to be fast-paced and will likely eschew traditional aerospace life cycles and design paradigms in favor of rapid prototyping, consumer electronics parts, and even on-orbit testing. Microsats

R. Harvey (✉)
Planet, Inc., San Francisco, CA, USA
e-mail: Rob.harvey@planet.com

J. N. Pelton (ed.), *Handbook of Small Satellites*,
https://doi.org/10.1007/978-3-030-36308-6_87

have somewhat different flight software implications and requirements than traditional satellites. This chapter discusses some of the flight software aspects of Microsats, along with design trades, processes, and the role of the flight software group in small organizations. Certain aspects of the flight software are called out for Microsats, including satellite safe modes, configuration updates, on-orbit software upgrades, and security. The flight software life cycle for Microsats is discussed in the context of a shifting and multiple-launch schedule. The intent of the chapter is to lay out guidelines for new flight software engineers such that while building out new Microsats, they also lay the groundwork for launching their product at scale.

Keywords

Planet · FSW · Flight software · Satellite telemetry · Satellite constellations · Cubesat, Microsat

1 Introduction

At the time of this writing, Planet Labs Inc. ("Planet") operates one of the largest satellite constellations in history (chapter ▶ "Planet's Dove Satellite Constellation"). Planet's approach to satellite design and operation is a departure from traditional methods, including a focus on using consumer-off-the-shelf (COTS) components, leveraging the Cubesat form factor and launching early and often with primary and secondary payload opportunities. In the beginning, this approach to satellites was largely unverified and certainly not applied to scale. This novel approach has implications for the flight software (FSW) and the flight software engineer, not least of which is that the flight software engineer may be working on an aggressively small team and the lines related to traditional functional roles may be blurred. The intent of the chapter is to lay out the challenges that can face new flight software engineers when working toward Microsat missions and provide a way of thinking that can ensure mission success for Microsat designs that may eventually be launched at scale. The content that follows is based on the author's experience at Planet; it is not based on any extended experience in the traditional aerospace fields. This does not reflect the opinions of Planet Labs.

2 Flight Software and Microsats

Is flight software for Microsats different than flight software for traditional satellites? There is an argument that Microsats are inherently the same as other satellites (although generally smaller) and therefore the flight software should have the same properties as any previous flight software system, regardless of design heritage. There is assuredly much truth in this. Microsat flight software systems must address power, thermal, and guidance concerns. They are responsible for telemetry and system limit checking, as well as enabling the payload to execute on its mission. There is a vast heritage of documentation and design for traditional flight software that should be considered and potentially used for any Microsat mission.

However, these similarities are most pronounced when the flight software is considered as a decomposed system, looking internally at the functional blocks inside the flight computer itself. Taking a step back, some differences start to become apparent: items such as shorter development life cycle, the blurred role of the flight software engineer, the global software ecosystem for the mission, the abbreviated and sometimes opportunistic launch schedule, the tight integration of subsystems, and the contract manufacturing process. These items will create a different environment for the flight software engineer than traditional satellite design.

For a fleet of Microsats, the differences become even clearer. In this model it is likely that iterative hardware design is taking place and every launch opportunity diversifies the satellite hardware in orbit, which must be handled by flight software. This is magnified by "tech demos" and the desire to test features in space whenever possible (definitely not traditional). In the case that many identical satellites are launched at once, the reality is that Microsats degrade at a faster rate and have less redundancy which means that the functional capabilities degrade and become diversified over time across the constellation (months to several years). This also must be addressed in flight software.

The concept of operations for Microsats is also different. In a fast-paced launch environment and especially with tech demos, it is likely that flight software is not completely ready prior to launch. The reality in this case is that on-orbit software updates must be common, reliable, and generally not exciting. It is also likely that FSW resets may be common and may even be the preferred way to achieve a known state in the satellite. This is a far cry from very expensive satellites where a reset may be decided by committee and can result in expensive downtime.

The methodology for building Microsats is also different from regular (and more costly) satellites. Traditional aerospace design will use a tremendous waterfall life cycle, from requirements down through testing and deployment. The waterfall life cycle itself is often a large living artifact, being copied from program to program and tweaked as appropriate for dates and details of the new program. This development process is intended to ensure contract value to the customer, enforce milestones for product maturity, ensure compatibility between components developed by different vendors, and keep complicated projects on track. In contrast, the reality for Microsats is that they tend to be built in agile and limited resource environments such as academia, space startups, and even small incubator groups in much larger organizations. The designs are less complicated and the risk posture is completely different. Requirements are often eschewed for rapid prototyping, and design documentation is often replaced with whiteboard conversations. The feeling of moving rapidly motivates the (small) team and often is completely necessary to meet an aggressive launch date with a new product design.

It is unfortunately the case that aggressive timelines tend to erode standard SW engineering practices. There could also be pressure to deliver an initial (suboptimal) working design in order to secure funding. The resulting pitfall for Microsat projects unfortunately is that you will eventually be successful and you will pay for your shortcuts later. There is then an appropriate balance in Microsat development methodology that is somewhat difficult to achieve, namely, to be lightweight, fast, robust, and future-proof.

Regarding whether Microsat flight software is different than traditional flight software, Microsat FSW components may be very similar to regular satellite programs, but the path to achieving mission success with Microsat FSW can be quite different. Being aware of the above points, especially early in the development cycle, can contribute to initial mission success, long-term company success, as well as a smoother running organization. Some of these points will be discussed as appropriate in the rest of this chapter.

3 Role of the Flight Software Team

In the lightweight planning world of Microsats, it is insightful to consider the reality facing the flight software team. In small team environments, team members will likely be stretched across multiple disciplines, and the "flight software team" could be a somewhat loosely defined organization. These members of the flight software team could have diverse experience with respect to developing software, sometimes being non-SW subject matter experts with some coding background. It should be noted that the earlier the team subscribes to standard software engineering practices, the more robust the system will be to on-orbit testing, modification, new hires, and feature requests. Standard practices would include coding for unit test, modular design, re-factoring when needed, consistency in coding standards, and continuous integration (CI) methods.

It is very useful to define the scope of work for the flight software team as early as possible. Microsats are limited in volume, and it is often the case that much of the avionics must be compressed into a few printed circuit boards in order to provide room for the payload. This generally limits the ability to buy pre-built subsystems which can be integrated into the satellite bus. The result is that many subsystems end up being homegrown, and inevitably they require some form of software. Consider Fig. 1 which shows a somewhat generic diagram for a Microsat system.

The satellite block on the left contains the avionics (top) and the payload (bottom). The avionics are controlled by a flight control computer (FC) which must deal with power, attitude control, GPS, and the tracking, telemetry, and control (TT&C) radio. Note that many of the blocks are indicated with their own Microcontroller (uC). The payload as shown has its own processor (payload computer) and is shown integrated with a high-speed radio and the payload sensor through a field-programmable gate array (FPGA). The flight control computer is the device first thought of when discussing the responsibilities of the flight software team. This is also the device that will have a functional block representation that is similar to flight computers pretty much everywhere. As can be realized quite rapidly when looking at the diagram, there are a multitude of other processors besides the FC that could require software, and this can easily fall into the flight software group

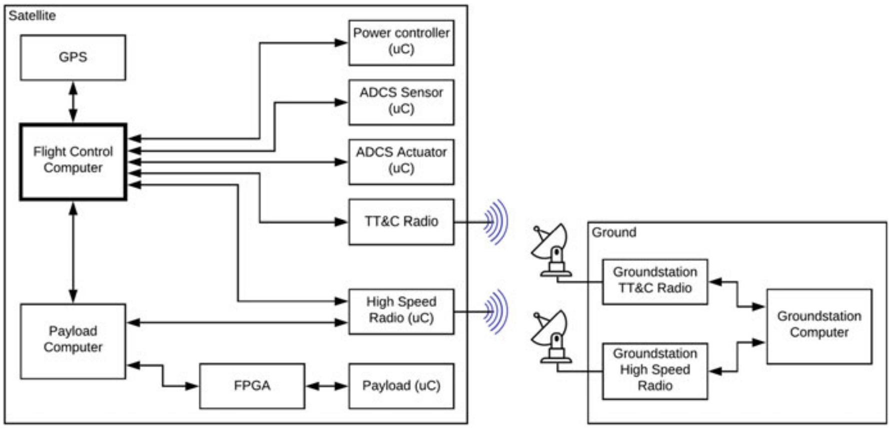

Fig. 1 Example satellite and ground system. (Image © 2019 Planet Labs Inc., licensed for one-time publication)

purview (every block in the diagram could be an independent software project). There are some things that are less obvious but still impactful to FSW. All of the devices with software must be software upgradeable post-launch. This includes devices that have third-party software on them! So even without primary responsibility for device software, there is still a requirement to develop a way to upgrade software on that device.

It might seem that the payload system is developed by the payload team in isolation, but this is not the case. There are likely to be common device drivers (especially for sensors and I/O) and also common telemetry, message protocol, and transfer routines. In some cases, the Microsat design does not have a clean boundary between payload and avionics, and the various software components need to coexist on a processor. This places a requirement on the flight software team to build portable, robust code and to possibly consider multiple processor architectures when architecting the code layout, repository, and build system. It may also be an opportunity to speak with the payload engineers about coordinating software engineering practices.

It is also the case that FSW responsibility may extend to the ground system (GS). If custom radios are being used, the code might be similar between the satellite and the ground, or maybe the communication link is the proprietary technology being developed. In the case that the GS is from a third party, it is still likely that communication libraries for message serialization/deserialization and telemetry ingestion will have common components with the satellite FSW.

There is another reality for the flight software engineer that must be considered. The tools to test out the satellite at the board level and full-build level do not write themselves. In a very small team, sometimes the best way to test out the hardware is with the actual flight software, and who better to write the interface and testing tools than the flight software team? This is not a bad scenario, but it needs to be taken into account, and it needs an owner (flight software team or not). It should be noted that it is rare for the on-orbit control procedures to map well to test procedures which implies care in separating one from the other. If there is a dedicated test team available, there may still be a FSW role to develop software components to interact with the satellite, even if the test code is decoupled from it.

The above paradigms may seem daunting for a small team. In a completely custom Microsat design, the processor and FPGA count can easily hit a dozen. Software work may not be confined to just satellite components. Early knowledge of this should be considered an opportunity:

- To ensure that all work is considered in the schedule
- To check the work against available staff
- To drive interconnect design, code architecture, and code reusability
- To revisit subsystems that might be available off the shelf

The important takeaway here is that the FSW role is not confined to the "flight computer": there is a larger software ecosystem that can include multiple processors, multiple systems, and multiple disciplines. In a lightweight planning environment, the details of all this work may not be captured, and not accounting for this work can lead to a stressful environment.

4 Flight Software Life Cycle

Planet has launched satellites on more types of rockets than almost any commercial company (chapter ► "Planet's Dove Satellite Constellation"). The reality is that the life cycle for any given Microsat iteration (and launch) will be overlapping with other launches, sometimes aggressively so. This obviously impacts not only flight software but other development groups as well. Figure 2 shows a single mission life cycle on the top and then multiple instances of this overlapped for three launches at the bottom.

At a high level, the top sequence is similar to any satellite program. A Microsat program will of course have differences: the requirements phase may be abbreviated, the day-to-day planning may be laid out using established Agile methodologies (likely more familiar to the technical team), the integration and test phase may be limited by access to flight simulation capabilities, and it is almost for certain that the overall time frame will be significantly shorter.

An impactful time for flight software (and everyone else) is when the first printed circuit boards for flight hardware arrive in-house (start of board verification). There is a collision of disciplines who will want to check out the boards. Flight software will presumably have been developing software on development boards or possibly non-form factor in-house boards. Electrical engineering will first do initial board bringup for smoke test, power rails, etc. Almost immediately after this, there will be the request to "talk" to the board (assuming it has a processor on it). Does the

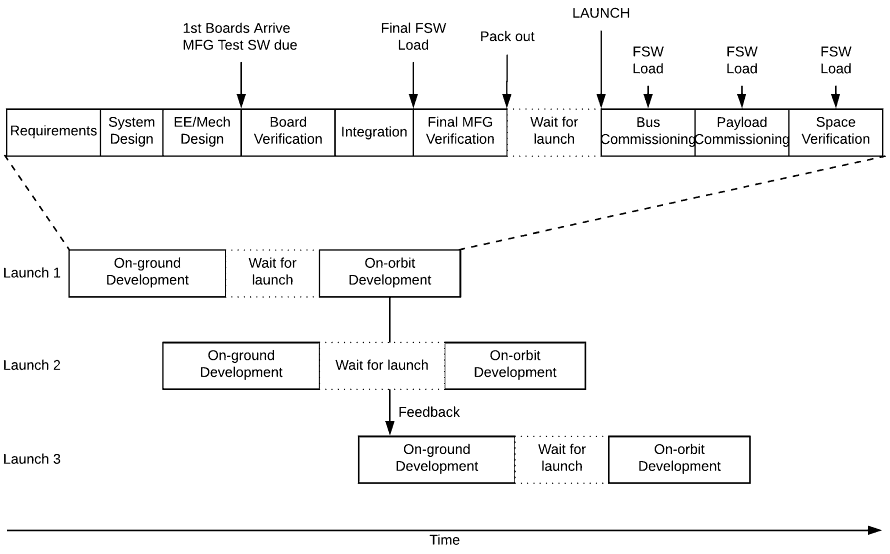

Fig. 2 Flight software life cycle. (Image © 2020 Planet Labs Inc., licensed for one-time publication)

software exist to do this? Be very mindful of this period in the life cycle, and consider carefully what the expectation will be from a software standpoint and then plan for it. This includes both satellite software and test harness software. Is the flight software team responsible for both? How can the flight software be ready if this is the first time that flight hardware has been available? In many cases the flight software needs specific flight hardware in order to be written, especially for drivers. The flight software engineer should also be cognizant of critical path timing not accounting for development work on the actual flight hardware. Note also that there is a distinction between test software and full flight software support for a given device or system on the satellite. A device or subsystem may be identified as not requiring full software support at launch, but the hardware will have to be validated on the ground regardless.

The launch date (or more accurately the pack-out date) will be a huge driving factor in schedule. The Microsat or possibly small fleet of Microsats will likely be a secondary payload for the targeted launch meaning that the actual launch date will be determined by a third party and potentially occur at a point which creates an overly aggressive schedule. The Microsat team will have to meet the payload integrator's window for delivery which may mean prioritizing development to ensure that critical systems are complete and robust while other systems are pushed off to a future on-orbit flight software or payload software upgrade. This may strike some readers as a disastrous or at the least irresponsible situation. How could one possibly launch a satellite without a full software suite that has been rigorously tested on flight hardware on the ground? This is the reality of Microsats.

The goal of any flight software team should be to optimize around this reality. This will involve a heavy focus on the on-orbit flight software update design including a bootloader and security capabilities as well as prioritizing the truly mission-critical components as early as possible (see section "Satellite Safe Mode").

Once pack-out has occurred, the flight software team will likely be working with the satellite operations group to prepare for new concept-of-operations (con-ops) procedures. There will also be ongoing development for flight components that have not had full software support implemented yet (potentially newer sensors or payload). Once launch has occurred, there will be possible flight software upgrades, for issues related to bus commissioning, payload commissioning and then infant mortality, experience with slow decay of components and workarounds, etc. Note that any schedule for the program should include work after the launch and not just terminate at launch. This is important when multiple launches are in play, discussed next.

There is an argument to say that many of the above issues are just schedule related. If there is time to do everything correctly, then the satellite and software will be fully featured, and you will be done at pack-out. Except for on-orbit component failures, this could possibly be true for a launch or two. In a growing company which is iterating on satellite design, building out on-orbit capability, and trying to maintain service-level agreements (SLAs), this is most assuredly not true. Consider the bottom part of Fig. 2. Multiple launches are occurring in order to prove out hardware design and meet SLAs, on-orbit and on-ground development is overlapped, and many iterations of hardware are in play. Consider that critical feedback from an on-orbit satellite design becomes available in the middle of board verification for a future launch and new boards must be created to account for it (note that time frames are shown such that the feedback can skip a launch). This is new, unplanned work for multiple teams, but the schedule is fixed by the third-party launch date. Possibly there is a critical problem with an on-orbit satellite component that must be addressed immediately by the FSW team, but there is also an ongoing and critical board bringup underway. The on-orbit issue will take precedence, especially if the mission is jeopardized. Or potentially a launch date happens to occur just before the pack-out for a subsequent launch, meaning that there is a requirement to support on-orbit commissioning while at the same time the on-ground manufacturing phase is at a critical juncture.

The point is that multiple-launch schedules can interact with each other in somewhat unpredictable ways, and the time to be allotted for development is hard to calculate. With vagaries in launch dates and idiosyncrasies in Microsat hardware, it is rare for software development work to be complete at pack-out even if the schedule is closely monitored, unless there are multiple teams available to handle the different phases in the various ongoing life cycles (unlikely in a small company). Being able to decouple software development work from the launch schedule is actually a very important opportunity. Planning out a robust system architecture with emphasis on mission safety and on-orbit flight upgrades will help enable this.

5 Flight Software and Processor Selection

In the Microsat development process, the flight software team should be participating in design discussions related to processor and sensor selection. Anything related to software should be considered from the standpoint of software toolchains, open-source versus closed-source code, space heritage, code reusability, and complexity. There are also questions related to processor load and memory sizing that the FSW team should be involved with. Note that when viewed as a software development problem, a Microsat project has much in common with other systems which are operated remotely and need to be robust against power loss and possible frequent software updates. This includes Internet of Things (IoT) projects and even some automotive projects. Much of the following will look familiar to people who are experienced with those systems.

Any processor selection should be judged against a list of requirements. The requirements should be carefully selected to meet the mission goals and maintain mission safety. In many cases, there is no obvious choice even after the requirements have been assessed. This means judging which requirements are most favorable to the teams and mission. Selection of a particular processor can also be intertwined with other devices, such as sensors (which might support a particular bus). Figuring out the requirements for processors can be somewhat daunting; the following bullet items lay out some informational groundwork for the process:

- **Space heritage** – Electronic devices which have already been proven to work well in space are ideal choices if they are available and meet the mission requirements. This can reduce early and unexpected mortality for Microsats. It can also reduce the cost of performing radiation testing on critical devices. The difficulty with space heritage parts is that they are either very expensive or difficult to identify because of export restrictions and a lack of public documentation. NASA has a published list of radiation tested parts that can be interesting to look over (NASA n.d.). Note that "space heritage" is sometimes blindly assumed to be a positive thing, but it is possible that a device has space heritage but performed poorly.
- **Off-the-shelf software** – Selecting parts that have readily available third-party software that can be applied to the mission can save time and allow the flight software team to focus on other flight software projects for the satellite. As mentioned in **Role of the Flight Software Team**, there will always be multiple software targets in the spacecraft. The organizations behind the off-the-shelf software presumably have already spent time debugging and verifying the software and possibly also time validating the associated hardware component(s). There are possible difficulties here, the first of which is that the third-party software may only be sold or compatible with hardware which may not fit in the satellite (Microsats tend to have a very constrained volume, and the payload generally dominates with the avionics packed around it). The second is that often only a particular component of the third-party software is desired, but it is very difficult to decouple from the complete system or the underlying protocols. Often this frustration will just lead designers to create the software "in-house."

- **COTS, or not** – There are radiation-hardened and radiation-tolerant electronic parts that may be worth considering, especially in the safety-critical domain. These will tend to be much more expensive and generally of a much older electronics vintage (less capable) than regular consumer-off-the-shelf (COTS) parts. In Microsats, it may be possible to use a radiation-hardened MCU or FPGA, but it will be unlikely to be able to use a full rad-hardened subsystem due to cost and bulk. As Microsats will generally be majority COTS based anyway, this option may not be useful, but it is worth considering. If there is a safety-critical area where a rad-tolerant part is being considered, it may be useful to consider some of the safety-critical automotive MCUs and components that are now available. Note that the effect of radiation on COTS components is an intense area of study (Sinclair and Dyer 2013) and may require its own team.
- **Common build systems** – There will be multiple processors in the design, with the possibility of multiple manufacturers, different CPU cores, and different build systems and environments. It is very advantageous to limit the number of build tools and environments that are required to build all of the software targets for the satellite. This should only be done as appropriate, but it is much better to select a single, slightly suboptimal solution for multiple use cases and then to try and drive down on optimal and unique processors at every opportunity. Note that this is a “soft” requirement, in that it is intended to allow for smoother development, higher efficiency across the team, and possibly a lighter load on the team when it comes to supporting build and artifact publishing infrastructure. Over the long term, this can be a very appropriate choice.
- **Debug capabilities** – The software team should understand the debug options available for the processors that are being selected. At the least, this should include JTAG support which can be used for debugging as well as boundary scan verification during manufacturing. There may be other debug options that are desired, potentially some kind of embedded trace module. The debug options for very high-speed devices and memory could be more complex, but this should be considered when choosing these devices.
- **Board support packages (BSP)** – A significant portion of the work in flight software is in writing drivers. Processors which have a comprehensive board support package for a board that might have similar characteristics to the flight hardware can be very beneficial. Note that there are two levels for drivers, there are the peripherals that are pinned out of the chip (UART, I2C, ADC, etc.), and then there are the drivers for all the sensors and actuators that are connected to the processor. The availability of a hardware abstraction layer (HAL) is also beneficial since this can make porting between operating systems and processors easier (see section “Flight Computer Software Design”).
- **Processor speed** – It is necessary to select a processor that is powerful enough to execute all its required functions in a timely manner. This will be a function of the processor’s clock speed, as well as memory access timing and any hardware acceleration that is relevant (floating point units, cryptographic cores, DSP instructions, etc.). It is highly valuable to run processor intensive algorithms on third-party development boards to try and benchmark throughput requirements

for possible processor candidates. Also be aware that repeated access to devices on a communication bus, and the addition of new devices, can add up and impact available throughput which can be aggravated when sensors are unavailable (failed or disconnected) and device access time-outs start accumulating. This can cascade and cause throughput problems if not handled appropriately. Once there is an idea of the computation-intensive operations and the bus operations, the processor requirement should probably be increased 50–100% for future growth. Note that the above statements are more relevant for the flight computer and processors which are more embedded in nature. The payload compute system may have a different set of requirements, likely involving fast read/write memory and bus support. Also be aware of any circumstances which may require the control loops to run at a faster rate. This will require more processor capability.

- **Peripheral and bus options** – An audit of the available processor pins should be completed. This is more of an electrical engineering (EE) and system design role, but it is worth mentioning. The iterative nature of Microsats is generally accumulative, in that devices and capability will likely be added as opposed to swapped. This will require more general-purpose I/O (GPIO) pins and more traffic and addressable parts on the various communication buses. Do not select a processor that is immediately maxed out for peripheral capability and bus I/O. It is also useful to try and limit the types of communication buses that need to be supported, regardless of their availability on a given processor.
- **Open source versus closed source** – Buying closed-source code from a vendor may be an option for accelerating development. It may also meet some requirement for reliability or real-time responsiveness. Note that there is a difference between buying source code from a third party and compiled code (libraries). Not having the actual source code can make it difficult to debug and fix critical issues in a timely manner. Source code from a third party may come with build environment and seat licensing issues that can complicate development and notably also server-side continuous integration. Open-source code is becomingly an increasingly valid method for building out Microsat capabilities and systems.
- **Avionics versus payload** – It should be recognized that the requirements for the payload computer and the flight computer are different. Although it might be initially desirable to combine the two functions into a single processor, it may be better to keep them separated and optimized for each function. The flight computer must be reliable and continuously active. It is directly responsible for satellite safety and this should not be compromised. The payload computer could be shut off for extended periods and reset if there is any detected problem. An error in the payload memory will not cause the satellite to spin up or generally cause harmful behavior, especially if the flight computer is monitoring telemetry from the payload system and can take appropriate action. The avionics will likely require a less complex processor than the payload which is easier to make robust. It is also undesirable to impact the avionics system with a payload software change. There are other differences which are addressed in the next bullets.

- **Internal versus external memory** – Larger processors will tend to have more complicated memory which is located externally to the chip. It may be possible to select a smaller processor for the flight computer that has some form of internal memory that would be more robust to layout and manufacturing issues. The payload system will likely have much different memory requirements, including very fast read/write access and very large data storage requirements. It should be noted that large, fast, and complex memory devices such as solid-state drives (SSDs) are much more prone to failure than simpler flash implementations (Costenaro et al. 2015). It is important to realize that a memory device that is chosen for high-speed data collection and large storage capability may not be an appropriate place to have operating system or program memory. Note that complex memory devices may be performing maintenance operations under the hood with their own microcontroller (such as wear leveling). This will have its own failure modes and again could be less robust than a simple flash implementation.
- **Error-correcting code (ECC) capability** – There are many COTS processors and memory devices with ECC. These devices are generally able to correct for a single bit flip and detect a double bit flip. This type of technology can protect against single-event upsets (SEU) which are one likely type of radiation damage expected in low earth orbit (LEO) (Sinclair and Dyer 2013). Note that this is particularly desirable for devices which are continuously on where boot CRC checks and a reloading of random-access memory (RAM) are not taking place. Some automotive safety-oriented processors will also be able to check memory in I/O registers and throughout the memory pipeline, not just RAM and flash. The use of ECC may not make the system "radiation-tolerant," but it is a readily available technology that can mitigate these kinds of errors, and it should be incorporated where possible.
- **Operating system (OS) selection** – Processor selection will be tied to the operating system selection. In most cases, this will be a real-time OS (RTOS). The flight software team should be comfortable with the RTOS capabilities and maturity with respect to the processor. Although possible, it is generally not a good idea to start off a new program with a port of an RTOS to a new processor. Certain processors may force a more limited set of RTOS options onto the team which must be weighed against the perceived benefits. If multiple processors require an RTOS, it will be beneficial to select the same RTOS as appropriate. Note that the payload system may benefit from a different OS, maybe even an OS without full RT support such as Linux. This might enable a better development experience on the ground for the payload team and provide a much more feature rich environment that is similar to a laptop or desktop. For limited processors, it may make sense to forgo an OS entirely and generate a bare-metal implementation. This can initially be simpler and faster to get to a testable product. However, care should be taken as bare-metal projects tend to suffer from a lack of software structure and code entropy can take its toll as multiple developers work on the program. There is also an assumption that a tasking infrastructure is not needed which can become problematic as more sensors are added and delays start to pile

up in the processing chain. In many cases, a bare-metal decision may just be putting work off to a later date when the code will need to be re-factored and potentially rewritten for an RTOS.

The above talking points will hopefully be useful but are not comprehensive. For any device selection, there are the standard questions about end-of-life (EOL) time frame, supply chain, second source, environmental limits, and process technology. There may also be the need to radiation test critical devices. These are issues for the larger team.

The above conversation is focused on processors, but other devices (such as sensors) which must be accessed through software should also be evaluated for software impact. It is appropriate to standardize the possible sensors in the satellite design. It should not be the case that there are two different kinds of temperature sensors on different boards that could possibly be the same device. This doubles the software work and will siphon off time that could be spent on other tasks.

When assessing processors and devices, the flight software team should always remember to advocate for homogeneity where possible because this can have a direct impact on the flight software team workload. Also be careful of selecting a processor for a very specific reason such as an embedded security feature. These decisions can influence the general software architecture and may make porting to new processors very difficult.

6 Software Build System Architecture

As mentioned in previous chapters, the flight software team will likely be responsible for multiple processors on board the spacecraft with each requiring a different software image. They may also be responsible for ground software and/or dictionaries that are used in ground systems communicating with the spacecraft. There will also be multiple generations of spacecraft that will need to be supported, which will likely require different software or configuration. This is a difficult situation to manage from a build and release standpoint, and defining an appropriate software build architecture early on in development can be very helpful. Figure 3 demonstrates a possible build architecture that can support multiple platforms over multiple generations. On the left are the inputs to the build system; build artifacts are generated as the flow moves down and to the right. The first takeaway is that the inputs on the left are likely managed by different teams which means that a process needs to be negotiated between teams that allows for smooth integration of new information (data format, release management, release notes, etc.).

6.1 Build Process Inputs

The following bullet points define the input blocks on the right side of Fig. 3:

- **Satellite HW Specification Database** – The electrical engineering (EE) team will be designing multiple boards for each satellite. As new generations of

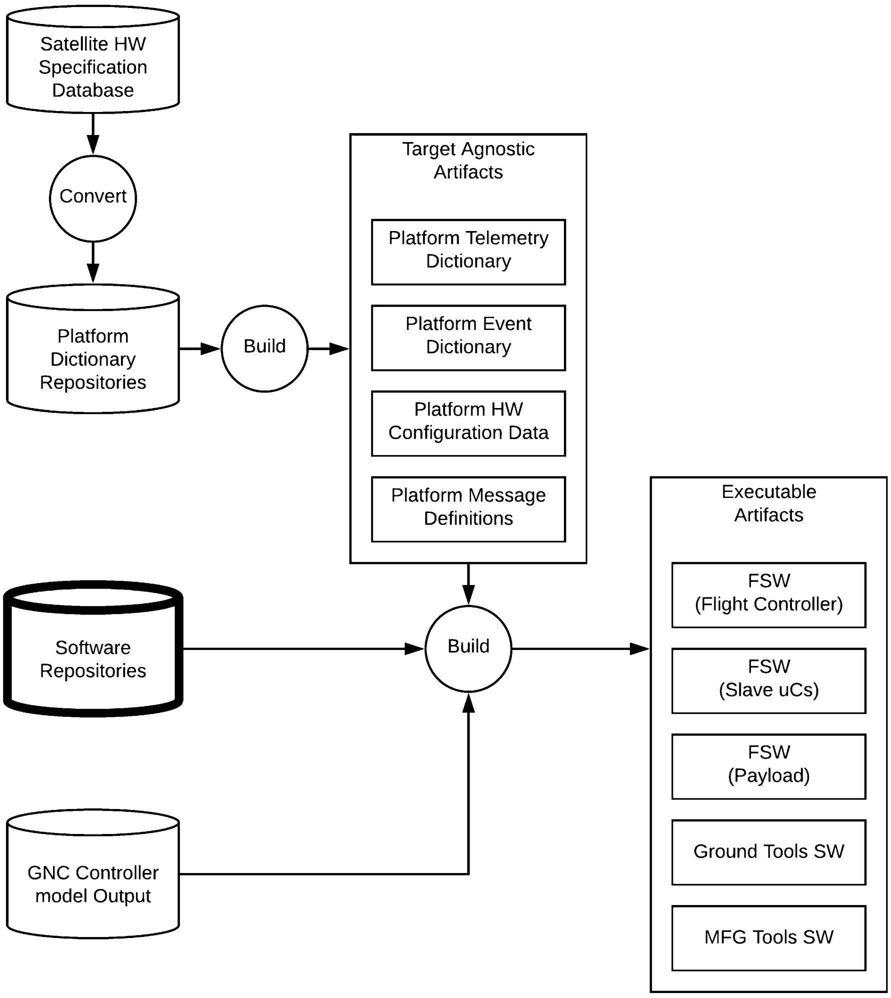
Satellite HW Specification Database
Convert
Platform Dictionary Repositories
Build
Target Agnostic Artifacts
Platform Telemetry Dictionary
Platform Event Dictionary
Platform HW Configuration Data
Platform Message Definitions
Software Repositories
Build
Executable Artifacts
FSW (Flight Controller)
FSW (Slave uCs)
FSW (Payload)
Ground Tools SW
MFG Tools SW
GNC Controller model Output

Fig. 3 Flight software build architecture. (Image © 2011 Planet Labs Inc., licensed for one-time publication)

satellites are created, the board designs will change, and this information must be transferred to the software team in some kind of reliable way. In order to fully describe what the software must do, this database can be very detailed. It should contain the list of all processor GPIOs and their functions, topology for communication buses and device addresses on the buses, a list of sensor devices (make and model), mapping of power rails to devices, power rail control information, and even resistor values where appropriate for sensor reads. This documentation will likely be contained in a format that is easy to integrate and use with EE tools. It will have to be exported to the platform dictionary repository. The necessity of this information being maintained correctly and at this level of detail is often noted when the team must go back in time and investigate older designs that may be having on-orbit issues. Note that a documentation system that does not allow easy detection of small changes will be hard to use in any event.

- **Platform Dictionary Repositories** – Each satellite variant (platform) will have a unique set of characteristics that should be maintained in the platform dictionary repository. This will include the definition (configuration) of the satellite hardware, imported from the EE HW specification database as shown in Fig. 3. Each platform will also have a unique set of telemetry channels and event descriptions that must be described and maintained. The intent behind the platform dictionary is to have a code-agnostic, single source of truth for the platform information. This information can have its own set of tools, rules, and schema to ensure that the data is consistent. It should be possible to generate the platform information for each satellite variant at any time. This system can also be used to define protocols and message structure.
- **Software Repositories** – The software repositories contain code, build instructions, and other collateral that is required to build the flight software and potentially other software as well. This may be one or multiple repositories depending on how the flight software team structures the code base. The software repositories are combined with the output of the platform dictionaries at build time to create the executables that can be loaded to the various processors in the satellite ecosystem.
- **GNC Controller Model Output** – A common model in satellite design is to use a third-party tool to build all the guidance, navigation, and control (GNC) models and then export the models as C code which can be integrated into the flight software. The GNC design will generally be owned by the GNC group and not FSW. The integration of the auto-generated code and the FSW must be well understood, since the inputs to the guidance algorithms must be well-conditioned, correspond to the right frame of reference, and have the right units. The output must also be understood so that actuators can be driven appropriately. The GNC model output may differ for satellite iterations and variants, and the GNC output should be versioned and described. Note that it would be possible to store the model code in the same repository if desired.

6.2 Build Process Artifacts

The intent of the build system is to generate artifacts that can be used to run and improve the satellite ecosystem. There are two kinds of build artifacts generated in Fig. 3, target-agnostic artifacts and executable artifacts which are target specific. Note that the target-agnostic artifacts can be used as input for building the executable artifacts, but they can also be exported to other teams for ingestion in other build environments. The target-agnostic artifacts may tend to be in declarative languages such as JSON or YAML, which can be consumed by multiple groups operating in different coding languages and operating environments. It might be the case that the artifacts target certain coding languages and some output artifacts can be consumed directly by those languages (like C headers or Python libraries). The point of having the target-agnostic artifacts is that an update to a platform dictionary can be rolled out and tracked across multiple systems with only a single change, not by having to coordinate independent changes in multiple code bases. The rollout to a live system must be orchestrated, and the satellite ecosystem must support more than one active platform. When operating and building satellites for any length of time, it will be the case that multiple platform definitions will always be in play. Note that the idea of target-agnostic artifacts (or artifacts that are published and used by multiple systems) is hardly new. The CCSDS has published documents related to this concept for many years (CCSDS 1987, 1992).

6.2.1 Target-Agnostic Artifacts

- **Telemetry dictionary** – Each satellite platform will have a unique set of telemetry channels that are used to monitor and control its operation. Telemetry information that is created on the satellite will generally not be self-describing due to file size concerns and may also be in binary format. The telemetry dictionary is used to convert machine-generated telemetry points into human-readable data points. It is also used to describe the relationship between the telemetry point and the hardware (source device or subsystem), provide design context if appropriate (description), and also give the units. The telemetry dictionary can be used to convert telemetry in real time (during a radio pass), potentially into graphical displays for operators. It can be used by back-end tools to display historical data, and it can also be used by developers to understand how to build out automation. These scenarios will all involve different tooling and coding languages which is why the dictionary must be code agnostic. The telemetry dictionary will exist across the ecosystem and should be immediately indexable by satellite platform type with multiple telemetry dictionaries existing at once. Note that it is possible for some telemetry channels to be common across satellites (attitude parameters, for instance). This is generally dependent on the subsystem. How one handles the common parameters is up to the designer, either by generating a new (but duplicated) list or by using some kind of inheritance.
- **Event dictionary** – The event dictionary has much in common with the telemetry dictionary. The distinction here is that telemetry is considered time series data (like system voltages), whereas events are one-time data (like a configuration change, limit check warning, etc.). Events tend to have more complex data structures associated with them which makes it appropriate to build out a separate event system from the telemetry system. The manner in which the event dictionary is used is similar to the telemetry dictionary except it will have to describe the more complicated data structure for each event.
- **Hardware (HW) configuration data** – Each satellite variant should start with a hardware configuration which will enumerate the bus layout, the individual device hardware, and the device configuration. This will come in large part from the EE interconnect information. It will also include physical data related to the satellite such as sensor and actuator mounting alignments. The HW configuration information can be pulled into the FSW build process in a similar way that device trees are used in an

operating system (OS). A device tree is generally a binary blob that an OS knows how to parse that describes how it is connected to external hardware (like clock source, pin assignments, etc.). There is another important job for the HW configuration information which is to be imported into the mission operations center (MOC) satellite database with an entry for each new satellite. This database is critical to satellite operations, especially at scale. The MOC database will contain this hardware information; it will detail which software version is on each processor and what the backup software version is. It will detail the current operational mode for each satellite, and it will provide information about security keys, power models, calibration data, etc. Very importantly it will also track degradation of satellites over time which can be tracked against the original HW configuration. The full extent of the MOC database will not be described here, but when flight software is loaded on-orbit and needs to adjust based on new or degraded hardware on a particular satellite variant, it is likely that information will come from the MOC database.

- **Message definitions** – Communication with the satellites will be through a defined interconnect. It is useful to create and store the message definitions such that they can be used to generate message serializers and deserializers on both the satellite and the ground. These serializers/deserializers will be pulled into the flight software as part of the build process. They can also be exported to groundstation code and manufacturing code as appropriate.

6.2.2 Executable Artifacts

The executable artifacts are more recognizable than the target-agnostic artifacts. The need to address multiple deliverables is discussed in **Role of the Flight Software Team**.

- **Flight software for flight computer** – This is the software image for the flight computer. This will likely also include the bootloader image for the flight controller.
- **Flight software for slave microcontroller(s) (uC)** – As mentioned in a previous section, there will be many microcontrollers and possibly microprocessors in the Microsat design that require software. The build process will have to generate all of these software products, and it is likely that the build products will require different tools and possibly different build environments. This may include bootloader images for some of the processors.
- **Flight software for payload** – The payload computer and system will require software. This software may or not be in the same code repository as the avionics flight software. Depending on how the payload processor is architected, this could include bootloader images.
- **Ground tools software** – This would include ground-side radio software, groundstation software, possibly data analysis software, and maybe components that are interoperable with mission control.
- **Manufacturing tools software** – This includes software that might be needed to interface with the satellite while it is on the production line, potentially using a different (and protected) interconnect definition that is only appropriate on the ground.

It should be noted that Fig. 3 shows the build inputs and artifacts but does not indicate where the artifacts are stored. Artifacts will have to be published to some location that makes them available to satellite operations and manufacturing personnel while still meeting any security considerations. The above build architecture is fairly complex, and it may be appropriate to leave out certain pieces until they are needed or to implement it in a different fashion. Ultimately, there will need to be a server level build option which can reliably, repeatedly, and automatically execute the build, unit test, and publishing pipeline steps. However, it is still useful to understand why the design is laid out this way in order to avoid future problems.

7 Flight Computer Software Design

The flight computer is responsible for maintaining the satellite in a power-positive, ground-responsive, and thermally survivable envelope. It is also responsible for enabling the payload to succeed in its mission, through execution of time-sequenced flight control commands, attitude adjustments, orbital maneuvers (if applicable), and power sequencing of payload components. The flight computer must enable the Microsat to downlink its payload data as appropriate. In general, there is the same command and telemetry requirement as all flight computers. There is a significant amount of existing and comprehensive literature about satellite design that should be considered when developing the flight computer architecture (Brown 2002; Wertz et al. 2011). There are also third-party vendors who offer flight software products specifically for Microsats who may be able to supply part or all of the flight computer components, allowing the team to focus on other components such as the payload and its software.

NASA has open-sourced a comprehensive flight software suite called the “Core Flight System” (NASA 2014, 2019). There is a wealth of experience behind this effort, and it is very informative to investigate. There are many similarities between the content of this section and the CFS, which is to be expected since they solve some similar problems. The CFS is actually a superset with respect to functionality since many different kinds of missions can be supported (including deep space), whereas the content here is for LEO Microsats. The CFS may be appropriate as a candidate for Microsat FSW. If one can use the CFS, many capabilities may come for “free” at a later date when they are required. The downside to the CFS is that it can be complex, and the initial learning curve will be steep. In the case that only certain components are desired from CFS, it is a little difficult to separate them out from the larger system. The CFS is also currently ported to certain operating systems which may not be appropriate for the processor that was selected for the flight computer, or the processor may be not be sized appropriately. Note also that the CFS does not necessarily solve certain software problems on the satellite, such as the infrastructure for supporting the local communication bus or integration of sensors and actuators (highly dependent on hardware design). This will somehow have to be integrated into the CFS system. The build system and artifacts for CFS may not match well with other processors or programs that the software teams are working on, including server-side continuous integration (CI) and build. Integrating custom over-the-air protocols may also be more work. However, it might be the case that third-party groundstations are going to be used, and these use some kind of standard protocol schemes, such as those defined by the Consultative Committee for Space Data Systems (CCSDS). The CFS may be able to support some of these protocols natively. In any event, this should all be evaluated from the point of the satellite ecosystem and requirements.

Figure 4 details a possible software architecture for the flight computer. This is a static view of the software that demonstrates the hierarchy between components. The blocks that are on the bottom are generally independent of the ones on top. It should be noted that many of the blocks here are consistent with the NASA CFS diagrams and should be consistent with almost any flight computer design. This is the architecture for the flight computer, but note that there are many components in the diagram that would also be applicable to microcontrollers or other processors in the satellite. This is an opportunity for code reuse and modularity that should be taken advantage of.

Figure 4 has several layers which are described below:

- **Applications** – The application layer as defined here represents the high-level functions that the flight computer must perform. These are generally mappable to requirements such as thermal management, power monitoring, spacecraft attitude tolerances, and time maintenance and distribution. The flight computer must be able to provide enough telemetry for each of these applications to be monitored against its design and performance envelope. Note that at the software level, the code for an application may be split among a task, an interrupt service routine, and potentially an asynchronous messaging handler while still sharing common data. The "Command Processor" is the externally facing consumer of messaging which can provide protocol related capability and is a gateway to the internal messaging bus. The "Power" application will monitor the power state of the system and effect brownout or safety related actions as appropriate. It will also be used to command various parts of the satellite to power on and off. The "Thermal" application will maintain the satellite in its thermal envelop by monitoring temperatures and activating thermal control systems as appropriate. The thermal application will be subject to any restrictions that are being enforced by the power application. The "Runtime Sequencer" will accept arrays of instructions from the ground that represent sequences of future states of the spacecraft. The runtime sequencer will ensure that the satellite state is executed as defined in the timing of the sequences by issuing commands to the other applications. The sequences represent the steps that the spacecraft must execute to complete some phase of the mission (like pointing at a groundstation for downlink). Note that the runtime sequencer should ensure that satellite changes are atomic. It must also be robust against sequence interruption and failure.
- **Services** – Services as defined here are software constructs which enable the applications to perform their jobs. Some of these may not have related requirements but are dictated by software design. This includes internal messaging

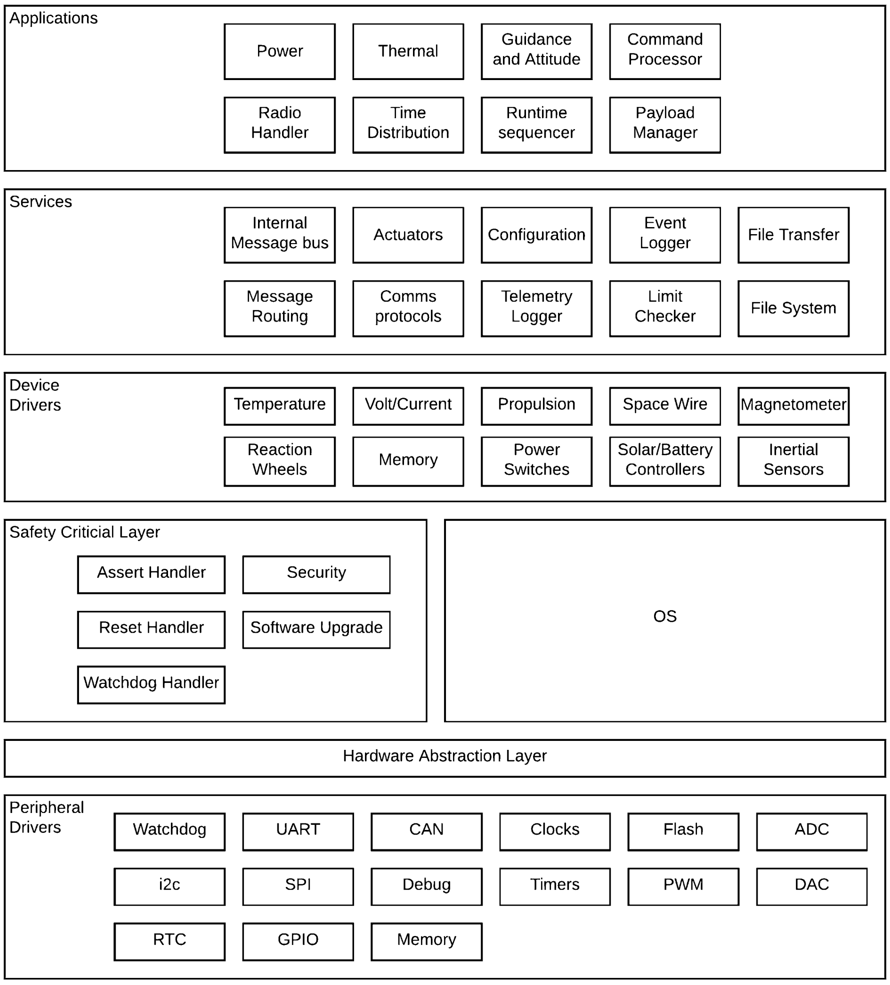

Fig. 4 Flight computer software architecture. (Image © 2011 Planet Labs Inc., licensed for one-time publication)

buses, configuration services, telemetry and event logging services, file system implementations, etc. Note that it is important for these services to be independent of the satellite-specific content. It is not appropriate to build a telemetry or configuration service that is unique to the telemetry or configuration for a given spacecraft variant. This will allow easy portability to new satellite systems.

- **Device Drivers –** Device drivers will be required for all components that are accessed through the various communication buses (such as SpaceWire, UART, I2C, SPI, etc.). They will have command sets and register definitions which are not local to the processor. These are distinct from devices which are natively supported by the processor chip architecture (see peripheral drivers below). Note that the intent would be that these device drivers are portable to other processors. The device drivers will often have to be written by the flight software team. The device driver layer is shown over the OS, but it is possible that applications may be written to bypass the OS and access these devices directly. This is highly dependent on the OS and possibly memory ramifications (allocation/deallocation, direct memory access, etc.).

- **Safety-Critical Components and OS** – This layer has somewhat less of a strict definition to it, in that the relationship between the safety specific components, the OS, and other components may be more complicated than the simple layer approach shown. The safety-critical components are called out explicitly because they are some of the most important software features in the spacecraft. These must be well-designed, robust, tested, and extremely well understood. These features are discussed in further detail in sections "Satellite Safe Mode" and "Security".
- **Hardware Abstraction Layer (HAL)** – This layer is self-explanatory. It abstracts the OS and the higher-level features from the specifics of the hardware implementation. A good HAL means that porting the flight computer software over to a new processor will be easier (e.g., if the flight computer chip is end-of-life (EOL), design changes require a more powerful processor or potentially for a different satellite product). The HAL may be tightly integrated into the peripheral driver layer.
- **Peripheral Drivers** – The processor will require drivers for all peripherals that are to be used to control and monitor the spacecraft. These are generally the peripherals which are part of the processor and have associated internal registers and pins. This code will generally be processor type specific and can many times be available in board support packages or third-party code repositories.

The flight computer architecture as presented here is only one possible example. When designing the system, the flight software team should always look to minimize or eliminate blocks when appropriate (e.g., by standardizing on an external communication bus and external sensors). They should also be looking to share code wherever appropriate among various processors.

8 Flight Software and System Design

There are many aspects of the satellite system design and larger ecosystem that impact flight software or can be impacted by flight software. The following details some specific cases that will be helpful when building out flight software. It is also useful to call out some topics which deserve special attention, including satellite safe mode, software upgrades, and security.

8.1 Satellite Safe Mode

Microsats are generally not designed with the same redundancy and reliability as traditional satellites. This is obviously the result of the much reduced cost. However, they still require the concept of a "safe mode" that the satellite can enter when a fault occurs or for a general mission safety condition. An understanding of the satellite safe mode should be built up very early in the design process, and components related to the safe mode should be prioritized and worked on first in order to give them maximum runtime on the ground prior to launch.

The safe mode should involve a minimum number of subsystems and should allow those systems to have authority over as much of the rest of the satellite as possible. An absolute minimum set of subsystems would be the flight computer, the power control board, and the TT&C radio (assuming no subsystem redundancy). Focusing on these subsystems is extremely important.

Two common failures with Microsats are electronic latchups and processor memory corruption. Latchups can often be cleared by removing power from the device and then reapplying it. This creates two design considerations; first, the ability to remove power from the satellite and its subsystems should extend as far upstream as possible in the power infrastructure. Second, it is very useful to ensure that the flight computer firmware can control power to as many subsystems as possible (as opposed to systems being driven by a non-switchable high-level power bus). This may allow the flight computer to detect and assert a fix, instead of having to force a satellite-wide reset or maybe wait for a planned low-power state (brownout) event if the latchup is exceptionally bad. It is also a good idea to ensure that the flight software is able to actively bring the satellite into a power-on reset (POR) state after a software reset. This means actively asserting all devices under control into a known state, either through direct I/O pin manipulation or setting registers in devices which are connected via communication buses. The flight software should also be capable of actively shutting down power to subsystems in a benign and orderly way. The firmware should be able to make assessments of which buses it believes are healthy.

From an operational point of view, it should be possible to try and address a device failure first through flight computer commands, then through a flight computer reset, and then possibly through a full satellite reset. It should be the case that the flight computer can request (or force) a full satellite reset in some way. From a recovery standpoint, it may also be useful to be able to use the TT&C radio to reset the satellite in response to a radio message, assuming that this can be done as a secure operation.

Watchdogs should be implemented on the satellite in order to try and bring the satellite back into a known state if there is a fault that renders one or more of the safe-mode subsystems unreliable. Processor internal watchdog capabilities should always be used if available. It may be desirable to implement an external hardware watchdog of some kind. All critical systems should have a recurring requirement to "pet" the watchdog and push off its reset function. In the case that a watchdog triggers, the faulty system should be considered completely unreliable, and the watchdog system should not use that system for recovery (by design). It is useful to have some watchdog countdown timer implementation that is reset after every contact with the ground. Note that any system that can reset the satellite should not have a pathological condition where it continually resets the satellite and the time between resets is very short.

The processor assert handler also plays a role in the safe mode of the spacecraft. The assert handler can be called from either hardware or software. The triggering of an assert could be from an unexpected operational request or from undetected memory errors. The assert handler should be designed using the same concepts as the watchdog implementation, and the result of triggering an assert could be a reset of the processor or possibly of the spacecraft. The various possible asserts should be categorized and actions taken as appropriate.

From the flight software perspective, the reset sequence for the satellite should be considered carefully. As noted above, resets of the flight computer and other processors can be a useful tool for trying to bring the satellite into a known state. As such, the path that the reset sequence takes as it executes will be an integral part of the safe mode implementation. A processor will likely go through some kind of bootloader code before starting the application. What happens if this bootloader code remains active for any period of time or stays active because of an application failure? The flight computer bootloader may be one of the most important pieces of software that the flight software team will write. It may have to contain hardware initialization code to ensure that the satellite is properly and safely configured. It may have security requirements or it may have to deal with the TT&C radio. The bootloader will have to ensure that an application that it intends to launch is integrity checked and possibly authenticated, and it must understand what to do when this fails. The bootloader is very importantly one of the few software components that is generally not updated in space. There are other considerations as well for flight software, for instance, the flight computer should prioritize and initialize the most critical systems first and then proceed down through the less critical systems.

Memory errors are another common form of radiation-induced error on spacecraft. Processors and FPGAs should be capable of detecting and/or fixing memory errors at some level (for instance, with ECC). When an uncorrectable memory error has occurred, the processor should immediately be considered suspect. One of the safer approaches is to tie an uncorrectable ECC error into some kind of hard reset circuitry which could be triggered through the absolute minimum amount of software (maybe through only an interrupt service routine with no application code). Another approach is to disable the watchdog monitoring code (ensuring watchdog will fire) and then to try and capture as much information about the state of the processor as possible before attempting a reset through software. This can dramatically aid debug work. This is a less robust approach, but it also allows for the possibility of shutting down high-power systems that might be active in a benign way.

The safe mode should have a very well-understood outcome for guidance and control. The simplest model is to shut down all active guidance, stop all actuators, drop into a low-power mode, stop executing on pre-stored command sequences, and await instructions from the ground. This will put the satellite into a tumbling state, which would have to be understood from a systems standpoint (total possible spin-up, pathological solar panel pointing, etc.). The next level up would be to assert some basic form of guidance to at least keep the satellite from tumbling. Care and thought should be put into this kind of design because at some level, an algorithm such as this will have to assume that certain sensors and actuators are functioning correctly. If these sensors and/or actuators have failed or are somehow calibrated incorrectly (for instance, through memory corruption), then the impact of applying a guidance mode could be disastrous. This kind of design would expand the boundaries of the critical systems required for safe mode.

8.2 Atomic Configuration Updates

It should be possible at any given point to have a clear understanding of the configuration of the satellite. More importantly, the satellite should never end up in a state where only partial configuration has been applied. This places a requirement on the system design to ensure that configuration changes are atomic. The potential for partial configuration changes could occur during interrupted radio links or if the satellite is executing pre-stored command sequences and some form of fault occurs that aborts the set of command sequences before they are completed. The satellite may then not execute a command sequence that was intended to restore some background state for the spacecraft.

Configuration is used in two senses here; there is the concept of "configuration data," such as calibration data, satellite sensor frame data, security settings, payload settings, current orbital parameters, guidance parameters, etc. This tends to be longer-term data which is updated from the ground during a radio link pass. There is also the way in which the satellite is "configured" at runtime, such as varying power rail settings for payload and heaters, satellite pointing mode for solar panels, etc. This generally comes from the satellite flight software design and the ongoing execution of pre-loaded sequences to execute on the mission. Both of these types of changes should be atomic in nature and should be revertible (if appropriate) at reset or fault.

This idea of atomic configuration updates may seem obvious, but configuration information can be spread across multiple systems and subsystems. It may also be the case that certain configuration data is required to be updated more often and considered less critical, creating the desire to implement different messages for different configuration parameters. It may also be the case that different system owners have devised different schemes for how they ingest configuration requests.

The amount of configuration data for a satellite, even a Microsat, is considerable. It is unlikely that the whole configuration state can be applied at once, even if desirable. A system-level breakdown of configuration data into manageable and appropriate groupings is appropriate, ideally with each element in the group then being atomic in nature.

For configuration data, it is appropriate to create wrapper logic around configuration set commands that can atomically apply new configuration (once verified) and which more importantly can assert a known or default state at reset or fault. Verification of the data should involve a cyclic redundancy check (CRC) or potentially security-related verification (HASH) depending on how security is implemented and should also include bounds checking where appropriate.

For runtime configuration of the satellite (pre-loaded command sequences), it is useful to define a higher-level "manager" application that will ingest the request for state change and act on it as appropriate (as opposed to executing some low-level command directly). This manager function can then also contain the logic for reset or fault cases and can restore default or safe state as appropriate.

8.3 On-Orbit Software Upgrades

As stated previously, the philosophy for Microsats should be that software upgrades while in orbit should be reliable, possibly often and not exciting. As mentioned in **Flight Software Life Cycle**, there are many reasons why the flight software may not be ready at the time of launch. It is also the case that since full space simulation may not be possible on the ground, lessons learned during actual on-orbit testing can only be incorporated using on-orbit flight upgrades. Verification of payload may be even more tied to on-orbit testing, and updates to the payload software should be expected.

Planning out software upgrades to the satellite is one of the most important system design questions to be answered by the flight software team. This includes the flight computer, peripheral microcontrollers, and the payload system.

Processors should always be able to maintain at least two different images for software, with one being active and the other idle and available for update. This means that memory architectures should be sized appropriately for two images, as well as multiple copies of configuration data, redundant security information where appropriate, and room for a bootloader. Payload data should (if possible) be kept in a separate memory device than the flight software. It will have a different use profile and will likely require a much larger memory device which may be less tolerant of the space environment. When updating flight software, it is useful to enforce a design where both flight software images get updated in a ping-pong fashion such that one is just ahead of the other in version. The advantage of this is that it reduces the "staleness" of the software in orbit. The impact of stale software can be a tedious, long road of sequential updates that must be executed by the satellite operations group to bring the satellite up to date if a flight system must be reverted to a backup copy of software that is extremely old.

Software upgrades will be somewhat sporadic with respect to the mission life cycle and could potentially cause large peak loads on radio links and internal bus links between processors. It is important to ensure that these links are sized appropriately for software upgrades and avoid the tendency to only focus on the steady-state operation of the satellite with respect to bus and communication design. It should not be the case that the processor selection and flight software design result in software image sizes that are beyond or barely within the transfer capability of the satellite uplink. It should also be taken into consideration how long the satellite may be unable to focus on its mission while in the software upgrade mode. When uplinking software updates to the satellite, the design should be robust against losing the radio link, subsequently being able to resume the upload at a later time without resending data.

In the software upgrade design, it is possible to consider both atomic upgrades of the software and partial upgrades. An atomic upgrade would replace the entire image with an uplinked version. This version can be tested and validated on the ground for each satellite variant as appropriate. It is also possible to have a partial update scheme, using a package manager or virtual container for high-level operating systems or by uplinking a "binary diff" file that can be applied to static content on the satellite, integrity checked, and then loaded into nonvolatile memory. Partial upgrades can reduce bus traffic but have two downsides. The first is that a series of partial updates applied to various satellite iterations can be hard to test, track, automate, and revert. This can be very painful for satellite operations. The second is that a partial upgrade does not protect against corruption of the existing content on the satellite (for instance, through memory corruption). It is useful to be able to minimize the software cross section that cannot be updated atomically in orbit, in particular for unrecoverable radiation damage. In most cases, it should be possible to update everything except the bootloader.

8.4 Accountability in Operations

In a fleet of Microsats, the level of automation in satellite operations will be quite high. There will likely be a computing infrastructure that is monitoring the satellite fleet and generating data analytic reports that are fed into the various organizations in the company. The monitoring capability will be tied into an automated escalation and alarm system. Generally there will not be live personnel monitoring the satellite system 24/7. This automated monitoring will benefit from satellite data that is easy to machine parse and which is consistent in time stamp. For simple time-series values such as voltage, current, spin rates, etc., the monitoring is straightforward, and there can be a threshold set for both warning and fault levels. However, it is also useful to find alternative solutions for cases where the automation must perform complex processing on the received data in order to arrive at a conclusion; this includes having to apply some kind of waveform processing to time-series data, for instance, trying to estimate battery charge state from historical current and voltage levels (which may be sparse or incomplete).

The need for hardware-related telemetry on a satellite is more obvious than the need for system-level telemetry and monitoring. In a Microsat ecosystem, there are multiple systems in play, and each of them may be rapidly evolving, including mission control software, groundstation software, radio software, and satellite software, not to mention possible changes in groundstation hardware and cloud-computing infrastructure. When operating a fleet of Microsats, it is likely that there are legitimate reasons for some payload windows to fail; this includes technical reasons and also operating reasons such as aggressive time windows, operating close to the edge of the power envelope, etc. The point is that there will always be a high level of general noise that can make a new failure difficult to detect and isolate. Once detected, it can become a very daunting exercise to figure out why certain operations have not taken place if the observability of the system is poor, especially if they are finally noted as a general downward trend in system-level performance. Observability is then extremely important to be able to track down and diagnose issues that arise in the program.

In general, the flight software team and all other developers should embrace this idea of system "observability," the notion that at any given time, it is possible to trace how the satellite got into a particular fault state or failed to execute on its payload mission for some interval. This "observability" should also be designed such that it is easy to integrate into the automated monitoring system. An example of why this is important is measuring fleet-wide system performance against a customer service-level agreement (SLA) for a remote sensing type of mission. First, it is important to understand that there are a very large number of reasons that payload remote sensing data does not make it to the customer. Consider Fig. 5, which lays out how a payload acquisition request results in product being delivered to the customer. For convention, the left half of the diagram is labeled as "Reaction Time" which is the time between payload data request and payload data acquisition. The right half is labeled "Latency"; this is the time between payload data acquisition and delivery to the

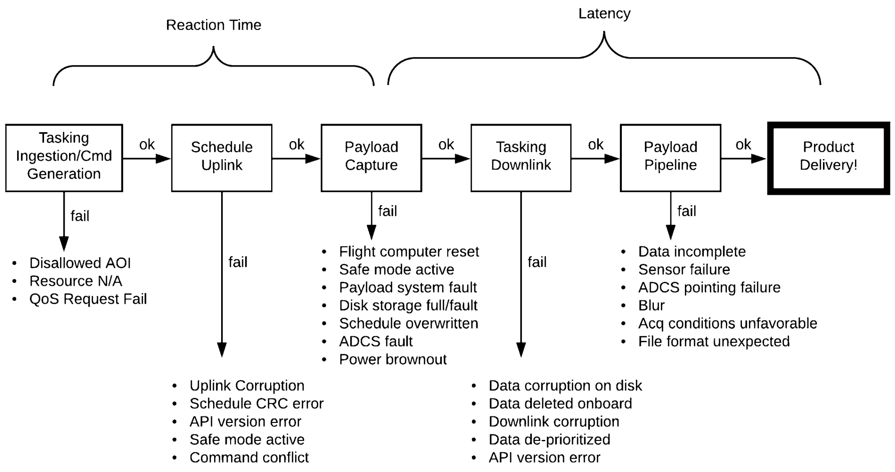

Fig. 5 Accountability for payload delivery. (Image © 2011 Planet Labs Inc., licensed for one-time publication)

customer. Both the reaction time and the latency may have SLA-related limits. For each step in the process, there are some example failure conditions listed.

The leftmost block is the initial task request, which may fail due to an inappropriate area of interest (AOI), satellite resource conflict, or potentially a failure due to an unmeetable quality of service request. Once declared acceptable, the request is turned into a series of command sequences that must be uploaded to the satellite, which is the next block in the sequence. There are multiple reasons that this can fail as well, including the satellite being in safe mode, potentially a command conflict, or the radio pass just happens to fail for equipment or planning reasons. This pattern continues to the right until data delivery to the customer. At the system level, each step of the process will have a non-zero failure percentage that must be observable and monitored. As can be seen from the rest of the diagram, there are many fault paths for the payload sequence (and many that are not shown). The intent for the system monitoring should be twofold: first, the customer facing team must be able to answer the question of why a particular payload sequence was not acquired, and, second, it must be possible to rapidly track down which step in the process has degraded. And then use that information and the observability built into the design to understand why the SLA may be at risk.

The reality here is that it can be very difficult to estimate the impact of what appears to be a small change, but (by design) it should be very easy to monitor it once it is deployed. Changes are always being made to the system, and this must be built in to the monitoring expectation. Early on in the design process and at regular intervals, the team should audit the event and telemetry streams and ensure that high-level functions in the end-to-end satellite ecosystem can be monitored easily. Each team should also be on the lookout for changes which may derail payload delivery and ensure that those events are reported and ingested into the automated monitoring.

8.5 Security

A critical aspect of the satellite ecosystem is the security posture and security implementation. The security posture for a mission will vary depending on the perceived value and lifetime of the mission. An academic mission of a single Microsat may have a less rigorous posture than a large satellite fleet that has ongoing launches for replenishment (and a longer window for security practices to become stale). The security posture will also be dictated by regulatory bodies such as NOAA in the USA. In any event, a satellite mission should have a security plan that comes from the security posture (requirements) which allows the security to be implemented in a way where risks are properly assessed and appropriate measures taken to achieve the desired level of security.

There is a reality that security design on a Microsat may not be given as much priority as other components and subsystems in the spacecraft. This is especially true in the early phases when basic systems are being developed, resources are short, and there is a significant chance that the early Microsat will not even reach full mission status after launch. There is then a likelihood that security features will be implemented “over time” as the design and potentially the company become more mature. From a flight software standpoint however, it is very beneficial to understand the eventual security capabilities early in the design process and identify the possible impacts to the flight software design. The use of standard security paradigms can greatly aid this process, and they should be considered (this will also help to avoid creating security vulnerabilities). The literature around security practices and possible implementations can be daunting, but an applicable and good starting point for a Microsat satellite design is to understand the IP security (“IPsec”) open standard Wikipedia article and then the various standard documents themselves (Kent 2005; Kent and Seo 2005). A good satellite-based security document to read after IPsec is the “Space Data Link Security Protocol-Summary of Concept and Rationale” published by CCSDS (CCSDS 2018). This document can reinforce some of the IPsec concepts through a satellite-based example. When choosing a security paradigm, the encrypted and authenticated option should be preferred. Note that an interesting aspect of the CCSDS implementation is the concept of the “authentication bit mask” which allows for selective exclusion of some fields in the message authentication code (MAC) which can make the application of authentication slightly more flexible.

The CCSDS publishes many security-related documents which are worth reading: the “Security Guide for Mission Planners” (CCSDS 2019a) is a good overview of security for the whole mission. It also lays out how the various CCSDS security documents are interrelated. The “Report on the Application of Security to CCSDS Protocols” (CCSDS 2019b) also has great information on security for satellites.

Returning to the flight software team, the goal of the team should be to break down security requirements into capabilities and then come up with a schedule to implement them, preferably as soon as possible. Availability of these capabilities can provide security to the Microsat even when the security plan is not completely fleshed out. At a high level, these capabilities will relate to requirements for hardware and software cryptographic capability, tamper detection, the storage of crypto keys in nonvolatile memory, and implications for the design of communication protocols. It is easy to fall into the trap of building out mission functionality first and then being in an awkward position trying to implement security on top of the resulting implementation. It is best to try and avoid this.

Before examining possible security capabilities, it is worth mentioning some concepts that can impact the flight software security design at a fundamental level. The first is the concept of a security boundary and a cryptographic module, and the second is the notion that the endpoints of a security paradigm may terminate in different places on the ground.

In the context of satellite security design, establishing the security boundary means having a clear understanding of how data and commands transition from a trusted to an untrusted domain and vice versa. A cryptographic module defines the perimeter within which cryptographic processing is performed, such as command and telemetry decryption and encryption. From a flight software standpoint, establishing the security boundary and cryptographic module on the satellite will help identify which devices on the satellite must have cryptographic capability and access to keys and which do not. This may also impact how devices are interconnected.

With a less stringent security posture, the security boundary could be defined as being at the physical satellite boundary itself, potentially at the interface where the satellite feeds data to/from the radios. The cryptographic module could encompass all the devices except for the radios, so keys could be freely shared among all devices, and one or more devices would be responsible for key management and cryptographic operations. This model imposes fewer design constraints in terms of satellite design and may be adequate for a satellite with a less stringent security posture.

With a more stringent security posture, the security boundary could again be defined as being at the physical satellite boundary itself, but with a singular subsystem (device or devices) forming the cryptographic module responsible for key management and cryptographic operations. While this may impose more design constraints, it should (hopefully) provide for tighter control around the security processing on the satellite. Having the security code separate from other satellite functionality should also support easier auditing of the security operation. Note that security boundaries and cryptographic modules also apply to ground systems that perform encryption and decryption of satellite data.

It is important to understand that the endpoints for security capabilities may vary depending on the purpose. Figure 6 indicates some possible security paradigms. Note that these concepts are also discussed in CCSDS references (CCSDS 2011, 2019b). The point here is that there are security links that can terminate at the local groundstation and also security links which can potentially terminate in dedicated or cloud-based servers for the mission operations center (MOC) and payload processing. It is therefore inappropriate to focus on a security paradigm which is only designed for the over-the-air (OTA) datalink, for instance, and it must be the case that secure messages can be routed from a groundstation through the ground network to the appropriate downstream endpoint (mission control or payload processing). In the diagram, "Mission Operations Center" represents the server or cloud-side processing system which automatically manages the satellite constellation. The following describes the links in Fig. 6:

1. **Mission Operations Center and Satellite** – The connection shown here is that of a traditional "bent pipe" where the groundstation merely relays packets back and forth (both telemetry and command). The groundstation may implement some form of data link security and may have the authority for message retries.
2. **Groundstation and Satellite Radio** – It may be the case that the round-trip delay between the groundstation and mission operations center is inappropriate for certain operations, and there is a desire to implement some autonomy on the groundstation itself. A prime example of this is adaptive coding and modulation (ACM) where it is desired to keep the feedback loop driving the radio as tight as possible. Automatic repeat request (ARQ) can also be done locally at the groundstation.

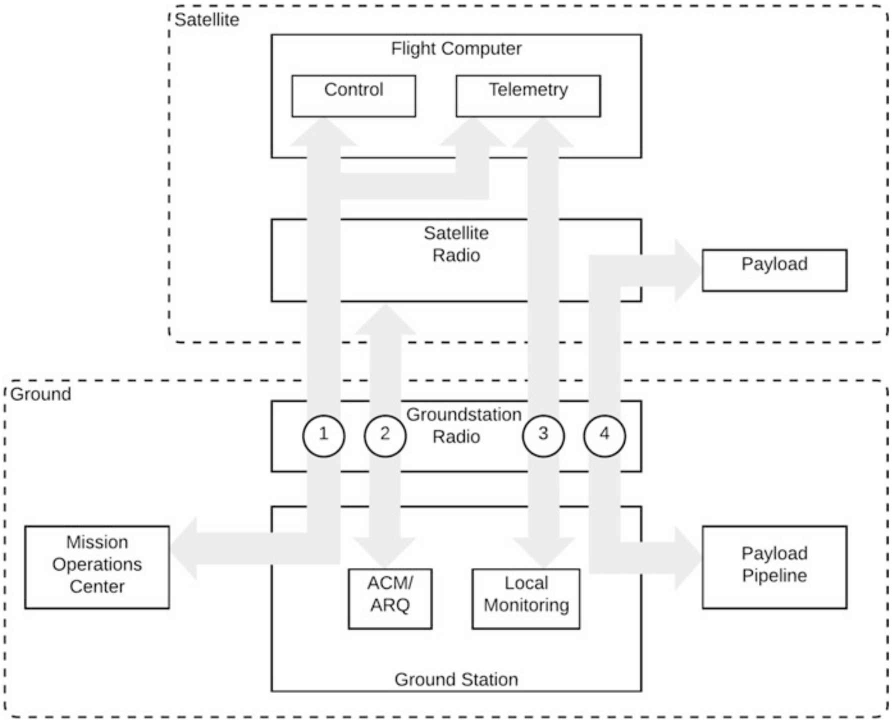

Fig. 6 Security routing endpoints. (Image © 2011 Planet Labs Inc., licensed for one-time publication)

3. **Groundstation and Satellite Telemetry** – This is a more uncommon case, but there may be a reason for satellite telemetry information to be available at the groundstation. This would typically just comprise telemetry received directly from the satellite.
4. **Satellite Payload-to-Payload Pipeline** – The payload data will likely be delivered directly to a cloud-computing endpoint or potentially a customer endpoint for pipeline processing. The ground system will have to understand how to route this information to the payload endpoint. Note that it may be the case that the payload does not have to be encrypted at all. This will depend on business and regulatory conditions.

8.5.1 IPsec Concepts and Capabilities

The concepts behind the IPsec model represent capabilities that will very likely need to be implemented on the satellite. Note that implementing simple models for these capabilities while still following best practices for the actual encryption and authentication can go a long way in securing the satellite. The following list gives some context around the capabilities that may be useful to the flight software team:

- **Security association (SA)** – The security association is an agreement between two entities about how they will use security capabilities and practices to communicate in a secure fashion. The security association for an early stage Microsat can be simple, potentially decided entirely in advance with no runtime negotiation and based on pre-loaded symmetric keys. The pre-loaded keys can be used directly for encryption and authentication purposes. There should be a security association entry for each security relationship in the system (as shown in Fig. 6). At a minimum, the SA should have a version and detail a minimum set of cryptographic algorithms to be used (encryption, MAC), key attributes, and a list of valid key IDs. Alternatively for satellites desiring more flexible key rotation, the SAs on board the satellite may be updated using a form of key exchange to generate fresh traffic protection keys (CCSDS 2011). Consideration needs to be given to the number of message round trips required for the establishment of an SA as this adds latency to the process.
- **Security header** – The security header is added to the data content and is intended to indicate how the corresponding data should be decrypted and also to allow for authentication of the data. The name "security header" is presented to be generic, in that the header can be designed as appropriate for the Microsat system. It should have similar parameters as the Encapsulating Security Payload (ESP) from IPsec (Kent 2005) where necessary, such as the Security Parameter Index, the initialization vector (IV), a sequence number, and an ICV or MAC. The CCSDS document for the Data Link Security Protocol (CCSDS 2018) has both a "security header" and a "security trailer" which when combined have similar fields to the ESP. Since authentication is generally a good idea, combining the CCSDS "security header" and "security trailer" may be an appropriate choice to consider. Note that the IPsec definition is tied to IP and the CCSDS design reference above is tied to a data link layer. It may be appropriate to create a security header that is agnostic of protocol or communication layer.
- **Security Parameter Index (SPI)** – The SPI is included in the security header and indicates which SA the content is intended to be used for. For a Microsat, this can be as simple as a scalar value which indicates which security association (SA) to use. It is potentially useful to also include a version into the field in addition to the SPI.

8.5.2 General Concepts and Capabilities

- **Software cryptography** – From a software standpoint, the types of cryptographic algorithms must be understood and implemented. It is useful to try and identify cryptographic libraries which are appropriate for the processor in the device selection phase. There is no reason that this should be done in-house. Note that software cryptography can create a large burden on the processor that must be accounted for. It is also an idea to ensure that there is headroom to grow on the processor in the case that the cryptographic algorithms must change. Cryptographic libraries can be very comprehensive and will likely contain many more algorithms than the satellite system needs, consider removing unused routines from the software, and also remove security algorithms that are no longer considered secure.

- **Hardware cryptography** – In the design and device selection phase, there should have been some consideration of hardware security capabilities. This would be either hardware implementations of cryptographic algorithms (like the advanced encryption standard AES) or maybe hardware acceleration with floating point libraries or digital signal processing (DSP) blocks. Note that hardware cryptographic capabilities are not upgradeable so if the hardware is unable to provide sufficient protection at a later date (for instance, if the key length or cryptographic algorithm is insufficient), then the update will have to be moved to software, or a new processor must be selected and integrated. A way of reducing the risk of this would be selecting cryptographic algorithms and key lengths that are thought to be secure for the expected lifetime of the satellite constellation. For constellations with a design lifetime of many years, designing in crypto agility can be challenging due to emerging quantum computing capabilities.
- **Key storage** – The flight software must be able to store the permanent keys that are provisioned at manufacturing, and it must do this in a reliable way. In the flight software work, the store will likely be one of the larger efforts. The key store must be persistent, potentially duplicated for redundancy, and protected against write failure. Note that in a Microsat, security keys are likely maintained in flash, and flash is generally fairly robust when not written to. It is a really good idea to not use a flash storage device that implements a flash translation layer (FTL) under the hood. This could mean that the memory storage device is actually moving the keys around for wear leveling, and a radiation event could be detrimental if it disrupts this write operation. The key management scheme should be tied into the SA that is discussed above, with each SA being associated with at least a pair of keys. For more complex security schemes, there will likely be a hierarchy of keys such as master keys and traffic encryption keys (CCSDS 2011) and also the notion of key life cycles and cryptoperiods. This will have to be reflected in the key storage design.
- **Protocol definitions** – It is important to ensure that the security header can exist in the protocol definitions from the start. When using third-party communication designs, this will likely not be a problem. However, in a home brew system, there may be some data overhead concerns if the maximum radio packet size is small. Care should be taken as well to ensure that the security implementation (encryption) does not interfere with any message routing implementations.
- **Replay protection** – Protection against the replaying of previously sent commands is almost as important as the authentication of commands. IPsec achieves this through the use of a sequence number associated with each SA that is incremented on each packet sent. The receiver tracks the sequence number for each SA and enforces ordering to reject replayed (repeated) packets. There should always be replay protection built into the design.
- **Fallback plan** – There should be a well-understood design for the security system if there is some kind of degradation or compromise. This may be a fallback security key or key store or potentially a different algorithm suite that is not normally used. It should be predictable to the satellite operations group when the fallback plan is activated.

Having the SW/HW cryptographic capabilities, the key storage, the SA, and the security header design will allow for implementation of a reasonable first level of security. If these aspects are handled correctly for the initial (or early) Microsat design, then the flight software team may only have a supporting role when the system is scaled up to many satellites. The bulk of the extra work will be in key management on the ground, both inside the mission operations center and the manufacturing process.

Security must be understood in all aspects of operation, including software upgrades and during resets and faults. Security must be active during the software upgrade process, and a failure in the upgrade should not result in a compromised security position (there should never be a possibility that the security is not active). The security design should also be robust against loss of time on the satellite, and as in any system, the security will only be as good as the weakest link.

9 Conclusion

When developing Microsat software, it is important to properly understand the scope of work involved for the flight software team and also the development life cycle when multiple launches are taking place. This will enable the flight software team to develop appropriate architecture and build philosophies that allow critical software components to be delivered in a timely fashion. The flight software team should ensure that on-orbit software upgrades and an appropriate safe mode are well established as early in the design process as possible. The environment in which the flight software team operates can be challenging for long-term planning and development. Where possible the flight software team should look to the future and ensure that they try and position themselves for future success at scale. This includes focusing on items such as building observability into the design, identifying security capabilities, and being involved early on in the processor and device selection process. A successful Microsat company could end up with a very diverse set of satellite hardware in orbit, both from ongoing developmental improvements and from random degradation of hardware once it gets into the space environment. The flight software team should plan for the implications of this reality and help ensure that the company can successfully operate a diverse fleet of satellites.

10 Cross-References

- ▶ High Altitude Platform Systems (HAPS) and Unmanned Aerial Vehicles (UAV) as an Alternative to Small Satellites
- ▶ Hosted Payload Packages as a Form of Small Satellite System
- ▶ Network Control Systems for Large-Scale Constellations
- ▶ Overview of Small Satellite Technology and Systems Design
- ▶ Planet’s Dove Satellite Constellation
- ▶ Power Systems for Small Satellites

- ▶ RF and Optical Communications for Small Satellites
- ▶ Small Satellite Antennas
- ▶ Small Satellite Constellations and End-of-Life Deorbit Considerations
- ▶ Small Satellite Radio Link Fundamentals
- ▶ Small Satellites and Structural Design
- ▶ Spectrum Frequency Allocation Issues and Concerns for Small Satellites
- ▶ Stability, Pointing, and Orientation

References

C.D. Brown, *Elements of Spacecraft Design*, Education Series (AIAA, Reston, VA, 2002)

CCSDS, *SFDU Operations: System And Implementation Aspects, Report Concerning Space Data System Standards, CCSDS 610.0-G-5*, Green Book, Washington, DC, February 1987

CCSDS, *Standard Formatted Data Units – A Tutorial*, Report Concerning Space Data System Standards, CCSDS 621.0-G-1, Green Book, Washington, DC, May 1992

CCSDS, *Space Missions Key Management Concept*, Report Concerning Space Data System Standards, CCSDS 350.6-G-1, Green Book, Washington, DC, November 2011

CCSDS, *Space Data Link Security Protocol-Summary Of Concept And Rationale*, Report Concerning Space Data System Standards, CCSDS 350.5-G-1, Green Book, Washington, DC, June 2018

CCSDS, *Security Guide for Mission Planners, Report Concerning Space Data System Standards, CCSDS 350.7-G-2*, Green Book, Washington, DC, April 2019a

CCSDS, *The Application of Security to CCSDS Protocols*, Report Concerning Space Data System Standards, CCSDS 350.0-G-3, Green Book, Washington, DC, March 2019b

E. Costenaro, A. Evans, D. Alexandrescu, E. Schaefer, C. Beltrando, M. Glorieux, *Radiation Effects in SSDs*, Flash Memory Summit, IROC Technologies, Santa Clara, 2015

S. Kent, *IP Encapsulating Security Payload (ESP)*, RFC 4303, ISOC, December 2005

S. Kent, K. Seo, *Security Architecture for the Internet Protocol*. RFC 4301, ISOC, December 2005

NASA, core *Flight System (cFS) Background and Overview*. (2014), PDF File, https://cfs.gsfc.nasa.gov/cFS-OviewBGSlideDeck-ExportControl-Final.pdf. Accessed 1 Nov 2019

NASA, *Radiation Effects and Analysis – GSFC Radiation Database*. NASA website (n.d.). Retrieved from https://radhome.gsfc.nasa.gov/radhome/RadDataBase/RadDataBase.html. Accessed 15 Dec 2019

NASA, *core Flight System – A paradigm shift in flight software development*. (NASA website, September 13, 2019), https://cfs.gsfc.nasa.gov/Introduction.html. Accessed 1 Nov 2019

D. Sinclair, J. Dyer, *Radiation Effects and COTS Parts in SmallSats,* 27th Conference on Small Satellites, AIAA, August, 2013

J.R. Wertz, D.F. Everett, J.J. Puschell, *Space Mission Engineering: The New SMAD* (Space Technology Library, Hawthorne, 2011)