Preface

This document is a preliminary CDR outline for our presentation. It will be periodically updated and as such, document changes will be highlighted as follows:

- Additions since last version will be highlighted in magenta.
- Deletions since last version will be highlighted in red.

If there are any problems with this document, contact Cory Maccarrone (maccarro@colorado.edu) immediately.

Introduction

Background

- The purpose of the software subsystem is to interface the user with the hardware through commands given either directly or through a schedule.
- We will maintain the safety of the spacecraft by automatically taking preventative measures in the event of an emergency or problem.
- We will provide software for all ground stations to facilitate sending commands to the spacecraft.

Agenda

1. **Design Overview** - Detailed description of our design, including components, software structure and layout, operating system design, and process communication mapping

2. **Prototyping Efforts** - Outline of the results of any prototyping we’ve done up to now (this includes loading a kernel onto the flight board, messing with the pre-built Arcom Linux system, etc)

3. **Changes to System Operation** - This includes updated operations scenarios and scripts, as well as system performance considerations.

4. **Changes to Major Software Components** - An updated account of any major software changes we’ve made since PDR (the addition or stripping of certain modules, etc).

5. **Requirements Traceability Matrix** - An outline or table of requirements, both assigned to us and by us, due dates and implementation dates, etc.

6. **Software Reuse Strategy** - THIS ONE’S IMPORTANT! This will go over the code we plan to reuse (in this case the 3CS code), including any changes to our original proposal, any new or revised reuse tradeoff analyses (to reuse or not to reuse, that is the question...). Also, we need to outline our precise strategy for reusing the code, as well as what we plan to use and not to use, including reasons and statistics.
7. **Testing Strategy** - REALLY IMPORTANT! How will we test this software once it’s built? We’ll need to go over simulators, ways of running the software without the use of the flight computer, scripts to automate the testing of all commands and functionality, and anything else we can come up with.

8. **Required Resources** - Hardware required, internal storage requirements, disk space, impact on current computer usage, impacts of compiler (?)

9. **Changes to the Software Development / Management Plan** - Have we changed anything?

10. **Implementation Dependencies** - This will describe the order in which components should be implemented to optimize unit and packaging testing. We need to decide on this so we can map out our porting priorities.

11. **Updated Software Size Estimates** - How big will our software be? See below.

12. **Milestones and Schedules** - When we expect things to be moving. This is mapped on the website in a general form, but we may want to be more specific.

13. **Issues, Risks, Problems, TBD items** - Things we’re concerned about. I hope we won’t have anything TBD, but that really depends on other subsystems (science & their algorithm, COMM and the communications rate they can get us, etc)

**Design Overview**

- Flight software Runlevels (FSWR)
  1. Satellite init and deployment (INIT)
     - Software orients satellite, starts deployables
  2. Normal Daytime Operations (DNOP)
     - Images periodically shot, algorithm working, sensors polled
  3. Normal Nighttime Operations (NNOP)
     - No images taken, algorithm working, sensors polled
  4. Emergency Operations Mode 1 (EMER1)
     - All nonessential tasks are stopped (no science)
     - Hardware sensor probes are increased in frequency and verbosity
     - All data in memory is written out to flash
     - Commands are executed by priority to accommodate emergency operations
     - This mode assumes something has gone wrong with a subsystem on the satellite
  5. Emergency Operations Mode 2 (EMER2)
     - All non-essential tasks are stopped (no science)
Hardware sensor probes are increased in frequency and verbosity

All data in memory is written to RAM disk

Commands are executed by priority to accommodate emergency operations

This mode assumes something has gone wrong with primary flash filesystem

- **Hardware I/O managers**
  - All hardware I/O managers inherit a base IOMGR class, of which data going to hardware passes through odata and from hardware through idata.
  - Wireless 802.11 devices will connect through cameramgr
  - Hardware I/O managers implement drivers specific for their hardware (in this case, interfaces to the appropriate parts of the kernel)
  - Protocols for talking to hardware can be both uni-directional and bi-directional

- **Software I/O modules**
  - Each section of the flight software will be separated into processes that run independent of each other
  - Inter-process communication (IPC) will be accomplished through the use of socket connections (either hand-written or through the SCL data bus)
  - Hardware interfacing will be accomplished through upper-level kernel calls instead of direct hardware interfacing (pertains to GPIO and serial communications only)
  - USB handling will for the most part be untouched by the flight software (Any requests will be passed on to GPhoto2, handled by a separate cameramgr)
  - FSW inner workings will be exactly the same as current implementations, but modified to work with a Linux environment and to fit mission needs
  - Commands will be added to all modules depending on individual subsystem needs
  - Science modules will cover analysis and conversion of images, and will interface with cameramgr for retrieving and taking images
  - **Power**
    - Responsible for sending commands to the power module, reading sensor values from power, and reporting results
    - Makes use of gpiomgr
  - **Science**
    - Responsible for the acquisition and analysis of cloud images
    - Makes use of cameramgr, gpiomgr
– Science Analysis - Cloud Height Algorithm
  • Find common points on two images (height reference and rotation reference)
  • Triangulate pixel changes
  • Notify SCL if we image a significant cloud formation
– Science Analysis - Topo Map Algorithm
  • Cloud heights taken on grid points are combined into file
  • Interpolation between points may be possible

○ Tip Mass
  – Responsible for communication with tip mass, and for downloading images from tip mass camera
  – Makes use of cameramgr

○ ADCS
  – Responsible for aligning the satellite in FSWR 1 (INIT)
  – Responsible for making slight adjustments as needed in FSWR 2, 3 (DNOP, NNOP)
  – Responsible for corrective changes in FSWR 4, 5 (EMER1, EMER2)
  – Uses gpiomgr

○ EDC
  – Responsible for monitoring and checking filesystem and flash partitions for consistency
  – Responsible for monitoring running flight modules and keeping them alive (restarting them if they crash, or logging an error condition if they keep failing)
  – Responsible for taking corrective measures in the event of a catastrophic failure (outlined below in Error Correction)

○ COMM
  – Responsible for establishing communications with ground (starting pppd), sending network data out as needed, broadcasting H&S packets in NOP
  – Responsible for broadcasting emergency beakons in EMER
  – Responsible for broadcasting status beakons in INIT
  – Uses serialmgr

○ Health
  – Responsible for gathering sensor readings of vital systems
- Responsible for assembling data into a H&S packet

- Responsible for signalling COMM to terminate PPP for a broadcast of H&S (Reid-Soloman)

- Responsible for sending H&S to COMM and signaling COMM to send a packet

- Responsible for signaling flight software into FSWR 4 (EMER1) upon sensor error

- Uses serialmgr

○ Deployment

  - Responsible for one-time deployment of all deployables at the proper times (scheduled by SCL)

  - Operates only in FSWR 1 (INIT)

  - Terminates after finish of deployments

  - Uses gpiomgr

○ Message Dispatcher (FSW base class ProcessTask) (See Figure 1)

  - Receives a command through the CMDBLK from Route__cmd

  - Initial processing drops into the Process__CMD function

  - Responses are sent to and received from hardware through IOBLK

  - Processing a response from hardware is handled through Process__Reply

  - Returning a status for each command is done through Process__Reply

  - Responsible for communicating results to SCL and dispatching commands from SCL

• Communications Protocols

  ○ TCP/IP based, meaning the satellite will have an IP address with password and firewall security

  ○ Standard FTP and telnet daemons will be used (FTP is resumable in the event of broken downloads, and broadcasts of health and status (H&S) will be done without PPP and can be received by any computer with a radio)

  ○ TCP communications will be tunneled through compressed SSH

  ○ SCL will use built-in socket connections

• SCL (see Figure 2)

  ○ A command comes in via a socket

  ○ The RTE (Real Time Engine) decides what to do with it (immediate command, run a script, or run a flight software command)
The SCL model contains:

- Scripts to execute mission objectives and correct problems

  - FSWR 1 (INIT)
    - Positions Satellite using ADCS commands
    - Deploys deployables using Deployment commands
    - Switches to NOP runlevel
  
  - FSWR 2 (DNOP)
    - Keep science operations going (Take pictures, process them, repeat)
    - Make small corrections in course as needed
    - Switch to FSWR 3 as needed
    - Switch to FSWR 4 upon notification of a problem

  - FSWR 3 (NNOP)
    - Keep science analysis going (no pictures taken)
    - Make small corrections in course as needed
    - Switch to FSWR 2 as needed
    - Switch to FSWR 4 upon notification of a problem

  - FSWR 4 (EMER1)
    - Stop Science module from running
    - Signal processes to save all data to flash
    - Make small corrections in course as needed
    - Execute proper actions depending on problem reported
      - Turn off devices as needed, vast corrections in course, etc
    - Switch to FSWR 2 or 3 after problems corrected

  - FSWR 5 (EMER2)
    - Stop all non-essencial services from running
      - This includes ADCS, so we may drift during this!
      - Power, Health, COMM, EDC will remain running, or will restart upon filesystem switch
• Save all data to RAM (flash assumed bad)
• Signal EDC to begin switching filesystems
• Wait for signal to resume operations from EDC
  – ADCS and any processes that needed to stop will be restarted into FSWR 4 operations while we fix things
  – EDC will attempt recovery of bad filesystem
• EDC will signal when problems are fixed
  – Signal running software to save to RAM and exit (FSWR 5 operations)
  – Copy RAM data back to flash
  – Signal EDC to switch filesystems
  – Upon success, restart stopped tasks and continue in FSWR 4 to ensure no further problems and to correct for ACDS drift

• The database contains records that hold sensor and derived information
• Telemetry is gathered from the database and sent to the ground
• SCL Model for DINO
  – Scripts to execute mission objectives (ICD from each subsystem will enable us to create scripts to run their components. Software needs commands, parameter, and timing information.)
  – Rules to fix problems (The Systems Team will tell us what can be done if a sensor goes out of limits)
  – Database definition (Each sensor from every subsystem will have an entry. Each value we want to calculate (derive) will have an entry. Each piece of hardware will have a field to store operational status.

• VMOCC (See Figure 3)
  – Get commands from users worldwide
  – Deliver archived data (if it satisfies request)
  – Create a schedule for the satellite
  – Automatically initialize a radio (anywhere in the world) and send the command
  – Accept telemetry from any radio in the world and integrate into VMOCC model
  – Deliver requested data back to users
• Use TCP/IP protocol similar to CX
• Ground database (MySQL) needed for telemetry being sent
• Ground SCL model (i.e. ground scripts and ground rules) to alert mission operators of problems
• STK model needed to get communication opportunities, science data, and an attitude visualization
• Web interface (i.e. webpage) needed for commanding and receiving data

• Linux Base System and Flash Software
  • Linux and the kernel will be installed into separate flash partitions, bootable using the RedBoot bootloader
  • Kernel will be made from a patched vanilla 2.4.19 kernel (Open to upgrades)
  • Base filesystem will be a compressed JFFS (Journaled Flash File System) with a maximum uncompressed disk size of 14 mb
  • RedBoot will initialize the kernel with proper parameters
  • Kernel will initialize hardware and start the system init scripts
  • Init scripts will load up the network, modules, scheduling support, etc.
  • Final script will load up the flight software and initialize log in programs

• Hardware communication
  • All subsystems will have communication either through one of our 5 serial ports, with one being devoted to COMM, or through general purpose I/O.
  • Any general I/O will be done through the 8 general purpose I/O pins and add-on PC-104 interfaces using gpiomgr
  • USB communication will be handled through external libraries and/or programs (GPhoto2 using cameramgr)
  • Tipmass communication will be handled through the cameramgr module using the Wiser 802.11b wireless radios (interfacing tipmass module with serialmgr and cameramgr)
  • External flight software modules will handle hardware interfacing on a first-come-first-serve basis (normally) or by priority (emergency)
  • Hardware modules will be called serialmgr (serial manager), gpiomgr (general-purpose I/O manager) and cameramgr (camera manager)

• Error-correction methods
  • Periodic MD5-sum comparisons will be made through the use of a background process (EDC module)
- Resident module will watch other flight modules to make sure they’re not crashing or hanging. Errors will be fixed, or logged if unfixable.

- Any errors detected in files will result in a log entry being written, the old copy backed up (in compressed form), and a known good copy being replaced.

- The kernel will be stored in two partitions in Flash, as well as on the main JFFS filesystem. Periodic checks will be performed against all copies of the kernel to maintain consistancy. Any errors will result in a re-program of the bad area of flash.

- The RedBoot bootloader will have startup scripts that will compare MD5 sums of the kernel partitions and in the event that the main partition is bad, replace with the good partition and boot. If both are found bad, it will stop and wait for manual input and the uploading of a new kernel through serial (This may not work too well, as we have a slow connection with a small window of opportunity.)

- Use CompactFlash modules as a backup filesystem

  - In the event of massive corruption, utilities loaded onto a RAM drive could be used to power up the CompactFlash and switch the primary filesystem over to it (using pivot_root)

    - Flight software enters FSWR 5 (EMER2) (all open files are written to RAM drive instead of flash as in FSWR 4)

    - System drops to low runlevel (init and maybe one other task running such that we can switch root filesystems)

    - cardmgr and cardctl on RAM drive are used to power on compact flash and mount it

    - pivot_root to CompactFlash

  - Utilities on the CompactFlash would attempt to rebuild the on-board flash while signalling the flight software to continue operations from CompactFlash

  - Upon successful reprogramming, pivot_root would again transfer control to the on-board flash

  - Flight files saved to RAM would be written both to onboard Flash and to CompactFlash

  - CompactFlash would then power-down until needed again.

  - By keeping the CompactFlash always powered down, we reduce the chance of harm due to radiation

  - If filesystem correction fails, signal failure to SCL, install modified kernel (so we can use compactflash filesystem instead) and prepare for operations from compactflash alone.

    - The only time this should happen is if some part of the compact flash goes corrupt as well, preventing a successful reprogram of the onboard flash.
• We will maintain a complete image of the flight filesystem, so unless that goes corrupt, we should never get here.

Prototyping efforts

• Arcom default Linux image has been loaded to the flight board using serial transfer
  ○ Kernel boots perfectly, Linux system starts up without a hitch
  ○ All hardware (as far as we can tell) is operational through the default image
  ○ System software takes 7 mb of uncompressed space (≈ 13mb compressed), leaving us with 8mb to work with
  ○ Only one kernel partition is loaded by default
  ○ Some base software could be removed to free up additional space (XFree86 takes ≈ 7mb)

• Test kernel built from patched sources and loaded to flight board
  ○ Kernel booted without a hitch, detected all available hardware
  ○ System was unable to boot as we did not upload a JFFS filesystem image along with the kernel.

• Conclusions
  ○ Building a kernel for the board will be a simple task
  ○ Uploading will be simple as well, using either slow serial communications or through the 10/100 ethernet adapter
  ○ Linux system image will be very small and will easily accommodate our needs

Changes to System Operation

• Linux Flight Software
  ○ Trade Study in implementing Python or other small replacement for current flight software
  ○ Results: A combination of old and new software will be used, with new/changed modules being implemented optionally in Python, C/C++, or Perl, depending on needs.

Changes to Major Software Components

• We will not be making use of the I²C bus as originally planned, due to the use of GPIOs, serial communications, and add-on boards. This support was never finished in 3CS.

• Serialmgr will be modified to handle all serial port access not going through to a camera

• Cameramgr will be added to handle all camera traffic
• EDC module will be added to handle all error-correcting and consistancy checking
• Deployment module will be added to handle all deployment specific operations

Requirements Traceability Matrix

• See Requirements Traceability Matrix document and DINO Flight Software IPC Web

Software Reuse Strategy

• Software from the 3CS satellite project will be used as our main FSW
• All modules will be used as they are, with slight modifications necessitated by variances in our mission plans
• **TRADEOFF:** COMM module may need massive modification if used as-is (including Dave’s protocol used for 3CS). This module may not be necessary if we use strictly PPP
  - Results: PPP will be used instead of Dave’s protocol
  - PPP provides much more capability than Dave’s
  - Dave’s is very proprietary towards 3CS and would require a huge effort to port
  - COMM will be written to handle starting ppp daemon, broadcasting health and status
• Software will be re-structured so as to be completely modular (separated into separate “programs”)
• Code will be completely documented and cleaned up to facilitate reuse in future projects (Documentation Project)
• Current plans: Reuse all current 3CS software, with slight modifications
  - Reasons:
    - Ease of implementation and porting
    - Lack of time to implement a new system
  - Trade Study for New Implementation
    - Implement only what we need in a modular, configurable manner using Python
    - Documentation would be easy
    - Language is easy to understand and learn
    - Modular nature of Python is good for us
    - **CON:** Not portable to VxWorks
    - Result: Use a mixture. Flight software will need to be modified anyway to accommodate Linux. Python / Perl modules may be used at option to supplement any additions and changes.
Testing Strategy

- Linux Flight System
  - Testing will be done in the form of radiation testing (randomly corrupting an area of memory to test any correction algorithms)
  - JFFS partition will be subject to random bit-flips to test the robustness of the filesystem
  - All testing for above will be done using testing software built for the purpose of randomly corrupting memory areas
- Flight Software
  - Automated scripts will be written to test all commands the flight software supports, using all ranges of parameters and settings
  - Scheduling of events will be tested automatically using scripts
  - Hardware communications will be monitored through the use of software or hardware “sniffers” that will monitor ports for activity (RCPOD could be used for this)
- Hardware Testing
  - Automated scripts will proceed one at a time through all the functions of the hardware in a logical manner such that all subsystems are tested with all available software commands.
  - Results will be displayed and logged
- Non-flight hardware testing
  - Flight software can be tested (although not completely) using different compile flags such that any x86-based Linux computer can run the flight software.
  - Hardware differences will need to be taken account of for this, but #ifdefs can be used
- Code Documentation
  - Language Unit Test Framework
    - Automatically tests code to make sure it performs as expected
    - Available for C, C++, Python

Required Resources

- Hardware Requirements
  - Flight Computer $\Rightarrow$ Arcom Viper
  - CompactFlash ($\geq 16$mb)
  - Cameras from Science and Tip Mass
• Storage Requirements
  o At least 16mb of on-board non-volatile flash RAM
  o Compact Flash module no smaller than 16mb
    – Will be used for backup flight-software in case of severe problems
    – Can also be used for temporary storage for images and topo maps

• Software Requirements
  o GNU C compiler built for cross-compiling Linux binaries x86 → ARM

Changes to the Software Development / Management Plan
• Are there any?

Implementation Dependencies
• Components should be implemented in this order for optimum testing:
  1. Power
  2. C&DH
  3. All the rest
  4. COMM

• Testing can happen as soon as the flight board + peripherals are attached to power.

Updated Software Size Estimates
• Kernel: 500 ± 100 kb
• Base Software: 3.0 ± 1.5 mb
• SCL: 1.0 ± 0.7 mb
• Flight Software: 2.0 ± 0.5 mb

Milestones and Schedules
• Documentation Project done by March 19th
• Porting started by March 19th
  o Minimal functioning system by May 31st
  o Porting complete with modifications by mid August
• Linux Kernel and Base System done by April 9th

Issues, Risks, Problems, TBD Items

• One concern will be the speed of communications. We’re working on getting 9600 baud, but we may be limited to 1200 baud.
  ○ This will mean we’ll need to scale our transfers down dramatically, implement compression algorithms, etc

• Flight software makes use of POSIX mqueues, which are unsupported under Linux. These will need to be replaced with SysV-style queues or some other mechanism.
  ○ Possible implementation is a wrapper library for mqueues that translates the functionality into a SysV compatible format, but works transparently to the flight software
Figure 1. Flight Software Uplink (Downlink works in reverse)

Figure 2. SCL Flow Diagram
Figure 3. VMOCC Downlink