Design, Implementation, and Evaluation Of A Distributed Real-Time Kernal For Distributed Robotics

(Dissertation Proposal)

MS-CIS-90-40
GRASP LAB 220

Robert King

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

July 1990

Design, Implementation, and Evaluation of a Distributed Real-Time Kernel for Distributed Robotics

Robert B. King

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania

Dissertation Proposal
Supervised by Dr. Insup Lee

July 5, 1990

Abstract

Modern robotics applications are becoming more complex due to greater numbers of sensors and actuators. The control of such systems may require multiple processors to meet the computational demands and to support the physical topology of the sensors and actuators. A distributed real-time system is needed to perform the required communication and processing while meeting application-specified timing constraints.

We are designing and implementing a real-time kernel for distributed robotics applications. The kernel's salient features are consistent, user-definable scheduling, explicit dynamic timing constraints, and a two-tiered interrupt approach. The kernel will be evaluated by implementing a two-arm robot control example. Its goal is to locate and manipulate cylindrical objects with spillable contents. Using the application and the kernel, we will investigate the effects of time granularity, network type and protocol, and the handling of external events using interrupts versus polling. Our research will enhance understanding of real-time kernels for distributed robotics control.

Contents

1 Introduction

Real-time applications differ from other computer applications by requiring logical correctness *and* satisfaction of certain timing constraints. Satisfying timing constraints is required to avoid possible catastrophic results from the underlying physical processes being controlled [Lee84]. The term *hard real-time* is used when these timing constraints must be met in all cases. Complex real-time applications tend to be naturally distributed. A *distributed real-time application* consists of a collection of real-time processes cooperating to perform some computation. These processes communicate with one another and with the external environment. Examples of distributed real-time applications include flight control systems [WLG*78], space shuttle avionics [Car84], and robotics.

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 [LG85].

1.1 Distributed Real-Time Systems

A *distributed system* consists of multiple nodes connected by one or more networks capable of handling communication between the nodes. A *node* consists of one or more processors that share a single backplane which includes memory, devices, and

network adapters. A *network* consists of channels to allow one node to communicate with one or more nodes at any given instant in time. A distributed system utilizes *dynamic scheduling requirements* if the scheduling requirements directly depends upon external stimuli and the external stimuli depends upon a combination of the environment and other components of the system. Since these systems may receive sporadic information from the environment and the rate of information may change as the system runs, the resulting scheduling parameters are not fully known in advance of the system's execution. The term *static scheduling requirements* refers to scheduling requirements that do not change as the system runs.

The most important quality that a distributed hard real-time system should possess is *predictability.* A real-time system is predictable if its functional and timing behavior is as deterministic as is necessary to satisfy system specifications [Sta88]. The system must respond to external events within bounded time intervals to avoid potential catastrophe. Predictability should be foremost in the minds of those who design, debug and maintain real-time programs [LKP89].

A *distributed real-time kernel* consists of a single kernel that is replicated on each node throughout a distributed real-time system. Most distributed real-time kernels are not designed as multipurpose kernels, but to serve the specific needs of real-time processes [CMM87]. A *message-based distributed real-time kernel* consists of two essential components: a scheduler which guarantees process completion within specified timing constraints and a message transmission service which delivers messages within specified timing constraints. Meeting of dynamic scheduling requirements is crucial in these systems.

1.2 Robot Control Systems

To motivate our discussion of distributed hard real-time systems, consider the class of applications known as robot control systems which are used to control robot manipulators in conjunction with other sensors and actuators. Robot control systems can be decomposed into several hierarchical levels [Cla89,LGCS89,MWB89,Sal89,SARTICS89] [SHKK89]. Servo loops form the lowest levels of most robot control systems. A servo

loop consists of a periodically scheduled process which takes as its input the error of some part of the system and produces corrections which when applied to the system's actuators will reduce the error. Functioning at higher levels, supervisory control involves the coordination of multiple sensors, actuators, and subsystems to form coupled systems. Such control systems must communicate with other computers to perform higher-level tasks such as motion planning or factory-level coordination. In addition, they provide support functions such as process monitoring and error recovery.

1.2.1 Application

Consider a two manipulator robot control system which is used to locate circular objects with spillable contents, push them around, pick them up, and pour their contents out. Attached to the end of one manipulator arm is first a six-degree-of-freedom compliant wrist ([XP88]) and then a wedge-shaped end effector. The other arm has a wedge-shaped end effector attached to it. These wedge-shaped end effectors have two three inch square faces connected at a right angle with simple contact sensors mounted on the interior side of each face. Most arm movements occur as Cartesian motion (straight-line motion) rather than joint motion. Objects are picked up by applying forces to the opposite sides of the objects by the two arms (the friction between the object and the wedge allows the object to be lifted). Once an object is lifted, it is moved over a bucket and its contents are emptied out. Then the object is moved over a box and dropped into the box. Figure 1 illustrates the application from above – the large circle indicate the reach of each robot arm and the objects themselves are represented by grey circles.

There are three fundamental modes of operation in this two-arm pushing and lifting application. We enumerate them in the order of their occurrence:

Mode 1. The initial configuration has each end-effector located a constant distance above a flat surface which contains the objects. Each arm starts an *independent search* and uses the contact sensors on the end-effectors to detect contact with any objects. The goal is to find an object which is later lifted, emptied, and dropped. As each arm moves, it sends location information to a collision avoid-

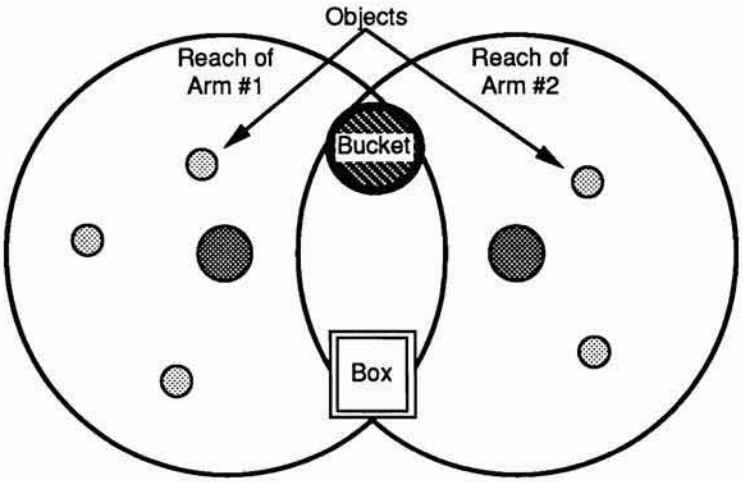

Figure 1: Robot Application (top view).

ance module which computes the distance between the arms. If the distance between the arms is decreasing at a rate which exceeds a predefined system constant, then warning messages are sent to both arms. In addition, if collision is imminent, the arm which initiated the movement in this direction last is instructed to change its path. Since the two arms are operating in an independent fashion, the communication requirements are minimal except when collisions are feasible between to two arms.

Mode 2. Once an object has been found, the two arms operate in an *uncoupled, cooperative* manner. Our goal is to push the object into the common area between the two robot arms and move the end-effectors until each one is contacting opposite sides of the object. Let arm A be the one which found the object and arm B be the other. Arm A carefully pushes the object so as to prevent spilling of the contents of the object. Arm B moves to "meet" the object from the other side of the common area. The communication requirements are moderate since the update messages are sent more frequently as the distance between the arms decreases.

Mode 3. The object is now manipulated by the two arms in a *coupled, cooperative* manner. Our goal is to lift the object, move and empty its contents into a bucket, and move and dispose of the object into a box. The arm with the

compliant wrist is called the *slave* arm and the other arm is called the *master* arm. The master arm initiates the moves and the slave follows by sensing the forces acting on the wrist. The communication requirements are extensive since the slave arm can not lag the master by more than a few sampling periods (the exact number will be determined in our experimentation). Once the object is dropped, the arms move the end-effectors back to the location from which they lifted the object. At this point, we return to Mode 1 where the search proceeds for another object.

This distributed real-time application can be subdivided into three processing components: one for each manipulator arm, and one for the collision avoidance and cooperative components. The robot arm with the wrist and wedge end-effector consists of six basic tasks. First, the *hand* task only executes sporadically in modes 1 and 2 when contact occurs and then the task updates the contact information. Second, the *wrist* task only executes periodically in mode 3 where it updates the force information. Third, the *arm* task executes periodically in all modes to compute the next joint angle for each joint on the manipulator arm. Once the computation has been completed, the results are passed to the joint control subsystem. Fourth, the *search* strategy task executes sporadically only when the arm requests a new goal path. There are two communication tasks: *network in* which is used to receive messages and *network out* which is used to transmit messages. These tasks only execute whenever other components communicate with this control system, or vice versa.

The system architecture contains dedicated processors at the lowest levels for joint actuation which communicate to the higher levels via dedicated hardware links. The higher level consists a distributed real-time system forming the supervisory control system. The distributed real-time kernel discussed in this proposal is used to handle the supervisory control aspects of such a system and to handle the sensor processing. Section 5 examines this application in further detail.

1.2.2 Problems

We now identify four problems that must be solved when implementing a real-time application such as this one.

1. *How does the system guarantee timely response to sporadic entities?* This problem refers to the tradeoffs between polled and interrupt-driven input and their respective costs. Although our example does not contain a specific case where this problem is crucial, we can extend the application to illustrate our point. As described, our application assumes that the objects are located in a static environment – that is, the objects will not be moved unless pushed by a wedge. However, if the objects are in a dynamic environment where an object may be given to an arm, then this problem is crucial. Consider the *hand* task which processes the data from the contact sensors located in the wedge and only executes when contact has been detected. Interrupt-driven input is required for this task since the system does not know when the sensor may detect contact. Polling can only be used if the system knows how frequently or when polling should be performed in advance. For the contact sensor, the manipulator arm moves at a specific rate and this rate can be used to compute the polling frequency. However, since the environment in which the robot arm moves is dynamic, one does not know in advance how the environment will change. Thus, if polling were used, the contact might not prevent the pushing of objects before the correct direction has been determined.

2. *How does the system guarantee that the overall scheduling of tasks is maintained when external events occur?* The overhead in processing interrupts should not preempt processing of tasks which are more critical. The traditional approach is to disable all interrupts or to disable interrupts from selected devices. In the first case, critical interrupts cannot execute their handlers since they too are disabled. In the second case, knowledge of the individual devices would be required. Consider the *network in* task where message processing and ordering depends upon the contents of the message. Ideally, we would like the task to only run if the message received is more critical than any other task within the system.

3. *Scheduling requirements may change as the system executes or may depend on the system state. How does the system deal with this?* Dynamic scheduling requirements include timing constraints, priority, and any other parameter appropriate to the application. Our robotic application illustrates this problem in two ways: tasks only run in certain modes and the communication tasks (such as network in and network out) have scheduling requirements which change depending upon the mode.

4. *How easy is it to convert application requirements into desired scheduling characteristics?* This problem is only relevant when one considers that real-time systems must be used in order to be considered useful. If scheduling requirements (timing requirements and message ordering) can not be scheduled with the scheduling algorithm supported by the real-time system, then the usefulness of the system is lost.

1.3 Research Motivation

Our research is motivated by the difficulties encountered when we attempted to implement time dependent distributed programs such as a distributed robotics 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.

Even with currently available real-time operating systems, it is difficult to develop distributed robotics programs. First, they typically provide schedulers based on priority rather than timing constraints. Programmers must therefore ensure that timing constraints are met by making the proper process priority assignments. This can prove to be quite a difficult task in a complex system, and can often result in very low processor utilization [SLR86]. Further, most off-the-shelf operating systems provide a very limited set of primitives to manage time, making it cumbersome to implement programs whose correctness depends on exact timing. The typical kernel interface

includes primitives used for setting and clearing alarms, or for making a process sleep (or wait). But since these functions do not explicitly affect process scheduling, it is not easy to use them for predictable time management in real-time programs. Current operating systems also lack services necessary for time-bounded communication. For example, the scheduling of message transmission is not deadline-driven; instead it generally adheres to first-come-first-served or priority-based paradigms. Furthermore, the end-to-end communication delay of a message cannot be bounded since the scheduling of processes and message delivery are not integrated.

To deal with these problems, we believe that a distributed real-time kernel must adhere to the notion of *consistent scheduling*. Under the consistent scheduling paradigm, all threads (the basic unit of execution) are scheduled for execution in a uniform manner with a user-definable scheduler. The scheduling parameters specified by each thread may be modified as the system executes. Therefore, if the scheduling requirements change as the system executes or as the system state changes, then the scheduling parameters can be dynamically modified. Furthermore, device interrupts are isolated from the system by employing the notion of an *event* and a thread. Whenever a interrupt occurs, the appropriate event is triggered. The interrupt handler is written as a thread and suspends itself until the associated event becomes valid. At this point, the scheduling of the interrupt thread occurs via the same scheduler as the other threads. Finally, port-based asynchronous message passing should be enhanced to allowing individual message priorities which are used to order messages in the queue at each port.

To facilitate the development of robotics applications that require real-time capabilities, we propose the design and implementation of a distributed real-time kernel (Timix (v. 2)) to support the timely execution of time critical threads and communications. A user-definable scheduling algorithm is used to determine which thread to execute next. A two-tiered interrupt scheme is provided using event and lightweight threads which allow all device interrupts to be mapped into threads scheduling using the standard scheduler. The kernel 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 threads and message

transmission and reception. 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.

1.4 Proposal Outline

There are five sections to follow. Section 2 classifies existing real-time kernels according to their scheduling approach and their technique for responding to external events. Section 3 examines the goals of our proposed research and our approach for undertaking the research. The kernel design is described in Section 4. Section 5 examines the two-arm pushing and lifting robotics application in further detail. Finally, we conclude with a summary of the current status, a description of future work, and a timetable for this work in Section 6.

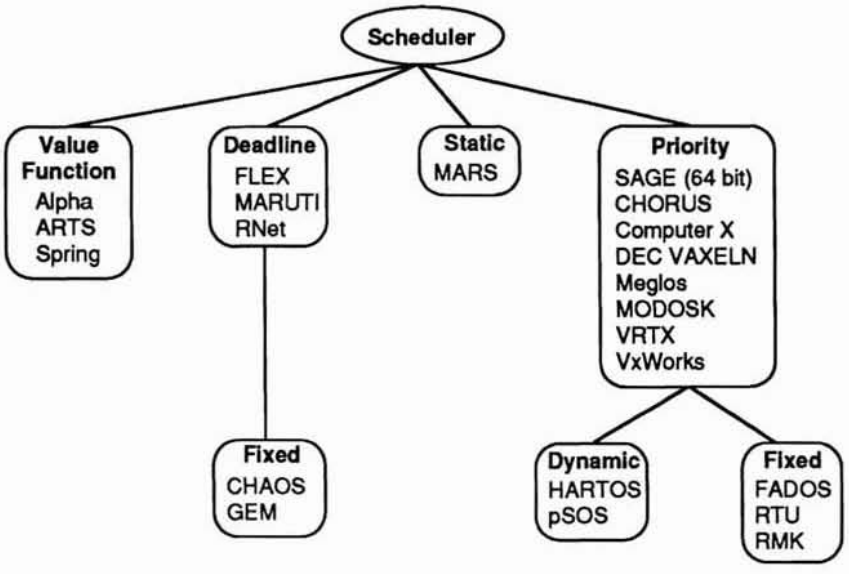

Figure 2: Taxonomy of Techniques that Real-Time Kernel Schedulers Use

2 Related Work

Using the problems that were identified in the previous section, we can classify existing real-time kernels in two separate taxonomies: scheduling approach and technique for responding to external events. Next, we summarize our work on Timix (v. 1) and its use in robotic applications. The problems found with Timix (v. 1) provides the motivation for this work.

2.1 Scheduling

Many real-time kernels impose their scheduling policies on the application programmer. For instance, the kernel may use first-come-first-serve, priority-based, or rate monotonic scheduling policies. Since the specification of an application does not usually reflect these notions, the programmer must convert application specified scheduling requirements into notation which the kernel's scheduler may use. Furthermore, the programmer may not be able to tell whether an application meets its scheduling requirements without simulating all possible execution choices.

Scheduling algorithms for real-time kernels can be divided into four basic categories: by static, by priority, by deadline, and by value function. Figure 2 classifies existing systems in a taxonomy based on their scheduling approach. We discuss these systems in the order of least flexible to most flexible.

Static The scheduling information utilized by a *static scheduling* approach is determined before the system executes. The primary advantage of such an approach is predictability – since the scheduling is determined in advance, one will know if the system is as deterministic as is necessary to meet the required timing requirements. However, there are several disadvantages. First, it is difficult to predetermine how everything within the system should be scheduled. Second, once a schedule is obtained, it is difficult to modify a program without having to recompute the entire schedule. Baker [BS88] defines a *cyclic executive* as a control structure for explicitly interleaving the execution of several periodic processes on a single CPU. **MA**intainable **R**eal-time **S**ystem (MARS) [DRSK89,KDK*89] utilizes an extension of the cyclic executive approach as its scheduling approach. Hard real-time tasks are cyclic tasks that receive, process, and send messages. However, with this system, they can be no hard real-time tasks which are not periodic. Acyclic tasks are provided in the form of soft real-time tasks and utilize the idle time of the CPU in low load situations. MARS' off-line scheduling approach considers the maximum execution times of hard real-time tasks, their cooperation by message exchange, and the assignment of messages to slots on the communication network. To provide some flexibility, changes between two schedules can occur only at predefined points in time and the maximum delay between an immediate switch is eight milliseconds.

Priority Most current commercial real-time operating systems utilize a priority-based scheduling approach. A preemptive, priority-based scheduling algorithm allows higher priority tasks to run as soon as they are ready by preempting lower priority tasks. This algorithm may use either *static* or *dynamic* priorities. The static priority of a task is usually set when a task is created. Examples of systems utilizing preemptive, priority-based algorithm on tasks with static priorities include: CHORUS (version 4) [RAA*88], Computer X [KB87a,KB87b], DEC VAXELN [Ste88],

Meglos [Gag86,GK86], MASSCOMP's RTU [Hen88], Real-time Multiprocessing Kernel (RMK) [Stu88], Versatile Real-Time Executive (VRTX) [Rea86], and VxWorks [VxW,Mah88a,Mah88b]. The priority of a process in FADOS [TH86] contains two parts: the *hardware* priority determines the priority level of the processor and the *software* priority. The hardware priority is only relevant for processes which handle interrupts. With dynamic priorities, the priority of a task may change as the task executes. Modular Distributed Operating System Kernel (MODOSK) [GLR82] allows processes to change their priority level dynamically.

There are two other systems which use priorities in a more generic way. The SAGE [Sal88,Sal89] system allows process priority to be specified as 64 bit integer values and compares them according to macros specified at system compile time. As a result, the system supports any scheduling policy that allows priorities to be specified in 64 bit integers. In pSOS [KKS89] (and HARTOS [KKS89] which uses pSOS), processes may dynamically change their mode (preemptable or non-preemptable) and their priority as they execute.

Deadline Since many real-time processes have deadlines by which their execution must complete, a better way to schedule processes is to employ these deadlines directly. The earliest deadline first algorithm (EDF) runs at every instant the ready process with the earliest deadline. On a single processor, EDF has been proven to be optimal when the processes are independent [Mok84]. Processes are said to be *independent* if their interprocess communication primitives do not impose any scheduling restrictions. RNet [CMM87] is an example of a system using the EDF algorithm for scheduling processes.

MARUTI, GEM, and CHAOS extend the traditional notion of deadline-based scheduling in three different ways.

- MARUTI [LTCA89] imposes both start-time and finish-time constraints on all computations, whether periodic or aperiodic. The scheduler supports on-line and off-line scheduling disciplines: on-line is for guaranteed jobs and off-line is for both non-deterministic execution time bounds and non-real-time jobs.

- Generalized Executive for real-time Multiprocessor (GEM) [SBWT87] associates the following scheduling information with each process: *deadline* specifies the time interval (relative to the time it was made ready) which the process should complete the execution of all of its microprocesses that are ready to run; *period* specifies the process' period of execution and is undefined if the process is sporadic; and *priority* specifies the scheduling priority of the process. The scheduler uses this information to schedule processes first by priority and then, within each priority level, by the shortest deadline first (EDF) algorithm. This is a non-preemptive scheduler since processes run until completion.

- Concurrent Hierarchical Adaptable Object System (CHAOS) [SGB87,GS89] can adapt its behavior to deliver the specified performance. This performance requirement is specified in terms of an overall deadline for the application. Invocation deadlines are recomputed dynamically so that they remain consistent with this overall deadline. When there is no longer enough slack time available to recompute the invocation deadlines, the application is dynamically modified by moving it along a continuum of different programmer specified versions, each with different functionality and performance.

Value Function A value function provides the most flexible way to schedule processes for execution. Associated with each process is a set of parameters. To determine which process executes next, the value function is executed for each process with its set of parameters, and the process with the "most value" is chosen to execute next. There are some drawbacks to this approach: (1) the computational overhead required to execute the value function and (2) the method required to deal with time dependent value functions. The value returned by a time dependent value function depends on the current time. For instance, a value function for the minimum laxity first paradigm would return the highest value for the process with the least laxity. Laxity is computed by subtracting the remaining execution time from the deadline. The laxity associated with each process is static except for the process that is executing. Laxity represents the maximum delay before which a process can not complete and meet its deadline if it consumes its maximum execution time. Alpha [Jen89] allows the expression of application timing constraints in terms of value to the system in com-

pleting each activity, a function of each activity's completion time. Every evaluation is performed collectively for all executing and pending activities.

Two other systems use the value function algorithm in conjunction with other scheduling algorithms.

- ARTS [TM89] uses the Integrated Time-Driven Scheduling (ITDS) model which provides a notion of "capacity preservation" to deal with hard and soft real-time activities. ITDS determines schedulability of *hard* periodic tasks, uses value functions for *soft* real-time task scheduling, and provides overload control based on the value functions of aperiodic tasks. Simple, non-communicating and independent periodic tasks are scheduled using *rate monotonic* scheduling. The basic rate monotonic scheduling method has been extended to deal with dependent tasks, the synchronizing of tasks in critical sections, and transient overload. A deferrable server for soft real-time tasks selects most important aperiodic task among runnable aperiodic tasks using a value function.

- In Spring [SR87a,SR89], the system processors offload the scheduling algorithm and other operating system overhead from the application tasks both for speed, and so that this overhead does not cause uncertainty in executing already guaranteed tasks. All critical tasks are guaranteed a priori and resources are reserved for them. Essential tasks have deadlines and are guaranteed on-line. Although scheduling is subdivided into four modules, we only examine the local scheduler which is used to guarantee that a new task can make its deadline. This scheduler uses the heuristic function,

  $$H(T) = T_D + W * T_{est}$$

  where T_D is the task's deadline, T_{est} is the task's scheduled start time, and W is a weight.

2.2 External Events

The development of software used by an operating system (or an application) to interact with devices has always involved a classic question: polled input/output or

interrupt-driven input/output. The solution to this classic question must consider the reasons for which interrupts may occur. Output interrupts usually indicate when a device has finished handling some output and is ready for some more. This kind of interrupt can usually be predicted since such an output interrupt does not occur unless output to the device has already been initiated. Input interrupts usually indicate that a device has some new data ready to be used by the application and they present the greatest problem.

There are tradeoffs on using polling to initiate input from a device and on using an interrupt to signify that a device has input. The computer hardware used by most kernels let interrupt handlers execute immediately after an interrupt occurs unless the processor is executing a handler for a higher priority interrupt. This may not be in the best interest of the application as its execution may be more important than the execution of the interrupt handler. For example, a network communication interrupt may not be important while executing code to prevent the blind robot from colliding with an object. A traditional solution to this problem involves the use of polled input. If network communication was polled instead of interrupt-driven, then messages would only be received when the system requests them. However, a polled interaction wastes[1] execution time as it polls devices which are not ready for input. In addition, if a single device's sole purpose is to notify the system that some fault has occurred, then the determination of the polling frequency must tradeoff wasted processor execution time with the delay in determining that a fault has occurred. The same tradeoffs exist for output, but they are less critical since the occurrence of an output interrupt can usually be predicted.

Methods for dealing with external events within real-time kernels can be divided into three basic categories: assigning a dedicated processor to handle the interrupts, polling the device, and interrupting an application processor. Figure 3 classifies existing systems in a taxonomy based on their approach in dealing with external events.

[1]Is wasting processor execution time necessarily a bad thing? From an accounting perspective, i.e., justification for such a machine, we wish to obtain the highest utilization of the system possible and still meet the timing requirements. However, from an engineering perspective, we are more concerned that it be predictable than it necessarily obtain the highest utilization possible [Smi90].

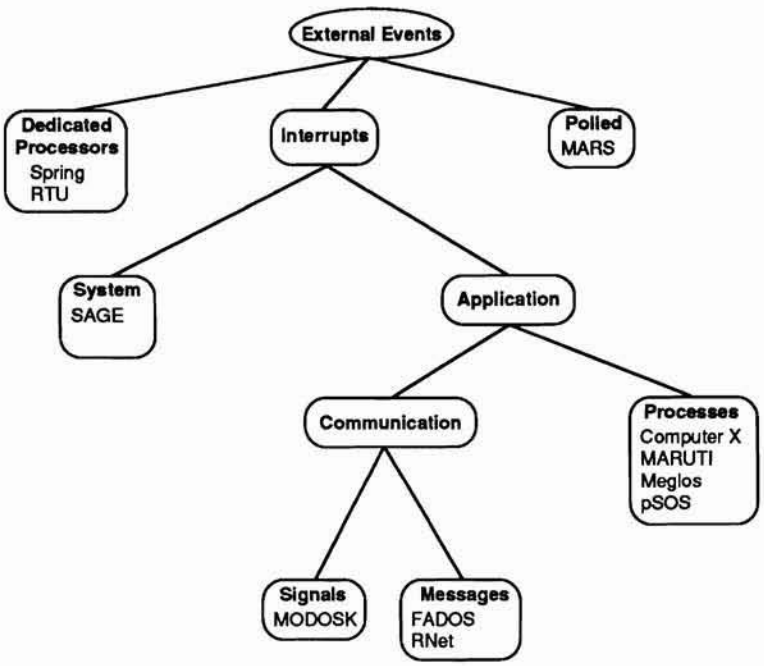

Figure 3: Taxonomy of Techniques that Real-Time Kernel Use for External Events

Dedicated Processors With MASSCOMP's RTU [Hen88], all interrupts are serviced on the boot processor to free the other CPUs from associated latencies and uncertainties. At system creation the system manager can make a block of external memory available for direct access by user processes to provide rapid access to devices without having to write new device drivers. However, if interrupts are to be serviced, then a device driver still is required. Depending upon the system configuration, the interrupt latency is bounded in the five to ten millisecond range.

The Spring [SR87a,SR89] kernel executes on a distributed network of multiprocessors where each multiprocessor contains at least one application processor, one or more system processors, and an input/output subsystem. Input/output devices are subdivided into two types, according to their speeds. Slow devices are multiplexed through a front-end I/O subsystem, are preallocated, and are not part of the dynamic on-line guarantee. Interrupts from the I/O subsystem are handled by system processors. Fast devices are handled with a dedicated processor or have dedicated cycles on a given processor or bus. Resources are preallocated for fast devices.

Polled In **MA**intainable **R**eal-time **S**ystem (MARS) [DRSK89,KDK*89], there is only one interrupt, for the periodic clock interrupt. Since the remaining devices are polled within the clock interrupt, strictly periodic and deterministic system behavior is maintained. The clock interrupt handler is divided into two levels. Every millisecond, the handler interrupts any executing task or system call and polls the other devices. Every eight milliseconds, the handler only interrupts tasks but follows any system call that may be executing, updates the system data structures upon sending or receiving data from the other level, and allows state changes to occur.

System Interrupts Interrupt-based real-time kernels can be subdivided into two basic categories: kernel responds to interrupts and application responds to interrupts. The notion *kernel responds to interrupts* defines those operating systems which respond directly to the interrupt and do not permit the application to directly interact with devices. Experience with existing systems, such as UNIX-based ones, have shown that interrupt handler latencies can be huge, particularly for those handlers that handle requests for more than one process at a time (i.e., disk or network requests). For instance, consider the interrupt handlers provided by SAGE [Sal88,Sal89]: each handler runs on the current process's kernel stack, can access any valid kernel address, and can only invoke a limited number of operations. Because it shares the process's stack, an interrupt handler cannot block, and it can only be preempted by a higher priority interrupt handler.

The notion *application responds to interrupts* defines those operating systems that permit applications processes to directly response to device interrupts, either through message-based communication or by viewing the interrupt handler as a process. The message-based communication approach is used by RNet [CMM87] and Computer X [KB87a,KB87b]. The viewing of the interrupt handler as a process is used by pSOS [KKS89], Versatile Real-Time Executive (VRTX) [Rea86], and MARUTI [LTCA89]. In pSOS, the interrupt handlers can only use a restricted set of primitives to interact with other processes. VRTX provides more flexibility by placing no constraints on the operation of its interrupt handlers. Any object in MARUTI that requires an interrupt must reserve the interrupt service as an enabling condition for its execution. Once the reservation has occurred, the object state is set to *IDLE*. When the interrupt

occurs, the interrupt service object is set to *ACTIVE* and it returns to where it was executing.

2.3 Robot Control with Timix/RCI

Timix [Kin87] (extended in [LK88,LKY88,Kin89]) was an attempt to provide some of the features missing in existing real-time systems. Timix supports processes with independent address spaces that execute, communicate and handle devices within timing constraints. Signals and asynchronous message passing are the two basic communication paradigms supported in Timix. New devices, which are directly controlled by application processes, can be integrated into the system without changing the kernel. In addition, dynamic timing constraints are used for scheduling processes and interprocess communications.

The Robot Control Interface (RCI) [Llo85] was designed to provide a programmer with primitives for writing simple control procedures to operate the individual joints of a robot. RCI consists of two components: a *control* task which periodically executes at a high priority in a non-interruptible context to produce commands for each joint and a *planning* task which provides high level directives to the control task. The control task is restricted from executing UNIX system calls and its code and data pages are locked into memory.

In UNIX, the timing constraints imposed on the system by the robot are not adequately represented. The author ported RCI onto Timix in two different versions: one which does not utilize robot timing constraints for scheduling purposes and one which attempts to employ those timing constraints.

The first port of RCI to Timix provided a set of library routines to mimic the functionality that RCI provided with UNIX. This package required no modifications to the kernel as was required to implement RCI on UNIX: each process is always resident in memory and the device interface allows new applications to be easily added. The same communication program that runs on the PUMA controller for RCI on UNIX is used for RCI on Timix.

The second port attempted to map timing constraints into the various compo-

nents of the RCI interface. At this point, several design flaws within Timix were discovered. Namely, signals and processes are not scheduled for execution in a consistent and uniform manner. Also, if a temporal scope is defined for the main part of a process with a start time in the future, an interrupt can not be processed until the process reaches its start time. With these flaws, there was no straightforward way to implement a version of the RCI using timing constraints for scheduling purposes.

3 Goals and Approach

3.1 Goals

Modern robotics applications are gradually becoming more complex by involving greater numbers of sensors and actuators. The control of such systems require multiple processors as a single processor may not be able to meet the computational demands and certainly cannot physically be connected to all the sensors and actuators due to location and other constraints. Providing a modular system usually involves the specification of a single task for each sensor or actuator in the system. Distributed robotics systems tend to be structured as follows: each sensor task collects data and sends it off to a task to interpret the data, the interpretation task integrates the sensor data and sends the results to the appropriate tasks, and the actuator tasks use the integrated data to determine what to do next. A distributed real-time system is required to perform the required communication and processing while meeting the application specified timing constraints. Our research goal is to:

> **Study the structure of a real-time kernel which facilitates the development of distributed robot systems.**

We are concerned with the development of a suitable real-time kernel which supports distributed robot control systems while solving our previously identified problems. The kernel must provide predictable response with respect to time. Further, the kernel is employed by a single robot control application, i.e., the kernel is dedicated to executing a single application at any given time. However, since the kernel will be used in a robotics laboratory where a single platform is utilized for several different experiments, the kernel must allow for the easy development and modification of applications. The software abstraction employed by the kernel should provide a simple, concise, and consistent operator set.

Using our four previously identified problems, we can briefly examine four subgoals dealing with kernel design which enables our kernel to solve the problems. Note that our research approach is summarized in the next subsection.

1. **The system must guarantee timely response to sporadic entities.** The kernel conforms to a notion of *consistent scheduling*. By consistent scheduling, we mean that all threads are scheduled for execution using the same scheduling paradigm. Periodic and sporadic entities are scheduled in a uniform manner using the same scheduling paradigm. With periodic entities, we use the event mechanism to periodically make the thread runnable (as we explain later).

2. **The system must guarantee that the overall scheduling of tasks is maintained at all times.** Normally, tasks are scheduled to execute based on their timing requirements. If all devices (other than the clock) are polled, then the overall scheduling of tasks is intrinsically maintained. However, if interrupts are used, the kernel does not know in advance when the interrupt will occur. Traditional real-time systems use computer hardware to schedule the execution of interrupt handlers without consideration to the timing requirements of the application. If device interrupts occur more frequently than expected and exceed the design specification, then the processor can be overtaxed and cause timing constraints to be missed.

   To solve this problem, a two-tiered interrupt paradigm using events and lightweight threads is used to guarantee that the overall scheduling of tasks is maintained at all times. The only processing not occurring within a thread is the clock and a minimal interrupt handler. The execution overhead for the clock can easily be predicted – it occurs at regular intervals. Although interrupt handlers execute whenever an interrupt occurs, the processing overhead introduced is strictly bounded and relatively small. The purpose of such an interrupt handler is to convert the interrupt to a passive entity called an event. These events may be used as start conditions for the execution of threads. All threads, whether initiated by communication or by devices, are scheduled using the same scheduler.

3. **Scheduling requirements may change as the system executes or may depend on the system state.** The kernel allows the specification of *explicit dynamic timing constraints* which may be modified by each thread while the system executes. Therefore, if the scheduling requirements change as the system executes or as the system state changes, then the scheduling parameters can be

dynamically modified.

4. **Conversion of application requirements into scheduling parameters should be straightforward.** The ease of design and change in real-time application development is crucial in a research environment where the real-time system changes as the research requirements change. In our notion of consistent scheduling, the kernel provides a default scheduling algorithm based on the earliest-deadline-first algorithm. However, we allow the application programmer to redefine the scheduling algorithm by specifying the desired characteristics among those predefined within the kernel. Thus, the ease of conversion of application requirements into scheduling parameters directly depends on the choice of the scheduling algorithm selected by the application programmer.

3.2 Approach

Our approach is to design a real-time system model suitable for distributed robotics, to realize the system model by implementing a distributed real-time kernel, and to design and implement a robot control application which is typical of those found in distributed robotics.

The real-time system model employs a single scheduling algorithm which is applied to each thread in a consistent manner. This algorithm uses a user-definable scheduling algorithm with application specified timing constraints to provide a predictable response to external events. Applications requiring dynamic scheduling requirements are supported by allowing the application to change the timing constraints as it executes. Furthermore, threads may propagate the timing constraints when they communicate. Communication between threads can functionally follow two paradigms: synchronous message passing and asynchronous message passing. With synchronous message passing, the timing constraints associated with the sending thread is propagated to the receiving thread. The priority associated with the message is used for scheduling the message's communication. With asynchronous communication, the scheduling parameters and message priority associated with the message are only used for communication purposes and not for scheduling the thread's execution.

Not only may applications directly respond to interrupts, but a two-tiered interrupt mechanism is used to provide a greater degree of system predictability. All device interrupts other than those from the clock are converted into events. An event is "triggered" when the interrupt occurs which wakes up any threads waiting on that event by placing the thread in the ready queue. The handlers (or threads) associated with each interrupt will then execute only when its timing constraints put the thread at the head of the ready queue. This approach essentially merges the advantages of both polled and interrupt-driven input/output. A byproduct of this approach is the ease in which experimentation of the tradeoffs of polling and interrupts can proceed.

Given the real-time system model, we intend to realize it by implementing a distributed real-time kernel (Timix (v. 2)) on a network of DEC MicroVAX II processors connected through an Ethernet. The design of the kernel attempts to isolate as many of the low-level hardware dependencies from the operator set as possible.

To evaluate our real-time system model and our kernel implementation, we intend to implement a distributed robotics application. By implementing a "real" robotics application, we can reflect on our kernel design choices and speculate on the solutions to any difficulties caused by them. The robotics implementation will be modular to permit future extensions of the application, such as the addition of another arm or more sensors.

3.3 Research Contributions

The primary contributions of this research is the design, implementation, and evaluation of a distributed real-time kernel using a distributed robotics application:

1. A real-time system model is developed. Its salient features are the notion of consistent scheduling, the use of explicit timing constraints, and a two-tiered interrupt structure.

2. A kernel is realized using the system model as its base. The kernel permits experimentation with various levels of time granularity, with real-time protocols and networks, and with the tradeoffs between interrupts versus polling.

3. A distributed robot control application will be realized to show the effectiveness of the real-time kernel and the model on which it is based.

4 Distributed Real-Time Kernel

Timix (v. 2) is a distributed real-time kernel designed support distributed robotics applications with predictable execution and interprocess communication. The application programmer can specify timing constraints for thread execution and interthread communication. These timing constraints directly reflect the timing requirements of the application. The kernel uses these constraints for scheduling threads and communications. This approach of treating time explicitly facilitates the implementation and debugging of time dependent application programs.

The key components of the kernel include:

Execution environment is the basic unit of resource allocation, has an independent address space, and does not execute.

Thread represents a logically independent execution thread of control.

Scheduler permits threads to only wait on the occurrence of an event. Depending upon the type of the thread, timing constraints may be specified for each thread (start time, execution limit, and deadline).

Event is used for timing, system, and processor errors, synchronization, periodic events, and device interrupts. When it is triggered (sent), the value and the identifier of the entity which requested the trigger is entered into a circular queue. All event triggering occurrences are remembered until the queue fills.

Port has been extended for real-time communication by allowing the sender to pass timing constraint information in messages and the receiver to control message queuing and reception strategies.

Device may be directly controlled by application threads. All device interrupts, other than the clock, trigger events which may, in turn, cause a thread to execute.

Memory segment permits regions of memory to be shared between execution environments on the same processor.

4.1 "Process" Model

A distributed real-time application consists of a set of execution environments. Each execution environment consists of a set of threads and a set of resources. A resource is either a port, a device, a shared memory segment, or an event. Threads are defined by the programmer and represent a logically independent execution thread of control. Each execution environment services as the basic unit of resource allocation and has an independent address space. The execution environment does not execute.

All threads are initially non real-time threads. Threads may wait on events and events may be triggered by alarms (at a specific time or periodically), by ports when messages arrive, by devices when they interrupt, or by threads when they explicitly trigger an event. A thread becomes a real-time thread when it either specifies a timing constraints or recieves a synchronous message with timing constraints.

Unique system-wide id is used to refer to all execution environments, threads, ports, memory segments, devices, events, and alarms.

Every execution environment has read-only access to a global data page 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. Threads 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.

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 execution environments. Application devices are directly controlled by particular application execution environments and include the analog-to-digital conversion board required for the wrist and the parallel interface board required for the end-effectors. The kernel converts a device interrupt into an event and provides shared memory between a device and an execution environment.

The kernel Timix (v. 2) is viewed as a single execution environment with threads

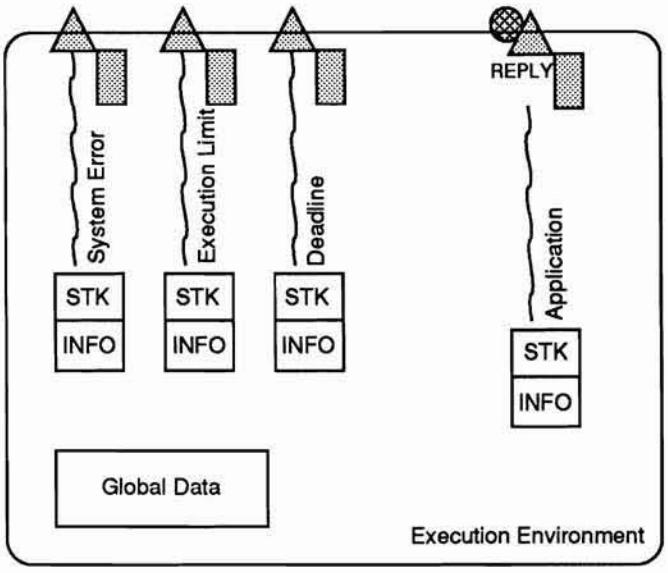

Figure 4: Default Execution Environment

connected to each service port. Threads issue service requests to the kernel by sending synchronous messages or by requesting a system call. A *service port* is a reception port used to receive requests for services from a thread. 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, events, alarms and interprocess communication. Server threads provide non-time critical services such as thread creation, terminal input/output and device manipulation. An application process sends a service request to a server thread and waits for a reply if needed.

In either case, the updated scheduling parameters (such as execution time remaining) are passed to the kernel code responsible for handling the request. In this way, the kernel only uses the CPU resources that have been previously allocated by the application making the request. The kernel can only attempt to satisfy the request as long as the application has not exceeded its timing constraints. When the kernel finishes its processing, the scheduling parameters of the application are modified by the resources utilized by the kernel. This is particularly applicable for execution limits.

4.1.1 Execution Environment

The kernel operator set for the execution environment requires three basic operations: creating a new execution environment, obtain status about an execution environment, and deleting an execution environment.

- error_t execenv_create(eid, codesize, code, data, datasize, dynamic, stack)
  The **execenv_create** operator creates a new execution environment and returns the unique exection environment identifier.

- execenv_status(eeid, status)
  The **execenv_status** operator obtains status information about an execution environment. Status information essentially describes the contents of the environment – such as the number of threads, memory segments, ports, devices, and events associated with the environment.

- error_t execenv_delete(eid)
  The **execenv_delete** operator deletes an execution environment. This operation is restricted to threads within the execution environment and to privileged threads.

4.1.2 Threads

Thread consists of register context, stack with thread information, and scheduling information. Within a thread, a single thread of control is maintained at all times. The read-only thread information structure contains reply information, current scheduling information, error event information structures, a pointer to the global data page, and start event alarm information.

The kernel operator set for threads require three basic operations: creation of a runnable thread, obtaining status about a thread, and deleting a thread.

- error_t thread_create(tid, eid, entry, stacksize, argstring, argsize)
  The **thread_create** operator creates a new, runnable thread within execution

environment *eeid* and returns a unique thread identifier. The next two arguments are buffers containing *code* and *data*. The final argument specifies the relative entry point within the code buffer.

- error_t thread_status(tid, status)
  The **thread_status** operator obtains status information about a thread. Status information essentially contains the processor status block and a copy of the current scheduling parameters.

- error_t thread_delete(tid)
  The **thread_delete** operator deletes a thread. This operation is restricted to threads within the same execution environment and to privileged threads.

- error_t thread_events(tid, syserr, execlimit, deadline)
  The **thread_events** routine indicates to the kernel which events should be triggered in the event of a system error, execution limit exceeded, and deadline missed errors.

- error_t thread_reply(tid, port, event)
  The **thread_reply** routine inserts into the thread information data structure, the default reply port and events – used when issuing synchronous library routines.

- error_t thread_save_context(regs, val)
  The **thread_save_context** routine is similar to the Unix routine *setjmp*.

- error_t thread_restore_context(tid, regs, val)
  The **thread_restore_context** routine is similar to the Unix routine *longjmp*.

4.1.3 Nameserver

A distributed nameserver is used for converting resource, thread, and execution environment names to their unique identifier.

- error_t ns_enter(id, name, size)
  The **ns_enter** operator is used to enter the string *name* of length *size* with the identifier *id*.

- error_t ns_lookup(name, size, id)
  The **ns_lookup** operator is used to lookup the string *name* of length *size* in the nameserver and *id* is returned.

- error_t ns_delete(id)
  The **ns_delete** operator is used to delete the identifier *id* and all the information associated with it.

4.2 Scheduling and Timing Constraints

Timix (v. 2) supports both real-time and non real-time threads. In addition, real-time threads are prioritized as imperative, hard real-time, and soft real-time since not all real-time processes are equally important [KDK*89]. Threads are executed in the order of priorities. Within the same priority, imperative threads are executed on a first-come-first-serve basis, whereas hard and soft real-time threads 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 threads are ready, non real-time threads 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 [Mok84]. 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.

- error_t sched_get_param (tid, type, startid, limit, deadline, prio)
  The **sched_get_param** operator is get the current scheduling parameters for the thread *tid*. The five remaining arguments are the same as the **sched_set_param** operator.

A timing constraint is violated if either the thread executes longer than the maximum execution time or the deadline for the thread is exceeded. When this happens, the kernel triggers the appropriate event. If the timing constraint is missed in a hard real-time thread, then a critical system error has occurred since the constraint was guaranteed by the scheduler. Thus, the thread associated with the maximum execution time event runs as an *imperative* thread so that a controlled shutdown of the system can occur as soon as possible. However, when a soft real-time constraint is violated, the thread associated with the maximum execution time event runs as a soft real-time thread 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.

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 thread is waiting for a response from a low priority thread and a middle priority thread preempts the execution of the lower priority thread [SLR86]. Extra delay is incurred by the high priority thread since the middle priority thread must complete before the low priority thread resumes. The timing constraint inversion problem can occur between threads of different priorities and between real-time threads with different deadlines. The extra delay from a middle priority thread 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 interthread communication. In the first case, suppose a soft real-time thread issues a request to a non real-time server thread. The timing constraints associated with the message are propagated to the server thread so that it becomes a soft real-time thread to handle the message. After a reply is sent, the priority of the server thread is reduced to that of a non real-time thread. Similarly, timing constraints are propagated from a sender to a receiver within the same real-time priority level.

4.3 Alarms

To facilitate the resumption of threads at particular times, an alarm package allows events to be triggered at specified times. There are two distinct kernel operators used to set an alarm: one for a single triggering at a particular time and one for a trigger at time *start* and repeating every *period* time intervals until it has been cleared.

- error_t alarm_trigger(aid, tv, eid, val)
  The **alarm_trigger** operator is used to insert a single alarm into the alarm queue so that at time *tv*, event *eid* is triggered with value *val.*

- error_t alarm_periodic(aid, start, period, eid, val)
  The **alarm_periodic** operator is used to insert a periodic alarm in the alarm queue so that at time *start*, event *eid* is triggered with value *val.* Every time period *period* thereafter, event *eid* is triggered with value *val.*

- error_t alarm_clear(aid)
  The **alarm_clear** operator clears the specified alarm.

4.4 Resources

In this section, we describe the operator set for the four basic resources: events, ports, devices, and memory segments. The required operators for resources may be classified according to five categories. First, a resource is *created* or defined with the appropriate attributes. At this time, it is not associated with any particular execution environment and no thread may access it until it is allocated. Next, a resource is allocated to an execution environment by *getting* it. The *status* of a resource may be queried by any thread. When the execution environment is finished with the resource, it is *freed* by a thread within that environment so that other execution environments may obtain access. Finally, if the resource is not needed within the system anymore, it may be *deleted.*

4.4.1 Event

The event manipulation operator set includes:

- error_t event_create(eid)
  The **event_create** operator creates an event.

- error_t event_get(eid, event_list, size)
  The **event_get** operator is used to allocate a previously created event and associates the event with a circular list to store triggering information (value and whom from).

- event_status(eid, status)
  The **event_status** operator obtains status information about an event.

- event_free(eid)
  The **event_free** operator is used to return an event back to the system.

- event_delete(eid)
  The **event_delete** operator deletes an event from the system.

4.4.2 Port

Asynchronous messages allows messages to be sent from one execution environment to another. Ports are used for specifying the destination of a message. Every thread is created with a default reception port. This port is used when requesting services from system server threads. Ports are used to queue incoming messages for delivery to an execution environment.

The port manipulation operator set includes:

- error_t port_create(pid)
  The **port_create** operator creates a port.

- error_t port_get(pid)
  The **port_get** operator is used to allocate a previously created port (*pid*).

- error_t port_free(pid)
  The **port_free** operator is used to return an allocated port back to the system. Thus, one execution environment may process a specified number of messages and then give the port to another execution environment for its threads to perform further processing.

- error_t port_delete(pid)
  The **port_delete** operator deletes a port from the system so that no more messages may be received by the port.

- error_t port_attrib(pid, change, attrib)
  The **port_attrib** operator can either be used to change the attributes of a port or to obtain status information about a port. Status information essentially contains the attributes about the port and the number of messages pending. There are various attributes that can be changed by the execution environment currently using the port. First, the ordering of messages within a queue is either by message sent time, by message arrival time or by lowest priority (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 [SBWT87]. Fourth, messages can be received explicitly or asynchronously. This is described later. For instance, if the port should notify the execution environment that a message has been received asynchronously, then one attributes specifies which event should be triggered when a message arrives.

4.4.3 Device

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 threads to directly control devices.

To control a device, a thread 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 thread may request to the device server that device interrupts be converted to 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 [CMM87]. 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.

The device manipulation operator set makes as few assumption as possible about the architecture under which the kernel is implemented. The device manipulation operator set includes:

- error_t device_create(did)
  The **device_create** operator creates a device.

- error_t device_get(did)
  The **device_get** operator is used to allocate a previously created device (*did*).

- error_t device_status(did, status)
  The **device_status** operator obtains status information about a device. Status information essentially contains information about the interrupt vectors and associated event identifiers.

- error_t device_free(did)
  The **device_free** operator is used to return a device back to the system. Note,

the device retains its interrupt attributes until it is deleted from the system.

- error_t device_delete(did)
  The **device_delete** operator deletes a device from the system.

The interrupt vector manipulation operator set also tries to make as few assumptions as possible about the architecture under which the kernel is implemented. The interrupt vector operator set includes:

- error_t intr_alloc(did, vector)
  The **intr_alloc** operator is used to allocate a floating interrupt vector. This vector is required for those devices that do not contain hardwired interrupt vectors.

- error_t intr_reserve(did, vector)
  The **intr_reserve** operator is used to allocate a fixed interrupt vector. This vector is required for those devices that contain hardwired interrupt vectors.

- error_t intr_assoc(did, vector, eid, val)
  The **intr_assoc** operator is used to associate an interrupt vector with an event and value. Whenever an interrupt occurs on this vector, the event *eid* will be triggered with the value *val*.

- error_t intr_disassoc(did, vector)
  The **intr_disassoc** operator is used to disassociate an interrupt vector with an event. If an interrupt occurs after this operation is performed, a "STRAY" interrupt occurs.

4.4.4 Memory

The memory segment operator set makes as few assumptions as possible about the architecture under which the kernel is implemented. The term *region* refers to a independent addressing spaces within the node's architecture. We assume that there are at most two such regions: data and device. The *private* region is the main memory directly accessible to the current processor (typically used by threads within execution

environments by default), the *device* region is used for input/output (used by devices and their interrupt handlers), and the *bus* region is used for devices and memory accessible through the bus.

The memory segment operator set includes:

- error_t memory_create(mid, type, start, size)
  The **memory_create** operator creates a memory segment in the region specified by *type*, starting at address *start* with a length of *size* bytes. The address is specified according to the region in which the segment is to be allocated.

- error_t memory_get(mid, location)
  The **memory_get** operator is used to allocate a previously created memory segment (*mid*) starting at address *location* within the data region.

- error_t memory_status(mid, status)
  The **memory_status** operator obtains status information about a memory segment. Status information essentially contains the attributes about the memory segment and the entity to which it is allocated.

- error_t memory_free(mid)
  The **memory_free** operator is used to return an allocated memory segment back to the system. Note, the memory segment retains its attributes until it is deleted from the system.

- error_t memory_delete(mid)
  The **memory_delete** operator deletes a memory segment from the system.

- error_t memory_onto_busmem(mid1, mid2)
  The **memory_onto_busmem** operator is used to map private memory segment *mid1* onto the bus memory segment *mid2*.

4.5 Communication

Real-time systems are asynchronous in nature and require predictably fast communication. Often more important than actual speed of communication is predic-

tability [SR87b]. Commonly used synchronization and communication primitives such as signals and messages based on ports have been designed without considering guaranteed response. Timix (v. 2) provides two basic communication methods:

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

4.5.1 Events

The simpliest and quickest communication technique is *events*. Associated with each event is a circular queue of pending events. We say that an event is *valid* is the circular queue is not empty – i.e., the event has been triggered at least once for which processing has not occurred. The circular queue allows the value and the entity which triggered the event to be remembered until the queue is full. Once the queue becomes full, an *overflow* flag is set to **TRUE**. Events may be triggered by other threads, notification of message arrival (ports), alarms, and on a device interrupt.

Events are used by the kernel to notify a thread than an error has occurred. The purpose of sending an event in this case is to give the thread a chance to clean up its state or to perform a controlled shutdown of the system. There are three types of errors: timing errors, process errors, and system errors. Timing errors are only with respect to the timing constraints of the current temporal scope. The kernel triggers the execution limit event if a real-time thread executes longer than the maximum execution time, and the deadline evnet if a real-time thread misses its deadline. Unlike timing errors, events may be triggered for process and system errors in non real-time threads. Process errors are errors due to an execution environment 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.

The operator set for trigger events and obtaining their values is:

- error_t event_trigger(eid, value)
  The **event_trigger** operator triggers the event *eid* and enters the following pair

in the event's circular queue: value *value* and from the current thread.

- error_t event_remove(event_list, val, from)
  The **event_remove** operator removes the next *value* and event *from* information from the circular queue associated with the event.

4.5.2 Messages

Timix (v. 2) provides asynchronous message-based communication as its most primitive communication paradigm. Its operators may be combined to support synchronous communication and blocked message reception. With synchronous message communication, the scheduling parameters associated with each message is propagated and used when scheduling the thread services the call. With asynchronous message communication, the scheduling parameters are only used for scheduling message transmission and ordering with the destination port's queue.

A thread 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 message priority and scheduling propagation parameters with each message. The message priority is used for scheduling message transmission and the scheduling propagation parameters are used for scheduling the execution of the receiver thread. The scheduling propagation parameters include the start time, the maximum execution duration, and the deadline. The propagation parameters are only used if they are more critical (of higher priority) than those currently specified by the receiver. This feature allows non real-time server threads to handle requests from threads of higher priority (i.e., real-time threads).

Send The message communication primitives include:

- error_t msg_send(hdr, data)
  The **msg_send** operator is used to send a message to another port.

- error_t msg_forward(device, protocol, hdr, data)
  The **msg_forward** operator is used to forward a message to the appropriate server port which corresponds to the specified device and protocol.

- error_t msg_reply(hdr, data)
  The **msg_reply** operator is used to send a reply message to the specified port.

Receive The message communication primitives include:

- error_t msg_recv(port, hdr, data)
  The **msg_recv** operator is used to receive messages from the port *port*.

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 **msg_recv** system call when the receiver needs to receive the message. Since it is possible that the message is not received before the scheduling parameters have exceeded its deadline, the kernel does not utilize the scheduling propagation parameters. To use these timing constraints after receiving the message, the receiver thread must explicitly update its scheduling information.

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 thread performs some task and incoming messages need to provide some simple service that can be performed at any time. To wait for a message when asynchronous delivery is enabled, a thread must specify that the event is the thread's start condition. The notification of message arrival is through an event associated with the port. When a message arrives on a port, an event is sent to the thread which owns that port.

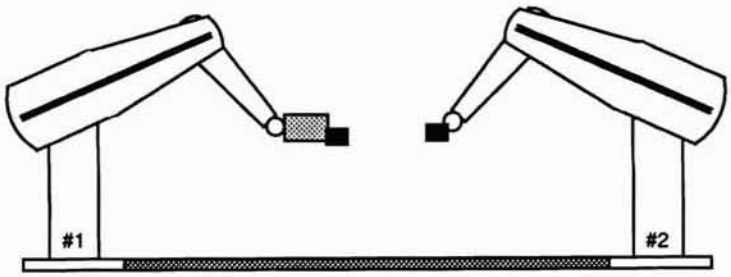

Figure 5: Robotics System Architecture

5 Robot Application

A two-manipulator arm robot control system was proposed in the Introduction. Its goal is to use a wedge locate circular objects with spillable contents, move them, pick them up, and pour their contents out. Since the wedge must maintain a gap of one half inch between it and the top of the surface until it is time to lift the object (in mode 3), the application requires the use of Cartesian motion. Paul [Pau81] defines Cartesian motion as movement along straight lines and rotation about fixed axes in space. When an object is found, the wedge pushes it into a common area reachable by each robot arm (see Figure 1 on page 4). Once the objects are in this common area, they are picked up by applying forces to the opposite sides of the object by the two arms. To permit coordination between the robot arms, it is essential that their controllers communicate with each other in real-time.

5.1 System Architecture

The system architecture of the robot control system consists of two six-degree-of-freedom manipulator arms (PUMA 560s) and is shown in Figure 5. PUMA #1 and PUMA #2 are each equipped with a wedge-like end effector. A wrist [XP88] is located between the end effector (hand) and the manipulator arm on PUMA #1.

The wedge-like end effector is shown in Figure 6. It is made of a three inch "L-bracket" – two three inch square aluminum pieces connected at a right angle. Located on the interior sides are two contact sensors made from two push-button

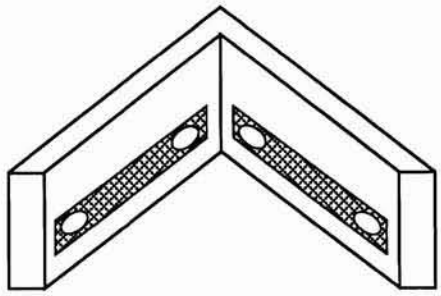

Figure 6: Wedge End-Effector

switches connected by a flat metal plate (similar in construction to a space bar). The contact sensors can be used to detect contact with either the left face, the right face, or both interior faces.

Each robot arm has a low-level dedicated processor which acts as a joint controller by issuing commands to the joint actuators and determining position from the joint encoders. 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 lowest levels of most robot control systems are servo loops running on dedicated processors. In these systems, the error values used as input are represented by continuously varying values. Since a missing value represents a continuously varying quantity, it is feasible for the receiving process to use the value from the previous cycle. This means that some of the requirements usually imposed on interprocess data communication can be relaxed. The loops must be fast enough to provide smooth operation of the device and to keep the device within operational limits. Once servo loop processes are given their input data, they can proceed to completion without further interaction and typically do not block [Cla89].

At a higher level, the robot control system consists of three MicroVAX processors connected through a 10 Mb Ethernet and a 10 Mb ProNET-10 token ring. These MicroVAX processors use Timix (v. 2) to provide the computation power needed for the timely execution of threads used in determining the path of the robot arms, in sensing contact by the end-effector, in determining forces on the wrist, and in

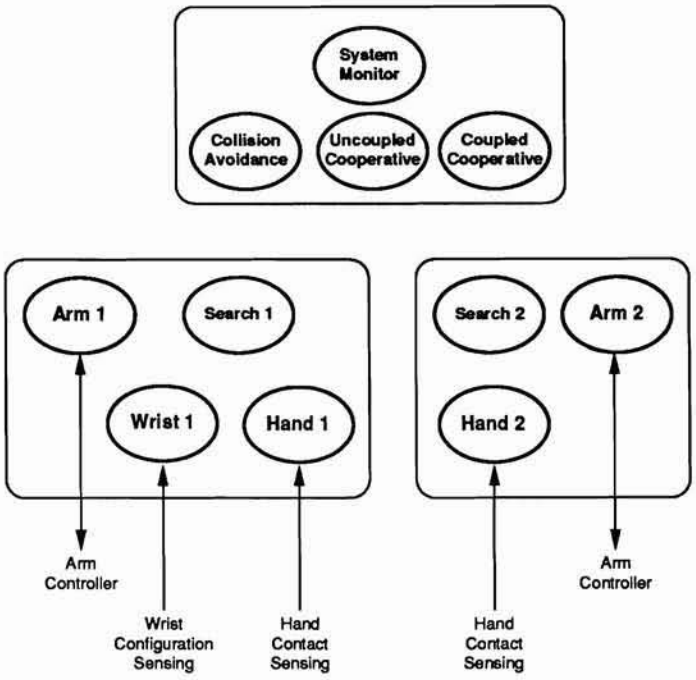

Figure 7: Two Manipulator Arm Robotics System Model

avoiding collisions between the manipulator arms. For simple sensors, we gather the information directly and bypass the use of dedicated processors. The output of each degree of the wrist sensor is an analog signal. This signal is converted to a digital value by the analog-to-digital converter board in a MicroVAX. The contact sensors on each hand (wedge) is connect to a MicroVAX via a parallel interface.

5.2 Logical Organization

Figure 7 provides an overview of the two manipulator arm exploratory robotic thread model and Table 2 lists the attributes of the threads. Each robot manipulator arm has a thread, a_1 and a_2, which is used to communicate with the low-level joint controller. These threads operate with a sampling period of 28 milliseconds. In addition, there is a thread for each wedge end-effector, h_1 and h_2. Until both arms are lifting an object (mode 3), the hand threads operate with a sampling period of 28 milliseconds. The wrist thread, w_1, also operates with a sampling period of 28 milliseconds when the two arms are lifting or manipulating an object.

Symbol	Thread	Type	Period/Deadline
w_1	wrist	periodic	28 ms
a_1	manipulator arm	periodic	28 ms
h_1	hand	sporadic	28 ms
s_1	search strategy	sporadic	
in_1	network input	sporadic	varies
out_1	network output	sporadic	varies
a_2	manipulator arm	periodic	
h_2	hand	sporadic	
s_2	search strategy	sporadic	
in_2	network input	sporadic	varies
out_2	network output	sporadic	varies
$avoid_3$	collision avoidance	sporadic	
$uncouple_3$	uncoupled cooperative	sporadic	
$coupled_3$	coupled cooperative	sporadic	
mon_3	system monitor	sporadic	
in_3	network input	sporadic	varies
out_3	network output	sporadic	varies

Table 2: Robot Control Threads

Operating at a higher level are the collision avoidance thread ($avoid_3$), the uncoupled cooperative thread ($uncouple_3$), the coupled cooperative thread ($coupled_3$), the search path threads (s_1 and s_2), and the system monitor thread (mon_3). The collision avoidance mode 1 thread executes in a sporadic fashion whenever new data arrives from the arms and determines whether collision is imminent. The uncoupled cooperative mode 2 thread executes in a sporadic fashion whenever new data arrives from the arms and determines a path to enable them wedges to meet on opposite sides of the objects so a "grasp" can occur. The coupled cooperative mode 3 thread executes in a sporadic fashion generating destination goals for the object and sending the goals to the arms to carry out the appropriate motion. The search path threads are used to compute in which directions the robot arm must move while searching for another object to pick up and empty. The system monitor thread is used to monitor system activity and display the actions as they occur.

In determining the scheduling requirements for the system, we must also examine the network communication requirements. There are six communication threads

associated with network communication. The network *in* threads (in_1, in_2, and in_3) are used for receiving messages from other machines. The network *out* threads (out_1, out_2, and out_3) are used for transmitting messages to other machines. These threads execute at sporadic intervals with varying degrees of priority and/or deadlines, depending upon the content of the messages.

5.3 Modes of Operation

In this section, we examine the three basic modes of operation in further detail. For each mode, there is a table summarizing how communication flows through the system by showing the source and destination of each message or shared memory operation. The first column, *source*, specifies the thread sending the message or writing the data into shared memory. The second column, *data*, specifies the type of data that is being transferred. The third column, *deadline*, specifies the deadline by when the message should arrive. Since the size of the period is 28 milliseconds for the arm threads, most communication should complete before the next period. The fourth column, *destination*, specifies the receiver thread. The fifth column, *model*, specifies the communication model used. The communication model may either be current values or historic values[2]. A task may use as input the *current values* of various hardware sensors and software variables and produce as output a set of current values. Inputs are always assumed to be present and the output values overwrite previous values. A task may use as input the *historic values* containing commands or data values and produce similar values as output. These historic values are usually in the form of messages which are queued up and never overwritten. The sixth column, *method*, specifies the method of communication: either messages or shared memory.

5.3.1 Mode 1: Search under collision avoidance

The initial configuration of the system has each end-effector placed about one-half inch above the surface in a known and calibrated position. In this mode, the two

[2]The two communication models were originally defined by Schwan in [SBWT87]. He used the term "discrete messages" where we have used the term "historic values."

Source	Data	Deadline	Destination	Model	Method
h_1	contact data	28ms	a_1	current values	shared memory
a_1	search path done	28ms	s_1	current values	shared memory
a_1	arm state	28ms	$avoid_3$	current values	message
s_1	next search path		a_1	historic values	message
h_2	contact data	28ms	a_2	current values	shared memory
a_2	search path done	28ms	s_2	current values	shared memory
a_2	arm state	28ms	$avoid_3$	current values	message
s_2	next search path		a_2	historic values	message
$avoid_3$	collision imminent	varies	a_1	current values	message
$avoid_3$	collision imminent	varies	a_2	current values	message

Table 3: Communication Flow Under Mode 1

arms are looking for objects in an independent fashion. The search thread computes a search path and then passes the information to the arm thread by sending a message to the appropriate port. The arm then moves along the path and its thread polls shared memory to determine if the wedge has contacted an object. In addition, the arm thread checks for messages from the collision avoidance thread to see if collision between the two arms is imminent and sends information to the collision avoidance thread at the end of each sampling period. The hand thread checks the state of its contact sensors every sampling period and relays the information to the arm thread via shared memory. Once stable contact with an object has been achieved, a message is sent to the system monitor thread. This thread then moves the system into *mode 2* by disabling the collision avoidance and search path threads.

5.3.2 Mode 2: Object found and push to common area

Once an object has been found, it must be moved into an area between the two robots so that each end-effector may approach the object from opposite sides. Let the arm which found an object be designated A and the other arm be designated B. Arm A carefully pushes the object by computing the path to a known location while monitoring shared memory to ensure that contact is maintained with the object. Arm B follows the directions received from the uncoupled cooperative thread $uncouple_3$

Source	Data	Deadline	Destination	Model	Method
h_A	contact data	28ms	a_A	current values	shared memory
a_A	arm state	28ms	$uncouple_3$	current values	message
h_B	contact data	28ms	a_B	current values	shared memory
a_B	arm state	28ms	$uncouple_3$	current values	message
$uncouple_3$	path correction	varies	a_A	current values	message
$uncouple_3$	path correction	varies	a_B	current values	message

Table 4: Communication Flow Under Mode 2

Source	Data	Deadline	Destination	Model	Method
w_1	wrist data	28ms	a_1	current values	shared memory
a_1	arm state	28ms	$coupled_3$	current values	message
a_2	arm state	28ms	$coupled_3$	current values	message
$coupled_3$	object path	varies	a_1	current values	message
$coupled_3$	object path	varies	a_2	current values	message

Table 5: Communication Flow Under Mode 3

and moves its end-effector to the appropriate position. While arm B is moving, the thread a_B monitors the shared memory containing data from its end-effector – if contact with an object is detected, the location is noted and the arm moves to one side of the object. Once the two arms are in the appropriate position, arm B moves until it contacts the object. At this point, we reach *mode 3.*

5.3.3 Mode 3: Manipulate object with both arms

Now that the object has been "gripped" using the forces applied by both arms, the object must be lifted, emptied, and dropped. The arm with the compliant wrist is known as the slave arm and the one without the wrist is known as the master arm. From this point on, the Cartesian reference frame for both robots is now located within the object. The coupled cooperative thread $coupled_3$ sends the path information to the master arm. The slave arm uses the data received from the wrist to determine how it should react to the movement while maintaining a stable grip on the object.

Once the object is over the bucket, the contents are emptied out by rotating the object. Then the object is moved over a box and dropped.

6 Research Plan

In this section, we subdivide our research into three areas: the distributed real-time kernel, the robotics application, and the evaluation of the combined kernel and robotics application. In each of these areas, we describe our current status and summarize the remaining research required to complete this dissertation. Finally, we conclude with a timetable.

6.1 Kernel

The design for the distributed real-time kernel has been completed. Furthermore, the realization of this kernel as Timix (v. 2) is about two-thirds complete. The current implementation supports the scheduling of threads with explicit timing constraints, allows threads to directly control devices, and permits threads to communicate locally via asynchronous message passing. The design of the kernel provides several additional features which have not yet been implemented. First, the scheduler does not currently guarantee in advance that hard real-time constraints will be met. Second, the shared memory segments have not been implemented. Third, there is no network or protocol support for either Ethernet or ProNET-10. The second and third features must be completed before the kernel can be used to develop applications. Finishing the network and protocol part of the kernel should be relatively straightforward as it can be based on Timix (v. 1).

6.2 Application

The basic design for the robotics application has been completed. The wedge-like end-effectors are being built by the machine shop and should be finished in several weeks. However, the crux of our remaining work is to build an implementation of the robotics application, include the robot control interface (RCI). We believe that the remaining work can be ordered as follows:

1. Port the RCI implementation from Timix (v. 1) to Timix (v. 2). Although this version of RCI works under Timix (v. 1), it does not utilize the explicit timing constraints that Timix provides.

2. Integrate timing constraints and real-time communication into RCI so that the implementation utilizes the real-time features provided by Timix (v. 2).

3. Add Cartesian mode to the Timix RCI so that the appropriate motions can be provided by the robot control interface. By default, RCI only employs *joint motion* which is not along straight lines or along any other simple, well defined path.

4. Program the two manipulator arms as separate components without any communication between them. This is to ensure that a single arm can search for objects and push them.

5. Program the two manipulator arms to work at pushing objects and communicating. This is to ensure that we can manipulate objects in a common workspace and that the robots are calibrated to one world view.

6. Program the two manipulator arms to work together to lift an object. This is the final step.

6.3 Measurement and Kernel Design Alternatives

Once the distributed real-time kernel and the robotics application have been implemented, we can experiment with various types of communication networks and levels of time granularity. We intend to measure how the changes affect robot performance.

To obtain accurate performance measurements, the Codar Technology timer/counter board must be integrated into the kernel. A library has been implemented to enable the control of this board which contains five individually controllable counters with a programmable resolution as fine as 250 nanoseconds. Once the library routines have been integrated into the kernel, the accuracy of the clock for timing and alarms can be set at one millisecond, instead of the ten milliseconds provided by the built-in interval time on the MicroVAX II CPU.

Ethernet is provided as the default communication network. However, as we know, the network access delay is not bounded. Another network option is to utilize a Proteon token ring (ProNET-10). We envision various experiments by comparing the communication delays encountered when using the different networks.

6.4 Timetable

The timetable for the proposed work follows:

1. Finish the implementation of Timix (v. 2) – three weeks.
2. Implement RCI under Timix (v. 2) – three weeks.
3. Implement robotics application – five weeks.
4. Write/defend dissertation – two months.

References

[BS88] Theodore P. Baker and Alan Shaw. The cyclic executive model and Ada. In *Proceedings of the Real-Time Systems Symposium*, pages 120–129, December 1988.

[Car84] G. D. Carlow. Architecture of the space shuttle primary avionics software system. *Communications of the ACM*, 27(9):926–936, September 1984.

[Cla89] Dayton R. Clark, Jr. *Data Communication in Robot Control Systems.* PhD thesis, Department of Computer Science, New York University, Courant Institute of Mathematical Sciences, May 1989. Also available as Technical Report No. 436 and Robotics Report No. 193, March 1989.

[CMM87] Michael F. Coulas, Glenn H. MacEwen, and Genevieve Marquis. RNet: a hard real-time distributed programming system. *IEEE Transactions on Computers*, C-36(8):917–932, August 1987.

[DRSK89] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz. The real-time operating system of MARS. *ACM OPERATING SYSTEMS REVIEW*, 23(3):141–157, July 1989.

[Gag86] R.D. Gaglianello. A distributed computing environment for robotics. In *Proc. of 1986 IEEE International Conference on Robotics and Automation*, pages 1890–1895, IEEE Council on Robotics and Automation, IEEE Computer Society Press, 1986.

[GK86] Robert D. Gaglianello and Howard P. Katseff. Communications in Meglos. *Software – Practice and Experience*, 16(10):945–963, October 1986.

[GLR82] Patricia Garetti, Pietro Laface, and Silvano Rivoira. MODOSK: a modular distributed operating system kernel for real-time process control. *Microprocessing and Microprogramming*, 9:201–213, April 1982.

[GS89] Prabha Gopinath and Karsten Schwan. CHAOS: why one cannot have only an operating system for real-time applications. *ACM OPERATING SYSTEMS REVIEW*, 23(3):106–125, July 1989.

[Hen88] John Henize. Understanding real-time UNIX. Marketing Brochure from MASSCOMP, February 1988. Reference Number 080-01134-00 0288-1134.

[Jen89] E. Douglas Jensen. Alpha promotes BM/C^3 operations. *Defense Computing*, January/February 1989.

[KB87a] Andrew Kun and John Barr. The Computer X distributed, real-time system. In *Proc. of the IEEE Fourth Workshop on Real-Time Operating Systems*, pages 55–58, July 1987.

[KB87b] Andrew Kun and John Barr. Computer X, Inc. real-time, distributed, operating system. July 1987. Photocopy of Transparencies from the Presentation at the IEEE Fourth Workshop on Real-Time Operating Systems.

[KDK*89] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed fault-tolerant real-time systems: the mars approach. *IEEE Micro*, 9(1):25–40, February 1989.

[Kin87] Robert Bruce King, II. *Design and Implementation of a Real-Time Distributed Kernel.* Master's thesis, Department of Computer and Information Science, University of Pennsylvania, August 1987.

[Kin89] Robert B. King. *Timix Manual.* GRASP Laboratory, University of Pennsylvania, Philadelphia, PA 19104, April 1989. Version 1.0.

[KKS89] Dilip D. Kandlur, Daniel L. Kiskis, and Kang G. Shin. HARTOS: a distributed real-time operating system. *ACM OPERATING SYSTEMS REVIEW*, 23(3):72–89, July 1989.

[Lee84] Insup Lee. A programming system for distributed real-time applications. In *Proceedings of the Real-Time Systems Symposium*, pages 18–27, December 1984.

[LG85] Insup Lee and Vijay Gehlot. Language constructs for distributed real-time programming. In *Proceedings of the Real-Time Systems Symposium*, pages 57–66, December 1985.

[LGCS89] Stephen Leake, Tom Green, Sue Cofer, and Tim Sauerwein. Hierarchical Ada robot programming system (HAPRS): a complete and working telerobot control system based on the NASREM model. In *Proc. of 1989 IEEE International Conference on Robotics and Automation*, pages 1022–1028, IEEE Council on Robotics and Automation, IEEE Computer Society Press, May 1989.

[LK88] Insup Lee and Robert King. Timix: a distributed real-time kernel for multi-sensor robots. In *Proc. of 1988 IEEE International Conference on Robotics and Automation*, pages 1587–1589, IEEE Council on Robotics and Automation, IEEE Computer Society Press, April 1988.

[LKP89] Insup Lee, Robert B. King, and Richard P. Paul. A predictable real-time kernel for distributed multi-sensor systems. *IEEE Computer*, 22(6):78–83, June 1989.

[LKY88] Insup Lee, Robert King, and Xiaoping Yun. A real-time kernel for distributed multi-robot systems. In *Proc. of the 1988 American Control Conference*, pages 1083–1088, American Automatic Control Council, June 1988.

[Llo85] John Lloyd. *Implementation of a Robot Control Development Environment.* Master's thesis, Computer Vision and Robotics Laboratory, Department of Electrical Engineering, McGill University, December 1985.

[LTCA89] Shem-Tov Levi, Satish K. Tripathi, Scott D. Carson, and Ashok K. Agrawala. The MARUTI hard real-time operating system. *ACM OPERATING SYSTEMS REVIEW*, 23(3):90–105, July 1989.

[Mah88a] Jennifer Maher. Vxworks enhanced with release 4.0. *VxWords Newsletter*, 1(1):1 & 3, Summer 1988. Marketing Newsletter for Wind River Systems, Inc.

[Mah88b] Jennifer Maher. Wind river expands vxworks kernel options. *VxWords Newsletter*, 1(1):2, Summer 1988. Marketing Newsletter for Wind River Systems, Inc.

[Mok84] Aloysius K. Mok. The design of real-time programming systems based on process models. In *Proceedings of the Real-Time Systems Symposium*, pages 5–17, December 1984.

[MWB89] Amante A. Mangaser, Yulun Wang, and Steven E. Butner. Concurrent programming support for a multi-manipulator experiment on RIPS. In *Proc. of 1989 IEEE International Conference on Robotics and Automation*, pages 853–859, IEEE Council on Robotics and Automation, IEEE Computer Society Press, May 1989.

[Pau81] Richard P. Paul. *Robot Manipulators: Mathematics, Programming, and Control.* The MIT Press, Cambridge, Massachusetts, 1981.

[SARTICS89] Proposal for the development of: standard architecture for real-time intelligent control systems (SARTICS). October 1989. Submitted to: LtCol Eric Mettala, DARPA, ISTO; Submitted by National Institute of Standards and Technology, Robotic Systems Division.

[RAA*88] Marc Rozier, Vadim Abrossimov, Francois Armand, I. Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser. CHORUS distributed operating systems. *Computing Systems*, 1(4):305–370, Fall 1988.

[Rea86] James F. Ready. VRTX: a real-time operating system for embedded microprocessor applications. *IEEE Micro*, 6(4):8–17, August 1986.

[Sal88] Lou Salkind. The SAGE operating system. In *Proc. of the Fifth Workshop of Real-Time Software and Operating Systems*, pages 54–58, May 1988.

[Sal89] Lou Salkind. The SAGE operating system. In *Proc. of 1989 IEEE International Conference on Robotics and Automation*, pages 860–865, IEEE Council on Robotics and Automation, IEEE Computer Society Press, May 1989.

[SBWT87] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee. High-performance operating system primitives for robotics and real-time control systems. *ACM Transactions on Computer Systems*, 5(3):189–231, August 1987.

[SGB87] Karsten Schwan, Prabha Gopinath, and Win Bo. CHAOS – kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, C-36(8):904–916, August 1987.

[SHKK89] Donald Schmitz, Regis Hoffman, Pradeep Khosla, and Takeo Kanade. CHIMERA: a real-time programming environment for manipulator control. In *Proc. of 1989 IEEE International Conference on Robotics and Automation*, pages 846–852, IEEE Council on Robotics and Automation, IEEE Computer Society Press, May 1989.

[SLR86] Lui Sha, John P. Lehoczky, and Ragunathan Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Proceedings of the Real-Time Systems Symposium*, pages 181–191, December 1986.

[Smi90] Johnathan Smith. March 1990. Private communication.

[SR87a] John A. Stankovic and Krithi Ramamritham. The design of the spring kernel. In *Proceedings of the Real-Time Systems Symposium*, pages 146–157, December 1987.

[SR87b] John A. Stankovic and Krithi Ramamritham. The design of the spring kernel. In *Proc. of the IEEE Fourth Workshop on Real-Time Operating Systems*, pages 19–23, July 1987.

[SR89] John A. Stankovic and Krithi Ramamritham. The Spring kernel: a new paradigm for real-time operating systems. *ACM OPERATING SYSTEMS REVIEW*, 23(3):54–71, July 1989.

[Sta88] John A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *IEEE Computer*, 21(10):10–19, October 1988.

[Ste88] Jeffrey A. Steinberg. VAXELN responds. *Digital Review*, 45–52, July 11 1988. Tech Report: Scientific & Engineering.

[Stu88] Martin C. Sturzenbecker. RMK: a real-time multripcoessing kernel. April 20 1988. Robot Systems Group, IBM Research Division, T.J. Watson Research Center.

[TH86] F. Tuynman and L. O. Hertzberger. A distributed real-time operating system. *Software – Practice and Experience*, 16(5):425–441, May 1986.

[TM89] Hideyuki Tokuda and Clifford W. Mercer. ARTS: a distributed real-time kernel. *ACM OPERATING SYSTEMS REVIEW*, 23(3):29–53, July 1989.

[VxW] VxWorks: a revolution in real-time. Marketing Brochure from Wind River Systems.

[WLG*78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. SIFT: design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(11):1240–1255, October 1978.

[XP88] Yangsheng Xu and Richard P. Paul. On position compensation and force control stability of a robot with a compliant wrist. In *Proc. of 1988 IEEE International Conference on Robotics and Automation*, pages 1173–1178, IEEE Council on Robotics and Automation, IEEE Computer Society Press, April 1988.