SSC17-P1-41

Building Modern Cross-Platform Flight Software for Small Satellites

Ryan Plauché
Kubos
525 North Elm Street, Denton, TX
ryan@kubos.co

ABSTRACT

The current software ecosystem for small satellites is characterized by proprietary libraries, tight coupling with hardware and little thought for interoperability. These qualities also marked the software found in the early days of the computing revolution. As modern computing has advanced and hardware platforms have standardized, there has been a rise in software packages which operate using open standards and are designed with multiple platforms in mind. The software surrounding small satellites is posed to undergo a similar revolution. Increased interest in the market will only drive the need for flexible, reusable software libraries. This paper discusses the approaches used to create flight software capable of reuse on multiple small satellite platforms.

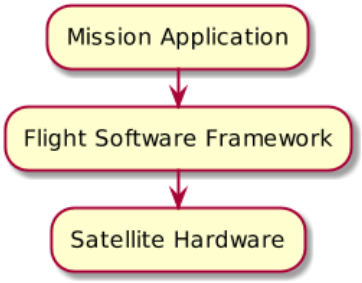
INTRODUCTION

Over the past several years the small satellite market has seen significant growth, particularly in the cubesat sector[1]. An increasing number of missions and new opportunities have brought about numerous innovations. The majority of these innovations have dealt primarily with hardware components. However, small satellite software has not benefited from the same level of attention. In order to fully leverage new hardware functionality, software engineers must have equally innovative components to work with.

An inevitable consequence of increased small satellite hardware innovations is a fractured hardware ecosystem. Driven by evolving customer needs, vendors introduce new hardware platforms and components. While this variety gives developers expansive choices when planning a mission, it complicates software development[2]. Engineers are given the choice of either using proprietary vendor software, or developing their own from the ground up.

The greater community does not provide many options to leverage in the software development phase. Frameworks that support hardware from multiple vendors are absent for the most part. A search for community based, or open source, software projects reveals a variety of work designed for specific missions or platforms. In most cases the engineer looking for reuse would be forced to perform a fair amount of refactoring before the software is suitable for their specific use case.

Kubos has invested significant time in the design and creation of KubOS, a cross-platform flight software framework[3], in an effort to promote rapid development and future innovation. This paper presents the thought process and high level design used in the creation of this framework.



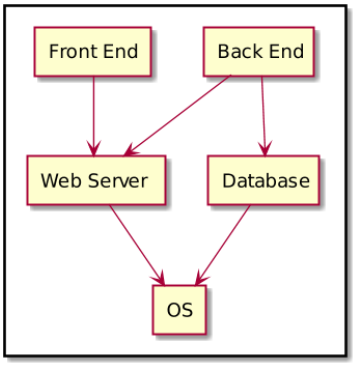
High Level Framework Context

Kubos emphasized high software reuse, interoperability and cross-platform compatibility when architecting and implementing KubOS[4]. A top down design approach was taken: Kubos found a high level architecture and used it as a structure for details and abstractions. The chosen architecture here is the “full stack”, a term borrowed from the web development sector, and more formally known as an *N-tier architecture*.

An N-tier architecture allows for a clean separation of concerns[5], and accurately models the layers created in embedded software development. At its core, a software architecture is just a framework and must be filled with the practical details necessary to fulfill specific use cases. The requirements of this framework have been split into three major categories: mission objectives, satellite services and hardware interaction. N-tiers of software span across these categories, creating rich functionality on multiple operating systems and hardware platforms.

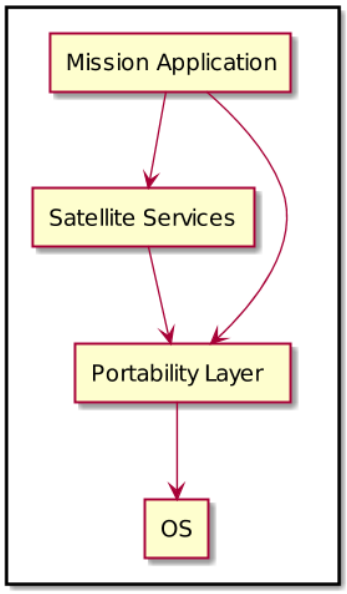
SYSTEM ARCHITECTURE

Full stack is a term originating in the world of web based software. This term typically refers to the layers of software and infrastructure required to build and run a web site. The typical full stack consists of an operating system, database, web server, server-side or back end software and client-side or front end software. All of these layers are required in order to receive, process, render and deliver requests for web pages. Each of these layers have different, well established responsibilities in this process. The full stack implementation of the N-tier architecture provides a robust model for separating concerns and responsibilities—one that is invaluable when building a software framework.



Web Software Full Stack

Designing a framework for satellite software requires evaluating the different requirements necessary for a successful satellite mission. Three major categories are immediately obvious: mission objectives, satellite services and hardware interaction. Mission objectives encompass any special data gathering, processing, or actions which are mission specific. Satellite services are common high level software components used in most missions. Lastly there is the need to operate on and interact with mission hardware.



Satellite Software Full Stack

The end goal of the framework must be to fulfill the requirements of all three categories. However the way these requirements are met varies. Frameworks exist not as complete solutions, but as tools to enable and empower. In this case the framework enables developers to meet their mission objectives. Software components that fulfill those objectives will be specifically tailored to the mission at hand. These components will be built on top of the framework, not included in it.

Satellite services is the next category serviced by the framework. Most frameworks include a collection of high level software functionality specific to the task at hand. In this case that functionality is exposed through a series of middleware services. These should be specifically designed to meet common needs which arise in most satellite missions. They are intended to be leveraged and reused across multiple mission implementations.

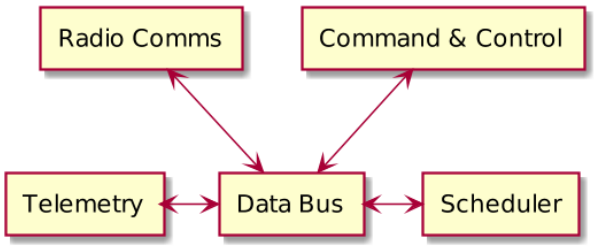
Lastly, a cross-platform framework must include components that enable hardware integration on various platforms. This portability layer has the dual purpose of abstracting away both hardware differences and operating system differences. Common hardware peripherals and interfaces must be identified and hidden behind unified software interfaces when possible. Likewise, common operating system metaphors and operations should also be unified. Other layers of the framework need not concern themselves with these differences.

MIDDLEWARE

As previously discussed, a major responsibility of the software framework targeted at enabling small satellite missions is providing common functionality unique to that domain. These satellite services, or middleware, provide high level functionality for application consumption. The middleware is built on top of the portability layers below it, and provides great leverage for rapid development. The middleware layer is the main difference between a framework enabling satellite-specific work and one that merely enables typical embedded work.

Small satellite software comes with its own specific set of challenges and goals that must be taken into account when choosing and designing the middleware services. Specific needs can and will be discussed, but it is important to start with the bigger picture in mind. The majority of a satellite's requirements deal with data. A satellite that is unable to send or receive data from the ground is useless. Likewise, a satellite sent into deep space without any method for data collection would be a waste of resources and funding. The routing, collection and transmission of data must be prioritized in the design of high level software services.

A system that prioritizes data should have an independent system solely for data routing. This data bus will give other services in the system access to interfaces for data submission, storage and retrieval. The bus must be independent of other services to ensure that no one client service can interfere with others' interactions with the bus. Initial framework designs have settled on a publisher-subscriber style data bus. The "pub-sub" model allows system services to provide and consume data in a highly decoupled fashion.



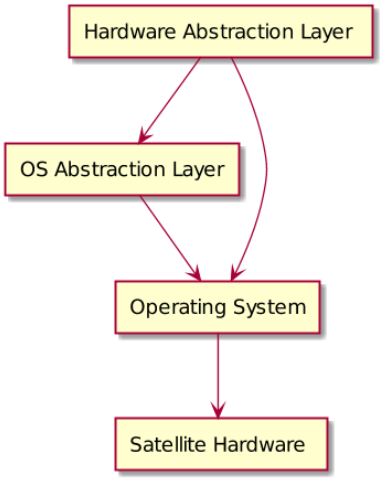
Services Architecture

Individual services in the small satellite system must be designed to serve the common needs and usage of the satellite. For instance, a graphical interface service would serve no useful purpose. However, a service that responds to ground station commands, or a service that controls the power, are each essential. The selection of specific services to implement must be a careful and intentional process. Common use cases that demand services include radio communications, command & control, activity scheduling, power management and the attitude determination and control system (ADCS) interface.

PORTABILITY LAYERS

Standardized hardware platforms are rare in the satellite industry. Most missions have unique hardware demands, and the majority of the hardware vendors use different hardware platforms. The variety of different available hardware platforms lends itself to highly customized software that is tightly coupled to mission hardware. While this approach creates high software/hardware compatibility, it usually ensures difficulties when attempting to reuse code. The KubOS portability layer is designed to address these difficulties, fulfilling one of the framework's major set of requirements.

Addressing the portability issue involved evaluating potential software use cases and expected platforms, and determining what could be reused versus what must be platform specific. Three main layers of functionality have been identified for use in building a robust portability layer: the operating system, the operating system abstraction layer, and the hardware abstraction layer.



Portability Layers

The operating system is the base layer of any embedded framework. In order to ensure compatibility with a range of small satellite hardware, two base operating systems were chosen: FreeRTOS and Linux. FreeRTOS is targeted at platforms with significant resource constraints, Linux provides developers with a more common tool set to leverage. Both of these operating systems are well known and come from the greater open-source commu-

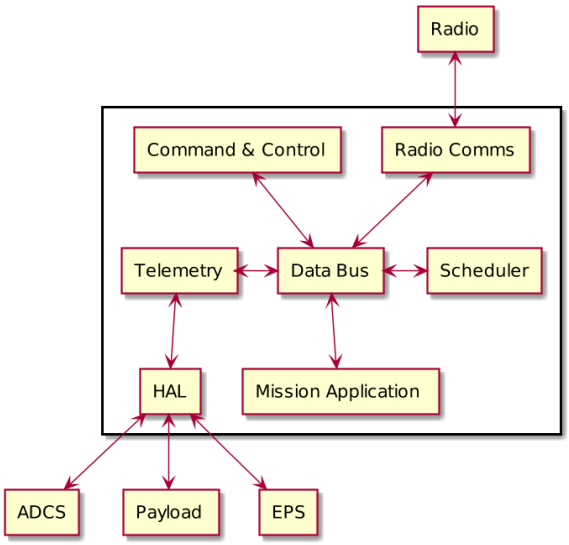
nity. Achieving hardware compatibility involves porting this base operating system to the desired hardware platform.

The operating system (OS) abstraction layer rests on top of the operating system. Different operating systems, including FreeRTOS and Linux, have different interfaces for dealing with common OS functionality. Programmers typically implement interfaces such as threads, semaphores and message passing in a heterogeneous fashion. A well designed OS abstraction layer allows the developer to build against a high level OS interface, rather than concerning their code with OS differences, or tightly coupling their software with a specific OS. This abstraction layer also opens up the possibility for running existing software on currently unsupported operating systems.

The hardware abstraction layer (HAL) lies alongside the OS abstraction layer and on top of the OS. The purpose of the hardware abstraction layer is to create unified software interfaces for common hardware interfaces. These hardware interfaces include system buses such as UART, SPI, and I^2C. Other hardware interfaces include higher lever functionality such as flash memory (SD cards, e.g.) support. The HAL abstracts away two main pieces for developers: platform specific hardware differences, and low level OS-to-hardware interfaces. Various microcontrollers (MCUs) require different steps to enable and use hardware functionality. The HAL takes these steps and hides them behind higher level interfaces. Likewise, different operating systems may expose hardware to user applications in different ways. The HAL also serves to take these differences and place them behind common interfaces. This design enables reuse of the same high level SPI code on an ARM-based platform running Linux as a MSP430-based platform running FreeRTOS.

SUMMARY

The implementation and integration of software layers across the three categories discussed—mission objectives, satellite services and hardware integration—allow for the creation of a rich satellite software ecosystem. All three categories are essential for allowing developers freedom of choice when selecting hardware and designing software. A complete and functional software ecosystem eases the process of integrating hardware components and subsystems, and grants custom software reliable access to needed data.



Detailed System Architecture

Creating a multi-platform software framework is not a simple task, and tailoring one for small satellite systems only increases the potential complexity. Software frameworks form the foundation for future software development, and the framework can either serve as an enabling boon or a hindering nuisance. Only careful, intentional design decisions and mindfulness of the domain will result in long term fitness. Functionality, layers and interfaces must all be crafted with the goal of simplicity and reusability in mind. The KubOS framework is designed around these concerns: using development platforms like this one, engineers can better understand, leverage and build software to achieve their mission goals.

Notes

[1]de Selding, Peter B. 2016. "The state of the satellite industry in 5 charts." *Space News Magazine*, June 20, 2016.

[2]Stringham, Gary. "Basics of hardware/firmware interface codesign." *embedded.com*, July 07, 2013.

[3]Kubos GitHub repository, accessed June 11, 2017.

[4]Culpepper, Marshall. "Eating Space with Software." Kubos blog, April 6, 2017.

[5]"Multitier architecture", Wikipedia. Accessed June 11, 2017.