The Phoenix Architecture: A New Approach to Student Satellite Software

Riley Pack
University of Colorado at Boulder
Advisor: Christopher Koehler
rileypack@gmail.com
April 17th, 2010

Abstract
At many universities, the prevalent approach to flight software development on student satellite projects involves writing a large amount of custom software. While this method ensures that the software will conform perfectly to the unique hardware constraints of the mission, it also means that the developers must spend a majority of their time building and testing a brand new architecture instead of directly focusing on supporting the mission. The resulting software is often delivered behind schedule and under-tested. The Phoenix Architecture aims to provide an alternative to this method by giving software developers a highly-adaptable, hardware-independent flight software platform for satellite projects. The architecture is built around an extensible message protocol that allows separate modules to communicate via the central message dispatcher. Each module is then built with the explicit requirement that it cannot contain a hard dependency on any other module, making the system flexible and configurable. Furthermore, the architecture decouples the application software from the hardware by defining a hardware abstraction layer (HAL). Once the HAL has been implemented for a particular hardware platform, the entire Phoenix framework will run seamlessly on that platform. With these two entities as a foundation for the rest of the codebase, as well as a library of modules that can be added to by each mission, satellite programs that choose to use Phoenix will be able to focus their development efforts on interfacing with the payload. Ultimately, this shift in focus will result in a shorter development cycle and more reliable flight software, which will greatly improve the chances of the mission succeeding.

1. Introduction

According to Frederick P. Brooks, author of one of the best-known software engineering essays [1], software development suffers from four fundamental, essential difficulties: complexity, conformity, changeability, and invisibility [1]. Many of the software development methods and tools available today do not directly attack these essences, but instead the so-called accidents of software development, such as programming languages and syntax errors. These accidents are themselves not inherent to software, but instead to the fact that humans must write software. In order to obtain an appreciable gain in development time, the four essences mentioned above must be attacked.

The Phoenix Architecture aims to improve upon three of Brooks’ essences. First, by providing a modularized, decoupled software system, the overall complexity of the system as seen by a programmer is reduced. Any feature being worked on by a developer is isolated from the rest of the system by the command interface discussed in section 2. Second, by enforcing a strict, simple interface, the software does not have to conform to a complicated interweaving of interfaces. Instead, all components must talk to each other through the same command interface, which reduces the restrictions on each module. Finally, because each module must be capable of operating without the presence of any other module, the Phoenix Architecture allows for a great level of changeability. Any part of the system can be removed without breaking the rest of the system, which makes changes much easier.

As described above, the Phoenix Architecture will attack the difficulties of student satellite software development with three major components. The first, called the Phoenix Core, will provide the necessary interfaces to each module so that they can communicate efficiently. The second component is the hardware abstraction layer, which removes dependencies between the high-level architecture and the low-level hardware dependencies. Finally, the architecture defines an interface for any number of modules, which are referred to as “servers” within the architecture. A server is a wholly-contained software component that provides a service to the software system, such as supporting a communications device or tracking the mode of the satellite. The servers within Phoenix are the elements that make the architecture flexible and easily adaptable.

To further improve the usability of this architecture and to make it easier to expand, a primary goal in its development will be to use object-oriented design principles. The implementation of all software modules will be achieved using the C++ language. This language offers a convenient mix of the low-level functionality present in C with the many high-level features of a full
object-oriented language. The language tools present in C++ will provide a stable development platform for the Phoenix Architecture.

2. Development Philosophy

Before discussing the design of the Phoenix Architecture, it is important to discuss the philosophy and practices used while designing and implementing the architecture. This project was created with the intention of following Agile development processes throughout its development. As a result, the individual details of each component have not fully been designed down to the function and variable. As the past 30 years have shown software developers, the so-called waterfall method often leads to software that is hard to manage due to feature creep and that does not meet the needs of its customer or user.

Agile processes aim to provide a solution to this problem. Instead of fixing the design before any development has begun, the software is grown from the ground up through iterations of development. In any given iteration, features (which are supplied by the customer) are implemented one-by-one in the simplest manner possible. No additional features that “might be needed later” are added, as they probably won’t be needed (and if they are, they’ll be added formally when a feature is introduced). The resulting code is easier to manage because there are no extraneous methods that need to be updated as the design changes.

Another important component to Agile development is the use of unit tests. The entire Phoenix Architecture will be implemented alongside a full suite of unit tests written from each feature description. Therefore, unlike much of the code written using traditional methods, the classes used in the architecture will be tested as they’re written. These unit tests will provide a guarantee that the code performs its required functionality, help prevent bugs, and lend credibility to the quality of the code.

Throughout the rest of this paper, there will often be notes about an initial approach followed by a correction. These statements are intended to capture this process and the mantra of always writing the simplest code first and then adapting and refactoring it as needed to make it easier to use and maintain. In addition, the latter parts of the paper will outline a probable approach to the features that they describe. Because these portions of the design have not yet been implemented, a concrete, class-by-class description cannot be provided; however, the overall idea behind any component will not change, as they represent the description of a feature (or many features) by the customer.

3. Core Architecture

The Phoenix Core provides the many software components that are required to support a flexible and extensible architecture. The main parts of the core are a message dispatching system, a logging system, and a file system. Each part will be discussed in detail below.

Before going into further detail with the architecture, it is helpful to get a better high-level understanding of the software interfaces. A functional block diagram of the Phoenix Core is shown in Figure 1. Each of the major components of the Core can be seen in the diagram, as well as their major interactions with each other and with other software components. In general, a software module can communicate with any other module by sending a message through the message dispatching system. The module can also interact with the satellite hardware through the hardware abstraction layer. These two components allow a server to perform a wide variety of actions on the satellite while maintaining a consistent interface.

![Figure 1: Phoenix functional block diagram](image)

3.1. Message Dispatcher Overview

The central component of the entire Phoenix Architecture is the message dispatching system, known within the architecture as Dispatcher. The main purpose of Dispatcher is to allow Phoenix servers to send messages to each other. In order to facilitate this functionality, the architecture also defines a message protocol and a permission arbitrator. If a server needs to send a command to or request data from another server, it creates a packet and sends it to the recipient. The arbitrator then checks the packet to ensure that the sender has permission to make the request in the packet. Once
the packet has been allowed through to the server, it is interpreted and handled by a user-defined message handler. Finally, the response is sent back to the sending server with either the result of the operation or the data that was requested.

The message protocol defined for the architecture defines informational and command packets that can be sent from server to server, as well as from the ground segment of the satellite to any server. Each packet contains a source, a destination, and a number. These fields allow the dispatcher to send the packet to the correct server and for the server to know where to send the response. The rest of the packet is composed of a message to the given server, which can currently be one of four types: a command message, a data message, an error message, or a configuration message. Each message first indicates its type, whether it is a response to a previous message, and if it is a response, whether the original message’s operation was successful or not. Then, each message contains message-specific data. A command is accompanied by a list of parameters that can vary from command to command, while both data and error messages contain a single data element that either contains the requested information or elaborates an error (note that an array can be considered a single data item). Finally, a configuration message contains a list of settings, which indicate the configuration item they describe and the value for that item.

As mentioned above, the permission arbitrator for a server ensures that every message sent to that server is allowed by the current satellite mode. Each arbitrator object keeps a list of permissions, one for each message type recognized by its associated server. The permission describes which servers are allowed to send the associated message to the receiving server. If a given message does not have a permission value defined within the arbitrator, then the message is assumed to be forbidden, which protects the server against unsupported messages.

3.2. Message Dispatcher Implementation

Because any reasonably-sized satellite software system will likely need to be managing several tasks at once, an operating system will need to be used in the Phoenix implementation. After searching through the many operating system choices for medium-complexity embedded microcontrollers, it was decided that FreeRTOS [2] is the best choice for the Phoenix Architecture. FreeRTOS is a lightweight, preemptive, real-time operating system that supports several parallel tasks, semaphores and mutexes, and message queues. Furthermore, it has been officially ported to 24 microcontrollers [2], which allows a system design more flexibility in the microcontroller used on their project. The use of the operating system will greatly simplify the implementation of the dispatcher, as well as allow the decoupling of each server from the rest of the system through the use of concurrent tasks.

In order to fully describe the many components of the message dispatcher, many helper classes are also necessary. The first, called VariableTypeData is an abstraction of a data value of an arbitrary type. For example, one VariableTypeData object might contain a string value while another holds an unsigned integer. The motivation for this type was to allow a command or other message to maintain a constant interface no matter the types of parameters or information involved. The only changing aspect in the parameter list is the number of parameters, which is much easier to handle in the message classes themselves.

The implementation for the VariableTypeData class was modeled after the External Data Representation (XDR) standard described in [3]. XDR defines a method for remote computer systems to communicate strongly-typed data. This concept was achieved in Phoenix by creating a Datatype class for each desired type. As can be seen in Figure 2, each type inherits from the standard Datatype interface, which allows the VariableTypeData object to treat each type the same. This lets a programmer add a new type to VariableTypeData with a relatively small number of changes and separates the functionality of each type into an encapsulating class.

![Figure 2: Current Datatype inheritance tree](image)

The other small helper class needed to implement the message protocol is Setting. A Setting object contains an identifier that indicates which configuration item it describes and the value for that item. Because a configuration item may take a variety of different types depending on the actual setting it represents, it is logical to represent the Setting’s value with a VariableTypeData object. In this way, all items in a configuration tree can share the same overall type, which again simplifies the interface.
With these two classes in place, the message protocol can be implemented. As mentioned in the overview section, each packet contains a source server, a destination server, a packet number, and a message. The Packet class defined in the architecture encapsulates this information. Packet objects will allow programmers a compact method for storing all of the information needed to communicate with another server.

In order to allow each message type to be treated the same by both the programmer and by the Packet class, a class hierarchy needs to be defined. Initially, an abstract Message class was created to serve as the interface that all messages must meet. From this abstract class flowed each of the four types of messages. However, it was quickly discovered that the command and configuration messages shared a large amount of code, as did the data and error messages. In the former case, both messages contain a list of items, while in the latter case, each message contains only a single data item. As a result, a significant amount of the code in either of the classes was duplicated in their related message.

To resolve the duplication of code, and to allow for the encapsulation of a response message (in the ReturnMessage class), the message class hierarchy was extended to include more abstract classes. First, the SendMessage class was added to contain the code common to all four types of messages. This information was separated from the Message interface because the ReturnMessage class does not directly need to contain this information, as described below.

Inheriting from SendMessage are two more abstract classes: SingleDataMessage and MultiDataMessage. SingleDataMessage serves as a superclass for DataMessage and ErrorMessage. This class encapsulates a message with a single VariableTypeData field and contains the code that is common to these two messages. The other class, MultiDataMessage, encapsulates a message with a list of data values. Because CommandMessage and ConfigMessage contain lists of differently-type data (VariableTypeData, CommandMessage, and Setting for ConfigMessage), MultiDataMessage must be implemented using C++ template classes. With this additional functionality, the common code between commands and configurations has been shared in this superclass.

The final element of the message tree is the ReturnMessage class. In order to simplify the way that responses to messages are handled, a separate class has been created to indicate that a message is a response. ReturnMessage has been implemented following the Decorator design pattern [4]. It is treated as any other message would be treated from the view of the Packet class but incorporates the additional functionality of indicating that the message that it decorates is a response and that the original message was either successful or unsuccessful. Other than providing this information, the object acts exactly like the message that it wraps by delegating all of the calls in the Message interface to the wrapped message.

To better visualize the interactions between these classes, a class hierarchy diagram has been included in Figure 3. Notice from the diagram that the unique interfaces of each of the four fundamental message types are greatly simplified, creating cleaner and more easily managed code.

The implementation of the permission arbitrator is much simpler than the message protocol. It consists of a single helper class, Permission, and the Arbitrator class. Permission encapsulates the permission of each server in the system to send a single message to the associated server. The Arbitrator class then keeps track of a Permission object for each Message that has been registered with the Arbitrator. The result is a simple permission-checking object that can be associated with each server.

The final component of the message dispatching system is the Dispatcher object itself. The Dispatcher class is the entity within Phoenix that performs the actual message passing between servers. Because multiple servers need to be able to interact with the same instance of Dispatcher at the same time, it was decided that this class should follow the Singleton design pattern [4]. By implementing this pattern, only one instance of the class can be created, which ensures that all servers send messages through the same Dispatcher object.

Within the Dispatcher object, it is also necessary to facilitate the use of the single class instance by multiple servers at the same time. This use has been supported by using the mutex and message queue features of the operating system. Mutexes are used to ensure that only one server is changing the state of the Dispatcher object at a time, which removes the occurrence of race conditions in several of the class’s methods. The class also contains a single message queue, which is used to pass Packet objects from one server to another. Message queues are guaranteed by FreeRTOS to be thread-safe, which removes the risk of corrupting the state of the object when multiple servers attempt to dispatch a message at the same time.

3.3. Logging System

An important component of the development and testing of any software system is a diagnostic logging system. In many systems, a single diagnostic log is maintained. This system keeps track of all of the actions of the software system and writes them to a storage location so that they can be used later to uncover or debug problems. As such, developers often value logs over many other debugging tools as they perfectly capture the state of their system as it is running.
Figure 3. Message Class Hierarchy
As with any other software project, a satellite software system can greatly benefit from diagnostic logs. However, these systems have the additional constraint that the logs must be easily available both on the ground during implementation and testing and on-orbit. As a result, it is often more useful to have several logging systems. One system may send diagnostic information over a ground support connection while another may write to a file in the file system. Regardless of the differences between any given instances, each logger shares several common traits. First, every logger allows a log entry to be written to a location. Second, a logger writes this information to a given storage location. Finally, a logger must be able to be used by several tasks at the same time.

From these criteria, it is easy to see that a Logger superclass can be created that encapsulates the overall functionality of a logging mechanism. The superclass can define an interface that provides a “log” method, as well as state management functionality. This interface will allow any application to use the logger without having to know the specific implementation of a particular subclass.

3.4. File System

The final common element to every system using the Phoenix Architecture is a file system. The purpose of the file system, as in modern computers, is to provide an organized method of storing information on the satellite. Because all satellite missions will require the storage of at least some data (be it diagnostic, science, or configuration), the file system has been integrated into the Core component of the architecture.

As with all other parts of this architecture, the actual file system used on any particular mission will be changeable. To allow for this changeability, the functionality of the operating system will be abstracted away by a common file system interface. This interface, which will be implemented as an abstract class in the actual system, will encapsulate the functionality provided by any file system. In particular, it will allow a user to open a file, read or write to it, and close the file. By using this interface to interact with the file system instead of the file system functions and objects themselves, the architecture can achieve a uniform interface and thus the ability to change the system based on the needs of the mission.

As an initial implementation, the first version of Phoenix will include a version of the FAT file system due to its availability and portability to a variety of memory systems. To incorporate this into the architecture, a subclass of the FileSystem interface will be created. This class will implement each of the methods of the abstract interface by taking advantage of the code already provided by the file system. The result will be a FAT object that can be used by all future versions of the Phoenix Architecture. If in the future other file systems were required, this process could then be repeated as necessary.

4. Hardware Abstraction Layer

Because the Phoenix Architecture is meant to act as a flight software platform for a variety of missions, it must be able to run on many different hardware configurations. In particular, no part of the architecture should directly use code that depends on a particular microcontroller or other hardware component. Instead, functions contained within the hardware abstraction layer (HAL) must be used.

4.1. Overview

As described above, the purpose of the HAL is to decouple the application software (Phoenix) from the hardware and low-level driver software. To accomplish this, two software components must be created. First, infrastructure must be created to allow the correct driver files to be built along with the rest of the architecture. This portion of the HAL should be transparent to the user so that once the HAL has been implemented for a hardware platform, a programmer does not need to deal with this infrastructure.

The other half of the HAL is a collection of classes that act as software abstractions to each hardware component. All interactions with the hardware will occur through these classes. As a result, the high-level code will not be affected by any changes to the hardware, as long as the first half of the HAL has been implemented correctly.

4.2. Implementation

While the HAL can be thought of as being two separate components, in the implementation the two parts will need to be merged together. For each hardware component, such as a timer, communication peripheral, or external memory, a class definition will be created. Then, each hardware platform will implement the class according to the needs of the hardware. Finally, the build environment for the architecture will choose the correct set of class implementations for the specified hardware platform.

5. Phoenix Servers

While the Phoenix Core offers a large range of functionality for a satellite system, the majority of the code for a given project will be written in Phoenix...
servers. Each server will offer a specific functionality and will communicate with the rest of the system through Dispatcher.

This method of incorporating new functionality into the architecture has several advantages over using direct method calls. First, because a message is sent to a server through Dispatcher, no hard dependency on the existence of that exists in any other server. If the destination of a message does not exist within the system, then Dispatcher will return an error message and the sending server can continue its work. Second, a server that needs to interact with many different objects within the system does not have to keep track of references to each of these objects. Instead, the module is able to send commands to each of the objects through Dispatcher, which simplifies the code in the server. Finally, because Dispatcher performs permission checking on each packet before it is handled, this interfacing system provides better security and error handling within the system. If direct function calls were used instead, a module could not guarantee that the calling entity has permission to call the given function, which makes the system less stable and reliable.

6. Conclusions

The Phoenix Architecture has the potential to greatly simplify student satellite software development. As the above sections describe, each component of the system has been designed with flexibility and modularity in mind, resulting in an easily changeable and yet incredibly capable software system. Furthermore, the use of a standardized interface between modules reduces the hard dependencies between different components in the system. This reduces the overall complexity and contributes to the changeability of the architecture.

Currently, the Phoenix Architecture is undergoing development in conjunction with the ALL-STAR project at CU Boulder. Development has begun on many of the components in the Phoenix Core. At the current pace of the software team, a majority of this software should be completed by June. At that point, development will split to both the HAL and servers specific to the ALL-STAR project. As each of these servers is completed, they will be added to the Phoenix software so that future projects can reuse their functionality as needed. The result will be a modular, diverse software repository capable of meeting the needs of many student satellite projects.

7. Works Cited