This paper appeared is a draft copy of the article of the same name that appeared in IEEE Transactions on Systems, Man, and Cybernetics, Vol. 22, No. 6, pp. 1282-1295, Nov/Dec 1992. Unfortunately the published article, as it appeared in the journal, is not available on-line.

The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications

David B. Stewart, Donald E. Schmitz, and Pradeep K. Khosla

Department of Electrical and Computer Engineering

The Robotics Institute

Carnegie Mellon University

Pittsburgh, PA 15312-3890

***Abstract**: This paper describes the Chimera II Real-time Operating System, which has been developed for advanced sensor-based control applications. It has been designed as a local operating system, to be used in conjunction with a global operating system. It executes on one or more single board computers in a VMEbus-based system. Advanced sensor-based control systems are both statically and dynamically reconfigurable. As a result, they require many special features, which are currently not found in commercial real-time operating systems. In this paper, we present several design issues for such systems, and we also present the features we have developed and implemented as part of Chimera II. These features include a real-time kernel with dynamic scheduling, global error handling, user signals, and two levels of device drivers; an enhanced collection of interprocessor communication mechanisms, including global shared memory, spin-locks, remote semaphores, priority message passing, global state variable tables, multiprocessor servo task control, and host workstation integration; and several support utilities, including a UNIX C and math libraries, a matrix library, a command interpreter library, and a configuration file library. Chimera II is currently being used with a variety of systems, including the CMU Direct Drive Arm II, the CMU Reconfigurable Modular Manipulator System, the Troikabot System for Rapid Assembly, and the Self-Mobile Space Manipulator.*

I. INTRODUCTION

Advanced sensor-based control applications, such as robotics, process control, and intelligent manufacturing systems have several different hierarchical levels of control, which typically fall into three broad categories: *servo levels, supervisory levels,* and *planning levels*. The *servo levels* involve reading data from sensors, analyzing the data, and controlling electromechanical devices, such as robots and machines. The timing of these levels is critical, and often involves periodic processes ranging from 1 Hz to 1000 Hz. The *supervisory levels* are higher level actions, such as specifying a task, issuing commands like *turn on motor 3* or *move to position B*, and selecting different modes of control based on data received from sensors at the servo level. Time at these levels is a factor, but not as critical as for the servo levels. In the *planning levels* time is usually not a critical factor. Examples of processes at this level include generating accounting or performance logs of the real-time system, simulating a task, and programming new tasks for the system to take on. In order to develop sensor-based control applications, a multitasking, multiprocessing, and flexible *real-time operating system (RTOS)* is needed.

An RTOS can be subdivided into several parts, including the real-time kernel, the multiprocessor support, the file system, and the programming environment. The *real-time kernel* provides local task management, scheduling, timing primitives, memory management, local communication, interrupt handling, error handling, and an interface to hardware devices. The *multiprocessor support* includes interprocessor communication and synchronization, remote interrupts, access to special purpose processors, and distributed task management. The *file system* provides access to secondary storage, such as disks and tapes, and to local-area-networks. The *programming environment* provides the tools for building applications; it includes the editor, compiler, loader, debugger, windowing environment, graphic interface, and command interpreter (also called a *shell*). The level of support provided for each part of the operating system (OS) varies greatly among RTOS.

In this paper, we present the Chimera II Real-Time Operating System, which has been designed especially to support advanced sensor-based control applications. Chimera II is designed as a *local OS* within a global/local OS framework, as shown in Figure 1. In such a framework, the global OS provides the programming environment and file system, while the local OS provides the real-time kernel, multiprocessor support, and an interface to the global OS. For many applications the global OS may be non-real-time, such as UNIX or Mach. However, the use of a real-time global OS such as Alpha OS [7] and RT-Mach [30] can add real-time predictability to file accesses, networking, and graphical user interfaces.

Most commercial RTOS, including iRMX II [5], OS-9 [13], and pSOS+ [24], do not use the global/local OS framework, and hence they provide their own custom programming environment and file system. The environments, including the editors, compilers, file system, and graphics facilities are generally inferior to their counterparts in UNIX-based OS. In addition, since much development effort for these RTOS goes into the programming environment, they have inferior real-time kernels as compared to other RTOS. Some commercial RTOS, such as VRTX [20] and VxWorks [32], do use the global/local OS framework. However, as compared to Chimera II, they provide very little multiprocessor support, and their communications interface to the global OS is limited to networking protocols, thus making the communication slow and inflexible. The commercial RTOS only provide basic kernel features, such as static priority scheduling and very limited exception handling capabilities, and multiprocessor support is minimal or non-existent. Previous research efforts in developing an RTOS for sensor-based control systems include Condor [18], the Spring Kernel [25], Sage [21], Chimera [23], and Harmony [3]. They have generally only concentrated on selected features for the real-time kernel, or were designed for a specific target application. Chimera II differs from these systems in that it not only provides the basic necessities of an RTOS, but also provides the advanced features required for implementation of advanced sensor-based control systems, which may be both dynamically and statically reconfigurable.

II. DESIGN ISSUES

Advanced sensor-based control systems should be *dynamically reconfigurable*, even for the simplest of applications. Consider the example of a robotic manipulator which is required to move an object whose location is known. This task can be broken up into three separate phases: pick up object; move to new position; put down object. When the manipulator is picking up the object, it must use force control for contacting the object, and gripper control to properly pick up the object. To move the object a different controller is required for the free motion phase. Possibly vision processing is also required to track the object's target location. To put down the object, force control and gripper control is again needed. The set of modules executing and the sensors required during each phase is different. Some of the modules and sensors are shared by the different phases, while others must be dynamically changed. As applications become more complex, the number of possible configurations also increases.

Advanced sensor-based control systems should be implemented on open-architecture hardware, so that new sensors and additional processing may be incrementally added to the system to increase the system's intelligence. In addition, the hardware and set of software configurations may have to be changed to support a different class of applications. Thus advanced sensor-based control systems must also be *statically reconfigurable*.

Several design issues, discussed in detail below, were considered in developing Chimera II, an RTOS for advanced sensor-based control systems that are both statically and dynamically reconfigurable.

A. Programming Environment and File System

The basic functionality found in all RTOS includes the programming environment, the file system, and the real-time kernel. The programming environment is required to quickly develop, debug, and maintain code. The basic requirements for the environment are an editor, a high-level language compiler, a linker, and a loader. The file system is required to store code and data on secondary storage, and to electronically transfer information to other systems. In order to provide all of the advantages of the full-featured programming environments and file systems of today's UNIX workstations, we adopted the global/local OS framework, and developed Chimera II as a local OS. Chimera II then requires a global OS to operate as the *host*, which makes all of the global OS's programming environment and file system features available to Chimera II. Given such a framework, we have developed a powerful interface between the host workstation and real-time environment, as described in Section III.B.

B. Open Architecture Real-Time Hardware

A typical hardware configuration for advanced sensor-based control applications may consist of multiple general purpose processors, possibly on multiple buses. The system may contain special processing units, such as floating point accelerators, digital signal processors, and image processing systems. The system will also include interfaces to several sensory devices, such as force sensors, cameras, tactile sensors, and range finders to gather data about the environment, and a variety of control devices such as actuators, switches, and amplifiers to control electromechanical equipment. We have implemented Chimera II around the VMEbus [16], since it is the most popular bus standard. However, much of the design of Chimera II is independent of the target hardware architecture, and hence can be used with other architectures.

Figure 2 shows a typical hardware configuration. General purpose processing is provided by commercially-available single board computers, which we call *real-time processing units (RTPUs)*. We have chosen to support the Motorola MC680x0 family of general purpose processors because of their popularity and their one-to-one mapping of hardware signals with the VMEbus signals [15]. The design of Chimera II allows other processor-based RTPUs, such as the Intel 80x86 and SPARC families, to also be used; however, we currently have not ported any software to those platforms. Any VMEbus-based I/O device or special purpose processor can be incorporated into Chimera II, by using the two-level device driver support offered by our OS.

C. Real-Time Kernel

The real-time kernel must include all the basic features found in any RTOS. These include task management, memory management, local shared memory, local semaphores, timer access, and hardware independence. *Task management* includes actions such as creating, destroying, blocking, waking up, setting priorities, and scheduling of concurrent tasks. *Memory management* is the allocation and deallocation of physical memory. We do not consider virtual memory, because we are not aware of any method to do paging in real-time. *Local shared memory* allows tasks on the same RTPU to share memory. *Local semaphores* provide basic synchronization for tasks on the same RTPU. *Timer access* allows the execution of tasks to be controlled by time. *Hardware independence* is a virtual machine layer, which allows user software to use the hardware without having to program hardware specific code. The level of hardware independence provided by an OS varies. We have developed an expanded real-time kernel suitable for reconfigurable systems. In addition to the basic kernel functions, our kernel also provides dynamic scheduling, two levels of device drivers, built-in control of timers, and global error handling. Details of the Chimera II kernel are given in Section III.A.

D. Interprocessor Communication and Synchronization

Most RTOS give very few mechanisms for interprocessor communication (IPC) and synchronization. The mechanisms available generally include support for an atomic read-modify-write instruction for limiting access to critical sections, and some form of message passing for higher-level communication. The VMEbus by default offers shared memory; however, most RTOS do not make any provisions for allocating that memory, or for automatically performing the address calculations required to access different parts of memory on the VMEbus. As a result, programs which use any of these mechanisms become bound to a single processor. The same code cannot execute on a different processor, without modification of VMEbus addresses, or in the case of message passing, modifying the names of the source and destination of the messages. In a reconfigurable system, modules must be designed such that they are independent of the target RTPU; hence these methods are not satisfactory. Second, these mechanisms do not provide all the tools desirable for quickly developing multiprocessor applications. In Section III.B. we describe the enhanced IPC and synchronization mechanisms developed in Chimera II, including global shared memory, spin-locks, remote semaphores, prioritized message passing, global state variables, and multiprocessor task control.

E. Predictability

The primary concern in a real-time system is not that it is fast, but rather that it is *predictable*. A fast system with unpredictable behavior can cause serious damage. In theory, the rate monotonic algorithm [10] is used to ensure predictable execution. However, this static scheduling algorithm is not suitable for dynamically reconfigurable systems and does not provide the CPU utilization achievable with dynamic scheduling algorithms [11]. We have developed the *maximum-urgency-first* algorithm to provide a predictable dynamic scheduling algorithm [26]. This algorithm has been implemented as the default scheduler for Chimera II. Support for the rate monotonic algorithm, as is provided in most RTOS, is also available with the default scheduler.

Most real-time scheduling theory concentrates on ensuring that tasks always meet their deadlines. However, nothing is said about what happens to tasks that fail to meet deadlines. In addition, even though a task can meet all deadlines in theory, abnormalities in the implementation may cause the task to miss its deadline in practice. In Chimera II we have addressed this issue by designing a deadline failure handling mechanism, which allows an exception handler to be automatically called when a task fails to meet its deadline. Possible error handling includes aborting the task and preparing it to restart the next period; sending a message to some other part of the system to handler the error; performing emergency handling, such as a graceful shutdown of the system or sounding an alarm; maintaining statistics on failure frequency to aid in tuning the system; or in the case of iterative algorithms, returning the current approximate value regardless of precision. Details of the Chimera II scheduler and deadline failure handling are included in Section III.A.

Deadline failures account for only one of the many types of errors which may occur in a system. Deadline failures are unique in that the errors are a function of time. Other errors that may occur in a system include hardware failures, software bugs, invalid data, invalid states, and processor exceptions. These errors may occur at any time, and are often detected within the user's code through the use of consistency *if-then* statements. The most typical method of dealing with these errors is to have the routine detecting the error to either handle it itself, or to return an error value, such as –1. Unfortunately, in a hierarchical software architecture, this method has two major problems. First, the exception handling code is embedded within the application code, making the code inflexible and difficult to maintain. Second, if an error occurs at the lowest level of the architecture, and it is only to be handled by a higher level module, then the error must be continually passed up through the intermediate software levels. This method is both cumbersome to code, and is also very inefficient, in that the time to enter the exception handling code requires the additional overhead of propagating the error through multiple hierarchical levels. In Chimera II we have developed a *global error handling mechanism*, which allows exception handling code and main program code to be coded independently. Details are given in Section III.A.

F. Modular and Reusable Software

In order to save on development time and make software more maintainable, it is generally accepted that applications should be designed in a modular fashion, and the software modules should be reusable. Chimera II extends this ideology to practice, by providing the tools which make it possible to quickly develop modular and reusable software. First, all communication mechanisms are processor transparent, and hence modules can be developed without knowledge of the final target hardware architecture. Second, the two-level device drivers supported by Chimera II provide a standard interface not only to I/O devices, as offered by other RTOS, but also to sensors and actuators. Third, the servo task control mechanism forces the programmer to develop code as reconfigurable, hence resulting in reusable modules. Each module can then execute on any RTPU at any frequency, and all modules can be controlled by a single task. The modules form a library of modules, any of which can be used in later systems without the need for recompiling any parts of the code. The intercommunication between the modules is handled automatically using a high-performance global state variable table mechanism. The servo task control mechanism and global state variable table are described in Section III.B. Details on developing a library of reusable modules can be found in [27].

G. Libraries

The usefulness of an operating system does not only lie in the features given, but also on the supporting libraries, which save on application development time. Like most other RTOS, Chimera II provides the standard UNIX C and math libraries. It also provides a concurrent standard I/O library (stdio), which is suitable for using the stdio facilities in a multiprocessor environment. In addition, several other libraries are provided, which are generally not found in other OS. These include a matrix math library, a command line interpreter library, and a configuration file support library. These libraries provide utilities which are often required by advanced sensor-based control applications. More details on these libraries are in Section III.C.

III. SOFTWARE ARCHITECTURE

The Chimera II software is divided into two distinct parts: 1) Code that runs on the RTPUs, and 2) Code that runs on the host workstation.

Figure 3 shows the data flow diagram for code that runs on the RTPUs. On each RTPU, user tasks execute concurrently, communicating with each other through local shared memory and local semaphores. They also have direct access to local I/O devices, and can communicate with other processors using any of the IPC mechanisms available. Each RTPU has a server task which constantly monitors the *express mail*. The express mail is used to initialize IPC services and to provide access to the extended file system. The server translates symbolic names into pointers, and performs any necessary calculations to translate addresses within various address spaces on the VMEbus. A copy of the Chimera II kernel executes on each RTPU. The kernel provides a real-time multitasking environment for executing the tasks concurrently.

The *host workstation* is also an integral part of the Chimera II environment. Figure 4 shows the data-flow diagram for code that executes on the host. Three categories of processes execute on the host: the server process, console processes, and user processes. All processes communicate via local shared memory and semaphore facilities available in the host's global OS.The host's server provides similar functionality as the server on the RTPUs. In addition, it can access the host file system directly, and it includes special primitives to support the console processes. The *console process* provides the user interface which serves to download and execute programs on the RTPUs. The console process also provides the *stdin, stdout, a*nd *stderr* files for tasks executing on the RTPUs. A single console process can control all RTPUs within the system. However, if multiple RTPUs are using *stdin*, only one of them can have it active at once. Other tasks reading from *stdin* block, and send a *Waiting for TTY Input* message to the user's terminal, similar to the way UNIX handles background processes. If multiple RTPUs require *stdin* simultaneously, then multiple instances of the console process can be created, each in a separate window on the host workstation.

User processes are just like any other UNIX process running on the host workstation, except that an additional Chimera II library is linked in, allowing the process to use the Chimera II IPC package. By using this library, the host workstation appears as an RTPU to the other RTPUs, thus making communication between the host workstation and RTPUs transparent. The host processes can thus read from and write into the memory of any RTPU, send or receive messages, or synchronize using remote semaphores. This powerful feature allows users to create their own custom user interfaces, which possibly include graphics or windowing facilities offered by the host workstation.

A. Real-Time Kernel

One of the important goals of the kernel is to provide the required functionality at the highest performance possible. In our system, this is achieved by sacrificing traditional operating system features, such as virtual memory and inter-task security. Our basis for measuring performance is the amount of CPU time during normal execution which must be dedicated to the operating system functions, such as scheduling, task switching, and communication overhead. The major design decisions that we made in developing the Chimera II kernel are described below.

1) Tasks: A task in Chimera II is also known as a thread or lightweight process in other operating systems. A user program which is downloaded into an RTPU consists of a single executable file. The kernel is supplied as a C library and is linked into the executable image. When a program is downloaded to an RTPU and executed, some kernel initialization is performed, following which the user's *main()* routine is *spawned* as a task. Any other task can then be started from *main()*.

2) Inter-task Security: Typically, all of the tasks running on a given RTPU (or set of RTPUs) are designed to cooperate. We have sacrificed inter-task protection, allowing one task to access the address space of any other task. This eliminates a large amount of overhead incurred in performing system calls or their equivalents. In general, inter-task security is desirable for two reasons: 1) to prevent multiple users or independent processes from infringing on other processes in the system; and 2) to contain software errors within a faulting module for better control of error handling. Although Chimera II is multitasking and multiprocessor, it is designed to be single user. Two totally separate control systems should each have their own installation of Chimera II. This is necessary if predictable execution of each system must be maintained. Therefore the first reason for wanting inter-task security is not applicable to Chimera II. As for containing errors within a software module, inter-task security prevents corrupting memory of other modules, and causes the faulting module to abort. These types of errors typically occur due to software bugs. The global error handling mechanism has some facilities for automatically detecting bugs, such as memory corruption errors or bad arguments to system calls, and hence provides an alternate method for handling software errors.

3) Task Management: Chimera II provides the kernel task management features typical to all RTOS, including creating, suspending, restarting, preempting and scheduling. In addition, Chimera II provides a different approach to handling task timing. Whereas other RTOS require the user to program timers, the Chimera II kernel performs all timer programming automatically. A *pause(restart_time)* routine is provided, which tells the kernel to pause the task until the specified restart time. The kernel schedules the tasks using virtual timers, all based on a single hardware timer. Using this method the number of tasks in an application requiring the use of timers is not limited by the number of timers available on the RTPU. Therefore the user does not have to manually perform any timer hardware multiplexing, as is necessary with most other RTOS when there are insufficient hardware timers.

4) Local Inter-task Communication: There is no parent/child relationship among tasks. Any task can communicate or synchronize with any other task either through local shared memory, local semaphores, or user signals. Within a single executable file, all global variables are automatically shared. Local semaphores provide high speed synchronization between tasks, and can be used either to provide mutual exclusion during critical sections (binary semaphores), or to provide synchronization among tasks (general or counting semaphores). User signals provide an alternate method of synchronization, allowing the receiving task to be interrupted when the signal arrives, instead of explicitly checking if the signal has arrived as done with the local semaphores. Any of the IPC mechanisms described in Section III.B. can also be used locally.

5) Memory Management: The total address space used by all tasks on one system is limited by the physical memory available on the RTPU. Chimera II does not provide any virtual memory, as the memory management and swapping overhead not only decreases the performance of a system drastically, but it also causes the system to become unpredictable, thus violating one of the important rules of real-time systems. Chimera II provides its own version of the *malloc()* family of routines to allocate physical memory.

6) Interrupt Handling: The Chimera II kernel provides the interfacing routines to easily define and install C-language interrupt handlers for VMEbus *IRQ* interrupts, local *LRQ* interrupts, and a user-definable number of mailbox interrupts, even if the hardware only supports one or two mailbox interrupts. Utilities are also given to enable and disable interrupts, and to indefinitely lock a task into the CPU for atomic or critical emergency handling code.

7) Real-Time Task Scheduler: Chimera II supports both static and dynamic preemptive scheduling of real-time tasks. The default scheduler supports the *rate monotonic* (RM) static priority scheduling algorithm [10][11], the *earliest-deadline-first* (EDF) and *minimum-laxity-first* (MLF) dynamic scheduling algorithms [11][33], and the *maximum-urgency-first* (MUF) mixed (static and dynamic) scheduling algorithm [26]. In addition, the scheduler is designed as a replaceable module, allowing user-defined schedulers to easily override the default scheduler, just by linking the new scheduler with the application.

The wide range of support for scheduling algorithms with the default scheduler allows Chimera II to be used in many applications, without being restricted by the default scheduler, as is the case with the commercial RTOS, which restricts the programmer to using a static priority scheduling algorithm. For example, RM is usually the algorithm of choice when developing a single-configuration system. EDF and MLF are used in dynamically changing systems when transient overloads of the system are not possible, or in static systems when maximum CPU utilization is required. MUF is an algorithm we have designed especially for dynamically reconfigurable systems, where critical tasks can be guaranteed to execute, even during transient overloads of the system [26].

8) Deadline Failure Handling: One novel and powerful feature in Chimera II is its deadline failure handling mechanism. Whenever a task fails to meet its deadline, an optional failure handler is called on behalf of the failing task. The failure handler can be programmed to execute either at the same or different priority than the failing task. Such functionality is essential in predictable systems. Any of the actions specified in Section II.E. and other user-defined actions can be implemented using the deadline failure handling available with our MUF scheduler.

Estimating the execution time of tasks is often difficult. For example, most commercially-available hardware is geared towards increasing average performance via the use of caches and pipelines. Such hardware is often used to implement real-time systems. As a result, the execution time cannot necessarily be predicted accurately. Under-estimating worst-case execution times can create serious problems, as a task in the critical set may fail. The use of deadline failure handlers is thus recommended for all tasks in a system. The Chimera II default scheduler provides this ability.

9) Error and Exception Handling: Three types of errors may occur in an advanced sensor-based control system: hardware errors, state errors, and software errors [2][4]. A *hardware error* is either directly generated by the hardware, such as a processor exception (e.g. *bus error*), or be detected by software, as is usually the case with I/O devices. Sometimes these errors are only transient, or can be corrected by software; other times human intervention is required to reset the hardware. *State errors* are software generated, generally after some form of *if-then* comparison. These errors are more abstract than hardware errors, and indicate that an invalid software state has been detected. For example, a state error occurs if a robot *pick-up* command fails, because the object was not there or the grasp operation failed. The hardware itself does not generate an error; but the state *holding object* after the pick-up operation is not correct. *Software errors* are due to software design oversights, limitations, or bugs. In general they should be fixed promptly. However, if they occur while the system is executing, appropriate error handling is required to ensure predictable execution.

In a predictable system, every error must be handled in a known fashion. In the best case, an error handler is called that corrects the error. An intermediate solution is to operate the system with degraded performance; this is often necessary with fully autonomous system. In the worst case, the system must be shutdown. To prevent damage to the system and the environment, system shutdown must be graceful, such that moving actuators slowly come to a halt, and all power is turned off after the actuator comes to a halt.

One of the most powerful features in Chimera II is the *global error handling mechanism*. It allows user programs to be developed without ever having to explicitly check the return value of errors. Instead, whenever an error is detected, the error handling mechanism is invoked, by generating an *error signal*. By default, a detailed error message is printed, and the task at fault is aborted. The mechanism allows error messages to be very specific, thus aiding in tracking down errors in the system. When the debug mode is enabled, the source code filename and line number are also printed. The programmer may override the default handler, on a per-error-code, per-module, or per-scope basis. After handling the error, the default action of aborting the task can also be overridden, either by continuing at the line after the error, or by returning to a previously marked place in the program. Every task has its own set of error handlers.

The design of the global error handling allows the definition of programs and error handlers to be kept separate. In traditional UNIX C code, error handling is built-in to the code by using *if* statements to check return values of routines, as shown in the following code segment:

```
if ((fd = open("filename",O_RDONLY)) == -1) {
    fprintf(stderr,"Error opening file %s\n",filename);
    exit();
}
```

When the global error handling mechanism is enabled, the *open()* routine would generate an error signal instead of returning –1. The default error handling of printing a detailed error message and aborting the task would then occur, unless a user-defined handler was previously installed to handle errors occurring in the *open()* routine. As a result, the main program becomes much simpler, in that error checking after each system call is not needed. The above code segment then reduces to the following:

```
fd = open("filename",O_RDONLY);
```

It has also been our experience that most software bugs go unnoticed because return values are not checked. Routines such as *read()* and *printf()* return error codes when an error is detected. However, because they are almost always successful, programmers tend to not check the return values, in the interest of increased performance, decreased program complexity, and faster development time. The global error handling will automatically detect and handle any errors in these routines, thus eliminating the need for the programmer to check the return values. As a result the code becomes much more robust, and hence behavior is more predictable, as even unexpected or rarely occurring errors are handled.

State errors are usually detected by software. The following code segment is a typical example of detecting a state error:

```
if (count > 10 && !found) {
    errnum = ENOTFOUND;
    return(-1); /* error, object not found */
}
```

If using the global error handling mechanism, the following code segment would instead be written as follows:

```
if (count > 10 && !found)
    errInvoke(moduleptr,ENOTFOUND);
```

The routine *errInvoke()* generates an error signal, which is handled either by the default error handler or by a user-defined error handler. The *moduleptr* argument is obtained during initialization of the particular module containing this code.

The basic design philosophy of the global error handling mechanism in Chimera II is similar to that found in some languages, such as Ada, AML/X [17], and Exceptional C [2]. However, by making it part of the OS it has several major advantages such as programming language independence, automatic detection of system call and kernel errors, and the ability to automatically detect such things as bad arguments to system calls or memory corruption, which are generally a result of software errors.

The global error handling mechanism coexists with standard UNIX error handling. If it is not enabled, then any routines that generate errors, such as system calls, return an appropriate error value (e.g. –1 or *NULL*) and set the global variable *errno*, as is consistent with UNIX. Processor exceptions, such as *bus error* and *divide-by-zero*, also generate error signals. However, when the global error handling is disabled, C-language processor exception handlers can be installed to override the default action of aborting the task.

10) I/O Devices, Sensors, and Actuators: The Chimera II kernel adopts a two-level approach to developing and using device drivers to isolate the user from the details of special hardware. The first level drivers provide a hardware independent interface to I/O devices, such as serial ports, parallel ports, analog-to-digital converters, digital-to-analog converters, and frame grabbers. It supports the *open, close, read, write, ioctl,* and *mmap* drivers, as is customary in most UNIX-based OS. These drivers are usually much simpler to write than their UNIX counterparts because inter-task security is not a concern. The second level device drivers are not required, but are highly recommended. They provide a standard hardware independent interface to the sensors and actuators which are connected to the system via the I/O ports. This modular interface allows applications to be developed without prior knowledge of the details of the sensors and actuators, and allows the sensors and actuators to be developed independently, without knowledge of the target application.

B. Multiprocessor Support

1) Express Mail: The express mail mechanism is a high-speed communication protocol we have developed especially for backplane communication. It is the lowest level of communication used by the system level software. It has three basic functions: 1) provide a layer of processor transparency, so that all other IPC mechanisms remain processor independent; 2) handle the I/O between the host workstation and RTPUs; and 3) send system signals and mailbox interrupts between RTPUs. In order to keep IPC overhead to a minimum, most other IPC mechanisms only use the express mail during initialization. After a communication link is made, all subsequent communication bypasses this layer, so that unnecessary overhead is removed. User's cannot directly send and receive express mail messages; rather, the other IPC mechanisms must be used.

2) Global Shared Memory: The VMEbus has several different address spaces, and each processor will address the spaces differently. By using the automatic address resolution features of the express mail, memory can be allocated from any RTPU on any other RTPU or memory card, and it can be shared by tasks on all RTPUs. All the address resolution and address offsets are performed during initialization of the shared memory segment, so that transfers between local and shared memory have no operating system overhead.

3) Spin-Locks: Spin-locks make use of atomic *test-and-set* (TAS) instructions in order to provide mutual exclusion for shared data [14]. In general they require the least amount of overhead of any synchronized IPC mechanism. However, they use polling to obtain the lock, which may waste significant CPU time. The polling time and a bounded time for retrying before issuing a time-out error can be set by the user.

4) Remote Semaphores: Most real-time operating systems only provide local semaphores; Chimera II also provides remote semaphores which allow tasks on multiple processors to use semaphores. Like the spin-locks, the remote semaphores use TAS instructions to obtain a lock; however, unlike the spin-locks, a task will block instead of polling for the lock. When the lock is released, the blocked task will automatically be woken up. Both binary and counting remote semaphores are supported. The remote semaphores can be used either for mutual exclusion when accessing a shared memory segment or for synchronizing tasks on different RTPUs.

5) Priority Message Passing: The Chimera II priority message passing allows typed messages to be sent between tasks on the same or different RTPUs. The message passing uses the express mail to initialize a message queue. Variable-length messages are then sent by placing them into the queue. The length of the queue is user-definable. Messages are received by retrieving them from the queue, either on a *first-in-first-out, last-in-first-out,* or *highest-priority-first* basis. The sending and receiving of messages bypasses the express mail, thus eliminating unnecessary run-time overhead.

6) Global State Variable Table Mechanism: A state variable table mechanism has been developed to allow control tasks on multiple RTPUs to share state information, and to update the state information quickly and correctly. This mechanism automatically creates one global table, and a local copy of the table for each task which requires access to the state variables. Tasks always make use of the local copy, with periodic updates (typically at the beginning and end of each periodic cycle) from the global table. This mechanism allows the exact calculation of the required VMEbus bandwidth and transfer time required when integrating multiple real-time control tasks [27].

7) Multiprocessor Servo Task Control: In conjunction with the global state variable table mechanism, this mechanism is designed to support the automatic integration and control of reconfigurable servo task control modules. One task in the system may take control over some or all of the RTPUs in the system. The task can then spawn new tasks on any RTPU, and control the execution of the other tasks through simple “on”, “off”, “control”, and “status” mechanisms. The integration of multiprocessor applications becomes fairly simple. Reconfiguring an application to use different controllers can be performed dynamically. This mechanism also makes the updates of the local copies of state variable tables (described above) automatic, thus removing that responsibility from the programmer.

8) Extended File System: The real-time system makes use of the file systems on the host workstation, instead of requiring a separate disk file system in the real-time environment. This allows the real-time tasks to read and write the same files as any process on the host workstation. A standard UNIX system call interface, using the *open(), close(), read(), write(), ioctl()*, and *lseek()* routines, are used to access the extended file system. The concurrent standard I/O library can also be used for buffered I/O. All remote operations are completely transparent to the user.

9) Host Procedure Calls: Tasks running in the real-time environment can trigger execution of routines on the host workstation by using host procedure calls. In such cases, execution of the routines is performed on the host, but it *appears* to the user that it is executing in the real-time environment. Therefore, an interactive program running on an RTPU can call an editor (such as *vi* or *emacs*), and allows the user to edit information. Once the user exits, execution of the task on the RTPU then continues. The host procedure calls provided include changing current directory, getting an environment variable, and executing any stand-alone program, such as *ls, vi,* and *grep*.

10) Host Workstation Integration: The host workstation is fully integrated into the real-time environment; it appears to all the RTPUs as just another RTPU. As a result, processes on the host workstation can make use of the global shared memory, remote semaphores, or prioritized message passing to communicate rapidly with tasks in the real-time environment. For example, a host process may be graphically displaying data on the workstation. The data is being collected in real-time, and being stored in a shared memory segment on one of the RTPUs. The host process can then periodically read that location, and update the graphics display. To bypass the lack of timing on the host workstation, the task in the real-time environment can use the binary remote semaphores to periodically signal the host process, using the more accurate real-time timers, instead of the less accurate UNIX timers.

11) Special Purpose Processors: Special purpose processors are generally added to real-time systems in order to increase performance for specialized computations. Examples of special purpose processors are floating point accelerators, image processors, digital signal processors, LISP machines, and transputers. In order to simplify the integration and use of the processors, we have designed a hardware independent interface which allows a task to download, copy data, and call subroutines on the processors. The processors are treated as slaves in the system which do not execute a kernel. However, non-preemptive scheduling, such as *first-come-first-serve* or *shortest-job-first* can be used to schedule multiple subroutine calls to these processors.

C. Libraries

Chimera II provides several libraries which ease the programming of large applications:

1) UNIX C and math libraries: These libraries provide compatibility with UNIX-based OS and other RTOS. The C library includes routines for manipulating strings, block copy, fill, and move operations, random number generating, sorting and searching routines, time-of-day utilities, and memory allocation routines. The math library includes the transcendental and logarithmic functions, as well as other useful floating point functions.

2) Concurrent standard input/output library (stdio): This library provides a multiprocessor version of the UNIX *stdio* library, which allows multiple tasks on multiple processors to share files, including the *standard input*, *standard output*, and *standard error* of the remote console.

3) Matrix math library: This library provided optimized floating point routines for manipulating vectors and matrices. Functions available include matrix addition and multiplication, dot and cross products, determinants, Gaussian elimination, and matrix copy and sub-copy.

4) Command interpreter library: This library provides a set of routines for quickly building custom command line interfaces. The custom command interpreter automatically has built-in searching for closest match, shell execution, help files, and script file support.

5) Configuration file library: In a reconfigurable system, configuration files are used to define the current configuration. This library provides the utilities for quickly reading a standardized, yet general format for configuration files. When using this library, most error checking is automatically performed by the library routines, removing that need from the programmer.

Details of any of the features described in this section can be found in the Chimera II program documentation [28].

IV. IMPLEMENTATION

Chimera II has been used with several different systems, both at Carnegie Mellon University (CMU) and elsewhere, including at the Jet Propulsion Laboratory, California Institute of Technology; the Air Force Institute of Technology; Wright State University; and the University of Alberta. At CMU, it is being used with the CMU Direct Drive Arm II (DDArm II) [8], the CMU Reconfigurable Modular Manipulator Systems (RMMS) [22], the Self Mobile Space Manipulator [1], the Troikabot System for Rapid Assembly [9][19], and the Amada Steel Sheet Bending System. Each of these systems is very different in nature, but can be controlled using the same base operating system, which is proof of the flexibility of Chimera II.

As an example, a block diagram of the system components of the CMU DDArm II is shown in Figure 5. The system has a total of 12 processors in the real-time environment: three general purpose MC680X0 processors, a 20 Mflop peak Mercury floating point accelerator, six TMS320 digital signal processors, and an Imaging Technology vision system with two Weitek processors. The Chimera II kernel is running on two of those processors: the Ironics MC68020 and Ironics MC68030 processors. The other processors are all treated as special purpose processors. Note that the Heurikon MC68030 is executing the OS-9 real-time kernel. The only reason for not using Chimera II on this processor is that the commercial libraries for the vision system were only available as OS-9 object code. With such a setup, we show that Chimera II is capable of working in conjunction with other RTOS, if the need arises.

The system has a total of 4 buses: the host VMEbus, the control subsystem VMEbus, the vision VMEbus, and a multibus for the robot controllers. Bus adaptors are used to isolate the bus traffic of each subsystem. Chimera II automatically handles the address translations required when crossing adaptors.

Several sensors are connected to the system, including a 6-degree-of-freedom trackball, a tactile sensor, a force/torque sensor, a radiation sensor, and a camera mounted on the end-effector. By using Chimera II, sensors can easily be added or removed, and cooperating software quickly written. As a result, application programmers can concentrate on the higher level details of sensor integration, as opposed to the low-level and hardware details.

A. Performance

The success of a real-time operating system in developing robotic applications is based both on the ease of use and on its performance. Writing code to run under the Chimera II environment is as simple as writing a C language program to run under UNIX. Several critical operating system features were timed to provide a general idea of the performance of Chimera II. The timings shown in Table 1 were performed on Ironics model IV3230 RTPUs [6], each with a 25 MHz M68030 CPU and a 25MHz M68882 floating point coprocessor (FPU). All times were measured using a 25 MHz VMETRO VBT-321 Advanced VMEbus Tracer [31].

A task can be created in 220 µsec, and destroyed in 99 µsec. In general, these operations are done during initialization and termination, and not during time critical code. As a result, performance of these routines is sometimes sacrificed, in favor of performing any computations beforehand, so that the performance of run-time operations, such as context switching, can be improved. It takes 20 µsec to update the static priority of a task, which includes rearranging any queues sorted on priority. The time to update the dynamic priority of a task is included in the reschedule time, described next.

One of the important performance criterion of a multitasking kernel is the reschedule and context switch time. Upon the expiration of a time quantum, a timer interrupt is generated, forcing a *time-driven* reschedule. If the currently running task has the minimum laxity or is highest priority, it remains running, and the timer interrupt results in no task swapping. The scheduling time in this case is 28 µsec, resulting in a peak CPU utilization of 97 percent with a one millisecond time quantum. At the other extreme, a full context switch is needed, which executes in 66 µsec, for a total of 94 µsec, thus providing minimum CPU utilization of 91 percent for CPU-bound jobs. The scheduling time includes all checks for missed deadlines, updating dynamic priorities, and selecting a new task if the highest priority task changes. A full context switch involves suspending the current task by saving its entire context, including the FPU registers, and resuming the new task by restoring its entire context. Note that saving and restoring the FPU registers accounts for over half the context switch time alone. Although it is possible to improve the context switch time for tasks which do not use the FPU, it was decided that most control tasks use at least some floating point operations. Instead of adding overhead to keep track of when floating point operations were used, the full FPU context is always saved and restored at each context switch. A reschedule and context switch arising as a result of the running task blocking due to resource contention does not have to recalculate any dynamic priorities; it takes 82 µsec. Worst case CPU utilization can be increased to over 99 percent by increasing the time quantum to 10 milliseconds, which can be done for all but demanding servo level tasks which must run at frequencies typically above 100Hz. A task can also be locked into the CPU, thus disabling the preemption. In this case only 6 µsec are used each timer interrupt to update the system clock.

The local *P()* (wait) and *V()* (signal) semaphores each execute in 7 µsec when there is no blocking or waking-up occurring. A blocking task adds the time of the resource contention context switch before the next task begins executing. The *V()* operation takes an additional 27 µsec to wake up a blocked task.

Inevitably, remote semaphores take longer than local semaphores. A non-blocking *semP()* remote semaphore operation takes 27 µsec. If the *semP()* operation results in blocking the running task, then the time of the resource contention context switch is also needed before the next task can execute. The *semV()* remote semaphore wakeup operation takes 26 µsec, if no tasks are to be woken up. Waking up a blocked task which is on the same RTPU takes an additional 38 µsec, while waking up an off-board task takes another 36 µsec to send a mailbox interrupt to the proper RTPU for a total of 100 µsec. There is a hardware bug in the implementation of the TAS instruction on the Sun workstation, preventing it from being atomic over the VMEbus. A TAS workaround is built-in to all Chimera II IPC mechanisms. Thus the remote semaphore operations include an 11 µsec overhead, which can be saved if a host workstation without the bug is used, and hence the TAS workaround not needed.

For the multiprocessor priority message passing, it takes 62 µsec for a task to place a 24-byte message into a queue which is physically stored in local memory, or 79 µsec to store the message into a queue in remote memory. These times include the C subroutine call overhead. It takes 64 µsec for a task to retrieve that message from a queue in local memory, or 81 µsec for a queue in remote memory. Thus an interprocessor message can be sent and received in 160 µsec. These times all include the 11 µsec overhead for the TAS workaround.

There is no software overhead involved in accessing global shared memory. The speed of shared memory transfers across the VMEbus is limited only by the hardware. As a result, global shared memory is the fastest means of communication among tasks within a multiprocessor system.

Note that the times above are subject to small variations as Chimera II continues to be developed. Nevertheless, the timing measurements provide a fairly good representation of the performance of Chimera II. The high performance of Chimera II allows it to maintain over 90 percent CPU utilization for control tasks running up to 1000 Hz, thus allowing it to be used with the most computational and time demanding servo tasks.

B. Comparison of Chimera II with other Real-Time Operating Systems

In this section, we briefly describe how the features and performance of Chimera II compare with other RTOS. We are only interested in local RTOS for control applications, which are capable of supporting tasks up to 1000 Hz. Thus, global RTOS such as ARTS [29], Real-Time MACH [30], and Alpha [7] will not be considered. Although they provide various powerful real-time features, their designs do not allow them to be used effectively with tasks executing up to

1000 Hz because of significant amounts of system overhead. These operating systems, however, are excellent candidates for providing real-time facilities on the host workstation, allowing the host workstation and the RTPUs executing Chimera II to communicate with each other in real-time.

It has been very difficult to obtain performance times of the same features with each RTOS because we did not have their required hardware and software available. In addition, the timings published for one RTOS are not necessarily comparable to those in a different RTOS, because of differences in functionality for similar features. What is needed is a standard, independent benchmarking of each RTOS for proper comparison. Such an effort has begun [12] but has not yet been completed. However, based on the performance benchmarks available in the literature referenced earlier, the following observations can be made.

- The context switch time for Chimera II is 66 µsecs on a 25 MHz M68030 CPU, of which at least half of the time is for saving and restoring all the M68882 FPU context and registers. VxWorks claims a 35 µsec context switch time, but that does not include the time to save FPU registers. OS-9 advertises 55 µsecs, which possibly includes the FPU context save. As for research operating systems, SAGE takes 170 µsec, and no timings are available for Harmony and Condor.
- Interprocessor priority messages in Chimera II take 79 µsecs to send a 24-byte message, and another 81 µsec to retrieve that messages, even if the message is being sent to the host workstation. SAGE takes as much as 920 µsecs for a 4-byte RTPU-to-RTPU message. It takes as much as 15 milliseconds when the host is involved, using the UDP protocol. Condor takes at least 200 µ, for RTPU-to-RTPU, and as much as 34 milliseconds when the host workstation is involved in the transfer. No times are available for interprocessor messages using any of the commercial RTOS, although those systems that do support multiprocessing use TCP/IP for messages, which puts the timing of these messages on the order of milliseconds.
- VxWorks estimates 10 µsec for a non-blocking local semaphore operation on a 25 MHz M68020 processor. VRTX takes 24 µsec on a 25MHz MC68030. Chimera II performs the same operation in 7 µsec on a 25 MHz M68030 processor. Some of the other RTOS do not even support local semaphores, and only Chimera II supports remote semaphores.

The above observations show that Chimera II performs as well, if not better, than other RTOS. However, the strength of Chimera II lies in the multitude of features it has which are highly desirable for developing advanced sensor-based control applications, but are not available with the other RTOS. Following is an annotated list of the features we have designed especially for advanced sensor-based control applications, and implemented in Chimera II:

- *Combined static and dynamic scheduling:* the default Chimera II scheduler is capable of supporting the rate monotonic, earliest-deadline-first, minimum-laxity-first, maximum-urgency-first, and user-defined real-time scheduling algorithms.
- *Two-level device driver support:* a first level of device drivers provides the standard UNIX-like interface to I/O devices. The second level provides a standardized, hardware independent layer to sensors and actuators.
- *Special purpose processor interface:* a standardized interface to special purpose processors makes it easy to integrate and use such processors for enhancing real-time performance.
- *Global error handling mechanism:* programmers no longer have to worry about checking the return values of any routine that may generate an error. All errors in the system, including those which can detect software bugs, produce an error signal, which by default prints an error message and aborts a task, but can be overridden with any user-defined error handler.
- *Express mail:* a new communication protocol based on the shared memory available on open-architecture standard buses allows high-speed system communication between RTPUs.
- *Global shared memory:* the automatic address resolution and transparent no-software-overhead access of shared memory allows RTPU-independent modules to use global shared memory efficiently.
- *Remote semaphores:* both binary and counting semaphores can be used by tasks on different RTPUs, instead of being restricted to a single RTPU as in other RTOS.

- *Global state variable table mechanism:* this mechanism has been developed to allow control tasks on multiple RTPUs to share state information, and to update the state information quickly and correctly. The mechanism allows the maximum bus bandwidth and execution time of the system to be computed *apriori*.
- *Multiprocessor servo task control mechanism:* this mechanism is designed to support the automatic integration and control of dynamically reconfigurable servo task control modules in a multiprocessor environment.
- *Host workstation integration:* by linking a Chimera II library with processes on the host workstation, those processes can communicate with tasks running on the RTPUs, using the global shared memory, remote semaphores, and priority message passing. The host workstation appears to the RTPUs as just another RTPU.
- *Host procedure calls:* facilities that are not available in the real-time environment, but are available on the host workstation, can be accessed transparently by tasks running on RTPUs by using the host procedure calls.

In addition to the above features, Chimera II also contains several other features useful for advanced sensor-based control systems, which are similar to features found in other RTOS:

- *Priority message passing:* many RTOS offer this feature, but the Chimera II implementation, based on the express mail, allows variable-length user messages to be transferred across processors in 160 μsec.
- *Extended file system:* all RTOS provide some form of file system. In Chimera II, the extended file system uses the express mail to provide high-speed access to the file system of the host workstation.
- *C language interface to interrupt handlers:* the programming of interrupt handlers is difficult in most RTOS, because assembly language must be used. Chimera II provides an interface allowing C-language handlers to be installed without the need for any interfacing assembly code.
- *User signals:* a task on an RTPU may interrupt another task on the same RTPU by sending a user signal. The handling of signals is completely user-definable.
- *Spin-locks:* the Chimera II implementation of spin-locks allows the polling time and a bounded retry time to be specified by the user.

Chimera II also provides a matrix library, command interpreter library, a configuration file reading library, and a concurrent version of the standard I/O library, in addition to the standard UNIX C library. These libraries reduce the development time required for applications by providing already-debugged utility routines for often used functions.

V. SUMMARY

When implementing a real-time advanced sensor-based control systems, too much time is typically spent with low-level details to get the hardware to work, as opposed to higher level applications which allow the system to do something useful. Chimera II is a real-time operating system that adds a layer of transparency between the user and the hardware by providing a high-performance real-time kernel and a variety of IPC features. The hardware platform required to run Chimera II consists of commercially available hardware, and allows custom hardware to easily be integrated; and the design allows it to be used with almost any type of VME-based processors and devices. It allows radically differing hardware to be programmed using a common system, thus providing a first and necessary step towards the standardization of reconfigurable systems which results in a reduction of development time and an increase in productivity. Chimera II is already being used with several different systems, both at CMU and elsewhere.

VI. ACKNOWLEDGEMENTS

The research reported in this paper is supported, in part, by U.S. Army AMCOM and DARPA under contract DAAA-2189-C-0001, by NASA under contract NAG5-1091, by the Department of Electrical and Computer Engineering, and by The Robotics Institute at Carnegie Mellon University. Partial support for David B. Stewart is provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a Graduate Scholarship.

VII. REFERENCES

[1] H. B. Brown, M. B. Friedman, T. Kanade, "Development of a 5-DOF walking robot for space station application: overview," in *Proc. of 1990 IEEE International Conference on Systems Engineering*, Pittsburgh, Pennsylvania, pp. 194-197, August 1990.

[2] I. J. Cox and N. H. Gehani, "Exception handling in robotics," *Computer*, vol. 22, no. 3, pp. 43-49, March 1989.

[3] W. M. Gentleman, S. A MacKay, D. A. Stewart, and M. Wein, "An introduction to the Harmony real-time operating system," *IEEE Computer Society Technical Committee Newsletter*, pp. 3-6, Summer 1988.

[4] J. B. Goodenough, "Exception handling: issues and a proposed notation," *Comm. of the ACM*, vol.18, no. 12, pp. 683-696, December 1975.

[5] Intel Corporation, Santa Clara, CA 96051, *Extended iRMX II.3*, 1988.

[6] Ironics Incorporated, *IV3230 VMEbus Single Board Computer and MultiProcessing Engine User's Manual,* Technical Support Group, 798 Cascadilla Street, Ithaca, New York 14850.

[7] E. D. Jensen, J. D. Northcutt, "Alpha: a nonproprietary OS for large, complex, distributed real-time systems," in *Proc. IEEE Workshop on Experimental Distributed Systems*, Huntsville, Alabama, pp. 35-41, October 1990.

[8] T. Kanade, P.K Khosla, and N. Tanaka, "Real-time control of the CMU Direct Drive Arm II using customized inverse dynamics," in *Proc. of the 23rd IEEE Conference on Decision and Control*, Las Vegas, NV, pp. 1345-1352, December 1984.

[9] P. K. Khosla, R. S. Mattikalli, B. Nelson, and Y. Xu, "CMU Rapid Assembly System," in *Video Proc. of the 1992 IEEE International Conference on Robotics and Automation,* Nice, France, May 1992.

[10] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *Proc. 10th IEEE Real-Time Systems Symposium*, Santa Monica, CA, pp. 166-171, December 1989.

[11] C. L. Liu, and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment," *Journal of the Association for Computing Machinery*, v.20, n.1, pp. 44-61, January 1973.

[12] S. Malone, "Benchmarking kernels as a means of evaluating real-time operating systems," Master's Thesis, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, December 1991.

[13] Microware, *The OS-9 Catalog,* (Des Moines, Iowa 50322), 1989.

[14] L. D. Molesky, C. Shen, G. Zlokapa, "Predictable synchronization mechanisms for multiprocessor real-time systems," *The Journal of Real-Time Systems*, vol. 2, no. 3, September 1990.

[15] Motorola, Inc., *MC68030 enhanced 32-bit microprocessor user's manual*, Third Ed., (Prentice Hall: Englewood Cliffs, New Jersey) 1990.

[16] Motorola Microsystems, *The VMEbus Specification*, Rev. C.1, 1985.

[17] L. R. Nackman et al., "AML/X: a programming language for design and manufacturing," in *Proc. of 1986 Fall Joint Computer Conference,* pp. 145-159, 1986.

[18] S. Narasimhan, D. Siegel, and J. Hollerbach, "Condor: A revised architecture for controlling the Utah-MIT hand," in *Proc. of the IEEE Conference on Robotics and Automation,* Philadelphia, Pennsylvania, pp. 446-449, April 1988.

[19] N. P. Papanikolopoulos, B. Nelson, and P. K. Khosla, "Monocular 3-D visual tracking of a moving target by an eye-in-hand robotic system," in *Proc. 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '92)*, Raleigh, North Carolina, July 1992. (submitted)

[20] J. F. Ready, "VRTX: A real-time operating system for embedded microprocessor applications," *IEEE Micro*, vol. 6, pp. 8-17. August 1986.

[21] L. Salkind, "The Sage operating system," in *1989 IEEE International Conference on Robotics and Automation*, Phoenix, Arizona, pp. 860-865, May 1989.

[22] D. E. Schmitz, P. K. Khosla, and T. Kanade, "The CMU reconfigurable modular manipulator system," in *Proc. of the International Symposium and Exposition on Robots* (designated 19th ISIR), Sydney, Australia, pp. 473-488, November 1988.

[23] D. E. Schmitz, P. K. Khosla, R. Hoffman, and T. Kanade, "CHIMERA: A real-time programming environment for manipulator control," in *1989 IEEE International Conference on Robotics and Automation*, Phoenix, Arizona, pp. 846-852, May 1989.

[24] Software Components Group, Inc., *pSOS+/68K User's Manual*, Version 1.0, (San Jose, California 95110) 1989.

[25] J. A. Stankovic and K. Ramamritham, "The design of the Spring kernel," in *Proc. of Real-Time Systems Symposium*, pp. 146-157, December 1987.

[26] D. B. Stewart and P. K. Khosla, "Real-time scheduling of sensor-based control systems," in *Proc. of Eighth IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, Georgia, pp. 144-150, May 1991.

[27] D. B. Stewart, R. A. Volpe, and P. K. Khosla, "Multiprocessor integration of control modules," in *Proc. 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '92)*, Raleigh, North Carolina, July 1992. (submitted)

[28] D. B. Stewart, D. E. Schmitz, and P. K. Khosla, *Chimera II Real-Time Programming Environment*, Program Documentation, Version 1.12, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213; 240 pages, April 1992; email: *<chimera@ri.cmu.edu>*.

[29] H. Tokuda and C. Mercer, "ARTS: A distributed real-time kernel," *ACM Operating Systems Review,* vol. 23, No. 4, July 1989.

[30] H. Tokuda, T. Nakajima, and P. Rao, "Real-time Mach: Towards a predictable real-time system," in *Proc. of the USENIX Mach Workshop*, Oct. 1990.

[31] VMETRO Inc., *VBT-321 Advanced VMEbus Tracer User's Manual*, 2500 Wilcrest, Suite 530, Houston, Texas 77042.

[32] Wind River Systems, Inc., *VxWorks Reference Manual*, Release 4.0.2, (Alameda, California 94501) 1990.

[33] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Scheduling tasks with resource requirements in hard real-time systems", *IEEE Transactions on Software Engineering*, v.SE-13, n.5, pp. 564-577, May 1987.

Table 1: Chimera II Performance of Selected Features

Task-Management	
Create a task with default arguments	220 µsec
Destroy a task	99 µsec
Modify static priority of a task	20 µsec
Timer-driven reschedule, no task swapping	28 µsec
Context switch time, including CPU and FPU save and restore	66 µsec
Timer-drive reschedule, with context switch	94 µsec
Resource contention reschedule, with context switch	82 µsec
Timer interrupt, CPU locked and preemption disabled, no rescheduling	6 µsec
Local Semaphores	
P() operation, no blocking	7 µsec
P() operation, blocking	89 µsec
V() operation, no waking up	7 µsec
V() operation, waking up task	34 µsec
Remote Semaphores	
semP() operation, no blocking	27 µsec†
semP() operation, blocking, with resource contention reschedule	109 µsec†
semV() operation, no waking up	26 µsec†
semV() operation, waking up task on local RTPU	64 µsec†
semV() operation, waking up task on remote RTPU	100 µsec†
Multiprocessor Priority Message Passing	
Place a 24-byte message into queue, queue in local memory	62 µsec†
Place a 24-byte message into queue, queue in remote memory	79 µsec†
Retrieve a 24-byte message from queue, queue in local memory	64 µsec†
Retrieve a 24-byte message from queue, queue in remote memory	81 µsec†
Global Shared Memory	
Overhead for reading from shared memory	0 µsec
Overhead for writing into shared memory	0 µsec

†Time includes 11 µsec overhead for TAS workaround.

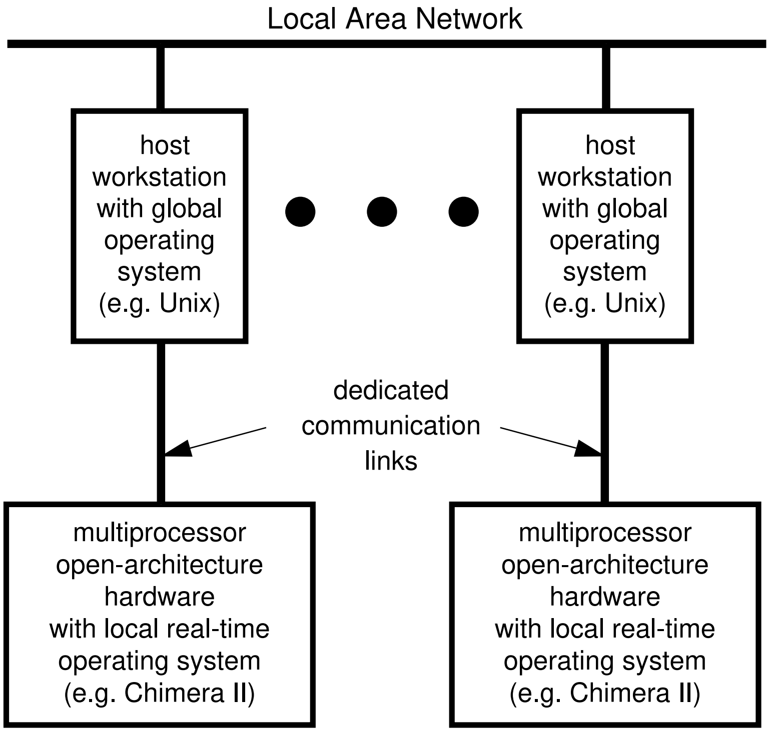

Figure 1: Framework of global/local operating system separation

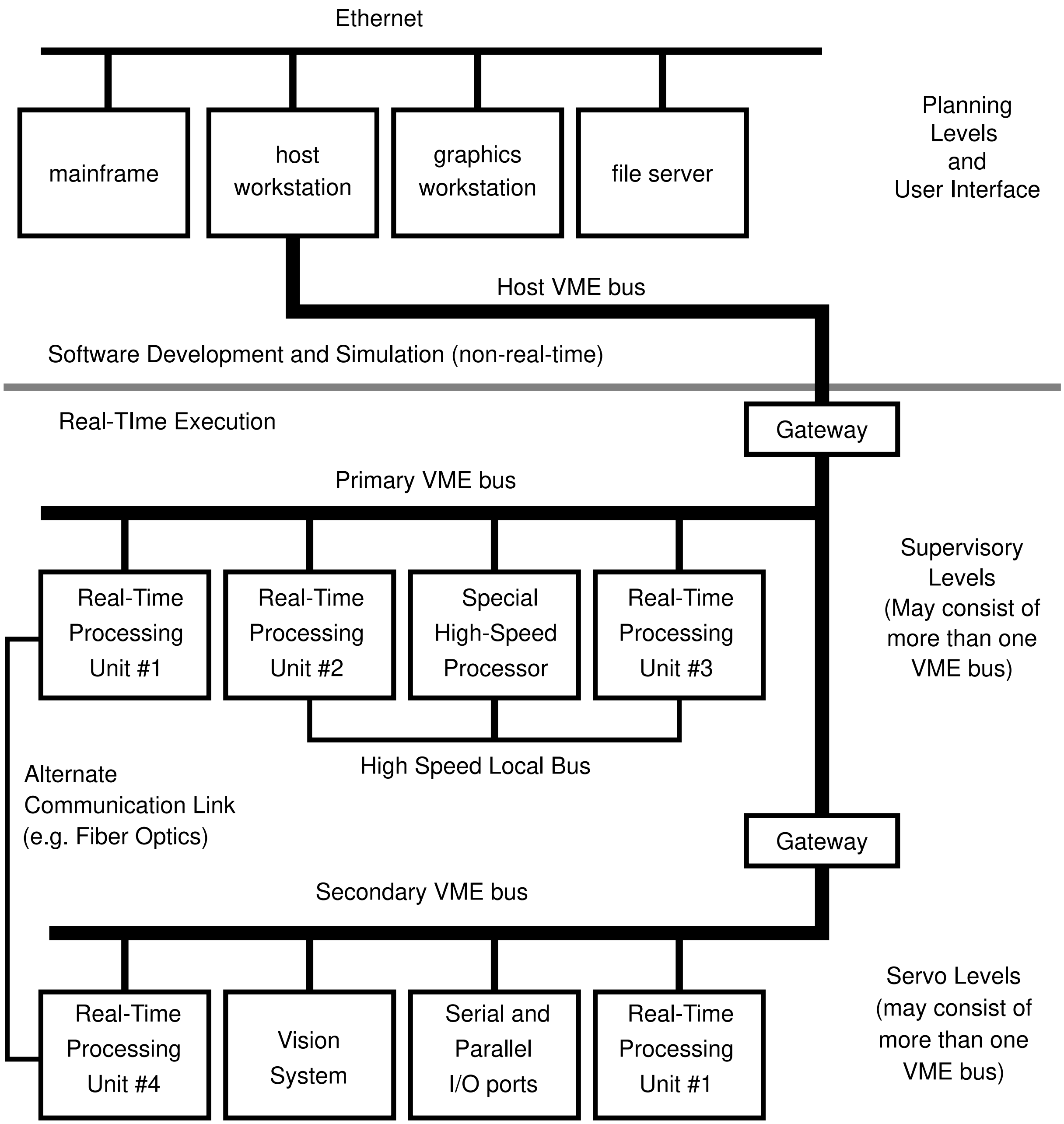

Figure 2: Open-architecture hardware typically used in advanced sensor-based control systems.

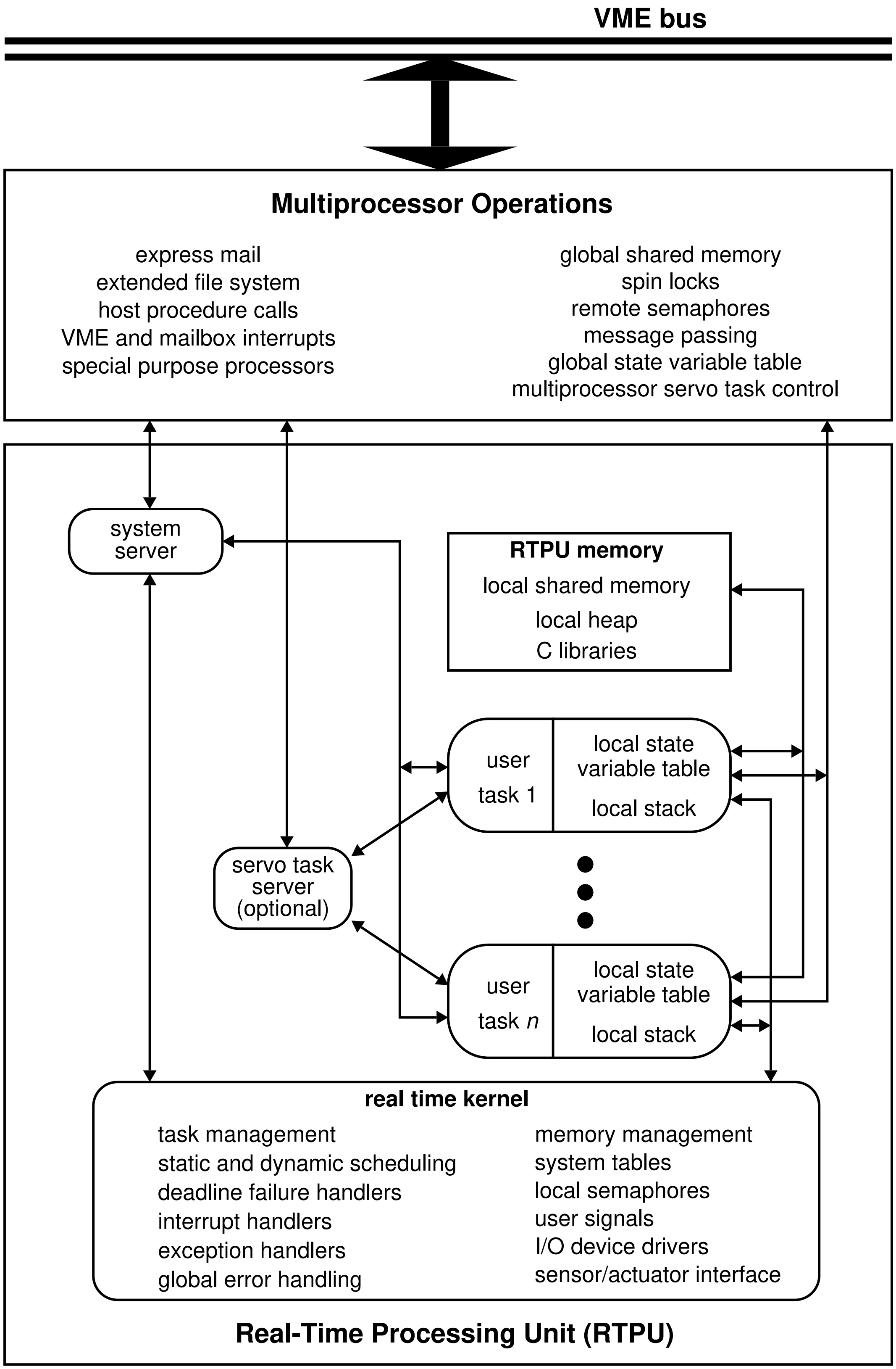

Figure 3: Software architecture of a real-time processing unit (RTPU)

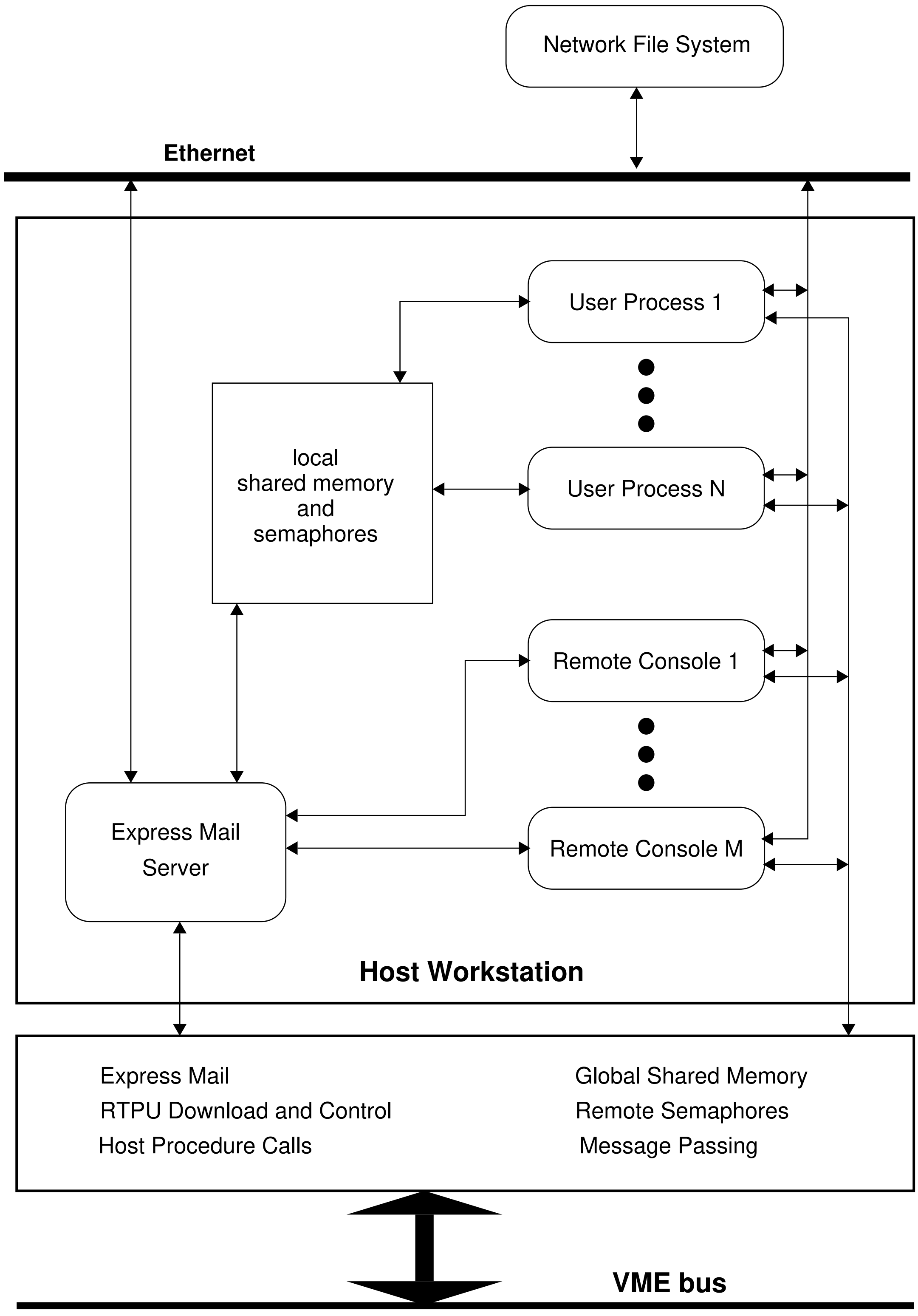

Figure 4: Software architecture of host workstation

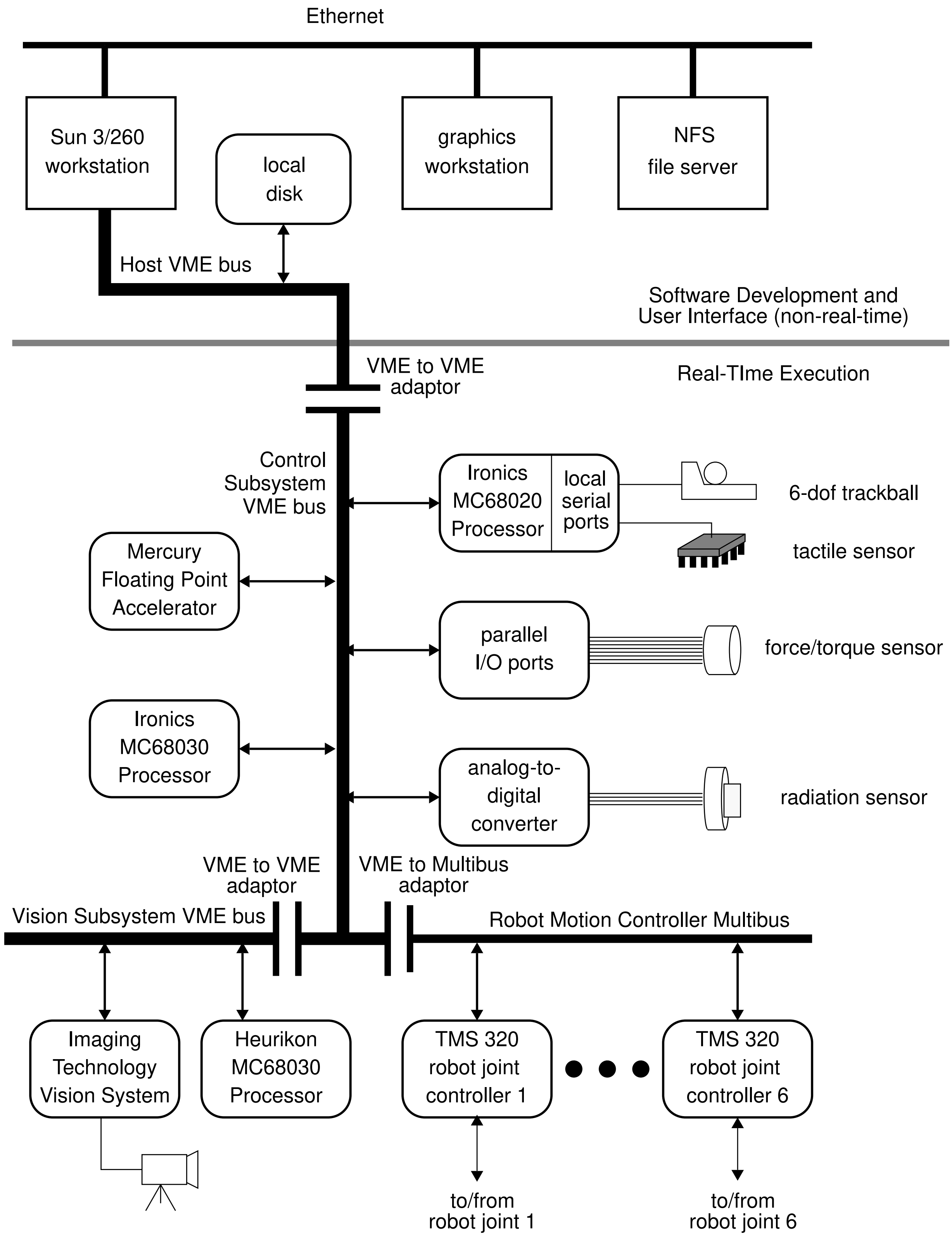

Figure 5: Block diagram of system components of theCMU Direct Drive Arm II