Fermi National Accelerator Laboratory

FERMILAB-Conf-89/131

A Real Time Integrated Environment for Motorola 680xx-Based VME and FASTBUS Modules *

David Berg, Peter Heinicke, Bryan MacKinnon, Tom Nicinski, and Gene Oleynik
Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510 U.S.A.

May 1989

* Presented at "Real Time Computer Applications in Nuclear, Particle, and Plasma Physics," Williamsburg, Virginia, May 16-19, 1989.

A Real Time Integrated Environment for Motorola 680xx-based VME and FASTBUS Modules

David Berg, Peter Heinicke, Bryan MacKinnon,
Tom Nicinski, Gene Oleynik

Online and Data Acquisition Software Groups
Fermi National Accelerator Laboratory
Batavia, IL 60510

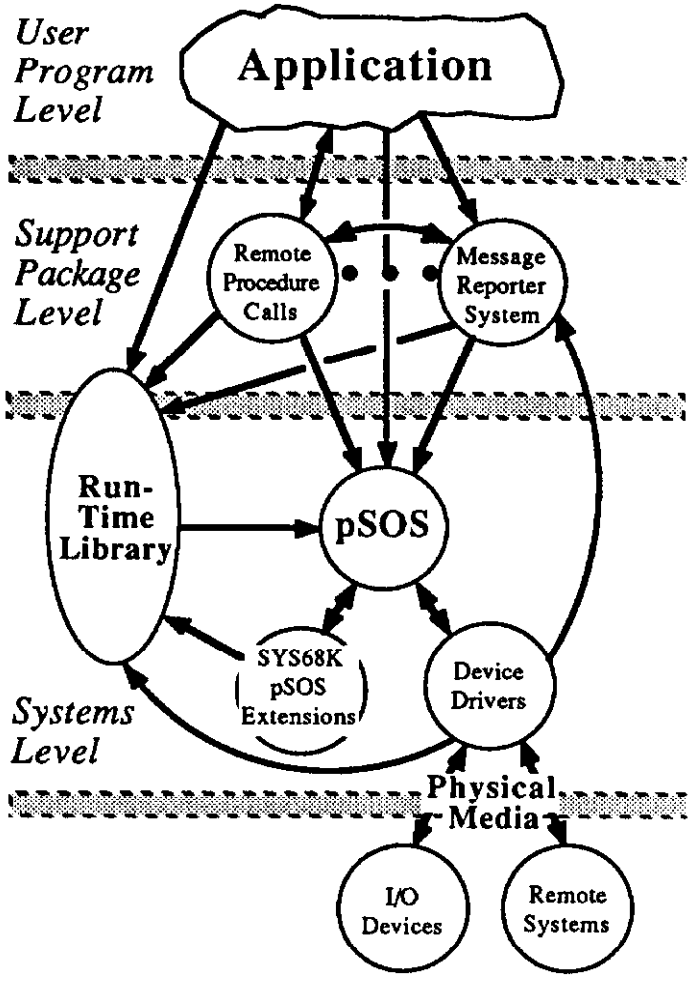
Abstract

The Software Components Group pSOS (1) operating system kernel and pROBE (1) debugger have been extended to support the Fermilab PAN-DA system for a variety of Motorola 680xx-based VME and FASTBUS modules. These extensions include: a multi-tasking, reentrant implementation of Microtec (2) C/Pascal; a serial port driver for terminal I/O and data transfer; a message reporting facility; and enhanced debugging tools.

I. INTRODUCTION

Experiments at Fermilab require the development and maintenance of very flexible real time applications, including data acquisition systems. The entire suite of hardware and software, from the low-level data read-out systems to the top-level VAXs that provide the user interface, system control, and data monitoring, is the PAN-DA system [1,2]. Within PAN-DA, a variety of Motorola 680xx-based processor modules may be used at one or more levels and in VME or FASTBUS. Some of these applications are intended for a specific hardware environment, but most are designed to run in more than one.

In general, the design philosophy has been to code each component of the system at the highest appropriate level. This provides both flexibility and portability, while making efficient use of limited software resources. Consistent with this philosophy, we wanted a single software environment for all the microprocessor components of PAN-DA. We could not find a commercial product that, by itself, satisfied all our requirements. We therefore purchased two products and integrated them into a software environment that we call the SYS68K product [3]. (SYS68K is not restricted, however, to use within the PAN-DA system.)



II. COMPONENTS OF SYS68K

The Figure illustrates a general application and the logical relationships among its components. The scope of this paper includes the Systems Level as shown in the figure, plus the

---

(*)sponsored by DOE Contract No.DE-AC02-76CH03000

(1) pSOS and pROBE are trademarks of Software Components Group, Inc.

(2) Microtec is a registered trademark of Microtec Research, Inc.

Message Reporter System; the Remote Procedure Calls support package is the subject of a companion paper [4].

The heart of the system is pSOS, a multi-tasking operating system kernel from Software Components Group (SCG). This provides the primitives for task and memory management, intertask communication and synchronization, and an I/O driver interface. It is essentially hardware independent (though currently limited to 680xx-based processors). To this we have added specific device drivers, support for higher level languages, debugging facilities, and system utilities. There is no command line interface (other than the pROBE debugger); the user interface to the application will reside on an external Host system (e.g., a VAX). There is no file system; all software is either resident in ROM or downloaded from a host system.

With pSOS we purchased from SCG a resident debugger, pROBE, that works in conjunction with pSOS. Through it, a programmer may access any processor memory location or register, obtain information about processes, exchanges, and other system structures, disassemble code and execute instruction steps, etc. Its primary utility lies in its intimate knowledge of pSOS internal data structures, and in turn, the debugging hooks from the kernel into pROBE.

Boot code that initializes the module hardware and starts up pROBE, together with pSOS and pROBE themselves, is packaged into a ROM file in SYS68K. There is one such file for each supported module; it is used to burn EPROMs that are installed on the boards. Application code is downloaded from a Host system as required, using pROBE.

The backbone of the system is a suite of cross-development tools from Microtec Research (MRI), including compilers, assembler, linker, librarian, and run-time libraries (RTL) for higher level languages.

All code management, compiling, and linking is done on our VAX cluster. Our applications are written mostly in C, some in Pascal, and little or none in assembler language. Even device drivers and interrupt service procedures are written mostly in higher level languages. Thus all levels of software depend upon the RTL. We have merged support for C and Pascal into a single RTL that is consistent with multi-tasking under pSOS.

Two device drivers have currently been implemented in SYS68K: a serial port driver, described in this paper, and a driver for the CMC ENP-10 Ethernet controller, described in a companion paper [5].

III. RTL Reentrancy and Process Context

An application, consisting of one or more programs, is linked as a whole against the objects and libraries that comprise the environment. In particular, the Run Time Library, utility routines, interfaces to pSOS system services, device drivers, and other system code are shared among all processes. Concurrent processes may even be multiple invocations of the same program code. In a multi-tasking system, this multi-threaded code must be made reentrant.

There are two reasons why portions of the RTL, as provided by MRI, were non-reentrant:

- o The routine references global (or static) variables which are used for returning statuses or maintaining context between calls to the routine. For example, many mathematical routines, such as ACOS, use ERRNO to return their statuses.

- o The routine uses heap or calls a routine which uses heap. The main culprits are the C routines MALLOC (and its cousins) and FREE.

There is a close connection between reentrancy and process context. pSOS provides three callout routines to supplement its actions when it

- o activates a process,
- o switches between process contexts, or
- o deletes a process.

We used these callouts to make references to global variables and use of the Floating Point Coprocessor reentrant, at least with respect to context switches. This required only minor changes to a few of the RTL source modules.

Non-reentrant use of heap storage was eliminated by implementing heap management using pSOS memory management primitives. This has the additional advantage that all dynamic memory use, whether system or process, is merged into a single region; this is more efficient use of limited memory resources. (For some processor modules, a single logical region may be composed of two physically disjoint regions.)

The reentrancy provided is between processes; some restrictions apply to ISPs (interrupt service procedures). Furthermore, since the microprocessor component of the PAN-DA system is intended to be non-interactive, the file I/O routines in the RTL were not modified to make them reentrant.

A. Process Extended Context

There is an extendable system data structure, the Process Control Block (PCB), that is maintained by pSOS for each process. Our PCB extension (PCBE) is the data structure by which additional process context information is maintained. This includes storage space for RTL and Floating Point contexts, stack information used for debugging, control and status flags, and fields accessible to support packages and user programs. Components of the SYS68K System use these fields to maintain their own per-process contexts. For example, RPX (a remote procedure call package) uses a PCBE

field as a pointer to its own context block. The PCBE address is made easily available to programs through a global variable that is loaded when a process is switched in.

B. Process Activation

When a process is activated (created) by pSOS, the callout routine performs three duties:

- o Allocate space for the PCBE (Process Control Block Extension).
- o Initialize PCBE entries for stack watching and optionally fill the per-process stacks with a high-water mark pattern. This facilitates the monitoring of stack usage.
- o Establish the initial process context. A prologue routine runs in each process the first time it is switched in.

C. Process Context Switch

pSOS schedules processes based on priority, with preemption. When a process becomes blocked (waits for a resource), pSOS will switch it out and switch in the highest priority process that is ready to run. Likewise, when a higher priority process becomes ready (a resource is now available), pSOS preempts the current process and switches the context. Our process context switching callout then performs the following duties:

- o Saves the out-switched process' RTL context. This involves copying five variables that are declared as global by the MRI compilers. The Floating-Point Coprocessor's (MC6888x) context can also be optionally saved.
- o Checks the out-switched process' stacks for any violations. This involves checking whether the stack pointers are within the bounds of their respective stacks, and whether a stack ceiling pattern still exists.
- o Restores the in-switched process' RTL context. The Floating-Point Coprocessor's (MC6888x) context can also be optionally restored.

Since Floating-Point context saving/restoration is quite time consuming, it is optional on a per-process basis. For processes that have not enabled Floating-Point context saving/restoration, the switching callout routine can be made to check whether the Floating-Point Coprocessor was used since the last time that process was switched in.

D. Process Deletion

When a process is deleted (terminated) by pSOS, the callout routine simply deallocates the PCBE. If this is the last active process, however, the pROBE debugger will be entered before the process terminates (otherwise, the system would idle indefinitely). A process may cause itself to be deleted simply by returning from its top level routine.

IV. THE SERIAL PORT DRIVER

The Serial Port Driver (SPDRV) is the resident terminal driver for the SYS68K product. It was designed to provide an adequate set of functions for use in normal terminal dialog with the user and as a communication medium over an RS232 line. Its features include:

- o Fully interrupt driven: Minimizes the impact on system performance.
- o Device independence: SPDRV provides a standard set of functions via a standard interface so that applications coded for one type of serial hardware need not be modified to run on another.
- o Multitasking: Arbitration for multiple processes accessing a given port is automatically handled.
- o Multiport access: Designed to support any number of ports on a single system.
- o User control: The user has control over how a particular port operates including echoing of input, line terminators, input line editing, flow control, interrupting execution of the processor via a BREAK, and support for higher level language requirements.
- o Optimized path I/O: A higher speed, reduced functionality data path is provided through the driver for applications that require it. This enables an RS232 line to be used as a data transfer medium while minimizing the impact on system performance.

A. Integration

SPDRV is well integrated into SYS68K and the MRI higher level language environment. Therefore, applications written in higher level languages such as MRI C and Pascal will have full access to SPDRV via standard language I/O (scanf, printf, writeln, readln). When using the SYS68K product, SPDRV is automatically incorporated into the application. However, if necessary, programs may bypass standard I/O and call the driver directly.

B. Callable Procedures

The driver procedures which may be called directly from a program are:

- o spd_read: Reads a buffer of bytes.
- o spd_write: Writes a buffer of bytes.
- o spd_alloc, spd_dealloc: Allocates, deallocates a port.
- o spd_set_mode, spd_enb_mode, spd_dis_mode: Changes the mode mask (characteristics) of a port.
- o spd_get_mode: Returns the current mode settings.

o spd_get_info: Returns information about a specified port.

The standard language I/O functions are implemented using the spd_read and spd_write routines.

C. Driver Structure

SPDRV is organized into two code levels: device independent and device dependent. The device independent code, written in C, handles the vast majority of the driver's work. The device dependent code, written in assembler, handles the interface between the device independent level and the hardware. The interface between the two levels is well defined; therefore, adding support for another type of serial hardware requires only writing the device dependent routines (seven in all).

Functionally, SPDRV is partitioned into interrupt processing and non-interrupt processing. Output handling is done completely via interrupt processing. When the driver receives a buffer of bytes for output, the first byte is written to the output port; when the driver is informed via an interrupt that output is complete, it immediately writes the next byte out. This continues until all bytes have been written.

Since input processing is significantly more complex, it is not practical to provide all input functions during interrupt processing without seriously compromising system performance. Instead, the more complex functions such as echo and line editing are deferred to a pSOS process associated with the driver. Input operations that use the previously mentioned optimized path I/O are serviced entirely via interrupt processing.

V. MESSAGE REPORTER SYSTEM

The SYS68K Message Reporter System is a subroutine package that provides programs with a means for displaying informational and error messages on a local console or at a remote system. The message routines operate asynchronously; thus, important system components generating messages will not be delayed while waiting for a message to be displayed. Messages can be generated by processes, ISPs (interrupt service procedures), system initialization, etc.

The Message Reporter System is an example of a package that builds upon the pSOS/Microtec environment extensions. It is written in C, uses pSOS system services, the Serial Port driver, the RPX system (which in turn uses the ENP10 driver), etc.

VI. ENHANCED DEBUGGING TOOLS

Good debugging tools are required not only during program development, but also after the product has been released and is in use at experiments. SYS68K provides a number of tools, starting with the stack watching and Floating Point Coprocessor usage checking implemented as part of the process context switch callout described above. The Message Reporter System only logs messages; it is not a debugging tool. The facilities provided by pROBE, though extensive, have proved to be insufficient and cumbersome for our purposes. Therefore, we have enhanced the debugging environment in three areas:

o pROBE extensions
o Run-Time tracing
o Postmortem analysis

A. pROBE Extensions

An application can easily incorporate additional pROBE commands, which are most useful during the code development stage. pROBE provides a mechanism to call a user-supplied routine whenever it cannot recognize a command; SYS68K supplies this routine, together with a mechanism for generating a table of such commands. The routine looks for the unrecognized command in the table. If the command is not located, control is returned to pROBE with an error. Otherwise, the command handling routine associated with the command in the table is called.

SYS68K's unrecognized pROBE command routine passes the remainder of the command line to the command handling routine after establishing a suitable context. The command handling routine can be written in a higher level language such as C. Terminal I/O routines that can be used at elevated IPL (interrupt priority level), including a version of C's PRINTF, are provided. In addition, routines are provided with SYS68K to permit the parsing of tokens within the line. These tokens match the pROBE syntax. For example, an expression consisting of numbers and process registers, etc., can be evaluated, or a process identifier can be obtained.

One extended pROBE command provided with SYS68K dumps the contents of a process control block, including our extensions, in a formatted fashion. Part of this dump routine determines how much of a process' user and supervisor mode stacks has been used and whether any stack overflows occurred; this information is not readily available using the standard pROBE commands. A HELP command is also supplied to produce a list of all available extended commands, with a brief description of each.

B. Run-Time Tracing

When control is transferred to pROBE through a serial port BREAK or some processor exception, the programmer may enter commands to examine the system. Alternatively, the tracing facility allows the saving of a program's execution history at run-time without any interaction from the programmer. The program calls trace routines that place trace entries in a circular buffer. This buffer can, independent of the

program, be periodically dumped to determine which sections of the program had executed.

The programmer decides the depth of tracing by the number of trace calls coded into the program. Trace entries also have activation levels associated with them. For example, the ENP10 driver permits multiple levels of tracing to be done, from scant trace entries for major operations to verbose details, all by changing only one value.

C. Postmortem Analysis

Once the initial stages of development and debugging are done, trace entries might not provide information about the overall system state sufficient for the detection of subtle bugs. pROBE can be quite cumbersome to use effectively, even with extended commands. Furthermore, it is most undesirable that a production system be unavailable while a failure is diagnosed. Instead, a postmortem analysis can be done using MRI's XRAY68K (3) Symbolic Debugger on a Host system; after a postmortem dump is made, the production system may be brought back into operation immediately.

An extended pROBE command is provided in SCG68K to permit the saving of one processor's complete context, including pSOS and pROBE, in another processor's memory. This information is then uploaded to a Host machine where XRAY68K is available. XRAY68K executes high-level source or assembly language programs. It also enables the programmer to control program execution. Since pSOS and pROBE were uploaded, pROBE commands and extensions can be used under XRAY68K to look at system data structures, processes, etc. XRAY68K's major limitation is that it cannot easily recreate the I/O devices' hardware environment.

VII. Impressions of Vendor Supplied Software

SYS68K utilizes products from two different vendors: Software Components Group, Inc. and Microtec Research, Inc.

A. Software Components Group, Inc.

We have found the products from Software Components Group, Inc., the pSOS operating system kernel and the pROBE debugger, to be of high quality. The documentation is well-written, complete in most areas, and accurate. When technical questions arose, we were able to contact Software Components directly via telephone and speak immediately with a technical person who gave quick, accurate answers. They have also been very responsive to bug reports (practically none) or suggestions that we might have.

B. Microtec Research, Inc.

The products from Microtec Research, Inc. have proved adequate. We use their C and Pascal compilers, assembler, linker, librarian, and XRAY68K debugger. Though we have been using their products for more than a year, we continue to encounter problems that take a significant amount of our resources to deal with.

VIII. Summary

The SYS68K product provides a solid foundation upon which we can build a variety of applications for 680xx-based processor modules. These include, but are not limited to, data acquisition system components for PAN-DA. The integrated environment spans software development, debugging, and production running, and represents about two person-years of work. We currently support three modules: the Motorola MVME133(A), the CERN GPM-68000 (from CES), and the CERN GPM-68020 (from Struck). In the near future, we expect to support the Fermilab FSCC (FASTBUS Smart Crate Controller); we believe we can easily port the environment to additional modules, perhaps for other bus architectures (e.g., NuBus, Futurebus).

IX. References

[1] See accompanying paper: Don Petravick, et al, "The PAN-DA Data Acquisition System"

[2] See accompanying paper: Ruth Pordes, et al, "Software for FASTBUS and Motorola 68K Based Readout Controllers for Data Acquisition"

[3] David Berg, et al, "SYS68K - Compound Product for pSOS/68K development", Fermilab Computing Department Note PN-387.

[4] See accompanying paper: Don Petravick, et al, "Remote Procedure Execution Software for Distributed Systems"

[5] See accompanying paper: Gene Oleynik, et al, "Uniform Communications Software Using TCP/IP"

---

(3) XRAY68K is a trademark of Microtec Research, Inc.