THE DESIGN OF A SOFTWARE SYSTEM FOR A SMALL SPACE SATELLITE

BY

MICHAEL J. DABROWSKI

B.S., University of Illinois at Urbana-Champaign, 2003

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

CCA FORM

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Overview

This paper will cover the design and implementation of a software system for a small satellite. The software system developed is responsible for performing all of the operations of the satellite including control of onboard hardware devices, scheduling of operations, maintenance of data, and communications with a ground station on Earth. The system developed was designed and built by a small group of students over the course of eighteen months for the ION satellite as part of the Illinois Tiny Satellite Initiative.

First, the design process of the ION satellite software system leading up to the resulting satellite interface will be discussed. The details of the design and its implementation will then be covered, followed by an overall analysis of the system. The paper will conclude with further technical details, recommendations for future small satellite developers, and comments on general difficulties encountered.

1.2 Small Satellites

In the past few years much of the attention of the space industry has shifted towards the development of small satellites. These satellites, often called picosats, nanosats, or microsats are generally less than 200 kilograms and, in many cases, are as little as 10-50 kilograms. Such satellites, which range in size from refrigerators to small soda cans, offer many potential benefits over traditional space satellites [1].

Traditional space satellites are typified by geostationary communications satellites which range in mass from 1000 to 4000 kilograms [1]. Such satellites require millions of dollars to develop and have historically been large expensive projects requiring five to ten years to construct. Because of the enormous costs and time allocated to such projects, very little risk tolerance exists. As a result, very little room exists for innovation and such satellites are often limited to the use of space-proven, though often outdated, technologies. Furthermore, an enormous amount of money and effort is placed into the development of redundant systems and the maintenance of outdated techniques and procedures. As a result of the resources required, the development of traditional satellites has historically been limited to first world counties with large military and commercial budgets.

Small satellites provide an amazing alternative to traditional space satellites. Such projects are driven by a "smaller, faster, better, cheaper, smarter" mentality which allows for a fully functioning space satellite to be built in a fraction of the time and cost of a traditional space satellite [1]. Often, such satellites may be designed, built, and launched within a period of six to thirty-six months with labor investments of a few to ten man-years.

As a result of the inexpensive nature and short development time of small satellites, project developers are more willing to accept higher risks and an increased probability of mission failure. The designs of small satellites are more open to the use of new, unproven technologies. Often such technologies not only reduce the size, weight, and cost of the satellite, but also greatly increase the available functionality. Small satellite projects are also able to accept higher risk payloads, allowing for more interesting satellite experiments. Furthermore, as a result of resource limitations, small satellite developers are often forced to experiment with new and innovative designs, techniques, and procedures.

One of the driving philosophies of small satellite design is the use of standard, easy to use, commercial off-the-shelf (COTS) components designed for nonspace applications. This allows for fast and inexpensive construction, reducing satellite complexity. The use of standardized platforms and reusable components further shortens the development process.

The low cost and limited time investment required to construct small satellites greatly reduces the cost of entry to space. Such projects make space much more accessible to amateurs, researchers, entrepreneurial efforts, and small governments [2], [3]. Over the past decade an enormous variety of small satellites have been developed including a large number of educational efforts by universities [4] - [7].

1.3 CubeSat Satellites

To both further speed the development process and aid in obtainment of a launch opportunity, many university small satellite projects choose to follow the CubeSat specification [8]. This standard outlines a set of physical launch interfaces as well as mechanical requirements for a small satellite.

According to the specification, a standard CubeSat satellite is a 10 x 10 x 10 cm cube weighing at most one kilogram. Currently, up to three such cubes may be combined, creating a single satellite of dimensions 10 x 10 x 30 cm with a maximum mass of 3 kg.

In addition to providing a standard launch interface, the CubeSat program at the California Polytechnic State University creates a community for developers of educational satellites, provides satellite integration services, and attempts to provide launch opportunities [9]. By combining multiple small satellites together in a single package through the CubeSat program, universities are able to present themselves as a more attractive customer to launch providers.

Small satellite projects such as CubeSat often are troubled by a similar set of difficulties and limitations. Educational projects are often limited in terms of time, student experience, financial budget, and development infrastructure. Payloads are very limited in terms of physical space available due to the extremely restricted size and mass of a CubeSat satellite. This seriously restricts payloads requiring large optics or bulky components.

The limited surface area of a CubeSat restricts the amount of solar power that may be generated, restricting power available for computation, communications, and payloads. Restrictions on space, time, and power necessitate that CubeSat satellites incorporate limited payloads, slow communications links, little redundancy, and minimal information processing capabilities. Nevertheless, many CubeSat satellites are surprisingly complex and ambitious. One such satellite is the ION satellite constructed at the University of Illinois.

1.4 The University of Illinois ION Project

The ION satellite is a two-cube CubeSat satellite constructed at the University of Illinois over the course of 4 years. The primary mission of the ION satellite is to provide a large interdisciplinary educational project for undergraduate engineering students.

The ION satellite project was run as an interdisciplinary engineering class in which senior year students typically participated for two semesters. The development of the satellite was entirely student driven with all work performed by five to six teams of three to five students and two teaching assistants. Three faculty advisers provided both active mentorship and logistical and financial support. ION is currently expected to be launched into a 650-km low earth orbit along with 13 other CubeSat satellites.

While built largely as an educational project for students, the ION satellite also has a number of science and technology mission objectives. First, ION is to measure an oxygen airglow brightness in the Earth's upper atmosphere using a filtered photometer. Oxygen atoms

located at an altitude of 90 km recombine to emit a dim glow of light not visible from the ground due to atmospheric absorption. Oxygen density can be determined by measurement of this emission which brightens and dims as a result of disturbances from waves in the atmosphere. Satellite mapping will aid in understanding wave energy transfer across large areas of the Earth's atmosphere. This will contribute to knowledge of atmospheric dynamics in the upper atmosphere.

Second, ION is to test a microvacuum arc thruster (μVAT) system. These thrusters operate by forming an electric arc across an anode and a cathode. Cathode material is vaporized and ejected at a high velocity, thereby producing thrust. The successful demonstration of this technology will aid in the development of an efficient and versatile low-mass satellite propulsion system [10].

Third, ION is to test the use of a black and white CMOS camera for photography of the Earth. Results from this imaging system will guide the design of imaging systems onboard future small satellites.

Finally, ION will attempt to demonstrate the use of an active attitude control system. Magnetic torque coils will generate magnetic fields which will interact with the Earth's field, creating the necessary torque to orient the satellite into favorable positions. Feedback on satellite orientation will be provided by a collection of sensors. The successful demonstration of active attitude control onboard a CubeSat satellite will be critical in establishing small satellites as a viable platform for payloads requiring a high degree of control over satellite orientation.

In addition to the sensors and actuators previously detailed, the ION space satellite incorporates a number of additional components. ION is powered by solar panels and batteries. Communication with a ground station on Earth is made possible through the use of a communications system composed of a radio and a modem. All operations performed by ION are controlled by a flight computer.

This thesis will outline the design and implementation of the software system running onboard the ION satellite. The software's primary purpose is to fulfill all the mission objectives of the satellite. The process of designing the software system will be traced in Chapter 2. This discussion will generally be nontechnical and will end with a description of the resulting method of ION satellite operations. Chapter 3 will give a more technical overview of the ION software implementation, including in-depth discussion of the four major components of the software

system. Further technical details of the software system as well as a nontechnical discussion of lessons learned will be detailed in Appendices A and B.

2. DESIGN PROCESS OF THE ION SATELLITE SOFTWARE SYSTEM

2.1 Overview of the Design Process

In this chapter the design process of the ION satellite software system will be outlined. When development of the software system began, no detailed mission specification for the satellite existed. Instead, the details of the satellite's mission had to be "reverse engineered" from the collection of hardware onboard ION. This hardware, in addition to the satellite mission determined, suggested the implementation of a generic scheduling system which treated the ION satellite as a passive collection of instruments. As a result, all operations performed by the ION satellite must be explicitly scheduled by operators on the ground.

First, the physical electronics hardware onboard ION will be described. A knowledge of the hardware onboard ION, along with its functionality and limitations, will allow for the formal specification of a general set of satellite mission requirements. The software requirements of the satellite will then be determined as a result of both the electronics hardware present and the determined mission specification. These software requirements will suggest an appropriate software model to be implemented. The details of implementation will be discussed in Chapter 3. This chapter will conclude with a description of the resulting interface to the satellite and a description of how the ION satellite is operated.

2.2 Physical Components Onboard the ION Satellite

To help clarify the mission requirements of the ION satellite and illustrate what problems the ION software system had to solve, the hardware components onboard ION, along with their operation, functions, and interface are described. Nearly all of these components are found to require very detailed control from the single flight computer onboard the satellite. This complication, termed the "Dumb Device" problem, suggests that the operation of hardware devices will be a primary responsibility of the software system running on the SID flight computer. The unique nature of the SID computer suggests that the majority of the required software would need to be developed from scratch.

2.2.1 Listing of components

In order to fullfill the general mission that was pictured for the ION satellite, a variety of electronics were placed onboard. These electronic components are grouped into five main areas: the flight computer, components specific to the scientific payload of the satellite, components specific to satellite attitude determination and control, components specific to communications with a ground station on Earth, and components which provide fundamental electrical support. Nearly all of the electronics components directly interface to the flight computer as illustrated in Figure 2.1.

The flight computer

Nearly all space satellites have at least one flight computer devoted to performing tasks related to communications and data handling. Many satellites have additional computers devoted to the operation of specific subsystems or tasks such as attitude control or scientific payloads. This allows for simple delegation of operations and may provide redundancy in the event of computer failure.

The SID

The ION satellite contains only one computer. A central computer known as the Small Integrated Datalogger (SID) performs all processing onboard ION. This credit card sized single-board-computer is a first revision of a commercial product and is built around a RISC microprocessor running at 7 MHz. The SID computer has a number of features including: 256 KB of EEPROM, 1 MB of RAM memory, 8 MB of nonvolatile flash storage, 2 serial ports (UARTs), 3 real time clocks, 32 channels of analog input, 24 general purpose input/output lines, 4 power output lines, built in latchup protection, and a built-in hardware watchdog timer [11].

Nearly all of the hardware onboard ION interfaces to the SID and must be directly controlled by this computer.

Payload components

Most space satellites have a specific mission performed by their main payload

components. The general scientific mission of the ION satellite consists of measurement of an oxygen emission from the Earth's upper atmosphere, testing of microvavuum arc thrusters, and photography of the Earth. The components below provide the functionality needed to perform ION's scientific missions.

CMOS camera

To fullfill the photography mission of the ION satellite, a small black and white CMOS camera is used. The camera is directly powered by the SID over one of the SID's power output lines. Whenever powered, the camera streams gray scale images at a resolution 640 x 480 pixels at 24 frames per second

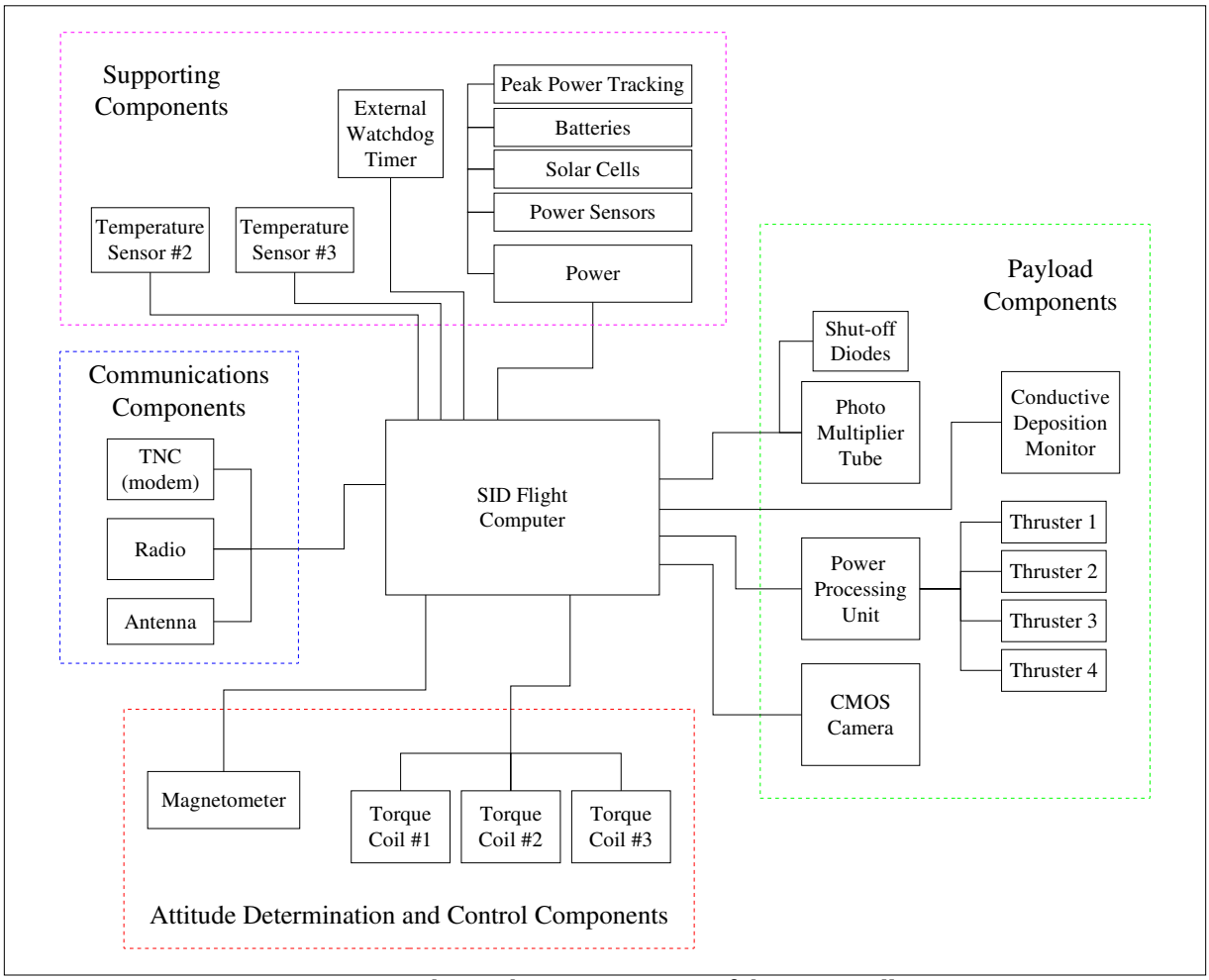

Figure 2.1. The Hardware Components of the ION Satellite

Nearly all devices onboard ION interface directly to a single flight computer. These devices are very simple and must be directly operated by the ION software leading to a complication referred to as the "Dumb Device" problem.

directly to the SID through an interface of eight data lines. An additional I^2C interface to the camera allows the setting of registers to control camera operation.

The SID is responsible for controlling power to the CMOS camera, setting camera register values, and collecting digital image data streamed from the device.

Thrusters

Four microvacuum arc thrusters are onboard ION. These thrusters are wired to a device known as the Power Processing Unit (PPU), which is responsible for generating the necessary voltages for proper thruster firing. The PPU is wired directly to five digital output lines on the SID, used to enable the firing of the thrusters.

Feedback on the performance of thrusters is provided by a sensor known as the Conductive Deposition Monitor (CDM). The electrical resistance of this device changes as material vaporized by the thrusters accumulates on the sensor. The CDM is wired to a power output and analog input line on the SID.

The SID is responsible for enabling the firing of each thruster, powering of the CDM, and sampling the analog output of the CDM.

Photo multiplier tube

In order to map airglow phenomena in the atmosphere, a Hamamatsu H7155 Photo Multiplier Tube (PMT) is included onboard ION. Whenever a photon of light strikes the sensor of the PMT a 5-V pulse is produced on a digital input line connected to the SID. Power to the PMT is controlled by a signal line from the SID, but this signal is mitigated by a set of photo diodes which prevent powering of the PMT when light intensity is too great. A second output line from the SID allows the computer to override the photo diode power cutoff.

The SID is responsible for controlling the power to the PMT, counting pulses on an input line, and, if necessary, issuing a signal to override the photo diode cutoff.

Attitude determination and control components

To properly orient themselves in space, satellites often perform active attitude control. In order to do this, sensors providing feedback on spacecraft orientation along with an actuator mechanism to perform control are required. On many satellites, this functionality is provided by an independent subsystem which includes its own processor. The ION spacecraft has no such luxury; instead the individual sensors and actuators are “dumb" devices which must be controlled directly by the SID computer. The components below provide the functionality needed for ION to perform active attitude control.

Torque coils

In order to generate the necessary magnetic fields used to orient the spacecraft, ION incorporates a magnetic torque coil on each of its three axes. Each of these coils is driven by an h-bridge chip controlled by two digital output lines from the SID. The strength of the magnetic field generated is determined by the duty cycle of a square wave signal output by the SID onto one of the control lines. The direction of the magnetic field is determined by a signal on the second line.

The SID, therefore, enables three onboard magnetic torque coils by generating the appropriate square wave and direction signals.

Magnetometer

To aid in determination of the satellite's attitude in space, a Honeywell HMC20003 three-axis magnetometer is used to measure the Earth's magnetic field. When powered, this device produces three analog signals corresponding to a measured magnetic field strength directly to analog inputs on the SID. Power to the magnetometer is controlled by a output control signal from the SID. A set/reset circuit that is used to remove magnetic history and temperature effects is packaged with the magnetometer and controlled by two output lines from the SID.

The SID is responsible for controlling power to the magnetometer,

generating the appropriate set/reset signals, and sampling the analog output of the magnetometer.

Solar panels

Determination of the satellite's attitude is further aided through the use of current readings of the onboard solar panels. By recording the electrical current generated by a solar cell it is possible to use the cell as primitive sun sensor to gage the position of the sun relative to the spacecraft. Five solar cells are directly wired to analog input lines on the SID to record current generation.

The SID is responsible for sampling the analog output of the five solar panels.

Communications components

Communication between the Earth and a satellite is commonly accomplished through the use of a radio along with some form of modulation of digital data. In some cases these functions are performed by an independent subsystem with its own processor. ION uses the SID flight computer along with a modem and a radio to perform communications functions.

Terminal node controller

To modulate outgoing digital information and demodulate a received analog signal, a PicoPacket Terminal Node Controller (TNC) is included onboard ION. In addition to acting as a 1200 baud modem, this device also implements the AX.25 communications protocol which specifies a data format for digital information [12]. This device is directly connected to a serial port on the SID and to the onboard radio. Power to the TNC is controlled by a control signal output from the SID.

The SID is responsible for serial communications with the TNC and controlling the power of the TNC.

Radio

ION communicates with a ground station through the use of a TEKK

KS960 two watt radio. To send data to Earth, the radio receives an encoded audio signal from the TNC and transmits it as a radio signal. Data from Earth is obtained by receipt of a radio signal which is output as an audio signal to the TNC for decoding. Power to the radio is controlled by a control signal output from the SID.

The SID is responsible for controlling power to the radio.

Antenna

A dipole antenna is included onboard ION for use by the radio. Upon launch of the satellite the antenna is in a stowed configuration and must be deployed once ION is in orbit. Deployment of the antenna is controlled by a control signal output from the SID.

The SID is responsible for assertion of a signal to deploy the antenna.

Supporting components

Three additional hardware components which perform basic functions for satellite operation are included on ION.

Temperature sensors

In order to record the temperature onboard ION, three Dallas Semiconductor DS1820 temperature sensors are included. These devices are wired directly to a serial port on the SID and can be queried for a temperature reading using the Dallas Semiconductor 1-Wire communications protocol [13].

The SID is responsible for querying the onboard temperature sensors over a 1-wire bus.

Watchdog timer

In addition to the watchdog timer incorporated on the SID computer, an additional external watchdog timer is included on ION. This device acts as a safety mechanism, disconnecting power to the entire satellite for a period of two minutes if it does not regularly receive a signal from the SID specifying that

everything is running well. The watchdog timer is connected directly to one of the digital output lines of the SID.

The SID is responsible for regularly “kicking the dog” in order to prevent system reset.

Power system

Power onboard ION is provided by a power system consisting of two lithium-ion batteries, 20 solar panels, a custom built power control board, and a custom built peak power tracking (PPT) board. Sixteen channels of analog output from the power control board specifying system voltages and currents are wired directly to analog inputs on the SID. Battery charging, battery use, and solar power use are controlled by the power control board as a result of four input signals from the SID.

The SID is responsible for sampling the analog output of the power control board and control of four output signals specifying power policy.

2.2.2 Discussion

From the listing of devices and their interfaces it can be seen that nearly every electronics component onboard ION must be operated directly by the SID computer. Operation details of these devices require specific timings and control of signal lines. Under ideal conditions this would have been performed by individual processors dedicated to performing the operations of a subsystem. Unfortunately, very few of these devices are packaged as such intelligent hardware “black boxes” and therefore a large amount of responsibility is placed upon the SID computer and its software. We refer to this complication as the "Dumb Device" problem.

The SID flight computer is a very unique and proprietary piece of hardware for which very little software has been developed. In fact, only about 12 such computers exist and support is very limited. As a result of the minimal support available for the SID along with its unique nature, it was quickly determined that standard software systems such as Linux could not be easily adapted to the ION satellite. Instead, the majority of the software written for the ION satellite would need to be written specifically for the SID computer.

2.3 Determined ION Operational Requirements

The previously detailed hardware onboard ION and the general scientific mission were used to determine a comprehensive, though very general, list of ION satellite operational requirements. The operations determined were the obvious result of the functions performed by onboard hardware components and were found to naturally fall into three categories: input sample functions, output actuator functions, and communications functions. Very few of these operations were found to occur autonomously, allowing the entire ION satellite to be treated as a passive collection of instruments controlled from the ground. This suggested that providing a robust system of remote control would be another primary responsibility of any software system onboard the ION satellite.

2.3.1 Introduction

Space satellites typically perform a standard set of tasks. First, satellites perform power management operations, including control of solar panel operation, management of power distribution, and control of battery charging. Many satellites also perform attitude control in order to allow them to maintain specific orientations with respect to the Earth's surface. Attitude control may be performed passively using magnets and gravity gradient stabilization, or may be actively performed by a control mechanism such as gas thrusters, magnetic torque coils, or momentum wheels.

Communications and data handling operations are performed by most satellites. These operations include transmission of telemetry data to Earth, receipt of commands, management of data, and execution of any necessary computations.

Finally, satellites perform what can be termed payload operations. These operations are specific to the payload of a satellite and may include imaging tasks, communications relay tasks, or tasks to perform scientific experiments.

Traditionally, each of these four main areas of functionality has been performed by an independent hardware subsystem. In the ION satellite this was not the case, as explained in Section 2.2.1. Onboard ION, the central flight computer and its software would be responsible for performing any satellite operations that were determined to be necessary. To guide the

development of the ION software system it was therefore important to formally outline all of the operations that the ION satellite would be expected to perform.

2.3.2 ION operational requirements

The operational requirements listed below are an expanded and organized list of those deemed necessary in Section 2.2. The functionality has been organized in terms of items that perform an input function, items that perform an output function, and items that perform a communications function. The operational requirements are defined in a very high level manner; little consideration is yet given as to how these functions are actually performed.

1) The following input actions are to be performed:

Take and record a power sample at undefined times and at an undefined frequency.

Take and record a picture at undefined times and at an undefined frequency.

Take and record a temperature reading at undefined times and at an undefined frequency.

Take and record a magnetometer reading at undefined times and at an undefined frequency.

Take and record a photo multiplier tube count at undefined times and at an undefined frequency.

Take and record a conductive deposition monitor sample at undefined times and at an undefined frequency.

Take and make use of a real time clock sample at undefined times and at an undefined frequency.

2) The following output actions are to be performed:

Fire magnetic torque coils at undefined times and at an undefined frequency.

Fire microvacuum arc thrusters at undefined times and at an undefined frequency.

Control power of the communications system at undefined times and at an undefined frequency.

Issue a beacon over the communications system at undefined times and at an undefined frequency.

Release the communications antenna at an undefined time.

Kick the external watchdog timer at undefined times and at an undefined frequency.

Output appropriate signals for power management as a function of system state.

3) Communications with a ground station on Earth is to be performed:

Return data resulting from satellite operations to the ground station.

Accept commands specifying operations to perform from the ground station.

These operational requirements are graphically illustrated in Figure 2.2.

2.3.3 Discussion

The specification of formal satellite mission requirements should be completed as a group effort with much oversight very early in the project development cycle. Unfortunately, the operational requirements of the ION satellite were not developed as a result of such an effort. Instead, these requirements were only formally determined and detailed by the software development team to provide direction in determination of what problems the ION software system would be required to solve. As a result, the determination of what functions the ION satellite would perform was left to the whim and limited knowledge of a small group of software developers.

The ION operational requirements detailed in the previous section were driven by the hardware onboard the satellite. This influence can be seen very clearly in the listing of input and output functions. Nearly all of the operations expected to be performed by the satellite are simply the obvious result of the functions performed by onboard hardware. The requirements which were detailed were extremely general and written in a manner which allowed for determination of mission specifics in the future. For example, nearly all operations performed by the ION satellite are required to be performed at any time with any frequency.

The determined satellite operational requirements begin to suggest a number of software requirements and software design decisions. First, because of the Dumb Device problem, it is

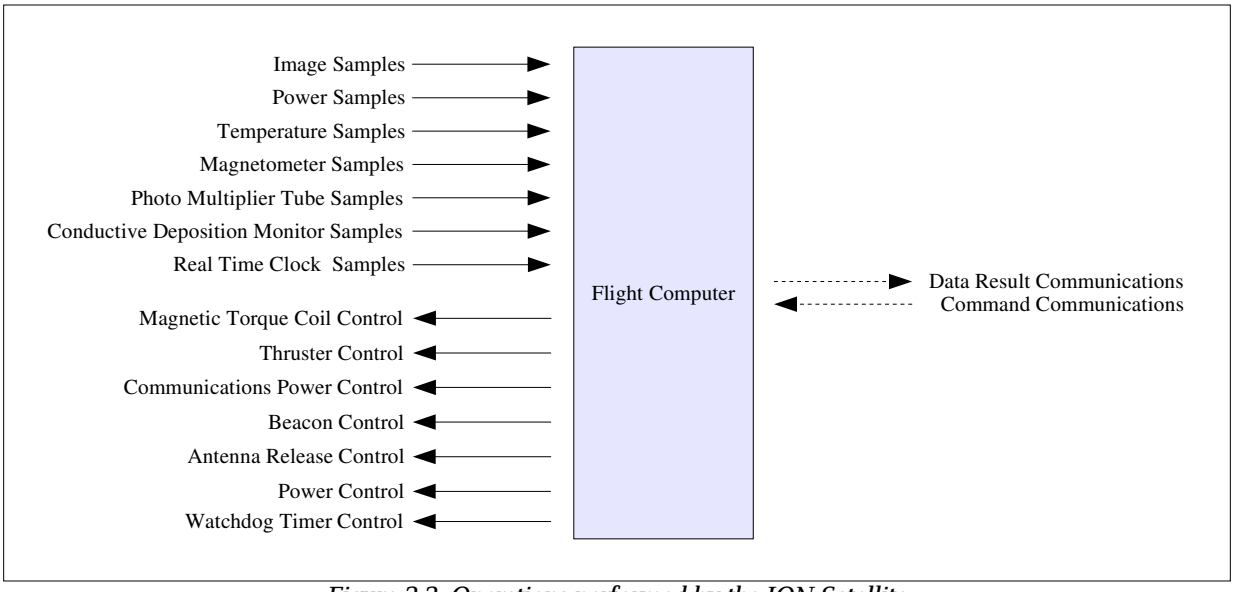

Figure 2.2. Operations performed by the ION Satellite

All ION Satellite operations can be classified as input, output, or communications functions.

clear that the central flight computer will be responsible for performing all of the above functions. None of the above actions will be automatically performed by a hardware black box, suggesting that the software system on the SID will be relatively complex. The software system will span both a number of functions and a number of levels of functionality, from low level control of a device to high level recording and management of resulting data. Any software model to be developed should segment both the different functionality needed as well as differing functionality levels.

Second, nearly all of the functions performed by ION can be classified as either the control of an actuator device or the reading of a sensor. Even the real time clocks on the SID can be treated as time sensors. This provides for a relatively simple view of the majority of the hardware onboard ION despite its wide range of functionality and manner of operation. Additionally, this suggests that there should be a way to develop a small number of reusable software interfaces to perform all of the hardware input and output functions of the satellite.

Third, all of the functions the ION satellite must perform, with the exception of communications functions and power management, can simply be considered regular, though arbitrarily scheduled, operations of a hardware device. The only autonomous activity that must occur onboard ION is radio communication with the ground and control of the power board. Standard satellite functionality such as active attitude control is not autonomously performed onboard ION. Therefore, nearly the entire satellite may be treated as a passive device which is somehow controlled from the ground.

The view of ION as a passive device is a significant departure from the usual manner of satellite operation in which a satellite may perform the majority of functionality autonomously with little control from the ground. This suggests that one of the major functions of any software system onboard ION will be to provide a robust system of remote control. This reinforces the requirement that the communications mechanism operating onboard ION must accept commands specifying satellite operations in addition to data return.

2.4 Determined Software Functional Requirements

With knowledge of the hardware onboard ION and a formal specification of required satellite operations, a three-tiered system of software functional requirements was developed. Every operation the ION satellite performs was found to require two pieces of software: a

mechanism to schedule execution of the operation and a mechanism to implement the operation. The software functional requirements developed led to a software model in which satellite subsystems, traditionally implemented in hardware, would be simulated as software subsystems. Each subsystem was determined to be responsible for management of its own data as well as its own schedule of operations, furthering the view of the ION satellite as a passive collection of schedualable instruments.

2.4.1 Tier 1 - Direct requirements

The first tier of software requirements are externally imposed on the software system. These requirements directly result from the previously determined satellite operational requirements along with the limitations and requirements of the hardware onboard the ION satellite.

As determined in Section 2.3.3, the majority of ION satellite functions may be classified as regularly, though arbitrarily, scheduled input or output actions. As a result of the Dumb Device problem, the software system must solve two problems for each operation the satellite performs. First, a mechanism is needed to regularly perform the required operation based upon a schedule or command received from the ground. Second, a mechanism to handle the exact details of performing the operation is needed.

For example, the ION satellite functional requirement to "Take and record a picture at undefined times and at an undefined frequency" can be considered to consist of two main problems. The first problem requires the creation of a mechanism to interpret a request from the ground to take a picture, begin the process of taking a picture, and appropriately store the resulting image. The second problem requires the creation of a mechanism for the hardware details of making the camera take a picture. This includes: powering on the camera, configuring camera settings, starting a memory transfer operation, and powering off the camera.

The direct software system functional requirements are therefore:

(1) Provide a mechanism to handle the low level details of operating every hardware device onboard the ION satellite.

As a direct result of the Dumb Device problem a mechanism is needed to

control each of the hardware devices on ION. Ideally the mechanism should completely hide the operational details and provide a simple way to use the device. Furthermore, since all devices can be simply considered sensors or actuators, it should be possible to use a similar design across all devices.

(2) Provide a mechanism to use each device or group of devices appropriately based upon an arbitrary schedule or, in limited cases, system state.

The requirement to use each device at arbitrary times necessitates a mechanism to schedule device use. Ideally the mechanism should continue to hide the details of device operation of each device or group of devices and should record any data that is created as a result of device operation. The notion of performing regularly scheduled work and recording resulting data is relatively generic; thus, it should be possible to use a similar design across all devices regardless of the specific operational details.

(3) Provide a mechanism for returning recorded data to earth and for accepting sequences of commands.

The passive nature of the satellite and the requirement to use devices at arbitrary schedules makes it clear that a large amount of interaction with the satellite from the ground must occur. The relatively undefined nature and frequency of device use along with the variety of data collected requires that this mechanism provide for the bidirectional transfer of generic data of any format.

Decisions made

It has been previously noted that traditionally many satellites have separated their functionality into multiple physically independent hardware subsystems. Each hardware system would be responsible for performing a single function such as attitude determination and control or operation of a payload device such as camera. The hardware design of ION does not allow for this. Instead, a single central computer is required to perform all satellite operations. The separation of physical systems is therefore simulated through a software design that provides

segmentation of associated functionality.

It was realized that each group of associated satellite functionality could be performed by a software subsystem. Each subsystem would consist of two parts, corresponding to the two problems which needed to be solved for each satellite operation. The first part would be a generic mechanism, known as an *application*, which would use a hardware device or group of devices based upon either a schedule or system state. The second part, known as a *device driver*, would be a mechanism to solve the Dumb Device problem. Each subsystem would be responsible for handling its own data and managing its own schedule of operations to perform, thereby completely hiding the details of the operation of the subsystem. Figure 3.1 on page 27 illustrates an ION subsystem simulating what is typically a hardware subsystem.

2.4.2 Tier 2 - Secondary requirements

As a result of the decision to build the ION satellite software system around multiple software subsystems devoted to performing small sets of satellite operations, a second group of software requirements immediately became apparent.

(1) Provide a mechanism for allocating processing time and other resources to software subsystems.

Each independent software subsystem needs to be given processing time to run on the central computer as well as any other resources it requires. A mechanism was needed to allocate these resources to each subsystem. Ideally the mechanism would hide the details of resource allocation and provide a simple way to use the entire satellite from the ground. This mechanism should allow for some limited interaction between subsystems, allowing individual subsystems to affect other portions of satellite operations.

(2) Provide a mechanism to store data.

Because each independent subsystem was determined to be responsible for management of its own data, it was clear that a data storage mechanism would be necessary. Data used and generated by subsystems may be of large size and undefined format; therefore, this mechanism must be generic. Furthermore,

the mechanism should be easy to use, should hide the details of its operation, and should allow for data to persist across system reboots.

Decisions made

The best manner in which to fulfill both the requirement of segmentation between software simulations of satellite subsystems and the requirement to easily allocate processing resources to such simulations, was determined to involve the use of independently executing tasks. Each task would be responsible for execution of the software performing the functions of a satellite subsystem. Processing time would be provided to subsystems through the use of context switching.

In order to permanently store and access general data onboard ION it was decided to implement a file system to use with the flash memory chip onboard the SID computer. The file system would need to provide the ability to create, read, write, and delete files.

2.4.3 Tier 3 - Supporting requirements

In addition to the above outlined software requirements it was clear that the ION software would need to implement a large number of small details that support the implementation of the above requirements. Few of these requirements were identified initially and most were simply developed as the need for then arose. These requirements can be grouped into two main areas.

(1) Provide support for the previous requirements in a reusable manner.

Very few details of implementation are listed in the first two tiers of requirements and it is clear that a large amount of supporting software would need to exist that performed small specific tasks. This software should be written in a manner that would allow for its reuse whenever necessary. Examples of such software include compression of images, implementation of standard data structures such as queues, and support for performing data integrity checks.

(2) Provide support for reliability, safety, and software recovery

In addition to directly implementing the satellite mission, the software system needed to provide some level of reliability, redundancy, and features for

recovery in the case of any errors. Few of these requirements were identified initially and most were developed as time allowed. Examples include support for new software updates from the ground, recovery mechanisms in the case of device failure, assessment of general system health, and logging of system activity.

Decisions made

The majority of the functionality required from this tier was not initially known, and as a result, few design decisions were made. Instead, most of this functionality was designed and written when necessary.

As none of these requirements directly contribute to any of the identifiable satellite mission requirements, it is tempting to consider them of lesser importance. In reality, a large portion of critical software infrastructure falls within the third tier of requirements. Despite this, because of the unknown nature of these requirements, very little of this functionality was considered early on in the design process.

2.5 Resulting ION Satellite Interface

Given a space satellite which could be considered a passive collection of instruments to be used at arbitrary times, an operations interface was developed in which a user on Earth would upload schedules of work to perform and in exchange would download the results of any work performed. All operations onboard ION are abstracted away by the ION satellite interface which requires users on the ground to explicitly schedule every operation performed by the satellite, effectively turning it into a very expensive radio controlled toy.

2.5.1 Introduction

Interaction with space satellites, especially small satellite such as ION, is limited as to two main problems.

First, communications windows are very short and infrequent as a result of the typical low earth orbits of small satellites. In general, communications between a satellite in low earth orbit and a single ground station on Earth is limited to two 15 min windows every 24 h. The majority of satellite operations must occur while there is no contact with the satellite. This

requires a satellite to either perform activity autonomously or based upon a predetermined schedule.

Second, data bandwidth between small satellites and ground stations is often very limited. Communications links are slow as a result of limitations of power, financial budget, and physical space. A slow communications link, coupled with a short communications window, means that very little data can be transfered between a satellite and a ground station. It is expected that within a single communications session fewer than 100 kilobytes of data can be transfered between ION and a ground station.

The limitations on contact time and data transfer amounts forces satellites to operate largely autonomously and perform a large amount of processing and decision making onboard. The ION satellite, on the other hand, uses a different model of operation stemming from the passive nature of its operations.

2.5.2 ION satellite operations

Because ION is rarely in contact with the Earth, most operations must occur while there is no possibility of commanding the satellite from the ground. Therefore, satellite operations must be previously scheduled to occur when there is no contact between the satellite and Earth.

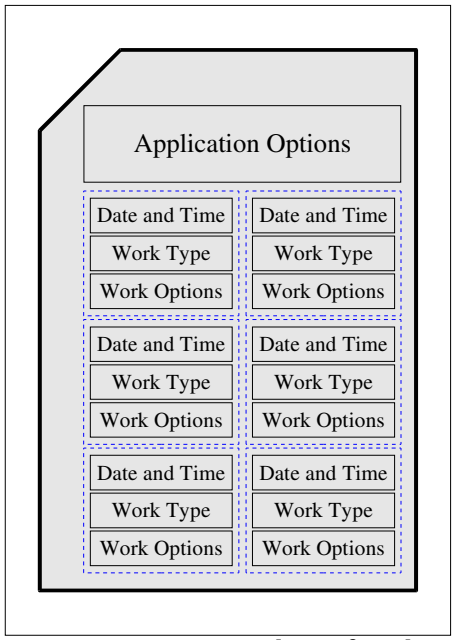

Figure 2.3. An Example Config File

A config file, consisting of a schedule of work to perform, provides the details of subsystem operation.

Command of the ION satellite begins at the ground station where a user uses custom software on a Linux computer to generate a schedule of work for each software subsystem onboard the satellite. These schedules of work consist of multiple pieces of data known as *work units* and are stored in a standard format known as a *config file*. An example config file is illustrated in Figure 2.3. Each work unit specifies an operation to perform, the date and time it should be performed, and any associated options with the operation. In addition to storing a collection of work units, each config file also specifies some global details of operation for a subsystem.

Once a collection of config files has been created on the ground, the next time communication is established with

ION these files are transmitted to the satellite using a custom communications protocol implemented as another piece of Linux software.

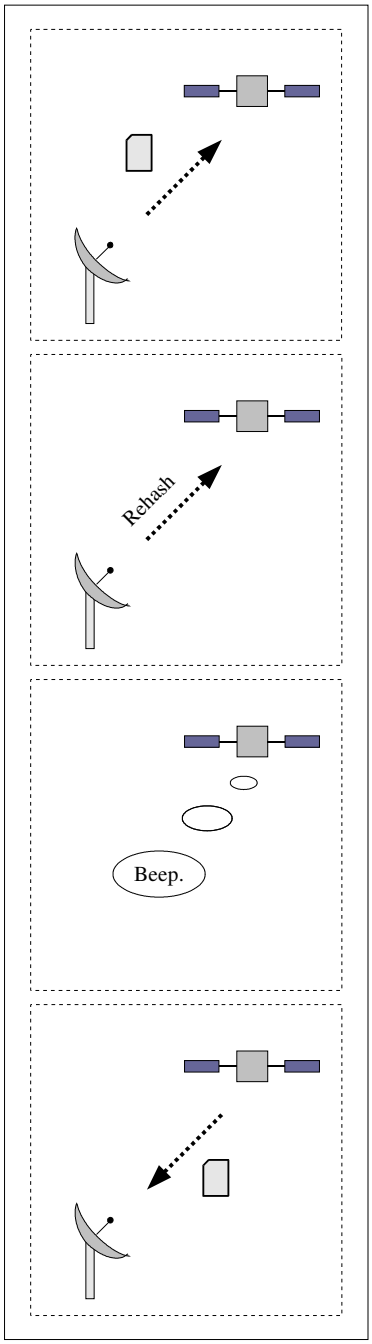

Figure 2.4. ION Operations

A config file containing a schedule of work is first uploaded. Following a Rehash the satellite performs the work scheduled. Results of operations are later downloaded.

After the config files have been uploaded to the satellite, each running subsystem onboard the satellite is commanded to read and interpret its uploaded config file using a special command built into the communications protocol known as Rehash().

The satellite communications window soon ends and over the course of the next 24 h each subsystem does its best to perform the work requested. In most cases all work scheduled will be executed. For reasons of system schedulability or safety, however, some work may be dropped, though it is the responsibility of the user creating work schedules to make sure that the expectations placed upon the time and power margins of the satellite are realistic and that no damage will occur as a result of execution of the scheduled operations.

During the next communications window any data files that were created as a result of satellite operations are downloaded. These files may include samples from various instruments, system logs, or even photographs. Interpretation and processing of the resulting data are performed on the ground and any operations that are to be taken as a result of returned data are encoded as work units for upload. The complete process of ION operations is illustrated in Figure 2.4.

2.5.3 Discussion

The described interface to the ION satellite is only possible because the satellite may be treated as a passive collection of occasionally used input and output devices. As very little is happening autonomously onboard, the satellite can be considered the equivalent of a very expensive radio controlled

toy. The operation of all instruments is performed through a communications channel which multiplexes the command of operations through a system of file uploads and downloads.

The end result is that operation of every instrument onboard ION is completely abstracted away to the user on the ground. A user specifies a time and manner in which to use an instrument and the operation is silently performed. Any resulting feedback from the operation may then be obtained a number of hours later. This model of operation allows for ground based active attitude control or other closed loop control of ION to occur on a very large timescale.

2.6 Summary of the Design Process

In this chapter the design process behind the ION satellite software system was detailed. The hardware onboard the ION satellite and general mission specifications were used by the software development team to develop a formal set of satellite operational requirements. Very little functionality was needed to occur onboard ION autonomously and most functionality could be considered a scheduled input or output operation.

The satellite operational requirements along with limitations of the onboard hardware suggested a set of software functional requirements which were organized in terms of three tiers. These requirements were used to determine some general guidelines for the design of the software system. The decision to treat the entire satellite as a passive collection of instruments, each operated by a piece of software simulating a hardware subsystem, was outlined. Finally, the method of operating ION by transmitting schedules of work and receiving resulting data was described.

The method of implementation of all of this has not yet been described and follows in Chapter 3.

3. ION SOFTWARE SYSTEM DESIGN

At this point, the reader should be familiar with the hardware onboard the ION satellite, the operations performed by the satellite, the problems the ION software system solves, and how the satellite is operated from the ground. In this chapter the implementation details of the satellite software will be covered.

First, an overview of the components of the ION software system will be provided. The design and implementation of these components will then be covered, along with discussion of design benefits and limitations. The components will be covered in the order of development which will aid in the understanding of the relation between components. The chapter will end with a discussion of the system design along with suggestions for future improvements.

3.1 Overview of the System Design

The ION satellite software consists of four main components each responsible for fulfilling one or more of the software functional requirements. These components are outlined here in order of tier of functionality.

TIER 1. Device drivers and applications

Two software components known as *applications* and *device drivers* completely fulfill the first tier of the software functional requirements outlined in the Chapter 2.

As a result of the Dumb Device problem, any software system written for the ION satellite was first required to provide a mechanism to handle the low level details of operating every hardware device onboard the satellite. This was accomplished with a set of software components referred to as device drivers. Typically, each device driver is responsible for directly interacting with a single hardware device and completely abstracts away the details of device operation. Nearly all of the hardware devices onboard ION can be considered sensors or actuators so all device drivers follow a very similar design providing a simple way to perform an output action or perform an input sample.

The ION software system was next required to provide a mechanism to use each device or group of devices at the appropriate time based upon input from the ground. This was accomplished with a set of software components referred to as applications. Typically, an application interacts with one or two device drivers in order to operate hardware devices as

needed. Applications perform all management of data associated with device operation, including interpreting and maintaining schedules of work from the ground and recording results of operations into files on the file system.

An application, plus any associated device drivers used, completely abstracts away the details of operation of a satellite subsystem. Figure 3.1 illustrates an ION subsystem simulating what is traditionally a hardware subsystem. A total of nine such subsystems run on the SID computer. Each is implemented by one of nine applications and uses one or more of sixteen total device drivers.

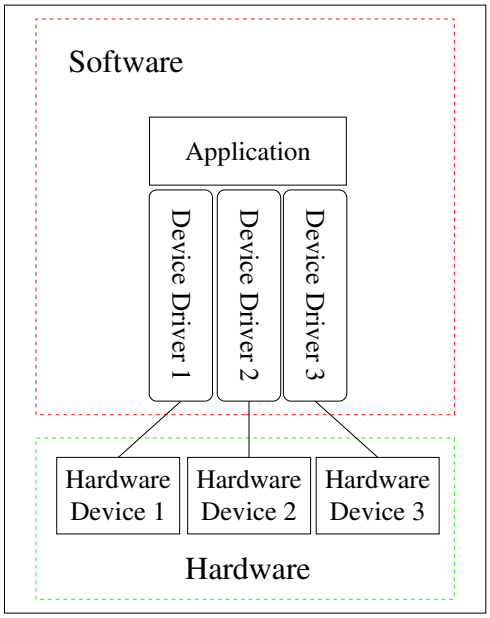

Figure 3.1. Satellite Subsystem

An ION satellite subsystem is composed of software and hardware components. This design simulates the segmentation of traditional physically independent hardware subsystems.

One example subsystem is the communications subsystem which consists of a communications application along with a device driver for the Terminal Node Controller, Radio, and Antenna. This subsystem, consisting of an application component and multiple device driver components, fulfills the final of the first tier requirements. This requirement necessitated a mechanism to return recorded data to Earth and to accept schedules of commands from Earth.

TIER 2. System software

The *system software* component of the ION satellite software attempts to solve a portion of the second tier requirements. This software provides a mechanism for providing processing time and other necessary resources to applications running on the SID. The system software is responsible for bringing the software system up to a known state upon satellite power up, starting and stopping applications as necessary, providing a limited communications mechanism for applications to affect the operations of the entire satellite, and cleanly shutting down the system upon satellite power down. The system software and any associated subsystems running on the satellite completely abstract away the details of operations of the ION satellite.

TIER 3. Supporting software

The remainder of satellite software falls into the *supporting software* component. This component fulfills the remaining software functional requirements.

Standard operating system functionality such as message passing and event logging is provided by the supporting software component. A custom file system provides a mechanism to store arbitrary data on the satellite. A collection of libraries performing functions such as ELF binary interpretation, CRC, and JPEG compression help further provide the necessary support required by the other software components.

As a whole, the entire ION satellite software system may be visualized as the hierarchical circle illustrated in Figure 3.2. Each ring in the hierarchy solves a specific set of software functional requirements. As the center of the ring is approached, a larger amount of satellite

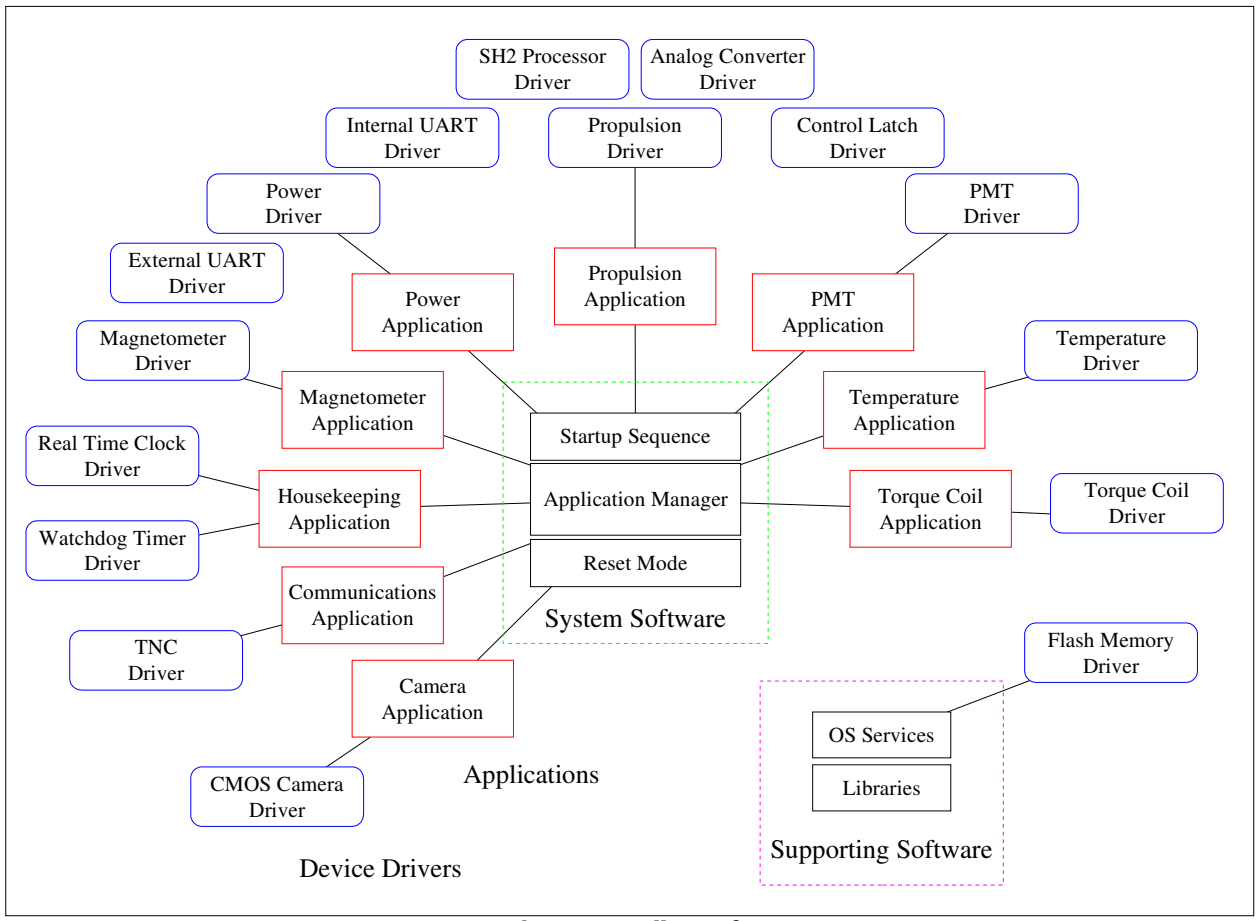

Figure 3.2. The ION Satellite Software System

The ION software may be visualized as a hierarchy of rings, each solving a specific set of software functional requirements. As the center of the ring is approached a greater amount of satellite functionality is abstracted away.

operations specifics are abstracted away. For example, the first ring abstracts away individual hardware devices through the use of device drivers. The second ring abstracts away individual subsystems through the use of applications. The third, innermost ring, abstracts away the entire satellite by presenting the appearance of a single satellite instead of multiple software subsystems executing simultaneously.

With the exception of a portion of the supporting software component, these four components are packaged into an ELF format binary file of about 950 kilobytes which is stored as an array of files on the file system of the SID. This binary is known as the *system load* and is executed by the ION satellite. Figure 3.3 illustrates the packaging of the ION satellite software.

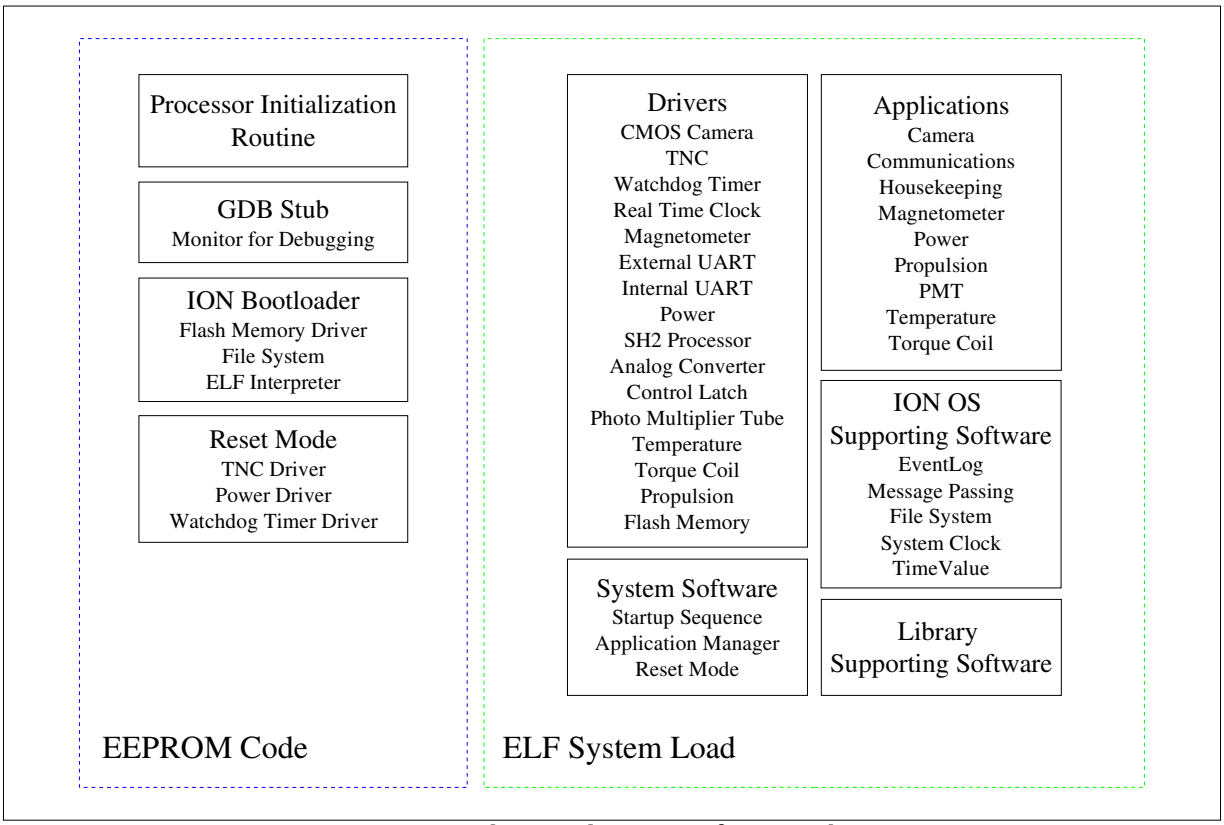

Figure 3.3. The Complete ION Software Package

The system load software is formatted as an ELF binary and stored on the file system of the SID.

3.2 Device Drivers

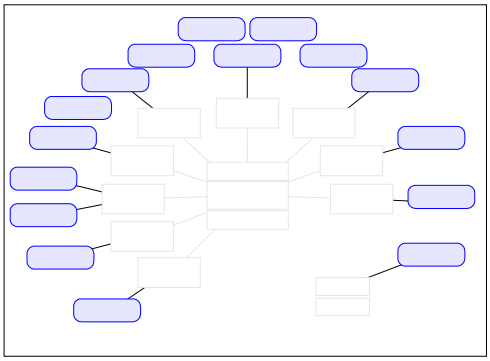

Figure 3.4. Device Driver Components

Device drivers abstract away the details of operating the hardware of the ION satellite.

3.2.1 Overview of device drivers

In order to solve the Dumb Device problem, a number of software black boxes that completely abstracted away the details of operating the hardware onboard the ION satellite were developed. This was necessary in order to not have to deal with the details of device operation when it came time to write software components that were to use the devices based upon actual mission requirements. Most every driver follows a standard design, easing the process of programming and management of development. Additionally, the black box abstraction was very robust, allowing for simple changes to software resulting from changes to hardware interfaces.

Device drivers can be considered to be the solution to part one of the first tier of the software functional requirements. That is, they provide a mechanism to handle the low level details of operating all hardware onboard the satellite. Figure 3.4 illustrates the components under discussion.

3.2.2 Introduction

Control of hardware devices is often one of the primary responsibilities of a software system. The details of control of hardware are very complicated, requiring very specific control of input and output control lines at very specific timings. In order to aid in the development of large systems, the details of such low level operations are hidden by software components known as device drivers.

Such software often abstracts operations of a hardware device and provides a very simple mechanism by which to use a piece of hardware. When it comes time to write higher level software that is responsible for operation of multiple devices, software developers can simply use device driver software to operate devices without having to worry about the details of how such operations are actually accomplished. Higher level software, termed customer software, can simply perform software function calls upon device driver software. Device drivers can therefore be considered "enabler" code which actually makes hardware devices work.

The implementation of device driver software is often very specific to how a physical device electrically interfaces to a computer. The implementation of a device driver is free to change as different hardware or interfaces are used, but the software interface presented to customer software should stay constant. This interface should strive to provide all of the features of the device in a way that is easy to use. Few limitations and assumptions on device use should be made, allowing for the device driver to be a reusable software component.

A fictional device driver is illustrated in Figure 3.5. This device driver is directly responsible for control of a single piece of hardware and presents a simple software interface for device use. All details of device operation are hidden from higher level software.

The ION software system contains 16 such device drivers, each responsible for the operation of one of more hardware components. These device drivers along with the functions they perform are listed in Table 3.1.

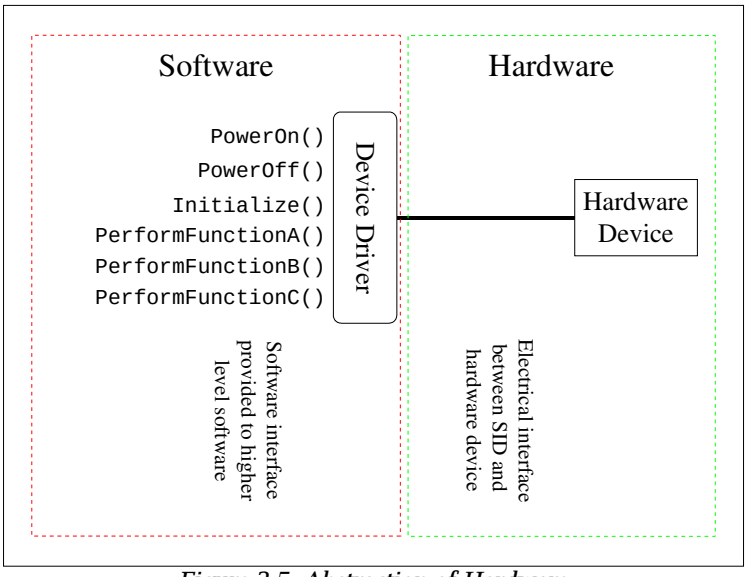

Figure 3.5. Abstraction of Hardware

An ION device driver abstracts away the details of hardware device operations and provides a simple to use software interface.

Table 3.1. Device Drivers of the ION Satellite

Driver	***Purpose***	***Functions***
CMOS Camera	Abstracts away the details of taking a picture.	- Power camera on and off - Take a single picture - Read and write register settings such as integration time and contrast - Implement DMA transfer for image data - Perform image cropping and line adjustment
Control Latch Controller	Abstracts away the details of setting and reading the control latches onboard the SID computer.	- Read and write an arbitrary bit on any of the eight control latches - Clear a latch
Analog Converter	Abstracts away the details of using the analog->digital converter on the SID.	- Perform an A->D conversion on any of the 32 analog input channels
Flash Memory	Abstracts away the details of writing and reading from the nonvolatile flash memory.	- Read, write, or erase any of the 1024 blocks on the 8 MB flash memory chip - Automatically handle flash memory chip write/erase rules - Automatically perform de-fragmentation on blocks - Allow chaining of blocks in linked-list manner
Magnetometer	Abstracts away the details of measuring the Earth's magnetic field.	- Power magnetometer on and off - Perform a three axis reading of magnetic field
Photo Multiplier Tube	Abstract away the details of measuring airglow intensity.	- Power PMT on and off - Provide ability to override automatic power shutoff - Start counting photons - Stop counting photons and return count
Pulse Propulsion Unit	Abstracts away the details of firing microvacuum arc thrusters and measuring performance.	- Fire any of four microvacuum arc thrusters - Perform a reading of the conductive deposition monitor
Power	Abstract away the details of obtaining information on the state of the power system and of setting power policy.	- Perform a reading of the state of sixteen system voltages and currents - Set the state of four output control signals controlling power system policy - Provide a communications mechanism for permission to use high power devices
Real Time Clock	Abstracts away the details of using the real time clocks onboard the SID.	- Read and write the time on any of the three hardware clocks onboard the SID
Internal Serial Bus	Abstracts away the details of using the SID serial port that is connected to other devices on the SID.	- Set serial port mode and speed - Set serial port destination - Read and write single characters - Implement the I2c protocol
External Serial Bus	Abstracts away the details of using the SID serial port that is connected to devices external to the SID.	- Set serial port mode and speed - Set serial port destination - Read and write multiple characters - Automatically perform asynchronous storage of incoming serial data
Terminal Node Controller	Abstracts away the details of communicating with the ground.	- Power the TNC and radio on and off - Configure TNC for use - Send and receive a stream of arbitrary characters to the ground - Transparently perform any necessary character conversions - Automatically perform hardware flow control with the TNC - Automatically perform transmission rate limiting - Release the radio antenna - Cache and transmit telemetry beacons
Temperature Sensor	Abstracts away the details of measuring onboard temperature.	- Perform a reading of any of the three onboard temperature sensors - Automatically configure temperature sensors - Implement Dallas Semiconductor 1-wire over UART protocol
Torque Coil	Abstracts away the details of generating magnetic fields from onboard torque coils.	- Enable the generation of a magnetic field between -2 and 2 gauss on any one of the three onboard torque coils - Generate the appropriate PWM signals to control magnetic field strength
External Watchdog Timer	Abstracts away the details of “kicking” the watchdog timer external to the SID computer.	- Kick the watchdog timer external to the SID computer
SH2 Processor	Abstracts away the details of functions provided by the SH2 CPU on the SID computer.	- Allow for writing and reading of any of the ~100 system registers on the processor - Allow installation and removal of interrupt service routines

3.2.3 Device driver design

During the development process of all of the device drivers onboard ION, six design decisions were followed. The design decisions along with factors influencing their acceptance are outlined below.

(1) Every identifiable electronic hardware component has a unique device driver responsible for the details of its operation.

Early in the design process it was identified that ION contained many "dumb" hardware devices that on their own could not perform any functions. This is in contrast to traditional satellites which often include either devices which can be considered intelligent or self-contained hardware black boxes. Nearly all hardware devices on ION are directly wired to the flight computer and, because of their unintelligent nature, rely on this computer to handle the details of their operations in order to make them work.

Many of the devices onboard ION are extremely complicated and require very specific timings and interaction. In order to hide the details of these requirements, every identifiable hardware component was abstracted with a software black box. Nearly every device on ION is controlled by a dedicated device driver. Even devices such as the processor on the SID have a software abstraction. This made device use very easy as simple function calls such as `TakePicture()` and `GetTemperature()` could be used by higher level software. This had the effect of simulating a hardware black box through the use of a software black box.

A large benefit as a result of this design decision was the ease with which software development work was delegated. As the ION satellite did not have hardware black boxes that could be given to students to develop, it was often difficult to delegate work. The use of empty software black boxes to be given to students made it possible to more effectively delegate work.

One of the largest benefits as a result of the hardware abstraction was robustness gained in the face of changing hardware interface definitions. Whenever a change in hardware occurred, it was very simple to remap the device driver to use a different port or pin on the SID and rewrite any applicable portions of a device driver without changing any of the higher level software.

There are some exceptions to this design decision which were made as a result of time constraints and convenience. The software controlling the CDM sensor is incorporated in the

microvacuum arc thruster driver instead of as a separate device driver. This makes the device driver into what could be considered a thruster subsystem driver. In similar spirit, the TNC device driver performs functions that could have been segmented into a radio driver, antenna driver, and a TNC driver.

(2) Every device driver follows a standard design.

Nearly every hardware device driver follows a standard design. All drivers are implemented as a C++ class that derives from a base driver class. When necessary, assembly code is embedded within the C code, though most drivers do not contain any assembly.

Each device driver has a set of standard calls which are inherited from the parent C++ class. These include a function to control power to the device and a function to initialize the device. It should be noted, though, that the `Initialize()` function does not actually perform any interaction with the hardware device itself and simply acts as a constructor initializing the software driver. This allows for the ability to safely call the `Initialize()` functions upon system startup without having to worry about strange system behavior as a result of misbehaving physical hardware devices.

Each device driver includes a set of function calls specific to the operations that the device performs. These are typically intuitive, based upon device characteristics. In most cases these functions are very simple to use and completely synchronous. A single call to such a function allows the device to perform a task and return any feedback. This cleanly abstracts away the details of how the device actually performs its action.

A very large benefit to using a standard design across all device drivers is that previously implemented device drivers could be used as models for students who were given an empty device driver to fill in. Device driver software interfaces could very easily be mentally pictured ahead of time and easily described to students. Furthermore, code reviews of driver software were simple because the standard design had to be understood and trusted only once.

There are some exceptions to this model. The driver for the SH2 processor on the SID consists mostly of system register definitions and support for managing interrupt service routines. The external serial port device driver is uniquely asynchronous and uses an interrupt service routine to buffer data coming into the serial port by inserting it into a queue.

(3) Device drivers may be layered.

As a result of the SID computer's design, in many instances there exists an intermediate hardware device whose use is required in order to use another hardware device. This means that in some cases higher level drivers need to make use of the functionality performed by simpler device drivers. An example of this is the Power device driver, which must make use of the analog sample functionality of the SID in order to read voltages and currents.

Therefore device drivers are layered and allowed to make calls into lower level device drivers in order to not duplicate code and to maintain the authority of a single piece of code over a hardware device. This is accomplished by passing references of required device drivers to the `Initialize()` call of higher level device drivers. It is interesting to note that this creates a dependence on the order in which device drivers are initialized.

(4) Device drivers are self contained pieces of software.

During development of device driver software it was not known what additional software would be running on the SID computer and what sorts of services would be provided by any operating system which also happened to be running on the processor. For this reason it was decided to make device drivers entirely self-contained pieces of software and make no assumptions or requirements on any operating system services or system calls which might be provided.

Therefore, if a device driver requires a piece of information such as the system time, it cannot perform a function call such as `GetTime()`. Information such as this must be provided to the device driver by the customer piece of software.

This not only resulted in self-contained device drivers, but also had the benefit of allowing device drivers to be compiled and tested as independent software loads to run on the SID during the development process.

(5) Device drivers are intelligent.

Driver software should strive to make use of hardware devices as simple as possible by the calling customer. Drivers should perform any complex functionality transparently and provide any further functionality that would aide in making customer code simpler.

ION device drivers incorporate a few features in order to accomplish this goal. Device

drivers sometimes maintain internal state and may be queried for what the state of a hardware device is. Device drivers may automatically perform functionality that would needlessly complicate the process of using a hardware device. For example, the CMOS camera device driver includes functionality to crop images and perform minimal processing on images obtained from the camera. The TNC device driver includes functionality to transparently perform soft and hard flow control, functionality to transparently perform conversion of special characters which are not correctly transmitted by the TNC hardware, and functionality to perform automatic rate throttling of transmissions. Many device drivers automatically perform hardware device initialization the first time the device is used.

As device drivers become more intelligent they also become more complex pieces of software. By trying to be as user friendly as possible and performing a large amount of functionality automatically, programming errors may be introduced into the device driver. Since device drivers are fundamental components of the ION satellite system, they need to be entirely trusted and completely understood because they must always perform as expected. Efforts were therefore made to keep device drivers as simple as possible while allowing them to be intelligent and user friendly.

One such example is found in the real time clock device driver. Three real time clocks exist on the SID computer and ideally it would be possible to use a single function call in the real time clock driver to read all three and return an average time. Yet, writing the driver in this manner would destroy the redundancy provided by the real time clocks and create the risk that none of the real time clocks could be used if a single one was malfunctioning and providing incorrect data. Instead, the real time clock device driver requires customers to explicitly specify which real time clocks are to be read and averaged.

The primary benefit obtained from the development of intelligent device drivers is that when it came time to write higher level software, most of the required functionality was already implemented. As a result, the remaining application software to be written was very simple.

(6) Every device driver is used by a single customer.

The original software design called for every device driver to be used by a single customer piece of software. This was meant to prevent multiple customers from attempting to perform hardware device operations simultaneously, which could result in interference.

Unfortunately, this decision was not sustainable and it was realized during the development process that in many cases multiple applications required the use of a common device driver. For example, as a result of limitations in interapplication communications to be outlined later, nearly every application running on the SID uses the power device driver in order to request permission to use devices which require high power consumption. Many applications also make use of the TNC device driver to update telemetry values that are included in the regularly issued radio beacon.

Although this design decision was in general unsustainable, efforts were consistently made to strongly limit the number of customers which made regular use of a device driver.

3.2.4 Discussion

The development of device drivers to enable the use of hardware devices onboard ION was one of the biggest and most challenging tasks of the ION software development process. Unfortunately, much of the work performed on device drivers is very specific to the hardware of ION and the manner in which it interfaces to the SID computer. Therefore, the majority of this effort cannot be reused by future projects.

As a result, an effort has been made to clearly describe the design decisions of the device driver component instead of its details of operation. While the decisions described above are in no way revolutionary and are in many cases intuitive and obvious, there may be value to future developers in an explicit listing of design decisions made.

3.2.5 Summary

In order to solve the Dumb Device problem, and fulfill the first of the tier 1 software requirements, a number of software components known as device drivers were developed. These software black boxes completely abstract away the details of operating the hardware onboard the ION satellite and provide many benefits such as a robustness to changing hardware interfaces and simpler delegation of work to students.

While device drivers offer the functionality necessary to use a hardware device, they themselves do not perform any activity without explicit command from pieces of software which make use of device drivers. Such customer software, known as application software, will be outlined in Section 3.4.

3.3 System Software

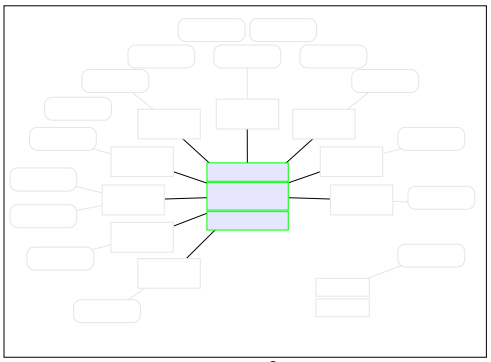

Figure 3.6. System Software Component

The system software manages system state and allocates processing resources to applications.

3.3.1 Overview of system software

The tier 2 functional requirements are now addressed. A collection of software which could be considered an operating system was developed. This software is intended to support any running subsystems on the SID computer and provides for an environment in which applications could be written.

The system software component consists of a startup sequence which reliably brings the software system up to a known state, a central intelligence which, based upon an uploaded specification, appropriately manages the allocation of resources such as processing time to different applications, and a piece of software which safely shuts the software system down and provides for a back door into the system in the event of catastrophic failure. These three pieces provide the core of the ION satellite software system. Figure 3.6 illustrates the components under discussion.

3.3.2 Introduction

To efficiently allocate the hardware resources of a computer, most software systems incorporate a collection of software known as an operating system. While modern operating systems incorporate a large amount of additional functionality, the main purpose of an operating system is to allocate processing time and memory to independently executing pieces of software often referred to as tasks or processes.

Most computer systems contain only a single processor allowing only one task to be executed at a time. In order to simulate the appearance of multiple simultaneously executing tasks, operating systems typically allocate very short discrete periods of processing time to each task in a process known as multitasking. Multitasking is performed by a piece of an operating system known as a scheduler and requires the ability to temporarily suspend and record the state of an executing task and to later resume it. The process of transitioning from the execution of one task to another is known as a context switch [14].

Multiple models of multitasking exist. One of the simplest and oldest models is known as cooperative multitasking. In this model, a running task is regularly provided as much processing time as it needs with the expectation that the task will limit itself and freely return control of the processor when a discrete unit of work has been performed. This model of multitasking was originally used in Windows 3.1 and early Macintosh operating systems [15].

3.3.3 System software organization

Similar to a traditional operating system, the functionality of the system software is to provide a mechanism for allocating processing time and other necessary resources to applications running on the SID. Additionally, this software is also responsible for bringing the software system up to a known state upon satellite power up, starting and stopping applications as necessary, providing a limited communications mechanism for applications to affect the operations of the entire satellite, and for cleanly shutting down the software system upon satellite power down. The system software plus any associated subsystems running on the satellite can be considered to completely abstract away the details of operations of the ION satellite.

Three pieces of software make up the system software component. The *Startup Sequence* is the entrance point to the entire ION software load. This software configures the basics of the software system, bringing it up to a known and trusted state. Control is handed to the *Application Manager*, the central piece of software on the ION satellite. This software appropriately provides processing time to applications running on the SID and acts as a robust version of what is typically the scheduler in an operating system. Upon system shutdown or critical error a piece of software known as *Reset Mode* is executed. This software provides a back door mechanism to access and use the the ION satellite and its hardware components directly during the reboot process. This control flow is illustrated in Figure 3.7.

3.3.4 The Startup Sequence

The entrance point to the ION satellite software system is known as the Startup Sequence. The responsibility of the Startup Sequence is to get the software system up and running into a known and usable state and then hand system control over to the Application Manager.

The Startup Sequence begins by creating an instance of every device driver available and creating a central device list. The Startup Sequence then initializes every device driver on the

system one at a time. As the initialization process does not physically use any hardware device, there is no risk of system lockup due to misbehavior of a physical device. Next, the Startup Sequence initializes the entropy pool onboard ION by performing an XOR operation over the entire 1 MB memory space of the SID. With a working source of entropy, the Startup Sequence randomly selects one of the three real time clocks onboard the SID and obtains the system time.

At this point the Startup Sequence brings up the file system. This is done by first confirming the consistency of the file system by making sure that a default set of uncorrupted config files exists on the system. If there are any problems, the file system is completely erased and a default set of config files is written out to the file system. This allows for a known system state to be entered upon startup through both allowing for maintenance of system state across reboots in the case of a well behaved system and the destruction of any accumulated state in the case of any errors or strange conditions.

With the fundamentals of the ION software system configured, the Startup Sequence

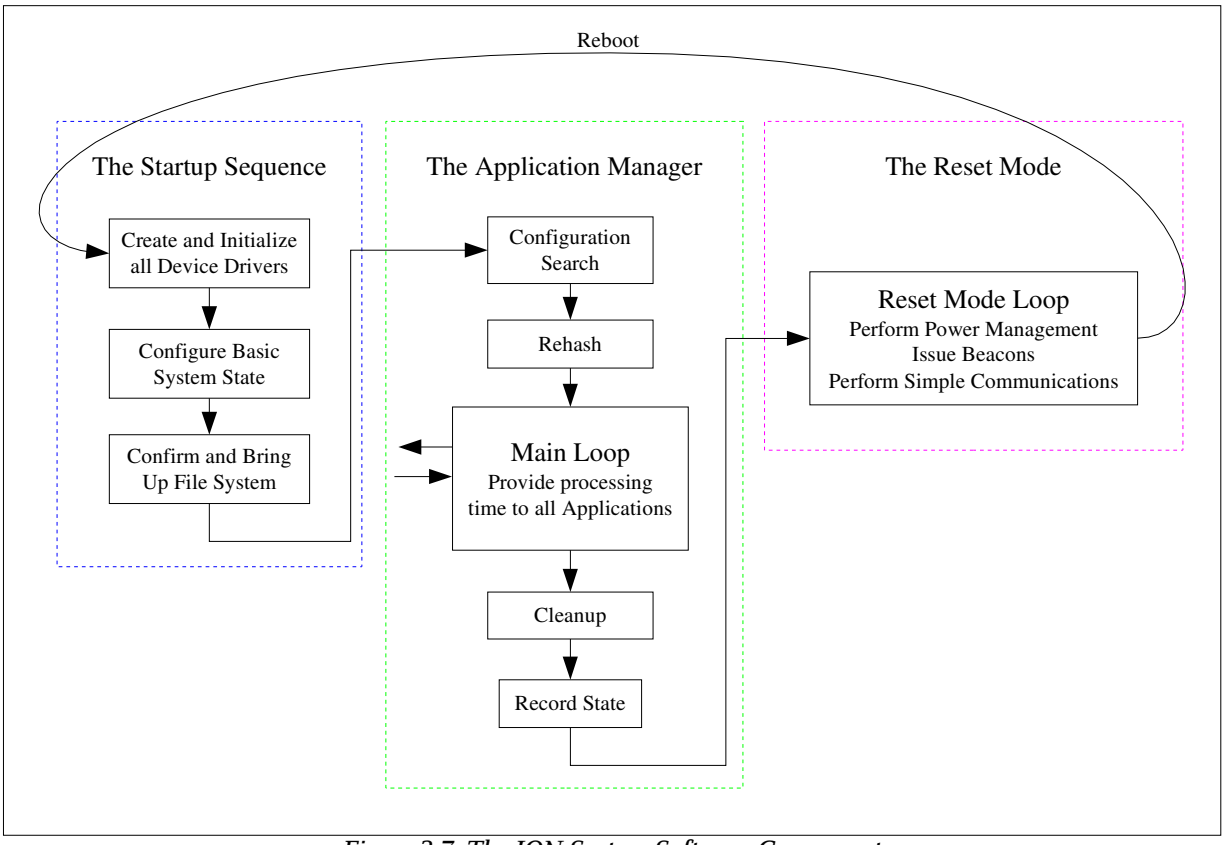

Figure 3.7. The ION System Software Component

The system software is the core of the ION satellite software and includes traditional operating system functionality such as configuration of system state and allocation of processing time to independently executing tasks.

finally enters the Application Manager. If at any point during the Startup Sequence any unrecoverable errors occur, the Startup Sequence does not transfer system control to the Application Manager and instead directly enters Reset Mode.

3.3.5 The Application Manager

The Application Manager is the central piece of software on the ION satellite, allocating processing time to all of the applications running onboard ION, and providing a view of the satellite as a single unit instead of multiple executing subsystems. This functionality requires the ability to appropriately start and stop applications as specified in configuration files provided from the ground and to perform system recovery in the case of misbehavior of any running application. As such, the Application Manager may be considered a robust and intelligent version of the scheduler that is often the core of standard computer operating systems.

Application Manager startup

Upon startup, the Application Manager searches the file system for a configuration file that details which applications should be running on the system as well as the parameters that each application should be given when it is started. Multiple configurations may be stored on the file system and the correct one is chosen based upon a search that is adjusted as a result of the manner in which the Application Manager was previously shut down.

If the Application Manager has previously shut down cleanly, then the last recently used configuration file is selected. In the case of previous shutdown as a result of system error a default safe configuration file is selected in which a minimal set of applications is specified to run.

The selected configuration file is loaded and parsed in a process known as a rehash. All information specified in this file is inserted into a database that the Application Manager maintains where it may be easily queried when needed. The Application Manager then enters its main loop.

Application Manager main loop

The majority of the Application Manager's execution time is spent in a loop in which every running application is given processing time in a round robin manner.

Upon every iteration through the main loop, the Application Manager queries its database for the state of each application. The state of each application includes information on whether the application is currently in the queue to be allocated processing time and if the application should be given processing time based upon the loaded configuration file. The state of each application is then assessed individually and one of the following actions is taken:

- If the application is not being provided CPU time and should not be, then no action is performed.
- If the application should be provided CPU time but is not, then it is allocated a new task and mailbox. The application is then started up as a new task, provided with a set of startup parameters, and given processing time.
- If the application is being provided CPU time but should not be, then it is sent a message to shut down and given processing time to shut down cleanly. The application is then marked as not running in the database.
- If the application is being provided CPU time and should continue to do so, then a message is sent to the application to continue running and a context switch is performed to allow the application to run.

Anytime a context switch occurs and an application is given processing time, it is given complete control of the SID's processor and no other software executes. Eventually, the application freely returns the CPU to the Application Manager and reports back its status as well as any other messages for the Application Manager. This mechanism of CPU allocation, known as cooperative multitasking, is illustrated in Figure 3.8.

Cooperative multitasking contains an inherent danger as an application that either misbehaves or enters an infinite loop can completely stall the entire software system by not returning the processor. The Application Manager therefore contains a recovery mechanism which detects applications that do not freely return the processor within a specified amount of time. If this occurs, a callback function is executed by a system interrupt service routine and the Application Manager resumes execution. The misbehaving application is then forcibly shut down. During the next iteration of the main loop, there will be an inconsistency in the

application's state and the application will be restarted. If an application is forcibly shut down too many times, then it will be permanently disabled.

A communications mechanism between applications and the Application Manager has been referred to. This mechanism, implemented as part of the later described supporting software component, provides the Application Manager and each application with a mailbox allowing for very simple messages to be passed between them. This allows for subsystems, as implemented by applications, to provide feedback to the Application Manager and assert some control over the operations of the entire satellite.

After returning the processor to the Application Manager, applications are required to use this communications mechanism to return a status message. Applications may signal that everything is going well, or they may request to shut themselves down and not receive future processing time. Applications may also return other limited information to the Application Manager such as system shutdown requests or requests to rehash the Application Manager using a different configuration file.

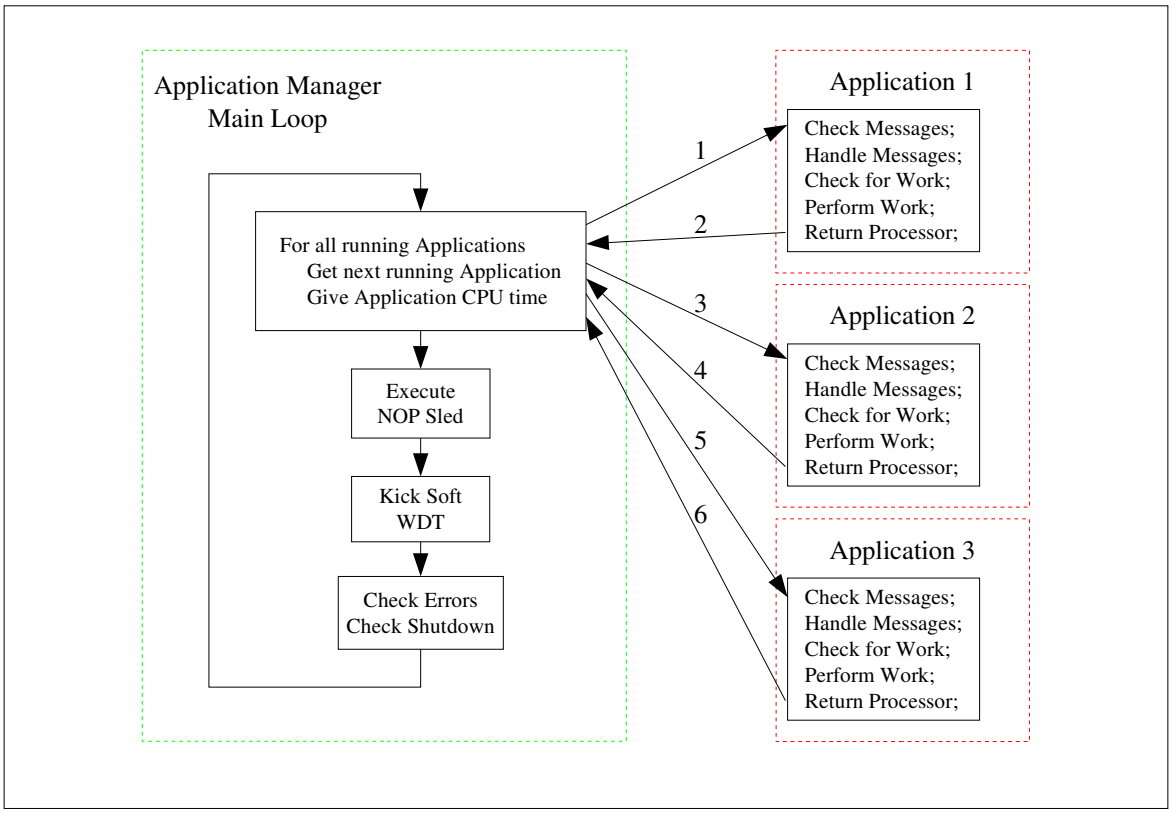

Figure 3.8. The Main Loop of the Application Manager

The main loop of the Application Manager uses cooperative multitasking to allocate processing time to independently executing applications.

Within the Application Manager's main loop, two additional small functions are performed. Upon every iteration through the loop the Application Manager kicks an internal software watchdog timer which is intended to restart the SH2 processor on the SID in the event that the system freezes. Each time through the main loop the Application Manager executes a series of 100 NOP instructions known as a NOP sled. This is a component of a rudimentary recovery mechanism which allows assembly level CPU instructions to be written in place of the NOP's from the ground. The watchdog timer and NOP sled are portions of safety mechanisms which are the result of functionality provided by software described later in this chapter.

Application Manager shutdown

Upon detection of a serious unrecoverable error or receipt of a message requesting system shutdown, the Application Manager performs shutdown and enters Reset Mode. During the shutdown process, each application is shut down and given final processing time to perform any necessary cleanup functions. Once all applications have been shut down, a file is written out to the file system which specifies whether the system is shutting down as a result of a shutdown request or as a result of serious error. This file is used on the next startup of the Application Manager to determine the appropriate configuration file to use. The Application Manager then passes system control to the Reset Mode.

3.3.6 The Reset Mode

The Reset Mode provides an alternative mode of satellite operation in which a back door into the system exists immediately prior to a system reboot. Upon system shutdown, Reset Mode is entered for a period of time ranging from one hour to one day depending upon the method of entrance and the severity of any problems encountered previous to the shutdown decision.

The SID computer executes a very simple loop in this mode of operation, which is believed to mimic the basic operation of most other CubeSat software systems [5] - [7]. In this loop, rudimentary power management is performed and every five minutes a radio beacon containing telemetry such as system status, error conditions encountered, and reasons for entrance into Reset Mode is issued.

The Reset Mode implements a very simple communications protocol which supports reads and writes of arbitrary memory locations in the SID's RAM. Function calls to arbitrary memory addresses may be performed over this protocol. This feature allows CPU assembly instructions to be directly written to memory addresses and executed if need be. This allows for system debugging from Earth and aids in performance of any necessary repairs should serious problems arise.

The Reset Mode contains no reliance on any external software, such as services provided by the tier 3 software requirements. Instead, the model of operation is one in which a few absolutely critical and completely trusted device drivers are directly used. It can be considered that the entire operational model of simulating subsystems using applications is experimental and not entirely trusted; therefore, Reset Mode provides for the recovery mechanism in the event of failure of this experiment.

3.3.7 Discussion

The use of a cooperative multitasking model to allocate processing time to applications by the Application Manager provides a number of pros and cons. The biggest benefit to this design is that the entire software system is both largely synchronous and easy to understand. No race conditions exist that need to be checked for. The whole system software can be considered to simply be polling each application to see if it requires some processing time. The entire mechanism to perform context switches and automatically send appropriate status messages has been very cleanly abstracted away as a set of two high level C functions that are used by both the Application Manager and the applications.

Unfortunately, system responsiveness is seriously affected by the cooperative multitasking mode of operation. The Application Manager's scheduling mechanism has no concept of priorities for different tasks and, as a result, is in no way real time and makes no guarantees of schedulability. Most pieces of work performed by applications have been found to take between 200 and 800 ms. With nine executing applications it is possible for some applications to run 5 s behind schedule. A mechanism to solve this problem and to help the system maintain schedulability is incorporated within the applications themselves and will be described in Section 3.4.5.

When applications are performing regular operations it is estimated that approximately four context switches per second are occurring. When system load is low and little work has been scheduled for the applications, context switches are estimated to occur at approximately 100 Hz.

The system software component of the ION software system provides the core functionality needed for satellite operations. This component provides for an environment in which applications specific to performing the tasks of subsystems can be written. The process of providing computing resources to each of these software subsystems is completely hidden and the system software prevents a view of a single satellite instead of multiple executing subsystems. Therefore, the system software, along with any associated subsystems running on the satellite, can be considered to completely abstract away the details of operations onboard the ION satellite.

The majority of this software, with the exception of Reset Mode, does not directly deal in any way with hardware devices or perform any sort of mission tasks. Instead, its primary function is to provide a framework in which the remainder of the tier one requirements may be implemented. While not all of the second tier of requirements have yet been fulfilled, the system software has gone quite a ways forward doing so.

3.3.8 Summary

The system software component of the ION satellite software system was introduced. This component, similar to a traditional computer operating system, provides an environment in which applications which implement subsystems can be written. The three main pieces of the system software bring the software system and software services up to a known state upon startup, provide a safe mechanism to allocate processing resources to applications, and provide a recovery mechanism in the event that the planned model of operation fails. In the next section, applications which provide a software simulation of hardware subsystems, the primary reason for the existence of the system software, will be discussed.

3.4 Applications

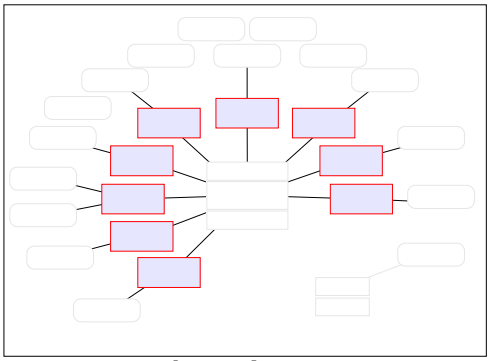

Figure 3.9. The Application Component

Applications simulate hardware subsystems and abstract away the details of subsystem operation.

3.4.1 Overview of applications

In order to fulfill the second half of the tier 1 requirements, a standard software component known as an application was developed. This component provides a software simulation of a hardware subsystem and is completely responsible for the regular use of and control of one or more hardware devices. The passive nature of ION satellite operations causes nearly all of the functionality performed by applications to consist of regularly scheduled input and output operations.

A standard set of work units was developed which allows a user on the ground to specify operations to be performed by applications at arbitrary times. Each application maintains an independent schedule of such work units which is created in memory from an uploaded file known as a config file. Figure 3.9 illustrates the software components under discussion.

3.4.2 Introduction

The majority of work performed by traditional software systems is done by programs referred to as applications. Applications, each implementing a specific set of functionality, often execute as independent, self-contained tasks. Tasks are provided time to execute on the processor of a computer system by an operating system.

A software system is said to be schedulable if a feasible schedule of allocating processing time to tasks exists. In the event that programs are overburdened with work, the software system is said to be unschedulable and will begin missing critical deadlines.

3.4.3 Application design

Having developed a collection of hardware device drivers, a mechanism was needed to appropriately make use of onboard hardware devices as specified by the second tier 1 requirement. As a result of the ill defined ION satellite mission requirements, it was clear that this mechanism would need to be very general, allowing for the performance of all possible

satellite operations at arbitrarily scheduled times and rates. This mechanism is implemented by a software component known as an application.

An application performs a small set of related functions based upon commands from the Earth. The majority of these functions consist of simple input and output operations performed by device driver software. Because the entire ION satellite could be treated as a passive collection of input and output instruments it was determined that the execution of all application operations could simply be scheduled from the ground as needed. Each running application maintains its own schedule of work to perform. This schedule is created from a configuration file, known as a config file, that is loaded and interpreted when the application is started. The use of multiple independent applications each maintaining their own schedules of work to perform is very decentralized, helping to provide the segmentation that typically exists in hardware on satellites.

In fact, each application can be considered to be a software simulation of a traditional hardware subsystem. Although applications have access to a device list specifying all of the hardware device drivers on the system, applications are typically limited to the use of only the device drivers applicable to the subsystem they simulate. To further maintain the segmentation between the simulated subsystems, no mechanism exists which allows communications or data sharing between applications. Nearly all data that an application requires for operation is obtained from the config file on the file system; any data created as a result of work performed by the application is directly written to a file on the file system.

The basic function of an application is to provide a general framework to schedule operations, so it was possible to create a generic application from which all other applications derive. This application provides the common functionality needed by all applications. Such functionality includes loading and interpretation of config files, the implementation of a scheduler to maintain schedules of work to perform, a mechanism to obtain and release processing time, a message passing interface to allow limited communications with the Application Manager, and an interface to obtain permission to use devices which draw large amounts of power.

Applications also contain functionality that is specific to the operations performed by the subsystem they implement. Such functionality can include using a device driver, writing data to the file system in a specific format, performing data compression, performing decisions such as

power management, performing system housekeeping functions, or moving packets of data to and from the ground.

All applications are implemented as statically allocated C++ classes which derives from a base application class. Each class has its own stack space allocated within the class. Nine such applications exist onboard ION. Their functions and options are shown in Table 3.2.

3.4.4 Work units, config files, and data files

Each application maintains an internal schedule of work to be performed. This work is represented by a standard piece of data known as a work unit. Work units are generated on the ground as needed and packaged into files known as config files. Config files additionally specify some global parameters for the the behavior of an application. After being uploaded to the satellite, these files are interpreted by the applications. The work units extracted from config files are inserted into each application's internal schedule of work to be easily queried in the future.

Two types of work units exist: single work units and recurring work units. A single work unit specifies the type of work to perform, options related to the work, and the time the work is to be performed. A recurring work unit specifies the work to perform, options related to the work, a time to start performing work, a time at which to stop, and an interval time which specifies the frequency at which the work is to be performed. Recurring work units allow for regularly occurring operations to be easily scheduled using only one work unit. All times used on the ION software system, included those specified in work units, specify UTC time in seconds since January 2000 representation (J2000).

Each work type is defined to be either critical or noncritical. Noncritical work types may be discarded by applications if they would be executed too late. In the event of unschedulability when the entire software system is running behind schedule, this allows the system to get back up to speed by allowing applications to drop noncritical work.

Applications which generate data record their data in well defined binary file formats. Typically the formats used are tight binary structures. With the exception of JPEG images output by the camera application, compression is not performed on data files. While communications bandwidth is sacrificed, this allows for the simple reconstruction of data files in the event of partial receipt on Earth.

Table 3.2. Applications of the ION Satellite

Application	*Rate*	*Work Types*	*Work Options*
Camera	1/day	Take 640x480 picture	Image Name
	1/day	Store camera register settings	Register number and value
	1/day	Clear camera register settings	None
Communications	12/hr	Power up communications system	Prevent power down for 15 min
	12/hr	Power down communications systems	Ignore power down prevention
	12/hr	Issue beacon	Beacon types
	1/lifetime	Release antenna	Burn time
Housekeeping	12/hr	Write event log to disk	Log level to write
	3/hr	Sync file system to disk	None
	12/hr	Set system time based on hardware clocks	Clocks to use
	N/A	Print full file system listing to event log	None
	N/A	Set system time	Time
	1/week	Request a system reboot	Reboot type
	12/hr	Kick external WDT	None
Magnetometer	N/A	Take a number of 3-axis magnetometer samples	Power on delay, number of samples
Photo Multiplier Tube	N/A	Take a number of PMT samples	Integration time, number samples, override
Propulsion	N/A	Fire a set of thrusters	Thrusters to fire
	N/A	Disable a set of thrusters	Thrusters to disable
	N/A	Take a CDM sample	None
Power	12/min	Perform a power management decision	None
	15/hr	Take a power reading for management decision use	Record sample to disk
	N/A	Manually set a power management state	Power management state
Temperature	15/hr	Take a temperature sample	Temperature sensors to use
Torque	N/A	Set the magnetic field being generated by a torque coil	Coil duty cycles

3.4.5 Application behavior

This section will outline the structure of an application.

Application startup

An application is first started by the Application Manager. Upon startup, a collection of parameters is passed into the entrance point of the application. These parameters specify a config file to use, references to mailboxes for communications, and a list of all device drivers on the system. First, the application loads and interprets the config file that was specified, in a process known as a rehash. All work units specified in the config file are inserted into the application's scheduler.

The application next performs any further initialization necessary such as opening

appropriate output files or initializing any required device drivers. The application then enters its main loop and immediately returns the processor to the Application Manager.

Application main loop

The majority of a running application's time is spent in the main loop. The main loop can be summarized as follows: receive processing time from the Application Manager, query the scheduler for any work to do, perform work if it exists, and return the processor to the Application Manager.

When an application is given processing time by the Application Manager, the application first checks its incoming mailbox for any messages. In most cases, there will be a message from the Application Manager which instructs the application to continue running as normal. There may be a message requesting that the application shut itself down or perform what is known as a rehash operation. In the case of a shutdown request, the application exits the main loop and enters the shutdown portion of the application.

A rehash request simply commands the application to read in a new config file specifying a new schedule of work to perform. A rehash request is always the result of a command received from the ground that has been propagated through the Application Manager. Rehash is accomplished by returning to the Startup section of the application and loading a new config file.

Generally there will be no shutdown or rehash messages in the application's mailbox and the application will be free to continue with normal main loop operations. In this case, the application queries its internal scheduler for any work that needs to be performed. Based upon the current system time, the scheduler may return a piece of work to be performed. If a work unit is returned, the application performs the work specified and then returns to the beginning of the main loop where the CPU is returned to the Application Manager. If no work is returned or the work unit returned is of a non-critical type and has long since expired as a result of the software system running behind schedule, no work is performed and the application immediately returns to the beginning of the main loop.

In the process of returning the CPU to the Application Manager, a message is sent to the Application Manager specifying that everything is running normally and that the application is freely returning the CPU. In the event of any problems or critical errors, applications may instead choose to send a message which asks the Application Manager to shut the application

down. Some applications may return other limited information or requests to the Application Manager, allowing subsystems to provide feedback to the Application Manager and influence aspects of operations of the entire satellite.

Application shutdown

If a critical error is encountered or a shutdown message is received from the Application Manager during execution of the main loop, the application exits the Main Loop and shuts itself down. This is accomplished by closing any open files, performing any necessary clean up operations, and clearing the internal scheduler. The application then permanently returns the CPU to the Application Manager. A message specifying that the application has shut itself down and does not want anymore processing time is sent to the Application Manager. A flow diagram of application behavior is presented in Figure 3.10.

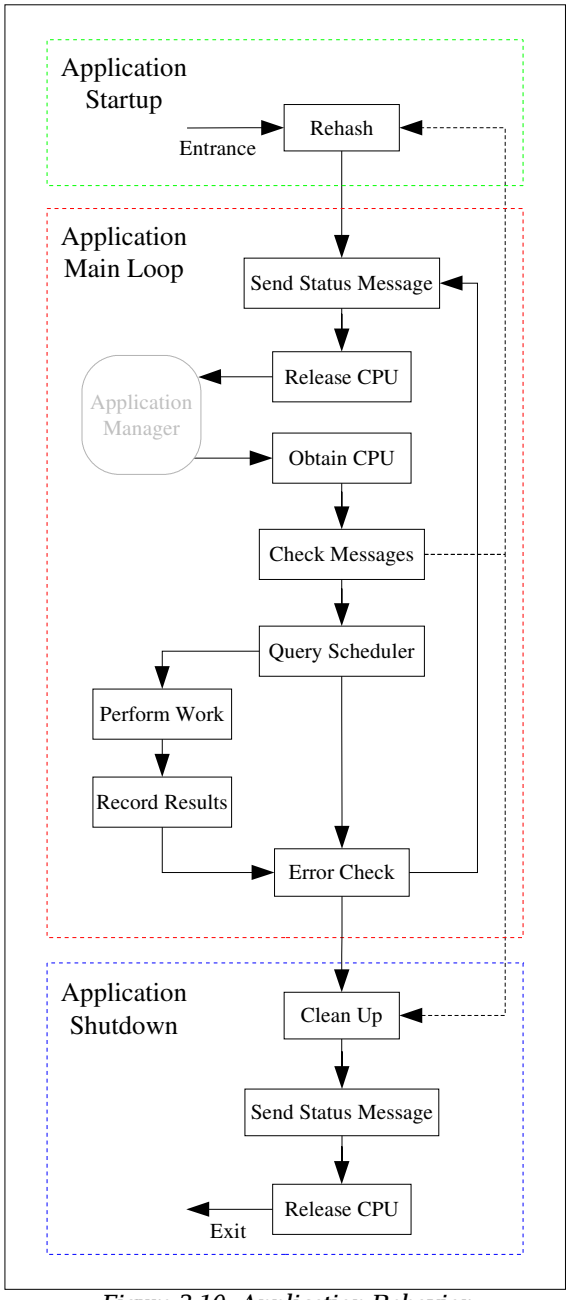

Figure 3.10. Application Behavior

Applications, which execute as independent tasks, simulate hardware subsystems by performing operations scheduled from the ground.

3.4.6 Discussion

The primary task of an application is to provide a mechanism to perform a small set of operations based upon an uploaded schedule. With the exception of the details of performing an operation, all applications are generally identical, allowing for the use of a single common application format coupled with the use of software inheritance. The implementation of all of the applications was greatly eased as the majority of application functionality was automatically provided through inheritance. Applications were further simplified because device drivers tend to be intelligent pieces of software which perform

all of the details of hardware device operation automatically. Development of specific applications simply focused on the appropriate use of a device driver and the proper recording of any data as a result of device driver use.

One of the largest benefits stemming from the use of generic scheduling mechanisms to implement subsystems of the ION satellite is the ability to adapt to a changing mission specification. As the mission requirements of the satellite were developed independently by the software development team, concerns existed that incorrect assumptions on satellite operational requirements may have been made. By implementing a general way to use the full functionality of all onboard hardware, this concern was alleviated because the exact details of mission requirements could be determined at a future time.

Applications force the use of the ION satellite as a passive device which performs only scheduled events, making it possible to completely understand and characterize the behavior of the satellite. No operation is performed onboard ION unless explicitly scheduled by a user on Earth. Accurate simulations of satellite behavior may, therefore, be performed on Earth, before sequences of operations are uploaded.

One of the general design philosophies of application behavior is that each application would perform one complete work unit every time processing time was allocated by the Application Manager. Although this provided for a very nice synchronous system, some operations performed by applications require large amounts of time to execute, resulting in very poor system responsiveness. Such operations must be must be broken up across multiple context switches.

An example of this can be found in the PMT application. One work type the PMT application can perform is to take forty 10-s PMT samples. The execution of this example work unit requires at least 400 s with the great majority of this time spent busy-waiting. If the PMT application performed this entire work unit before returning the processor to the Application Manager, then the entire ION software system would be over 6 min behind schedule by the time execution was finished. In order to prevent this, the CPU is regularly returned to the Application Manager during the execution of this work unit, allowing for other applications to execute. Although some asynchronism is introduced to the software system and the model of application behavior presented is violated, no other solution which aids system responsiveness has been found.

3.4.7 Summary

In order to allow the arbitrarily scheduled use of device drivers onboard the ION satellite and fullfill the second tier 1 software requirement, a software component known as an application was developed. This component provides a simple way to schedule any operation the ION satellite performs. Furthermore, applications provide a segmentation of satellite functionality by simulating what are traditionally hardware satellite subsystems.

3.5 Supporting Software

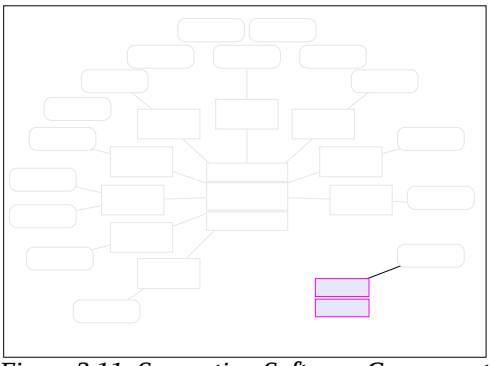

Figure 3.11. Supporting Software Component

The supporting software implements necessary functionality such as operating system services and standard library functions.

3.5.1 Overview of supporting software

The supporting software component of the ION satellite software system includes a wide variety of software which implements necessary, though rather uninteresting, functionality. Typically, this software fulfills the requirements of the third tier of software functional requirements, providing software support for the implementation of tier 1 and tier 2 solutions. Figure 3.11 illustrates the components under discussion.

3.5.2 Introduction

One of the guiding philosophies behind software development is the use of abstraction and segmentation. Segmentation of problems into individual pieces, each with a solution provided by a piece of software abstracting away the details of implementation, allows software developers to build large complex systems. Software which solves general problems is often written in a reusable manner and packaged into software components known as libraries which may easily be incorporated into multiple software projects.

Modern operating systems perform many services in addition to allocation of resources. Operating system services may include a file system for data storage, a message passing interface allowing independent tasks to share information, and error logging facilities.

Computer software to be executed is often stored on a file system in binary form. This software must be copied into memory before it may be executed. A bootloader, commonly

permanently written into read only memory of an embedded system, is the piece of software which loads an operating system into memory for execution.

3.5.3 Explanation and justification of supporting software

To support the previously detailed software components, much supporting software needed to be written or obtained. This software implements very common functionality, which is often necessary for the development higher level software. A large amount of this functionality may be found as freely available COTS-like software libraries which can simply be incorporated into a software project.

As a result of stubbornness of the ION software developers and of limitations due to the proprietary nature of the SID flight computer platform, a large portion of this requisite software was rewritten for the ION satellite. While this effort could be considered an inefficient allocation of resources, some benefit was obtained from it. Nearly all of the software code that the ION satellite executes has been written by the ION software developers. This allows for complete knowledge and trust of nearly every single instruction the SID computer executes; practically no untrusted or unnecessary code is ever executed onboard the ION satellite.

During the design process, with the exception of a file system, very little was known about the details of what functionality the supporting software component would need to provide. Therefore, very few initial design decisions were made and the supporting software was designed and developed as it was needed.

Towards the end of the development process it was realized that the supporting software that had been written could be organized into three main areas: software which provided modern operating system services, software which implemented common functionality found in standard libraries, and software, flashed into the EEPROM of the SID's processor, providing mainly boot loading and debugging functionality.

3.5.4 ION OS Supporting Software

The *ION OS Supporting Software* provides functionality that is typically found in modern day operating systems. This software is loosely considered to be what could be termed the ION operating system.

EventLog

The *EventLog* provides a central mechanism for logging of all activity that occurs on the ION satellite. Any piece of software may insert a text message into the EventLog. All entries are automatically flagged with a time stamp and a priority level. EventLog messages can easily be filtered based upon priority levels allowing for simple differentiation between critical errors and debugging commentary.

The EventLog is stored entirely in statically allocated memory instead of on the file system so that there exists no risk of generating additional log entries during insertion of an entry. The housekeeping application occasionally traverses the EventLog and records interesting entries to the file system where they may be downloaded.

Message passing interface

A message passing interface which provides mailboxes to applications and the Application Manager was implemented. This system allows the queuing of messages and provides for limited communications between applications and the Application Manager.

System clock

The ION software system includes a collection of functions to read and write the system time as well as an interrupt service routine which is executed every 10 ms. This software is also responsible for regularly kicking the hardware watchdog timer of the SH2 processor and implements a soft watchdog timer which must be occasionally kicked by the Application Manager. The system clock includes support for scheduling alarms and the installation of callback functions to execute at specified times.

TimeValues

A system wide standard method of representation of dates and times known as *TimeValues* was developed for the ION satellite. All times on the system are UTC and represented using seconds since January 2000 with a resolution of 10 ms. The TimeValue software also provides a large collection of arithmetic operations such as addition, subtraction, and comparison that can be performed on dates and times.

File system

A custom file system was written to allow storage of general data on the flash memory chip. The file system provides standard functions such as read, open, append, set position, get position, delete, close, and directory list.

The file system is a flat file system containing no directories and is limited to 128 files and filenames of 12 characters. The implementation makes use of a file allocation table (FAT) which specifies the names, sizes, and locations of files on the flash memory chip. Blocks which contain a contiguous file are chained together as a linked list in portions of the blocks reserved for extended data.

Due to limitations on the speed and number of write operations available, the FAT is typically kept in memory and is only synchronized to the file system upon close of a file or an explicit sync operation. Caching of file system blocks during read operations is performed to greatly improve speed. Only the append writing method is supported as it was found that the application software would not need to write to arbitrary locations in files.

3.5.5 Library Supporting Software

The *Library Supporting Software* implements common functionality typically found in standard libraries. This software is inspired by the standard library implementations and strives to provide functionality in a generic, reusable, and efficient manner.

Example functionality includes implementations of queues, `itoa()`, `snprintf()`, CRC, JPEG compression, an ELF binary interpreter, and endian conversion.

3.5.6 EEPROM Supporting Software

The *EEPROM Supporting Software* consists of a collection of software that is flashed into the EEPROM of the SID's processor and not part of the system load. This software functions mainly to load the satellite software from the file system and into memory where it can be executed. This software is segmented into four sections: processor initialization code, a debugging monitor, a bootloader, and Reset Mode.

Processor Initialization

This is the first software that is executed when the SH2 processor is powered. It consists

of a combination of assembly and low level C code. The primary purpose of this software is to initialize system registers, enable the external RAM on the SID, and configure an initial stack.

GDB Stub

This is the monitor that allows for the debugging of software running on the SID. The GDB Stub software can only be entered before launch, when the satellite software is being debugged [16], [17].

ION Bootloader

If the monitor on the SID is not entered than the ION Bootloader software is executed. This software loads the ION satellite software into memory for execution.

The ION satellite software is stored as an ELF format binary which is broken up into multiple files on the file system. Upon startup of the satellite, the ION Bootloader software searches the file system for a software load to execute. Once located and checked for corruption, the software load is copied into memory and executed.

New ELF binaries may be uploaded to the satellite's file system, and if appropriately named, will be located and used by the ION Bootloader.

Reset Mode

An implementation of Reset Mode, identical to the version that is in the system software, along with requisite device drivers is flashed into the EEPROM of the SH2 processor. This allows the ION satellite to continue to run in a limited manner in the event of complete failure of the flash memory chip or corruption of the binary file containing the ION software load.

3.5.7 Discussion

A large portion of the supporting software components are often available as prewritten software. The ION software developers chose to rewrite a large portion of this software in order to be able to completely trust the software executing on the SID computer. This required a greater amount of time and effort than predicted. In hindsight, with the exception of the EEPROM software, little value was obtained by rewriting this software.

Time spent developing and testing custom implementations could instead have been used

to thoroughly test already existing implementations. If possible, it is strongly suggested that future developers attempt to use already existing software standards.

3.5.8 Summary

The supporting software component of the ION satellite software system serves to fulfill the third tier of software requirements. This software provides standard functionality often required by higher level software such as the application and Application Manager components. The majority of the functionality provided by the supporting software can be obtained through the use of standard libraries or COTS-like pre written software packages.

3.6 Discussion of the System Design

The design of the ION Software System is the result of pressures associated both with the hardware onboard the satellite and the poorly defined mission requirements. The entire ION software can be considered a mechanism which allows the remote operation of a passive collection of instruments onboard the ION satellite. The software itself makes very few assumptions on how instruments are to be used, therefore allowing for the details of mission requirements to be adjusted and invented as needed.

Both the differing functionality and the levels of functionality required of the software system have been segmented through the use of application and device driver components. Such segmentation allowed for relatively simple implementation of all of the functionality that was expected of the ION satellite.

A number of limitations arise from the software design used, most of which result from the use of multiple independent applications to implement the majority of the satellite operations. There is no central oversight of system state because applications do not communicate how much work they are performing or what the immediate future looks like. For example, it is not possible to put the CPU to sleep for a few minutes in order to conserve power. Each application must continuously be polled by the Application Manager in the event that there may be work to perform. The risk that the system can become unschedulable exists, as there are no limitations placed on the amount of work that applications may perform. It is entirely the responsibility of the user on Earth create schedules that are possible to fulfill.

These two limitations can be solved if additional features are added to the ION software

system. Central oversight of system load could exist if applications provide additional feedback, such as how full their schedule is and how much work they expect to perform in the next few minutes, when returning the processor to the Application Manager. This would allow for more intelligent allocation of processing resources and the ability to power down the flight computer when no applications expect to perform work.

To limit the possibility of over scheduling the ION software system, some form of credits could be used to allocate processing time to applications. A set total of credits could be available to each application for each minute of operation. Upon consumption of all processing credits, an application would not receive anymore processing time until new credits were granted.

Another large problem stemming from the use of independent applications to implement subsystems, is the lack of support for data sharing between subsystems. For example, no single piece of software has access to both magnetometer samples and solar cell current readings. This makes the implementation of an autonomous process such as onboard attitude control impossible. This problem could be solved with a more robust communications system which allows for communication between applications. Such a feature, though, may run the risk of destroying the segmentation that exists between subsystems. Furthermore, one of the fundamental assumptions made in regards to ION operations is that the satellite would be considered a passive collection of instruments whose use was to be scheduled. Intelligent, fully autonomous processes were not expected to be supported by the ION software system.

Nearly every operation executed by the ION software system must be scheduled and performed by an application. Even tasks which are semi-autonomous such as power management or file system synchronization have been explicitly designed as operations which must be scheduled. While this aids in the understanding of system performance and behavior, it does make software development more difficult. In many cases software operations could have easily been written to occur automatically every time an application received processing time. Instead, in order to maintain the passive system design of the satellite, these operations were, with additional effort, implemented as scheduled items.

One of the largest problems encountered during the implementation of the ION satellite software system was testing. While testing of individual device drivers was very simple, the original software functional requirements determined before system design did not specify any requirements for the ability to test the complete software system. As a result, no mechanism was

designed which allowed for quick, easy, and transparent operational testing. Therefore, all satellite operations had to be tested exactly as if they were being performed from a ground station on Earth.

This was a very slow process as it required the creation and upload of config files followed by a wait for the scheduled events to occur. It was not easily possible to see what occurred while the system was running because system information can only be obtained from files on the file system. The EventLog as well as any applicable data files had to be downloaded in order to piece together the overall state of the system. This problem could have easily been prevented if the original software functional requirements specified that the software system be more transparent and easily testable.

Despite these frustrations, the software system that was designed and implemented for the ION space satellite provides an amazing abstraction of the satellite, allows the very general use of all hardware onboard, and provides flexibility that would otherwise not be available. While the model of operations planned has not yet been "space rated" it is expected that there will be few difficulties or unexpected limitations encountered during the lifetime of the ION satellite.

3.7 Summary of the System Design

The main components of the ION satellite software were outlined along with the problems they solved. These four components were the direct result of the hardware design of the ION satellite along with the ill defined mission specification.

The Dumb Device problem led to the development of a collection of modules known as device drivers which abstracted away individual devices and performed the detailed functions required for hardware device operation.

As a result of a relatively undefined mission specification given to the software development team and the determination that the satellite could be considered a passive collection of instruments, a set of independent modules known as applications were written. These applications simulate what are normally hardware subsystems, are not able to communicate with one another, and typically perform functions based upon a modifiable schedule.

In order to manage operations of all subsystems and present the appearance of a single satellite instead of a collection of independent subsystems, a software component known as the

system software was written. This software is responsible for providing processing time to applications and managing satellite start up and shut down.

Finally, a large amount of software which provided basic functional support commonly found in libraries was written. This software was classified into a component known as supporting software.

APPENDIX A: FURTHER TECHNICAL DETAILS

Further technical details relating to the implementation of the ION software system will be presented. The majority of these details are expected to be only of value to future satellite software developers. An overview of all software reliability and safety mechanisms will be provided along with details of the development process.

A.1 Software Reliability and Safety Features

A number of reliability and safety mechanisms have been mentioned in previous chapters. These features will be summarized here.

A.1.1 Remote memory access

The communications protocols used by the ION satellite provide support to read and write from arbitrary memory addresses of the SID. By maintaining a copy of the flight software load binary with symbols on the ground, it is possible to read or write any software variable that is statically allocated in memory.

A.1.2 NOP sleds

Portions of the ION software system routinely execute large contiguous blocks of NOP instructions. It is possible for assembly instructions to be written over these NOP instructions, directly into the path of execution, through the use of the previously mentioned ability to write information to arbitrary memory locations from the ground. These instructions may perform any small software fixes necessary or even make function calls to other locations in the flight software binary, allowing for small, temporary changes to the satellite software to be made from Earth.

A.1.3 Software upload

Since all of the ION software system is stored as an ELF format binary on the file system it is possible to upload new versions of software to the satellite. If appropriately named, this software will be loaded for execution by the ION bootloader instead of the default system load.

Unfortunately, a full system load is very large and a full replacement of the software onboard ION would take approximately two weeks. It is possible to only replace individual

binary files because the software load onboard the file system is broken up across a number of small files. This process would need to be done very carefully as no changes could be made to the resulting memory locations of any code or variables which followed the changes.

A.1.4 System state checking

Three mechanisms exist to make sure that system state upon startup is safe. The bootloader first looks for a valid system load that has been recently uploaded. If no such load exists or if it is corrupted, the bootloader reverts to the default system binary which exists on a separate file system.

Upon startup the system software checks the consistency of the main file system by confirming a default set of config files. If there are any problems with these files, the entire file system is recreated and a new set of default config files is written out.

Upon startup of the Application Manager, a file on the file system is read to determine the reason for the last shut down. If no strange conditions caused the previous shutdown, then the Application Manager restores the running state of all applications to before the previous shutdown. If, on the other hand, this file does not exist or an error was the cause of the previous shutdown, then the Application Manager starts only a minimal set of applications specified by a default configuration.

A.1.5 Software watchdog timer

In addition to the hardware watchdog timer functionality present onboard ION, a software watchdog timer is provided by the ION software system. The hardware watchdog timer of the SH2 processor must be kicked approximately 50 times a second. This is performed by the system timer interrupt service routine. In the event that the software system seriously misbehaves, this service routine will not execute and the hardware watchdog timer will restart the SID's processor.

Unfortunately, it is possible for the system timer service routine to continue firing even though the software system may have entered an infinite loop or be otherwise broken. Therefore, a soft watchdog timer exists which must be kicked once a minute. If this is not performed, then the system timer interrupt service routine will not kick the processor's hardware watchdog timer and the processor will be restarted.

A.1.6 Alarms and callbacks

One of the risks associated with cooperative multitasking is that applications may lock up and not return control of the processor to the mechanism which allocates processing time. A system of alarm callbacks was implemented to handle the possibility of application misbehavior as a result of coding mistakes or hardware failure.

This mechanism allows the Application Manager to install a callback function to execute one minute after providing processing time to an application. In the event that the application does not return the processor within one minute, this callback will perform a context switch back to the Application Manager which may then shut down the misbehaving application. For more information refer to Section 3.3.5.

A.1.7 Reset Mode

Normal ION satellite operations are performed by uploading schedules of work to the satellite. The work specified in these schedules is executed by a number of software applications which make use of device drivers to operate hardware devices. Feedback as a result of work performed is returned to Earth as downloadable data files.

A completely different mode of satellite operations known as Reset Mode also exists. This mode is entered upon any critical software error or prior to a satellite reboot. In this mode of operation a simple communications protocol allows for the direct use of hardware devices on the ION satellite. For more information refer to Section 3.3.6.

A.1.8 System wide error reporting

A standard set of error codes exists in the ION software system. Nearly every major function returns an `ion_error` variable which specifies any errors that may have occurred. Error codes are very specific and allow for immediate determination of where and why and error occurred. These errors codes are propagated throughout the entire software system and are inserted into the EventLog, the beacon, communications protocol replies, and Reset Mode, if necessary. This allows for multiple data paths to be used to provide specific feedback on any errors that have occurred.

A.2 Details of Development

All development of the ION satellite software was performed in a Linux environment using the GNU Binutils [18]. GDB was used to remotely debug code executing on the SID computer through the use of the GDBStub monitor in the SID's EEPROM [16], [17]. CVS was used to manage all code developed for the satellite [19]. Regrettably no bug tracking system other than paper was used.

A simple, though extremely long, makefile, explicitly listing every code file, was used to guide compilation of various portions of the ION software. A custom written linker script was used to appropriately arrange binary code layout during the linking and relocation process.

The majority of all software was written in C++ or C, though it is based heavily upon C style coding. Nearly every individual component of the software system is implemented as a C++ class though each component is typically treated as a C module as opposed to a C++ object. Where applicable C++ inheritance was used.

No generic memory management support existed or was written; as a result dynamic memory was not generally available. Because `new()` and `malloc()` were not available all variables are either statically allocated or allocated at run time on the stack. Lack of dynamic memory also limited the number of C++ features which were usable. All variable types used have been custom defined in order to aid in understanding of variable sizes and representations. Names of variables are inspired by Hungarian notation. For safety, whenever possible, efforts were made to write only synchronous code in order to limit the possibility of race conditions. Very few interrupts are used and all context switches are performed in a controlled and explicit manner.

Software development was usually performed by two person teams. One developer would be be responsible for inventing, verbalizing, and freely coding ideas. The other developer would provide feedback and observe to catch trivial errors in the details of implementation. Code reviews were required prior to commitment of code to CVS.

Very little of the code used in the ION software system was not developed by the ION software developers. To the best of the abilities of developers, efforts were consistently made to understand every single line of code the SID computer would execute. This self imposed elitist development philosophy required all software including bootloaders, operating systems, file systems, application software, device drivers, and libraries to be written in house.

During the development process, immediate though unclean solutions, would be found to problems faced. These hacks often required the creation of special cases or resulted in a violation of some intuitive, though unverbalized, design. Such solutions were never taken. Instead, problems would often be struggled with for days or even weeks despite the existence of quick solutions. While this slowed the initial development process, such policy is highly recommended as it greatly eases the later testing and maintenance of software.

Overall a combination of bottom-up and top-down development was used. As traced in Chapter 3, low level device drivers were first developed. This was followed by high level system software which was expected to provide processing time to some middle layer of application software. Supporting software was written as needed. The implementation of the ION software system finished with the quick and easy implementation of applications.

A snapshot of the source code to the software installed onto the ION satellite is available on the Internet [20].

APPENDIX B: LESSONS LEARNED

B.1 Difficulties Encountered

During the development process of the ION software a number of difficulties were encountered. Some of these were general difficulties which are common to CubeSat projects whereas others were specific to the project management or technical details of the ION project. The difficulties encountered are listed below along with suggestions for future developers. It is hoped that a documentation of the difficulties encountered may be helpful in allowing future academic small satellite projects to plan for and possibly prevent such difficulties.

B.1.1 Inheritance of project with no mission definition

The original design and selection of the hardware components onboard the ION satellite was performed by a group of students who graduated one and half years into the project. At that point the entire project was restaffed with new teaching assistants and new students who had no knowledge of the formal mission requirements of the satellite.

As no formal mission specification detailing the specific operational requirements of the ION satellite existed for the ION project, the new developers were forced to reverse engineer the mission requirements of the satellite. As a result of informal discussion with faculty mentors and assessment of onboard hardware, the software development team managed to invent what was believed to be the satellite mission. This mission both guided and forced the implementation of a very general software system.

The decision on what operations a space satellite should perform should not be left to the invention of the software team despite the fun and power associated with such work. The lack of such a specification gives software developers no direction on what to write, and its invention is a distraction from the actual task of implementation of satellite functionality. Furthermore, when satellite operational requirements are determined in this manner no oversight exists.

In addition to the obvious recommendation that a three to four year project not be completely restaffed halfway through, the formal mission requirements of a satellite should be very clearly documented at the start of the project. In addition to defining every operation a satellite is to perform, this documentation should also include all data that is to be sampled, requirements for sample resolution, and justifications for hardware design choices.

B.1.2 Regular student turnaround

The ION project was run as an engineering class in which students participated from one to two semesters making student turnaround very large. This required constant retraining of new students in order to bring them up to speed. In many cases, because students would be leaving soon, it was very tempting to simply give students a small project to work on without giving them an overview of the whole scope of the project.

While this saves time because effort is not wasted on teaching students details of the satellite they will never encounter, this policy seriously limits the number of people involved in the project who have any sort of global view. Despite the temptation to limit student involvement to small manageable components, it is important to take the time to fully entrench some of the more promising students. At least a few students or teaching assistants should be involved who have been working on the project for longer periods of time and can provide continuity across new semesters.

Student turnaround is inevitable in undergraduate projects, making it important to have good, up to date primer documents which can be given to students. These primers should give both a global overview of the project including history, as well as specific technical details of the smaller projects students will be working on.

B.1.3 Use of a single central computer and "dumb devices"

The hardware design of ION made delegation of work to students very difficult. Because there was only one flight computer onboard ION and it was completely responsible for the operations of all of the hardware onboard ION, it was not possible to provide students with hardware subsystems to independently develop according to some physical interface. Instead, all development occurred on the single flight model of the SID computer which caused a large number of scheduling conflicts.

This problem was partially eased as a result of the subsystem simulations provided by the software system. It is nevertheless highly suggested that future CubeSat satellites attempt to use a more traditional segmented hardware design in which intelligent subsystems are used.

Such subsystems should each perform their own processing and data management using whatever processing capabilities are required. A digital communications bus could be used by a

central flight computer to communicate with these subsystems allowing the central computer to request data or poll status. With clever software development, it may be possible to allow the central flight computer to upload software updates to each of the individual subsystems, maintaining the code update benefits that the use of a single computer provides.

B.1.4 Difficulties due to nonstandard hardware

ION's flight computer, the SID, is a unique computer of which only approximately a dozen exist. This computer was designed by an engineer who provided a full schematic but otherwise little documentation of the type that was needed by undergraduate engineering students. There existed no community on the Internet which could provide help and support as there are with many other types of embedded processors and hardware.

The proprietary nature of the SID prevented the easy adoption of standard software and, therefore, a large amount of pre-existing software such as operating systems, device drivers, and file sytems was needlessly rewritten. Furthermore as a result of the relatively sparse documentation of this proprietary hardware an enormous amount of time was spent reverse engineering the design decisions and operation of the SID. For example, over three weeks were spent on development of software to make the real time clocks onboard the SID work reliably.

In light of these difficulties, it is strongly recommended that small satellite projects continue to follow the philosophy of using COTS components and standardized platforms.

B.1.5 Difficulties in getting hardware working

An enormous amount of development effort of the ION software system was devoted to the creation of device drivers to use the hardware onboard the ION satellite. As previously mentioned, it is recommended that standard, easy to use hardware components with existing device driver implementations be used to limit such efforts.

If this is not a possibility, it is critical that the software development team have access to good electrical engineering tools, knowledge, and experience. Access to logic analyzers and power supplies, familiarity with electrical engineering, ability to interpret data sheets, an ability to speak with engineers, and experience implementing specifications are all critical requirements for a software development team.

B.1.6 Lack of embedded development experience

Software development on embedded systems requires practical knowledge that is rarely taught in computer science curricula. The majority of incoming student software developers on the ION project had previously had no embedded software development experience. Furthermore, no access to faculty mentors with embedded experience existed and few faculty approached at the University of Illinois had any interest in taking time to teach students.

This created a very steep learning curve for both teaching assistants and students as fundamental concepts of embedded development had to be independently learned and reinvented. It is extremely important that software developers have embedded systems development experience or are provided with resources or basic instruction on the subject.

B.1.7 Changing development timeline

The delivery and launch dates of the ION satellite were delayed over five times granting an additional one to four months of development and testing time after each delay. At nearly every point in the development of the ION satellite, a launch date loomed approximately three to four months away. This regularly forced hardware and software design decisions which supported quick, simple, and understandable implementations but provided both limited functionality and limited redundancy.

With one month to two months remaining before launch, the launch date would be delayed, granting additional time. The state of the software system would then be reevaluated and a wishlist of features would be accepted for inclusion into the system. Inclusion of these new features into the software system and the general improvement of the system often required a great deal more effort than what would have been required if such work had originally been performed. This iterative system design process resulted in a large amount of additional work which could have been prevented if better knowledge of project duration had been available.

Similar small satellite projects should make serious efforts to have accurate information about the duration of the project and about expected launch date changes. While launch dates are typically not under the control of small satellite developers, better information should be provided within the CubeSat program so that developers may more effectively plan the development process.

Serious consideration should be given to a development process in which the duration of

the project is absolutely determined ahead of time, the project is implemented according to this schedule, and then obtainment of a launch is considered.

B.1.8 Bad interface definitions

It was not until two and a half years into the ION project that a formal set of documents detailing the electrical interfaces between the central computer and the hardware components existed. Previous to that point all software development of device drivers required that software developers guess at interfaces and regularly harass hardware developers for information both on how a hardware component worked and for information on any assumptions that hardware developers had made.

Additionally, as hardware development continued, the lack of well defined electrical interface specifications allowed hardware developers to freely make changes to device interfaces, requiring software developers to regularly reimplement portions of the software system. While this process was eased as a result of the abstraction device drivers provided, this still required a great deal of time and effort.

While it is understandable that at the beginning of a project is it not clear how all of the hardware components operate, it is important to create a single document specifying electrical interfaces as early as possible. This document should clearly explain how all hardware components operate, hardware pinouts, what functions will be expected of the software system, and all assumptions that are made by both software or hardware teams. In the ideal case that standard buses such as I^2c or USB are used to communicate between intelligent hardware components, this interface needs to specify only the logical communications protocol.

B.1.9 Duration and scope of project

The ION satellite project was very ambitious and far too complicated for an undergraduate student project. As a result of the large complexity, scope, and duration of the project only one to two people involved in the project had a complete view of it. Furthermore, the details of the project were extremely overwhelming to the new students who joined the project each semester. Students would often loose interest as a result of the complexity and difficulty in grasping the scope of the project.

Additionally, the duration of the project caused strain on those involved. After one to two

years of involvement even graduate students involved needed to focus on other classwork, graduating, and moving on to other projects.

B.1.10 Ineffective data organization

The ION satellite project was consistently plagued by horrible documentation and an unusable data management scheme. An enormous amount of documentation existed but very little of it was dated, logically organized, easily accessible, or credited properly. As a result it was very difficult for developers to determine what documentation was applicable as neither dates nor authors of changes to documentation could be determined.

One of the documentation requirements of the ION project was that each semester a project final report was to be submitted. The creation of this documentation typically consisted of each team appending to a single 50 megabyte document that had been in existence for years. Not only did this document become completely unreadable as a result of its size, but it continued to be carried forward completely outdated and with contradicting information.

In the future it is recommended that a single information repository exist for all information. The repository should be easily accessible and logically organized by teams. A clear distinction must be made between what is historical data that is being kept for posterity and what is currently applicable data that is being updated. Every piece of documentation should have a clear revision history including dates, authors, and reasons for changes. Each piece of documentation should have an individual or limited group of individuals responsible for its maintenance.

B.1.11 Aloof faculty involvement

The involvement of faculty advisers on the ION project was very limited. The entire project was carried and directed by what seemed good ideas to the students and teaching assistants with occasional input from faculty when explicitly queried.

Generally, the faculty had absolutely no knowledge of any of the technical details of the project or the actual project state. All faculty awareness of project details was the result of oral student reports given during weekly meetings. Unfortunately, because of self interest in obtaining good grades, the project state and work being performed was always spun to give the appearance that there were no problems.

As a result of this, up until the last year of the ION project, faculty advisers consistently believed that the project was much farther ahead than it was in reality. While this was not a problem because faculty did not make any decisions and all development was student driven, this did result in very little formal pressure to perform quality work. Generally, student efforts lacked any quality control as work was performed as pleased with only self-enforcement or mean teaching assistants creating any pressure to make sure things were done correctly.

It is strongly suggested that faculty provide much more oversight on such projects. Faculty should be involved with technical details and should provide more direction and enforcement of proper industry standard procedures.

B.1.12 Difficulties in testing

Testing of both the ION software system and the ION satellite was very difficult and time consuming. The software requirements originally determined for the satellite did not include any requirements for simple testing mechanisms. As a result, no mechanism was designed which allowed for quick, easy, and transparent operational testing. All satellite operations had to be tested exactly as if they were being performed from a ground station on Earth.

Additionally, no formal testing procedures or milestones were outlined for the testing of the ION satellite. This made it difficult to keep track of what portions of the satellite hardware and operations were working and trustworthy, resulting in much redundant testing.

It is suggested that software requirements should include testing requirements so that any software system developed is designed for test. Additionally, formal operations testing procedures should outlined before testing of a satellite begins so that testing is performed in a systematic manner.

B.2 General Comments

Software development of the ION software system consisted of much more than just software development. The entire mission of the satellite was invented and defined by the software team based upon a general knowledge of what the satellite was to do along with the hardware onboard. While this is not that much of a problem as the primary goal of the satellite was an educational experience and not the construction of a commercial satellite, this is a large burden to place on a team whose formal task is to write software.

An unknown mission requirement led to the implementation of a generic and flexible software system for the ION satellite. This created quite a bit more work for the software team than just making a satellite work. The end task that was given to the software team could be summarized as follows: “Here is a one of a kind computer and a collection of hardware. No one is entirely clear on how either work, but we'd like you to make them work and do what we have in mind for them. But we're not going to tell you exactly what we want to do with this hardware."

An enormous amount of time and effort was spent "reinventing the wheel." While educationally very useful, there is a feeling of wasted effort as the end product developed is in no way better than what had previously existed. Furthermore, little of the product created is in anyway reusable outside of the scope of the ION satellite project.

The ION project was certainly meant to be an educational experience to students and teaching assistants. A great sense of accomplishment and educational value exists with the development of a bootloader, operating system, file system, collection of application software, and collection of device drivers. Unfortunately, only two software developers truly obtained the benefits of this experience. These two students implemented an entire operating system and supporting applications given absolutely nothing and now appreciate the work and design decisions behind standards such as Linux, TCP/IP, FAT16, ELF, `malloc()`, posix, and cooperative multitasking. Unfortunately, all other software developers were merely transient students who performed a small piece of work and did not gain the benefit of implementing a full satellite software system from scratch. To these students, it may have been of much greater value if more standards had been used as they could have seen and learned the use these standards in a realistic environment.

Furthermore, it would have been of greater benefit to the ION satellite's success to decide to trust and use standard components. Instead of devoting time and resources to rewriting existing software with intentions of making it trustworthy it may have been more beneficial to simply use the time and effort to thoroughly test existing software standards.

While there is great appeal in developing all software from scratch, it is of little benefit to do so. One of the guiding principles behind the current small satellite movement is the use of commercial off the shelf hardware components. This same philosophy should be adopted to software.

While the design process outlined in Chapter 2 of this paper follows a very intuitive and clean path, the actual development process was never conscious of this path. The development path taken has only been realized upon reflection. In reality, nothing was this clear and while what was outlined did happen as outlined, it was never this explicit of a decision or process.

At the time of writing, the ION satellite has not yet been launched. How it performs in space and whether the effort outlined in this paper will be successful is still to be determined. Regardless of satellite success, the experience gained from the development of the ION satellite software system is absolutely invaluable and it is hoped that future engineering students have similar opportunities.

REFERENCES

[1] Surrey Space Centre, University of Surrey, "Small satellites home page," April 2005, http://centaur.sstl.co.uk/SSHP/index.html.

[2] K. Baker and D. Jansson, "Space satellites from the world's garage - the story of AMSAT," presented at the National Aerospace and Electronics Conference, Dayton, Ohio, May 1994.

[3] M. Long, A. Lorenz, G. Rodgers, E. Tapio, G. Tran, K. Jackson, R. Twiggs, and T. Bleier, "A CubeSat derived design for a unique academic research mission in earthquake signature detection," presented at 16th Annual AIAA/USU Conference on Small Satellites, Logan, Utah, 2002.

[4] L. Alminde, M. Bisgaad, D. Vinter, T. Viscor, and K. Z. Østergard, "The AAU-CubeSat student satellite project : architectural overview and lessons learned," in *Proceedings of the 16th IFAC Symposium on Automatic Control in Aerospace*, 2004.

[5] S. Waydo, D. Henry, and M. Campbell, "CubeSat design for LEO-based Earth science missions," April 2005, http://www.cds.caltech.edu/~waydo/papers/IEEE2002.pdf.

[6] B. A. Larsen, D. M. Klumpar, M. Wood, G. Hunyadi, S. Jepsen, and M. Obland, "Microcontroller design for the Montana Earth orbiting pico-explorer (MEROPE) CubeSat-class satellite," April 2005, http://www.ssel.montana.edu/merope/abstracts/F196_1.pdf

[7] J. Gruenenfelder, "Operating system for control of small satellite systems," presented at 16th Annual AIAA/USU Conference on Small Satellites, Logan, Utah, 2002.

[8] B. Twiggs and J. Puig-Suari, "CUBESAT design specifications document," Stanford University and California Polytechnical Institute, August 2003.

[9] CubeSat Program, California Polytechnic State University, "CubeSat program website," April 2005, http://www.cubesat.org/index.html.

[10] F. Rysanek, J. W. Hartmann, J. Schein, and R. Binder, "Microvacuum arc thruster design for a CubeSat class satellite," presented at 16th Annual AIAA/USU Conference on Small Satellites, Logan, Utah, 2002.

[11] J. A. Carroll and M. D. Fennell, "An autonomous data recorder for field testing," April 2005, http://www.ntsb.gov/events/symp_rec/proceedings/authors/carroll.pdf.

[12] W. A. Beech, D. E. Nielsen, J. Taylor, *AX.25 Link Access Protocol for Amateur Packet Radio*, Tucson Amateur Packet Radio Corporation, 1998.

[13] Dallas Semiconductor, Appl. Note 214, April 2005, http://www.maxim-ic.com/appnotes.cfm/appnote_number/1189.

[14] A. S. Tanenbaum, *Modern Operating Systems.* Upper Saddle River, NJ: Prentice-Hall, 2001.

[15] Wikipedia contributors, "Co-operative multitasking," Wikipedia: The Free Encyclopedia, April 2005, http://en.wikipedia.org/wiki/Cooperative_multitasking.

[16] B. Gatliff, "Embedding with GNU: the GNU debugger," April 2005, http://www.embedded.com/1999/9909/9909feat2.htm.

[17] B. Gatliff, "Embedding with GNU: the GDB remote serial protocol," April 2005, http://www.embedded.com/1999/9911/9911feat3.htm.

[18] Free Software Foundation, *GNU Binary Utilities Manual*, April 2005, http://www.gnu.org/software/binutils/manual/index.html.

[19] Free Software Foundation, *CVS-Concurrent Versions System Manual*, April 2005, http://www.gnu.org/software/cvs/manual/index.html.

[20] M. Dabrowski and L. Arber, *ION Satellite Software Code*, April 2005, http://www.interave.net/cubesat/Technical_Data/ion-sat-FA03/.