Design of Flight Software for the KySat CubeSat Bus

Samuel F. Hishmeh, Tyler J. Doering, James E. Lumpp, Jr.
Department of Electrical and Computer Engineering
University of Kentucky
Lexington, KY 40506
859-257-8042
jel@uky.edu

Abstract—This paper[1] [2] describes the design, implementation and testing of flight software for KySat-1 a picosatellite scheduled to launch in 2009. The paper also discusses the challenges of developing dependable software in an academic environment and the development of dependable software for commercial off the shelf (COTS) hardware in space applications. Techniques employed to design for reuse and examples of software reuse in recent sub-orbital and near-space missions are also described. The software architecture, software engineering practices, and testing techniques developed for KySat-1 will serve as the basis for a series of future Kentucky Space Consortium missions.

TABLE OF CONTENTS

1. INTRODUCTION

Background

CubeSats—In order to expedite small satellite design and reduce costs, students at Stanford University and California Polytechnic State University developed a picosatellite standard known as the CubeSat [1]. CubeSats are pico-class satellites, with the smallest having the dimensions of a ten centimeter cube (1U), and a maximum weight of one kilogram. Satellites with the dimensions of approximately 10x10x20 centimeters and 10x10x30 centimeters are called 2U and 3U CubeSats respectively. CubeSats are deployed using a standardized launch vehicle interface (LVI). There are several LVIs currently used to deploy CubeSats, including: the original CubeSat deployer the P-POD (Poly Picosatellite Orbital Deployer) was developed by California Polytechnic State University, the T-POD (Tokyo/Toronto Picosatellite Orbital Deployer) from the University of Tokyo and University of Toronto, the X-POD (eXperimental Push Out Deployer) from the University of Toronto, and the SPL (Single Picosatellite Launcher) developed by the German company Astrofein.

Despite their size, CubeSats provide significant utility and NASA, NSF and DoD have established CubeSat programs. CubeSats typically have full duplex radio systems, electrical power systems, and attitude control systems. Students designing CubeSats are exposed to many of the same challenges that aerospace engineers in industry are faced with. In addition to these challenges, the compact sizes and relatively small power budget of the satellites offer some unique challenges.

Since the creation of the CubeSat standard in 1999, more than eighty universities and corporations have started CubeSat projects [2]. The most recent launch in June, 2008 included six international, student-built, CubeSats. NASA has built and launched three CubeSats of its own, GeneSat-1, NanoSail-D and PRESat, and have plans to launch PharmaSat-1 in the first quarter of 2009, all of which use the GenSat-1 bus [3]. Several commercial CubeSat projects are underway, such as Boeing's CSTB-1 (CubeSat Test Bed 1) and Aerospace's AeroCube-1 and AeroCube-2 [4]. These organizations are realizing that CubeSats can provide access to space, but at a much lower cost and shorter development time.

Kentucky Space—The Kentucky Space consortium is a collaborative effort of public and private partners throughout the state of Kentucky focused on small satellite development and access to space for small payloads. In 2006, the consortium was formed under the leadership of Kentucky Science and Technology Corporation (KSTC), a nonprofit corporation committed to the advancement of science, technology and innovative economic development in Kentucky. Kentucky Space's goal is to establish a competitive space program in Kentucky by developing the infrastructure and expertise needed to sustain a program that will be beneficial to the state.

KySat-1, Kentucky's First Satellite—Kentucky Space Consortium's first mission was the development of the CubeSat KySat-1. KySat-1 a 1U, one kilogram CubeSat being developed by college students across the state of Kentucky. To follow along with the Kentucky Space mission, the primary mission of the satellite is to interest K-12 students in Science, technology, engineering, and math

1

[1] 978-1-4244-2622-5/09/$25.00 ©2009 IEEE.
[2] IEEEAC paper#1527, Version 4, Updated 2008:12:21

(STEM) fields. Once on orbit, KySat-1 will be made available at no cost to Kentucky schools, students, teachers, and parents for educational purposes and applications. While the primary mission of KySat-1 is educational outreach, the goals of the Kentucky Space program also include creating educational experience for secondary and post secondary students, cultivating an aerospace and satellite technology base in Kentucky, and developing a reliable and reusable satellite bus that will form the basis for future education and commercial satellite missions. The timeline for KySat-1 was aggressive and off-the-shelf technology was leveraged whenever possible. Following are the primary mission goals of the satellite.

(1) Implement a Concept of Operations that will Attract Kentucky Students to Space Technology

(2) Create an Infrastructure to Develop CubeSats

(3) Develop Reliable and Reusable CubeSat Bus Architecture

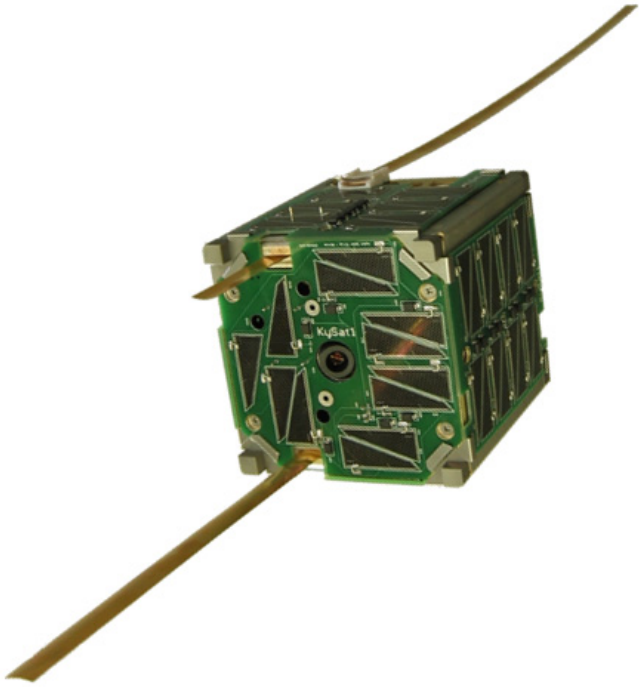

Figure 1 – KySat-1

This paper focuses on mission goal (3).

The IEEE (Institute of Electrical and Electronics Engineers) definition of reliability is as follows. "Reliability is a design engineering discipline which applies scientific knowledge to assure a product will perform its intended function for the required duration within a given environment" [5]. For space systems reliability is mission critical. A latent software error could have detrimental affects to the satellite. Once on-orbit, if an error does occur, the error is usually not correctable. In addition, if an error occurs which prevents communication, this is not an option. Therefore it is essential that a spacecraft be highly reliable.

Developing software for a standardized satellite bus relies on software reuse. The satellite bus should be reused for many missions with minimal integration effort for different payloads.

Software development reuse depends on thorough documentation, especially in an academic environment. New students entering the project, who are unaware of the current progress, may repeat some task that has already been completed. This repeated work will slow down the development of future missions, and increases the likeliness of errors. For these reasons, developers must make status easily available to incoming students.

The remainder of this paper describes the design, implementation and testing of the software for KySat-1. This includes the development of infrastructure, methods used, software design, and challenges faced. All aspects of the design focused on creating reliable and reusable software. The following section discusses the software requirements established by the design team.

2. SOFTWARE REQUIREMENTS

A complete set of requirements for KySat-1 were developed over time. The software team knew that the primary goals of KySat-1 were to gain the interest of students and design a satellite bus. The primary requirements of the software were to establish ground communication, play audio files over the radio when a satellite operator requests, capture images and transfer them to back to earth, and report satellite telemetry. Within these requirements a large set of more specific requirements branched off. To make contact with the satellite, for example, it was decided that the software should issue periodic radio beacons. These beacons would be used by ground stations to alert them that satellite is in range, and that it is ready for communication. To play audio files the software must be able to receive files, and accept commands to playback the uploaded file. To acquire images from the satellite the software should accept commands to capture photographs, and have the ability to transmit them over the radio. For satellite telemetry, there should be commands to request telemetry from the satellite, and the software should respond with the requested information. The requirements continued to branch off until the point the programmers were 100% clear on what was to be done. Following are the derived primary requirements for KySat-1:

(1) **Prelaunch Configuration** – The KySat-1 developers must have the ability to configure non-volatile memory on the satellite, to prepare it for launch.

(2) **Deployment Startup Sequence** – Upon deployment from the LVI, the satellite can not release deployables, such as antennas, for thirty minutes, and high power transmission can not begin until after forty-five minutes. During this window the satellite should perform some unique startup sequence that will occur only after launch deployment.

(3) **Provide Radio Communication** – To send and receive data to and from the satellite, the software must command and control the KySat-1 radios.

(4) **Command and Data Handling** – The software must implement a system to execute and schedule incoming commands.

(5) **Manage Radio Power** – There are two radios on-board the satellite. Due to power constraints only one radio can transmit at a time. To ensure this, and that receivers are powered down when not in use, a system to manage radio power must be put into place.

(6) **Report Telemetry** – To gather system status information, the software should provide a method of gathering telemetry from the satellite.

(7) **Provide File Access** – Many operations on the satellite require storing data. This data is abstracted using a file system on an SD card. The software must provide satellite operators the ability to upload and download data files.

(8) **Playback Audio** – To support outreach to K-12 students, and as well Amateur Radio (HAM) radio operators, the software should implement audio playback.

(9) **Capture Photographs** – To support outreach efforts, the software should have the ability to capture photographs from the on-board camera.

At the beginning of KySat-1 development, there was no infrastructure to develop software. The following section discusses the creation of the necessary infrastructure.

3. SOFTWARE DEVELOPMENT INFRASTRUCTURE

In this section we will discuss the basic components of a software development infrastructure. The following subsection discusses what hardware and software tools to use, followed by the importance of creating development standards, and in the last subsection several development strategies are discussed. Throughout the discussion, the software development infrastructure used for KySat-1 is used as an example.

Tools

Hardware—It is important to use a development board, or "flat sat", to test hardware and software early in the development process. A development board should electrically "similar" to the flight hardware, but be physically different in that it gives developers access to all microcontroller pins, so that test equipment can more easily be attached. With embedded systems programming, it is possible to have a software bug which damages hardware. Development boards should use circuitry, which may not be present on the flight computer, to protect hardware. The KySat-1 software team used development boards from Pumpkin, Inc., that include features which alleviate some of the challenges of software development, and serve to reduce development time [6]. Figure 2 shows photographs of the development board and flight module (FM430).

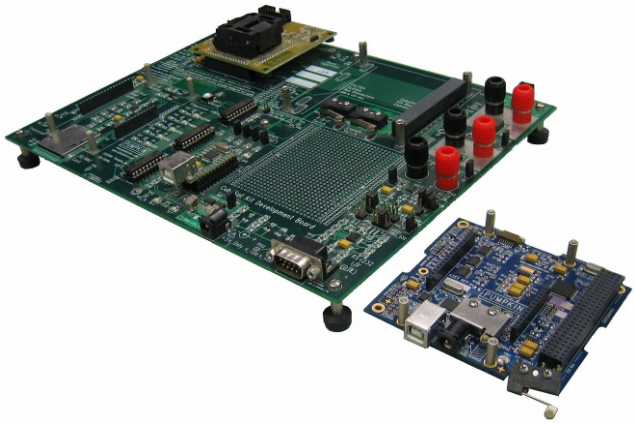

Figure 2 –Development Board and Flight Module

Software—Any large scale software development project will benefit from a Version Control System (VCS). A VCS typically stores all software source files, but can also store documentation, software tools, diagrams, and other useful files. It can be thought of as a backup server for the software, but most importantly it has the ability to maintain multiple versions of software. The version control system used by the KySat-1 team was SubVersioN (SVN) [7]. It is an open source project, available for free download online. SVN is very similar to the older Concurrent Versions System (CVS), being different mostly in its implementation. It supports all VCS features previously mentioned, making it a suitable VCS choice.

The choice of which Integrated Development Environment (IDE) to use is just as important (if not more) as the selection of the processor. A good IDE can save countless hours of development time, and a bad one can cause frustration and schedule delays. The IDE used for KySat-1 was CrossWorks from Rowley Associates Limited [8]. It has a built-in customizable text editor, capability to handle multiple processors in one project, thorough debugging capability with multiple breakpoints, variable watch windows, hardware register viewing, a compiler with

excellent optimization options, assembler, linker, processor core simulator, terminal emulator, and more. The time to learn to use the IDE was short and new students were able to begin coding nearly immediately.

Static analysis tools are an excellent way to find problems in code. Static analysis tools like Lint use strong type checking and value tracking, and catches likely areas of problems such as "==" in an assignment, "=" in an logical expression, loss of precision with assigning a larger variable into a smaller one, uninitialized variables, and much more. It can also assist in code cleanup, because it alerts the programmer when variables are unused and when header files are unused. Static analysis can help to catch errors before they occur, hence adding an extra level of safety to the software. For static analysis, the KySat-1 software development team used Cleanscape C++ lint [9]. C++ Lint provides an easy to use GUI on top of a PC-lint engine. All code was scanned using the analysis tool before it was considered flight-ready.

Software Development Standards

Firmware Project Organization Guide—To simplify the use of a Version Control System (VCS), and to deal with programmer turn-over, every programmer involved with the project should maintain identical directory structures for the source code associated with the project. From a newcomer's perspective, this will create a standard for future missions to build on and make it easier for them to understand the project organization. From a programmers perspective this assists in keeping the code organized and simplifies any discussions of file locations. The guide created by the KySat-1 team specifies how files should be organized on the VCS server, and as well code stored in the software's project directory.

Coding Style Guide—A coding style guide is essential for any large software project with multiple developers. The guide should be aimed at making the code uniform across all files and developers. When a reviewer looks at the files, they should appear as if they were written by one person. The coding style guide should keep the code layout well organized, well commented, and easy to read. If a programmer can read and understand the software, the likeliness that the software is reused is higher. Programmers find it frustrating to read files which have no comments, lines of code commented out, improper indentation, poor variable names, etc. When they come across files like these, they are more likely to rewrite the software than to reuse it. The style guide should include rules for, but is not limited to syntax, file layout, comments, naming conventions, indentation, loop declarations, proper macro usage, type definitions, the maximum number of columns in a file, global variables, floating point usage, typecasting, and parenthesis usage. No module used for KySat-1 was considered completed until it conformed to all of the guidelines established by the team.

Development Strategies

Software Releases—As software development began, the KySat-1 team saw the development time for software as a risk. Because CubeSats fly as secondary payloads, the launch delivery could come before the software was complete. To assure that there was always a version of software prepared for release; the software was developed in multiple versions, known as revisions. Each revision was built in such a way that it could be used as a final software product, if the project schedule demanded it. With each new revision more features were added which fulfilled the system requirements. Every revision was written with future revisions in mind, so that adding more features did not cause major changes, if any, to existing code. The team developed four flight-ready revisions of the KySat-1 software, including the final, full featured version, prior to launch.

Module Development—For this paper the word "module" is used to define software that performs some specific functionality. This functionality could consist of one or more tasks. For example, a UART module might contain functionality to send and receive data to a hardware UART, and a packet receiving module might contain code to receive incoming packets from a digital radio. In terms of files, a module usually consists of a header file containing one or more function prototypes, and a C file containing the function definitions. Programmers using a module should be able to use the functions within it, by looking only at the header file. The file containing function definitions should be a black box to other programmers using of the module.

Modularity determines software quality in terms of evolve ability, changeability, and maintainability [10]. To make the code more flexible and modifiable, every aspect of the design should be highly modular. Every piece of software functionality should be broken into specific, well defined modules. This makes the code easier to modify, and prevents modules from interfering with one another. With this method modularity is increased, and the division of work among programmers is simplified. At the highest level, the team divided the software into four modular layers: at the top are application specific tasks, next are hardware specific hardware libraries, at the bottom are microcontroller specific drivers, and supporting all layers are general purpose software libraries.

For modularity to work well it is very important to have simple, logical interfaces between modules. Good interfaces make the code easier to read and understand. This makes the project more maintainable, and code more reusable. Module developers should abstract away the details of a function call, and focus solely on the action performed. For KySat-1 careful consideration was put into every interface to make it as simple, and intuitive as possible. The team spent many hours deciding on module

functionality, names, interfaces, function calls between modules, and documenting module interfaces.

For KySat-1, before a module is considered complete, it must be reviewed by another programmer. The review process is formal and well documented. Code reviews do not begin until the code author agrees the code is in its final form, and it has been tested completely. At this point the author informs the rest of the team the module is ready for review, and a review is scheduled with another developer. In an ideal situation, every programmer would review every module, but given schedule constraints this may not be feasible. The reviewer must check that the module conforms to system specifications, the code is correct and efficient, it passes all tests, and verify that the coding standards are met. If any problem is found, the author is informed so the problem can be corrected, and after the fix the review is resumed. If any modifications are made which might affect module functionality, then all tests should be repeated. If coding standards are not met or comments are added for clarity, then the reviewer can make modifications with the consent of the author.

The KySat-1 software was designed and written using the infrastructure and techniques just described. Following is a discussion of the software design that resulted from the application of these techniques.

4. SOFTWARE SYSTEM DESIGN

The software design is presented at a high level with emphasis on how data flows through the software. To assist in understanding, following is a brief overview of the KySat-1 hardware.

KySat-1 Hardware

The satellite hardware includes the following: VHF/UHF (Amateur "HAM" band) transceiver, Dual-Tone Multi-Frequency (DTMF) Decoder, Electrical Power System (EPS), S-Band transceiver, Camera, Camera Heater, Camera Temperature Sensor, SD Card, EEPROM, Real Time Clock (RTC), Antenna Deployment Circuitry, serial to USB bridge, and external Watchdog timer (WDT). Figure 3 depicts the KySat-1 hardware, and how the microcontroller interfaces with the bus and payloads.

From Figure 3, it can be seen that there are some microcontroller hardware resources which must be shared by off-chip hardware. The KySat-1 microcontroller, a Texas Instrument's MSP430F1611, contains two Universal Synchronous/Asynchronous Receive/Transmit (USART) peripheral interfaces. One of the USART peripherals can be configured for UART, I^2C, or SPI operation, and the other in UART or SPI. To share these hardware resources in terms of software, operating system (OS) semaphores are used. Any software module using a shared piece of hardware must obtain the hardware's designated semaphore, before the device can be used. Along with sharing microcontroller hardware, high-power operations must run mutually exclusive to one another. This mutual exclusion is also achieved by using an OS semaphore. In total, there are three global semaphores: USART0, USART1, and high-

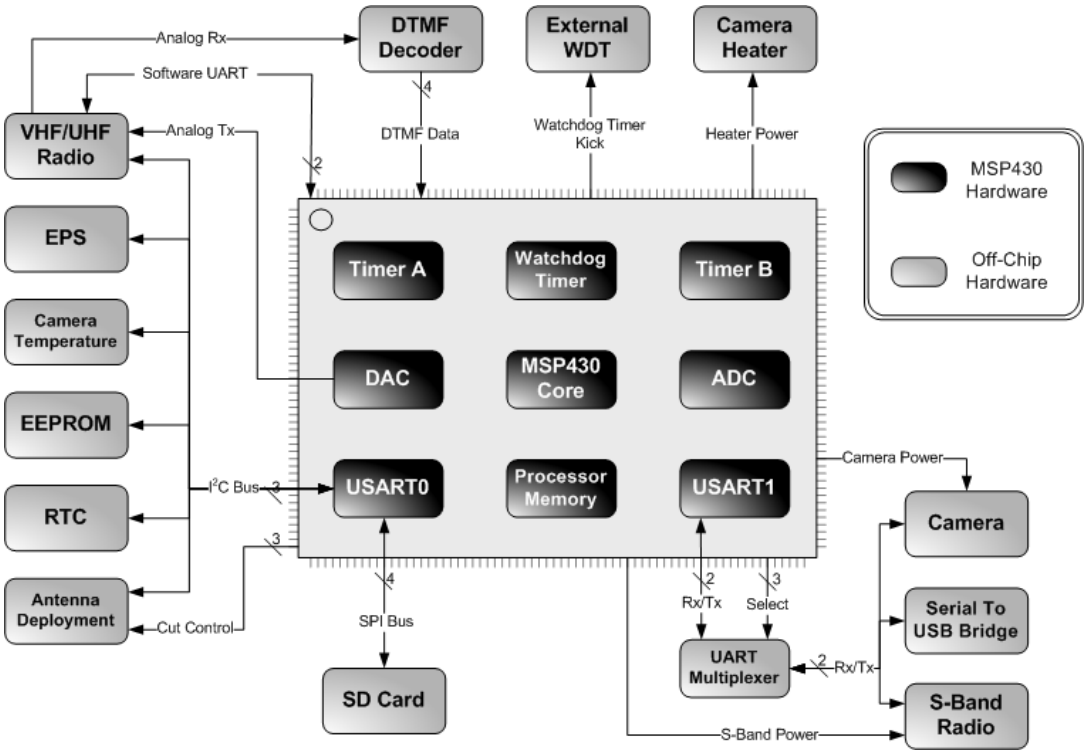

Figure 3 – Microcontroller Hardware Interfaces

power.

Design Introduction

Figure 4 shows a control flow diagram depicting general data flow in the system. The gray rectangles represent hardware external to the microcontroller. This hardware is interfaced with using microcontroller software drivers, represented by the white rectangles. In black, the operating system tasks pull data from the drivers, and then inter-task communication begins.

Data enters the software through one of three channels, seen on the left side of the diagram, they include: the VHF receiver, S-Band receiver, or the DTMF decoder. To handle incoming digital data, a packet receiving task (Packet Rx) pulls packets from the UART1 and software UART drivers. To check for incoming DTMF tones, a DTMF receiving (DTMF Rx) task polls a driver, which reads General Purpose Input/Output (GPIO) pins. A VHF radio power manager (Rx Radio Manager) task is used to stop and start the DTMF and packet receiving tasks, because they should only run when the radio is powered. The receiving tasks alert the radio manager when a packet is received, or a DTMF tone is received, so that the receiver will remain powered during communication.

After data is received either from the DTMF Rx task or the Packet Rx task, it is added to a queue where it will be parsed and executed. The Command Executor task is at the center of all software control. It is responsible for parsing data, and taking an appropriate action. This task can signal one of many tasks, including: Audio Playback, Photograph Capture, Digital Beacon, Telemetry Window, Directory Report, File Save, File Send, Command Scheduler, and more. The executor also sends acknowledge packets back to the satellite operator. After parsing the data, if it is found to be a scheduled command, the executor passes the data onto the command scheduler, where it is held until the requested time. At this time the scheduler adds the data back to the command executor's queue, where it will execute immediately.

There are several tasks which are not part of the primary channels of data flow through the software. Tasks such as the Digital Beacon, Continuous Wave Beacon (CW Beacon), and Event Logging, run continuously, performing their specific task. Tasks such as Report Directory, File Send, File Save, Telemetry Window, Photograph Capture, and Audio Playback, run only when they have been commanded through the executor task. The details of these tasks are discussed further in following sections.

To send data out of the satellite, there are three channels: digital S-Band packets, digital UHF packets, and analog UHF audio. The S-Band Transmit (S-Band Tx), and UHF Transmit (HAM Tx) tasks are responsible for sending digital packets. These two tasks pull from one transmit queue. Depending on the current transmit radio, one of the tasks will start when a packet is added to the queue, and it will send all packets until the queue is empty, or the current

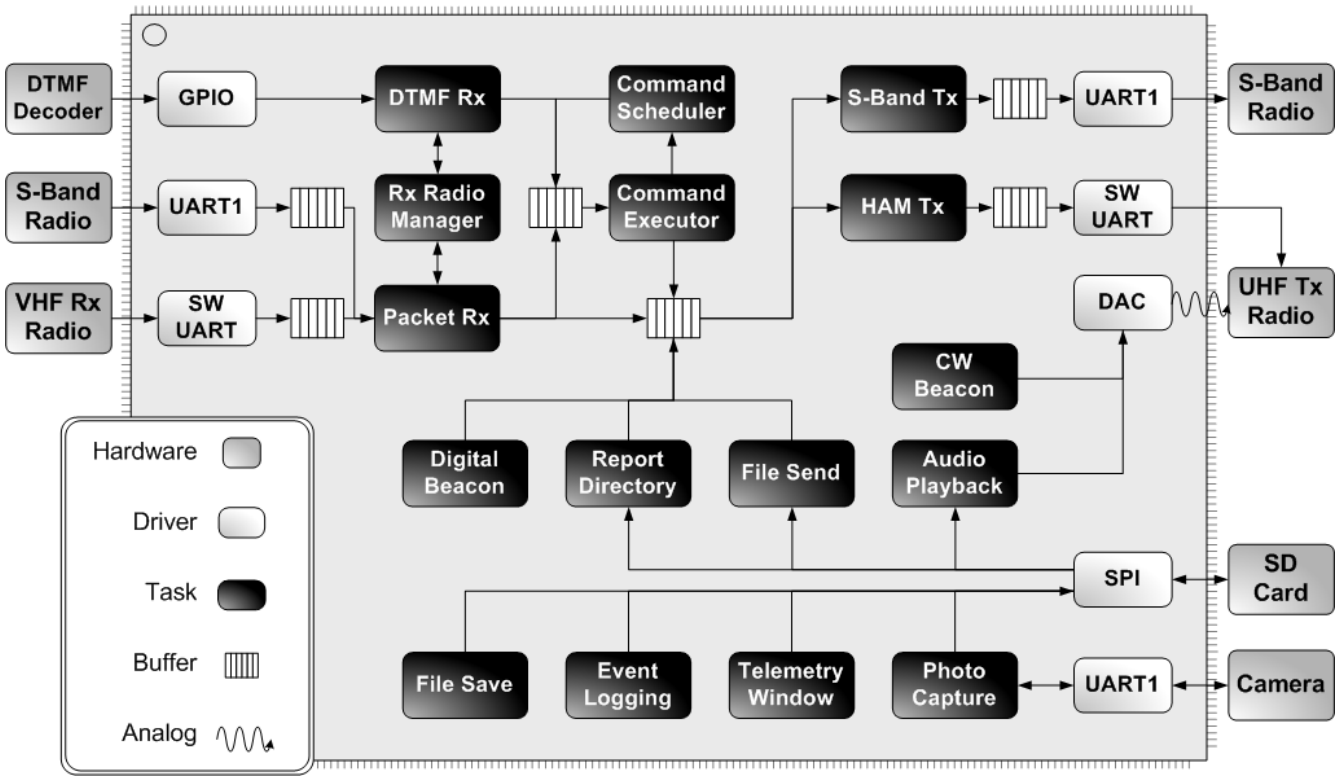

Figure 4 – High Level Software Data Flow

transmit radio is switched. To send UHF analog audio, the CW Beacon or Audio Playback tasks place digital samples onto the Digital to Analog Converter (DAC), which is connected to the UHF radio.

Requirements Fulfillment

Prelaunch Configuration—To configure the satellite for testing, and as well preparation for launch, a configuration mode was required. While in this mode, a KySat-1 developer can configure non-volatile memory devices on the satellite, via the satellite's USB port, known as the umbilical. While in this mode, the developer can read data from, write data to any EEPROM address. The mode is entered by inserting a USB cable into the satellite, when the satellite is fully powered. A developer can then interface with the satellite, by writing custom terminal software that conforms to the accepted inputs and outputs when in this mode. These inputs and outputs are fully documented in the system specifications. Not only is this mode used to configure the spacecraft to its launch state, it can also be used to test the satellite by injecting values into the EEPROM, in turn affecting the way the spacecraft behaves on boot.

This mode is implemented using a loop that polls a UART buffer for incoming bytes. When bytes are received, they are parsed to determine what action has been requested. The software then either executes a command, or listens for more bytes. All commands sent to the spacecraft are acknowledged, so the controlling individual can verify that the command was received properly, and the appropriate action was taken. If no input is received within several minutes, the satellite will timeout and boot the system normally. This timeout was added for additional reliability, in the event this mode is ever entered in flight.

Launch Deployment Startup Sequence—Upon release from the LVI, a typical ICD requires a minimum time before starting normal operation. During this window, KySat-1 will perform a unique startup sequence. When the satellite is powered for the first time, upon being released from the P-POD, it will capture a sequence of pictures for thirty minutes. During this time, the KySat-1 design team hopes to capture pictures of other CubeSats and the launch vehicle.

After the photograph sequence is complete, the satellite will deploy its antennas. Antenna deployment control was one of the most important aspects of the software. Before the deployment sequence can be discussed, it should be noted how the deployment hardware works. The microcontroller controls current regulating circuitry through a series of GPIO pins that apply current to a nichrome burn wire. This wire is setup so that it makes contact with a mono-filament line, which holds the antennas in their launch configuration.

Telemetry Reporting—All telemetry is held in a telemetry database. The database is a software library, and each telemetry point is defined by a few macros. Adding or removing telemetry points is done by modifying these macros. This database provides software modules with the ability to set and increment any telemetry point. All telemetry is stored in one large array. Each telemetry point macro value is used as an index to this array, and the index specifies the location of the telemetry point's data.

Status reports are used to provide the satellite operators with immediate access to any telemetry point. A status report consists of several similar telemetry points, grouped together to fit into one packet. Like the telemetry points, the status reports are created using macros in the telemetry database software library. Many telemetry points are read from the EPS using I^2C, so to keep the database up-to-date at all times, a refresh task runs every second, and it updates these telemetry points. This makes implementation simpler, because a task does not have to be started upon every request for a status report. With this method, it is assumed that the telemetry is up-to-date within a second, and the software can simply read the telemetry values from RAM.

A “telemetry window” can be used if satellite operators desire multiple samples of a single telemetry point. Telemetry windows samples telemetry channels a specified number of times, at a specified rate. Up to six channels can be sampled per window. To further increase the reliability of a telemetry window, the window's status is saved periodically to the EEPROM, and if there is a reset, the window is resumed after reboot. The telemetry data is written to an ASCII readable file. Satellite operators can download a partial file during the window, or a complete one after the window is finished.

Radio Communication—Packets are received using a task which continuously checks the input queues for new data. When a packet delimiter is received, the task uses a packet codec software library to decode the packet, and check it for validity. If the packet is valid, then it is either passed along to the command executor, or added to the transmit queue if it is to be digitally repeated (digipeated). The task has a very high priority, and yields for only a small amount of time, to make sure that incoming data is processed as quickly as possible. If both radios are turned off, then the task is destroyed (removed from the scheduler) until one of the radios is turned on. The task uses two queues, one for VHF data and one for S-Band. By using two queues, both the VHF receiver and S-Band radio can be used simultaneously. The task checks for buffer overflow in the unlikely case a malformed packet is received. This task increments several telemetry points, including packets which are not addressed to KySat-1, received HAM packets, received S-Band packets, digipeated packets, and invalid packets.

KySat-1 also includes DTMF (analog tone) decoding and commanding. DTMF hardware decodes the analog tones and passes the digital information to the software. DTMF receiving is implemented as a task, which continuously polls the DTMF hardware for incoming DTMF data. If a valid tone is received, the task executes a command based on the numeric value of the tone. The task increments the received DTMF tones telemetry point. If an invalid tone is received an error is logged.

Like digital packet receiving, packet sending can occur through either the HAM radio or S-Band radio. To keep interfaces simple, all digital packets are added to one queue, but two separate tasks can pull from this queue. A HAM Tx task controls UHF radio power and sending digital packets. When the task runs, it empties the transmit queue by encoding the raw data, and sending it through the radio as encoded packets. It is here the telemetry point sent UHF packets is incremented.

KySat-1 also has the ability of analog transmission with the UHF radio. Satellite operators can playback audio or Morse code beacons over the radio. To start playback, software modules put eight bit samples onto a Digital to Analog Converter (DAC), to modulate the sample with the carrier frequency. The data must be put onto the DAC at some predefined rate to create a sensible audio signal.

There are two beacons which are used to assist in making contact with, and tracking the satellite. The first beacon is an analog Morse code beacon. Due to the relatively slow data rate Morse code beacons can be received using minimal ground equipment; making them an excellent choice for satellites with HAM band radios. The second beacon, the digital beacon, is formatted to conform to the APRIS-IS telemetry standard, so it is also called the APRS beacon [11]. It is a beacon which sends one digital packet containing telemetry information. The analog beacon can play anywhere from ten seconds to twenty-five seconds, depending on the message length, therefore it consumes much more power than the digital beacon. Even though the analog beacon consumes more power, it is easier to track. As operators of the satellite become more experienced, and better at tracking the satellite, it is desired to have a longer interval between analog beacons and a short digital beacon period, to conserve power.

The digital beacon includes more telemetry information and uses less power than the analog. Like the analog beacon, satellite operators can be request it at any time, and it is disabled when digital communication has begun. The beacon is implemented as a task which waits on a local semaphore with a timeout, if the beacon is enabled, or without a timeout if the beacon is disabled. When an operator requests a beacon, the semaphore is signaled. When this occurs, or the timeout expires, the task reads telemetry from the EPS. After the telemetry is read, it is converted into the APRS format. The HAM receiver is turned on after a periodic digital beacon, giving the operator an opportunity to start digital communication.

Command Processing—To parse incoming data and take an appropriate action, a command executor task is used. This task is started when incoming data is added to its queue, and it runs until its queue is empty. For each incoming packet/command the task checks if the timestamp is to be scheduled, and if so passes the command on to the command scheduler, else the command is executed immediately. After the appropriate function is called, the task checks on the return value to determine if command parameters were valid, and the command was executed appropriately. If the command is executed successfully, then a success acknowledgment is sent back to the operator. If it was not, then a command rejected acknowledgement is sent. The task increments the telemetry points rejected commands, ignored commands, and executed commands.

A command scheduler task is used to give satellite operators the ability to send a command to be executed at a future time. When the command executor adds a command to be scheduled, the scheduler task is started, and the timestamp only is saved to a buffer in RAM. To conserve RAM, the packet data is saved to the EEPROM. The task then takes the time difference of the current time, and time to be scheduled, and delays for this amount of time. When the scheduler awakes from the delay, it adds the scheduled command to the command executor's queue.

During this time operators can query scheduled commands and remove scheduled commands. A scheduled command is identified by its unique identifier, not to be confused with its frame identifier. Every scheduled command is assigned a unique identifier, and to the satellite operators this identifier should be considered random. In software though, the identifier actually corresponds to the index of the command in an array of timestamps. In addition, this unique identifier/index corresponds to the offset of the frame identifier and decoded payload of the command in the EEPROM. To obtain the unique identifier, there is a command to request all scheduled commands, the reply sent returns one packet for each scheduled command. Each packet contains the scheduled command's unique identifier, frame identifier, and timestamp to be scheduled. With the unique identifier, there is a command to query the decoded payload of the command, or remove it from the commands to be executed.

Radio Power Management—The VHF (HAM band) receiver power is controlled via a task. The beacons, and the packet receiving task, can request to turn the receiver on for a specified time. After this request the task is started. It awakes, turns on the receiver, and delays for the time that either the beacons or packet receiving task specified. After this delay, the task waits for the semaphore again to turn the radio off. If another request occurs when the task is delayed, then the task restarts and repeats the algorithm, in

turn leaving the radio on for the most recently requested amount of time. The satellite also has the capability to keep the receiver on continuously. This case is implemented to increase the chances that the satellite is available during a given pass. If this is requested, the task starts as normal, turns the radio on, but instead of delaying, it unschedules itself while the receiver is on, hence leaving the radio on indefinitely.

The UHF transmitter power, one module is used which provides other modules with functions to put the radio into analog mode. These modules must signal to the UHF transmitter controller that they want to use the radio, and then they must wait until the radio is ready. During this time another task starts, which turns the radio on, turns on external modulation, and then the requesting task is signaled. When the requesting task is finished with the radio, they must signal to HAM task that they are complete. After this signal, the HAM task turns the radio off if there are no more tasks needing to use the radio.

Unlike the HAM radio, the S-Band radio does not have separate transmitter and receiver power interfaces. To control S-Band power, a task is used in very much the same way as the HAM receiver task; when the satellite operator requests radio power, a task is started to do so. When an S-Band packet is sent or received, the radio is left on for the specified time. The same task is used to ensure the radio remains powered after a packet is sent or received.

File Access—All telemetry windows, audio files, and photographs are stored as "files" on the satellite. There is a command to request the download of a file. A file request consists of a directory, a filename, a start chunk number, and the number of chunks desired from the file. All chunks is the terminology used to define partial pieces of a file, and a full sized chunk would be thirty-eight bytes. This is the number of remaining bytes in the packet after adding the directory, filename, and number of chunks fields.

Satellite operators also have the ability to upload files to the satellite. The packet format that the satellite receives to save a packet is identical to the ones it sends. Receiving is much simpler than sending, because every packet received contains the data to be saved, and the location to store it. To provide the operator with an easy way of knowing what files are currently on the satellite, there is a command to report a directory's contents. A request consists of directory name, a start timestamp (filename), and a maximum number of packets to return. It is expected that directories may become very large, so the maximum packets parameter is used to prevent long-term packet sending. After a directory report request, the satellite replies with a list of files and their file sizes, with up to five files per packet.

Audio Playback—KySat-1 has the ability to playback audio over its HAM radio. There are two requests for playing audio. The first request is to load an audio file to be played. The file can be played immediately, or can it can be loaded and wait for an operator to request playback. The second request is to start playback of the currently loaded audio file. The purpose of holding an audio file for playback later, is to play the audio file using DTMF tones. The limitation of DTMF commanding is that there is no convenient way to specify a filename to play. The best option is to specify a filename prior to issuing the DTMF command, and simply press a key which will play the file that is currently loaded.

Photograph Capture—Because KySat-1 is passively, magnetically stabilized, and not controlled on the spin axis (parallel to the antennas), considerations were made in the spacecraft software to ensure that bandwidth is not consumed transmitting images with out useful content. Upon capturing each frame, the image can be requested from the compression engine at four different resolutions (640x480, 320x240, 160x128, and 80x64). All images are stored permanently onto the SD card after they are downloaded from the camera. It is possible to retrieve these pictures at any time, in any order, over the lifetime of the spacecraft. Every image captured is stored at the size requested, and also at the smallest size. Employing this method, satellite operators can quickly download the smaller image to manually determine if it is worth committing the transmission time required to get larger versions.

Photograph capture is achieved using a task which controls the capture process. Before attempting to capture a picture, the camera is first heated by duty cycling the header in a timer ISR. While the camera is being heated in the ISR, the task reads the camera temperature, waiting for it to reach the specified set point. If it does not reach the set point within some timeout period, heating is stopped and the photograph is attempted anyway. After the heating process, the task attempts to capture the picture. Some redundancy is put in to place to ensure a photograph is taken. This is done by checking for a successful capture, and if one did not occur, the capture process is attempted up to three more times.

This section described how the software fulfilled the primary requirements of KySat-1. In addition to the requirements, several methods were used to ensure that the KySat-1 software was reliable. Following is a discussion of what features were added to ensure reliability.

5. RELIABILITY

The majority of the subsystems in KySat-1 have little or no flight heritage. To mitigate these risks, several features were added to the software to ensure that the satellite was reliable. Following is a discussion of some of the most notable of these features.

Reconfigurability

Parameters on the operation of KySat-1 can be adjusted based on on-orbit behavior. For example, even though the theoretical values of the average power received from the solar cells were known, these estimates impact beacon periods and other power management schemes. All commands were made as flexible as possible; accepting a wide range of inputs. All beacon periods are can be changed from seconds, up to hours, so that power consumption can be adjusted. The camera also has several configuration parameters that can be changed dynamically. Features such as Digital Repeating, and DTMF Tone Receiving, can be turned off, so handheld radio operators can not abuse the satellite by consuming too much power. The satellite has the ability to set the RTC, in the case that power is lost. In the case that the system is not performing normally, a soft reset can be issued, and with the option of restoring all satellite parameters to their launch values, to try to correct the problem.

The S-Band radio consumes approximately three watts continuously. To prevent power outages, this radio can not be left powered for long periods. The radio is commanded to power on when the HAM receiver is on, and it will turn off automatically if a packet is not received within a specified amount of time. There are also commands to specify how long the HAM receiver should be left on after the S-Band radio is turned on, so if the link can not be closed, then the HAM radio can be used to turn the S-Band radio back off. If the ground operator can communicate with the S-Band radio without problems, then the HAM radio can be turned off quickly to conserve power. The S-Band radio has over fifty configurable settings. The software allows operators to change nearly all of these S-Band radio settings to optimize its operation for reliability and performance.

Event Logging

An error reporting system is included to help identify and isolate problems. An error is defined as some event that should never occur in software. The event could be caused by a hardware failure, single event upset (SEU), or possibly an error in programming logic. Some examples might be an EEPROM write failure, software buffer overflow, or an invalid packet decode. These events are reported by logging each one to a file. When an event occurs, an identifier and the time of the event are written to the log as ASCII readable text. By saving readable text rather than raw data, operators can more easily identify the error. There is a telemetry point which can be used to determine how many error events are currently stored on the satellite, and if errors have occurred, the log can be downloaded. New filenames for the log files are created periodically, so operators do not have to download a large file to view recent error events. Error reporting can also be very helpful during the development stage for debugging purposes.

The idea of error logging to monitor the satellite is extended even further by also logging packet events, incoming or outgoing. These packet events are written to log files as the encoded packet data. By logging packets, operators can track how the satellite is being used. To differentiate between error and packet events, the two distinct events are written to different files in separate directories. The information from the packet logs can be used to improve the software for future missions; by adding and removing features depending on their usage.

Fault Tolerance

The space environment is very harsh. Temperatures swing rapidly from one extreme to the other as the satellite flies in and out of eclipse. Radiation can cause SEUs corrupting data and control flow. To prevent failures, and increase safety, several fault tolerant features were added to the software.

Watchdog Timers—To recover from a system hang-up, two watchdog timers are used on KySat-1. If either one of the timers expires, then the flight processor is reset. The timers mitigate the risk of infinite loops due to invalid states, when the system is unable to clear the timers.

The first WDT is located on the MSP430 microcontroller. The timer is reset in one primary location; an operating system task that continuously runs. The timer is configured to timeout in one second. The task is allowed to restart the timer every 500 milliseconds, half of the timeout. In the worst case, this gives every other task 500 milliseconds to run before it must give itself up to the scheduler. Any task which needs more time than this is responsible for clearing the timer. This guarantees that the operating system is scheduling tasks, no task is hogging the processor, and the interrupt service routine (ISR) which increments the OS timer is running properly. To ensure that 500 milliseconds is adequate time for any task to run, all software was tested with the timer configured to a 750 millisecond timeout; leaving tasks only 250 milliseconds, in the worst case, to run.

The second WDT is external to the primary microcontroller. It is configured, in hardware, with a sixty second timeout. It is different than the internal WDT, in that it has the ability to disconnect microcontroller power to perform a hard reset. It was decided to use a similar method for the external timer; an OS task which clears the timer at half of its period, every thirty seconds in this case. It is different however, in that the task waits on all global operating system semaphores, before it resets the timer. Appropriate usage of semaphores can be complicated. There is the possibility of a dead-lock, in the case that two tasks both wait on the same two semaphores. If each task gets one of the two semaphores, then they are both waiting indefinitely for the other one. A dead-lock like this would likely put the software in a state where contact would never be made with

the satellite again. If, for some reason, a dead-lock does occur, or a task does not give up a semaphore, then the external WDT will not be restarted, and the system will reset. This strategy leaves tasks using the global semaphores thirty seconds, in the worst case, to release their semaphores. While this method does add complexity to the software, it provides added safety to prevent system failure if the semaphores are used improperly.

Redundancy—Radiation can result in SEUs that can alter data. If data is stored redundantly, it is often possible to recover the original data. To mitigate the risk of failure due to SEUs, redundancy is used in multiple places. The most prominent and notable example is used by the EEPROM hardware library. All data stored in the EEPROM, is stored at three locations, spread across its address space. Ideally, if an SEU does occur, or data is corrupted, it will only be corrupted in one location, leaving the other two locations intact. When corrupted data is read from the EEPROM, the software will vote on the data, see that one location is corrupted, and correct the fault. If this type of fault does occur, an error is logged.

Parameter Defaulting—All satellite parameters and settings are read from the onboard EEPROM upon boot-up. If, for some reason, this memory can not be read, the data and variables stored at that location are set to their default/launch value. The default values are documented in the system specifications. Operators also have the option of setting all parameters to their default/launch value using a System Reset command. If the satellite is not behaving properly this can be used to restore the satellite back to a valid state.

Error Scanning and Handling—Error scanning is used to scan the system to monitor its general health, and also correct any critical errors. The error scanner checks that a packet has been received within ten days, and if one has not, the scanner restores the system parameters to their default values. After the system has been restored, the error scanner continues to wait for incoming packets. The scan also scans parameters that are vital for communication, for example the beacon periods, or time to leave the receiver on during digital communication. If a parameter is found to be out of the acceptable range, it is restored to its default value. The scan also checks the drivers and hardware libraries for possible errors that may have been flagged. If an error did occur in any of the modules, then it is logged appropriately.

Failure Modes—A command to restore the satellite is useful if the satellite communications are working properly, but if it is not, an automated method was put into place. If no packets, either via the primary or secondary radio, are received in a ten day period then the satellite restores itself to its launch state. After this, the system will reboot assuming worst case, that the antennas were never deployed, and will try to deploy antennas again. Once antenna deployment is complete, it will reboot and all periods and configuration options will be as they are defined in the launch defaults specification. Hopefully at this point, whatever problem that was preventing the satellite from receiving digital packets will have been corrected.

After restoring the system to its launch state and attempting to deploy antennas, the system returns to normal operation. If no packets are received within a three day period after this restoration, the satellite boots into system recovery mode. When in recovery mode, the system uses a minimal set of hardware and software. If a hardware fault has caused the failure to occur, then hopefully the system can work around it. While in recovery mode the satellite toggles between the S-Band radio and HAM radio. It first listens for digital data on the S-Band radio, then issues a continuous wave beacon, and then listens on the HAM radio. It is in this mode the operator can control the satellite with one of two unique packets. The first tells the software to boot as normal. The second commands the satellite to boot normally, but change the S-Band radio to the default radio instead of the HAM.

While in System Recovery Mode a continuous wave beacon is issued approximately every 4.5 minutes. The beacon message consists of the number of beacons issued while in this mode, including the current beacon, and the number of digital packets (other than the system reset packet) received while in this mode. This is done so that a ground operator can see that packets are being received, even though no acknowledgments are returned. From this, operators can determine that the satellite is in recovery mode, and determine which radio is functioning properly.

6. SOFTWARE VERIFICATION

The testing process should be very formal and well documented. Figure 5 following shows an ideal testing process. From the diagram, it can be seen that first step in the testing process should be an analysis of the system requirements. From this analysis, a test strategy, regarding the best method to test the software, should be formulated. After this, the next step is to begin planning the tests. The test planning should consist of allocating resources and creating a testing schedule. After making these decisions, the next step is to begin creating the test cases. After test cases have been created, the next step is to write some script, or software, to implement the test cases. After writing the test script, the next step is to perform the test. After the test is complete, the tester should release a report, and then have meetings to evaluate it. The evaluation should be used to determine if the test produced correct results. It may be difficult in some cases to determine if results are correct, because some tests may have what seem to have unexpected output, but they actually conform to the requirements. After the evaluation, any errors should be corrected, and the test case re-evaluated. This re-evaluation

is necessary to refine the testing process. After doing a complete cycle with one test case, testers will have more insight into the system and be able make improvements to the testing process. This procedure of testing, correcting errors, and refining the process, should continue until the testing process has been perfected, and all executed test cases conform to the requirements.

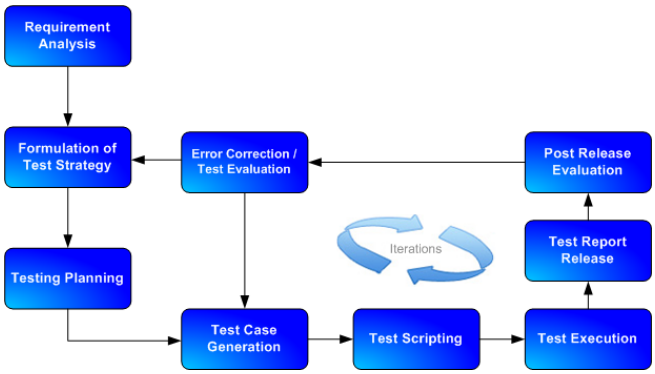

Figure 5 – Software Testing Process

After assessing the software requirements, the KySat-1 developers realized a flight software test bench was needed to completely verify the KySat-1 software. In order to test KySat-1, the team stopped flight software development briefly, and wrote satellite test bench software, known as the Flight Support Software (FSSW). Like the flight software, the FSSW was written in C. This was done, so that many flight software modules could be reused, with very little integration effort. The FSSW requirements were radio communication via a serial port, ability to issue any command to the satellite, ability to schedule commands, and transfer files.

To go along with the FSSW test bench, the KySat-1 team developed a testing document. The document is an exhaustive test bench of all the satellite commands, plus other satellite features. Every command is tested with normal parameters, boundary conditions, and invalid inputs. In addition to this, the document includes test cases for satellite failure modes, proper hardware sharing, and other features, which are not specific to any command. To speed up testing, the document was posted online, so that multiple testers could modify and update it. With the testing document, testers could track which features had been tested and which had not. Testers used the FSSW to test the satellite software, and after tests, they would update the testing document. This testing method of letting multiple people test the software, and tracking the testing progress, was a great way to break up the very large testing process. Testing the exhaustive list of commands and other satellite features was necessary to ensure the satellite performed as expected in flight.

Static analysis using lint was used to further verify the software by scanning for lint compliance. Every file in the code base was scanned before it was allowed to be used for flight. Static analysis revealed several coding mistakes. It also helped to improve code reliability by alerting programmers of unhandled cases.

After KySat-1 software development was completed, and all code was reviewed and tested by the flight software team, final testing began. The task was given to new students starting the project. The goal was for them to use the satellite completely, with only the requirements and specifications documents. With any student ran project, student turn-over is high, so it is very important that the documentation be complete to pass down ideas. Selecting people who are not familiar with the system to test it, ensures that the documentation is complete and thorough. If any gaps or problems were found in the documentation, it was updated appropriately.

During testing, each bug that was found was documented with the date it was found, a description, and the software version number. Every bug found, was promptly corrected, and then the test case which revealed the bug was retested. By tracking the bugs, the software team was able to plot the number of bugs found versus time. In an ideal testing situation, the number of bugs found will increase exponentially in the early stages of testing, but as testing progresses, the system becomes more reliable, and fewer bugs are found. At the end of testing, the bugs found versus time curve should become flat. Figure 6 is a diagram produced from the testing of KySat-1 software, which shows that this behavior was seen during the testing process.

7. DISCUSSION

Many challenges were faced and lessons learned during the development of the software for the KySat-1 bus. Following is a discussion of some of these challenges, and issues future developers should be aware of.

Hardware Challenges

There were several hardware limitations that increased code complexity, and limited the capabilities of the satellite. The most significant limitation was the sharing of processor peripherals, such as USARTs and timers. To overcome these limitations, a method to share these peripherals was added to the software. The method used to share hardware typically made the software more complex, but was unavoidable.

Avoiding sharing peripherals is the desired option, but if not possible, it is important to understand the hardware prior to software development. The KySat-1 developers did not do this, and the initial lack of understanding caused significant delays in software development. At the start of development programmers wrote modules, unaware that the hardware being used would not always be available. After

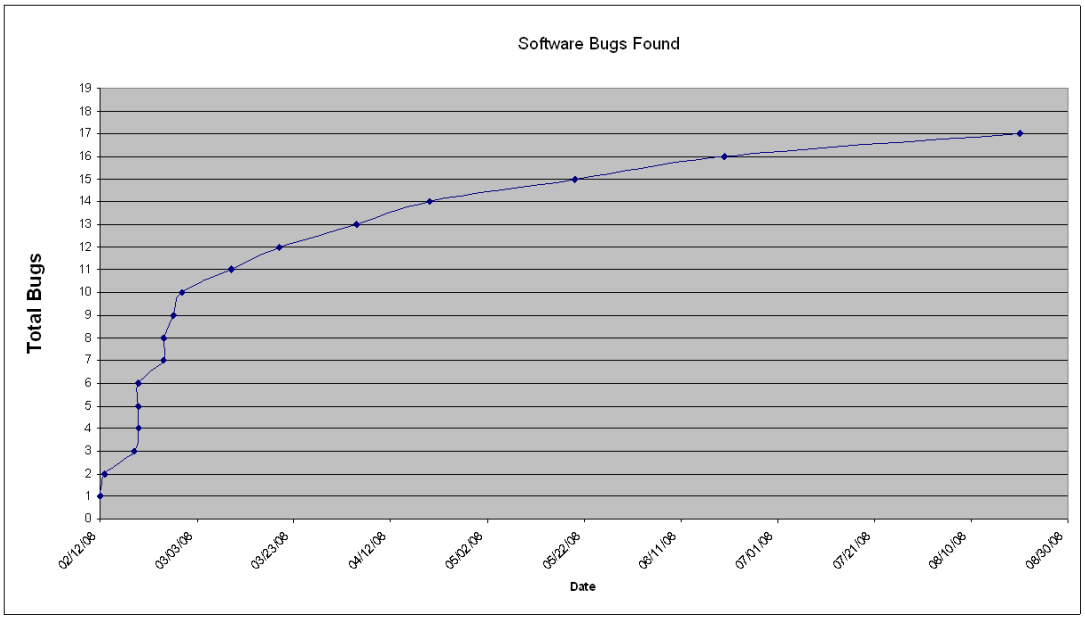

Figure 6 – Software Bugs Versus Time

module completion, and thorough white box testing, the modules had to later be rewritten with the new realization that the hardware was to be shared. This introduced many software errors that would have likely been avoided had the team known what hardware was available, and what hardware was to be shared from the beginning. Modifying any piece of software after it is completed, almost always imposes errors. At the beginning of development, the software team should interface with the hardware team, completely learn the hardware, and how the software interfaces with it.

Student Involvement—One of the biggest problems faced during software development was the lack of student involvement. With only a handful of programmers, few were willing to put in the time needed to produce reliable and reusable software. For some, this was due to lack of commitment, for others lack of time, and others were graduating and moving on to jobs. These factors placed the burden of software development and testing on a few individuals. If the team had known this in advance, more time would have been spent in the beginning of the project recruiting more software developers.

Not only is it recommended to recruit more programmers early during development, but to recruit programmers who are expected to stay for a longer period of time. For the case of an academic environment, this means recruiting younger, less experienced programmers. While it may take more time to train a less experienced individual, time will be saved when they stay involved in the project for a longer period of time.

Scheduling—It was found to be very difficult to establish and meet software development deadlines. This was possibly due to the lack of programmers, but more likely the large learning curve that the team had to hurdle. All hardware and software being used was new to the developers, and therefore even tasks that seemed to be simple at first proved to be very challenging. Developing software for a large system is complex and the design team should be prepared for delays in development.

To better keep track of the development schedule and progress, use of some management software should be considered. Programs such as Microsoft's Project, and Numara's Track-It, are examples of programs that are used for these purposes. They allow developers to setup tasks, sub-tasks, assign task priorities, create schedules, delegate tasks, and more. Using such software would have helped to ensure that tasks were not incomplete, that the team was focusing on the highest priority tasks, that software development was staying on schedule, and that the project was on schedule.

Programmer Accountability—At times programmers did not meet deadlines. In an academic environment, it is very difficult for another student, as a peer and not a boss, to take action when this occurs. Programs such as that just mentioned, Track-It and Project may have helped to organize the work, and make programmers feel more accountable if a job was not completed. Gantt charts could have been used to divide up the tasks, and create a detailed schedule. With a schedule in hand, it could be shown to the programmers, to better help them understand how their role fits into the overall project schedule. This strategy may have helped to increase a programmer's obligation to complete their work in a timely manner.

While some programmers did not meet deadlines, others had trouble conforming to the coding standards, in particular with regard to comments in the code. There is no way to stress enough the importance of thorough documentation. Some programmers choose to write comments while in the process of developing a module, while others prefer to complete the module, and comment afterwards. The latter style may actually result in more useful comments. If a programmer spends several weeks developing a module, he/she may have forgotten implementation decisions after completing the software. When the programmer reviews and comments the code, they will see the code from more of an outside perspective, and be able to write comments where code readers are likely to become confused. Some programmers may prefer the former method mentioned, because after a module is completed, they might not have the motivation to go back and comment it. Either way, every programmer should choose a method that is best for them while conforming to the coding standards.

8. CONCLUSION

This paper has overviewed the KySat-1 software architecture and the KySat-1 approach to reliability and reuse. KySat-1 software development began in the summer of 2006, and was complete in the Fall of 2008. The software team wrote more than 11,000 source lines of code that constitute the flight software for KySat-1. The software was written to be reusable to form a basis for a bus for future missions. Portions of the KySat-1 software were reused for three sub-orbital missions, which took place along with the KySat-1 development. The software that was reused in these missions was done so without any modifications, and in all three missions the software performed without problems. The software developers learned many lessons about meeting deadlines, interfacing with a team of developers, fault tolerance in software, design for reuse, using documentation to pass down ideas, creating a development infrastructures, and much more. KySat-1 is scheduled to launch in 2009. To follow the progress of KySat-1, and other Kentucky Space missions, visit www.kentuckyspace.com.

ACKNOWLEDGMENTS

This work was funded by Kentucky Science and Technology Corporation (KSTC) and the Kentucky Space Grant Consortium (KSGC).

REFERENCES

[1] J. Puig-Suari, C. Turner, and W. Ahlgren. "Development of the Standard CubeSat Deployer and a CubeSat Class PicoSatellite." IEEE Aerospace Conference. Big Sky, Montana. March 2001.

[2] G. Chandler, D. McClure, S. Hishmeh, J. Lumpp Jr., J. Carter, B. Malphrus Jr., D. Erb, W. Hutchison III, G. Strickler, J. Cutler, and R. Twiggs, "Development of an Off-the-Shelf Bus for Small Satellites." IEEEAC paper #1365. IEEE Aerospace Conference. Big Sky, Montana. March 2007.

[3]NASA GeneSat, http://genesat.arc.nasa.gov/, November 27, 2008.

[4] The Boeing Company, http://www.boeing.com/news/releases/2007/q3/070816a_nr.html, November 27, 2008.

[5] Institute of Electrical and Electronics Engineers, Inc., http://www.ieee.org/portal/site/relsoc/menuitem.e3d19081e6eb2578fb2275875bac26c8/index.jsp?&pName=relsoc_level1&path=relsoc/Reliability_Engineering&file=index.xml&xsl=generic.xsl, November 27, 2008.

[6] Pumpkin CubeSat Kit, http://www.CubeSat Kit.com/, November 27, 2008.

[7] Open Source Software Engineering Tools, http://subversion.tigris.org/, November 27, 2008.

[8] Rowley Associates Limited, http://www.rowley.co.uk/msp430/index.htm, November 27, 2008.

[9] Cleanscape Software International, http://www.cleanscape.net/products/cpp/index.html, November 27, 2008.

[10] S. Huynh, and Y. Cai, "An Evolutionary Approach to Software Modularity Analysis." The fifth ICSE Workshop on Software Quality, Minneapolis, MN, May, 2007.

[11] Automatic Packet Reporting System-Internet Service, http://www.aprs-is.net/, November 27, 2008.

BIOGRAPHY

Samuel F. Hishmeh *is a graduate student at the University of Kentucky pursuing a Master of Science degree in Electrical Engineering. He completed his undergraduate career at the University of Kentucky in spring 2006, with Bachelor's degrees in Computer Science and Computer Engineering. His research consists of embedded systems programming, with a focus in fault tolerant software techniques. He has been a member of the Kentucky Space team for two and a half years, and is currently the lead Software Architect for KySat-1 flight software development.*

Tyler J. Doering *is a Graduate Research Assistant in the Space System Lab at the University of Kentucky. He is currently pursing a Master of Science degree in Electrical and Computer Engineering at the University of Kentucky. He completed the requirements for his bachelor's degree in Electrical and Computer Engineering at UK in May of 2007. His current research focus is in reliable embedded systems software and hardware architectures, primarily focused around small satellites. Tyler has been a member of the Kentucky Space team for two and a half years, and is currently the student team leader for the KySat-1 mission.*

Dr. James E. Lumpp, Jr. *is an Associate Professor in the Department of Electrical and Computer Engineering at the University of Kentucky. He received the BSEE and MSEE degrees from the School of Electrical Engineering at Purdue University in 1988 and 1989 respectively, and the PhD from the Department of Electrical and Computer Engineering at the University of Iowa in 1993. He joined the faculty at the University of Kentucky in 1993. His research interests include distributed embedded systems, safety critical systems, aerospace education, and small satellites.*