The Design of a Highly Configurable, Reusable Operating System for Testbed Satellites

Student Author: Rajesh K. Batra*
Dept. of Aeronautics and Astronautics
Stanford University
Stanford, CA 94305-4035

Faculty Advisor: Robert J. Twiggs†
Space Systems Development Laboratory
Dept. of Aeronautics and Astronautics
Stanford University
Stanford, CA 94305-4035

1 Introduction

Quick testbed satellites, typically inexpensive, built in quantity, and designed to run customer specified experiments offer researchers the opportunity to test, in a relatively inexpensive manner, experiments in space. Likewise, quick testbed satellites offer companies the ability to prototype new products in a relatively short design cycle.

The design of an operating system (O/S) for a quick testbed satellite is different from ordinary operating systems found on today's satellites. In the case of a typical satellite, the mission is specific and therefore the software on board is designed with that mission in mind. For a testbed, the satellite bus has been designed ahead of time in a few standard configurations. A researcher would select a bus that most satisfies his needs. Therefore, it is important to note that the experiments are not known up-front. Rather, interested researchers identify their experiments at a later time. Since there can be multiple researchers, the experiments may come in at various times prior to launch.

Unable to identity the experiment up-front, or whether a particular researcher can construct his experiment on time before launch can produce many problems for the testbed provider. If the testbed satellite is dependent upon one experimenter who did not complete the experiment on time, the satellite is forced to either fly with one less payload or is forced to delay the launch. In either case, the dependency causes financial loss. In addition, not knowing the experiment up-front, or not having the payload hardware available can cause the software engineer, whose job is to interface the payload to the command and control software, to begin programming and debugging late in the design cycle. This rush can lead to software that is not fully tested and as history has shown can lead to the demise of entire satellites.

What we propose is the design of an *extendible, reusable* operating system that is easily configurable to both the customer's specific mission and to a specific satellite. Since this O/S is highly configurable, there is nothing to stop it from running on a personal computer and acting as a satellite simulator. A researcher, armed with this simulator, could develop and integrate his payload completely independently of other customers and the satellite manufacturer. The testbed provider is now no longer dependent on a particular researcher. Those customers, who have completed their payload before launch, can quickly integrate their payload and *tested* software into the satellite. By severing the dependency of the payload provider from the testbed provider, we have made the design cycle more efficient, and hence cost-effective.

For proof of concept, we have designed a configurable O/S that has been implemented on both a SQuiRT (Satellite Quick Research Testbed) class satellite known as SAPPHIRE [1, 2] and on a personal computer. The operating system, known as Chatterbox, has been designed independent of the customers' specific missions. A common interface and basic facilities (to be described below) are available to all customers on all SQuiRT satellites. Therefore, it is not important for a customer's payload to be on a particular satellite [1]. In fact, in future missions, we encourage multiple customers to develop their experiments in tandem. Those that complete their payload in time would make the coming launch, otherwise, customers

*Graduate Student, Aeronautics and Astronautics Department. Email: raj@kaos.stanford.edu

†Professor, Aeronautics and Astronautics Department. Director of Space Systems Development Laboratory. Email: btwiggs@leland.stanford.edu

[1] We assume for simplicity that each satellite will be launched in the customer's target orbit.

will have the opportunity to launch on the next available SQuiRT satellite.

The intent of this paper is to discuss some of the common elements needed for constructing a reusable, configurable operating system (Section 2), the user interface for such an O/S (Section 3) and finally how to architect such a system (Section 4). Along the way, the benefits and effectiveness of the system will be discussed in greater detail; and the process required to configure the O/S for a specific mission will be described.

2 Facilities provided by a testbed O/S

Every customer be it researcher or corporation that runs an experiment or tests a prototype on a satellite is concerned with the following issues:

1. health of the satellite;
2. feedback from their payload;
3. notification of any payload errors/malfunctions.

Other facilities that may be of interest to the customer:

1. Active control over payload;
2. Data storage acquired from payload and transmission of data to ground;
3. Security and privacy of payload and acquired data;
4. Time base activation of payload.

These concerns listed above remain the same from satellite to satellite. So rather than rewrite code which can potentially introduce bugs and requires additional testing, have the operating system be responsible for providing these facilities. If written correctly, the operating system can be reused on other satellites. The customer's responsibility is merely to exploit the facilities provided by the O/S. The benefits of this open architecture design is tremendous. Payloads can design both hardware and software independent of the satellite bus.

Satellite simulators could be distributed to all customers. This would allow the individual customers to test their payload and corresponding software prior to integration with the real satellite. By testing ahead of time, this could produce huge cost savings. The company would not have to send personnel on-site to a bus provider for large periods of time, nor would the company be restricted to the time constraints of a particular satellite launch. Testing and debugging would be restricted solely to the experimenter.

The facilities in the above lists, including a portable satellite simulator have been developed for the SAPPHIRE satellite and are discussed in detail below.

2.1 Health monitoring

The health of a satellite and its payloads is critical to a mission's success. Each component of the satellite has operational limits. If a limit is exceeded, various contingencies can take place. At a minimum, an error or warning is logged, leaving the responsibility of the component's welfare to the ground-station crew. At a maximum, the satellite may reset or completely shutdown the faulty component.

The engineers that design a particular payload or component know it best; they are cognizant of the operational limits and actions required to minimize damage. Therefore, allowing the customers the capability to program the parameter limits and to test the health of their payload would be best.

SAPPHIRE has implemented a simple to use health table system reminiscent of a spreadsheet. Each row of a table is composed of four entries. The first entry is the test variable; the second and third entries are the allowable upper and lower limits on the test variable; and the last entry specifies where to branch when the test variable falls above or below the upper or lower limits. For example, suppose a particular payload is powered from two sources, a primary and secondary source. At least one source must provide a minimum of 10 volts for the payload to operate normally. Sensors are connected to both power sources. A possible implementation would be to use two tables. The first table's test variable reads the voltage from the primary sensor. The lower limit is set to 10; and the branch is set to launch the second table. Table 2 has entries similar to the first table. Table two's test variable reads secondary sensor, and the branch is to an error logging and correcting function (see Section 2.3). So if the voltage falls below 10 volts on sensor 2 an error function is called. Otherwise control is returned back to table 1.

The use of tables provides a flexible, and configurable mechanism to manage the health of a satellite. A major advantage to using tables is that they can be reprogrammed in space. This may be necessary as batteries begin to deteriorate. Voltage limits may need to be relaxed. Another benefit is that table entries can be prioritized. For example, in the case of

power management, non-critical payloads can be listed first in the table. These payloads would then be powered off before any others, potentially providing the satellite the required power to sustain itself.

The health monitoring system is tied to the beacon system [3]. This provides a way for the satellite or payload to call for help to anyone with a basic receiver. There is no need to log onto the satellite to check for faults. So for example, a corrective action to a payload may be to shut it down and to broadcast a "payload failure" message.

2.2 Feedback and control facilities

An O/S should provide an interface to the satellite bus's input and output ports. Since the O/S is portable, the communication layer to these ports should be abstract enough to allow for future growth in bandwidth and number of ports. On the SAPPHIRE satellite, we have provided 32 digital input/output lines primarily used for powering on and off payloads. In addition we have provided 32 analog to digital channels for measurements and four serial ports with configurable baud rates. Future satellites may provide more serial ports, higher bandwidth parallel ports, or more control lines. The function call syntax to the O/S would remain unchanged. As mentioned in the health monitoring section, we have tied the A2D channels to the O/S's health monitoring facilities. Based on limits of these channels, corrective actions maybe taken.

Once the O/S provides a conduit to the hardware, a payload provider merely needs to write interfacing code to their particular payload. In the personal computer world, this is known as a device driver. We have designed SAPPHIRE's operating system, Chatterbox, to be easily extendible. The customer can quickly attach his device driver and commands to operate his payload. He can fully test the feasibility of his design on the satellite simulator. Once he is assured of the operations, the code and payload can quickly be integrated into the flight satellite. This provides a huge cost savings to both the payload and satellite bus providers. An in-depth discussion of Chatterbox's interfacing language is explained below.

2.3 Error logging and correcting facilities

An operating system designed for testbed payloads must provide an error reporting mechanism accessible to all payloads. By having the O/S manage the errors, we guarantee consistent error reporting, and provide corrective actions at a level that may not be available to the payload provider. For example, if a payload reports a general error message stating that it needs more power, the O/S maybe able to turn off non- critical payloads to solve the problem. Having the O/S manage and make decisions over the satellite concerns, payloads are able to submit complaints and have them resolved in a safe manner.

2.4 File system

Chances are high that a researcher using a testbed satellite would like to store and retrieve measured data. Therefore, an O/S should provide a mechanism for creating files, similar to O/S's found on ground based personal computers. The files could be stored in RAM or space designed disk-drives. However the means for storage, the O/S should provide the file system abstraction. The benefits to a file structure are numerous: All data on the satellite, regardless of the payload would be handled the same way. For example, the retrieval of a file to Earth would be the same for a payload that has acquired a photograph and for a payload that has measured data from a horizon detector. A file system provides information such as creation time, data size, and owner. A file system can also provide security over data. This could allow for multiple experiments by different parties to be on the same satellite and would ensure privacy of the results for the respective customers. At a lower level, a file system is beneficial for memory management. By having this layer of indirection, we have the benefit of localizing all memory allocations, which then can be replaced with more robust memory algorithms for space environments, such as software error detection and correction (EDAC).

2.5 Simultaneous multi-user support

There are several management methods for handling multiple customers on a testbed satellite. The most restrictive would be to have the satellite provider be in charge of all operations on the satellite. This would require the payload provider to either send personnel to the satellite operation facilities or to train the satellite provider's team on the operations of the payload. This is neither the most efficient time or cost solution.

A solution that we have adopted is to design an easy to use operating system which allows each customer to construct their own commands to operate the satellite. This would allow the customer to operate their own payload and to log on directly to the satellite. Since the commands are designed by the

customer and can be fully tested using a satellite simulator, the customer benefits by being able to simulate operating procedures months before the launch. Also, the payload provider now has the option of not needing to send support personnel to the satellite manufacturer, provided that the payload company has a ground station. Allowing customers the freedom to manage their own payload adds the possibility that multiple customers may want to access the satellite simultaneously. Therefore, an operating system which supports multiple users is preferred. We have designed the SAPPHIRE satellite to allow up to 26 users simultaneous access. The Chatterbox operating system allows as many users as bandwidth and memory allow.

2.6 Scheduler

Researchers are often interested in acquiring data when the satellite passes over a particular region of the Earth. Chances are high that they will not be logged in at that time. Either the researcher needs to network with more ground stations or have the operating system execute user specified commands at certain times. The Chatterbox operating system has been designed with a scheduler system. Commands, specified by the researcher, are executed with the researchers access rights and privileges at a future time. Commands can be added, listed, or removed from the scheduler.

Another solution that we are studying to simplify the interaction with testbed satellites is to have a customer use the world-wide web to request operations to be executed on the satellite [4]. Remote access would remove the necessity of sending personnel to particular site locations. In addition, by having enough ground stations networked, almost continual coverage of the satellite can be attained. This would, additionally, free the customer of operation costs for a groundstation. Chatterbox's interface has been designed with the notion of eventually using automated scripts. See next section below.

3 User interface

The interaction of a user with a satellite is by far one of the most important issues to consider. A confusing interface can lead to severe errors and at best require unnecessary training and a guaranteed obsolete operating system. No matter how intricate the design of the operating system, if the interface is clumsy, only under duress will a user be willing to learn the system. Is it no wonder that systems have moved from command line to graphical user interfaces (GUI). And even in the windowing environment, we see continual improvements. It has been the author's experience that satellite interfaces are reminiscent to interfaces similar to the Altair 8800 (merely switches), and at best, systems such as DOS, UNIX or VMS which has a hodgepodge of commands lacking a consistent thread[2]. Of course, installing a GUI interface on a satellite at this point in time would be overkill and a waste of bandwidth. Only when the number of users increase to a high enough level, that it becomes more efficient for an O/S to be obvious than to expend the resources in training would it merit going to a graphical interface. And, the first logical GUI interface would be implemented on the ground and merely translate the mouse clicks to actual satellite commands. However, even at this level it pays to have a well structured O/S command set.

An operating system interface for a testbed satellite varies from a conventional O/S in that the login time is limited to very short periods. For SAPPHIRE, a typical login window is about 15 minutes. Therefore, an efficient command set is necessary. Actions need to be correctly executed using the minimum description from the user.

Regardless, an O/S should be easy to learn and to a certain extent intuitive. This design criteria will have huge payoffs when an unexpected contingency occurs. An obvious command set may actually save the satellite or payload from a customer who had not implemented complete emergency procedure. If a command is difficult to recall or its semantics inconsistent, chances are reduced of successfully applying an action. The vocabulary should be reused consistently. Language defines and bounds our thoughts and ideas. If the semantics of an O/S interface are limited or inconsistent, the ideas and actions will be too. Ideally, basic behaviors or words should be combinable to form sophisticated behaviors or thoughts. Additional benefits of a reusing vocabulary is that there are less commands for a customer to learn, and typically makes the software implementation more compact.

3.1 Chatterbox user interface

For the SAPPHIRE satellite, a sophisticated language requiring a grammatical parser was beyond the

[2]The author is currently a system administrator (on the side) for a cluster of Sun and Silicon Graphics workstations at Stanford University, and is proficient with several (out)dated O/S's including VMS, and AmigaDOS. In no way is he attempting to malign the progress that been made in the history of operating systems He merely acknowledges that as with most things, improvements are always forthcoming.

Component	Description
voice	Addresses voice synthesizer payload
camera	Addresses the camera payload
sensor	Addresses telemetry & IR payload
os	Addresses system commands
help	Provides help on any component

Table 1: Addressable components on SAPPHIRE.

Action	Description
acquire	Activate payload, spools data to file.
list	lists all files associated with Component
get	sends file to ground
delete	removes file
view	Activate payload & spool data to screen
speak	Enunciates given string to ground

Table 2: Actions for voice, camera and sensor components.

scope of its mission timeline. However, a consistent vocabulary and syntax was defined. The first parameter(s) describe which component an action will be applied to, i.e. a noun. The next parameter is the explicit action (verb) followed by any modifiers. So command sentence would be constructed as follows:

(Component) (Action) [optional modifiers]

For SAPPHIRE, the payloads were known up front. They are a voice synthesizer for enunciating user sentences to ground, a camera for taking pictures, and infrared sensors. There are also housekeeping commands such as viewing logged errors, viewing the current time, etc. that are compartmentalized under system commands. Note, for ease of use, the help component which is clearly a part of the system has been moved up to the top level for immediate access. Therefore there are five main components for SAPPHIRE: voice, camera, sensor, os, and help. A description of each is listed in Table 1.

The number of actions for the payloads were surprising small. They are listed in Table 2. Note that if a customer is providing a new payload, his job would be to create a component name, and any new action words required to operate the payload. For the voice synthesizer, the action word speak became necessary and was added solely to the voice component vocabulary.

A few examples will now be given to clarify Chatterbox's usage: To take a picture, or to acquire data for channels 1 through 10, the commands are very similar:

camera acquire
sensor acquire 1-10

Notice for the sensor command, a modifier indicating which channels to acquire the data was specified. Additional modifiers such as sample rate and duration could have been specified, but instead the default values were accepted. To list what files the payloads have created, the following similar commands are used:

camera list

Entry #	Size (Bytes)	Owner	Date
1	22,000	kb8tfa	12:15 6/6/97
2	12,000	kb8tfa	12:20 6/6/97

sensor list

Entry #	Size (Bytes)	Owner	Date
1	10000	ke6qmd	13:15 6/5/97

To retrieve the files to ground, one merely addresses the component with the get command and specifies which entry they wish to get.

camera get 1,2
sensor get 1

In a similar vein, a basic vocabulary which is reused often has been applied to the entire satellite. Since this paper is not intended to be a users guide for operating the Chatterbox O/S, a complete description of all system commands will not be discussed. Parties interested in a detailed implementation of Chatterbox may contact the author directly.

4 Architecture for implementation of a testbed O/S: Modular design

We have given the customer the freedom to design their payload, write the required software drivers (see section 2.2), and to test the software/payload interaction independently of the satellite. As stated previously, a satellite simulator capable of running on most personal computers is provided to the customer. Once everything is operational, the payload and software can be installed on the next available satellite. This section discusses the elements required to successfully implement an O/S capable of being ported to various platforms, and a framework for allowing user customization.

At the highest level, the O/S should be broken up into fundamental units known as a modules. Similar to an atom which is composed of neutrons, protons, and electrons, a module can be decomposed into an interface, and implementation section. The implementation section is a body of code that performs the behavior (or operation) of the module; it is encapsulated

(or hidden). The interface section is the front end of a module; it is a liaison between the user and the implementation section. The interface should never change. It is in the coarsest sense the language needed by the user to have the module perform an action. Changing the language affects the user directly. However, the implementation can and often times will change. Typically code is rewritten to optimize certain operations. Since the implementation is hidden behind the interface, changes will not adversely affect the user. Examples of some of SAPPHIRE's optimized implementations are given below.

There are many benefits to creating modules, we will list only a few of these benefits. Modules can be portable and most certainly reconfigurable. To realize the benefits of *portable and reconfigurable* requires a bit of background explanation. A properly designed module can be categorized as Machine Independent (MI), Device Independent (DI), or Machine and Device Independent (MI/DI). A machine is defined to be the hardware on the satellite on which the O/S is run. It is typically called the CPU (central processing unit), and varies from satellite to satellite. A device is a peripheral added to the satellite. For example, a customer payload is a device. The corresponding software required to operate (or drive) the device is a device driver. So, a Machine Independent module is code that is completely independent of the actual CPU that it is running on [3]. For example, a module that controls a customer's payload is MI. Even though a customer may use commands from another module to communicate with the RS232 port; he is accessing the interface of another module (in this case, a machine dependent module). Regardless of what other modules the customer may access, so long as he never calls machine specific code directly, his module will remain Machine Independent. Of course, since his code is designed to communicate with his payload, it is not Device Independent. An example of a MI/DI code, is an algorithm that sorts numbers (such as a quick sort). Now, with that explanation in hand, machine portable code is code that is Machine Independent (MI). Hence, device drivers are portable modules. A module can be reconfigured by merely changing the implementation section. So, suppose we have a RS232 communication module designed to run on the Motorola processor. Now, suppose we want to run this module on a simulator that uses an Intel processor. The interface code the customer uses to receive and transmit data remains unchanged; all that is required is to change the implementation section to Intel specific code. By changing just this one module, all MI code above will automatically work on the simulator. We have saved a tremendous amount of work which directly translates to cost savings. Another benefit to modules is that they can be shut down or replaced easily. If a payload is inoperable or will be replaced by another payload the corresponding software module (and the calls to that module) need be replaced. Note, that there is some effort involved; however, modules try to minimize that effort. In order to communicate with the Chatterbox O/S, the first parameter required was the component (see Section 3.1). From an implementation point of view, a component is nothing more than a module. If the Camera payload fails in space, all commands addressing the camera module can be shutdown with no affect on the other components.

Modules can have access levels. Since payloads are owned by various customers, a certain amount of privacy may be required for their data. Just as a module can be shutdown, a module can limit the access to certain parties. On the SAPPHIRE satellite, the IR sensors are a Jet Propulsion Laboratory (JPL) experiment and only authorized personnel have access to the sensor controls.

Modules can be optimized and designed in tandem. Since the implementation is encapsulated, a customer can make sweeping changes to increase speed or minimize memory usage without affecting the entire system. On the SAPPHIRE satellite, the IR sensors were originally required to acquire data at 4 Hz. In time, the specification changed to 100 Hz. This required substantial modifications to the implementation; however, once implemented, it was transparent to the ground station personnel. Customers can design various modules in tandem. As mentioned before, multiple payloads can be built and added to the framework.

5 Additional thoughts

The concept of modular design has been preached throughout academia and has been implemented with varying degrees of success in particular industries. However, a certain amount of sloppiness is inherent to a project of any substantial level. Satellite fabrication, as most would agree, is a project of large magnitude requiring the cooperation of many people, specialized in many fields, across various industries. In order to reduce some of the inherent sloppiness in a system, a certain amount of rigor must be asserted

[3] The author assumes the language the code is written in is Machine Independent (i.e. ADA, C, C++ not Assembly/machine language)

in the language and protocol. Object-oriented programming languages have been developed to enforce this modularity notion and to aid in removing some of the inherent sloppiness found in procedural languages. However, the author would like to entertain the idea that not only should software modules exhibit behavior, but in fact so should hardware. If hardware is designed with an interface, the hardware can be requested to go off on a task and notify the requester of its completion. Adding this level of autonomy has just modularized the entire design process, and has given the capability of replacing hardware devices with more optimized devices without affecting the entire system. The author concedes that establishing the protocols, and design specifications for customers to follow is a substantial task. But in the long run, a standardization will produce far more benefits and advancements to satellite technology.

Acknowledgments

I would like to thank Kenneth P. Koller for his extraordinary effort in helping me to focus the design, sharpen the vocabulary, and implement the Chatterbox O/S. The actual name for our O/S came about one Sunday morning as Ken and I drove up to campus listening to Garrison Keillor's Prairie Home Companion Theater. When we found out that Pete's Pretty Good Groceries was right across from the Chatterbox Café, we knew we were on to something.

About the author

Rajesh K. Batra: Graduate Research Assistant at the Aeronautics and Astronautics Department at Stanford University. He received his BSc in 1993 in Aerospace Engineering at the University of Cincinnati, and his MSc in 1994 in Aeronautics and Astronautics from Stanford University. Rajesh K. Batra has extensive experience in object oriented design and simulation. He has consulted in industry in the area of software design and is the primary operating system designer for the SAPPHIRE satellite.

References

[1] C. A. Kitts and R. J. Twiggs, "The satellite quick research testbed (squirt) program," in *Proceedings of the 8th Annual AIAA/USU Conference on Small Satellites*, Aug. 1994.

[2] C. A. Kitts and W. H. Kim, "The design and construction of the stanford audio phonic photographic infrared experiment (sapphire) satellite," in *Proceedings of the 8th Annual AIAA/USU Conference on Small Satellites*, Aug. 1994.

[3] M. A. Swartwout and C. A. Kitts, "A beacon monitoring system for automated fault management operations," in *Proceedings of the 10th Annual AIAA/USU Small Satellite Conference*, Sept. 1996.

[4] C. A. Kitts and C. Tillier, "A world wide web interface for automated spacecraft operation," in *Proceedings of 32nd Annual International Telemetering Conference*, Oct. 1996.