UNIX-COMPATIBLE REAL-TIME ENVIRONMENT FOR NASA'S GROUND TELEMETRY DATA SYSTEMS

Ward Horner
Data Systems Technology Division
Code 520

Charles Kozlowski
RMS Technologies, Inc.
Code 520.9

Mission Operations and Data Systems Directorate
NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771

KEYWORDS

UNIX-compatible, Real-time, Software, Telemetry System, Multiprocessor

ABSTRACT

NASA's ground telemetry data systems developed by the Microelectronics Systems Branch at the Goddard Space Flight Center, use a generic but expandable architecture known as the *"Functional Components Approach."* This approach is based on the industry standard VMEbus and makes use of multiple commercial and custom VLSI hardware based cards to provide standard off-the-shelf telemetry processing functions (e.g., frame synchronization, packet processing, etc.) for many telemetry data handling applications. To maintain maximum flexibility and performance of these systems, a special real-time system environment has been developed, the Modular Environment for Data Systems (MEDS). Currently, MEDS comprises over 300,000 lines of tested and operational code based on a non-UNIX real-time commercial operating system. To provide for increased functionality and adherence to industry standards, this software is being transformed to run under a UNIX-compatible real-time environment. This effort must allow for existing systems and interfaces and provide exact duplicates of the system functions now used in the current real-time environment. Various techniques will be used to provide a relatively quick transition to this new real-time operating system environment. Additionally, all standard MEDS card to card and system to system interfaces will be preserved, providing for a smooth transition and allowing for telemetry processing cards that have not yet been converted to reside side-by-side with cards that have been converted. This paper describes this conversion effort.

INTRODUCTION

The Microelectronics Systems Branch at the Goddard Space Flight Center is transitioning from a non-UNIX self hosted real-time software environment to a UNIX-compatible workstation hosted real-time environment over the next year. This environment offers the software developer a much more robust set of development and debug tools for real-time software design. This paper outlines the transition plan in detail so that programmers and managers alike will have a thorough understanding of what is involved in such a transition.

The new development environment is a set of UNIX workstations that act as a software front-end to the real-time target system. A complete set of compilers, linkers and source level debuggers is available to the software developer under the new environment that were previously unavailable or available only as primitive tools. One of the most robust features of the new environment is the TCP/IP based socket concept that is borrowed from the UNIX domain. This allows the programmer who has previously worked in a UNIX environment a smooth transition to the new real time environment. The target machine is connected to the host development machine through an Ethernet cable. Programs are loaded and debugged from a host workstation to the target machine using the TCP/IP protocol.

The new real-time target environment allows the developer to choose from 680x0, 683xx, SPARC, and 80960 based target architectures. This offers a great advantage over the old real-time target environment that only supported 680x0-based architectures.

The new software development environment allows the developer to choose from a number of workstations as development hosts. This offers a great advantage over the old software development environment that was self hosted.

The compiler, linker, and source-level debugger are written and freely distributed by Free Software Foundation. This means that source code is available for all of these tools. These tools are well documented by both on-line documentation and hard copy documentation available to the software system developer.

The first phase of moving from the old to the new environment is to support the current software environment shell. The primary shell that ran on top of the old operating system was the Modular Environment for Data Systems (MEDS). In order to support the existing inter-process communication and data structures, it is necessary to transfer all of the current MEDS functionality into the new environment. This migration is already underway and is outlined in the following paragraphs.

CURRENT SOFTWARE ENVIRONMENT

The telemetry data systems designed, built, and programmed in the Microelectronics Systems Branch all have a similar pipelined, multiprocessor, dual bus, hardware architecture as a platform on which to build application specific hardware and software .

The Modular Environment for Data Systems (MEDS) is designed as a general purpose software shell that can be expanded and customized by application programmers to suit their particular requirements. It provides the basic software functions needed in multiprocessor telemetry data systems, namely, the ability to setup application specific hardware and software, process the telemetry data based on the setup parameters, monitor the processing, and supply network support for remote operator interface and data transfer. MEDS supplies an infrastructure to pass data between systems, processors and tasks as well as support for operator interface development.

A complete system is built by adding custom code to the general purpose MEDS code. Therefore, MEDS spares the application developer from the burden of creating an infrastructure for each new system and adds consistency in all system design, implementation and maintenance. With MEDS software, application specific real-time code has a strong modular foundation on which to build.

A MEDS based system unites and manages the standard multiple processor hardware platform. The processors are organized as a single master processor directing multiple subordinate application specific custom cards. The master processor is the single point of control within the system; it interfaces with the operator, on either a local terminal or a remote workstation. Using a set of operator defined setup files, the master processor will initialize the custom cards and monitor their processing on various status pages. Telemetry data may enter and exit the system through the remote interface as well. In any case, it is the pipeline of custom cards that process the telemetry data.

MEDS software resides on the master processor and on each custom card. The basic MEDS functions include:

- Setup system and subsystems for processing (e. g. setup custom chip registers).
- Control the application specific processing (e.g. enable, disable, reset a card).
- Monitor the system and subsystems (e.g. gather and display card processing status).

- Stream data over network (e.g. transfer telemetry data to/from a workstation).

The overall MEDS software design is modularized into packages that supply general purpose system functions such as operator interface support, status gathering support, command handling support, interprocessor communications, and network communications. Each package implements a set of functions that serve as a resource to the application software while hiding the details of their implementation. Consistent interfaces have been defined for each package so code within a package can be changed without affecting the application code if the functions and interfaces remain constant.

A MEDS based system is a group of cooperating tasks built on these packages. The tasks are spread across the processors of the system, both tightly coupled (on the VMEbus) and loosely coupled (on a network between the VME rack and remote workstation). Some of the packages exist as linkable libraries that the programmer will use to build a task such as a command handler. Other packages exist as customizable source code files, where the applications programmer will copy the source file, add in his specific code and compile, such as a status task. In all cases, the programmer does not need to know the details of the MEDS code only the interface to it. All processors in a MEDS system run the real-time operating system.

CONVERSION APPROACH

Currently, MEDS comprises over 300,000 lines of tested and operational code based on a non-UNIX real-time commercial operating system. The size and complexity of this porting effort combined with the need to maintain existing deployed systems, requires that existing MEDS interfaces preserved. All standard MEDS card-to-card and system-to-system interfaces will remain the same. This will provide for a smooth transition from the old operating system to the new by allowing existing telemetry processing cards that have not yet been ported to reside side-by-side with cards that have.

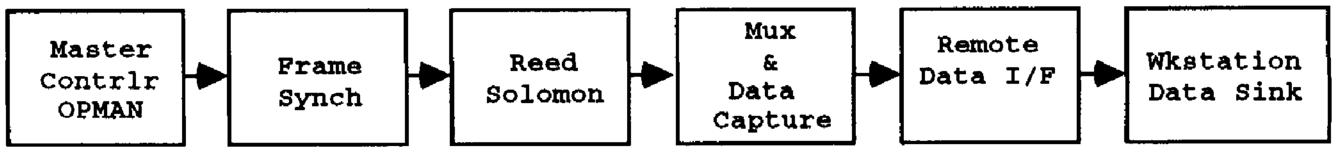
Second, most of the current MEDS code will be reused. Code that does not work under the new operating system will be left in the source code, but removed from compilation using conditionals. For the initial release, additional features provided under the new operating system will only be used if necessary. Source code backward compatibility will be maintained during the conversion process. Also, errors found by testing under the new operating system will be backannotated to systems using the old operating system.

Finally, a separate development team has been formed. This team is trained in use of both the old and the new operating systems. In order to make this porting project more visible, it has been tied to an existing project and must meet the needs and schedule of that project as well as it's own.

The remainder of this paper outlines the specifics of the conversion approach.

Development plan

The implementation plan has been broken down into several builds. The first build, demonstrated in August 1993, consists of the basic telemetry system components (Frame Synchronizer, Reed-Solomon Decoder, Multiplexer, Data Capture, and Remote Interface), master controller routines including the VT100 or terminal emulator based user interface (OPMAN), and all MEDS library routines. The remote interface will be used to transfer data to a workstation and not for remote commanding.



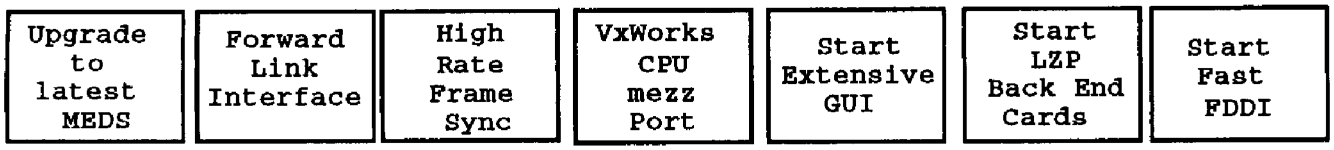
Build I release (August 1993)

Build II, demonstrated in December 1993, will add more cards (Sorter Card, Data Buffer, and Slow FDDI using commercial drivers) and a simple generic graphical user interface running on the workstation. Also, Test Data Generator conversion will be started.

Simple generic GUI
Sorter
Data Buffer
Slow FDDI
Start Test Data Generator

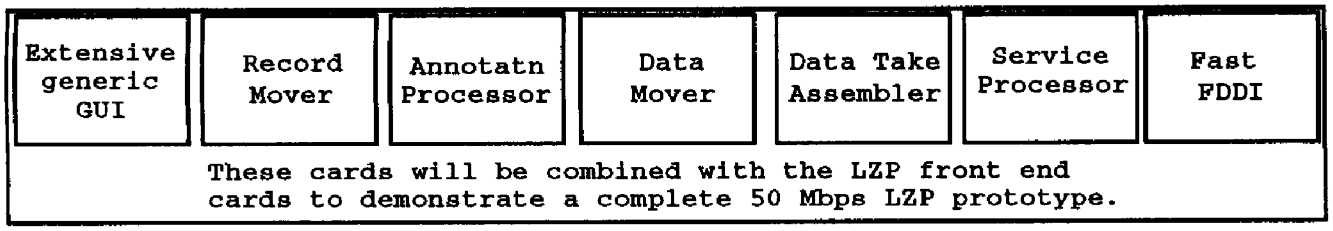
Build II release (December 1993)

Build III, demonstrated in April 1994, will add two more cards (Forward Link Interface and High Rate Frame Sync). Additionally, the new operating system will be ported to an internally designed CPU mezzanine. All currently ported cards (Master Controller, Frame Sync, Reed-Solomon, Mux, Remote Interface, Sorter, Data Capture, Data Buffer, Slow FDDI) will be brought up to the latest MEDS release. Porting the Level Zero Processor (LZP) back end cards (Record Mover, Annotation Processor, Data Mover, Service Processor, and Data Take assembler) will be started. An extensive generic graphical user interface will be started. Also, a fast FDDI interface based on custom UDP/IP and drivers will be started.



Build III release (April 1994)

Build IV, demonstrated in Summer 1994, will add the LZP back end cards (Record Mover, Annotation Processor, Data Mover, Service Processor, and Data Take assembler), a more extensive generic graphical user interface, and a fast FDDI interface based on custom UDP/IP and drivers. This build will provide UNIX-compatible software for the 50 Mbps LZP system.


Extensive generic GUI
Record Mover
Annotatn Processor
Data Mover
Data Take Assembler
Service Processor
Fast FDDI
These cards will be combined with the LZP front end cards to demonstrate a complete 50 Mbps LZP prototype.

Build IV release (Summer 1994)

Emulation Library

In order to ease the transition from one operating system to another, an emulation library has been designed. This library simulates the commonly used operating system unique subroutine calls. This library has been designed into 6 major subelements; 1) real-time task control calls, 2) character I/O, 3) system calls, 4) date/time calls, 5) file I/O calls, and 6) miscellaneous support calls.

By using this library, approximately 80% of the previously developed code ports to the new UNIX-compatible operating system with no changes. Had this library not been developed only about 40% of the source code could have ported with no change.

The other important aspect of this library is that it allows for simple and almost trivial porting from the new UNIX-compatible real-time operating system to any other UNIX-compatible real-time operating system. All of these systems have operating specific calls. By merely mapping these calls in the emulation library, porting becomes much simpler .

Conditional Compilation

Conditional compilation is used to further isolate operating specific code. For example, the old operating system used an odd file naming convention. By using conditionals around the filenames, both the old and the new names are supported.

Porting Guides

Early in the transition process, several porting guide documents were developed. These guides cover both the major design differences as well as the detailed coding modifications necessary. Additionally, since there is still significant design and coding activity around the old real-time operating system, a portability document was written to show how to write code that is easily ported to the new operating system.

OPERATING SYSTEM DIFFERENCES

This section highlights a few of the issues and differences that were encountered during the porting process.

File Naming

The old operating system uses uppercase characters for file names and the colon for extensions. The new operating system uses standard lower case characters for file names and the period for extensions. To preserve backward portability to old operating system, the C conditional macro "#ifdef" is used to conditionally include files and code. Using a conditional macro, system specific include files and code are selected without modifying the program module.

All operating system specific code is placed in conditional compilation statements as follows:

```
    #ifdef OLD_OS

    # include "HEADER:H"
    # define FILE_NAME "FILE:MAP"

    #else

    # include "header.h"
    # define FILE_NAME "file.map"

    #endif
```

Prototypes

The old operating system used a standard K & R compiler and mostly ignored prototypes. It does, however, use the return type for the function. Programmers have been encouraged to use prototype definitions, even under the old operating system. Because of this, the new operating system flagged a number of prototype errors, that were previously ignored. To be compatible with the old operating system, function declarations will still use the old K & R format rather than the new ANSI format. Part of the conversion process includes adding prototypes where necessary.

The old operating system doesn't totally ignore (although it doesn't use) the rest of the code in the argument declaration area of the prototype. It can't handle actual argument names in the prototypes. There is a few keywords and formats that it can't handle. For instance, it can't handle any typedefs in prototypes and it can't handle some qualifiers such as "register" in prototypes.

For Example:

ANSI Prototype with inline argument declaration:

```
    void ch_clock(int) ;
```

K+R Style Function Definition:

```
    void ch clock (which)
    int which;
    {
    ...
    }
```

Bit Fields

One of the trouble spots during the conversion occurs on programs that use bit field structures. Under the old operating system, structures are reversed end-for-end compared to the new operating system. The programmer must manually revise all bit field structures.

The new operating system performs all bit field operations using byte width instructions. This is not a problem when the memory is byte addressable; however, some control devices allow only word and long word width addressing. To solve this problem, a macro was devised that maps the bit field operation into a masking operation. The register is read into regular memory, the bit field operation is performed, and register is then updated with the new value. This operation requires that the register be both readable and writable.

Linker/Loader

The new operating system uses a dynamic linking loader to download and resolve software at load time. Programs do not have to be fully linked with all parts to be downloaded and executed. In fact, it is desirable that the software module not be fully linked. The linking process occurs automatically while it is being downloaded from the host system. All modules that are being referenced must have been previously downloaded and be present in memory.

This design provides an efficient software context, because code modules are reused by many callers. However, it does have its drawbacks, which is true for all reentrant software. The use of local static variables and non-static global variables must be closely examined. These are variables that hold their value from call to call. The problem occurs when a task calls a shared subroutine that contains a static variable and changes its value to reflect a task specific value. When another task calls the shared subroutine, the static variable will be incorrect for its context. The new operating system provides a facility to save these variables with the task context; however, the task scheduling is slowed as a result. Therefore, this issue must be examined carefully and an appropriate action taken to resolve it.

CONCLUSION

The UNIX environment provides a powerful and open environment. This environment offers the software developer a generic robust set of development and debug tools for many types of software design on many different systems. This paper has outlined the conversion of a software environment from a non-UNIX to UNIX-compatible real-time operating system.

The transition of software from one operating system to another can be time consuming, complex, and error prone. There are many approaches that could be taken to achieve such a transition. By reusing existing interfaces and code, the Microelectronics Systems Branch has chosen a path that is thought to minimize the conversion time, complexity, and error.