

Linux and the Spacecraft Flight Software Environment

Edward J. Birrane, Kathryn E. Bechtold, Christopher J. Krupiarz, Andrew J. Harris, Alan M. Mick,
Stephen P. Williams

The Johns Hopkins University Applied Physics Laboratory
11100 Johns Hopkins Road Laurel, MD 20723; 443 778-7432
Edward.Birrane@jhuapl.edu

ABSTRACT

Flight software development must evolve as the operational characteristics of spacecraft evolve. Flight development typically makes use of a monolithic architecture comprised of custom-built, tightly coupled software. This dense coupling precludes the development agility desired by small spacecraft software efforts. Specifically, mission requirements are becoming aligned with re-use centered, highly decoupled, distributed architectures otherwise popular in desktop and web development. To meet the engineering challenge of matching the needs of these missions, the software environment must be modernized and the software architecture decoupled.

The use of Linux in a flight environment promises to fill this need while significantly lowering the barrier of entry for new developers, especially in the university setting. To assess this promise, a team of flight software researchers at The Johns Hopkins University Applied Physics Laboratory (JHU/APL) have completed a study of the impact of real-time Linux in a real-world embedded environment. This study assessed the impacts of Linux on the spacecraft software development environment and explored the new types of software architectures enabled by that environment. At the end of the study, the team reached conclusions regarding the value of pursuing Linux in a flight environment.

INTRODUCTION

The Johns Hopkins University Applied Physics Laboratory (JHU/APL) has developed and flown dozens of spacecraft since the dawn of the space age including the currently operational Thermosphere, Ionosphere, Mesosphere, Energetics and Dynamics (TIMED), MErcury Surface Space ENvironment GEOchemistry and Ranging (MESSENGER), New Horizons, and most recently Solar TERrestrial Relations Observatory (STEREO) missions. JHU/APL is responsible for the end-to-end development of these spacecraft including mission design, spacecraft development, and mission operations. Included in this development is spacecraft flight software.

To build upon this experience, each year the JHU/APL Space Department requests ideas and concepts for funding via the internal research and development (IRAD) program. IRADs are requested for both of the Space Department's primary business areas: 1) Civilian Space and 2) National Security Space. The intent of this program is to provide seed money for concepts that cannot typically be funded directly by a mission and to explore ideas that can influence future spacecraft development.

Our work within the Space Department's Embedded Software Systems Group involves identifying and

researching ways to improve the quality and efficiency of mission critical flight software. The IRAD program provided an excellent opportunity, away from a particular set of mission constraints, to follow the inroads Linux was making in the world of embedded software.¹² We were impressed by the value proposition of a decoupled system and were curious if Linux could be a key technology to reducing costs for small spacecraft for both Civilian Space and National Security Space missions. As such, we submitted a response to the IRAD call and were awarded funding for FY07 with the goal of determining the applicability of Linux to our flight software architectures and its potential to reduce costs and risks.

Typical JHU/APL Architectures

To understand where Linux fits into our systems, it is necessary to understand a typical JHU/APL architecture, as shown in Figure 1. The main spacecraft computer resides in an integrated electronics module (IEM) with multiple wires extended from the IEM for communication with instruments, guidance and control elements, solar arrays, and other subsystems. Examples of these communication lines include MIL-STD-1553B, serial lines, and Spacewire.

The flight computer consists of one or more microprocessors performing various operations for

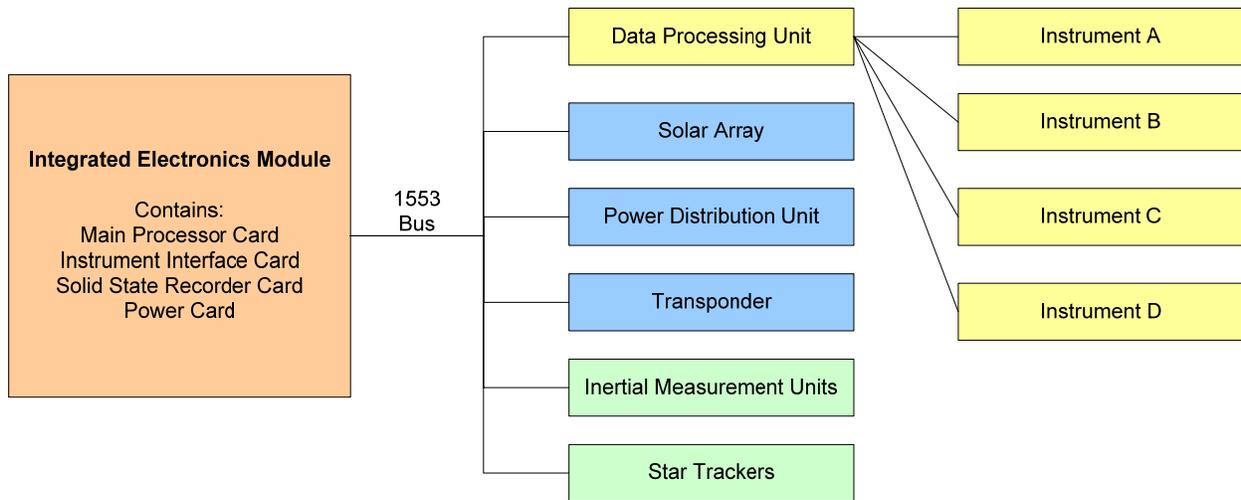


Figure 1: Typical JHU/APL Spacecraft Hardware

commanding and controlling the spacecraft along with a device for storing mission data. Examples of such systems are shown in Table 1, which provides an overview of past JHU/APL command & data handling (C&DH) subsystems and the amount of storage available on their solid state recorders (SSRs). These SSRs consist of static random access memory (SRAM) or other types of volatile or nonvolatile memory to store science and spacecraft housekeeping data.

Table 1: Command and Data Handling Computers for JHU/APL Missions

Mission	Year	CPU	Speed (MHz)	SSR (Gbits)
NEAR	1996	RTX2010	6	1 & .5
TIMED	2001	Mongoose	12	5
CONTOUR	2002	Mongoose	12	8
MESSENGER*	2004	RAD6000	25	8
New Horizons	2006	Mongoose	12	64
STEREO	2006	RAD6000	25	8

*Processor performs both C&DH and G&C

For these missions we developed the flight software over commercial real-time operating systems. TIMED, CONTOUR, and New Horizons used the Nucleus operating system from Mentor Graphics and MESSENGER and STEREO used VxWorks from Wind River Systems. Software for these computers is loaded as a monolithic block of code with multiple processes running within the operating system.

Development and Run-Time Environments

Given this legacy architecture, we identified two fundamental needs to support an extremely low-cost

flight software environment: a rapid development environment and a modular run-time environment.

Development environments are comprised of software tools (compilers, debuggers, profilers), development hardware (breadboards, testbeds, and emulators), and commercial off-the shelf packages (network communications, file systems, compression, and other services). A rapid development environment allows for the meaningful creation and preliminary testing of code before hardware is available and provides the ability to incorporate the widest array of existing, third-party utilities. Incorporating these utilities with little or no code modification (with a goal of plug-and-play flight software) requires a run-time environment that provides software modularity – the ability to permute logically separate software functions with minimal design, implementation, testing, and run-time impacts. The benefits of such a software environment are summarized in Table 2.

Paradoxically, it is the *software operating system (OS)*, and not the *flight software architecture*, that is responsible for creating the software environment. As such, Linux has the potential to fill that role for several reasons. Its process model is well known and its adherence to open standards significantly reduces hardware and software training, support, and integration costs. As flight hardware continues to decrease in cost and increase in capability, the processing overhead of running a generalized operating system becomes negligible. It is now possible to run real-time Linux variants on flight development systems and, subsequently, to extrapolate its ability to run on flight production systems.

As an operating system with a broad user base, several tools and utilities are freely available in Linux that are otherwise expensive or dated on other traditional flight operating systems. The kernel process model allows for distributed, independently upgradeable software components. Core utilities, such as the GNU suite, are both well known to the software development community and have been user tested for decades in several different application domains. The ability to build upon this existing, open body of work removes the need to design, implement, and test important capabilities of a flight-ready architecture, such as persistent storage and compression. This type of re-use allows a smaller flight software team to dedicate their efforts towards that which makes their flight application unique.

Table 2: Advantages of Linux Based Development in a Flight Software Environment

Features	Benefits
Decoupled Software	<p>The software environment is robust.</p> <p>Processes may be restored without CPU reset.</p> <p>Loss of one data collecting process will not interrupt other, functioning data collecting process.</p> <p>Memory protection vastly reduces chance that one process may negatively affect another.</p> <p>Software updates are smaller, targeting individual processes.</p> <p>Critical updates might not interrupt critical, on-going data collection.</p> <p>Radiation Damage Adaptation</p> <p>A single upset event will probably affect a single process.</p> <p>Robust environment will absorb this without CPU reset.</p>
Reduced Software Development Time	<p>Coding to interfaces speeds development</p> <p>Decoupled software provides for multiple programming languages (C, C++, perl) suited for each task.</p> <p>Unit testing to interfaces, done concurrently for each task, greatly reduces system integration time.</p> <p>Development distributed across hardware platforms and across development sites possible with decouple software architectures.</p> <p>Plug and Play Software</p> <p>Decoupled architecture eases COTS insertion</p> <p>Programming language variety eases existing domain software integration.</p>

The ability to simulate the desktop software environment on flight hardware also enables the converse: existing desktop and laptop computers become accurate simulations of flight resources. This allows earlier development on faster, less expensive machines for advanced prototyping and non-realtime

testing and integration. This not only allows fewer flight development hardware purchases but also allows significant progress in the event of hardware delays or the need to deliver testbed systems. Further benefits for Linux are listed in Table 3.

Table 3: Advantages of Linux Operating System in a Flight Software Environment

Features	Benefits
Standard Development Toolchain	<p>A larger developer community makes it easier to find/train expert users.</p> <p>A larger user-base provides more stable and feature-rich implementations.</p>
Standard Applications	<p>Linux provides hundreds of available tools whose implementation has been verified over decades of use in dozens of industries.</p> <p>Data compression (gzip, tar, compress)</p> <p>Flash File Systems (JFFS, JFFS2)</p> <p>Scheduling algorithms (cron)</p> <p>Data manipulation (sed/awk/perl/python)</p> <p>Scripting (sh, csh, bash, tcsh, etc...)</p> <p>The modular nature of these tools enables a modular (and responsive) architecture for the flight code that uses them.</p>
Optional Real-Time	<p>Real-time tasks execute in real-time</p> <p>1553 bus drivers exist for real-time Linux.</p> <p>Non-real-time tasks may <i>also</i> be implemented</p> <p>There are many non-real-time tasks in a given flight software application (data manipulation, flash activity)</p> <p>The ability to run some tasks outside of the real-time system allows for use of advanced tools/languages.</p>
Advanced Development and Testing	<p>Software development can be started without flight hardware</p> <p>Linux can be installed on a desktop or laptop PC and provide <i>the same</i> tool support as that which will run on the flight hardware.</p> <p>Significant debugging and unit testing can be accomplished on development machines.</p> <p>Testbeds can be constructed immediately for non-real-time testing</p> <p>A majority of code validation is algorithm correctness which can be tested independent of real-time capabilities.</p> <p>Peripheral hardware emulation easier using Linux's modular interface architecture.</p>
Technology Infusion	<p>Algorithms from other industries are more easily incorporated into a modern software development environment.</p> <p>Protocol security projects</p> <p>Modern (C++, Java, Perl, etc...) implementations of domain algorithms can be more easily ported.</p> <p>Incorporation of feature-similar hardware</p> <p>Linux device-driver model provides for hardware upgrades (even across manufacturers) with little or no core code changes.</p>

These benefits both reduce development schedule and development cost. Standardized, free development tools reduce training and support costs. Standardized, free utilities remove the need to design, implement, and test important capabilities in the software system (persistent storage, compression). The ability to develop and test non-real-time software on very fast and inexpensive Linux desktop and laptop computers will save money when developing testbeds. Even real-time software can be validated up to but excluding its real-time constraints. Finally, the ability to selectively upgrade heavily modularized system components greatly reduces regression testing efforts. Operating system selection (and the utilities provided by that operating system) is an extremely significant driver of any software effort.

PROJECT CONFIGURATION

Project Team

The IRAD project team consists of a range of personnel with varying amounts of Linux and flight software experience. All of the participants have experience on at least one JHU/APL mission with some having developed software for several spacecraft. Additionally, the project team has varying ranges of Linux experience from none to extensive. This wide range of backgrounds enables a wide range of insight. Extensive flight software experience helps to anchor ideas in the reality of flight software while more Linux experience provides insight into innovative methods for

flight software operation.

Hardware Platform

As seen in Table 1, the latest JHU/APL spacecraft use Mongoose and RAD6000 processors with future missions in considering the use the RAD750 processor. These processors, even as a non-rad hardened commercial single board computer (SBC), are too costly to be purchased on small IRAD funds. Hence, we needed an equivalent, but less expensive, board for our software development research. We selected the MCP750 SBC for this purpose as it has been used for other software prototyping at both JHU/APL, the NASA Jet Propulsion Laboratory, and the NASA Goddard Spaceflight Center. To continue minimizing costs we made our first foray into on-line auctions and acquired additional boards via eBay. Ultimately, the department procured six networked boards for general development.

The MCP750 SBC contains the MPC750 chip, which has a PowerPC architecture similar to the RAD750 developed. As such, it provides a reasonable facsimile of flight-ready hardware, albeit with a higher clock speed. The MCP750 has serial and Ethernet connections, on-board flash memory, and a PMC slot. MCP750 hardware features are shown in Figure 2. Developers can configure the MCP750 boards to boot using multiple methods, including booting directly to an onboard debugger and booting over an Ethernet network.

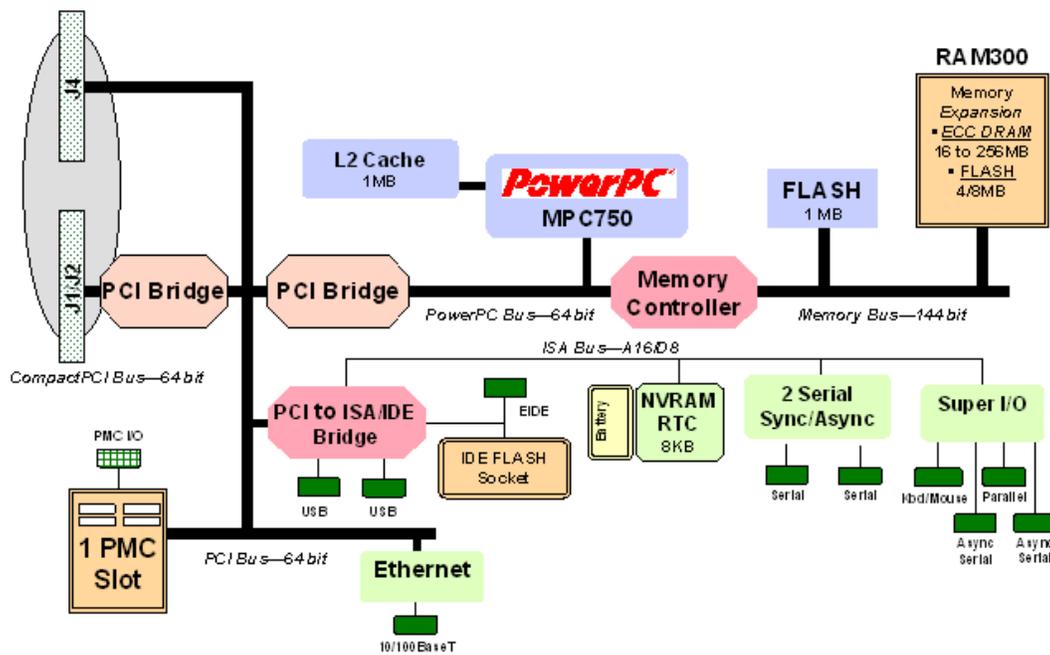


Figure 2: MCP750 block diagram³

Software Development

Throughout early development, code images for the MCP750 were loaded over an Ethernet network. We found it necessary to initially boot the MCP750 into its debugging configuration in order to set various network variables. Once an initial test was performed to ensure connectivity to a Linux workstation, the board was reconfigured to perform a network boot via the tftp protocol. This differs from our past flight software development where either the operating system kernel was compiled directly into the software image or was available in EEPROM. Loading the kernel in this fashion is an improvement over our past methods, although it is not unique to the Linux platform.

To boot, the SBC retrieves the Linux kernel from a server, stores the kernel in RAM, and jumps to the initial command to boot. During the startup process, the SBC mounts two directories from the server:

- The root directory used in the Linux distribution.
- A directory containing development code and executables.

This type of directory mounting is unacceptable in a deployed environment, but for a development configuration it simplifies code creation. In this situation, we deviated from our “test as we fly” approach to software development. Our development turnaround time was greatly decreased by this more efficient development boot process.

Developers login directly to the SBCs as if they were standard Linux workstations -- there is no discernible lag in executing code in this fashion. This is in contrast to significantly longer image load times typically seen in a flight software development environment.

As stated previously, our spacecraft computers operate at very slow clock speeds. The MCP750 boards operate much faster (233 MHz to 367 MHz). Unlike the commercial flight boards, we cannot clock down the MCP750 to replicate a realistic speed. As a result, timings and other metrics that we observe with the board need to be scaled mathematically to the rates seen on typical flight software. This is a disadvantage as our past flight experience has shown these results can have a margin of error significant enough that it can impact successful deployment of software developed on an alternate platform.

COTS Software

A board support package (BSP) is a set of software customized for a particular hardware board and a

particular operating system. It contains functionality used to operate critical board devices required by the operating system to boot the computer. Running Linux on an MCP750 requires a customized BSP. The BSP must provide driver support for all utilized devices on the MCP750. Constructing a BSP for Linux on the MCP750 is outside the scope of our project. As such, we use a commercial BSP.

The core operating system used by both the runtime and development environments of this project is the RTLinux Pro (RTL). RTL was purchased from Finite State Machine Laboratories (FSM) which has since sold its product to Wind River Systems. RTL is a realtime scheduling kernel which hosts a non-realtime operating system (either a Linux or BSD variant) as the lowest priority thread in the system. We use RTL to host a Fedora Core 4 distribution for development and an FSM Carrier-Grade Linux for the MCP750 target. We installed RTL on two separate machines already running Fedora Core 4: One containing an i386 tool chain and one containing a PowerPC tool chain.

Environment Familiarity

One of the goals of the project was to demonstrate distributed development with the Linux environment. To that end, we use several heterogeneous development environments when developing code for this project:

- A Linux (Ubuntu) development desktop in an office physically separate from the project laboratory, but behind the JHU/APL firewall.
- A Linux laptop connecting to the JHU/APL network over a VPN connection.
- A Windows machine using a GUI-based IDE for code generation and directory mounting to a Linux development workstation in the project lab.
- Directly compiling using Linux development workstations within the project lab.

In all cases, the development team uses a variety of familiar development tools in familiar development environments with their native office operating systems to generate code in whichever manner is most efficient for them. Different members of the development team preferred different methods of code development. For instance, we continue to trade opinions on the benefit of command line based code development versus integrated development environment (IDE) development and whether it is better to remotely log in to a system or to have everyone running their own version of Linux locally. We established a development

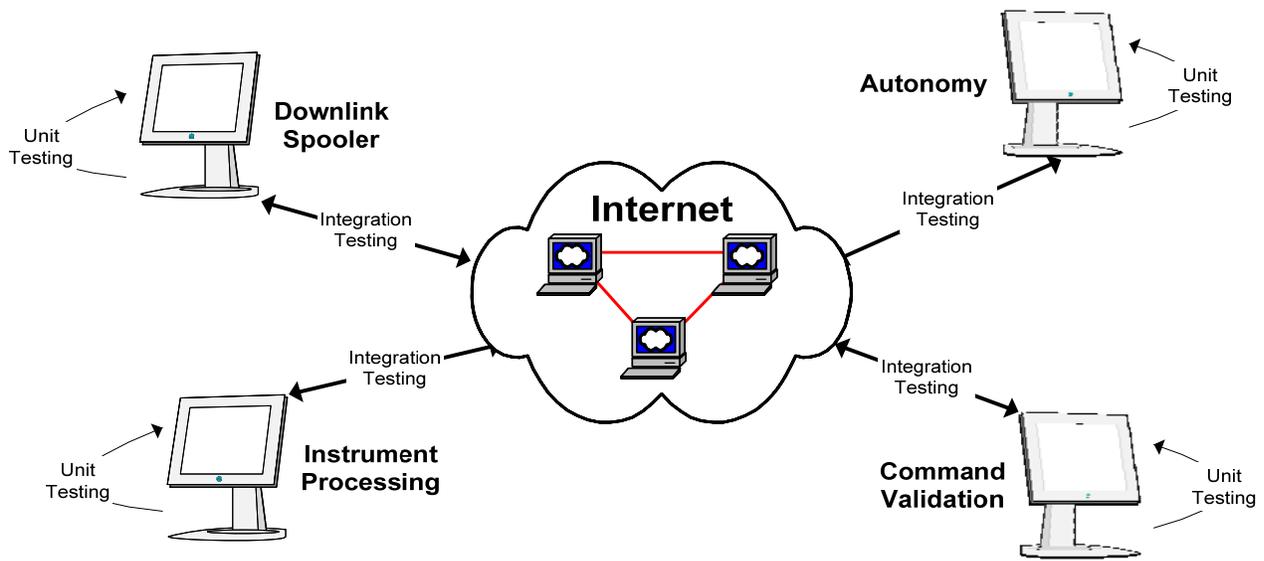


Figure 3: A heterogeneous development environment

system which allowed everyone to work using the tools and methods of their preference, which was an improvement over past development environments. The transfer of this code to a configuration-controlled cross-compilation machine poses no undue burden on development efforts and has been received as a welcome solution to preserving individual compiler and IDE preferences. This experience simulates the distributed development model illustrated in Figure 3.

Cross-Compilation

To perform an initial test of cross-compilation, we ported a suite of space network protocols primarily developed by the NASA Jet Propulsion Laboratory with additional code developed by JHU/APL. This port was only for the non-realtime aspect of RTL. Cross-compiling this code involved using the PowerPC tool chain for compilation instead of the i386 tool chain. Once compiled, the software was loaded to the MCP750 target and executed without issue. We have also cross-compiled other applications which either also had no issues or only required minor configuration tweaks. While not a comprehensive experiment, the strength of being able to cross compile with little or no effort cannot be emphasized enough. One of the major costs in flight software development is to ensure that the software can operate correctly on its delivered platform. While we have previously developed some code on high-powered development platforms, we still had to port it to more resource-constrained runtime target platforms. These porting efforts invariably lead to issues such as operating system compatibility or the

need to reconfigure the code to fit within a monolithic code image. With Linux on both the development platform and the target, these issues are diminished to those rooted in hardware differences (such as differences in CPU speed, available RAM, and available peripherals).

Runtime Target

The MCP750 SBCs were configured to boot over a network interface. The initial boot configuration required one-time modifications such as ensuring that the realtime clock was running and that the system could boot using PREP files.

Connectivity

There have been no difficulties communicating between the target and development platforms using Ethernet, serial cabling, and NFS mounts.

LINUX PERFORMANCE

There are three performance areas identified as important to the proper technical assessment of Linux: image loading, software jitter/drift, and interface performance.

Image Loads

Loading new software to the target, including receiving the operating system kernel over a network file system mount, remains significantly faster than loading similar kernel/software bundles in our other realtime operating systems. The ability to load new software executables

to an already running system without requiring reboot has sped development times significantly.

Once images have been loaded to the target, the boot time is approximately 60 seconds, with 40 seconds devoted to hardware verification and network loading. The actual Linux boot time is 20 seconds. A 20 second Linux boot time is considered very slow if not unacceptable for a flight system reset. However, we have not yet customized the Linux distribution to remove unneeded processes (such as the web server). Once unnecessary processes are removed from the distribution, we will readdress the issue of boot time.

Software Jitter and Drift

Jitter measurements are measurements of time representing variations in execution intervals for an otherwise strictly periodic task. For example, a task may execute every 10 milliseconds, +/- 1ms, where 1ms is the jitter measurement of the runtime environment. Jitter is expected in any system and has no consequence when it exists within tolerances specified by the applications running on the system.

Drift measurements are measurements of time over time that represents an accumulating delay in the execution of an otherwise periodic task. For example, a task that nominally should execute at 5ms, 10ms, 15ms, 20ms, etc... but actually executes at 5ms, 11ms, 17ms, 23ms could be considered to have a drift of 1ms per 5ms. Due to the cumulative nature of drift, no discernable measurement is acceptable in a real-time system.

All software that has been cross compiled has run with approximately similar timing between the development and target environments. There has been no observation of jitter greater than a few milliseconds., which we feel is satisfactory for an untuned demonstration system. There has been no measured drift on the MCP750.

This is a significant topic and an area of focused future research as more flight-appropriate hardware resources are available.

Interface Performance

This effort successfully uses Ethernet, serial, and PCI interfaces. Traffic over these interfaces has not introduced any measured delay in realtime scheduling.

Framework

A defined software framework is recommended for the rapid adoption of a realtime Linux solution for flight software. Beyond technical feasibility, programming

realtime applications using RTL requires experience to use effectively. The structure of programs in an RTL development environment is significantly different than the structure of programs in traditional flight operating systems such as VxWorks.

An example of such a difference is RTL's use of variable-length messaging FIFOs verses fixed-sized messaging queues provided by VxWorks. Variable length messages conserve bandwidth but require additional protocols to detect message boundaries. This additional software is not complex but does impact software architecture. Throughout the technical assessment we discovered several such differences between RTL and the more familiar VxWorks operating system. This led us to observe an interesting paradox. In coming into the project, we felt one of the primary strengths of using Linux was that developers would have the opportunity to use an operating system with which they were intimately familiar. We felt this was particularly true of recent graduates. However, in the cases where we have developers from several past spacecraft, they were instead intimately familiar with the operating systems of those spacecraft such as VxWorks and Nucleus and not as much with a desktop operating system such as Linux.

OUTSTANDING AND UNRESOLVED ISSUES

Although the internally funded project is complete, we received additional funding from the Naval Research Laboratory to further study the application of Linux and to develop a flight architecture that plays to the strengths of Linux. Upon completion of that project we intend to publish further results and to also resolve some of the issues we found during our internally funded project. This includes, but is not limited to, the following list.

Real-time Versus Non-Realtime

One of the first requirements we had for this project was that we definitely needed a hard real-time operating system. The software we write is critical software and we have quite a large number of timing constraints in a command and data handling platform. However, as we began to further analyze our architecture, it became apparent that there weren't as many hard real-time constraints as we first believed and, perhaps, there aren't really any that required a hard real-time system. Most of timing constraints are imposed by commanding and telemetry hardware interrupts and interfaces such as the MIL-STD-1553B. That brought us the question of whether we achieve a solid system that would react to these interfaces without a hard real-time system. Without this requirement more options are available for

Linux distributions and reduce the need for customized solutions such as the RTL.

Packages

Among the first advantages we saw in using Linux was the plentiful supply of open source solutions to problems in our flight architectures. Our thought was that it would be straightforward to take those solutions and apply them quickly and easily to our architecture. The JPL networking protocol software mentioned earlier is an excellent example. This is a set of software that provides a complete suite of software for interplanetary networking and easily cross compiled for the MCP750s. With this start, we were optimistic that we would be able to do the same for other software packages and began the process of downloading software for our boards. Unfortunately, we didn't fully grasp the extent of code of some these software packages. Rightly, many open source packages are designed for many platforms, contain an array of features, and are dependent upon already loaded software. This is a good solution for a workstation, but applying this architecture to an embedded application with very limited memory and a slow processor many times is not feasible. While we were able to run some applications, we were disappointed that we were not able to achieve success with the large number we had hoped. An initial effort to provide leaner implementations of common utilities would be a benefit to any flight-based Linux community.

Reduction of Distribution

In the future, perhaps, spacecraft will have interfaces, web servers, and other workstation applications, but no need for these features has been identified to date. Support of such unused features is a major contributor to code bloat. Spacecraft have limited resources for both storage of an application (and operating system), memory, and processor speed. When we run our Linux systems on a workstation in our lab, we have some non-critical processes using system resources, but the performance impact is negligible. This is not the case on a spacecraft which must use any spare CPU cycles for data processing (such as large image compression). Hence, either a distribution must be pared down or a distribution must only be built from the ground up. We hope the address which avenue is best in our current NRL project.

Flight Readiness

All new flight software eventually reaches the point of asking about its flight readiness. Of course, Linux is no different and while it has flown on spacecraft, we are still evaluating its use for future JHU/APL missions.

CONCLUSION

Our experience with the software and hardware resources allocated to this project has been very positive. Hardware integration into the project lab, and its subsequent configuration to Linux, occurred with minimal difficulty. The homogenization of the runtime and development environment has been a complete success. The development environment is easy to use and transitions seamlessly to the runtime target platform. This has enabled rapid development of test software for this project. A BSP and software architecture are necessary to use RTL on new target platforms. With these tools, follow-on efforts will see similar increases in software productivity, adaptability, and re-use.

ACKNOWLEDGMENTS

The authors wish to thank the JHU/APL Space Department IR&D Committee for awarding this contract as well as Mike Cravens and Adam Fluckey of Wind River Systems for their support.

REFERENCES

1. Huffine, Christopher, "Linux on a Small Satellite", *Linux Journal*, <http://www.linuxjournal.com/article/7767>, March 2005.
2. FlightLinux Project, <http://flightlinux.gsfc.nasa.gov/>
3. Datasheet: MCP750, <http://www.innovative-research.com/mcp750-spec.html>