FlightLinux: a Viable Option for Spacecraft Embedded Computers

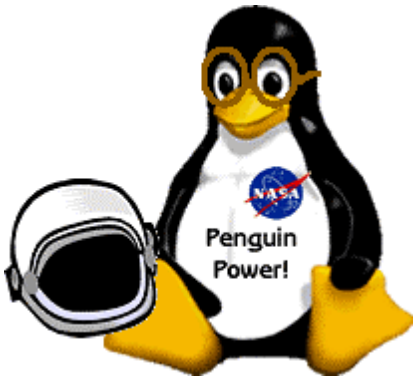
Patrick H. Stakem
Principal Investigator
QSS Group, Inc.
7404 Executive Place
Lanham, MD 20706

Abstract - The FlightLinux project has the stated goal of providing an on-orbit flight demonstration of the Linux operating system. This will result in a Technology Readiness Level (TRL) of 7. The FlightLinux proof-of-concept demonstration is being done in conjunction with the on-orbit UoSat-12 mission, from Surrey Space Technology, Ltd (SSTL). The Operating Missions as Nodes on the Internet (OMNI) project of Code 588 at the National Aeronautics and Spcae Administration's (NASA) Goddard Space Flight Center (GSFC) has a breadboard of the Surrey On-board computer (OBC), that is being used for testing. In addition, telecommunications facilities at GSFC allows direct communication with the UoSat-12 spacecraft.

This work was conducted under contract NAS5-99124-297, with funding by the NASA Earth Science Technology Office (ESTO) Advanced Information Systems Technology (AIST) Program, NRA-99-OES-08. The work was conducted by a team of QSS Group, Inc., with NASA/GSFC Codes 586 (Science Data Systems), 582 (Flight Software), and 588 (Advanced Architectures and Automation).

Because almost all of the effort in developing onboard computer hardware for spacecraft now involves adapting existing commercial designs, the logical next step is to adapt COTS software, such as the Linux operating system. Given Linux, many avenues and opportunities become available. Web serving and file transfer become standard features. Onboard LAN and an onboard file system become "givens." Java is trivial to implement. Commonality with ground environments allows rapid migration of algorithms from ground-based to the flight system, and tapping into the world-wide expertise of Linux developments provides a large pool of talent. Full source for the operating system and drivers is available on day one of the project.

We are currently wrapping up the integration testing of the FlightLinux port for the UoSat-12 onboard computer. We experienced some unanticipated non-technical problems with the "export" of our satellite comtrol software to our British partner, that we believe we have a viable work-around for. Testing and debugging on the breadboard proved to be more difficult than planned, due to lack of appropriate tools.



Since we posted our goals of keeping FlightLinux open source, within the meaning of the GNU license, we have had numerous offers of collaboration on the project. These include representatives of worldwide aerospace companies, and individuals. The interest in the FlightLinux Project is growing, due to increasing exposure of the website. We are aware of a dozen flight projects using Linux.A challenge for future projects of this nature will be to allow wider participation of the user communicty, within the bounds of a deliverable, verifiable working system.

An initial build of the FlightLinux software has been running since March 2001. The UoSat-12 breadboard has been available at the OMNI Lab since April 2001. The breadboard has an associated Windows-NT machine to load software via the Controller Area Network (CAN) bus or the asynchronous port and to provide debugging visibility. From our facility at QSS, we can access the breadboard facility via the

"PC-Anywhere" software using appropriate link security.

The initial FlightLinux software load is approximately 400,000 bytes in size. The nature of the UoSat-12 memory architecture at boot time limits the load size to less than 512,000 bytes. After the loader, which is Read-Only Memory (ROM)-based, completes, 4 megabytes of RAM memory is available. The software load includes 1) a routine to setup the environment and 2) a routine to decompress and start the Linux kernel. The kernel is the central portion of the operating system, a monolithic code entry. It controls process management, Input/Output, the file system, and other features. It provides an executive environment to the application programs, independent of the hardware.

Extensive customization of the SETUP routine, written in assembly language, was required. This routine in its original form relies on BIOS (Basic Input/Output System) calls to discover and configure hardware. In the UoSat configuration, there is no BIOS function, so these sections were replaced with the appropriate code. Sections of SSTL code were added to configure the unique hardware of the UoSat computer. The SETUP routine then configures the processor for entry to Protected Mode and invokes the decompression routine for the kernel. The SETUP routine is approximately 750 bytes in length and represents the custom portion of the code for the UoSat software port. This portion is usually referred to as the Board Support Package (BSP). The remaining code is the Commercial-off-the-Shelf (COTS) Linux software. This process is the same for any FlightLinux port. In fact, it is the same process for porting Linux to any architecture.

We modified the standard Linux SETUP routine, written in assembly language, to be Table-driven. This had the added advantage of addressing the export restriction issues, as the code structure, and the data, were separated.

The breadboard architecture includes an asynchronous serial port for debugging. We used this extensively for debugging the SETUP module. On the spacecraft, the asynchronous port exists, but it is not connected to any additional hardware.

FlightLinux is implemented in an incremental manner. The initial software build does a "Hello, World" aliveness indication via the asynchronous port and allows login. The synchronous serial drivers must be integrated to allow communication in the flight configuration. The bulk memory device driver, which uses the 32-megabyte modules of extended memory as a file system, will be added next. The breadboard has a single 32-megabyte module, and there are four modules in the flight configuration. The CAN bus drivers and the 10Base-T network interface will be added later.

Once we obtain 1) the permission to "export" the software and 2) the SSTL agreement to uplink, we can load a simplified "Hello, World" test kernel in the on-orbit spacecraft for testing. Additional modules can be uplinked later on an incremental basis.

Open-Source Usage

The FlightLinux Project is exploring new issues in the use of "free software" and open-source code, in a mission critical application. Open-source code, as an alternative to proprietary software, has advantages and disadvantages. The chief advantage is the availability of the source code, with which a competent programming team can develop and debug applications, even those with tricky timing relationships. The Open-Source code available today for Linux supports international and ad hoc standards. The use of a standards-based architecture has been shown to facilitate functional integration. It is a misconception that "free software" is necessarily available for little or no cost. The "free" part refers to the freedom to modify the source code.

A disadvantage of developing with Open Source may be the perception that freely downloadable source code might not be mature or trustworthy. Countering this argument is the growing experience that the Open-Source offerings are as good as, and sometimes better than the equivalent commercial products. What is needed, however, is a strong configuration control mechanism. For the FlightLinux product, the FlightLinux Team assumes the responsibility of keeping and maintaining the "official" version available.

Issues on the development and use of Open-Source software on government-funded and mission-critical applications are still to be explored.

Target Architectures

Various microprocessor architectures have been and are being adapted from commercial products for space flight use. For all of the primary architectural candidates we identified, Linux is available in COTS form. The primary hardware for flight computers in the near term are derived from the Motorola PowerPC family (RHPPC, RAD6000, RAD750), the SPARC family (EH32), the MIPS family (Mongoose, RH32), the Intel architecture (space flight versions of 80386, 80486, Pentium, Pentium-II, Pentium-III), and the Intel ARM architecture. Versions of FlightLinux for the PowerPC and MIPS family are important goals. Commercial versions of Linux are available for these architectures.

Given the candidate processors identified in missions under development and planned in the short term, we then examined the feasibility of Linux ports for these architectures. In every case, a Linux port was not only feasible, but is available COTS. Each needs to be customized to run on the specific hardware architecture configuration of the target board; ie, a board support package needs to be developed.

Existing space processors in recent or planned use include the RAD6000, the RH32, and the MIPS-derived Mongoose-V. Generally, Linux requires a Memory Management Unit (MMU) for page-level protection, as well as dynamic memory allocation. However, ports of Linux (uCLinux) exist for the Motorola ColdFire processor series and similar architectures, all without memory management. The Mongoose architecture does not include memory management hardware. A Mongoose port of Linux is feasible, and this has been examined by GSFC Code 582, Flight Software Branch. The future usage plans of these hardware architectures determines the direction of our efforts on the FlightLinux software ports.

Other emerging space processors include Honeywell's RHPPC, the Lockheed's RAD750, ESA's ERC32, and the Sandia radiation-hard Pentium. All are viable targets for FlightLinux. The RHPPC and the RAD750 are variations of the Motorola PowerPC architecture. GSFC Code 586 already has Linux running on the PowerPC architecture, in a laboratory environment. The Intel (Pentium) version of Linux is the most common, and can be found in the Code 586 tech lab as well. ESA's ERC32 is a variation on the SPARC architecture, and Linux is available for the Sun Sparc architecture. The term COTS in this context should be taken to mean that a commercial version for that processor architecture is available. A specific port for the Flight Computer embedded board involves coding specific device drivers. Linux is a 32-bit operating system, appropriate for matching the emerging 32-bit class of flight computers.

POSIX

POSIX is an IEEE standard for a Portable Operating System based on Unix. The use of a POSIX-compliant operating system and applications has many benefits for flight software. Among these benefits are software library reuse between missions and software commonality between ground and flight platforms. For compliant code, the function calls, arguments, and resultant functionality are the same from one operating system to another. Source code does not have to be rewritten to port to another environment. Linux variants are mostly, but not completely, POSIX-compliant. The POSIX standards are now maintained by an arm of the IEEE called the Portable Applications Standards Committee (PASC) with the associated web site http://www.pasc.org/.

POSIX compliance is certified by running a POSIX Test Suite, available from the National Institutes of Standards and Technology (NIST). We have pursued the IEEE POSIX compliance issues of standard embedded Linux, in parallel with an effort in GSFC Code 582, which has collectied a library of POSIX-compliant flight applications software. FlightLinux also enables the implementation of the Java Virtual Machine, allowing for the up-link of Java applets to the spacecraft.

The advantages of Linux are numerous, but the requirements for spacecraft flight software are unique and non-forgiving. Traditional spacecraft onboard software has evolved from being monolithic (without a separable operating system), to using a custom operation system developed from scratch, to using a commercial embedded operating system such as VRTX or VxWorks. None of these approaches have proved ideal. In many cases, the problems involved in the spacecraft environment require access to the source code to debug. This becomes an issue with commercial vendors. Cost is also

an issue. When source code is needed for a proprietary operating system, if the manufacturer chooses to release it at all, it is under a very restrictive non-disclosure agreement, and at additional cost. The Linux source is freely available to the team at the beginning of the effort, and this can provide a significant advantage. It can also, if not well managed, lead to a lot of tinkering that is not mission-focused.

As a variation of Linux, and thus Unix, FlightLinux is Open Source, meaning the source code is readily available and free. FlightLinux currently addresses soft real-time requirements and is being extended to address hard real-time requirements for applications such as attitude control and telemetry formatting. There is a world-wide experience base in writing Linux code that is available to tap.

The use of the FlightLinux operating system simplifies several previously difficult areas in spacecraft onboard software. For example, the FlightLinux system imposes a file system on onboard data storage resources. In the best case, Earth-based support personnel and experimenters may network-mount onboard storage resources to their local file systems. The FlightLinux system both provides a path to migrate applications onboard and enforces a commonality between ground-based and space-based resources.

Linux is not by nature or design a real-time operating system. Spacecraft embedded flight software needs a real-time environment in most cases. However, there are shades of real time, specified by upper limits on interrupt response time and interrupt latency. We can generally collect these into hard real-time and soft real-time categories. Examples of hard real-time requirements are those of attitude control, spacecraft clock maintenance, and telemetry formatting. Examples of soft real-time requirements include thermal control, data logging, and bulk memory scrubbing.

Unix, and Linux, were not designed as real-time operating systems, but do support multi-tasking. Modifications or extensions to support and enforce process prioritization are necessary to apply Linux to the embedded real-time control world.

In one model, a process may yield the CPU to another pending task. In a preemption scheme, a running process is stopped, and a pending process is started. In another scheme, time slicing, a "round-robin" priority scheme allows equal access to all tasks, or a variation, with a high-priority and a low- priority queue. It is generally agreed that a preemptive scheduling scheme allows for greater concurrency in a real-time system. Beyond the process-switching scheme is the interrupt prioritization. Here, we mean asynchronous interrupts from external sources. Interrupt prioritization is determined and enforced by the hardware configuration. Also, interrupt servicing supersedes software process execution in general.

Problems originate from the fact that the traditional Unix or Linux kernel is a monolithic entity that governs process prioritization. Interrupt drivers and the kernel itself do not participate in the prioritization scheme. The kernel typically has large stretches of non-preemptible code. This is necessarily in the design so that data structures can be modified in an atomic fashion. In a Linux kernel, all interrupt handlers run at a higher priority than the highest-priority task. In the Unix view, the kernel is the top level and most important task. In the real-time control world, this is not necessarily true. Patches to the kernel have been developed to address this.

One approach to correcting this is to implement a threaded execution approach for the kernel and the interrupt handlers. The question arises as to how much the Linux kernel can be modified and still be referred to as a Linux kernel. Another approach is to treat the kernel itself as a scheduled task, under a Real Time Task Manager that manages process prioritization and takes over control of interrupts. This has been referred to as kernel cohabitation.

Many real-time schedulers for Linux are available for download. These are a Rate Monotonic Scheduler, which treats tasks with a shorter period as tasks with a higher priority, and an Earliest Deadline First (EDF) scheduler. Other approaches are also possible. It is not clear which approach provides the best approach in the spacecraft-operating environment. This will be investigated and tested as part of our effort.

The Bulk Memory Device Driver

Spacecraft onboard computers do not usually employ rotating magnetic memory for secondary

storage. Initially, magnetic tape was used, but now the state of the art is to use large arrays of bulk Dynamic Random Access Memory (DRAM), with various error detection and correction hardware and/or software applied..

The current state of the art of spacecraft secondary storage is bulk memory, essentially large blocks of DRAM. This memory, usually still treated as a sequential access device, is mostly used to hold telemetry during periods when ground contact is precluded. Bulk memory is susceptible to errors on read and write, especially in the space environment, and needs multi-layered protection such as triple-modular redundancy (TMR), horizontal and vertical Cyclic Redundancy Codes (CRC), Error Correcting Codes (ECC), and scrubbing. Scrubbing can be done by hardware or software in the background. The other techniques are usually implemented in hardware. With a Memory Management Unit (MMU), even using a default 1:1 mapping of virtual to physical addresses, the MMU can be used to re-map around failed sections of memory.

Although we usually think of bulk memory as a secondary storage device with sequential access, it may be implemented as random access memory within the computer's address space. This is the case with UoSat-12.

The Flash File System (FFS) has been developed for Linux to treat collections of flash memory as a disk drive, with an imposed file system. Although we are dealing with DRAM and not flash, we can still gain valuable insight from the FFS implementation. In addition, the implementation of Linux support for the personal computer memory card international association (pcmcia) devices provides a useful model.

The onboard computers on the UoSat-12 spacecraft has 128 megabytes of DRAM bulk memory. It is divided into four banks of 32 megabytes each, mapped through a window at the upper end of the processor's address space. This is the specific device driver that we use as a model for future development of similar modules. The current software of the UoSat-12 onboard computer treats this bulk memory as paged random access memory and applied a software scrubbing algorithm to counter environmentally induced errors. The program memory is protected by hardware TMR.

The ram disk is a disk-like block device implemented in RAM. This is the correct model for using the bulk memory of the onboard computer as a file system. Multiple RAM disks may be allocated in Linux. The standard Linux utility "mke2fs," which creates a Linux second extended file system, works with RAM disk, and supports redundant arrays of inexpensive disks (RAID) level 0.

The RAID model was developed to use large numbers of commodity disk drives combined into one large, fault-tolerant, storage unit. The approach can be applied to bulk memory as well. RAID can be implemented in software or hardware. For the purposes of this document, we consider RAID software implementations. Software RAID is a standard Linux feature, available as a patch to the 2.12 Kernels and an included feature in Kernel 2.4.

This initial version of the driver uses memory mirroring, with memory scrubbing techniques applied. In the simplest case, we treat three of the four available 32-megabyte memory pages as a mirrored system. The memory scrubbing technique is derived from the current scheme used by SSTL, as is the paging scheme. The next version of the driver uses all four of the available 32-megabyte memory pages with distributed parity. The performance with respect to write speed is expected to be less than with the Level 0, but the memory resilience with respect to errors is expected to be much better.

It is unclear without further testing whether the RAID technique will be sufficient to counter the environmentally-induced errors expected in the bulk memory on-orbit. It is generally accepted that RAID is not intended to counter data corruption on the media, but rather to allow data recovery in case of media failure. A defined testing approach will be used with the bulk memory device driver on the breadboard facility. More extensive testing on-orbit with the UoSat-12 spacecraft is required to validate the approach.

Onboard LAN

Given that the Linux operating system is onboard the spacecraft, support for a spacecraft LAN becomes relatively easy. Extending the onboard LAN to other spacecraft units in a constellation also becomes feasible, as does having the spacecraft operate as an Internet node.

Interface between spacecraft components is usually provided by point-to-point connections, or a master/slave bus architecture. The use of a LAN onboard is not yet common. This is partially due to the lack of space-qualified components.

The avionics bus MIL-STD-1553 and its optical derivative, 1773, are commonly used between spacecraft components. This bus, used in thousands of military and commercial aircraft has a legacy of applications behind it. Also, 1553 is transformer-coupled and dual-redundant, providing a level of failure protection. The raw data rate is 1 megabit-per-second. It is a master/slave architecture.

For point-to-point connections that do not require the complexity of a 1553/1773 connection, a synchronous serial connection such as RS-422/23 with a bit rate of approximately 1 megabit-per-second is typically used.

A LAN-type architecture is typically used in office and enterprise environments (and spacecraft control centers). It provides a connection between peer units, or clients and servers. The typical LAN uses a coax or twisted pair connection at a transmission rate of 10 megabits per second, a twisted pair connection at 100 megabits per second, or optical at 155 megabits per second, with higher speeds possible. Wireless systems, such as 802.11, are also being widely deployed.

Usually, a LAN is configured with a repeating hub, a central switch, or a router between units. The standard protocol imposed on the physical interface is Transmission Control Protocol /Internet Protocol (TCP/IP), although others are possible (even simultaneously). The TCP/IP protocol has become a favored approach to linking computers around the world. The protocol is supported by Linux and most other operating environments.

The UoSat-12 configuration allows us to exercise the TCP/IP and CAN bus components of an onboard LAN. Evolving physical layer interfaces for use onboard the spacecraft include 100 megabit ethernet and Firewire (IEEE-1394).

For space-to-ground communication and vice-versa, FlightLinux utilizes the pioneering IP-in-space work validated by GSFC Code 588's OMNI Project.

FlightLinux begets FlightBeowulf

Having a Linux system enables a plethora of software that runs under Linux. One package is the NASA-developed Beowulf software which implements a cluster of Linux machines. This may use several machines colocated in one satellite, or multiple satellites in a constellation linking their computational resources via inter-satellite communications.

An approach to increase computational speed by the use of a multiprocessor system, dubbed a “Flight Beowulf,” involves the case where an original problem (application) is split into parts (processes) that are acted upon by separate processors simultaneously. Ideally, given ‘n’ processors, the increased computational speed would be almost ‘n’ times that of a single processor. At any given point in the technology maturity curve, we can apply this technique to multiply the onboard processing power available. Of course, we must be observant of the impacts to other mission parameters such as power, weight, heat production, and size. The advantage is the super processing power delivered upon demand.

This approach has been implemented and validated numerous times in a ground based environment. This lays the cornerstone for space-based cluster computing. Such a flight computer has the raw processing power capable of managing a complex scenario of sophisticated event detection and data mining algorithms for multiple instruments in a real-time, on-board environment.

We believe that Linux is a viable choice for flight computer operating systems. It has the potential to become the basis of a large variety of spacecraft missions, while reducing cost of development and maintenance. At the same time, the use of open source software for mission critical applications is in its infancy, and a new set of management controls must be evolved. However, this might be the basis for a new paradigm of onboard software, that leads to a revolution in computing techniques for NASA.
We can't afford to ignore this option, currently running on a variety of paltforms, from watches to mainframes, worldwide.

REFERENCES

[1] FlightLinux Project Website
http://FlightLinux.gsfc.nasa.gov

[2] OMNI Project Website
http://ipinspace.gsfc.nasa.gov/

[3] Beowulf website
http://www.scyld.com/