PHILIPS LABORATORIES - BRIARCLIFF
North American Philips Corporation

DOCUMENT NO. TN-89-142
DATE : November 9, 1989

MS-89-116

AUTHOR (S): Sandeep Mehta, Damian M. Lyons, Prabha Gopinath

TITLE : RS Execution Environment - Design Specification

KEYWORDS : RS, RSEE, distributed, real-time system, software, SPINE, C++, object-oriented, task planning, monitoring, execution, autonomous systems, robotics, scheduling

ABSTRACT

This is a working document that presents the current design specification of a distributed real-time software system called RSEE (*RS* Execution Enviornment) for Robot Task Planning and Execution for Autonomous Systems. The design is still evolving and therefore this document only represents a milestone in our design effort. It provides an overview of the system and a brief description of each component.

Implementation plans for different hardware platforms are described, but our initial implementation will be on SPINE. RSEE is being implemented using the object-oriented C++ language. The class definitions and stubs for important functions of RSEE are provided.

AUTHOR(S) SIGNATURES:

PHILIPS LABORATORIES - BRIARCLIFF
North American Philips Corporation

DISTRIBUTION LIST

DOCUMENT NO. TN-89-142

AUTHOR(S): Sandeep Mehta, Damian Lyons, Prabha Gopinath

TITLE: RS Execution Environment - Design Specification

ABSTRACT ONLY	COMPLETE DOCUMENT	
CONCERN RESEARCH BUREAU 10 Copies of Technical Reports Technical Notes 3 Copies of Manuscripts	*CONCERN RESEARCH BUREAU* 9 Copies of Technical Reports Technical Notes 3 Copies of Manuscripts	
CFT - L. van Dijl 1 Technical Report 1 Technical Note 1 Manuscript	*LEP - J. Bonnerot* All Technical Reports	
PL/B E. Arnold R. Benton R.N. Bhargava G.M. Blom A. Daniels C.A.A.J. Greebe F.J.A.M. Greidanus R.H. Kane E.W. Kent J. Ladell E. Lubchenko J.M. Mannarino H. Mauersberg R. McFarlane A. Miron S.N. Mukherjee A.J. Nijman M.M. Rochkind P.W. Shackle M. Shneier B.M. Singer F.W. Snyder E.H. Stupp A. Toth M. Tsinberg S.B. Ueland I. Wacyk J.L. Woo	*PL/B* Library Author(s) S. Mehta D. Lyons P. Gopinath O. Bakkalbasi P. Benjamin K. Bhaskaran F. Biemans A. Cameron L. Dorst R. Featherstone F. Guida A. Hendriks F. Holmquist I. Mandhyan M. Rosar D. Stimler K. Trovato T. Warmerdam R. Wendorf H-L. Wu H. Mauersberg R. Benton J. Wendorf C. Hill H-Y. Wang	

PROPRIETARY
CLASSIFICATION

AUGUST 1989

$\mathcal{RS}$ Execution Environment - Design Specification

Sandeep Mehta, Damian M. Lyons, Prabha Gopinath
Philips Laboratories
North American Philips Corporation
Briarcliff Manor, NY 10510

1 Introduction

This document is the first design specification of the $\mathcal{RS}$ Execution Environment (RSEE), a distributed real-time software system for Task Planning and Execution in Autonomous Systems, based on the $\mathcal{RS}$ model of computation. Our initial implementation will be on the SPINE multiprocessor system [2,3], but we also plan to implement this system on other hardware platforms, e.g., Encore MultiMax.

There are two design principles guiding this effort. Firstly, a policy and its underlying mechanism should be kept separate such that the mechanism is flexible enough to handle the policy and its variations. Secondly, we would like to keep the design simple which makes it efficient and easy to verify correctness.

Note that the specification is subject to change in future revisions as the RSEE design evolves.

The goal of the RSEE design effort is to build a multiprocessor $\mathcal{RS}$ system for real-time robot control. The following characteristics, in particular, are important:

- A separate run-time system and user-interface environment with a clean interface between them.
- Efficiency to meet the real-time requirements of an Autonomous System.
- Ability to distribute plans across the set of available processors.
- Ability to monitor plans from the user-interface environment.
- Flexibility to experiment with various policies in scheduling, real-time processing, process distribution etc.

The RSEE design will take advantage of the features of the target multiprocessor without overly depending on hardware, since that is liable to change.

We have broken up the RSEE design specification into 6 broad areas. These areas have been identified as the key elements of the real-time system. The RSEE user-interface provides a sophisticated front-end to the real-time system.

The components are:

- Process Definition
- Process Scheduling
- Communication (IPC)
- Memory management
- Plan Monitoring
- Process Distribution

The remainder of this document is structured as follows. The next section is an introduction to $\mathcal{RS}$, followed by a brief overview of the RSEE system. We then present a more detailed design specification for each of the 6 components in the real-time system listed above. This is followed by a description of the other hardware platforms on which we plan to implement RSEE. The C++ class and method definitions are listed in Appendix A.

2 Introduction to $\mathcal{RS}$

$\mathcal{RS}$ is a model of distributed computation, [4,7,8] describe the model. The structure of the lowest-level of description in the model is referred to as the *basic $\mathcal{RS}$ model* [4]. In a Turing machine model, for example, all the computation is 'funneled' through the read/write head of the machine. Thus, the model is inherently sequential. In a distributed model, computation can occur at a possibly infinite set of 'locations' usually called loci of computation. The mechanism that carries out the computation at each locus is usually called a computing agent (agent for short). Computation is performed in a distributed model by the *interaction* of a number of *concurrent* agents. We will use this concurrency to bring out the inherent parallelism within a robot task plan.

In order for computing agents to interact with each other, they need a communication method. All interaction in $\mathcal{RS}$ is expressed as *message passing* between connected agents. A connection between two (or more) agents sets up a message channel. Interaction occurs when agents read from or write to this channel. A set of connected agents is called a network.

The $\mathcal{RS}$ World View. The sensors and actuators of a robot system can be considered as a set of computing agents. The control program for the robot can also be considered as a network of computing agents. Furthermore, the environment containing the robot can be considered as a network of computing agents. This may be hard to visualize, since

humans do not normally think of the 'real world' in these terms (see Figure 1).

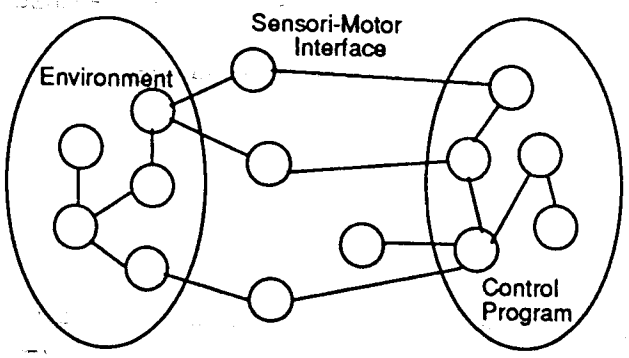

Figure 1: The $\mathcal{RS}$ World View

The behavior of these environmental agents is governed by the laws of physics. Particular networks of these agents represent particular environmental scenarios, i.e., objects interacting with each other in specific ways (supporting each other, sliding, moving in free space, etc.). The environment network and the control network are connected only through the set of sensor and actuator agents. Putting this another way, the control network and the environment network share the set of sensor and actuator agents. This is the 'world view' of the $\mathcal{RS}$ model.

Schemas and Schema Instances (SIs). A *schema* is a parameterizable description of a computing agent. A computing agent is created from a schema by the *instantiation* operation, and the term *schema instance;* SI, or process, denotes a computing agent. Instantiation creates an SI, sets some parameter values in the SI, and connects the SI to other SIs so that interaction can occur. A schema that directly represents an actuator is called a primitive motor schema. A schema that directly represents a sensor is called a primitive sensory schema.

Communication. In order to support the way in which the sensing and action in a robot task plan interact (or sensori-motor computation [4]) SIs must communicate with each other. Each SI has a set of communication objects called *ports*. At instantiation, connections can be specified between these ports and ports on other SIs. Communication occurs when an SI writes a value to one of its output ports, which has been connected at instantiation time to an input port on some SI, and the value is subsequently read by that SI from its input port. *Fan-in* (many-to-one) and *fan-out* (one-to-many) port connections are allowed.

Assemblage Schemas. The final piece of the basic $\mathcal{RS}$ model structure is a mechanism to group connected SIs together into a network in such a manner that the network appears to other SIs to be a single computing

agent. A network grouped together in this way is called an assemblage SI, and its definition is called an *assemblage schema* (also called a network). Some of the ports on individual SIs within the assemblage may appear as ports of the assemblage. The general rule is that if a port on a component SI has not been connected to anything, then it will appear as a port on the assemblage (perhaps under a new name). It is also convenient to be able to hide unconnected ports on occasion.

Task-Unit Schemas. The most important structure in $\mathcal{RS}$ is called the *task-unit*. This is a group of three types of computing agents that captures a characteristic. A task-unit schema is a 'structured' assemblage schema, a network consisting of a set of sensory schemas, a set of motor schemas and a set of task schemas (this is the 'control' program for the robot in the $\mathcal{RS}$ world view). This captures the robot controller structures as a sensory process pulling out important sensory data from the environment via the sensors, a motor process capable of controlling the appropriate robot actuators, and a task process that is using the sensory data to effect motions of the robot with the objective of achieving some goal.

While any set of connected computing agents is a 'program' in the $\mathcal{RS}$ model, task-units typify that sort of program usually called a robot *task plan*: a set of instructions for manipulating the world to achieve some goal. The sensory schemas defines the task-specific object model or world view, the motor schema defines a task-specific robot model, and the task schema defines how the task is implemented using the object/world and robot models.

The components of a task-unit can themselves be assemblages. This generates a sensori-motor hierarchy. Notice the two main improvements to the Albus [9] sensory motor hierarchy: Since the sensory-motor division is *internal* to each task description, the number of hierarchical levels in one task does not constrain the number in any other task. Secondly, each level of the hierarchy is a network: SIs at each level can communicate directly with their siblings.

Process Composition. A process composition operation is an operation on processes whose resultant is also a process. The name 'composition operation' is meant to indicate that the process arguments are composed together in some way to produce the resultant. Composition operations were developed in $\mathcal{RS}$ to explicitly represent the activities of the task schemas in a task-unit.

We have already described one form of process composition above, that is, the assemblage operation. The full set of $\mathcal{RS}$ composition operations and their algebraic properties are described in [8]. Some of the representational and analytical uses of the composition operations

are described in [6]. The following two composition operations are presented as examples; for more detail see [8].

Enabling or Precondition Composition. The precondition composition operation between SIs **P** and **Q** is written as:

$$T = P\langle v1,v2,...\rangle : Q_{v1,v2,...}$$

Informally, this means that the resultant SI **T** behaves like **P** until **P** terminates, it then behaves like **Q** with parameters chosen by **P** . This is called precondition composition because **P** enables the execution of **Q** .

Disabling or Transcondition Composition. The bidirectional transcondition operation between SIs **P** and **Q** is written:

$$T = P\#\#Q$$

The SI **T** behaves like the network **(P,Q)** until either **P** or **Q** dies, in which case **T** dies.

3 RSEE Overview

From the user's perspective the RSEE system appears as illustrated in Figure 2 below.

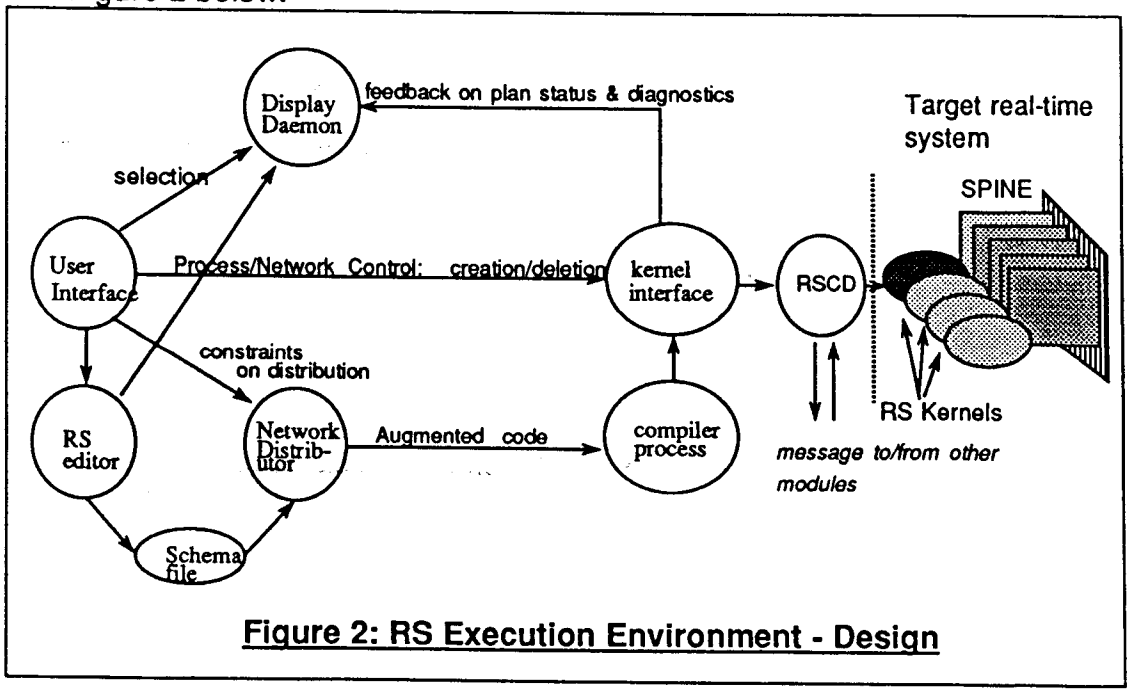

Figure 2: RS Execution Environment - Design

The modules shown in Figure 2 comprise the user-interface environment of RSEE and execute on the host machine. The six components listed in Section 1 comprise the part of RSEE which executes on the real-time system. The user-interface will be developed in the network transparent X Window System environment. The

processes in the interface environment communicate with the underlying real-time system via a single process: the $\mathcal{RS}$ communication daemon (RSCD). The RSCD also handles communication between the RSEE and other hosts which may be executing other modules in an Autonomous System such as Vision or Path Planning.

The RSEE user-interface is broken into the components based on functional separation. The "user-interface" shown in Figure 1 refers to the component that will interact with the user or an automatic planner that generates plans. If the user creates plans, the $\mathcal{RS}$ editor should be used. Initially, this will be a text editor (a major mode in GNU Emacs) but we plan to extend this to a graphical editor. Further research into graphical language representation of $\mathcal{RS}$ needs to be carried out. The editor (or planner) has the capability of generating files containing the plans (*schema files*). The Network Distributor can use the plans to distribute them over the processors in the real-time system, given some constraints on distribution by the user. The augmented code thus produced will be compiled into object modules executable in the real-time system. Initially the source code will be in the C language, however, we plan to implement a compiler for language that suitably represents $\mathcal{RS}$ (PlanRS). The RSEE interface communicates with the multiple kernels in the real-time system via a kernel interface. The main objective behind this is to provide a uniform interface to the real-time system. All graphical output from the RSEE user-interface as well as monitored and feedback data from the real-time system will be displayed via the Display Daemon. The user can interact with the Display Daemon to examine the data more closely, change viewing parameters, process data, observe animations of plans in execution etc.

The main means for RSEE to communicate with its real-time system and with the "outside world" is via the RSCD, illustrated in detail in Figure 3. It provides the capability to download executable plans, extract feedback data, alter control parameters (e.g., state of the kernels, state of networks), upload plan output and status to the host etc. The RSCD interacts with SPINE (our initial target system) via common memory accessible by host processes. With other processes in the UNIX environment it uses the *sockets* based IPC facility[10]. Access to remote hosts is available transparently via RSCD.

4 Process Definition

A process in the RSEE is an instance of a *schema* in execution. A *schema* comprises some local state, a code section, and a set of communication *ports*. All processes are *lightweight*. By *lightweight* we mean processes with the following characteristics:

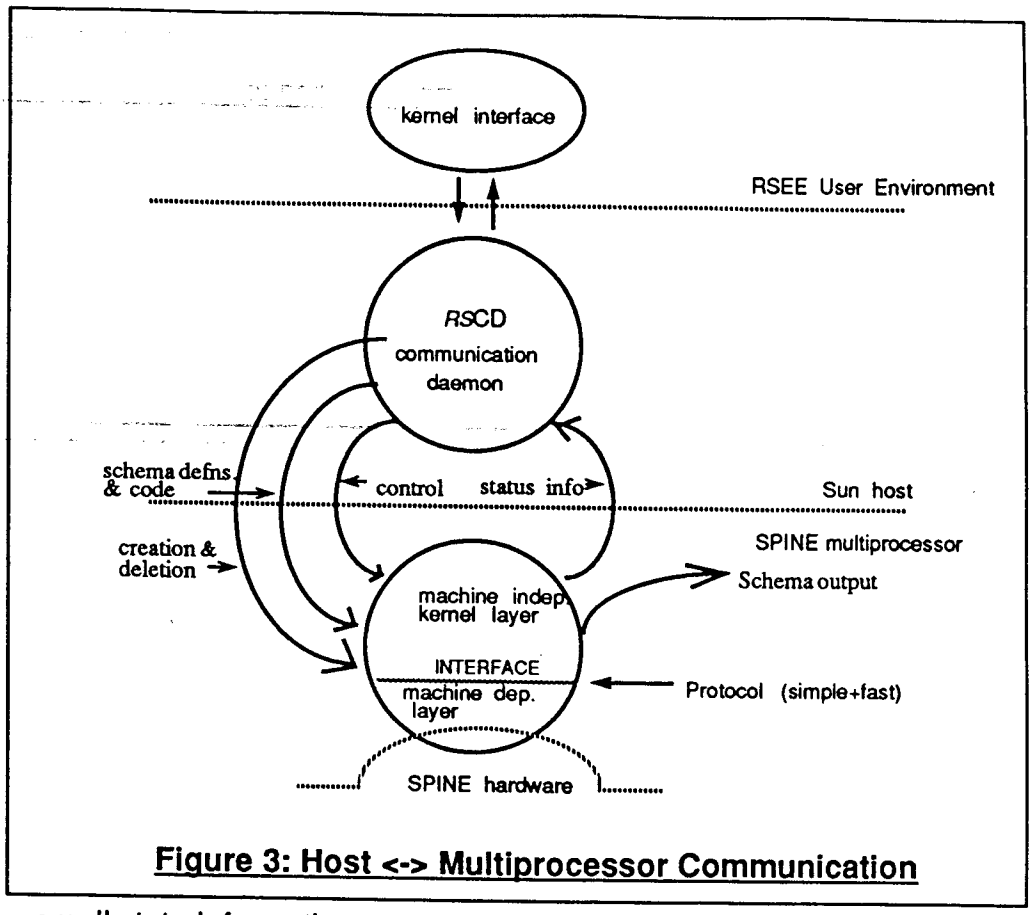

Figure 3: Host <-> Multiprocessor Communication

- small state information,
- quick context switching, and
- fast creation and destruction.

A context switch involves saving port information in the kernel (see section 4 on Communication), saving state information (Program Counter (PC) and registers) since we are using preemptive scheduling (see section 3 on Scheduling), and dequeuing and enqueuing by the kernel.

Because the RSEE system (and its processes) is executing in real-time, timing information needs to be associated with each process. Timing information can be added to schema definitions as default timing attributes. The timing attributes of a schema instance can be modified at run time, e.g., by the scheduler or by a parent process.

There is a set of *atomic* or primitive schemas from which the atomic process can originate. All other processes are networks of processes some of which may be atomic and some of which may in turn be networks. Networks, by nature of their composition, are heavier than the lightweight processes.

Process networks will have *a name scope* defined. The name scope of a network implies that all processes in that network are visible to processes only in that network. Since, networks could comprise of processes which are themselves networks there is a restriction. A process in a network can not see the component of a sub-network. The visibility works much like the inheritance mechanism in the object-oriented programming paradigm (see Appendix A for the definition of a process object and its methods). We expect to use a hierarchical naming scheme much like those used in UNIX file systems.

There are separate process control blocks defined for schema instances and for networks, called Schema Instance Control Block (SICB) and Network Control Block (NCB) respectively. Each schema and network are referenced by a Schema Descriptor (SD) and Network Descriptor (ND) respectively which are different from their instances (described by the control blocks). Figure 4 illustrates the structure of the descriptors and control blocks. The equivalence map in a Network Descriptor contains the mapping of ports on a schema in a network onto ports on a network as seen outside the network. These ports are equivalent and hence the name equivalence map.

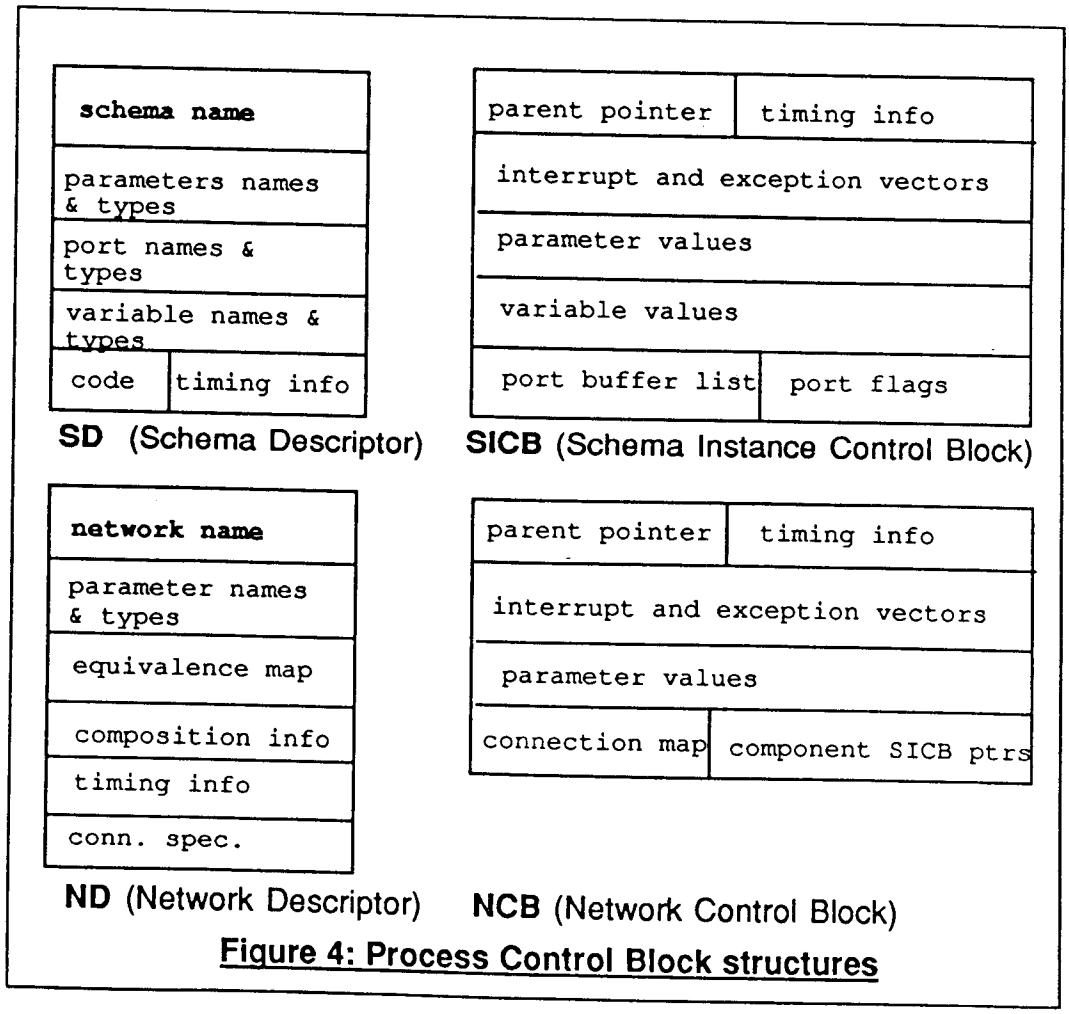

Figure 4: Process Control Block structures

Kernel operations will largely be on atomic processes. All operations defined on atomic processes are also defined on networks. However, networks will have special operations defined on them. For instance, *killing* a network could be either killing a *key* process or termination of the network when all processes, which form the network, die. Network *reduction* (where a sub-network is broken into its component processes) is a unique *kernel* operation.

It is our desire to apply the object-oriented programming paradigm [12] to the definition of processes and networks in RSEE. Thus, a network could be a two part object. In the base case, a network is an atomic (lightweight) process object. Otherwise, it is comprised of processes, process composition information and other data. Note that the regular semantics of object-oriented programming, i.e., inheritance, construction, destruction, derived classes etc., still apply to these objects.

5 Process Scheduling

A flexible scheduling mechanism is essential to RSEE because we want to experiment with a number of scheduling schemes. An example of a scenario in the robot programming domain will illustrate the scheduling needs. Consider a conveyor carrying parts which are part of an assembly. There exists some sensing device(s) (e.g., camera(s)) which can detect arrival of parts. A robot is expected to pick up the parts appropriate for the assembly which it plans to carry out. If the part is inappropriate or if the robot is otherwise occupied then the part moves beyond the reach of the robot. Thus, there is a time (or distance) window during which the robot has to complete it's pick operation. Figure 5 illustrates this example.

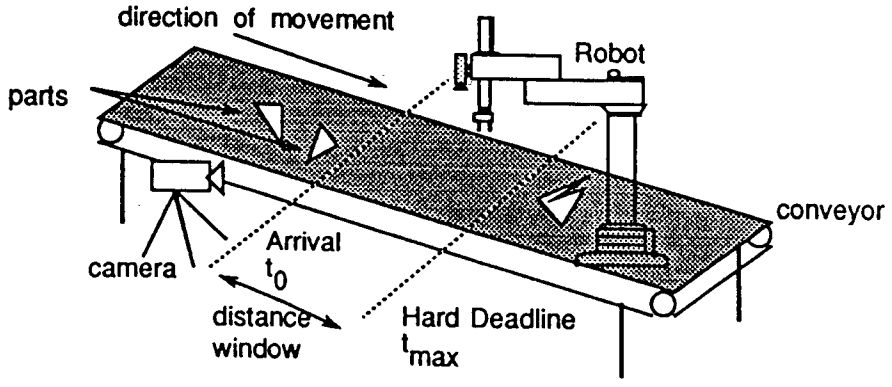

Figure 5: Conveyor belt example

In Figure 5, the time window has a soft estimated deadline (t_0) after

which the part is available to the robot. The second deadline (t_{max})is hard because after it's expiration the part will not be available.

All processes have 3 fundamental properties associated with them: *deadlines*, *priorities* and a *recovery* mechanism. From these fundamental properties we can derive two higher levels of classification: namely *urgency* and *periodicity*. Both derived properties reflect a combination of deadlines, priorities and recovery mechanisms, e.g., non-critical processes will not be allowed hard deadlines and cannot have more than a certain priority level. Of course, these parameters are adjustable.

Deadlines are the time constraints of an application during which an action has to be accomplished. They can be of two types: *hard* or *soft*. Hard deadlines are those that have to be met and their expiration implies incorrect behavior of the system. Soft deadlines can be adjusted or allowed to expire while some remedial action is taken.

One of the following schemes could be adopted for most processes:

- fixed deadlines with adjusted *bin*s, i.e., *bin compaction* can be carried out over a set of processes,
- single hard deadlines with a reduction in priority, as a function of the task being performed, after expiration of the deadline.

In a bin-based deadline scheduling mechanism [11] the kernel maintains a bin that represents total available processing time on a processor. The attributes of the bin are *StartTime*, *Deadline* and *AvailableTime*. At system start-up the *StartTime* of the bin is set to infinity. As each scheduling request arrives, a new bin is created with its *StartTime* set to the current time, its *Deadline* set to the deadline of the request, and its *AvailableTime* attribute set to:

$$AvailableTime = (Deadline - StartTime) - ExecutionTime(op)$$

ExecutionTime is the expected execution overhead of the process. When a new scheduling request arrives on the processor, its deadline is compared to the *Deadline* attributes of existing bins. If a match exists and if the *AvailableTIme* in the bin is sufficient, then the request is accepted and marked as such. The *AvailableTime* is updated to reflect the cost of the new operation.

Priority implies an order of importance of processes according to which they must execute. There need to be a range of **priorities** (e.g., 0 to 9, where 0 is lowest priority). Prior analysis of plans which generate the process networks can result in an even distribution of processes with a certain priority over the set of processors. Methods for plan analysis that yield a hierarchy of priorities have not been explored yet.

A heuristic will be used to implement a scheduler with both deadline and priority schemes, which are orthogonal in nature.

At present we recognize only a simple **recovery** (forward and backward) mechanism. One could recover from failed deadlines, faulty devices or sensors, processor failure etc. Deadline recovery can be addressed separately for periodic and aperiodic processes.

Processes have the following categories of **urgency**:

- Emergency - where immediate action is necessary,
- Critical - high priority action with hard deadline,
- Non-critical - action with a soft deadline and lower priority.

Periodic processes are those which execute repeatedly. Thus, periodicity P of a process is defined as $P = (\nu,\tau)$ where a process executes for a certain (possibly fixed) *duration* (or period) τ at a certain *frequency* ν. The frequency is user-tunable. The periodic property can be implemented using deadlines. A process can be marked as periodic and given a deadline which is equal to the required duration (ν). On expiration of the deadline, it is reset to its initial value and the process is rescheduled. Obviously, a termination condition for a periodic process has to be embedded in the process' code section.

The scheduling mechanism will consist of multiple priority queues, with a dispatcher. The scheduler will run at the system clock rate. In addition to the multiple priority run queues there will be two other queues: a Ready To Run (RTR) queue and a blocked queue for processes which are waiting for the arrival of an event. RSEE scheduling is pre-emptive. Therefore, the kernel has to do a state-save for the process being pre-empted. The cost of context-switching a network process is the sum of costs of switching all the schema instances which comprise the network, plus the cost of switching the network process itself.

6 Communication

Three types of communication are required in RSEE. These are all well supported by the target systems discussed in Section 10. The types of communication needed are:

- Intraprocessor - process to process on single processor; using local memory (via local kernel)
- Interprocessor - process to process across processors, e.g., using common memory or local dual-ported memory on the SPINE multiprocessor.
- Processor -> Host - process to process across hosts via RSCD.

The communicating processes cannot tell which type of communication they are using, i.e., transparency is maintained.

The $\mathcal{RS}$ kernel will maintain process communication tables required for interprocess communication. In any network the network process maintains the connection map for communication ports in that network (see section 2). Thus, when two processes need to communicate, the entire path is expanded from the connection map and communication table information, which are available from the network and kernel under which the network is executing. Figure 6, below, illustrates the mechanism.

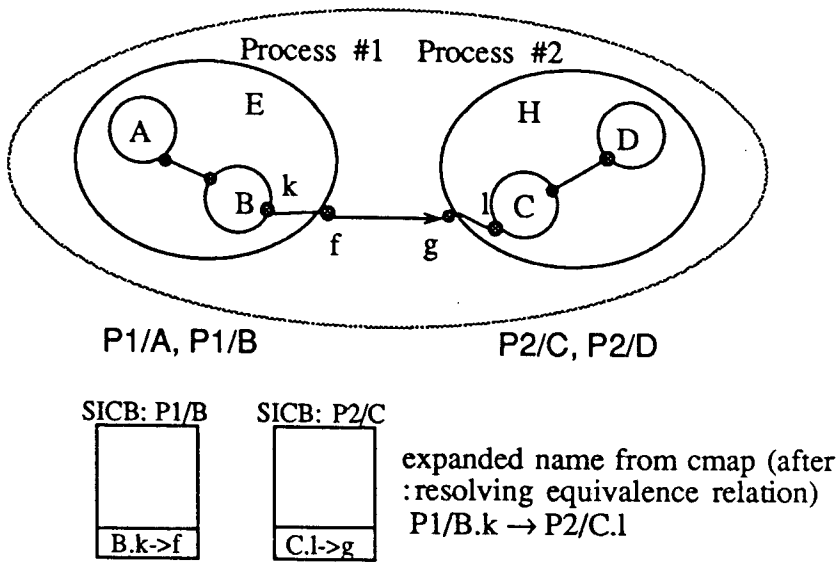

Figure 6: Port Communication example

The reads/writes are synchronous, singly buffered transmits and receives. The semantics of the operations should support the following notions of:

- overwriting (the buffer) - to allow updating of an old unread value,
- sticky reads (persistence) - to avoid having to store data internally,
- non-blocking read and write - for asynchronous communication,
- fan-in and fan-out on the read/write ports - to allow *many-to-one* and *one-to-many* communication.

All buffer management and port name mapping is done by the kernel(s).

An evaluation of necessary components of our initial multiprocessor system, SPINE, has already been completed [1]. That provides us with data on which we can base our choice of implementation.

7 Memory Management

The SPINE multiprocessor, our initial target hardware, provides both local and common memory management. The current version of the hardware provides local memory which is dual-ported.

The on-board MMU provides transparent access to common and local (dual-ported) memory. While accessing remote processors' dual ported memory, the memory has to be mapped into the local process' address space before it is accessible.

There is a parallel list of functions available for both common and local memory. No memory management needs to be done by the RSEE kernel or other processes.

8 Plan Monitoring

Plan monitoring provides the ability to monitor the status of individual networks and processes. From the kernel level, it requires that the information be made available to the monitoring data analysis and display module executing on the host. This module can construct plan status from the monitoring information it receives. Monitoring data can be transferred to this module from the kernel via the RSCD. In addition, it is desirable to put checkpointing information along with the timing information in the control block for a process or network. This checkpointing information can be used to augment the monitoring data on the process.

Based on our experience, we demand the following monitoring information from the processes:

- Process creation and deletion for all processes or for selected processes.
- Network communication activity for all networks or for selected networks.
- Port activity on individual ports or sets of ports; activity includes throughput rate as well as values passing through the port.
- Deadline information, i.e., proposed deadline schedule per node, a graph of the number of SI's executing per node, and a graph of load per node. Load for a node has to be specifically defined.

Two possible ways of implementing monitoring are the following:

1. To have a *lightweight* daemon process executing under the $\mathcal{RS}$ kernel. This process is periodic, i.e., it expires after some interval and transfers data and status information from local to common memory. The RSCD process picks up this data & information at a

suitable time and passes it onto appropriate processes in the interface environment. This way the arrival of the data and information does not directly affect the execution behavior of each individual kernel. In addition, the monitoring process is also subject to timing constraints, and thus can be timed not to conflict with other processes.

2. Each process (on the host side) and each daemon (executing under the $\mathcal{RS}$ kernel) sends a message to the kernel which raises a flag (or turns a bit on). The raised flag results in the kernel transferring data and status information into common memory. The RSCD picks up this information as described above.

The advantage of the former approach is that external stimuli need not be provided to obtain status information. Its disadvantages are that the expiration period may not sufficiently reflect the current needs of the user, i.e., it may provide data more (or less) often than required for monitoring purposes. Secondly, the daemon process when active will compete with the process networks which are executing a plan.

Using the latter approach has the disadvantages of having to rely on a message being sent to the kernel (or a daemon process), that could affect performance or itself be delayed. Also, if the kernel has to do the transferring then it is stealing time from the processes.

In both cases, some time will be stolen from the processes. The ability to fine-tune the monitoring capability is important. It is not necessary that the data returned to the RSCD process be the latest, i.e., plans can continue to execute assuming that the data will eventually reach the appropriate process, leaving the onus on the latter.

9 Process Distribution

The distribution of networks and processes over the set of physical processors will be governed by factors such as the following:

- The need to execute on a processor that can directly *control* a device such as sensors or robots,
- *Spread* the total number of processes, in a nested hierarchical network, equally over the available pool of processors.
- The *cost* of communicating over the bus or other similar transactions.

One mechanism to achieve process distribution over the set of processors is to associate a processor list with each schema process in a network. This means that a subset of the available processor pool is made available to each process in a network or to the entire network. Consider the following example (in $\mathcal{RS}$ notation [4]):

$(locate_{m1}^{[P1]} \langle r \rangle : place_r^{[P1,P2]}, locate_{m2}^{[P2]} \langle r \rangle : (place_r^{[P1,P2]}, fred^{[P2]}))$

The superscripts [Pi] indicate the subset of the processor pool associated with each action (process). Thus the execution profiles of the two processors might look like:

P1	P2
$locate_{m1}$	$locate_{m2}$
$place_r$	$locate_{m2}$
idle	$locate_{m2}$
$place_r$	fred

This results in an efficient utilization of the available processor pool. Details of this mechanism need to be worked out.

10 Implementation

10.1 SPINE Implementation

SPINE (Structured Processor INteraction Environment) is a multi-processor system with real-time capability [2]. In the current implementation it consists of a number of single board computers based on the MC68020 microprocessor. The boards reside on the VME bus and the SPINE station is connected to it's host, a Sun 3/160 workstation, via a VME to VME bus adapter. The SPINE station also has 8 Mbytes of common memory, on the SPINE VME bus.

SPINE software is comprised of a run-time library of functions and commands executable on the host [3]. The library functions provide block/character I/O, memory management, semaphores, alarms, timers, etc..

We have identified major components of SPINE that are necessary for the implementation of the RSEE kernel and evaluated their performance [1]. The components were as follows:

- SPINE Pipes support RS port communication,
- Schema downloading can be done using SPINE common memory,
- Processor to host communication can be efficiently implemented via Pipes or the SPINE mailbox facility.
- Monitoring and feedback to the RSEE user-interface on the host can also be done via common memory.

Evaluation of SPINE indicated that common memory transactions are susceptible to bus load, thus, appropriate use of common should be made. Pipes were found to be less susceptible to bus load for small (around 512 bytes or less) message sizes.

Each board will run a copy of the RSEE kernel. The kernel does all process and communication management as well as providing feedback data to the host. The RSEE user-interface executes on a Sun workstation. Theoretically, it could execute on any Sun on the network, because we are implementing it using the network-transparent X Window System. Communication between the user-interface and real-time system is done via the RSCD.

10.2 Encore Multimax Implementation

We plan to port the kernel to a four processor Encore Multimax. The Multimax runs under the Mach operating system from Carnegie Mellon University [5]. Fully BSD UNIX compatible, Mach provides much improved performance and features such as light-weight process capability in the form of the *cthreads* library. At present we cannot estimate the real-time performance of Mach on the Multimax, however, the RSEE port to that system will provide us with the a good testbed for different scheduling algorithms and monitoring mechanisms. The interface part of RSEE would still execute on a Sun using the network transparent X window system.

10.3 Sun Implementation

For purposes of debugging and simulation the RSEE kernel will also be implemented entirely in the Sun environment (SunOS). The *cthreads* library is available under SunOS and will be used. In addition, this port will also serve as a comparison in performance between the various implementation.

11 Conclusion

A first specification of the key components of the RSEE system has been described in this report. In Appendix A we present our progress to date in the C++ implementation of ideas presented in this document. Future reports will completely define each element of RSEE in adherence with our earlier stated design goals.

References

1. Sandeep Mehta, A Performance Evaluation of SPINE User Processes, *Philips Technical Note* TN 89-073, June, 1989.
2. T. Warmerdam, T. J. Harosia, SPINE (Structured Processor INteraction Environment) Description, *Philips Technical Note* TN 87-153, Nov, 1987.
3. T. Warmerdam, The SPINE User's Guide, *Philips Technical Report*, TN 88-014, Mar 1988.
4. Damian M. Lyons and Michael A. Arbib, A Formal Model of Computation for Sensory-Based Robotics, IEEE Transactions on Robotics and Automation, Vol 5(3), June 1989, pp 280-293.
5. M. Accetta, R. Baron, D. Golub, R. Rashid et.al., Mach: A New Kernel Foundation for UNIX Development, USENIX Technical Conference, 1986.
6. D. M. Lyons, A Process-based Approach to Task-Plan Representation, Submitted to the IEEE International Conference on Robotics and Automation, Cincinnati, OH, May 1990.
7. D. M. Lyons, Fundamentals of $\mathcal{RS}$ - Part I: The Basic Model, *Philips Technical Report* TR 89-031, June 1989.
8. D. M. Lyons, I. Mandhyan, Fundamentals of $\mathcal{RS}$ - Part II: Process Composition, *Philips Technical Report* TR 89-033, June 1989.
9. J. Albus, H. McCain, and R. Lumnia, NASA/NBS Standard reference Model for Telerobot Control System Architecture (NASREM). NBS Technical Report TN-1235, July 1987.
10. Sandeep Mehta, An Evaluation of Interprocess Communication for Distributed Applications under UNIX, *Philips Technical Note* TN 89-134.
11. Prabha Gopinath, Programming and Execution of Object-Based, Parallel, Hard Real-Time Applications. Ph.D. Thesis, The Ohio State University, June 1988.
12. Bjarne Stroustrup, What is Object-Oriented Programming ?, IEEE Software, May 1988, pp 10-20.

Appendix A: C++ Class and Method Definitions for RSEE

The main areas of development are listed below, with their components. Our implementation is being done in C++. For each of the following areas we specify C++ class declarations, which are attached.

Module	Class	Methods for
Process Mgmt	Class Process: BaseObject	Process Definition Process Creation Process Deletion Process Composition
Scheduler	Class Scheduler: BaseObject	Deadlines Priorities Periodicity Recovery Bin allocation Bin management
Communication	Class Process: BaseObject	Portmap mgmt Equivalence map mgmt Message Send & Recv (Kernel) Buffer mgmt Fan-in, Fan-out support
Monitoring	Class Monitor: BaseObject	Create/Destroy Sensor Operate Sensor Read, Transmit Sensor

In addition additional class declarations are needed for the kernel, e.g.,

1. Kernel - Host monitoring data protocol + message operations
2. User Interface - kernel interactive schema composition protocol.

These will be developed later after the basic methods have been implemented.

RSEE.h

```
//
//      FILE :          RSEE.h
//      AUTHOR :        Sandeep Mehta, MSR, Philips Labs, Briarcliff Manor, NY
//      DATE:           Thu Sep  7 17:56:45 EDT 1989
//      PURPOSE:        definitions for main classes in RSEE real-time system.
//
//
//      Modification History:
//              - Thu Sep  7 17:56:45 EDT 1989 - created - sxm
//              - Mon Oct 16 12:57:39 EDT 1989 - added Scheduler and Monitoring
//              stubs from Prabha - sxm
//              - Tue Oct 17 15:06:32 EDT 1989 - added structure definitions to
//              class Process, for SD, SICB, ND, and NCB.

#ifndef lint
#ifndef SABER
static char RSEE_h_sccsinfo[] = " %W% %G%";
#endif  SABER
#endif  lint

typedef enum                                    // Base class types - tentative
{
        SCHEDULER = 1, TASK = 2, TIMER = 3, MONITOR = 4, QUEUE = 5
        } OBJECT_TYPE;
typedef enum                                    // Task state
{
        IDLE = 1, RUNNING = 2, KILLED = 4
        } TASK_STATE;


// base class
class Base_Object
{
public:
        OBJECT_TYPE obj_type;           // SCHEDULER, TASK, TIMER, MONITOR, QUEUE
        void print(int);                        // for diags/debugging

};

class Scheduler: public Base_Object
{
        friend timer;
        friend Process;
        friend Base_Object;

 public:
        void print(int);

        // The function that creates the pool of schedule bins and
        // initializes them.
        int CreateBinPool():
        // The function that returns an expended bin to the pool  The bin
        // is pushed onto the stack of available bins.
```

```
        int ReturnBinToPool(SchedBinType *BinPtr):
        // The function that gets a bin from the pool if one is available.
        SchedBinType *GetBinFromPool();
        // The function that assigns an allocated bin to a process that has
        // to be scheduled.
        int AssignBinToProcess(SchedBinType *BinPtr, PCB *PCBPtr);
        // The function that checks if a bin can fit into an existing
        // schedule. A convenience function.
        int CheckSchedule(SchedBinType *BinPtr);
        // The function that schedules a process into an existing schedule
        // on a processor and then updates the timing attributes of the
        // schedule.
        int ScheduleProcess(SchedBinType *BinPtr);
        // The function that compacts an existing schedule of processes to
        // minimize fragmentation.
        int CompactSchedule();

private:
        // Define the maximum number of processes that can be maintained in
        // the scheduler at one time.
        int MaxScheduledProcesses = 50;
        // Define the strcture of a scheduling bin. Sequences of such bins
        // will constitute the schedule of a processor node. The PCBPtr
        // field points to the PCB of the process represented by the bin.
        // Chaining through this pointer gives access to other process
        // attributes such as its priority, code pointer etc.

        typedef struct
        {
                PCB                     *PCBPtr;
                long unsigned int Deadline;
                long unsigned int    EarliestStartTime;
                long unsigned int    ExcutionTime;
                struct SchedBinType *PrevBin;
                struct SchedBinType *NextBin;
        } SchedBinType;

        // Define the pool of bins that are statically allocated and
        // maintained. This ensures that the cost of obtaining a bin
        // dynamically is lower than if "malloc" and "free" had  to be
        // invoked. The pool is organized as a stack of pointers to
        // available bins.

        typedef struct
        {
                int                     PoolLock;
                int                     StackSize;
                SchedBinType            *AvailBins[MaxScheduledProcesses];
                int                     Top;
        } PoolType;

        // Define the node schedule. This is maintained as a link list of
        // bins that correspond to individual processes.
```

```
        typedef struct
        {
                int                 ScheduleLock;
                int                 ScheduleSize;
                long unsigned int   EarliestStartTime;
                long unsigned int   LatestDeadline;
                long unsigned int   TotalExecTime;
                long unsigned int   AvailExecTime;
                SchedBinType        *FirstBin;
                SchedBinType        *LastBin;
        } ProcessorScheduleType;
}

struct Timer : Scheduler
{
        timer (int);
        ~timer();
        void reset(int);
        void print(int);
}

class Process : public Base_Object
{
public:
        // construction/destruction  is equivalent to process
        // creation/deletion. the actions should be embedded in the
        // appropriate functions.

        // Constructor function (unintialized name & pid)
        Process (char * =0, int =0);
        // Destructor function ()
        ~Process ();
        TASK_STATE state;                       // IDLE, RUNNING, KILLED
        // GENERAL MEMBER FUNCTIONS
        void cancel(int);
        void delay(int);                // delay by int (time) value
        void preempt();
        void restore();                 // swap in new process
        void save();                    // save process state
        void sleep();
        void wait();
        // ATOMIC COMPOSITION OPERATOR FUNCTIONS
        void assemblage (char *namelist, int *conn_map, int
                                        *port_equiv_map);
        void precondition (char *argvec, char *schema_name);
        void sequential (char *schema_name);
        void transcondition (char *schema_name);
        // NON-ATOMIC COMPOSITION OPERATOR FUNCTIONS
        void bi_transcondition (char *schema_name);
        void async_recur_precondition (char *argvec, char *schema_name);
        void sync_recur_precondition (char *argvec, char *schema_name);
```

```
private:
        // PCB data structures comprise of Schema Descriptor (SD), Schema
        // Instance Control Block (SICB), Network Descriptor (ND), and
        // Network Control Block (NCB). Network refers to an RS network.
        // See section on Process Defintion in RSEE Design Spec.

        typedef struct
        {
                // embedded timing info details
        } Timinginfo;
        typedef struct
        {
                char *Schema_name;
                char **Param_vector;   // Parameter name-type pair list
                char **Port_vector;    // Port name-type pair list
                char **Var_list;               // Variable name-type pair list
                char *codeptr;                 // Pointer to code section
                Timinginfo Schema_timing;
        } SD;

        typedef struct
        {
                char *si_parent;
                Timinginfo si_timing;   // Timing info structure, not defined yet
                char *intr_vector;      // Interrupt and exception vectors
                char *except_vector;
                char *Param_values;     // Parameter and variable values
                char *Var_values;
                PortTable *si_portbufs; // Port buffer table not defined yet
        } SICB;

        typedef struct
        {
                char *Network_name;
                char **Param_vector;   // Parameter name-type pair list
                EquivMap *net_eq_map; // Equivalence map structure, not defined
                char *comp;                    // composition info, details not defined
                Timinginfo Network_timing;
                Connmap *Network_connmap;   // Connection specification, not defined
        } ND;

        typedef struct
        {
                char *nd_parent;
                char *intr_vector;      // Interrupt and exception vectors
                char *except_vector;
                char *Param_values;     // Parameter and variable values
                Connmap *Network_connmap;   // Connection specification, not defined
                char ** sicb_ptr_list;         // list of pointers to component SICBs
        } NCB;
};
```

RSEE.h

```
class Monitor : public Base_Object
{
 public:
        // constructor
        Monitor();
        // destructor
        ~Monitor();

                                                                          220
        int CreateSensor(int NodeId, char SensorType, char * PtrToVar, int
                                                InitialStatus);
        int SensorOn(int SensorId);
        int SensorOff(int SensorId);
        int ReadSensor(int SensorId);

 private:
        int MaxSensedValues = 100;
        int NodesInSystem;

                                                                          230
        // Define an entry of the Monitor Table. The SensorId is obtained
        // by concatenating the processor id with an integer representing
        // the sensor index for that processor node. The SensorType field
        // is used to identify whether it is a Short, Integer, Long,
        // Double, Float, or Char.

        typedef struct
        {
                int     SensorId;
                char    SensorType; /*S,I,L,D,F,C*/                       240
                int     SensorStatus; /*Off or On*/
                long unsigned TimeOfUpdate;
                union {
                        char    CharVal;
                        short   ShortVal;
                        int     IntVal;
                        long    LongVal;
                        double DoubleVal;
                        float   FloatVal;
                } Value;                                                  250
        } MonitorEntry;

        // Define the Monitor Table.
        typedef struct
        {
                int             MonitorTableLock;
                MonitorEntry    Array[NodesInSystem][MaxSensedValues];
        } MonitorTable;

                                                                          260
class Queue : public Base_Object
{
public:
        // constructor function w/ dynamic allocation
        Queue ();
```

```
        // destructor function frees entire queue
        ~Queue ();
        // GENERAL MEMBER FUNCTIONS

        Queue *get (Queue *);           // get from head                    270
        BOOLEAN put (Queue *);          // put on tail
        BOOLEAN insert (int, Queue *); // insert entry after some index point
        BOOLEAN del (int);                      // delete entry at some index
        void print (Queue *);           // print entire queue
        BOOLEAN is_empty();             // is empty ?

        Queue *find(int);                       // overloaded find
        Queue *fint(char *);

        Queue *q_next;                          // next item in queue       280
private:
        // a Queue is a pointer to a dynamically allocated null terminated
        // list of Queue items.
};

// Old RS RTE command/message structure, maintain backward compatible
// for now.
typedef enum
{
        COMMAND = 0x0, MESSAGE = 0x1                                        290
        } COMMAND_TYPE;

typedef enum
{
        INSTANTIATE = 0x2,
        DEINSTANTIATE = 0x4,
        EXITSIM = 0x6,
        INITSIM = 0x8,
} MSG_OP;
                                                                            300
class Message : public Base_Object
{
 public:
        // constructor & destructor functions
        Message();
        ~Message();
        void SendMsg();                         // asynchronous Send
        void RecvMsg(BOOLEAN asyncmode); // synchronous Receive, can be async

 private:                                                                   310
        struct Msg
        {
                short MessageType;    // OR of COMMAND_TYPE & MSG_OP, least
                                                        // significant bit indicates TYPE
                char *Dest_SI;
                char *Src_SI;
                union
                {
```

```
                    // some union of common data sent included in a message
                }                                                              320
}

//local Variables:
//mode: c++
//End:
```