A Distributed Embedded System for Modular Self-Reconfigurable Robots

Arvind Seshadri, Marcos Ferretti, Andres Castano and Peter Will

Abstract— *Self-reconfigurable or metamorphic robots are modular robots that can change their shape and size. Such capability is desirable in situations where robots may encounter unexpected obstacles or difficulties and to perform tasks that are difficult for fixed-shape robots. We want to build homogeneous metamorphic robots with intra and inter-robot metamorphic capabilities. Fulfillment of this objective calls for small-size, self-sufficient and low power modules, with enough computing power to run relatively sophisticated control algorithms, using mostly commercial-off-the-shelf components. Sophisticated programs need to be uncoupled from hardware management to make them easy to develop and maintain. Additionally, the modules need an on-board operating system, small in size and with a low overhead, able to run on a small-memory microcontroller while still providing a sufficient functionality. In this paper, we discuss the design of the electrical subsystem, the operating system and the communication mechanisms of the modules of our robot.*

I. Introduction

Self-reconfigurable robots could perform jobs such as earthquake search and rescue, battlefield surveillance and scouting and, space and interplanetary exploration, where the robots may face unexpected situations or obstacles, or may need to perform tasks too difficult for fixed-shape robots. To perform these tasks the robot has to be self-sufficient where we use the term to describe a robot with the on-board hardware needed to operate untethered.

We want to build metamorphic robots with both intra and inter-robot metamorphic capabilities [6]. The requirement of self-sufficiency requires that robots formed by a split (i.e., an operation in which a robot is divided into two smaller robots) are self-sufficient. Since the simplest split operation is one in which a robot splits a single-module robot then each module must itself be self-sufficient: it must have a CPU, power supply and control over its sensors and actuators. In addition, the modules need a communication system to coordinate their actions for global control of the robot and other inter-module communication. This system must provide an error correction mechanism that would permit inter-robot communication in noisy environments, where retransmission is not an effective option.

Our robots are homogeneous, i.e., all modules are identical and hence the position of a module in the robot determines its function. This makes our robots larger in size compared to metamorphic robots with heterogeneous modules. In general, we want each module to be as small in size and weight as possible, since some actions require a single actuator to move multiple modules. Weight and size considerations limit us to use *small, low power batteries* to power the modules.

Arvind Seshadri is with the USC Information Sciences Institute. E-mail: arvind@isi.edu

Marcos Ferretti is currrently at Dept of EE-Systems, USC. E-mail: ferretti@usc.edu

Andres Castano is now at the Jet Propulsion Laboratory. E-mail: Andres.Castano@jpl.nasa.gov

Peter Will is with the USC Information Sciences Institute. E-mail: will@isi.edu

The control of the robot can centralized or distributed. In the centralized control, either the CPU of one the modules or a remote host act as a master. In contrast, distributed control makes use of the collaboration of agents local to each module. The design of the module does not make any assumptions about the type of global control. Still, different control methods place different demands on the processing power of the CPU of the modules. Typically, global control with a remote host as master requires a small computational power while both the distributed control and the centralized control with a local master usually have high processing requirements. The CPU used by the modules has to be powerful enough to meet the requirements of the different control methods in addition to being able to control all the actuators and sensors and the communication system of the module.

We found that 16-bit microcontrollers provide an effective compromise in terms of *processing power, memory size, on-chip peripherals, size* and *power consumption*. Although much more powerful, 32-bit microcontrollers are not appropriate alternatives due to their size and power consumption, and their assets, high clock speeds and a wide variety of peripherals, would not be used by the module.

With respect to the software, the modules need a real-time operating system (RTOS) to facilitate the development and maintenance of application programs by uncoupling them from the underlying hardware. The RTOS must provide a sufficient functionality and be *predictable, efficient and small* since it has to be deployed on 16-bit microcontrollers that typically have between 20–32 Kbytes of memory.

Many commercial vendors have products like RTXC [8] and VxWorks [27] with small, real-time kernels. Unfortunately, the majority of these vendors support only 32-bit microcontrollers and therefore they could not be used as the RTOS of our modules. Furthermore, those few that were partially suitable required a substantial effort to port them to an unsupported microcontroller. Hence, we decided to develop a custom RTOS, relatively hardware independent and easy to port to any future microcontrollers that we may use. Barring some deviations, we used the UNIX [2] design philosophy for our kernel, borrowing ideas and algorithms from Chimera [11], Linux [4], REAL/IX [9] and EMERALDS [28].

Our work also provides some guidelines for system design issues in other areas of distributed embedded systems like sensor networks and ubiquitous/pervasive computing. As the Internet and computer networks become more prominent and connectivity becomes more easily available, these areas are becoming increasingly feasible and important. Like our modules, these applications use small microcontrollers. To ensure the correctness of these application programs, usually sophisticated and safety critical, it is necessary to use a RTOS that uncouples them from

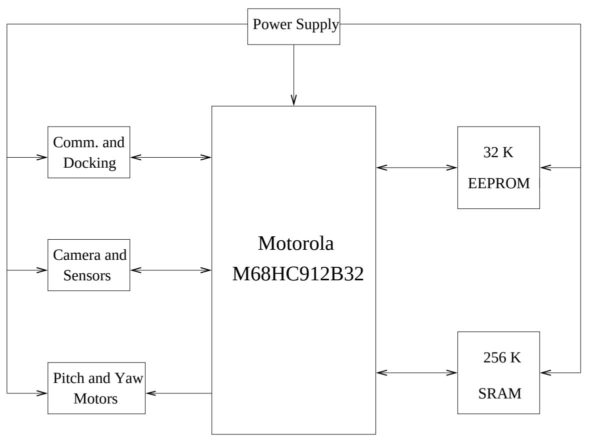

Fig. 1. Electrical Block Diagram

direct hardware management and facilitates their development and debugging [13]. Hence, system designers in these areas face some of the issues that we addressed during our system design stages.

In the next section, we describe the electrical design of the modules. Sections 3 and 4 describe our operating system and its performance, respectively. In Sec. 5 we compare our system with the Handy Board [14] and EMERALDS [28]. Finally, we present our conclusions in Sec. 6.

II. Electrical Design

Our implementation of a reconfigurable robot is called CONRO. As shown in Fig. 1, the five main units of its modules are the microcontroller and memory, the communication and docking unit, the power supply unit, the pitch and yaw motors, and the module sensors.

The first generation of CONRO modules used the Basic Stamp II as their CPU [5],[18]. Although this CPU presents advantages for prototyping and with respect to power consumption, it has shortcomings such as a limited program and data memory, a lack of interrupts, dedicated PWM generators and ADC, and a number of general purpose I/O ports that is insufficient for our needs.

Our second generation of CONRO modules, aimed to overcome these problems, is being designed around the Motorola 68HC12B32 [17] and two external memories, a 32K Atmel AT28HC256 EEPROM [1] for persistent storage and a 256K Toshiba TC55V2001STI-85L SRAM [25]. This new system satisfies all the requirements of the module. It allows us to drive the infrared (IR) pair on the faces of the modules used for serial communication between modules and docking tasks [6] and provides a RS-232 link for communication with a remote host. It provides interrupt capabilities that allows the module to handle asynchronous tasks. Likewise, the 68HC12 has dedicated PWM generators that control the pitch and yaw motors of the module which are high torque-to-weight ratio commercial-off-the-shelf Futaba S3102 RC servos. Finally, the CPU provides enough I/O ports to drive our module sensors such as a camera, tilt or touch sensors.

III. Operating System Kernel

Our kernel deviates from the UNIX philosophy in that user programs can be made aware of hardware events if they want to. UNIX provides user programs with an abstract, high-level view of the hardware through the file system. However, in real-time systems some programs may want to be notified of hardware events, since it may be the only way to ensure timely service of such events. Hence, we designed our kernel so that there is a mechanism by which user programs that want to react directly to hardware events can do so with very little overhead and in a consistent manner.

The 68HC12 does not have any hardware memory protection mechanism. Thus, a process with run-time errors can easily crash other processes or the kernel itself. Since application code changes frequently, we have to protect the kernel and user processes from run-time errors in other user processes. We achieve this by paging the external SRAM to give each user process its own address space.

The 68HC12 has a 64K address space. The external EEPROM, peripheral control registers and the 68HC12's internal SRAM and EEPROM are mapped to the upper 32K of this address space, as shown in Fig. 2. The lower 32K holds one of the external SRAM pages.

The kernel code resides permanently in the external EEPROM which is write protected. Along with the kernel code, the EEPROM contains the system call library code that is shared by all user processes. The EEPROM also holds executable program images. These images are used to spawn user processes at system start-up, as discussed in Sec. III-A.1.

The kernel allocates one of the eight available RAM pages for its own use. Each of the other seven pages serves as the user address space of a process. This effectively separates the address spaces of all processes from each other and from the kernel data segment. Having one process per page means that the system can support a maximum of seven processes at any given time. We found this number to be quite sufficient for our use.

The limit of a maximum of seven user processes means that many of the kernel data structures can be statically allocated as arrays. Also, in the absence of any hardware support for enforcing protection, there is little that the kernel can do to curtail activities of malicious user programs. Hence, the kernel does not support access control and authentication mechanisms. Both these factors lead to smaller and faster kernel code thus reducing the kernel overhead.

As seen from the memory map (see Fig. 2), the kernel code is permanently mapped into the address space of all user processes. Hence, the switch from user to kernel mode during system calls just involves a call to the entry point in the kernel. There is no need to use a trap to change the processor mode, since the 68HC12 does not have any hardware mechanisms to enforce protection. Thus, system calls have overheads similar to subroutine calls making them very efficient.

The kernel is fully preemptive. It handles concurrency by using semaphores to regulate access to all its shared data structures. Kernel semaphores use priority inheritance [20] to avoid the unbounded priority inheritance problem [24].

We now describe the two main subsystems in the kernel, the process control subsystem and hardware control subsystem,

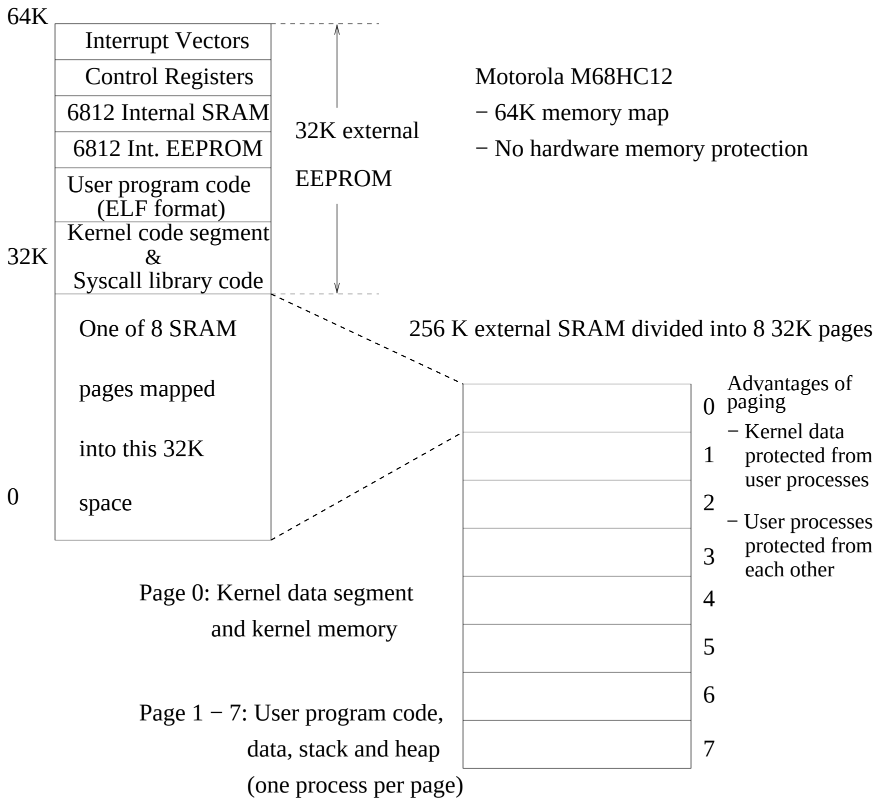
64K
Interrupt Vectors
Control Registers
6812 Internal SRAM
6812 Int. EEPROM
User program code (ELF format)
Kernel code segment & Syscall library code
32K
32K external EEPROM
Motorola M68HC12
− 64K memory map
− No hardware memory protection
One of 8 SRAM pages mapped into this 32K space
0
256 K external SRAM divided into 8 32K pages
0
1
2
3
4
5
6
7
Advantages of paging
− Kernel data protected from user processes
− User processes protected from each other
Page 0: Kernel data segment and kernel memory
Page 1 − 7: User program code, data, stack and heap (one process per page)

Fig. 2. Memory Map (not to scale)

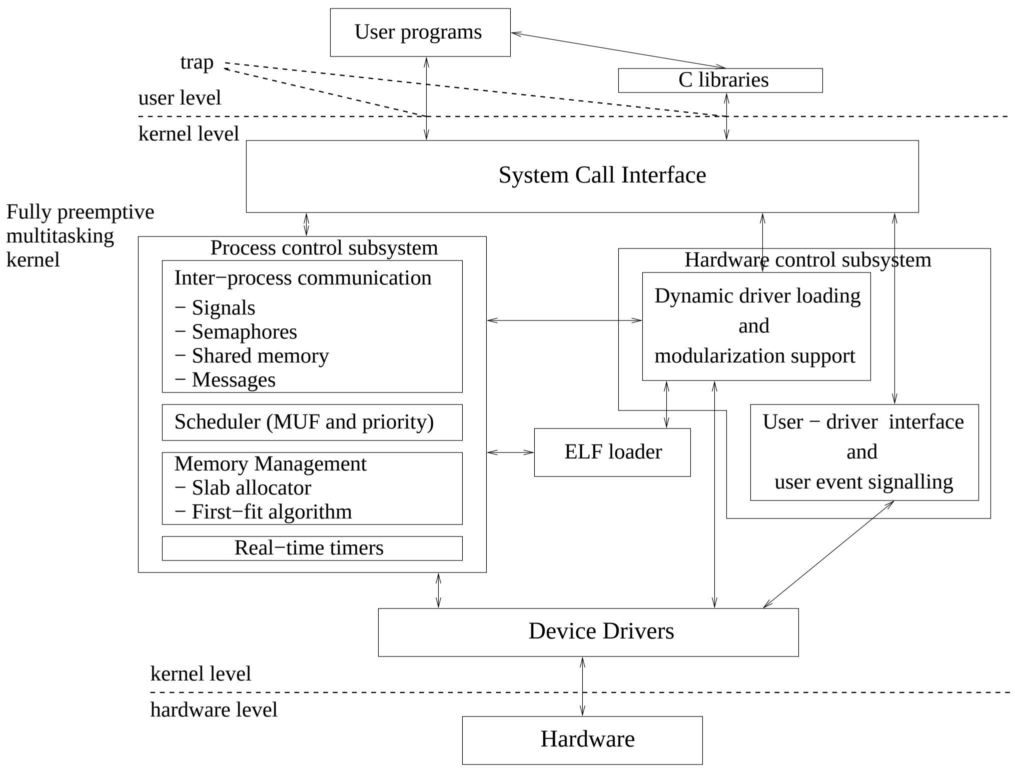

Fig. 3. OS Kernel Block Diagram

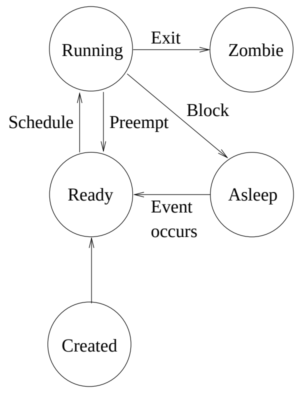

Fig. 4. Process States and State Transitions

shown in Fig. 3.

A. Process Control Subsystem

The process control subsystem is responsible for process scheduling, memory management, IPC mechanisms and timer management.

A.1 Scheduler and Process Management

During initialization, the kernel allocates an array of 8 process control block (PCB) entries. Each PCB entry is 128 bytes in size and includes the process kernel stack.

After the kernel initializes itself, it creates the null process "by hand" using PCB entry zero for it. The null process creates user processes by forking and execing the executable images in the external EEPROM. After spawning user processes, the null process goes to sleep awaiting one of the following three events: 1) it is scheduled when the ready list is empty, 2) it is periodically scheduled to run by the system to remove zombies and, 3) the slab allocator, discussed in Sec. III-A.2, may schedule it to run when it runs low on free objects in a particular cache.

Figure 4 shows process states and state transitions. New processes being created by the fork() call are in the created state. When fully created, they transition to the ready state. The highest priority ready process is selected by the scheduler to run. A running process may enter sleep state to wait for an event. Its PCB is added to the corresponding wait queue. The kernel module managing the wait queue wakes up the sleeping processes when the event occurs and then calls the scheduler. A process that exits enters the zombie state. Its parent may learn of its termination by issuing a wait() call.

The kernel has an ELF loader [16], [26] to handle the exec() system call and the insmod() system call that is used for dynamic device driver loading, as described in Sec. III-B.1.

The scheduler uses two scheduling algorithms (Fig. 5). The maximum-urgency-first (MUF) algorithm [23] schedules the critical set, that is computed offline. All processes in the critical set have a 3-bit criticality [23], a 3-bit dynamic priority [23] and an optional 3-bit user priority [23]. These three values are used to compute a 9-bit urgency value [23]. Processes not in the critical set are assigned a criticality value of zero. Users may specify a user priority for such processes.

Processes outside the critical set are scheduled by a fixed-priority scheduler based on their assigned user priority. Processes having the same priority are normally scheduled in a round-robin fashion by time-slicing where a time-slice is 4096μs on a 2MHz 68HC12. Time-slicing is optional and can be turned off by user programs.

The MUF scheduler allows the fixed-priority scheduler to run if there is time left over after all processes in the critical set have been scheduled. Thus, as far as the utilization of the critical set is less than one, processes outside the critical set will be run.

The fixed-priority scheduler is also useful because it gives users the ability to use the system as an event driven system, efficiently. To implement an event driven system, the users can disable the MUF scheduler by setting the criticality of all processes to zero. The kernel will then use the fixed priority scheduler for all processes. This leads to a lower scheduling overhead compared to using the MUF scheduler for this purpose.

A.2 Memory Management

This module consists of the kernel memory allocator and the user memory manager, as shown in Fig. 6.

The kernel memory allocator uses two methods, the slab allocator and first-fit algorithm.

The slab allocator [3] maintains caches of objects of the most frequently requested sizes. This reduces internal fragmentation and makes allocation and deallocation fast and makes it suitable for handling real-time memory requests. At system start-up, the slab allocator allocates caches for all commonly requested memory sizes. Device drivers may create their own object caches if they have special requirements.

As mentioned earlier, the slab allocator schedules the null process to allocate slabs run whenever it runs low on free objects in a particular cache. However, after profiling the kernel, we have been able to come up with slab sizes for the commonly used caches, such that they rarely run out of free space. This is possible because of the fixed and relatively small limit on the maximum number of processes.

The first-fit algorithm [12] handles non-real-time block memory requests from the program loader and hardware control subsystem. It also allocates new slabs to object caches of the slab allocator on receiving requests from the null process.

The user memory map consists of the process code segment, followed by initialized and uninitialized data of the process. This is followed by the heap. The stack pointer is initialized to top of user memory. Space in the heap is allocated and deallocated by brk(). User program obtain dynamic memory through calls to malloc().

A.3 Interprocess Communication

Interprocess communication mechanisms are signals, semaphores, shared memory and message passing (Fig. 7).

Signals are based on the signal semantics defined in POSIX [10]. In addition, signals have priorities and multiple signals of the same type are considered to be distinct. Signals also indicate the event or process that generated the signal. The priority of a received signal is a 16-bit value, the upper eight bits coming from the priority of the event or process that generated the signal and the lower eight bits coming from the priority of the signal.

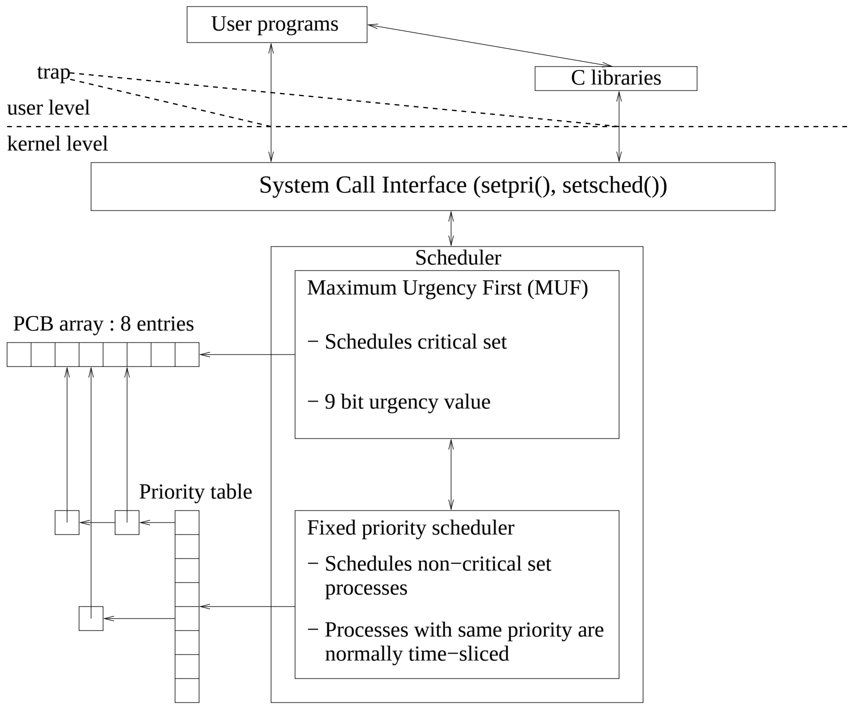
User programs
C libraries
trap
user level
kernel level
System Call Interface (setpri(), setsched())
Scheduler
Maximum Urgency First (MUF)
− Schedules critical set
− 9 bit urgency value
PCB array : 8 entries
Priority table
Fixed priority scheduler
− Schedules non−critical set processes
− Processes with same priority are normally time−sliced

Fig. 5. Scheduler

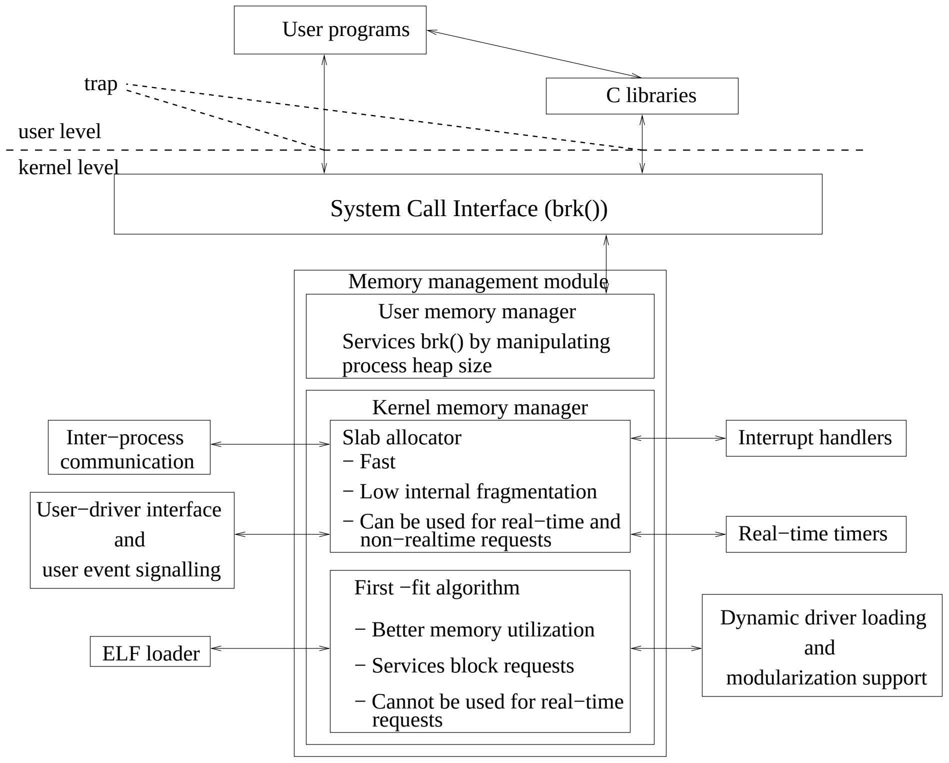
User programs
C libraries
trap
user level
kernel level
System Call Interface (brk())
Memory management module
User memory manager
Services brk() by manipulating process heap size
Kernel memory manager
Slab allocator
− Fast
− Low internal fragmentation
− Can be used for real−time and non−realtime requests
First −fit algorithm
− Better memory utilization
− Services block requests
− Cannot be used for real−time requests
Inter−process communication
User−driver interface and user event signalling
ELF loader
Interrupt handlers
Real−time timers
Dynamic driver loading and modularization support

Fig. 6. Memory Manager

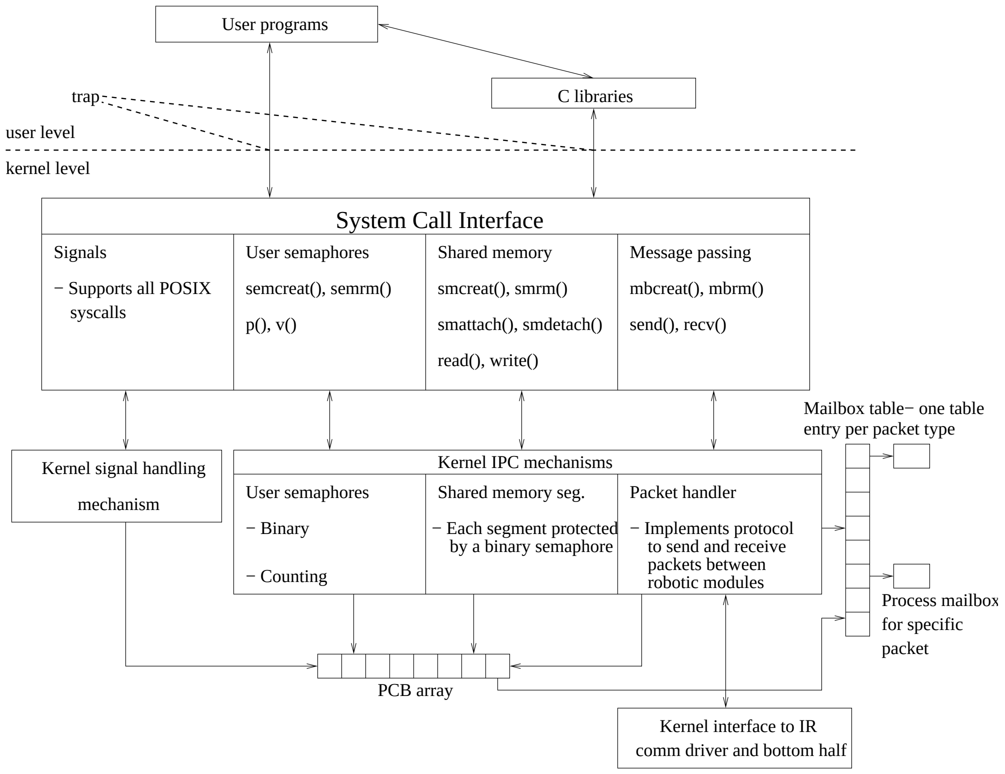

Fig. 7. Interprocess Communication Mechanisms

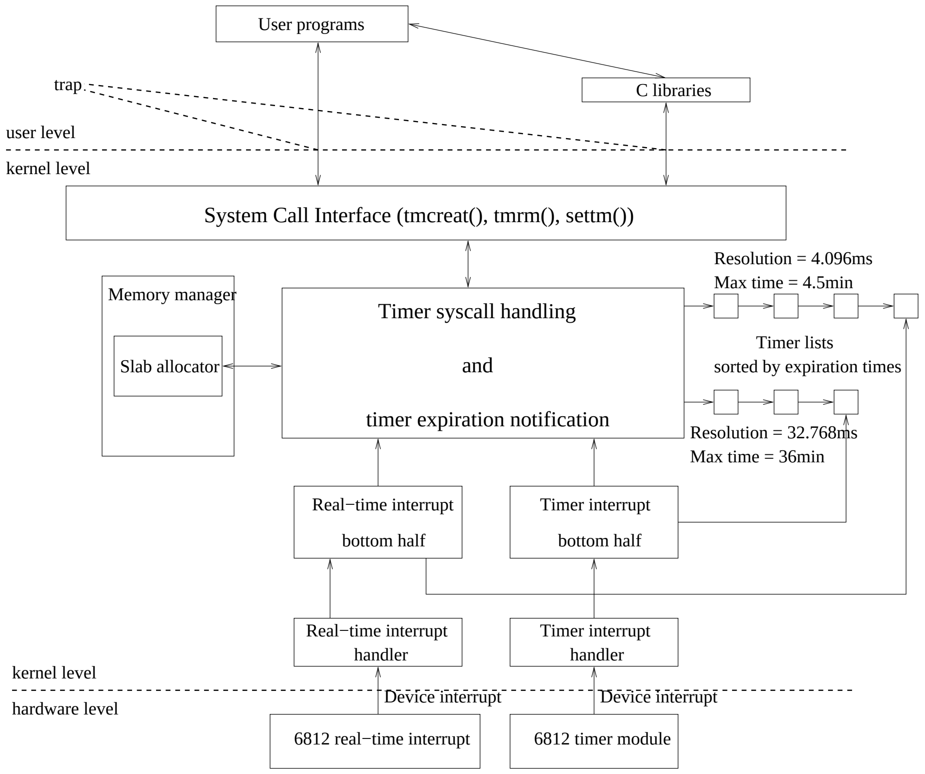

Fig. 8. Real-Time Timers

The highest priority non-masked signal is delivered to a process when it switches from kernel to user mode.

Both binary and counting semaphores are available to user processes. Priority inheritance [20] is used to avoid the unbounded priority inversion problem [24].

Shared memory provides a fast IPC mechanism for sharing data between processes running on the same module of a robot. Shared memory segments are allocated in kernel space and accessed through smcreat(), smattach(), smrm(), read() and write() system calls. They can optionally be protected by binary semaphores.

As mentioned in Sec. I, control of the robot can be centralized or distributed. In centralized control, the modules of a robot receive commands from a central controller, that they execute [7]. Distributed control uses hormones [21], which are content-based messages. The kernel provides a generic inter-module communication mechanism through mailboxes, that can be used with either control method. Processes create a mailbox for each type of hormone or command they want to receive. The kernel packet handler places a copy of a received packet in all mailboxes for that type of hormone or command and forwards the packet to all the neighboring modules of the robot.

There is a need for error detection and correction capabilities in the communication mechanism. Normally, the kernel ensures reliable communication by means of a selective repeat protocol [22]. This works well for inter-module communication between two modules that are part of the same robot. In this configuration, we hardly encounter any communication errors. However, for inter-robot communication, the errors encountered during IR data transfer depend on the distance between the robots and the noise in the environment. We find that even factors like strong fluorescent lighting lead to random bit errors in the data packets. If the selective repeat protocol is used for error recovery in noisy environments, the number of retransmissions required has mean value of three and a maximum value of eight. This means that the timing requirements for communication cannot be met using a selective repeat protocol. Since increasing the transmitted power is not an option due to the use of low power IR LEDs, we need error correcting codes for inter-robot communication.

The kernel uses (24,12) extended Golay code [15] that has the ability to correct up to three bit errors. Since error correction has computational and data overheads, its use depends on the operating environment. Application programs can explicitly ask the kernel to use error correcting codes for their messages or leave it up to the kernel to decide. In the second case, the kernel monitors the error rate of the communication channel and uses error correcting codes if the error rate crosses a threshold. It uses the selective repeat protocol otherwise.

A.4 Real-Time Timers

The kernel provides two types of dynamic real-time timers both varying in time from 1 to 65535 ticks but differing in tick duration (Fig. 8). The first variety uses the 6812's real-time interrupt [17] and has tick duration of 4.096ms (max time approx 4.5min). The other variety uses 6812's timer module [17] giving times in multiples of 32.768ms (max time approx 36min). Both timers can be one-shot or periodic. The timers are maintained as a list sorted by expiration times. The list implementation is simple and efficient since the number of dynamic timers in the kernel at any time is quite small.

User programs create and destroy timers using tmcreat() and tmrm(). There are two ways in which programs can be notified about timer expiration, they can sleep awaiting timer expiration or they can use the user event signaling mechanism of the kernel, as discussed in Sec. III-B.2.

B. Hardware Control Subsystem

This subsystem handles device driver modularization support and user event signaling.

B.1 Dynamic Driver Loading and Modularization

Device drivers can be compiled into the kernel code or linked dynamically with the running kernel (Fig. 9). Dynamic linking is preferred since it is not necessary to recompile and reload the kernel every time hardware is added or removed. This is similar to loadable kernel module support in the Linux kernel [4], [19]. After new hardware is added to a module, the object file for the device driver is transferred from a remote host to the module. The kernel allocates memory in kernel space for the driver and resolves all references to kernel symbols in the driver's object code by looking up its symbol table. The kernel then calls the driver's initialization routine. Drivers are removed by deallocating their memory after calling the driver's exit function.

B.2 User Event Signaling

User processes may have to be notified of occurrence of certain events. For example, if a process is driving one of the motors and the touch sensor interrupts the processor, then the process will have to be notified so it can stop the motor. This notification is performed through the user event signaling mechanism (Fig. 10) that is adapted from a similar mechanism in UNIX [9]. Processes that want to be notified of events register a call-back function and a flag variable with the kernel for the appropriate event. When the event occurs, the kernel calls the registered user function and sets the flag variable in user space.

This mechanism and the io_map() system call can be used to implement device drivers in user space very efficiently. In the absence of hardware protection mechanisms in the 68HC12, user processes can access device registers directly at the same speed as accessing them from the kernel. Hence, with a handler registered for the device interrupt, user space device drivers will be almost as fast as device drivers in the kernel.

The io_map() call is used to request device registers from the kernel. When the kernel receives this call from a user process, it checks that the registers being requested are not already in use by itself or another user process. If the registers are free it grants them to the requesting process.

C. Anatomy of a Device Driver

The 68HC12 has only one external interrupt pin. The kernel maintains all registered interrupt handlers in a chain and calls each of them in turn when an interrupt is received. Hence, the all interrupts remain disabled at least until the kernel finds the right handler for the interrupt. Thus, interrupts could be lost if the interrupt handler code is long and it keeps interrupts disabled

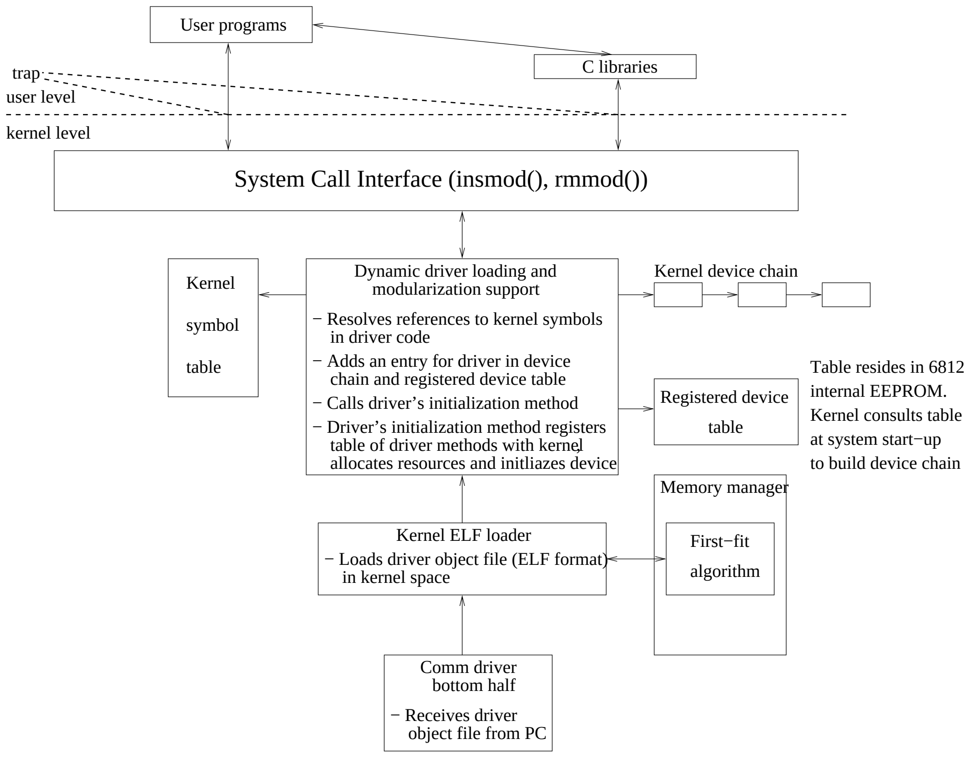
User programs
C libraries
trap
user level
kernel level
System Call Interface (insmod(), rmmod())
Kernel symbol table
Dynamic driver loading and modularization support
− Resolves references to kernel symbols in driver code
− Adds an entry for driver in device chain and registered device table
− Calls driver's initialization method
− Driver's initialization method registers table of driver methods with kernel allocates resources and initliazes device
Kernel device chain
Registered device table
Table resides in 6812 internal EEPROM. Kernel consults table at system start−up to build device chain
Memory manager
First−fit algorithm
Kernel ELF loader
− Loads driver object file (ELF format) in kernel space
Comm driver bottom half
− Receives driver object file from PC

Fig. 9. Dynamic Driver Module Loading and Linking

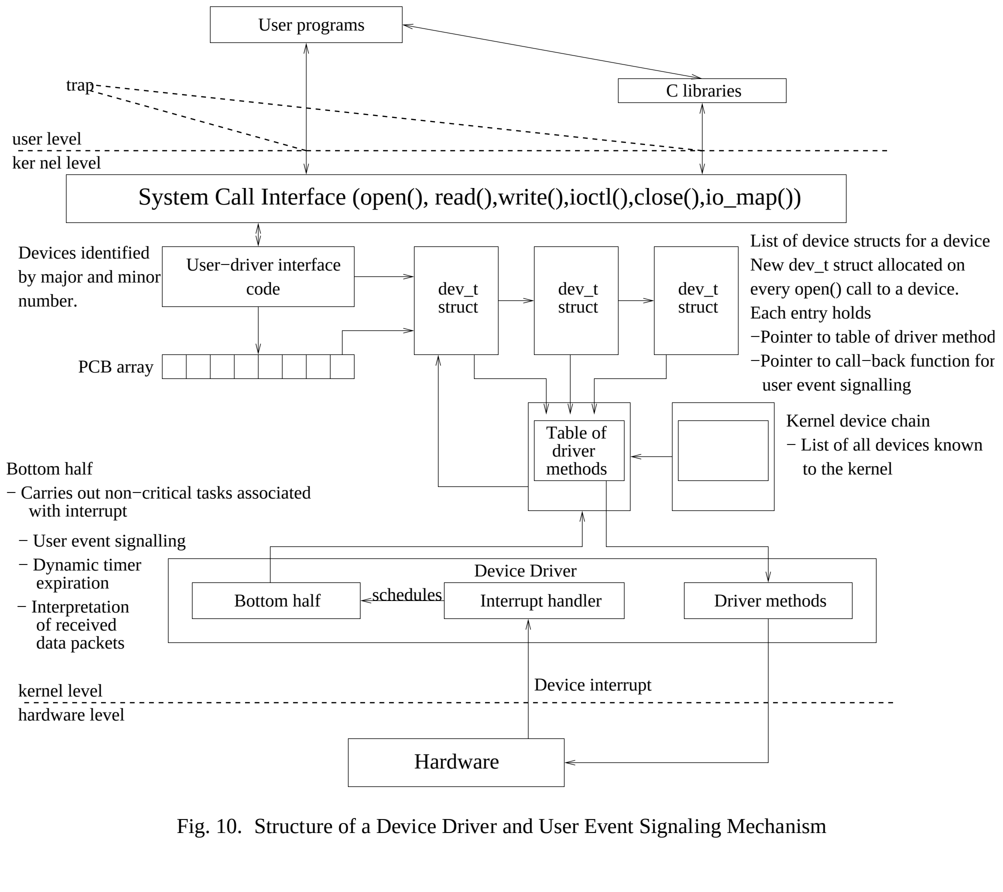
User programs
C libraries
trap
user level
ker nel level
System Call Interface (open(), read(),write(),ioctl(),close(),io_map())
Devices identified by major and minor number.
User−driver interface code
dev_t struct
dev_t struct
dev_t struct
List of device structs for a device
New dev_t struct allocated on every open() call to a device.
Each entry holds
−Pointer to table of driver method
−Pointer to call−back function for user event signalling
PCB array
Table of driver methods
Kernel device chain
− List of all devices known to the kernel
Bottom half
− Carries out non−critical tasks associated with interrupt
− User event signalling
− Dynamic timer expiration
− Interpretation of received data packets
Device Driver
Bottom half
schedules
Interrupt handler
Driver methods
Device interrupt
kernel level
hardware level
Hardware

Fig. 10. Structure of a Device Driver and User Event Signaling Mechanism

during its execution. One approach to alleviate the problem is to perform only critical actions in the interrupt handler and have another module do rest of the processing associated with the interrupt (Fig. 10). Hence, all device drivers, except the most trivial ones, have an interrupt handler and a bottom half [4], [19].

D. System Call Interface

The system call interface is implemented as a user library. The code in the library checks the system call parameters and calls the entry point in the kernel. It is also responsible for any clean-up to be done when the call returns. Most system calls pass their parameters to the kernel through processor registers. If the number of registers is insufficient, the extra parameters are copied to 68HC12's internal SRAM that acts as a buffer between user space and kernel space. The system call library code resides in the external EEPROM and is shared by all processes.

E. Bootloader

On power-up, the 68HC12 comes up in normal single chip mode [17]. In this mode, it sees only its internal memories. Hence the internal flash of the 68HC12 has the initial bootloader, that switches the 6812 to expanded narrow mode [17] and jumps to start of kernel code in the external EEPROM. The initial bootloader can also erase and program the external EEPROM, receiving the necessary data from a remote host. In addition it can switch the 68HC12 to a background debug mode [17] that is useful for debugging the kernel.

IV. Performance

The work discussed resulted in the construction of prototype hardware and an operating system that will be used by the second generation of CONRO modules. We expect the final version of the PCB to be about 20–25% bigger in area than the PCB of the Basic Stamp. The prototype board draws approximately 70-80mA at 5V. Since it is powered by a 6V K28L lithium battery with a capacity of 160mAH we get operating times of approximately 2 hours.

The kernel code is approximately 9 Kbytes in size. We use a 2MHz clock for the 68HC12 to keep power consumption at a minimum. At this clock speed, reschedule operation takes 156μs and 21μs on the MUF and round-robin scheduler respectively. The actual context switch takes 49μs. The interrupt handling code in kernel is able to service all interrupts without missing any scheduling deadlines, with seven user processes.

Inter-module communication between modules on the same robot is very reliable and fails only when the hardware fails. With a (24,12) extended Golay code, we did not find any errors in inter-robot communication during our tests, even in environments where we were previously encountering error rates as high as 20%. Hence, this scheme is faster than use of retransmissions in noisy environments even though it has higher computational and data overheads.

V. Comparison With Other Systems

The Handy Board [14] is a 68HC11 based controller board designed for experimental work in mobile robotics. Programs are written in a language called Interactive C, that is compiled by a custom compiler into pseudo code for a stack machine. The Handy Board has an interpreter for the pseudo code in the form of a resident mini-OS. Interactive C supports multitasking using a round-robin scheduler in the resident OS. There are C libraries for hardware interfacing functions like motor control, sensor input, IR communication, timer management and tone generation. An interpreted execution environment offers the advantage that programs can be checked for run-time errors. Thus, programs with run-time errors do not crash the system but this comes at the expense of program execution speed.

By following the UNIX design philosophy for our kernel, we are able to support a richer functionality compared to Interactive C, without sacrificing program execution speed. Writing programs only requires a C compiler capable of generating ELF executables for the 68HC12. We address in the issue of run-time errors in programs by separating the process address spaces (Sec. III). We are in the process of extending the debugging facilities of the kernel so that user programs can be debugged on the board using the background debug mode of the 68HC12. This should make it easy to track down and correct any run-time errors in programs. In addition, our kernel supports scheduling algorithms that enables us to handle real-time deadlines and events, a rich set of IPC mechanisms, a powerful memory management routines and real-time timers. User programs see hardware devices as files and access them using system calls like read() and write(), and the catch-all ioctl() call. Thus, neither the system call interface nor the C libraries change when new devices are added.

EMERALDS [28] is a real-time microkernel for small-memory microcontrollers that relies on the presence of a hardware memory management unit (MMU) to enforce memory protections. As of date, none of the 16-bit microcontrollers have a hardware MMU. But, in embedded systems all processes are co-operative and there are no malicious programs. Hence, it is possible, by paging the memory, to protect processes from errors in other processes. As the memory-chip-density-to-cost ratio becomes more favorable this method of memory protection looks more attractive on the smaller microcontrollers that do not have a hardware MMU. It has the added benefit of reducing the size of the kernel data segment since there are no page tables.

A second difference between EMERALDS and our kernel is the support for modular device drivers. This enables us to support dynamic reconfiguration of module hardware without using a microkernel architecture.

VI. Conclusion

As we worked with the current prototype, we found two limitations that we hope to overcome in the next generation hardware. First, we would like our system to work at lower voltages to prolong battery life and second, we would like to use a microcontroller with multiple external interrupt pins and a built-in programmable interrupt controller. We are looking for a suitable 3.3V microcontroller.

The operating system kernel is being enhanced by adding error and exception handling facilities similar to those in Chimera 3.2 [11] to our kernel. We will also be extending the debugging facilities of the kernel, so that user programs can be debugged on the board using the background debug mode of the 68HC12.

As microcontrollers become more capable with advances in VLSI technology and network connectivity becomes more easily available, embedded systems find new and interesting application areas, as we mentioned in Sec. I. We hope that by providing some insights on system design issues in these areas, this paper speeds up their development.

VII. Acknowledgements

We wish to thank Ramesh Chokkalingam and Nestoras Tzartzanis for their work on the initial design of the board. This research is sponsored by DARPA/MTO under contract number DAAN02-98-C-4032.

References

[1] Atmel Corp. *Atmel AT28HC256 Datasheet*, Dec 1999.

[2] Maurice Bach. *The Design of the UNIX Operating System*. Prentice-Hall of India Pvt. Ltd., 1996.

[3] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer 1994 Technical Conference*, pages 87–98, 1994.

[4] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly and Associates Inc., first edition, Jan 2001.

[5] Andres Castano, Ramesh Chokkalingam, and Peter Will. Autonomous and self-sufficient CONRO modules for reconfigurable robots. In *Proc. 5th Int'l Symp. Distributed Autonomous Robotic Systems*, pages 155–164, 2000.

[6] Andres Castano, Wei-Min Shen, and Peter Will. CONRO: Towards deployable robots with inter-robot metamorphic capabilities. *Autonomous Robots Journal*, 8(3):309–324, July 2000.

[7] Andres Castano and Peter Will. Representing and discovering the configuration of CONRO robots. In *Proc. of Intl. Conf. on Robotics and Automation*, May 2001.

[8] Embedded System Products Inc. *RTXC User's Manual*, 1995.

[9] Borko Furht, Dan Grostick, David Gluch, Guy Rabbat, John Parker, and Meg McRoberts. *Real-Time UNIX Systems: Design and Application Guide*. Kluwer Academic Publishers, 1991.

[10] IEEE. *Portable Operating System Interface for Computer Environments: IEEE Standard 1003.1*, 1990.

[11] Darin Ingimarson, David B. Stewart, and Pradeep K. Khosla. *Chimera 3.2: The Real-Time Operating System for Reconfigurable Sensor-Based Control Systems*, Feb 1995.

[12] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[13] Philip Koopman. Problems facing embedded systems. Talk Slides, 1999. Available on the web at http://www-2.cs.cmu.edu/ koopman/embedded.html.

[14] Fred G. Martin. *The Handy Board Technical Reference*, Nov 2000.

[15] M.J.E.Golay. Notes on digital coding. In *Proceedings of the IRE*, volume 37, page 657, June 1949.

[16] Motorola Inc. *M68HC12 Embedded Application Binary Interface*, first edition, 1998.

[17] Motorola Inc. *M68HC12B Family Advance Information*, third edition, 2001.

[18] Parallax Inc. *BASIC Stamp Programming Manual*, second edition, 2000.

[19] Alessandro Rubini. *Linux Device Drivers*. O'Reilly and Associates Inc., first edition, Feb 1998.

[20] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Computers*, 39(3):1175–1198, 1990.

[21] Wei-Min Shen, Yimin Lu, and Peter Will. Hormone-based control for self-reconfigurable robots. In *Proc. Intl. Conf. Autonomous Agents*, 2000.

[22] William Stallings. *High-Speed Networks: TCP/IP and ATM Design Principles*. Prentice Hall, first edition, 1997.

[23] David B. Stewart and Pradeep K. Khosla. Real-time scheduling of sensor-based control systems. In *IEEE Workshop on Real-Time Operating Systems and Software*, pages 144–150, May 1991.

[24] H. Tokuda and T. Nakajima. Evaluation of real-time synchronization in real-time mach. In *Second Mach Symposium*, pages 213–221. Usenix, 1991.

[25] Toshiba Corp. *Toshiba TC55V2001STI Datasheet*, Mar 1998.

[26] UNIX System Laboratories. *System V Application Binary Interface*, fourth edition, 1995.

[27] Wind River Systems. *VxWorks Programmer's Guide*, 1993.

[28] Khawar M. Zuberi and Kang G. Shin. EMERALDS: A small-memory real-time microkernel. *IEEE Trans. on Software Engineering*, 1999.