Development of Real-Time Software Environments for NASA's Modern Telemetry Systems

DEVELOPMENT OF REAL-TIME SOFTWARE ENVIRONMENTS FOR NASA S MODERN TELEMETRY SYSTEMS

Ward Horner, Steve Sabia
Data Systems Technology Division, Code 520
Mission Operations and Data Systems Directorate
NASA, Goddard Space Flight Center
Greenbelt, Maryland 20771

ABSTRACT

The design and development of generic, low cost, high performance telemetry components and systems require the optimum integration of custom and standard hardware elements with a number of real-time software elements. To maintain maximum flexibility and performance for Goddard Space Flight Center’s VLSI telemetry system elements, two special real-time system environments were developed. The Base System Environment (BaSE) supports generic system integration while the Modular Environment for Data Systems (MEDS) supports application specific development. Architecturally, the BaSE resides just on top of a commercial real-time system kernel while the MEDS resides just on top of the BaSE. The BaSE provides for the basic porting of various manufacturer’s cards and insures seamless integration of these cards into the generic telemetry system. With this environment, developers are assured a rich selection of available commercial components to meet their particular application. The MEDS provides the designer with a set of tested generic library functions that can be employed to speed up the development of application specific real-time code. This paper describes the philosophy behind the development of these two environments and the characteristics which define their performance and role in a final VLSI telemetry system.

INTRODUCTION

Modern real-time telemetry data systems needed to support NASA in the 1990’s and beyond, require the use and application of state-of-the-art real-time software techniques and approaches that are tightly coupled with high performance VLSI based hardware systems. To this end, Goddard Space Flight Center’s (GSFC) Data Systems Technology Division maintains expertise in both high performance VLSI based hardware design and state-of-the-art real-time multiprocessing system software design and techniques(l)(2)(4). This knowledge base is constantly being expanded and augmented by keeping abreast of the latest commercial hardware processing elements and systems; exploring and exploiting these technologies as they become available; and maintaining an active on-going dialog with commercial hardware and software system manufacturers.

With the push toward an automated data driven operation of NASA's next generation telemetry data handling systems, it is important that standard system functions be virtually turnkey in every aspect of their operation. Even though these systems are a tightly integrated mix of hardware and software elements, they are but single elements in a large, highly complex NASCOM telemetry data handling system; therefore, these systems must act as fully automated *black box* components.

Using the Base System Environment (BaSE) software package, system interactions between various manufacturer's cards are prototyped, explored, and tested before being placed into operational use. Using the Modular Environment for Data Systems (MEDS) software package, application specific real-time code has a strong modular foundation that begins with a generic multiprocessing shell and supports the basic software functions needed by all multiprocessor based telemetry data systems (see figure #1). Because of the dynamic nature and complexity of these high performance telemetry data systems, designated system experts with first hand experience, port BaSE and MEDS to the next generation telemetry systems and perform detailed anomaly analysis and system level education.

INTRODUCTION To BASE SYSTEM ENVIRONMENT

To maintain their state-of-the-art aspects, NASA's next generation telemetry systems must provide a fairly simple and fast path to future enhancements. For this reason, the electronics hardware is based around the widely supported VMEbus. Likewise, a versatile software environment, that could support the flexibility of the VMEbus, was also needed. There are many demands on this environment, but most important it must contain enough intelligence to automatically configure the system based on a changing hardware environment. It is the goal of BaSE to provide this type of functionality.

BaSE is more than just a piece of generic software or hardware, it is a developmental and operational philosophy. It is a means to easily allow the use of the very finest products that the VMEbus has to offer. It allows the designer to pick and chose those VMEbus based products that best fit the application at hand. It ensures that products from different manufacturers can be used together in a "plug and play" fashion. It allows seamless integration of NASA's custom VMEbus based telemetry processing cards into the operational environment. By providing a single environment that is used for all phases of system development, the use of BaSE spans from initial hardware test and checkout, to system software development, to final operational system deployment. The goal was to

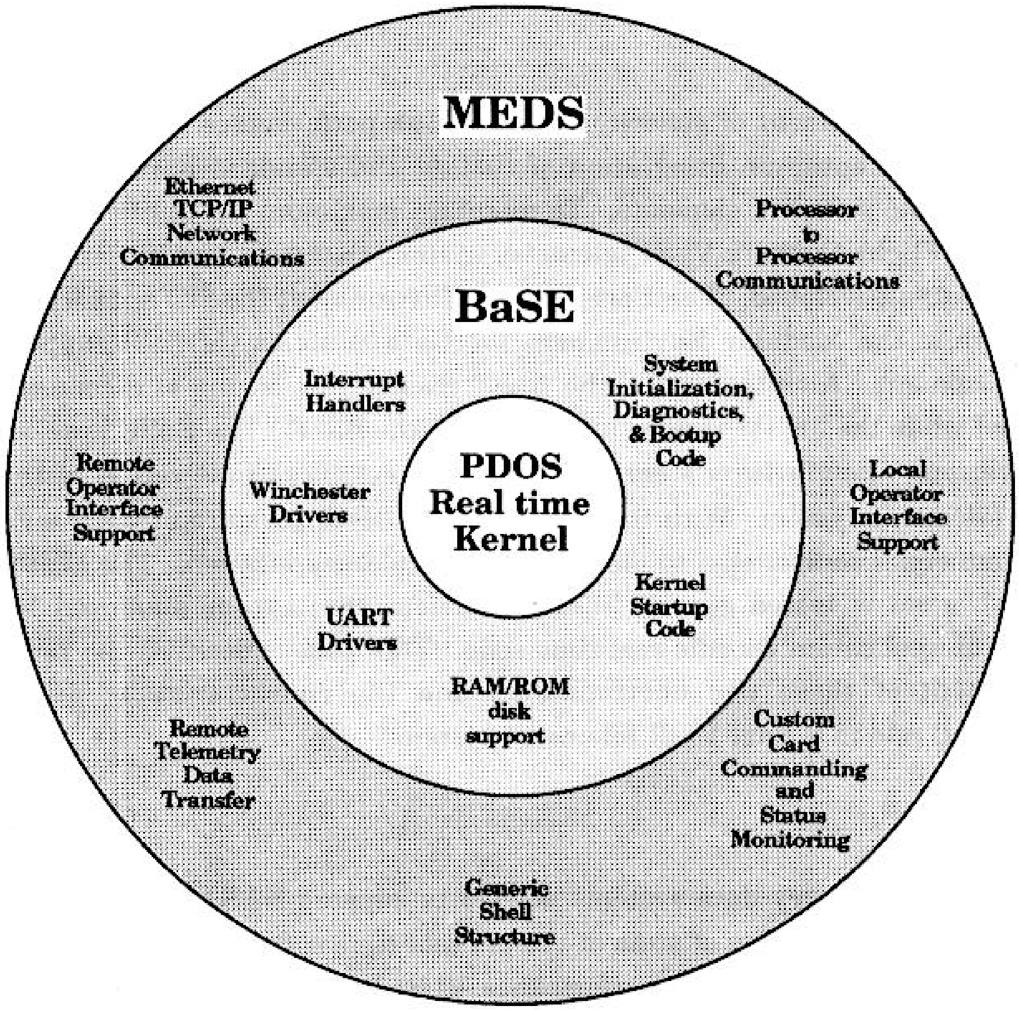
MEDS
Ethernet TCP/IP Network Communications
Processor to Processor Communications
BaSE
Interrupt Handlers
System Initialization, Diagnostics, & Bootup Code
PDOS Real time Kernel
Remote Operator Interface Support
Winchester Drivers
Local Operator Interface Support
Kernel Startup Code
UART Drivers
RAM/ROM disk support
Custom Card Commanding and Status Monitoring
Remote Telemetry Data Transfer
Generic Shell Structure

FIGURE 1. THE REAL-TIME SOFTWARE ENVIRONMENT

develop a cost effective telemetry platform that is generic in both hardware and software, can be used for both development and operations, and can be quickly and easily tailored to meet changing system needs.

GENERAL DESCRIPTION OF BaSE

BaSE works in conjunction with an unmodified commercially available real-time, multiuser, multitasking kernel and provides both a complete development environment and an identical operational environment. This multitasking kernel lies at the core of the development system and is used to build the system software. This same kernel lies at the core of the operational system, thus ensuring that both environments are identical. This dual purpose environment allows the system designer to develop the application directly on the target system. The use of BaSE eliminates the need for complex and expensive host software development workstations(5) and microprocessor emulators.

Each CPU card on the VMEbus carries its own copy of the operating system in ROM, thereby allowing the card to run stand-alone with or without a disk drive or network connection. The BaSE allows these multiple CPU cards to use VMEbus shared resources just as if they were the only CPU card in the system.

BaSE can be ported to virtually any Motorola MC68010/20/30 based CPU and its supporting peripheral equipment (e.g. disk drives) whether it be a commercial card, or imbedded in a custom application. It is quickly ported (within several days) to new more powerful hardware as it becomes available. The porting process is simplified because unique card specific code has been reduced to mere 10% of the overall BaSE code. A utility program has been developed to simplify the system generation of BaSE and subsequent ROMing that allows object oriented modifications to the default setup without changing the source code.

In the development environment, all application specific code resides on the system disk module, this includes the system startup code, MEDS, Ethernet TCP/IP network interfaces (3), utilities, etc. In the operational environment, application specific code can reside on a *ROM disk* that is placed either locally in the BaSE ROM or globally in a separate ROM disk card. In either mode, the application code is automatically started after bootup. The use of a ROM disk ensures that both the development and operations environments are identical. Plus, it alleviates the need for often tedious and complex *run module generation*(8), where the application tasks are intimately linked together with a real-time kernel into a single module.

BaSE is supported on custom systems with multiple imbedded CPUs. For example, a GFSC developed card called the *VLSI High Speed Packet Processor*(7) uses a ring of three MC68020/030 processors along two CMOS gate arrays. Each CPU runs its own private BaSE, and through the use of dual ported RAM disks and terminal ports, each CPU has its own independent development environment. BaSE is also used in an imbedded mode in the GSFC *Transportable Telemetry Workstation*(6). This system consists of a Macintosh II equiped with several custom NUbus cards that perform various telemetry processing functions. One of the custom NUbus cards supports two MC68020/030 processors both running BaSE. This card has a shared SCSI port for hard disk interface and two terminal ports for user interface allowing application specific system development and monitoring during the prototype stage directly on the target NUbus card.

OPERATING SYSTEM: RESOURCES

The real-time operating system that is used as the core for BaSE provides one of the highest performance real-time kernels available for the MC680x0 family. Other real-time operating systems were not suitable for this environment; they were either too slow, too large, or too expensive. Many did not provide target system development, the necessary multiprocessing tools, or the tight hardware coupling and versatility that was required. Again, the idea was not to develop a “one of a kind” system, but instead a cost effective generic platform that can be tailored to meet changing system requirements.

The complete real-time operating system has three parts: the real-time kernel, the Basic Input/Output System (BIOS), and the utility programs. The fully relocatable real-time kernel is supplied in object code form (source code may be purchased separately) and is fully generic across any MC680x0 processor. BIOS includes code for basic bootup, drivers, and I/O devices, for many VMEbus based CPU cards and peripherals. This code represents the portion of the operating system that is unique to a particular VMEbus computer system. It is supplied as source code so that the system developer can upgrade, modify, or tailor it to meet the need. The utilities include compilers, assemblers, linkers, editors, debuggers, etc.

The real-time operating system provides a basic set of tools, many of which are used by both BaSE and MEDS, that can be used effectively to build a multiple processor system. These tools support: global events, global memory installation, global message passing. Global events are used to synchronize tasks on different processors. The basic functions can set, clear, and test global event flags. Other functions include delayed set/clear and task suspension. Global memory installation tools are used to install non-tasking memory for use in global message passing. The basic functions can allocate, initialize, lock, and unlock a global memory area. Global message passing tools utilize prioritized queues for interprocessor/intertask message services. The basic functions can put and get messages. Other functions can create, delete, and initialize message queues.

ENHANCEMENTS TO BIOS

The original BIOS is designed for an environment in which all cards are from the same manufacturer and there is only a single CPU board on the VMEbus. Integrating and porting other manufacturer’s cards could be done with the original BIOS, but it proved to be custom in its approach and was not easily adapted to NASA’s generic systems concept. A BIOS was needed that was common on all CPUs used in the system, regardless of the manufacturer. A number of standard BIOS code modules supplied for different manufacturer’s CPU cards were analyzed for common code and potential generic pieces. This common/generic code was stripped out and placed into common code modules. Initially, the resulting unique code amounted to about 20% of the total BIOS code. This process continued and modifications were made through the use of CPU specific parameter files to modify the way the code was assembled leaving only about 10% of the total BIOS code as unique. The new common BIOS code was then enhanced to support the generic telemetry platform environment. Furthermore, the card specific code was modularized and formatted so that even the modules of unique code were identical in structure. The real-time kernel is not modified in any way, only the BIOS is altered to meet the requirements.

With the new modularized BIOS used as a base, many enhancements were added to achieve greater system functionality. One of the first enhancements involved placing the entire operating system in ROM on board each processor. With this first important step, processors do not have to depend on disk drives or networks for bootup, thereby making each CPU fully independent of system peripherals that may or may not be present. The CPU cards still have full access to these peripherals, but they do not require them for basic system operation. Resource locks were added so that multiple cards could access the shared resources virtually simultaneously. Intelligent handling of interrupts and sharing of global memory were added functions. A new *mode of operation* function was added, enabling all CPU cards to run in one of three modes: 1) as a single processor, in a single CPU environment (e.g. in personal development station), 2) as a master processor in a multiple CPU environment, or 3) as a slave processor in a multiple CPU environment. The mode of operation is automatically determined by the BIOS and each CPU card contains the necessary code to operate in any mode at any time. Finally, a number of diagnostic functions were added, including comprehensive memory checking and listing of global and local system configuration.

PORTING BaSE To NEW PROCESSORS AND PERIPHERALS

Porting BaSE to new processors requires writing a limited amount of card specific code to initialize the card, to process the on board UART, and to handle local interrupts. Also, several constants in a formatted parameter file must be defined. The most important code in the porting process, the initial bootup code, is generic across all CPU cards so the programmer is virtually assured that the system will boot to some degree.

The process of designing drivers for disk drive devices has been greatly simplified. The original disk drivers supplied in the BIOS were custom coded in assembler. To improve maintenance, the original code was modularized, recoded in the “C” language, and relinked to the kernel. Now, each driver has the same format, same set of routines, and the same structure. New disk drivers can now be written, debugged, and linked to the kernel in a matter of days, with relative ease, because the programmer concentrates on the differences between the new disk controller requirements and the library of previous drivers. Additionally, a library of common disk functions serves to reduce the amount of new code that needs to be generated.

Overall system maintenance is simplified because 90% of the BIOS code is used across all processors. Making a change in the common BIOS is reflected in all processors for that revision. The card specific code is small in comparison and rarely needs modification.

USING BaSE IN A MULTIPROCESSING SYSTEM

Our use of multiprocessing is fairly straight forward. Each CPU card is stand-alone and runs its own copy of the operating system in its local RAM. Each card with custom telemetry processing hardware may be viewed as a smart peripheral card.

Certain system setup problems must be addressed in this environment in the area of handling interrupts and allocating global memory. In the BaSE, the master CPU card handles all global VMEbus interrupts and initializes all global VMEbus memory. The master CPU card converts global VMEbus hardware interrupts to software events that are set/cleared in a global memory area using operating system supplied multiprocessing support services. Slave CPU cards suspend on these global events and thus do not have to process the actual interrupt. When sizing and initializing tasking memory, the master CPU card installs all global memory that is contiguous to its local memory into its memory allocation map. Slave CPU cards size and initialize only their own local memory. This way, multiple processors do not attempt to install the same global memory, which would be catastrophic if used as tasking memory. Non-contiguous global memory could be installed by a utility program into any card’s (master or slave) memory allocation map after multitasking system operations have been started.

INTRODUCTION TO THE MODULAR ENVIRONMENT FOR DATA SYSTEMS

The systems designed, built, and programmed in GSFC’s Data Systems Technology Division all have a similar pipelined, multiprocessor, dual bus, hardware architecture as a platform on which to build application specific hardware. When designing software for such a system, there are many questions which need to be answered. What data and parameters will each processor need to accomplish its job? How will the processors communicate with each other? And with the operator? How can the total job be subdivided into tasks? On which processors will they run? What data and parameters will each task need to accomplish its job? How will each task get data? MEDS was developed to help answer these questions and give an application programmera starting point for designing a system based on the standard hardware platform.

The Modular Environment for Data Systems (MEDS) is designed as a general purpose software platform which is expanded and customized by application programmers to suit their particular requirements. It supports the basic software functions needed in all 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.

GENERAL DESCRIPTION OF MEDS

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 (see figure #2). 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 via the remote interface as well. In any case, it is the pipeline of custom cards that actually process the telemetry data.

The MEDS software resides mainly on the master with cooperating software running on each custom card. The basic MEDS functions include:

- Setup system and subsystems for processing (e.g. setup VLSI 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 transfer over network (e.g. transfer telemetry data to/from a workstation).

MASTER CONTROLLER

The system operator interacts with the operator interface task running on the master controller. The operator interface task directs the other tasks of the system. Typically, the operator interface task sends commands to other tasks in the system to setup and control the processing. The operator interface task also gathers system status for display to the operator.

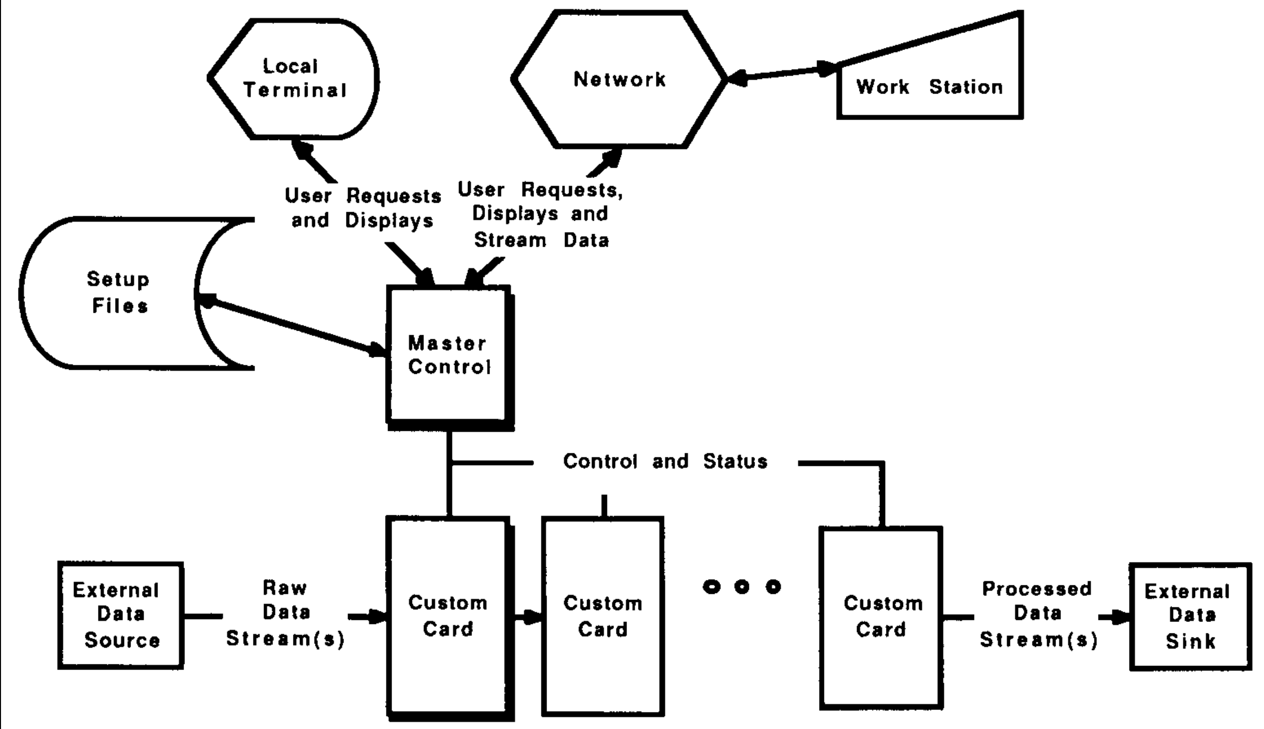

FIGURE 2. MEDS MULTIPROCESSOR ARCHITGECTURE

The following description explains the data flows within the master processor (see figure #3). Along the top path from left to right, operator requests are accepted and verified. The requests are translated into messages containing setup information which are passed to the lower level software. The messages are broken into custom card commands which are passed to the appropriate card's command buffer by the interprocessor communications package. A corresponding custom card task (shown on figure #4) picks up the commands, executes them and returns a response in the global memory. The master's interprocessor software waits for the response. When the response is available, it is sent back to the operator through the layers of software. Note, if the source of the commands was the remote operator, the commands and responses would originate and be returned to it. The remote interface is handled by a higher level task built on the the network interface software package.

Following the path on the bottom of figure #3 from right to left, the status display data flows from each custom card buffer to the operator. On each card, status is collected and placed in a global memory status buffer where the master will look for it (shown on figure #4). Each card's status block is copied on timed intervals into a system level buffer by the master. From the system buffer, the operator interface status display pages can be built. If the operator is on a remote device, the status is packaged and sent via network software.

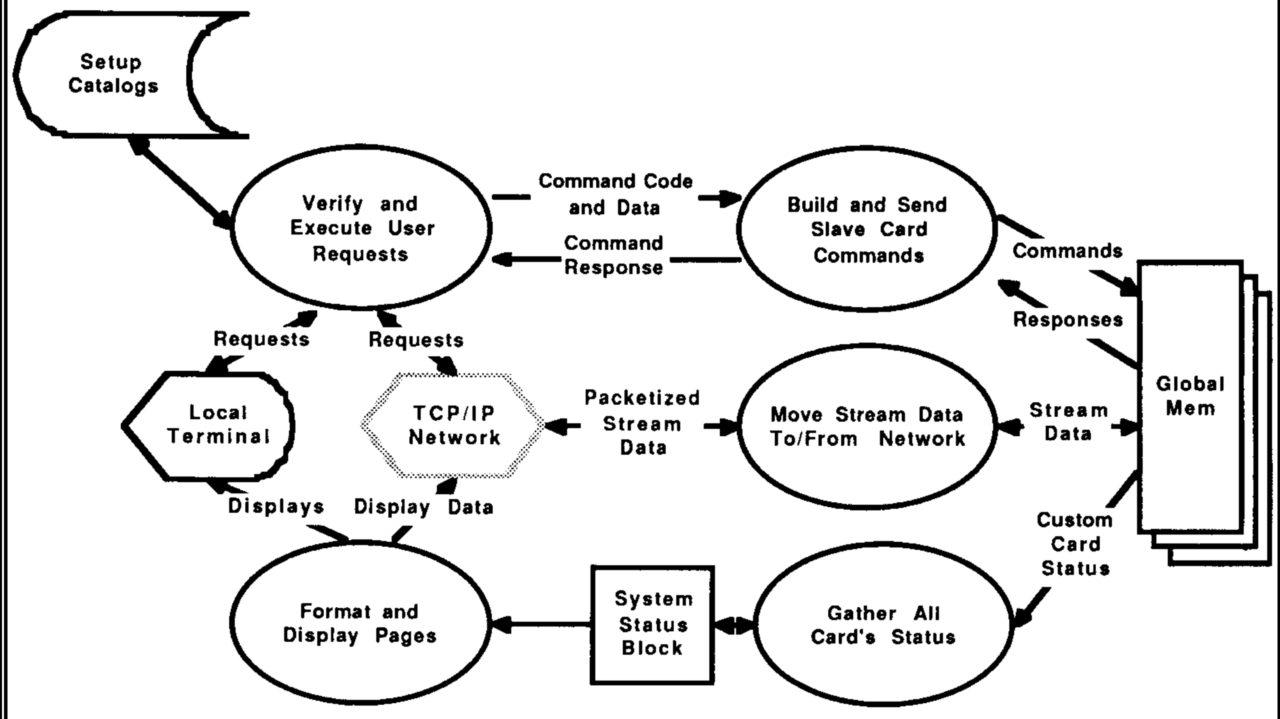

FIGURE 3. DATA FLOW WITHIN THE MASTER CONTROLLER

The path through the center the figure #3 represents the use of a remote device as a source or destination of telemetry data. Remote data stream transfer is handled by a separate task. This process picks up the data from a global memory buffer and transfers it to a remote device on the network or vice versa using the network communications software. The remote data transfer task can run on a separate processor dedicated to its function if needed.

CUSTOM CARDS

Since each custom card must interface with the master controller to receive commands and report processing status, a standard interface is employed. Usually two tasks are involved in the interface, the card's command handling task and a status gathering task (see figure #4). Both tasks exist as skeleton programs which are customized to the card's specific needs. The remaining custom card tasks are concerned with the specific pipeline data processing functions.

The global memory shown on the left of figure #4 corresponds the global memory shown on the right of the master data flow diagram (figure #3). The card software waits for commands to arrive in its command buffer. Once they arrive, the command handler, based on the command op code, calls the appropriate command routine. The command

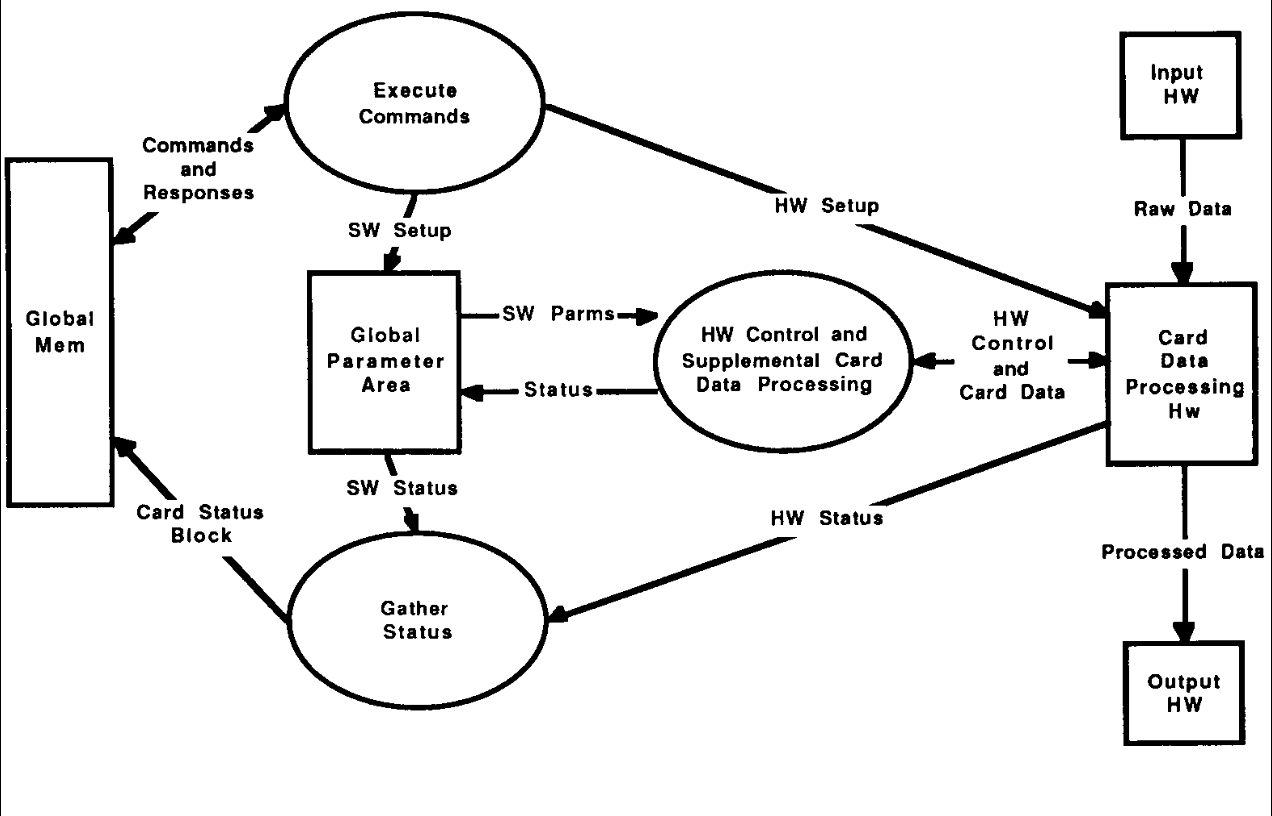

FIGURE 4. DATA FLOW WITHIN A CUSTOM CARD

is executed, a response is generated and it is placed in the global memory buffer by the command handler. When all commands are executed, the master is notified that a response is available.

There is a set of general commands which all cards must support such as reset, enable and disable. Each card has its own set of custom commands which depend on its specific functions such as packet filtering or CRC checking. Generally, a command initializes a card's hardware and software for operational processing or sets a hardware or software switch to control processing. The path along the top of the figure #4 illustrates command execution and response. A command execution may involve writing some custom hardware registers (HW setup) or software parameters in global memory (SW setup). A card software designer need not concern himself with the workings of the command passing, he only needs to analyze the functions and parameters needed of his card and write a list of commands to fulfill those needs. The command execution routines are the only portion the card developer must write and link to MEDS command handling package.

The actual pipeline data processing tasks and interrupt handlers are shown in the center of the figure as the process labelled "HW Control and Supplemental Data Processing". The number and design of these elements are dependent on the hardware design and functions required of the card. Using the HW and SW parameters setup by the card commands, these software elements interact directly with the custom hardware during telemetry data processing. Status is produced during processing by hardware and/or software.

The path along the bottom of figure #4 illustrates the status gathering function on a custom card. Each card produces status counts in hardware counters and/or software counters. A task dedicated to the status function runs on each custom card which collects the card status, formats it into the card status block and writes it to the global memory where the master expects it. The blocks are gathered on timed intervals by the master status task, and made available for display at the operator interface or for remote reporting. The custom card status task is optional. If all status counts are software generated, they may be incremented directly in the status buffer thus eliminating the need for a status task. Again, the card designer need not concern himself with how this works, he only needs to define what type of status his card will generate and where on the card it can be found.

Also available to the card designer, is the MEDS:LIB which is a software library of C routines that are commonly used by card developers. For example, move a word, move a long, move to fifo, move from fifo, move to circular buffer, move from circular buffer, display and prompt change of a memory location, selftest support routines and so on. These are reusable routines that appear repeatedly in custom card code. The source and object code are available and each routine is documented with functional description, input and output variables, error conditions and special notes.

MODULAR SOFTWARE

The overall MEDS software design is modularized into packages which 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 VME bus) and loosely coupled (on a network between the VME rack and remote workstation). Some of the packages exist as linkable libraries which 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 PDOS real-time, multitasking OS. MEDS software is written in ‘C’ and assembler is used for interrupt handlers.

CONCLUSION

To maintain maximum flexibility and performance for Goddard Space Flight Center’s VLSI telemetry system elements, two special real-time system environments were developed. The Base System Environment (BaSE) supports generic system integration while the Modular Environment for Data Systems (MEDS) supports application specific development. Architecturally, the BaSE resides just on top of a commercial real-time system kernel while the MEDS resides just on top of the BaSE. The BaSE provides for the basic porting of various manufacturer’s cards and insures seamless integration of these cards into the generic telemetry system. With this environment, developers are assured a rich selection of available commercial components to meet their particular application. The MEDS supports the basic software functions needed in all 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.

The need for these software platforms was apparent from the outset of our work in telemetry systems. Therefore, much effort was put into making every element of the first system general purpose. These elements are constantly evolving. Attempts are made to further generalize and enhance the designs of the hardware and software platforms in each revision. The pay off is in time, effort and money saved on current and future system development and system modification. It took 3 years to develop the first system from the ASICs to the operator interface, but current schedules for more complex systems are in the 1 year time frame, and would not be possible without the environments described in this paper.

REFERENCES

1. Hand, Sarah and Sabia, Stephen, “Functional Component Approach to Telemetry Data Capture Systems”, 88-058, Proceedings of the International Telemetering Conference, Las Vegas, Nevada, October 1988.

2. Chesney, James et al, “Multi-ASIC Strategies”, High Performance Systems, January, 1989, pp 22-34.

3. Looney, Andrew, “PDOS Networking”, Code 521, Goddard Space Flight Center, NASA, Greenbelt, Maryland, March 1989.

4. Horner, Ward and Sabia, Stephen, “Talking to Spacecraft: NASA’s New Multi-Processing Telemetry and Command System”, Proceedings of the Bus/Board Users Show and Conference, Marlborough, Massachusetts, October 1987.

5. Tardy, Jean E., “The Host/Target Approach To Embedded System Development Is Becoming Obsolete”, ACM Sigsoft, Vol 13, no 4, October 1988.

6. Chesney, Collins, Dominy, “Personal Telemetry Workstation For Future Distributed System Environments”, Code 521, Goddard Space Flight Center, NASA, Greenbelt, Maryland, March 1989.

7. Dominy, Carol and Grebowsky, Gerald, “VLSI High Speed Packet Processor”, 88-058, Proceedings of the International Telemetering Conference, Las Vegas, Nevada, October 1988.

8. Eyring Research Institute, Run Module Development, Provo, Utah, February 2, 1987.

NOMENCLATURE

BaSE	Base System Environment
BIOS	Basic Input/Output System
CMOS	Complimentary Metal Oxide Semiconductor
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
HW	Hardware
MEDS	Modular Environment for Data Systems
PDOS	Power Disk Operating System
RAM	Random Access Memory
ROM	Read Only Memory
SW	Software
UART	Universal Asynchronous Receiver/Transmitter
VLSI	Very Large Scale Integration
VME	Versabus Module Eurocard Format