September 1988

RK: A Real-Time Kernel for a Distributed System With Predictable Response

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Robert King
University of Pennsylvania

Richard Paul
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-78.

RK: A Real-Time Kernel for a Distributed System With Predictable Response

Abstract
Robotics applications must execute in real-time. In addition, complex robotics applications include many physically distributed components such as manipulator arms and sensors. This paper presents the real-time kernel RK which is designed to facilitate the development of a distributed sensory system with multiple arms and sensors. The goal of the kernel is to support distributed applications that require predictable timing behavior. Our kernel design guarantees predictable response times by scheduling processes and communications based on timing constraints. In addition, the kernel provides a set of primitives that can be used to implement applications requiring predictable timing behavior. These primitives allow the specification of timing requirements that can be guaranteed in advance by the scheduler and the direct control of devices by application processes for faster and predictable feedback control. To illustrate the use of our kernel, this paper also describes a multiple sensory system which is being ported to our distributed test-bed.

Keywords
distributed robot programming, distributed real-time system, multi-sensory systems, distributed computing, real-time scheduling, timing analysis

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-78.

RK: A Real-Time Kernel For A Distributed System With Predictable Response

MS-CIS-88-78
GRASP LAB 155

Insup Lee
Robert King
Richard Paul

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

October 1988

RK: A Real-Time Kernel for a Distributed System with Predictable Response*

Insup Lee, Robert King and Richard Paul†

General Robotics and Active Sensory Perception Laboratory
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389

September 28, 1988

Abstract

Robotics applications must execute in real-time. In addition, complex robotics applications include many physically distributed components such as manipulator arms and sensors. This paper presents the real-time kernel RK which is designed to facilitate the development of a distributed sensory system with multiple arms and sensors. The goal of the kernel is to support distributed applications that require predictable timing behavior. Our kernel design guarantees predictable response times by scheduling processes and communications based on timing constraints. In addition, the kernel provides a set of primitives that can be used to implement applications requiring predictable timing behavior. These primitives allow the specification of timing requirements that can be guaranteed in advance by the scheduler and the direct control of devices by application processes for faster and predictable feedback control. To illustrate the use of our kernel, this paper also describes a multiple sensory system which is being ported to our distributed test-bed.

Keywords: Distributed robot programming, distributed real-time system, multi-sensory systems, distributed computing, real-time scheduling, timing analysis.

---

*This research was supported in part by NSF DCR 8501482, NSF DMC 8512838, NSF MCS 8219196-CER and NSF CCR-8716975. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

†The authors may be contacted via electronic mail at the addresses *lee@central.cis.upenn.edu*, *king@grasp.cis.upenn.edu* and *lou@grasp.cis.upenn.edu*, respectively. In addition, the authors may be contacted at their offices by telephone (area code 215) 898-3532, 898-2911 and 898-1592, respectively.

1 Introduction

Most robotics application programs must not only be logically correct, but they must satisfy certain timing constraints. The adherence to these timing constraints are important for two reasons. First, the physical characteristics of manipulators and sensors require regular, carefully timed feedback control for continuous and smooth operations. Second, the high-level tasks may require timely execution to avoid possible catastrophic results. In addition to being real-time, the requirements on timely computations of many numerical and symbolic functions are such that they can only be met using multiple processors. Furthermore, complex robotics applications include many physical devices that are distributed across multiple processors. This distributed view of the underlying system allows robotics applications to be implemented as relatively independent processes which run asynchronously except for occasional synchronization and communication. Concurrent programs such as these, which need to respond to external and internal stimuli within specified deadlines, are called distributed real-time programs [1].

As an example, consider a distributed sensory system consisting of three subsystems: tactile, camera and fusion. Given a world model, the goal of the system is to explore the world and update the world model using sensory data. To support the sensing, two end-effectors, one with a tactile sensor and another with a camera, are attached to two six-joint manipulator arms. The fusion subsystem integrates the state information from each arm and the data from the two sensors. It then updates the world model based on the integrated data. Using the world model, the camera subsystem chooses the next viewpoint. The robot arm then moves to the appropriate position so that the camera subsystem can obtain an image from that viewpoint. Similarly, the tactile subsystem uses the world model to choose a sensor viewpoint and then moves its arm to obtain the desired information. Both subsystems then send their sensory data and position of their respective arms to the fusion subsystem for integration into the world model. As we describe later, this system is naturally distributed and must meet various timing constraints in order to operate.

To facilitate the development of robotics applications that require real-time capabilities, we have been designing and implementing a distributed real-time kernel (RK) that supports the timely execution of time critical processes and communications. Our research is motivated by the difficulties encountered when we attempted to implement time dependent distributed programs such as a distributed sensory system using existing operating systems. These difficulties arose because most operating systems are designed to provide good average

performance, while possibly yielding unacceptable worst-case response times. Furthermore, they provide a limited set of primitives for dealing with time, making it impossible to implement programs whose correctness depends on exact timing.

To support real-time applications, RK is designed to exhibit predictable timing behaviors and to schedule processes and messages based on their timing constraints. In addition, message delay and interrupt handler overhead is bounded. The kernel also allows the programmer to express timing constraints explicitly. The explicit specification of timing constraints makes real-time programs easier to write and maintain than those without explicit timing constraints. Furthermore, the kernel can use timing constraints for scheduling processes and message transmission and reception. For example, many real-time kernels use a preemptive priority driven scheduler rather than one that is time driven. Since timing constraints are implicit in these kernels, the programmer must assign priorities to processes so that all the implicit timing constraints are met. Assigning proper priorities is a non-trivial task and, even if the best priority assignment is used, the assigned priorities can potentially result in very low processor utilization [2]. Priority based real-time programming also makes it difficult to detect timing errors since their effect may become apparent only at later time. However, if explicit timing constraints are used with a predictable kernel, timing violations can be detected as they happen and the kernel can schedule processes and messages so that as many timing constraints as possible are satisfied.

The remainder of this paper is organized as follows. The next section discusses the architecture of our distributed real-time system. The distributed real-time kernel is described in Section 3; we restrict the discussion to time management, scheduling, communication and device management. Section 4 illustrates the features of our kernel using the previously described sensory system. Our description of this distributed sensory system includes a discussion of its communication and timing requirements. Section 5 summarizes the measured performance and discusses the bounding of execution behavior. The final section contains our conclusions and remarks regarding distributed real-time systems.

2 Overall Architecture

Our real-time kernel is used to implement a distributed sensory system. The architecture for this distributed sensory system is divided into two levels to reflect its logical structure and is shown in Figure 1. At the high level, the system consists of five MicroVAXes connected through a 10 Mb Ethernet. These MicroVAXes use our kernel to provide the computation

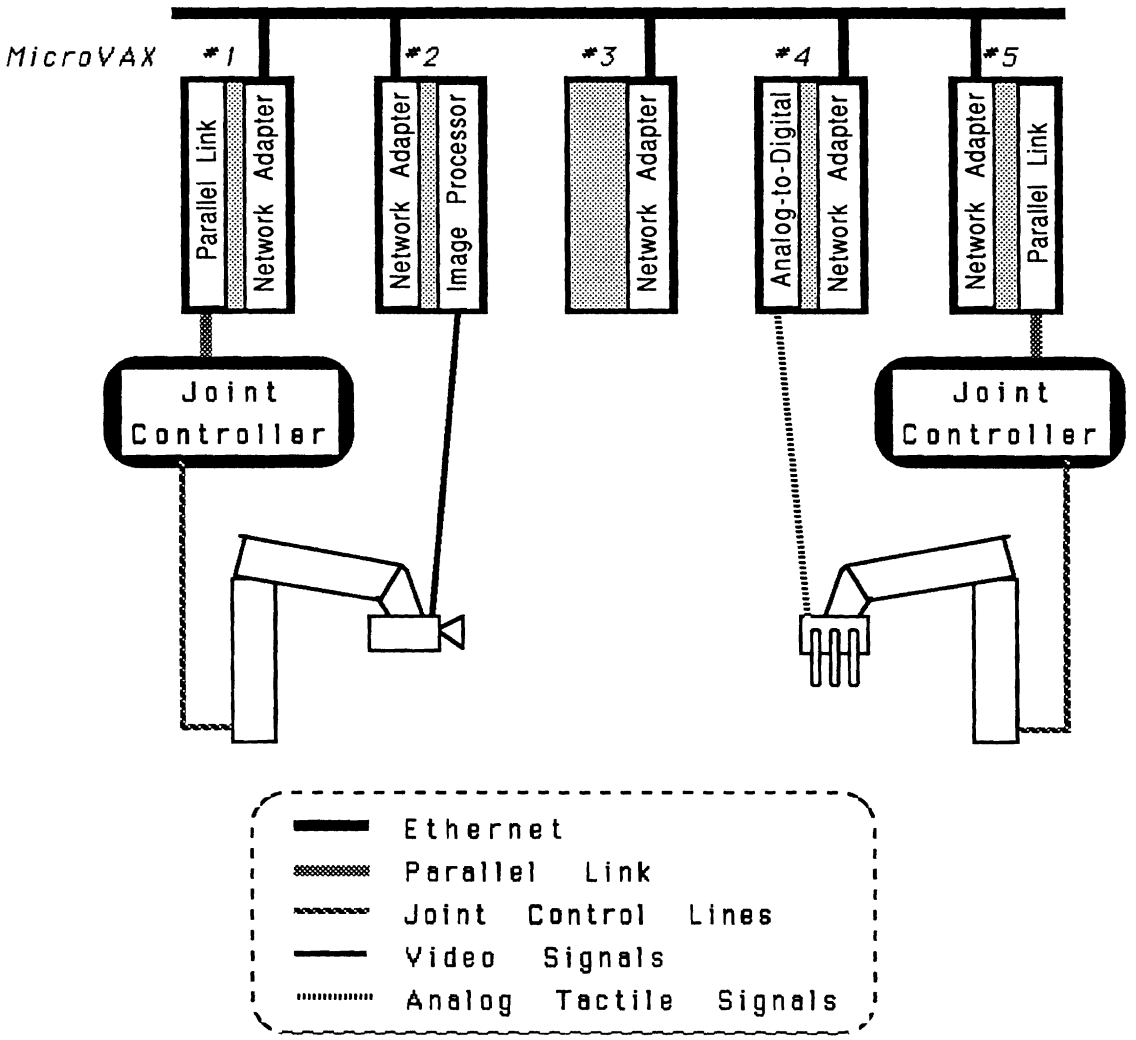

Figure 1: Architecture of the Distributed Sensory System

power needed for the timely execution of processes used in sensing such as deciding where to probe next, maintaining the world model and determining the path for the robot arm. At the low level, there is a joint controller for each manipulator arm and the two sensors, a camera and a tactile sensor. The joint controller consists of a supervisor processor and a joint processor for each joint of the manipulator arm and is connected to a MicroVAX via a parallel interface. The joint processors take the commands received from the MicroVAX via the supervisor processor and use them to control the joint motors. The output of each probe of the tactile sensor is an analog signal. This signal is converted to a digital value by the analog-to-digital converter board in a MicroVAX. The image processor reads the camera image in a frame buffer which is accessible from the MicroVAX as though it were memory.

3 Distributed Real-Time Kernel

As described in the introduction, our goal is to develop a real-time kernel (RK) that supports real-time applications with predictable process execution and interprocess communication. To provide predictable behavior, our kernel provides services with predictable timing behavior; that is, their worst case execution time is bounded. In addition, the kernel allows the programmer to specify timing constraints for process execution and interprocess communication. These timing constraints directly reflect the timing requirements of the application. The kernel uses these constraints for scheduling processes and communications. This approach of treating time explicitly facilitates the implementation and debugging of time dependent application programs.

The rest of this section describes the salient features of the kernel. We first describe the basic process structure. Then, we explain how to specify timing constraints and schedule processes based on them. Next, we describe the three types of interprocess communication: *signals* for timing, system and processor errors; *timed events* for synchronization, periodic events and device interrupts; and *ports* for message communication; these communication primitives allow the propagation of timing constraints from the sender to the receiver. Finally, we show how application processes directly control devices.

3.1 Process Model

A distributed real-time application consists of a set of communicating processes and a set of hardware devices. A process is defined by the programmer and represents a logically independent execution thread of control. Each process has an independent address space,

and consists of a main body, a set of event handlers and a read-only page of kernel specific data. All processes are initially non real-time processes and start executing with their main body. Event handlers execute either periodically or asynchronously when the process receives the corresponding events. A process becomes a real-time process when it either specifies a timing constraint or receives a timed event. Within a real-time process, scheduling of the main body and events handlers depends on their timing constraints.

Every process has access to a read-only page of kernel specific data, which includes the time of day that the system was booted, the current time of day, and port ids for various services, such as the nameserver. One key benefit of this page is that the current time of day is accessible without the overhead of a system call. User processes repeatedly read the current time of day until two consecutive readings return the same value. This ensures that the current time has not been corrupted by a system update.

The kernel provides the system services through either *system calls* or *server processes.* In order to bound the execution overheads of system calls, services provided through system calls are kept to a minimum. System calls are used only for services that require the crossing of address boundaries or predictably fast response. They include memory management, process switching, signals, timed events, alarms and interprocess communication. Server processes provide non-time critical services such as process creation, terminal input/output and device allocation. An application process sends a service request to a server process and waits for a reply if needed.

There are two kinds of devices: system devices, which are an integral part of the kernel; and applications devices, which are only pertinent to a particular application. System devices, such as clocks and network adapters, are managed by the kernel and used indirectly by many application processes. Application devices are directly controlled by particular application processes and include the analog-to-digital conversion board required for the tactile sensor and the image processing board required for the camera. The kernel converts a device interrupt into a timed event and provides shared memory between a device and a process.

3.2 Scheduling and Timing Constraints

RK supports both real-time and non real-time processes. In addition, real-time processes are prioritized as imperative, hard real-time, and soft real-time since not all real-time processes are equally important [3]. Processes are executed in the order of priorities. Within the same priority, imperative processes are executed on a first-come-first-serve basis, whereas hard and soft real-time processes are executed based on their timing constraints. The difference

between hard and soft timing constraints is that hard constraints must be scheduled in advance and if accepted, they are guaranteed to be met by the system. Soft timing constraints are not guaranteed to be met by the system and are considered less critical than hard timing constraints. When no real-time processes are ready, non real-time processes are executed on a first-come-first-serve basis.

Requirements for real-time processing can be viewed as the time when certain processing has to take place, for how long and how soon. There are two kinds of timing constraints: periodic and sporadic [4]. A periodic timing constraint becomes effective at regular intervals and a sporadic timing constraint can be imposed on at any time. These timing constraints are defined on the whole process or on part of the process. They are either explicitly requested by a process or implicitly specified when a process receives a message with a timing constraint.

Temporal Scope To facilitate the programming of timing constraints, RK supports a notion of *temporal scope*, which identifies explicit timing constraints with a sequence of statements and event handlers [1]. Each temporal scope consists of five attributes: a hard or soft timing constraint flag, start time, maximum execution time, deadline and a unique id. Whenever a temporal scope with a hard timing constraint is entered, the *scheduler* is consulted to check whether the adherence of the hard timing constraints can be guaranteed. The scheduler uses all previously approved hard real-time constraints when determining whether the requested constraint can be guaranteed.

Associated with each real-time process is a stack of temporal scopes which allow timing constraints to be nested but not overlapped. Only the current timing constraints, that is, timing constraints on top of the stack, are used for the scheduling of processes. To ensure that the adherence of the current timing constraint does not violate the timing constraints of nested scopes, we maintain the following invariance among nested timing constraints: the timing constraint of an enclosing scope can never have a later start time, a smaller execution time or an earlier deadline than its nested temporal scopes. In addition, the timing constraints nested within a hard temporal scope must all be hard.

There are three library routines for manipulating the temporal scope stack: *push_ts* to create a nested temporal scope, *pop_ts* to restore the previous temporal scope, and *change_ts* to change the timing constraints of the current temporal scope. They are called as follows:

- ts_id = change_ts (hs_flag, s_time, e_time, deadline)
- ts_id = push_ts (hs_flag, s_time, e_time, deadline)

- ts_id = pop_ts (hs_flag, s_time, e_time, deadline)

The *hs_flag* indicates whether the current constraint is hard or soft real-time. A process cannot start before the start time, *s_time*, and cannot execute longer than the execution duration, *e_time*. The deadline, *deadline*, specifies when a process must complete the current temporal scope. The system generates a unique id, *ts_id*, for each temporal scope. If the scheduler cannot guarantee the requested hard timing constraint, *change_ts* and *push_ts* return an error code without changing the temporal scope stack.

A timing constraint is violated if either the process executes longer than the maximum execution time or the deadline for the process is exceeded. When this happens, the kernel sends a signal to the process. If the timing constraint is missed in a hard real-time process, then a critical system error has occurred since the constraint was guaranteed by the scheduler. Thus, the process which missed the constraint becomes an *imperative* process so that a controlled shutdown of the system can occur as soon as possible. However, when a soft real-time constraint is violated, the process retains its status as a soft real-time process while the exception is being handled. This is not considered a fatal error so, at programmer control, the system may attempt to continue if desired.

Periodic Temporal Scope Most common timing constraints are periodic in nature. To allow the periodic execution of a function, the notion of temporal scope is extended to include a periodic temporal scope. A periodic scope is defined using the following system call:

- ts_id = set_periodic(function, hs_flag, s_time, p_time, e_time, limit)

The *function* is a pointer to the function that is executed periodically. The *hs_flag* indicates whether the periodic event is hard or soft real-time. For a hard periodic scope, the scheduler guarantees the timing constraints for all periods when the periodic scope is specified. The *s_time* is the start time of the first period and *p_time* is the size of each period. Within each period, the function cannot execute any longer than the expected maximum execution time *e_time*. The function is executed periodically for *limit* periods. Although periodic scopes themselves cannot be nested, other temporal scopes can be pushed from within a periodic scope as long as consistency is maintained.

Timing Constraint Inversion The timing constraint inversion problem can be viewed as a priority inversion problem. The term *priority inversion* refers to a problem which occurs when a high priority process is waiting for a response from a low priority process and a

middle priority process preempts the execution of the lower priority process [2]. Extra delay is incurred by the high priority process since the middle priority process must complete before the low priority process resumes. The timing constraint inversion problem can occur between processes of different priorities and between real-time processes with different deadlines. The extra delay from a middle priority process executing could cause a timing constraint to be missed. We prevent both types of timing constraint inversion from occurring by allowing the propagation of timing constraints for interprocess communication. In the first case, suppose a soft real-time process issues a request to a non real-time server process. The timing constraints associated with the message are propagated to the server process so that it becomes a soft real-time process to handle the message. After a reply is sent, the priority of the server process is reduced to that of a non real-time process. Similarly, timing constraints are propagated from a sender to a receiver within the same real-time priority level.

3.3 Communication

Real-time systems are asynchronous in nature and require predictably fast communication. Often more important than actual speed of communication is predictability [5]. Commonly used synchronization and communication primitives such as signals and messages based on ports have been designed without considering guaranteed response. RK provides three basic communication methods:

- Signals for critical system errors.
- Timed events for asynchronous notification of events with the propagation of timing constraints.
- Ports for asynchronous message passing with timing constraints.

3.3.1 Signals

Signals are used by the kernel to notify a process that an error has occurred. The purpose of sending such a signal is to give the process a chance to clean up its state or to perform a controlled shutdown of the system. There are three types of errors: timing errors (*SIG_TOOLONG* and *SIG_TOOLATE*), process errors (*SIG_PROCESS*) and system errors (*SIG_SYSTEM*). Timing errors are only with respect to the timing constraints of the current temporal scope. The kernel sends *SIG_TOOLONG* if a real-time process executes longer than the maximum execution time, and *SIG_TOOLATE* if a real-time process misses the

deadline. Unlike timing errors, signals for process and system errors can also be sent to non real-time processes. Process errors are errors due to a process itself; for example, an access to an invalid memory address. System errors are errors due to the kernel; for example, running out of buffers that have been guaranteed to a process.

When the kernel sends a signal to a process, the process executes a signal handler and it resumes the previous execution flow when the handler is finished. Since any errors to a hard real-time process can have a disastrous effect on the whole system, a hard real-time process becomes an imperative process when it receives a signal; its priority is lowered when it exits the handler. For soft real-time and non real-time processes, they remain at the same priority level when a signal is received. When a timing, system or processor error occurs and generates a signal, it is no longer necessary for the system to remain predictable, since the process itself determines whether to continue or to shutdown the system.

3.3.2 Events and Timed Events

Events are the most basic way to communicate between processes and are also used by various other components of the kernel such as asynchronous messages, alarms, scheduling and devices. Events provide features similar to the 4.2 BSD UNIX[1] signal package [6]. Like UNIX signals, events can be sent, waited on, delayed and preempted. In addition, timing constraints may be associated with each timed event and the sender can pass an integer value with an event. For each event, the kernel only remembers the last arrival of the event and the last value associated with the event.

In a manner similar to signals, whenever a process receives an event, it executes an associated event handler. The previous execution flow resumes once the handler is finished. In addition, a process can specify which events to wait on and the maximum amount of time to wait. In this respect, events differ from signals since a process never waits for a signal: signals are used to notify that a fatal error has occurred. Mutual exclusion on shared data between an event handler and a process is achieved by delaying any incoming events while executing within the critical section.

Timed events are events with timing constraints. There are two ways to associate timing constraints with events. The receiver of an event may specify a timing constraint for executing the event handler. Here, a temporal scope with this timing constraint is pushed by the system before executing the event handler and popped when it is completed. Alternatively, the sender of an event may include a timing constraint when it sends the event. If both the

[1]UNIX is a trademark of AT&T Bell Laboratories

sender and receiver specify timing constraints for the same event, then the timing constraint with the earliest deadline is used for the execution of the handler.

The scheduling of a timed event handler is based on whether or not a temporal scope for the event can be pushed on top of the current temporal scope. If so, the event handler preempts the current execution flow; otherwise, the event handler is delayed until its temporal scope can be pushed. If more than one timed event is pending, the timed event with the earliest deadline is handled first. Since any temporal scope can be pushed on an empty temporal scope stack, if a non real-time process receives a timed event, it is handled immediately. During the handling of the event, the process becomes a real-time process. This feature allows non real-time server processes to handle requests from real-time processes. It is a process error for real-time processes to receive an untimed event.

3.3.3 Ports

The notion of a port has been used widely for interprocess communication since it provides an easy to use communication abstraction [7]. In RK, we extend it for real-time communication by allowing the sender to pass timing constraints with messages and the receiver to control message queuing and reception strategies.

A process sends a message to a port and receives messages from a port. Each port has a unique system-wide id and has a data structure in the kernel to queue messages. Sending a message to a port is always non-blocking and its execution time is bounded to ensure a predictable delay. For time critical messages, it is important when a message is delivered to a receiver. Thus, the sender can include a timing constraint with each message. The timing attributes are the start time, the maximum execution duration and the deadline. Using these timing constraints, the sender can affect the scheduling of message transmissions and the execution of the receiver process.

Each port also contains information on where to deliver a message when it arrives at the port. If the delivery is to other ports, the message is forwarded to them. However, if the message is to be delivered to a process, the port also contains information on how to queue the message and whether to notify the process using a timed event. Messages are received either explicitly or asynchronously. The time when a message is received using an explicit receive is controlled by the receiver. Here, the receiver can specify a timeout to limit the delay in waiting for a message. For asynchronous receive, the receiver associates a timed event with a port and each message arrival is notified through the timed event.

Port Creation and Attributes. Every RK process is created with a default reception port. This port is used during initialization of distributed processes and to request services from system server processes.

Additional ports are created using the following system call:

- portid = port_create(type)

which returns the unique, system-wide port id. The argument *type* specifies whether the port is for receiving a message or for multicasting a message from the creator's point of view. For a reception port, any process can send a message to it. When a message is sent to or arrives at a multicast port, the message is forwarded to all ports connected to it. This forwarding of a message is repeated until the message reaches a reception port.

For a reception port, there are various attributes that can only be changed by the creator of the port. There are four attributes associated with each reception port. First, the ordering of messages within a queue is either by message sent time, arrival time or deadline. Second, the size of the queue limits the maximum number of messages; if overflow occurs, this attribute also specifies whether messages are thrown away at the head or tail of the queue. Third, messages are removed from the queue when the message is received by a process unless its *stick* attribute is set. Here, the message remains in the queue even after it is received. It is replaced only when a new message arrives [8]. Fourth, messages can be received explicitly or asynchronously. This is described later.

For a multicast port, its attributes are a list of destination ports to which messages are to be forwarded. Unlike reception ports, these attributes are changed when another process requests the insertion or removal of its port from the list of destination ports. The two commands for insertion and removal are as follows:

- port_insert(mportid, portid)
- port_remove(mportid, portid)

where *mportid* is the id of a multicast port and *portid* is the id of a port owned by a requesting process. After a reception port is included in the destination list of a multicast port, all messages sent to the multicast port are forwarded to the reception port. The removal of messages from the reception port is again controlled by the attributes of the port.

Send The only way to send a message is to invoke the non-blocking *send* system call. The syntax of the send call is

- send(portid, reply_portid, timing_record, msg, size)

The *portid* identifies where to send the message and *reply_portid* specifies where to send the reply to the message. The *timing_record* is a record containing three timing attributes which includes the deadline by which processing of the message must be completed. If the deadline is zero, then the message is not time-critical. The *msg* is a pointer to a message and *size* is the length of the message in bytes.

In order to provide a predictable kernel, the execution time required for sending a message must be strictly bounded. In addition, the overhead in performing intermachine communication must not cause the system to become unpredictable. Since RK is running on a network of machines that are isolated from other network traffic, if we limit the number of outgoing messages during any unit period of time, then the number of messages arriving at each machine can be bounded. Section 5 describes this bounding in detail. With this bound, we can compute the worst case hardware interrupt overhead in processing these messages. The scheduler uses this worst case overhead when deciding whether hard real-time constraints can be guaranteed.

Receive There are two ways to receive a message from a reception port. They differ in how the timing constraints are handled and in what message reception paradigm is desired. One way to receive a message is to explicitly invoke the *receive* system call when the receiver needs to receive the message. The syntax of the receive call is

- receive(portid, reply_portid, timeout, timing_record, buf, size)

The *portid* identifies from which port the message is to be removed and *reply_portid* specifies where to send a reply. The *timeout* specifies a non-negative relative amount of time that this primitive should block waiting for a message if none are available. If the value of timeout equals zero, the receiver process does not block waiting for a message. The *timing_record* points to a record containing the three timing constraints that the sender specified with the message. Since it is possible that the message is not received before its deadline, the kernel does not push the timing constraints on the receiver's temporal scope when the message arrives at a port. To use these timing constraints after receiving the message, the receiver process explicitly enters a temporal scope, processes the message and explicitly exits the

temporal scope. The *buf* is a pointer to a message pointer. The *size* is the size of the message which was received.

The other way to receive a message is to receive it asynchronously as it arrives on a reception port. Asynchronous message reception is useful when the main execution flow performs some task and incoming messages need to provide some simple service that can be performed at any time. The notification of message arrival is through a timed event associated with the port. When a message arrives on a port, a timed event is sent to the process which owns that port. The timing constraint specified with the message is used for sending the timed event. This timing constraint is propagated to the receiving process only if it has an earlier deadline than the one associated with the port by the receiver. The timing constraint is pushed on the temporal stack when the timed event handler executes and is popped on return from the event handler.

3.4 Application Devices

The purpose of most real-time systems is to either control or collect data from one or more application devices within timing constraints. Traditional operating systems provide a device driver which buffers requests between application processes and a device. This scheme allows the same device to be used by many processes; however, it introduces additional delay between the time when the device completes a task and the process is notified of its completion. It is difficult for application processes to control devices within timing constraints if traditional device drivers are used due to this additional delay. In distributed sensory systems, sensory devices are not shared among processes as they are controlled by individual processes that collect and preprocess the sensory data. Thus, our kernel allows processes to directly control devices.

To control a device, a process requests the device from the device server. After the request is granted, it is possible to share memory and device registers between the device and the process. In addition, a process may request to the device server that device interrupts be converted to timed events. An alternative approach is to let the interrupt handler collect the data and then send the data to the process in the form of a message [9]. Although our approach requires the programmer to know low-level details about devices, it inherently supports faster feedback control than the alternative approach since no process switching is needed to apply feedback to a device. Furthermore, the kernel need not be changed to reflect the addition or deletion of devices.

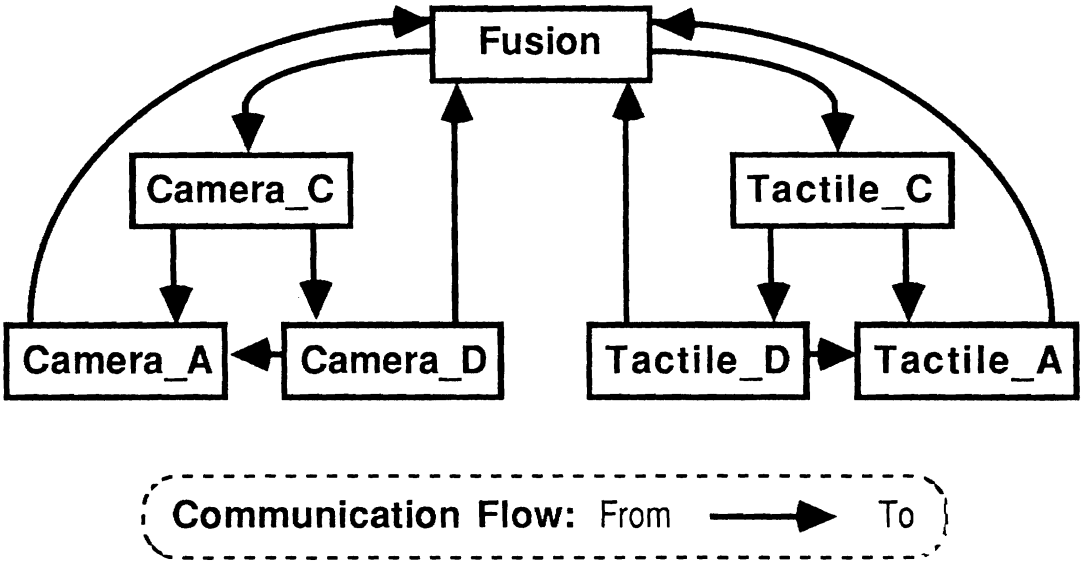

Figure 2: Logical Structure of the Distributed Sensory System

4 Distributed Sensory System

The kernel described in the previous section is being used to implement a distributed sensory system. The goal of this distributed sensory system is to generate a world model describing the locations, sizes and shapes of objects on a table. Initially, this world model contains the location and size of a table. The distributed sensory system uses the camera and the tactile sensors to complete the information contained in the world model. To support distributed sensing, each sensor is attached to a manipulator arm. These manipulators are located on opposite sides of the table. The camera subsystem collects the location of features in the camera plane and the tactile subsystem measures its proximity to a surface. These subsystems collect data at different rates. The fusion subsystem integrates the most recent data available from the two sensor subsystems into the world model. The fusion subsystem repeats this integration step until the world model is completed.

Figure 2 shows the logical organization of the distributed sensory system, which consists of three subsystems: camera, tactile, and fusion. The camera and fusion subsystems are based on the active sensory system in [10] and the tactile subsystem is based on the contour sensor in [11]. Each sensor subsystem consists of three basic process types: sensor arm, sensor data, and sensor control.

The *fusion* subsystem consists of one process that updates the world model based on the data it receives from the various sensors and manipulator arms.

The *camera* subsystem consists of three processes:

- The camera control process, *Camera_C*, generates a new sensor viewpoint using the world model. Once this viewpoint is determined, it sends the observation position to the camera arm process and the window parameters to the camera data process.
- The camera arm process, *Camera_A*, moves the six-joint manipulator (PUMA 560) arm into the desired position. After issuing the move command, the position of the arm is sent to the fusion process.
- The camera data process, *Camera_D*, collects and processes the appropriate image segment from the camera. It then sends the high-level description of the image segment to the camera arm and fusion processes.

Similarly, the *tactile* subsystem consists of three processes: the tactile control process, *Tactile_C*; the tactile arm process, *Tactile_A*; and the tactile data process, *Tactile_D*.

The distributed sensory system must meet various hard timing constraints to be correct. The joint controllers of each arm requires a sampling period of either 14, 28 or 56 milliseconds. At the end of every sampling period, each arm process must receive feedback from the appropriate data process to prevent that arm from becoming unstable. Since the camera data cannot be collected more frequently than every 56 milliseconds, the camera arm and data processes run periodically every 56 milliseconds. However, since the data from the tactile data process can be provided at a faster rate and since the tactile system needs faster feedback in order to apply compliance, the tactile arm and data processes run every 14 milliseconds.

The distributed sensory system is assigned to five MicroVAXes running our kernel. Since each application device is attached to a distinct MicroVAX (as shown in Figure 1), each sensor arm and sensor data process is allocated to the appropriate MicroVAX. That is, the camera arm, the camera data, the tactile data and the tactile arm processes are assigned to the first, second, fourth and fifth MicroVAXes, respectively. Each of these four processes has a hard real-time function which executes periodically every sampling period. Since the limiting factors for the sampling periods is the rate at which the data can be collected from the appropriate sensor, the computation requirements of the data processes are greater than that of the arm processes. Thus, data processes are assigned to their own processors. The camera control process is assigned with the camera arm process to the first MicroVAX and the tactile control process is assigned with the tactile arm process to the fifth MicroVAX. These

Source	Multicast	Data	Deadline	Destination	Port Attributes
Fusion	world_model	world model	14ms	Camera_C	Explicit Receive
Fusion	world_model	world model	14ms	Tactile_C	Explicit Receive
Camera_C		command	56ms	Camera_A	Asynchronous Receive
Camera_C		command	56ms	Camera_D	Asynchronous Receive
Camera_A		position	56ms	Fusion	Explicit Receive
Camera_D	vision_data	vision data	56ms	Camera_A	Explicit Receive
Camera_D	vision_data	vision data	56ms	Fusion	Explicit Receive
Tactile_C		command	14ms	Tactile_A	Asynchronous Receive
Tactile_C		command	14ms	Tactile_D	Asynchronous Receive
Tactile_A		position	14ms	Fusion	Explicit Receive
Tactile_D	tactile_data	tactile data	14ms	Tactile_A	Explicit Receive
Tactile_D	tactile_data	tactile data	14ms	Fusion	Explicit Receive

Table 1: Communication Flow for the Distributed Sensory System

control processes execute in the background and require approximately 20 to 30 seconds of real execution time to compute a new sensor viewpoint. The fusion process integrates the sensory data and the position information at a rate several orders of magnitude slower than the data is collected (i.e., about one second of real execution time for a single integration). The third MicroVAX is dedicated to the fusion process so that no other processes compete for the processor. Thus, the collected information is integrated into the world model as frequently as possible.

4.1 Communications

Table 1 summarizes how communication flows through the distributed sensory system by showing the source and destination of each message. The first column, *source*, specifies the process sending the message. If a multicast port is used for sending a message, its name is located in the second column, *multicast*. The third column, *data*, specifies the type of data that is being transferred. The fourth column, *deadline*, specifies the deadline by when the message should arrive. Since the size of the period is 56 milliseconds for the camera arm and data processes, all messages should arrive before the next period. Similar reasoning follows for the tactile arm and data processes. The fifth column, *destination*, specifies the receiver process. The sixth column, *port attributes*, specifies how the message is to be received.

Communication in the system involves four basic kinds of data. *Sensory data* is transmitted from the sensor data process to the fusion process and the sensor arm process at

the end of every sampling period. *Position information* is transmitted from the sensor arm process to the fusion process at the end of every sampling period. *Commands* are issued by the sensor control process to both the sensor arm and sensor data processes. The *world model* is sent from the fusion process to the sensor control processes.

The destination of messages may either be a multicast port or a reception port. Multicast communication is used by the fusion process to send the world model to the two sensor control processes and by each sensor data process to send sensory data to both its associated arm process and the fusion process. All other types of communication are one-to-one.

The main execution segment of each process in the distributed sensory system executes either iteratively, if non real-time, or periodically, if real-time. Depending upon the type of data involved, messages are either received from their reception ports at the start of each iteration (period) or asynchronously during the execution of their iterative (periodic) segment. The *explicit receive* method is used to receive messages at the start of each iteration by explicitly using a receive system call to receive the message. The *asynchronous receive* method is used to receive messages during a process' execution, by using a timed event to notify it of the arriving message. For messages that are received at the start of each iteration, the message with the most recent arrival time is the one received. The maximum number of messages permitted on the queue is limited to exactly one, always only keeping the latest message. For the messages that are received asynchronously, they are queued since each one must be received and processed individually. Since these asynchronous messages are used for controlling the iterative or periodic algorithm, they must be processed at the earliest possible moment.

The explicit receive method is used for the reception of sensory data, position information and the world model. In the fusion process, new sensory data and position information is received during each iteration of the fusion algorithm, since the time required to execute the fusion algorithm is much longer than the smallest length of time between any two sensory data or position information messages. Similar reasoning can be used for the sensor control and sensor arm processes which also use the explicit receive method.

The asynchronous receive method is used for the reception of commands. In the sensor arm processes, while their periodic component is executing, new commands may arrive from the appropriate sensor control process. The sensor arm process receives the command message when a timed event arrives. Within its event handler, it removes the message and processes the command. Sensor data processes receive command messages in a similar manner.

4.2 Logical Flow

Fusion Subsystem The fusion process is a non real-time process which integrates the sensory data and position information that is received from the sensor subsystems into the world model. Since the arrival of the sensor arm and sensor data messages is not guaranteed to occur at the same time, the position information and sensory data used may not correspond to the same sampling period. However, in the worst case, the time that the message was sent can be off by no more than one sampling period. The statistical model which integrates the data takes this problem into account. Once an updated world model is generated, it is sent via a multicast port to the sensor control processes. The fusion process repeats this integration step until the data provides no new information to the world model.

Camera Subsystem The camera control process uses the current world model to choose a sensor viewpoint and to generate the appropriate sensor control information. Once the control information is generated, the sensor viewpoint is sent to the camera arm process and the sensor control information is sent to the camera data process. Both of these command messages must not be delayed by more than 56 milliseconds (one sampling period).

The camera arm process is a hard real-time process which consists of three execution segments: its main body which initializes the process, a periodic function which moves the robot arm every sampling period, and an asynchronous port handler which receives and processes new commands. The most recent feedback information from the camera data process is received at the start of the period. The command used to move the manipulator to a particular segment is computed from the feedback information and the next destination. This command is sent to the joint controllers via a parallel link and the position information is sent to the fusion process. The deadline for this message equals the sampling period to be consistent with the deadline of the message sent from the camera data process to the fusion process. The asynchronous port handler executes whenever a new destination is sent to this process.

The camera data process is also a hard real-time process which consists of three execution segments: its main body which initializes the process, a periodic function which collects information for the camera every sampling period, and an asynchronous port handler which receives and processes new commands. During each sampling period, the periodic function cuts a window from the frame buffer, analyzes the window according to the methods found in [10], and sends the sensory data to the fusion and camera arm processes via a multicast port. The deadline for this message is the sampling period since this message is used for feed-

back by the camera arm process which must receive new information during each sampling period. The asynchronous port handler executes whenever a new set of window parameters is sent to this process.

Figure 3 shows the skeleton code segments for the camera data process in the C programming language. Three code segments are shown: *camera_data*, *collect_data*, and *recv_command*.

The main body of the process, *camera_data* prepares the image processor for execution. First, the image processor device is allocated using the function *alloc_dev*. Second, the function *map_breg_to_proc* is used to map the device register region into the process and returns the process address of the first device register. Third, the function *map_bmem_to_proc* is used to map the address of the frame buffer on the image processor board into the process. The constant *BUS_ADDRESS* specifies the bus address of the first page and the constant *NO_PAGES* specifies the number of pages to be mapped in. Fourth, the system call *evt_handler* is used to associate the function *recv_command* with the *EVT_COMMAND* timed event. The constant *EVT_COMMAND* specifies the numeric value for the timed event. The *set_periodic* statement is used to create a periodic temporal scope for the *collect_data* function. This periodic scope has hard real-time constraints since it must send new sensory data to the camera arm process every sampling period. Its first period begins immediately, may execute for no longer than some constant *EXEC_DUR* and has a period of 56 milliseconds. Since the robot arm continues to move until a stop command is issued, the limit is some large constant *FOREVER*.

Tactile Subsystem The logical execution flow for the tactile control, tactile arm, and tactile data processes are very similar to their camera counterparts. The primary differences are in the timing constraints and in the port names. Since the sampling period for the tactile manipulator is 14 milliseconds and the sampling period for the camera manipulator is 56 milliseconds, wherever a timing constraint of 56 milliseconds is used in the camera processes, it should be replaced with 14 milliseconds for the tactile processes.

5 Performance

In our real-time system, each processor executes either a process, system call or interrupt handler at any moment of time. The architecture of the processor permits interrupt handlers to preempt the execution of processes or system calls. Thus, to be able to predict the response time of a process, the worst case execution times of system calls and interrupt handlers must

```
camera_data()
{
    dev_id = CSR of image_processor;
    alloc_dev(dev_id);
    dev_regs = map_breg_to_proc(dev_id);
    dev_memory = map_bmem_to_proc(dev_id, BUS_ADDRESS, NO_PAGES);

    evt_handler(EVT_COMMAND, recv_command);
    tc_id = set_periodic(collect_data, HARD_RT,
                         now, 56msec, EXEC_DUR, FOREVER);
}

collect_data()
{
    /* inside frame buffer, cut out a window and perform analysis
       on that window (data_msg) */
    send(vision_data, NULL_REPLY, within 56ms,
         data_msg,    sizeof(data_msg));
}

recv_command()
{
    receive(c_d_command,  ctrl_reply, timing_rec,
            ctrl_msg,     ctrl_size);
    /* using control information (ctrl_msg), modify window
       parameters */
}
```

Figure 3: Logical Execution Flow for the Camera Data Process

Operation	Time (μsec)	
	Minimum	Maximum
Timing measurement overhead	4	11
Basic system call overhead	120	123
Sending an event	176	195
Receiving an event with empty handler	325	362
Waking up on a waited event	79	109
Device interrupt latency	514	525
Process switching overhead	240	240
Intramachine message send	913	957
Message receive	353	734
End-to-end communication delay	4864	5406

Table 2: Timing Measurements of the Kernel

be bounded. Using the worst case execution time of interrupt handlers and interrupt rates, the scheduler computes the amount of processor time available for processes. This processor time is allocated to guarantee hard timing constraints. This section presents the timing measurements of commonly used system calls and provides a simple formula that can be used to compute the overhead of interrupt handlers.

5.1 System Call Execution Times

The current version of our kernel resides on a network of MicroVAX II's, connected through a 10 Mb Ethernet. Table 2 shows the measured execution times of the individual system calls. A microsecond resolution hardware clock was used for the measurements. The times reported are the minimum and maximum values observed over ten thousand individual measurements. All the measurements other than intermachine communication delay were observed on a single MicroVAX with all interrupts disabled. For intermachine communication delay, we used two MicroVAXes connected by their own Ethernet.

The first line contains the overhead of performing a timing measurement; that is, the time elapsed between starting and stopping the clock. All of the other timing measurements include this overhead. The second line is the time it takes to execute an empty system call; that is, the overhead of changing execution modes and checking the number of arguments. This overhead is included in the timing measurements of other system calls.

The next two lines correspond to the times required to send and receive an event. Here, the receive overhead includes the execution time of an empty event handler. The subsequent line shows the times for unblocking a process that is waiting for an event.

The sixth and seventh lines indicate the device interrupt latency and the process switching overhead. The device interrupt latency is the time between when a device requests an interrupt and when a process starts executing the corresponding event handler. This delay is the speed with which a process can start executing an event handler after an application device requests an interrupt. The process switching overhead is the estimated overhead involved in performing a process switch.

The eight and ninth lines show the times required to send and receive a message between two processes on the same machine. Here, the receive succeeds immediately as there are always messages pending. The last line is the end-to-end communication delay of 1K byte message between two application processes running on two machines. This delay includes the following components: intramachine send (913 μs), network interrupts (transmit 1269 μs and receive 1289 μs), transmission delays (780 μs), unblocking of a receiver (79 μs), process switching (240 μs) and intramachine receive (353 μs). The sum of these components (4923 μs) confirms our end-to-end delay measurements since it is within the observed range. The 59 μs error could be attributed to a possible error associated with estimating the process switching overhead.

5.2 Guaranteed Processor Time

To determine the processor time that can be guaranteed for the execution of processes, the worst case overhead of interrupts must be bounded. The kernel contains two interrupt driven subsystems, time maintenance and network communication. The network communication subsystem uses a network adapter to transmit and receive messages over the Ethernet. The network adapter generates an interrupt whenever a message is received or transmitted. The time maintenance subsystem uses an interval timer to update the time of day clock and to set off alarms. A queue of alarms is maintained by the kernel for use by the scheduler and individual processes.

There are two interrupts associated with time maintenance. The interval timer generates a periodic interrupt every 10 milliseconds. It is used for recording the system time; the execution time of its handler is 135 μs. Within the interval timer interrupt, if an alarm has expired, then an alarm interrupt is requested. The alarm interrupt handler determines the process that owns the alarm, so that it can be notified through a timed event. Its

execution time is 250 μs per alarm. Since the interval timer interrupt handler blocks all other interrupts, this two level approach allows other high priority interrupts to only be delayed for a short time.

Network communication consists of two interrupts. A receive interrupt occurs when the network adapter receives a packet from the network. The receive interrupt handler removes the packet and forwards it to the appropriate process. This handler takes 1289 μs to execute. The transmit interrupt consists of two parts: one part to remove the last packet which was transmitted, and the other part to create a packet and pass it to the network adapter. The worst case execution time of this handler occurs when both parts execute (1269 μs).

We can compute the worst case interrupt overhead for each 10 millisecond clock period if the execution times and rates for each interrupt handler is known. To generate the worst case interrupt overhead, the equation below uses a simplified model where the maximum execution rates for each interrupt is known for each 10 millisecond clock period:

$$\sigma = 0.135 + 0.250A + 1.289R + 1.269S \quad (1)$$

where, A is the maximum number of alarms that can go off, R is the maximum number of messages that can be received, and S is the maximum number of messages that can be sent. Using this equation, at least $10 - \sigma$ milliseconds of processor time can be guaranteed for process execution.

Equation 1 can be used to compute the amount of time guaranteed for process execution of each of the MicroVAXes used in our distributed sensory system. For the first and fifth MicroVAXes, 4.249 milliseconds out of every 10 milliseconds can be guaranteed for the execution of the sensor arm and control processes since $A = 2$, $R = 2$ and $S = 2$. For the second and fourth MicroVAXes, 5.518 milliseconds out of every 10 milliseconds can be guaranteed for the execution of the sensor data processes since $A = 2$, $R = 2$ and $S = 1$. However, for the third MicroVAX, no time can be guaranteed for the execution of the fusion process since $A = 0$, $R = 4$ and $S = 4$.

Even though the fusion process is not a real-time process, this shows the limitations of the simple model. Our method of estimating interrupt overhead bounding is very pessimistic in that we assume that the interrupt rates are always at the highest rate for each clock period. However, as one can see with the timing constraints, the rates reach their upper bounds only on certain periods (i.e., for periodic camera data processes, every 56 milliseconds). We are in the process of formalizing a model that analyzes the interrupt overhead in greater detail. In addition to those parameters dealt with in the current model, the interrupt will

be prioritized as they are in hardware so that the interaction among the interrupts can be effectively modeled [12].

6 Conclusions

We have described a distributed real-time kernel which supports distributed real-time applications. Our notion of real-time is not on its speed of execution but on its predictability with respect to time. To ensure predictability, system calls are designed to exhibit bounded execution times and processes are scheduled based on their timing constraints. In addition, the scheduler guarantees hard timing constraints by computing the worst case overhead of device interrupts. This overhead bound is very pessimistic; we are currently working on a less conservative method to bound the worst case overhead of device interrupts.

We plan to make hardware enhancements for additional versatility in two areas: time maintenance and network communication. The built-in interval timer for the MicroVAX has a period of ten milliseconds which is not fine enough for robotics applications. An accuracy of at least one millisecond would be more suitable. We are adding a timer/counter board (Codar Technology) which contains fifteen individually controllable clocks with a finer resolution, and can be programmed to any range larger than one microsecond. Furthermore, each clock can generate interrupts at arbitrary fixed intervals without the kernel overhead of updating the clock between intervals.

We are also adding a token ring (Proteon) to our distributed system. As we saw from the example distributed sensory system, real-time communications are usually periodic and require bounded communication delays. Thus, we believe that a token ring with bounded transmission delay is more suitable than an Ethernet with nondeterministic transmission delay. Once we add a token ring, we will be able to compare the real-time suitability of the two communication media.

In this paper, we have described a real-time kernel that is predictable in the worst case. Its initial implementation has been completed, and the results shown in Table 2 are promising. We intend to refine the kernel by adding the hardware enhancements mentioned above. Once this work is finished, the RK kernel will provide support for many time-dependent, robotics applications.

Acknowledgements

The authors wish to thank Gregory Hager for his assistance in describing the example used in his dissertation and for his valuable suggestions on this paper. We also wish to thank Gaylord Holder for contributing to the initial design and implementation of various components of an earlier version of the kernel.

References

[1] I. Lee and V. Gehlot, "Language constructs for distributed real-time programming," in *Proceedings of the Real-Time Systems Symposium*, IEEE Computer Society, Dec. 1985.

[2] L. Sha, J. Lehoczky, and R. Rajkumar, "Solutions for some practical problems in prioritized preemptive scheduling," in *Proceedings of the Real-Time Systems Symposium*, pp. 181–191, IEEE, 1986.

[3] A. Damm, "Kernel aspects of the distributed real time operating system of MARS," Tech. Rep. Mars Research Report Nr. 6/87, Institut fur Technische Informatik, Technical University of Vienna, Feb. 1987.

[4] A. K. Mok, *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environments.* PhD thesis, Massachusetts Institute of Technology, 1983.

[5] J. Stankovic and K. Ramamritham, "The design of the Spring kernel," in *IEEE Fourth Workshop on Real-Time Operating Systems*, pp. 19–23, 1987.

[6] W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, *4.2BSD System Manual.* Computer Systems Research Group, Computer Science Division, EECS, University of California, Berkeley, July 1983.

[7] K. G. Shin and M. E. Epstein, "Intertask communications in an integrated multirobot system," *IEEE Journal of Robotics and Automation*, vol. RA-3, pp. 90–100, Apr. 1987.

[8] K. Schwan, T. Bihari, B. W. Weide, and G. Taulbee, "High-performance operating system primitives for robotics and real-time control systems," *ACM Transactions on Computer Systems*, vol. 5, pp. 189–231, Aug. 1987.

[9] M. F. Coulas, G. H. Macewen, and G. Marquis, "RNet: a hard real-time distributed programming system," *IEEE Transactions on Computers*, vol. C-36, pp. 917–932, Aug. 1987.

[10] G. D. Hager, *Active Reduction of Uncertainty in Multi-Sensor Systems.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1988.

[11] I. Lee and R. King, "Timix: a distributed real-time kernel for multi-sensor robots," in *International Conference on Robotics and Automation*, pp. 1587–1589, IEEE, 1988.

[12] M. Joseph, "On a problem in real-time computing," *Information Processing Letters*, vol. 20, pp. 173–177, May 1985.