Developing Picosatellite Flight Software

Alex Antunes
Capitol Technology University, Laurel, 20708, USA
Email: aantunes@captechu.edu

Randy Powell
Capitol Technology University, Laurel, 20708, USA
Email: rkpowell@captechu.edu

***Abstract*—**We present a decision path for creating flight software for linux-based university-class picosatellites. We favor languages and frameworks that support modularity and strong exception handling, and add that languages enabling fewer lines of code are easier to validate. Heritage and use of existing frameworks are useful but human factors-- that are often team dependent-- are more crucial for undergraduate teams. Additionally, picosatellites can benefit from "pico Agile" development methods so as to maximize time available for testing. We include case studies including core Flight Software (cFS), our C-based TrapSat sounding rocket payload, and our Python-based Cactus-1 CubeSat.

***Index Terms*—**picosatellite, CubeSat, flight software, Python, cFS, Agile, linux

I. Introduction

University-class picosatellites, particularly CubeSats, are an active area of undergraduate engineering and science development. Also, most fail [1]. Factors that greatly contribute to picosatellite success include ensuring sufficient testing time, creating a valid concept of operations, and ensuring the software is robust.

Our missions at Capitol Technology University fit soundly into the university-class mission category, consisting of sounding rocket flights (2014-2019) and our 3U "Cactus-1" CubeSat (launching 2019). Our flight software runs on Linux and uses mixes of locally written C, the Core Flight System (cFS) C framework, and Python. We are bandwidth-limited, with no capability for post launch reprogramming. Our teams consist of engineering students with typically one faculty advisor and (rarely) one graduate student.

We found trying to get high heritage (flown before), well-documented, open-source components is a 'pick any two' problem. Building a core CubeSat bus using primarily open source hardware and software requires focusing on interface adherence over performance statistics and an expectation that refactoring will be needed. In that framework, software language choice and development environment decisions will strongly affect your delivered software reliability, and we document best steps towards creating picosatellite flight software.

Through our experience and survey of current CubeSat flight software systems, in terms of language choice, we argue that readability, modularity, and exception handling are more crucial than performance or heritage. Human factors in software development-- especially the changing roster of undergraduate teams on long-term projects and the career utility of languages the team must learn-- are also important.

We focus on Linux based development environments. A 2017 study [2] notes that non-real-time operating systems (RTOS) such as Linux have gained in favor with small spacecraft, typically using Ada, C, or C++. Likewise, that Linux is increasing in use; the same study cites its use in QuakeSat, UWE-1, UWE-2, IPEX, STRaND-1, PhoneSat, the Dove satellites, and LightSale-1 (ibid). Similarly, Commercial Off-the-Shelf (COTS) Linux-based CubeSat boards are a significant market for CubeSat developers [3].

Of the missions listed, only LightSail-1 had a major software-related mishap. The LightSail-1 mishap is interesting: a bug lead to a file growing too large, but the problem was fixable after a single-even effect (SEE) caused a reboot and provided a window for a bug fix to be applied [2].

The LightSail-1 case therefore can be considered a coding error rather than an operating system error. We agree with the argument that, as Linux is complex, testing a Linux-based flight system will be complex [2]. However, for university-class missions, we generally rely on any OS being stable and thus aim for clarity in design of the components we must develop. Additionally, testing should be carried out on the same system as will fly. OS testing thus 'piggybacks' on the flight software testing, and our primary error catching is to ensure our flight software is developed with sufficient time to robustly test.

With this, the burden is to determine the best underlying core bus and payload module development environment. Particularly for a university environment, where we must train up our student developers at roughly the same time we are using them to develop software, language choice is a key concern.

Ref. [4] is more blunt, noting “Many university CubeSat missions have failed due to software errors. This is not surprising considering that most flight software is written in C, a language that is difficult to use correctly.”

II. Language Suitability

When creating a flight software stack for a university-class mission (such as a CubeSat), two primary considerations are the strengths of the development environment and the software’s fault tolerance capability. A third, “flight heritage”, is often called for as an advantage for space hardware. However, since each university-class mission is either new or an evolution of a past mission, heritage is not often feasible for university-class missions.

In addition, our Cactus-1 work has shown that inheriting a working but unknown (to the new student developers) system can be detrimental due to time penalties incurred by the learning curve. To better choose a language, we offer several criteria.

The language choice must be well suited for spacecraft, it must be known or learnable by our students, and it should be a language useful to a student's future career. This follows a general rule for choosing a language in any situation (shown in Figure 1).Therefore we focus on development environment as a primary consideration, by suggesting three general reasons for considering which language to use. Put simply, you choose a language because it’s the best for the job, because you know it, and because it’s available/installed. Many times this is a ‘pick any two’ option, in particular the availability of other mission software solutions means code developed by one university is probably not available to others.

We do note efforts at code sharing and open source satellite solutions are increasing, and also that most flight software is developed in C, C++, Ada, or (increasingly) Python (as cited in the extended examples used in this paper). Since the software is being either developed or modified by students on a short time-scale, the three factors in play to consider are whether the students know the language (and conversely, whether the language is easy to learn), whether the language is well suited for the work, and the number of lines of code required to complete a task.

There is always an advantage of having programmers with deep experience in the language, and this can be increased by going with industry-standard languages. Space-X selected Linux and C++ because “there are many more Linux and C++ developers than, for example, VxWorks and Ada developers” [2]. By the same token, we assert that training students up in C/C++ or Python is a career-relevant skill that conveys. Esoteric or advanced languages like LISP and Haskell are therefore exempt from being chosen.

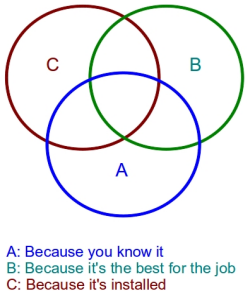

Fig. 1. Rules for creating software

This leaves the category of 'well suited to spacecraft' to consider. For the primary languages currently in use-- C, C++, Java, Ada, Python-- all fit the known/learnable and career-relevance concerns.

Knowledge of a language is a consideration, in that time spent learning a language is time lost to actually creating your software, and that novices will write buggier code than someone with experience in that language. The ability to have multiple ‘eyes’ on the project for debugging is also a factor; even one talented student programming in Rust is a risk if there are no other students nor faculty able to evaluate and debug their code. Therefore, spacecraft software writers should choose a language that is (a) openly available, (b) known to at least a subset of the team, and (c) learnable for those inexperienced with it.

We note two additional factors to choosing a spacecraft-specific language: ability of abstraction (and modularity), and ability to catch errors. And, we agree that, all else being equal, fewer lines of code leads to fewer errors [5].

III. Modularity and Exception Handling

Modularity is key in spacecraft software design. Individual hardware uses software drivers, often developed by component manufacturers (e.g. camera specifications, drivers from the vendor). Message based systems such as cFS or GMSEC use a central software handler (the bus) that polls for and pushes messages to individual software handlers. To add a new component, you write a handler that conforms to the interface or message spec, without needing tight integration.

If not using an existing bus, the team must develop their own bus. The likely designs use a primary loop that acts as an event handler, doing repeated tasks, scheduled tasks, and commanded or triggered actions. This handler can either have all functions written within it (typically as a library of team-created functions) or use sockets or files as a go-between to interact with separate software modules.

For a monolithic library-based approach, a simple looping handler might be (in pseudocode):

while Waiting:
do standard tasks
if (condition Q) do X
if (condition R) do Y
(etc)

Note that in this model, each possible activity is checked by the core routine to see if the conditions for it to be acted on is valid. The code design involves creating a set of rules that are stored within the core. Therefore, adding new routines requires modifying the core.

In contrast, a modular message bus uses an event handler to query autonomous modules for the existence of a task to be done:

while Waiting:
do standard tasks
check the queue for messages,
if so follow up on each

Under this model, the core contains a list of 'clients' to poll for messages, then if a message is received, either acts on that itself or calls a module. With this design, you can write new modules and simply register them to the core, requiring less modification to add modules but more overhead for each module (which must include the messaging API). Neither approach is 'better', but depends more crucially on your framework, your language choice, and your design philosophy.

For this paper, we suggest the advantage is in languages that make it easy to create modules that can be tested individually, then called by the flight software when needed. In a sequential language such as C or Ada, this means creating libraries of functions and subroutines so the code from the design stage is modular. For object-oriented such as C++, Java, and Python (or languages that support objects, like Ada), the objects themselves can provide modularity. The stronger the language support for modularity, the less likelihood of errors, in that a tested module that is validated can remain unchanged as development on further modules continues.

Any language can be misused or written poorly in, but some languages have stronger protection. Ada and Python tend to be 'stricter' in their coding, with strong typing, memory handling, and aggressive compiler (Ada) or interpreter (Python) pre-checking of the code.

Exception and error handling are needed due to the complexity factors in spacecraft. These include health and safety monitoring of nominal operations, communication uplinks and downlinks, the duty cycle required between routine and crisis operations before payload data is lost, and data issues (including completion, volume and timeliness, and calibration and validation).

The driver for requiring exception handling is that we can expect anomalies to occur in any of these aspects. A survey of 13 spacecraft over 2 years found 21 anomalies occurred [6], with a frequency of an anomaly about every 7 months of spacecraft operations. Of these 21, 4 had no mission impact, 11 resulted in loss of redundancy, 4 had degraded performance, and 2 lead to loss of mission. 16 of the 21 anomalies were in the data, payload, or attitude systems (with the remainder being launch, power, thermal or ground anomalies).

We assume that anomalies will occur, and that the flight software must catch errors and anomalies, ideally returning the spacecraft to a known, working baseline state. Two schools of exception handling exist. The 'failstop' principle argues code must stop once an exception occurs, because it indicates a bug and therefore the possibility of further problems [7], and notes the safest way is to stop the entire program. This may be a poor option for a spacecraft.

Anomalies are not always terminal; for example, a camera or sensor temporarily not being reached is a rare but tested event that our flight software must support. We prefer that unresponsive components fail gently, and remain in the polling loop for the next cycle of times or commanded events.

As CubeSat rapid development tends to be integration of multiply sourced drivers and components, a more viable approach is the controller must assume any component might fail. An exception that fails back to the main loop is therefore valid. Examples of good try/except use in spacecraft include, in order of effectiveness:

1) if error then skip
2) if error than call error function
3) if error than retry
4) if error than restart loop

The first two are preferable. Immediately and repeatably retrying a non-responsive component can lead to a blocking fault in the loop code. Similarly, restarting a loop due to an error can prevent subsequent necessary commands from being executed. Therefore, we adopt the philosophy that errors should be skipped, and crucial errors found in testing can be provided with error-handling code as testing and time allow.

A 2008 study noted that 1-5% of open source Java code is typically devoted to catching exceptions, and 3-46% of code then used to resolve exceptions (and that in the 80s, a similar survey indicated that up to 60% of code was devoted to exception handling) [8].

In terms of language support for exception handling, C is weak due to the lack of an internal try/except method.

Ada, Python, C++ and Java contain control structures to allow exception handling within the code. To implement error handling for Cactus-1, we use the exception handling built into Python. Because of its stronger exception handling using Try/Except clauses, our Python code can survive temporary or permanent errors in called modules.

The final factor we return to is lines of code required to complete a task, because it is easier to review a smaller code base than a larger one. The more code to create and test, the larger the chance of error. A 2013 report asserts Ada is preferable to C [5] in part because a task takes fewer lines in Ada than C, allowing for inspection as one test method. Some of this can be mitigated by modularizing your C code into libraries, however, effective object passing in C is itself a source of errors. Also, codes that emphasize readability and modularity (especially Ada and Python) are higher-level languages that require fewer lines to serve most spacecraft tasks.

For an example of readability, here is a a subset of a serial camera read on a Raspberry Pi using C versus Python (with error checking and comments removed), to illustrate the logic.

```
# In C (18 lines):
cam->frameptr = 0;
clearBuffer(cam);
serialPutchar(cam->fd, (char)0x56);
serialPutchar(cam->fd, (char)cam->serialNum);
serialPutchar(cam->fd, (char)FBUF_CTRL);
serialPutchar(cam->fd, (char)0x01);
serialPutchar(cam->fd,
 (char)STOPCURRENTFRAME);
unsigned int len;
len = serialGetchar(cam->fd);
len <<= 8;
len |= serialGetchar(cam->fd);
len <<= 8;
len |= serialGetchar(cam->fd);
len <<= 8;
len |= serialGetchar(cam->fd);
int32 pic_fd = OS_creat(file_path
 (int32)OS_READ_WRITE);
OS_write(pic_fd, (void *)image, imgIndex);
OS_close(pic_fd);

# In Python (12 lines):
ss = serial.Serial(PORT, baudrate=BAUD,
timeout=TIMEOUT)
reset(ss)
takephotocommand = [0x56, 0, 0x11, 0x00]
cmd = ''.join (map (chr, takephotocommand))
ss.write(cmd)
reply =  ss.read(5)
bytes = getbufferlength(ss)
photo = readbuffer(bytes,ss)
f = open(camfile, 'w')
photodata = ''.join(photo)
f.write(photodata)
f.close()
```

From a logic point of view, both are equivalent. Both examples also require non-intuitive documented driver calls (the 0x56, 0x11 and other byte patterns needed). The Python commands have higher clarity due to Python's typing, use of objects instead of pointers, and focus on readability. While the C version requires bit shifting logic (with “len |=” and “len << =8”), the only arcane Python commands are the uses of “join” and “map”. Python also more strongly separates data items (such as the byte specifications for the driver) from actions (generic items such as 'ss.write' and 'ss.read' to write/read from the serial port).

Readability is extremely crucial when considering software hand-offs and maintainability. As a university, we cannot assume continuity of developers. Each sounding rocket had a slightly different team, and Cactus-1 consisted of different phases of work spread out over 3 years (a very long time relative to an undergraduates' time availability).

In addition to readability, languages with a shorter development time leave more time available for testing and debugging. Code familiarity likewise speeds development time. Given these time factors, human factors in language choice again are more important than performance.

IV. Cactus-1 Use Case

Our Cactus-1 CubeSat system consists of health and safety sensors, two internally facing cameras, and a half duplex radio communications module. Our language-agnostic use case is on par with the needs of similar picosatellite missions. Our mission, being bandwidth-limited, with 1 ground station and no capability of changing on board programming, requires the ability to send command mnemonics, optionally overwrite stored mission parameters such as cadence and instrument modes, transmit health-and-safety packets plus image data, and comply with FCC shut-down requirements. A simple outline (in comment format) of our flight controller design is given as:

```
# “Cactus-I Flight Software”
# Initialize all systems
# have an OS-level monitor in case of crashes
# program eternally loops
        # get current time
        # CHECK FOR INCOMING COMMANDS
        # (file-based or listen-based?)
                # loop through what was received
                # check if it's valid, log errors
                # then carry out its function
        # REGULAR TIMED TASKS
        # HK data gather
        # payload data gather
        # optional sleep? if it saves power
```

We use the best practice of defining our code in comments, then adding code to the comments to build up the software, rather than writing code and commenting later. This allows the code to self-document and ensures the documentation matches the actual flight code.

Our command mnemonics are few. Table 1 defines our primary command mnemonics and their functions, which

also shows the low level of complexity required by our software. Most commands include optional arguments to set payload, power or communications parameters to values other than the pre-loaded defaults (that get reset to defaults when the base command is later issued).

The commands were defined by the mission and payload requirements, and all count as essential required items that the flight software must support. The use of optional arguments is a desired but not required feature set. In implementation we chose that overriding of default values does not persist across reboots.

V. Pico Agile Design

Regardless of language choice, the flight software must meet requirements and the payload must ship on time. Spacecraft launch times set a firm ship date, therefore software development cannot slide to later. Traditional “waterfall” development using Gantt charts and similar scheduling tools are one method for developing your software. However, given the rapid development time and small team size, we suggest Agile methods as being equally strong for managing your development.

Agile methods introduce a Burn Chart as a way to track the completion rate of a software project. In a Burn Chart, you start with 0% done and finish at 100% done. Agile typically creates “User Stories” as a task list of items that must be accomplished, and the Burn Chart counts down the number of User Stories left to implement.

For spacecraft software, you can consider the User Stories as the requirements and feature list, and that implementing each requirement and feature moves you closer to completion. In this fixed time environment, there are two approaches to completing software. If software development is running late, you can either add resources or reduce the feature set of the software, as shown in Figure 2.

Requirements are defined as items that must be complied with. Features are the set of items that would be useful but are not essential to flying. When designing flight software, 100% of requirements must be met, and then features added up until the software freeze date, after which no changes other than mission-critical bug fixes should be done.

We used an Agile approach named “Pico Agile” [9], which is an Agile method for project instantiation and control, but without formal Agile practices for day-to-day work. Pico Agile can act as an overlay with other Agile processes, serves as a work-flow version of a to-do list, and is highly suitable for solo projects or small teams with fixed resources on a set schedule.

Table 1. Cactus-1 Command Mnemonics

Core Commands		
Mnemonic	Functionality	Italic
downlink	high-power no-wait downlink of all data for up to 10 minutes	
beacon	sends only health and safety packets	
longwait	put into a low power mode for specified interval	
cadence	set alternative data modes and/or rates Subheading	
sporton/ sportoff	trigger mode for AMSAT outreach	
Anomaly Modes		
burn	re-trigger burn wire to re-try antenna deploy	
panic	start 1 day of beaconing	
fccoff	immediate cease of all transmissions	

Under Pico Agile, you create a 'to do' list of items in 4 categories: Prework/Infrastructure, Must Do Tasks, Should Do tasks, and Would Like tasks. Prework would be, for example, installing your development environment, databases, and outside packages and frameworks required.

The Must Do tasks are driven by the requirements and include basic functionality-- typically housekeeping, payload data gathering, and communications as a minimum set. Missions that have active power monitoring, attitude and/or maneuver control, and deployables would likewise add those to the Must Do category. FCC and launch provider requirements are also Must Do.

Should Do tasks include additional functionality as well as improvements to Must Do tasks. Should Do items often cross subsystems. For example, with Cactus-1, one Must Do requirement is that the system take an image of the Aerogel orbital debris capture substrate once per hour (minimum); a Should Do task is to be able to change that cadence to a ground-specified setting via commanding. Would Like tasks are the lowest priority items that, as the name states, are desirable but not core to the mission.

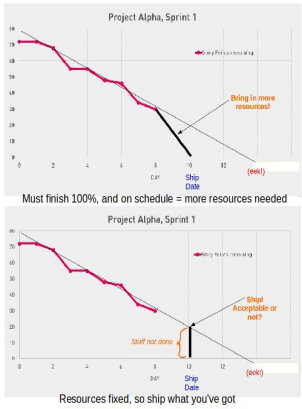

Fig. 2. Two burn approaches for fixed due date

Once you have your categories, the next step is to add Dependencies in order to track which items have other items that must be completed first. We suggest 3-digit #s, starting at 010, then 011, 012, 020, etc-- hopping in steps of 100 (categories) and 20-30 (feature sets)-- this leaves room to add items later and avoids the “number #1 must be most important” fallacy, unless it is actually relevant.

Use the numbers to build chains via increments for easier tracking. A sample roster of items for the Cactus-1 TrapSat payload system would be:

PREWORK:
- 001 install camera drivers

MUST DO:
- 100 take and save images
- 101 switch MUX to choose cameras
- 102 illuminate camera LEDs
- 200 take and save HK data

SHOULD DO:
- 130 set camera cadence rate
- 131 change camera resolutions (req 100)

WOULD LIKE:
- 170 add difference images (req 100)
- 270 trend HK data (req 200)

The last step is to ignore the Would Like items-- pragmatically, for most missions, the bulk of these will not be implementable in the time available. They are worth keeping for documentation purposes as well as in case someone wishes a side project, the mission launch is delayed, or if their implementation is trivial.

VI. Existing Frameworks and Heritage

Part of deciding on a language is evaluating the utility of existing frameworks. Frameworks can be previous versions of your flight software or external flight stacks provided by peers or the community. We briefly examine two community-provided frameworks: Core Flight System (cFS) and CubedOS, as well as our own multi-mission heritage with C and Python.

Core Flight System (cFS) is a C-based framework from the Flight Software Systems group at NASA/GSFC and is freely available. It is a bus that allows developers to write modules according to their Application Programming Interface (API) rather than have to work with the core code base [10. It is in C, and has a high level of abstraction through the API. In addition, user-written applications run as services and thus are added or removed during runtime.

The ADA language has the SPARK framework to enable good abstraction and better capture of errors [11]. One implementation is the CubedOS framework, consisting of 5991 lines of code-- of which 4095 are comments and SPARK test-case annotations [11]. The CubedOS team likewise maintain a C code base of 2239 lines. Their architecture is, like cFS, a message passing framework.

Heritage does not automatically guarantee success. We note the failure of the Ariane5 launch is attributed to code re-use, with Ariane 5 re-using the 10-year old Ariane 4 software and running into an integer conversion problem as a result [12]. We assert that you need to understand all of your code, even heritage code that has performed well but under a different hardware configuration or for differing data needs.

Internally at Capitol Technology University, we use a 'crawl, walk, run, fly' mission progression from lab through to high altitude balloons, to sounding rockets, culminating in the CubeSat [13]. This provides hardware continuity across projects. Similarly, we retain the software drivers for each component. However, the command and control needs for each step are vastly different, as are communication capabilities. Therefore, the core controller for our flight software stack is not guaranteed to retain heritage.

For our sounding rockets, the initial C based software was later merged into cFS. This two-step integration from a simple stand-alone C stack to the more robust cFS environment is equivalent to two complete software development efforts, as the integration is not trivial and required at least two developers for multiple semesters. For our Cactus-1 CubeSat, we found that prototyping a test Python-based flight stack was a task the co-author was able to achieve in under half a semester, and the author was able to then write the actual flight Python stack in a similar interval. This is due to the readability and modularity of Python as well as the small size of code our mission requires.

We therefore kept heritage for the hardware drivers and serial port protocols used, but did not retain heritage by continuing with cFS or revising the Python prototype. Instead, we used our Python payload and communications modules developed during building our flatsat, and integrated them under a controller program. Our flight controller model is simple enough that the redesign was preferable to refactoring.

We advocate Python as a strong language for flight software, and that is what we used for Cactus-1. Not counting drivers (which have to be written for hardware regardless of bus design), we can compare the development statistics for a C-based system to a Python-based system. We use as our cases our TrapSat Sounding Rocket (flown under NASA's RockSat-X program) as comparable to our Cactus-1 CubeSat (which flies the same detector).

Both use passive power monitoring (TrapSat via the rocket power line, Cactus-1 via a self-regulating solar-battery bus). TrapSat sent comms data via the parallel port; Cactus-1 operates a radio module via the serial port. Both have health and safety Housekeeping (HK) data. Cactus-1 can be commanded; TrapSat relies entirely on a pre-loaded schedule.

The comparison of the TrapSat sounding rocket,which used C and cFS, and the Cactus-1 CubeSat, which used in-house Python code, provides numerics on how language choice influences readability, code complexity, and the ability to transfer code across teams.

The TrapSat Sounding Rocket flight used a Raspberry Pi plus an Arduino, and included comms via the sounding rocket interface. TrapSat used cFS as its bus. Written in C, cFS itself can be considered stable and tested, so the code requiring creation and testing are the TrapSat applications.

The payload data interface consists of 6 C routines totaling 1127 lines. The comms/data handling interface is 7 C routines totaling 1394 lines. The health and safety monitoring is 5 C routines totaling 792 lines. This totals 3313 lines.

In contrast, Cactus-1 (shown in full in Figure 3) uses a Raspberry Pi. It has 1006 lines of Python code in 4 modules: a main Core controller program (147 lines), a library of core Bus functions (361 lines), a library of Payload-specific functions (463 lines), and a shared Config file (35 lines).

Half to two thirds of each routine is comments (e.g. only 57 of the 147 Core lines are actual code). Roughly half the code resides in the payload-specific library. The Core controller is small and easily tested. The Bus functions have high criticality, as they are responsible for communications and underlying bus functionality.

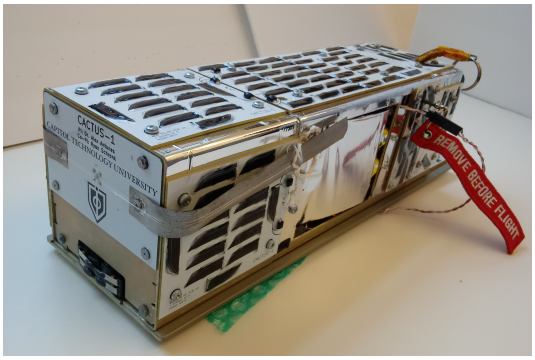

Fig. 3. The Cactus-1 3U (10cm x 10cm x 30cm) CubeSat

In terms of readability (for which Python is famous), the Cactus-1 code is highly readable, well commented, and has a high level of abstraction (very few lines of code per function). This makes it easier to visually debug, as well as making it straightforward to place try/except error handling checkpoints.

From a coding point of view, cFS requires more expertise to interface. The amount of new code required by TrapSat is roughly 3 times the amount of new code required by Cactus-1.

In practice, each subsystem hardware module from Cactus-1 during development was tested prior to integration. Therefore, the Cactus-1 flight controller is the least tested subsystem at the time of integration. Conversely, with TrapSat using cFS, the software and hardware were well tested but each cFS module was untested at the time of integration, because the prior lab bench code did not yet have the cFS bus protocols.

From a testing point of view, since Cactus-1 had to write their own bus, more testing time is required at the bus level before working at the Payload level.

In our cFS experience, its advantages is that it worked well for RockSat. cFS is also more scalable; once you've written one module, it is easier to write additional modules as you move to increasingly more complex spacecraft needs. cFS's primary disadvantage was its learning curve; we needed to learn its message passing API.

The advantage of the Cactus-1 Python system is the small code base, its simplicity, and that it is easy to find student Python developers. Its disadvantage is that the system interactions are more tightly coupled in the core function and thus you must modify the core bus when adding capabilities.

With this comparison, from a time resource stance, cFS is essentially overkill for a simple mission like Cactus-1, but could be useful if you have time and C developers. If you are not CS people (just engineers and scientists, like us), maybe not.

However, from a testing standpoint, the role-up of individual Python modules into our core scheduler and the great reduction in lines of code to track make the Python-based environment stronger for testing.

VII. Conclusions

When developing flight software, it is important to evaluate your language and framework choices not just on outside assessments of what is best or most commonly used, but more crucially based on your specific use case (especially regarding complexity and re-programmability) and the best use of your existing talent pool.

The choice of packages and languages for flight software should be made on the basis of team abilities and soft factors as much as on the language's capabilities. We favor continuity of developers over requiring high heritage for software. Agile processes and delineating which features are requirements and which are optional will be crucial, as spacecraft launch dates firmly set your development schedule.

Modular design that lets you inherit software components across prototypes and test builds will speed development and improve reliability. For language

choice, robust exception handling is a must, as anomalies should be expected to occur during a picosatellite mission. Additionally, when in doubt, going with known languages that are career-applicable is an appropriate decision in a university setting.

Ultimately, your pre-coding choices should aim for a faster development time producing readable code that will free time for more testing, as robust and frequent testing is a strong predictor of mission success. There are several existing frameworks and upcoming open source flight stacks available, and they should be considered within the use case of your specific mission. We suggest the use of Linux and Python, as with Cactus-1, as a good default choice for new teams looking at possibilities.

ACKNOWLEDGMENT

Support for Cactus-1 was provided by the Maryland Space Grant Consortium (MDSGC). The Cactus-1 launch opportunity is provided by the NASA CubeSat Launch Initiative (CSLI)..

REFERENCES

[1] Swartwout, M. (2018). Reliving 24 Years in the Next 12 Minutes: A Statistical and Personal History of University-Class Satellites. *32nd Annual AIAA/USU Conference on Small Satellites.*

[2] Leppinen, Hannu (2017). Current use of linux in space flight software. *IEEE Aerospace and Electronic Systems Magazine* 32(10), 4-13. DOI: 10.1109/MAES.2017.160182

[3] Kalman, A. (2012). Adapting Linux-based Computing to CubeSat. CubeSat Developers' Workshop/26th. *Annual AIAA/USU Conference on Small Satellites.*

[4] Chapin, P. (2019). Ten Years of Using SPARK to Build CubeSat Nano Satellites With Students. Retrieved from https://blog.adacore.com/ten-years-of-using-spark-to-build-cubesat-nano-satellites-with-students (Original work published 2019)

[5] Brandon, C., & Chapin, P. (2013). A SPARK/Ada CubeSat Control Program. Reliable Software Technologies – *Ada-Europe 2013*, 51-64.. DOI: 10.1007/978-3-642-38601-5_4

[6] Squibb, G, Boden, D., & Larson W. (2016). *Cost-Effective Space Mission Operations (2nd edition)*. McGraw-Hill.

[7] Weinreb, D. (2013). What Conditions (Exceptions) are ReallyAbout. Retrieved from https://web.archive.org/web/20130201124021/http://danweinreb.org/blog/what-conditions-exceptions-are-really-about (Original work published 2013).

[8] Weimer, W., & Necula, G. (2008). Exceptional Situations and Program Reliability. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30(2). DOI: 10.1145/1330017.1330019

[9] Antunes, A. (2019), Pico Agile [PDF file]. Retrieved from http://ghostlibrary.com/ag (Original work published 2019)

[10] core Flight System (2019). Retrieved from https://cFS.gsfc.nasa.gov/Features.html (Original work published 2017)

[11] Brandon, C., & Chapin, P. (2017). CubedOS: A SPARK Message Passing Framework for CubeSat Flight Software. *SPARK/Frama-C Conference*.

[12] Jézéquel, J.M., & Meyer, B. (1997). Retrieved from http://www.irisa.fr/pampa/EPEE/Ariane5.html (Original work published 1997).

[13] Schrenk, R., Strittmatter, M., Walters, A., & Jagarnath, M. (2017). Crawl, Walk, Run, Fly. 2017 *Academy of Aerospace Quality Conference Proceedings*.

Authors' Profiles

Alex Antunes has a Ph.D. in computational sciences and informatics, specially computational astrophysics, from George Mason University, 2009, and a Masters of Science in astronomy from Pennsylvania State University, 1992. His field of study addresses open source satellite work, cheap deployable sensors, machine learning, and engineering education.

He is a professor of astronautical engineering at Capitol Technology University in Laurel, MD. Prior work includes spacecraft science operations and astrophysics work at GSFC, solar physics at NRL under an NRC fellowship, design of the DIY TubeSat "Project Calliope", and freelance science writing. In addition to scientific research on solar physics, X-ray binaries, and collisional remnants, he has written five "DIY" technical books for O'Reilly/Maker Media on amateur space and machine learning topics.

Dr. Antunes is on the advisory board for the NASA Academy of Aerospace Quality (AAQ) and is a member of American Geophysical Union/AGU, American Astronomical Society/AAS, AMSAT, National Association of Science Writers/NASW, DC Science Writers Association/DCSWA, International Game Designers Association/IGDA, American Society of Engineering Education/ASEE, American Institute of Aeronautics and Astronautics/AIAA.

Randy Powell graduated with a Bachelor of Science in Computer Engineering from Capitol Technology University in May 2019. He worked with the Cactus-1 CubeSat mission as an undergraduate. He works in Systems Engineering at General Dynamics Mission Systems. He is skilled in PHP, Python, Java, C, C++, and Perl.