AN ABSTRACT OF THE THESIS OF

Sheida Saedi-Faez for the degree of Master of Science in
Computer Science presented on May 27, 1982.

Title: SES: A Real Time Message Oriented System for Easy
Intertask Communication

Abstract approved:

Redacted for Privacy

ABSTRACT

In process control systems where an entire process is being controlled by a computer, the component tasks of the system have to be monitored. An efficient message oriented environment could be an important factor in the optimization of such systems. In this paper we discuss the design of a real time message oriented system, where intertask communication is used for the synchronization and cooperation of the component tasks. Exchanging messages between tasks is one way to achieve such communication. A semaphore is a mechanism to coordinate exclusive access to shared system resources. In this design, semaphores are treated as special forms of messages. In a message oriented system, these
characteristics should be provided to guarantee an efficient and safe environment. The SES system is designed to provide such an environment for the processes that run on the Real Time Task Supervisor version 1.1 of August Systems series 300 computers, which guarantees a reliable and safe computer for process control applications.
SES: A Real Time Message Oriented System for Easy Intertask Communication

by

Sheida Saedi-Faez

A THESIS

submitted to

Oregon State University

in partial fulfillment of the requirements for the degree of Master of Science

Completed May 27, 1982
Commencement June 1983
This thesis is affectionately dedicated to my mother and father, Lili and Reza Saedi-Faez, to whom I owe my happiness.

I would like to express my appreciation to:

- My advisor Dr. Earl F. Ecklund
- My dear sister Shohreh Saedi-Faez
- My two good friends James B. and Lesley Ann Crossland
- August Systems Inc.
# Table of Contents

Chapter 1 INTRODUCTION 1

Chapter 2 RTTS 6

  2.1 USER INTERFACE 8
  2.2 ERROR HANDLING BY RTTS 9
  2.3 FAULT TOLERANT SERVICES 12
  2.4 FAULT TOLERANT RELATED SYSTEM CALLS 18
    2.4.1 DISABLING INTERRUPTS 19
    2.4.2 GET WORDS 20
    2.4.3 QUALIFY VOTE 21
    2.4.4 SYNCHRONIZATION 22
  2.5 TASK RELATED SYSTEM CALLS 24
    2.5.1 CHANGING A TASK'S PRIORITY DYNAMICALLY 25
    2.5.2 REQUESTING A TASK TO RUN 26
    2.5.3 DECREMENTING A TASK'S REQUEST COUNT 26
    2.5.4 ABORTING A TASK 27
    2.5.5 RELATIVE SCHEDULING 27
    2.5.6 TASK REMOVAL FROM THE SCHEDULING QUEUE 28
    2.5.7 EXAMINING THE PENDING REQUESTS 28
  2.6 EVENTS 29
    2.6.1 LINKING A TASK TO AN EVENT 30
    2.6.2 UNLINKING TASK FROM AN EVENT 31
    2.6.3 MASKING AN EVENT 32
    2.6.4 UNMASKING AN EVENT 32
    2.6.5 EXAMINING THE STATE OF AN EVENT 32
    2.6.6 RESETTIMG AN EVENT 33
    2.6.7 CAUSING AN EVENT 33
  2.7 DISPATCHER 36
  2.8 SCHEDULER 36

Chapter 3 SES 39

  3.1 FUNCTIONAL SPECIFICATIONS 39
    3.1.1 DEFINITION OF EXCHANGES 41
    3.1.2 REDEFINITION OF EXCHANGES 42
    3.1.3 MESSAGE LENGTH 44
    3.1.4 MESSAGE IDENTIFICATION 44
    3.1.5 MESSAGE TYPE 45
    3.1.6 RESPONDING TO A MESSAGE 46
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>REAL TIME TASK SUPERVISOR COMPONENTS</td>
<td>11</td>
</tr>
<tr>
<td>2.</td>
<td>BLOCK DIAGRAM OF AUGUST SYSTEMS' SERIES 300 COMPUTER SYSTEM</td>
<td>13</td>
</tr>
<tr>
<td>3.</td>
<td>REAL TIME CLOCK HARDWARE VOTER</td>
<td>15</td>
</tr>
<tr>
<td>4.</td>
<td>ASYNCHRONOUS INTERRUPTS AND INTERRUPT DISABLING IN A TRIPLE REDUNDANT SYSTEM</td>
<td>17</td>
</tr>
<tr>
<td>5.</td>
<td>TASKS STATE TRANSITION VIA SYSTEM CALLS</td>
<td>35</td>
</tr>
<tr>
<td>6.</td>
<td>A RELATIVE SCHEDULING EVENT</td>
<td>38</td>
</tr>
</tbody>
</table>
This paper describes a facility for intertask communication in a real time process control system. We assume that this system is used in a single user application composed of cooperating tasks. This fact implies that there is no malicious task, and that all tasks behave according to the designed communication protocol to guarantee the integrity of the system. Previous work of a similar nature has been done. Two such operating systems are Thoth[5] and iRMX86 [7].

Thoth is an operating system for a single user application which runs on Data General NOVA and Texas Instruments 990 minicomputers. Thoth provides a set of primitives to allow intertask communication via messages (Send, Receive, Reply, and Forward). A unique characteristic of Thoth is that sending a message automatically blocks the sender task until the receiver task responds to the message. We view this as a
deficiency in throughput of the system for the following two reasons:

1. Not every message in a system require a response. In fact many messages are mainly in the form of a signal. Requiring a response for every message, causes a slow down in the overall speed.

2. Even for a message requiring a response, the sending task may wish to choose an appropriate time to receive the response. The response may not be vital to the sending task's execution at the time the message is sent.

Similarly iRMX86 is designed for a real time environment with similar applications. This operating system is a product of Intel Corporation. iRMX86 also provides system calls for intertask communication via messages (RQ SEND MESSAGE, RQ RECEIVE MESSAGE, RQ DELETE MAILBOX, RQ CREATE MAILBOX), and semaphores (RQ SEND UNITS, RQ RECEIVE UNITS, RQ DELETE SEMAPHORE, RQ CREATE SEMAPHORE). Unlike Thoth, iRMX86 does not require a response for every message. The problem with iRMX86 is that all the messages are treated on a FIFO basis with respect to a mailbox and there is no message identification. In a preemptive real time system, where there is a large number of small tasks running, the flow
of control may become unpredictable based on external events. This may lead to incorrect delivery of messages. Therefore, the user must be extremely careful about the order in which the messages are delivered.

SES (SpeakEasy System) is designed to solve the above problems and also to increase the overall speed by providing extra facilities for the user. These include:

1. Making the need for a response optional. Any message can demand a response be sent back. Also, the sender of a message can request the response at any time. This allows the sending task to continue execution, until a response from the receiving task is necessary for the sending task to proceed. Therefore, more work can get done by both tasks, resulting in an increase in the speed and throughput.

2. Processing messages on a FIFO basis with respect to tasks, and not to the mailboxes, due to the fact that the flow of control within tasks is more predictable than between tasks. At the other extreme, processing messages on a message ID basis with respect to two tasks would add extra overhead with little gain.

3. Allowing broadcast messages to be sent, which
results in a great amount of processing time and memory resource being saved, since the message is copied only once as opposed to once for every recipient task.

4. Using the same data structure to represent mailboxes and semaphores. The ability to allocate and deallocate these entities, and the fact that the same data structure can be used as both a semaphore and a mailbox saves memory space, with no extra overhead.

SES runs on the Real Time Task Supervisor (RTTS) operating system version 1.1 of the August System Series 300 (AS300) computers. The Series 300 is a fault tolerant computer system designed for process control applications where a computer fault and system shut down can be very costly. SES is a subsystem of RTTS.

This paper is divided into 4 sections.

- Section one describes the RTTS operating system, since knowledge about RTTS is a necessary element for understanding and using SES. There are system calls and data structures embedded in RTTS that are used by SES. This section also contains a short description of the hardware scheme of the AS300 computers.
- Section two contains the functional specification of SES. It describes the system calls to perform synchronization, sending and receiving of the messages, and the utility routines to support this message passing environment.

- Section three contains a summary of the significant contributions of SES and a comparison to Thoth and iRMX86 operating systems.

- Section four contains the appendices: Appendix A contains a list of RTTS system calls. Appendix B contains SES system calls. Appendix C contains the SES commented source code listing.
Chapter 2

RTTS

RTTS is fault tolerant software that uses fault tolerant hardware features to accomplish maximum reliability and safety of the system for a process control environment.

The unique combination of hardware and software allows the AS300 to reach a maximum degree of reliability. Through software control, the advantages of fault tolerance can be utilized when and where necessary. RTTS handles all scheduling, dispatching, and intermodule coordination including synchronization, voting, process I/O, and interrupt handling. RTTS schedules and dispatches tasks on a priority basis for regular periodic tasks, tasks that are invoked upon occurrence of an external or internal event, and tasks that are invoked by request from other tasks.

RTTS is the first known 'SIFT' (Software Implemented Fault Tolerant) system implemented. In all, SIFT applies three basic techniques to achieve fault tolerance.
1. REDUNDANCY: Replicated hardware and software is the basic essence of this design (i.e. the computer itself actually consists of three identical microcomputers, namely Intel 8086 CPUs). In the event of a failure in one module, the replicated system will continue to operate correctly with no interruption of the computer function.

2. VOTING: Each module within the system performs critical functions simultaneously, and compares its result to the results of other units. In a properly functioning system, all three values agree. However, if one module produces an error, the two properly functioning units will outvote it.

3. ISOLATION: Even though the replicated system elements communicate with one another, they are effectively isolated from one another to prevent an error or failure in one module from degrading other modules in the system.

See section 2.3 (FAULT TOLERANT SERVICES).
2.1 USER INTERFACE

User tasks can be written in any language capable of producing Intel 8086 executable code. The user can specify the priority, method for invocation, and schedule time for execution of each task. The user interface to RTTS is provided by a system START UP module. This is a procedure written by the user in which he can specify system control parameters (e.g., the number of tasks in the system, the number of events in the system, etc). These control parameters determine the size of the system tables.

Tasks could be scheduled to run at a specific time relative to the current value of the real time clock, and/or in response to hardware interrupts, a request from another task, software interrupts, etc. The user is responsible for creation of the START UP procedure and his tasks. Once they are provided, they are linked to the operating system and located.

The located object code is loaded into one of the computers from which the other two can crossload the object code.
2.2 ERROR HANDLING BY RTTS

RTTS contains extensive facilities for error detection. These errors fall into two categories:

1. Those caused by improper use of system calls.

2. Those presumed to have been caused by faults.

The first category of errors can happen when a system call is invoked with invalid parameters. The system then informs the user about the error via a status return code. It is the user's responsibility to ensure the correct operation of the system by checking the status codes returned for every system call.

The second category of errors includes those detected during voting or during synchronization. For example, in the case of a real time clock interrupt, the event is hardware voted by all three computers to make sure the external event is recognized by all three processors at the same time. The second step of voting includes voting on the value of the real time clock to ensure the correct operation of the system. For example, if two out of three processors have a ten for their real time clock and the third processor has a nine for its real
time clock, they may in fact synchronize on the real time clock interrupt; but two processors may schedule a task that is waiting to be executed on the tenth clock tick while the third processor is still waiting for another clock tick to schedule the task.

In the case where a faulty processor has a different value for its clock, it will be outvoted and take the consensus value (i.e. automatic error correction by a faulty processor).

Figure 1 shows the essential elements comprising RTTS. Each element is described in a later section in detail.
REAL TIME TASK SUPERVISOR COMPONENTS

Figure 1
2.3 FAULT TOLERANT SERVICES

Figure 2 gives a basic sketch of the AS300 hardware arrangement. The system consists of three separate Control Computer Modules (CCMs). Each CCM is a complete computer system with a processor board (Intel 8086 CPU board), a control board, and a communicator board [1]. Each computer system has its own local memory which typically is 64K bytes. The communicator board consists of two communicators that talk to the other two CCMs. These computer modules are isolated from each other so that a failure in one module cannot possibly affect the function of the other two computers. Since the only means of communication for the processors is by reading data from the other computers, a CCM does not have the capability of writing to the other CCMs' memory or interrupting other CCMs. This 'read only' capability is used for voting purposes and/or for synchronization between processors. The concept of voting and synchronization are described later in this section.
BLOCK DIAGRAM OF AUGUST SYSTEMS' SERIES 300 COMPUTER SYSTEM

Figure 2
Triple redundancy has helped the software function more efficiently. For example, all the interrupt lines from the outside world (I/O) or from inside (the real time clock) are passed through a hardware voter as indicated by Figure 3. The input to this very simple circuitry is three lines, one from each of three interrupt generators (e.g., from three real time clocks in each system). These lines are voted and the voter output is the consensus of the interrupt lines. Therefore, if two CCMs generate an interrupt while the third one does not, the hardware voter guarantees that all three CCMs will receive interrupts at the same time. In the triple redundancy approach to the design of fault tolerant systems, the processors' outputs are compared in the voting circuit so that a faulty processor cannot influence the actual output determined by the majority.
FIGURE 3

REAL TIME CLOCK VOTER

REAL TIME CLOCK

CCM 1

REAL TIME CLOCK

CCM 2

REAL TIME CLOCK

CCM 3

TO INTERRUPT
Even though the interrupts are received by the three processors at the same time, all three may not be able to service the interrupt at the same time. A processor may have turned off its interrupts slightly before the other two processors, because of a small difference in speed of execution. But, when two processors are interrupted while the other one has a pending interrupt, the proper functioning of the system will not be affected. Through the software protocol of voting on the source of the interrupt, all three CCMs are forced to perform the majority action. Figure 4 depicts an example of this event.

It should be mentioned at this point that all the features of this fault tolerant system permit the system to survive and function properly under a single fault. The system is not able to survive and will not tolerate two faults. For example, if one processor receives a spurious interrupt (high), while the other two processors have no interrupts (low), and if a hardware voter in any one of the processors has a line that is faulty (always high), this will cause an interrupt (high) to be sent to that processor.
Figure 4

ASYNCHRONOUS INTERRUPTS AND INTERRUPT DISABLING IN A TRIPLE REDUNDANT SYSTEM
2.4 FAULT TOLERANT RELATED SYSTEM CALLS

What makes RTTS a special operating system is the ability to function properly in the presence of a single fault [2]. This ability is a result of close cooperation between hardware devices and software programs. This special characteristic requires certain conditions to be met. For example, one of the most important capabilities of the AS300 is that one CCM can read from any other CCMs' memory while both are running, and can vote on certain data items that are critical to their performance (e.g., to vote on the value of the data to be output before the actual transmission).

The AS300 is a tightly synchronized system, meaning that the three CCMs are closely synchronized in terms of their execution paths. The computers are synchronized at every interrupt (hardware or software interrupt). The process of synchronization involves a sophisticated algorithm in which the processors wait for each other to get to a certain point in their program. The waiting period is limited and is a function of several variables.

The following is a description of some of the fault tolerant system calls provided by RTTS.
2.4.1 DISABLING INTERRUPTS

Being able to disable interrupts simultaneously in three computers is a necessary requirement for the system. Due to the slight difference in the speed of each CPU, a 'DISABLE' command may be executed at different times by each of the three computers. The system call 'DISABLE INTERRUPTS' provides a means to disable interrupts simultaneously in the three CCMs.

Figure 4 demonstrates the need for this system call. Assume that CCM 3 runs slightly faster than the other two processors. At time t1 CCM 3 has disabled its interrupts while the other two are running with interrupts enabled. At time t3 the other two processors disable their interrupts. If an interrupt happens at time t2, CCM 3 will not be able to see it because its interrupts have been turned off. CCM 1 and CCM 2 will go to service the interrupt while CCM 3 is still executing its program. Hardware voting the interrupts has no effect here, because the hardware voted interrupt will be pending at the processors at the same time. The realization of the interrupt by individual processors is left for software to handle. CCM 3 which was faster and did not receive the interrupt will be out of synchronization with the other two processors shortly. Therefore, the need for a simultaneous disabling of the interrupts is obvious. This system call makes the faster processor wait for the slower
ones so that all three can execute the command at the same time.

2.4.2 GET WORDS

This system call allows the user to retrieve data from all three processors and manipulate that data. For example, the user may have some critical variables where the correctness of the process is entirely dependent on the correctness of those variables, and the user may wish to check and verify the values periodically or before making any decision based on those variables. Also, there are other situations which are application dependent in which the user may want to evaluate the data from all the processors.

This system call receives the address of the variable as an incoming parameter. It also receives two memory addresses in which the results retrieved from the other processors are placed. The computers are connected to each other through two ports at the back panel called port0 and port2. The connection method is completely arbitrary. The two memory locations received as the parameters correspond to the value from the computer connected to the port0 and port2 respectively. If the computer could not fetch the value from one of the other computers because it was not able to access the other computer's bus, the value returned is a default value FFFF
(hexadecimal) and a flag is set in an internal register on the communicator board.

2.4.3 QUALIFY VOTE

This system call allows the user to vote on the value of a particular variable. The 'QUALIFY' concept implies that this voting is performed on the value in the CCMs that are in synchronization with each other. This system call assumes a previous synchronization was performed immediately prior to its invocation for the consensus to be valid.

This system call votes on the value and returns to the user the consensus value among the synchronized processors and also a return code indicating:

- If there was an agreement between the processor and the processor connected to port0.
- If there was an agreement between the processor and the processor connected to port2.
- If there was an agreement between the processor connected to port0 and the processor connected to port2. Note that this situation implies that the local processor was in minority.
- If there was complete disagreement between all processors.
- If there was complete agreement between all processors.

2.4.4 SYNCHRONIZATION

In all systems that use multiple units, it is necessary that one unit which accesses data from another unit does so when the other unit is not changing the data. This is accomplished by synchronizing all CCMs before accessing any data. Then, each CCM can access the data in the other two units. In the basic procedure, each CCM is assumed to be at a similar point in its execution flow. By passing through the synchronization routine they exit at the same time.

Two important concepts must be addressed here.

1. SYNCHRONIZATION POINT: During program execution, synchronization is needed at several different places. In this scheme, the address of the calling instruction combined with other pieces of information is used as the synchronization point to guarantee the uniqueness of the 'synch point'. Thus each invocation of this procedure from a different
part of the program or at a different time has a unique synchronization point.

2. TIME LIMIT: The time limit represents the maximum time each CCM can wait for the other processors to see if they are synchronizing. In particular, this is definable in terms of the number of times each CCM looks through its window to see the other units. If the limit is reached, then the CCM that 'times out' is faulty and that synchronization is not possible.

The process of synchronization is as follows, a processor tries to look at the other processors through its window to see if they are trying to synchronize (that is, whether they are in the synchronization routine or they are executing a program). If it can see both processors trying to synchronize with identical synch points, then it exits the routine with a total synchronization declaration.

If it sees only one of the processors, it tries to see the other one until the time out occurs or the other processor shows up. If the third processor never shows up, it leaves with partial synchronization.

If it sees neither of the processors in the time period allocated, it leaves with a total nonsynchronization report. This case indicates that the
processor trying to synchronize is the faulty processor (e.g., this could be because it is executing a different path in the program, or is too fast or too slow compared to the other two processors). This synchronization report is used by the qualify voter to restrict the vote to valid data from synchronized processors.

2.5 TASK RELATED SYSTEM CALLS

Tasks are the main objects in RTTS. There are two types of tasks. System tasks and user tasks.

System tasks are tasks that are a resident part of the operating system. They are created by the system and perform RTTS functions. These tasks run at a high priority. Other than this feature, they are treated the same as the user tasks.

User tasks are tasks that are written by the user. A task consists of two parts: the actual code and the Task Control Block. Task Control Blocks are created by RTTS at system initialization time. They contain all the necessary information about the tasks needed for the system to schedule and dispatch the tasks. User tasks run at lower priorities than system tasks and they are
interruptable. Each user task is allocated a Task Control Block by a system call invoked by the user in the system START UP routine. The tasks are statically allocated and there is no dynamic creation or destruction of any tasks. Once the task is created, it always stays resident in the system. It can stay dormant, but the Task Control Block is permanently resident. The tasks are compiled separately and are linked and located together with the RTTS code. The operating system provides an extensive number of system calls by which the user can manipulate his tasks in a variety of ways.

2.5.1 CHANGING A TASK'S PRIORITY DYNAMICALLY

This is done by the system call 'CHANGE TASK PRIORITY'. Any task can change any other task's priority, as long as it is within the allowed range of priorities. The system tasks are protected against user interference because the user does not have access to the system tasks' Task Control Blocks. This dynamic change may cause preemption of the task that performed the system call, provided that the new task's priority is higher than the currently running task's priority. The currently running task is interrupted and is placed in the ready queue waiting for the processor. As soon as the interrupted task is the highest priority task in the system, it resumes execution. A task may change its own priority,
but the effect is recognized, after the current execution of the task.

2.5.2 REQUESTING A TASK TO RUN

This is done via the 'REQUEST TASK TO RUN' system call. Any task may request any other task to run. The user cannot request system tasks to run because the user does not have access to the system tasks' Task Control Blocks. This system call causes the requested task to be placed in the ready queue. If the task is already in the ready queue its number of pending requests is incremented by one. (Refer to the 'EXAMINE REQUESTS' system call.)

This may cause a preemption of the currently running task if the new task runs at a higher priority than the task that performed the system call. In this case, the new task is scheduled for running immediately and runs as long as it has more requests pending for its execution.

2.5.3 DECREMENTING A TASK'S REQUEST COUNT

A task may cause a decrement in the number of pending requests for execution of a task. This action does not cause a task to be removed from the ready queue. If the number of pending requests for the task is one, the 'ABORT TASK' system call should be used to remove a task
from the ready queue. The user cannot interfere with the system tasks, since he does not have access to the system tasks' Task Control Blocks.

2.5.4 ABORTING A TASK

A task may abort itself or another task. When a task is aborted, it is removed from the ready queue regardless of the number of pending requests for its execution. If the aborted task resides in a semaphore queue or a timer queue, it is removed from that queue. The Task Control Block remains in the system and the task may be requested to run again if desired.

2.5.5 RELATIVE SCHEDULING

A task may be requested to be executed at a later time relative to the current time. The timing is based on the value of the system's real time clock. The time interval between clock ticks can range from 1ms to 120,000 seconds. The user can specify the running time of the task in terms of the number of clock ticks from the current time. This is a double word (32 bits) and is passed via two 16-bit words. At every clock tick the scheduling queue is checked for any potential scheduling. After the number of clock ticks has expired, the task becomes ready and is moved to the ready queue from the scheduling queue. Preemption may occur if the new ready
task runs at a higher priority than the currently running task. Control is passed to the new task, and the interrupted task runs when it is the highest ready task in the system. The relative scheduling queue maintains the tasks that are to be scheduled for running at some later time. The tasks in the queue are ordered according to time, unlike the ready queue which is ordered by priority. If there are two or more tasks with the same time requested, they will be inserted in the ready queue in a priority based manner so that the higher priority task is executed first, even if it was requested after the second task.

2.5.6 TASK REMOVAL FROM THE SCHEDULING QUEUE

Removal of a task, from the scheduling queue, may be requested by the system call 'REMOVE RELATIVE TASK'. This may be done for rescheduling purposes.

2.5.7 EXAMINING THE PENDING REQUESTS

The number of pending requests of a task may be examined at any time by the system call 'EXAMINE REQUESTS'. The number of pending requests of the specified task is returned to the caller.

The state of the system can be changed by these system calls. A task that is ready may become dormant by
'ABORT TASK'. A dormant task may become ready by 'REQUEST TASK TO RUN'. Preemption occurs if the dormant task has higher priority.

2.6 EVENTS

Events are secondary objects in the RTTS operating system. They can be either user defined events or system events. System events are defined by RTTS and are used to control the internal functioning of the system. Examples are the interrupts or the events that happen when an exception occurs. User defined events can be anything the user wishes to treat as an event (e.g., the high temperature in any part of a process control system). The user can specify the action to be taken upon occurrence of these events. There is a relationship between events and interrupt processing in terms of function. When an interrupt happens, the running task is interrupted. After the context of the interrupted task is saved, depending on the interrupt level, a particular task is executed. When the interrupt service routine terminates control is returned to the interrupted task. The events function similarly as follows:

The user defines the events in terms of the
exceptions that can occur in his system. Then he defines a number of tasks which he wants to be executed upon occurrence of these events. Lastly, the tasks are linked directly to particular events. After this step the tasks are executed whenever the event occurs.

RTTS has provided seven event related system calls. Together with task related system calls, they give the user the capability to use the system in a way best suited to his particular application.

The interrupt service routine is executed on the occurrence of an interrupt. Similarly, the linked task to a particular event is executed upon occurrence of the event. The main difference between the two schemes is that the vectoring of the processor to the particular task in case of an interrupt, is hardwired. Unlike interrupts, transfer of control for an event to the particular task is done by software (i.e. the user checks for the exceptional case and causes the event).

The number of events is definable by the user. This is one of the system control parameters defined by the user at system initialization time. The user has the ability to dynamically link tasks to and unlink tasks from the events. In the same way that interrupts can be masked or unmasked, the user has the ability to mask or unmask events with system calls provided by RTTS.
2.6.1 LINKING A TASK TO AN EVENT

The linkage between tasks and events can be done dynamically by use of the system call 'LINK TASK TO EVENT'. A task can be linked to more than one event at the same time. For example, the user may wish to run the same diagnostic task in more than one case. However, an event can only be linked to one task at a time. If the user attempts to link a task to an event already linked to a task, the action will not take place and an error code is returned to the user. The user specifies the task to be linked, and the event number as incoming parameters to the system call.

2.6.2 UNLINKING TASK FROM AN EVENT

This action is done by the system call 'UNLINK FROM EVENT'. This system call causes the linkage between the event and the task to be broken. Unless another task is linked to that event, no action will take place when that event happens. If there is no task linked to that event, an error message will be returned to the user. The above two system calls allows the user to have different tasks linked to an event at different times.
2.6.3 MASKING AN EVENT

The user can mask a particular event dynamically with the system call 'MASK EVENT'. This system call causes the event to be disabled. The linked task (if any) will not be activated upon occurrence of that event. If an event happens and it is masked, the occurrence of that event is latched for only one time -- a set/reset bit is used for this purpose. The user can then inspect the event to find out its state while it was masked. If the event happens more than one time, there is no change in the state of the event. The user will be informed only about one occurrence of the event.

At system start up, all the events are initialized to a masked state. The user must unmask every event individually before he invokes the linked tasks.

2.6.4 UNMASKING AN EVENT

Unmasking of an event causes the event to be enabled. If the event happened while masked, the reenabling will not cause an automatic activation of the task linked to the event. Instead, the user must explicitly query the state of the event and request the task to be run, or cause the event.
2.6.5 EXAMINING THE STATE OF AN EVENT

The state of an event can be examined with the system call 'EXAMINE EVENT STATE'. This system call returns to the user the state of the event as either happened or reset. This system call is meaningful if it is called after the process of masking and unmasking the event.

2.6.6 RESETTING AN EVENT

An event can be reset from the 'happened' state to a 'reset' state. This system call clears the latch that was set when the masked event happened. This latch is a binary state latch and only indicates whether or not the event occurred, not the number of times it occurred.

2.6.7 CAUSING AN EVENT

Last, but not least, of the event related system calls is 'CAUSE EVENT'. This system call causes the event to 'happen'. That is, if there is any task linked to this event, it is activated. There are several possibilities that can occur when this system call is issued.

- The event is not linked to any task. There is no action to take place. The system call returns a status code indicating this fact.
- The event is linked, but it is masked. In this case there is no activation of any task, but this occurrence is latched on the set/reset bit of the event.

- The event is linked and it is unmasked. In this case, the causing of this event activates the linked task.

If the task is already in the ready list, the number of its pending requests is incremented by one. If the task is not in the ready list, it is moved to the ready list. If the task linked to that event runs at a higher priority (all the tasks that are linked to system events run at higher priority than the user tasks), preemption occurs. The task that caused the event is interrupted. The newly activated task is placed in the ready queue, and the previously running task is moved down the queue. If the event service routine runs at lower priority, no preemption occurs and the task that caused the event continues execution.

Figure 5 indicates the transitions in tasks states via the system calls.
1. ABORT TASK (RTTS)
2. REQUEST TASK TO RUN (RTTS)
2. CAUSE EVENT (RTTS)
3. CHANGE TASK PRIORITY (RTTS) DEFINITE POTENTIAL
4. RECEIVE MESSAGE (SES)
4. WAIT (SES)
4. WAIT ON BUFFER (SES)
5. SEND MESSAGE (SES)
5. SIGNAL (SES)
5. SIGNAL BUFFER (SES)

TASKS STATE TRANSITION VIA SYSTEMS CALLS

Figure 5
2.7 DISPATCHER

The dispatcher is responsible for transforming the state of a task from READY to RUNNING, by allocating the processor to the task. The dispatcher, upon activation, allocates the CPU to the first task in the ready queue. This is done by loading the proper machine registers (stack segment registers, data segment registers and program counter registers). It is guaranteed that the task at the top of the queue is always the highest priority task in the system eligible for execution.

To start execution of a new task, the program counter is loaded with the starting address of the task. If the task is restarting after an interruption, the program counter is loaded with the address of the next instruction from where it was interrupted.

2.8 SCHEDULER

The Scheduler is responsible for scheduling the tasks for execution at some later time, definable by the number of clock ticks of the real time clock of the system. The number of clock ticks, after which the user
wishes his task to be executed, is a two word value (32 bits). Together, with the wide range of the real time clock period, they give the user substantial flexibility in terms of relative scheduling.

The exact time of execution is recorded in the Task Control Block by the system call 'REQUEST RELATIVE TASK'. After the time has expired, the task is moved from the scheduler queue to the ready queue. Unlike the ready queue, which is ordered according to the priority of the tasks, the scheduler queue is ordered according to the relative requested time of the invocation of the tasks. The first task being the first one to be invoked. At every clock tick of the real time clock, the scheduler checks the queue for a possible task invocation. If there is a task in the scheduler queue, whose time has come for execution, it is moved to the ready queue for execution. This transfer from the scheduler queue to the ready queue may cause a preemption of the currently running task. In this case the new task begins execution.

Figure 6 shows a relative scheduling event.
<table>
<thead>
<tr>
<th>READY QUEUE</th>
<th>SCHEDULER QUEUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIORITY TIME</td>
<td>PRIORITY TIME</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>200</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

A RELATIVE SCHEDULING EVENT

Figure 6
3.1 FUNCTIONAL SPECIFICATIONS

The main design goal of SES is to support easy inter-task communication for a real time environment. The main object used in SES is an 'exchange' which could be either a mailbox or a semaphore.

A mailbox is an entity to which the tasks send messages, and from which other tasks receive messages. A semaphore is an entity to protect shared resources (soft or hard) between tasks from simultaneous access.

An exchange can be defined to be a mailbox at one time and redefined to be a semaphore at another time. SES treats both mailboxes and semaphores alike. SES provides system primitives for tasks to send messages and receive messages to/from a mailbox and to synchronize through semaphores primitives.

To send a message, the task provides information about the message to be sent, such as the destination
mailbox, the length of the message, etc. To receive a message, the task specifies the mailbox from which the message is expected, the task from which the message is to be received, etc. A task can also broadcast a message, which means the message is to be delivered to more than one task. The message stays in the mailbox until the last recipient task has received it. To send or receive a message, the task must be eligible with regard to the type of the mailbox. See section 3.1.1 (DEFINITION OF EXCHANGES).

SES is used in a real time environment on a real time operating system which has had impact on its functional and design aspects as follows:

1. The system is a real time system and has a single controller. It is not a multi-user system. There is one address space for RTTS that is shared by all tasks. Therefore, there is no need to copy messages from user buffer to system buffer and vice versa. There must be provisions, however, for signaling the senders when the user message buffer is free and can be reused by the sender.

2. Since RTTS is a fault tolerant operating system, the capability for voted messages is necessary. In a real time system, the speed of execution of system
calls is crucial. By making the 'QUALIFY RECORD VOTE' system call available to the user, this necessity becomes a user option. Therefore, rather than the system voting on all the messages, the user can vote on the messages that are critical to the process under control.

3. Unlike most conventional message-oriented systems, the messages in SES are not treated on a FIFO basis with regard to a mailbox. Rather, messages are identified with a sender-receiver code. This scheme still uses a FIFO method with regard to two communicating parties, but the probability for receipt of an improper message is reduced drastically. To receive a message, the recipient tasks must specify from which task the message is expected.

3.1.1 DEFINITION OF EXCHANGES

The user can allocate a new mailbox by using the 'CREATE MAILBOX' system call. The attributes of the mailbox are specified by the user through parameters. The user can define the mailbox to be either private, semi-private or public. In all three types of mailboxes, only the owner (i.e. the creator task) is privileged to destroy a mailbox.
- In the case of a public mailbox, there is no privileged function dedicated to the owner of a mailbox, except the privilege of destroying a mailbox. Any task can send a message and any task can receive a message to/from a public mailbox.

- In the case of a private mailbox, the owner specifies another task to be co-owner of the mailbox which defines a two way communication line between the creator and the co-owner. Both tasks can receive and send messages to/from a private mailbox.

- In the case of a semi-private mailbox, the owner is the only task that can receive messages from the mailbox. Any task can send a message to a semi-private mailbox.

Since there is no dynamic memory allocation in RTTS, the CREATE MAILBOX system call does not dynamically allocate memory for the mailboxes, instead the memory area for the mailboxes (and semaphores) is statically allocated at system initialization. This system call redefines an exchange.

For semaphore definition see section 3.1.11 (SEMAPHORE PRIMITIVES).
3.1.2 REDEFINITION OF EXCHANGES

Every exchange has an owner. The owner of an exchange is the task that created or defined the exchange. The owner is the only task that can destroy the exchange.

Destroying an exchange requires the exchange to be empty. In the case of a mailbox, this means that there is no message pending in the mailbox to be received and there is no task waiting for receipt of a message. In the case of a semaphore, it means there is no task waiting for semaphore units. But this restriction could make this system call useless, because the owner may not be able to access the exchange at a time when there is no task suspended on the exchange (and when there is no message pending in the case of a mailbox). On the other hand, destroying an exchange by blindly awakening all the suspended tasks, and discarding all the pending messages in the case of a mailbox, may not be suitable. Therefore in SES, destroying an exchange requires an empty exchange, but its owner can awaken the suspended tasks and/or discard the messages on a mailbox. To awaken the suspended tasks on an exchange the task can call 'PURGE MAILBOX' or 'PURGE SEMAPHORE', with parameters specifying the kind of action to be performed. In PURGE MAILBOX the task can specify to discard messages and let all the currently suspended tasks receive their messages, it can
specify to awaken the suspended tasks but allow all the pending messages to be delivered, or it can specify that both the tasks and messages be removed from a mailbox. Similar logic applies to a semaphore. The suspended tasks are removed from the exchange with a signal sent to them, indicating that the exchange has been deleted.

3.1.3 MESSAGE LENGTH

A message in SES is not restricted in length. It can be of any size. The size of the message should be provided by the sending task to fill in the message header. Since the messages are not copied into system buffers, this piece of information is passed directly to the receiving task. Only the pointer to the actual message is kept in the system message header.

3.1.4 MESSAGE IDENTIFICATION

The messages are identified and are passed between tasks only via a valid ID. The ID of a message is the concatenation of the sender task ID and the recipient task ID.

To send a message to a mailbox, the sender task specifies the recipient task. The system ensures that the message is delivered to the proper task. In case of a broadcast, the same logic applies except the sender
specifies a number of tasks as the valid recipients for that message.

To request a message from a mailbox, the task specifies which task the message is expected from. The mailbox is checked for a pending message from the specified task.

Therefore, two way checking is necessary for a transfer of any message to/from a task.

3.1.5 MESSAGE TYPE

A message in SES could be of two types:

- A single message -- Where the message is delivered to a single task. After the receipt of the message by the receiving task, the message is removed from the mailbox.

- A broadcast message -- Where the message is delivered to more than one task in the system. The message remains in the mailbox until the last task has received the message. A broadcast message is delivered only to authorized tasks which are specified by the sender of the broadcast message.
3.1.6 RESPONDING TO A MESSAGE

A task can specify whether or not a particular message requires a response. This information is passed directly to the receiving task. It is the receiving task's responsibility to respond to the message. The sender is not automatically blocked for the response. The task could continue execution and request the response when desired, because the response message may not always be crucial for the continuation of execution of the sending task.

3.1.7 RECEIVE MESSAGE

A task can request to receive a message from a public mailbox, from a semi-private mailbox of which it is the owner, or from a private mailbox which it is one of the two communicating parties.

A task provides the following information in order to receive a message:

- The mailbox from which the message is expected.

- A pointer to a user message header to which the information about the message is copied from the system message header. The system message header is
returned to the pool if the message is a 'single' message or if the message is a broadcast message and the receiving task is the last authorized task to receive the message.

- A wait period, indicating the amount of time the task is willing to wait for arrival of that message, if not already in the mailbox.

- A task ID, which identifies the task from which the message is expected.

- A pointer to a word through which the status of the system call is returned.

If there is a message pending in the mailbox with the correct sender-receiver code, it will be delivered. Otherwise, the task could wait for a period of time for arrival of the message, if desired. A task is preempted if there is no message available and it wishes to wait. After control is returned to the task, one of the following conditions may have occurred. This occurrence is indicated by the system call return code:

- The task has timed out waiting for a message.

- The task received the message within the wait time period.
- The mailbox has been deleted while the task was waiting for a message.

3.1.8 SEND MESSAGE

A task can send a message to a public or semi-private mailbox. To send a message to a private mailbox the task must be one of the two communicating parties.

A task provides the following information in order to send a message:

- The mailbox to which the message is to be sent.
- A pointer to a user message header which contains the information about the message. This information contains:
  * The length of the message.
  * Whether or not the message requires a response.
  * A pointer to the actual message.

This information is copied from user message header to system message header, so that the user message header can be reused.
- A wait period indicating the amount of time the task is willing to wait, if there is no system message header available. That is if the maximum number of outstanding messages has been reached.

- A pointer to a task which identifies the receiving task of the message.

- A pointer to a word through which the status of the system call is returned.

If the receiving task is already waiting for the message in the mailbox, the message is delivered directly to the task without being attached in the mailbox.

A task is preempted if there is no system message header available and it wishes to wait. After control is returned to the task, one of the following conditions may have occurred. This occurrence is indicated by the system call return code:

- The task has timed out waiting for a system message header.

- The task received the message header within the wait time period.

During the process of sending a message two preemption can occur -- when there is no message header
available and when the sent message activates a higher priority task. The sender task must explicitly obtain permission to reuse or alter the buffer containing the message, by a WAIT ON BUFFER system call. The receiver task must explicitly free the buffer by a SIGNAL BUFFER system call. This protocol must be exercised to guarantee the proper functioning of the system.

3.1.9 BROADCAST MESSAGE

A task can broadcast a message in the sense that instead of sending individual messages to the recipient tasks, one system call is executed with the receiving tasks specified.

A broadcast map (BCMap) is used to identify the recipient tasks. In SES the tasks are numbered from zero to one less than the maximum number of tasks (max - 1). The numbering scheme is ordered with regard to the order of creation of the tasks. For example, the first created task will be task number zero. The next created task will be task one, and so forth. Since the maximum number of tasks allowed in SES is 64, an attempt to broadcast to more than 64 tasks will result in an error condition. The BCMap is made by the system for the user, although the user must pass the following information for a broadcast:
- The mailbox to which the broadcast message is to be sent. Note that a broadcast message can only go to a public mailbox.

- A pointer to a user message header which contains the information about the message. This information contains:

  * The length of the message.
  * Whether or not the message requires a response.
  * A pointer to the actual message. The first byte of the actual message is the number of the tasks to receive the message.

This information is copied from use2 message header to system message header, so that the user message header can be reused.

- A pointer to a buffer area which contains the list of receiving tasks.

- A wait period, indicating the amount of time the task is willing to wait if there is no system message header available.
- A pointer to a word through which the status of the system call is returned.

To receive a message, the tasks are checked against the BCMap to validate their request. After the receipt of a message, the task is erased from the list of tasks and a second attempt will fail. The receiver of a broadcast message performs a normal RECEIVE MESSAGE. That is, the receiving end does not know about the nature of the message.

3.1.10 MESSAGE BUFFER PROTOCOL

In SES the messages are not copied and only a pointer to a message is passed to the receiving task. The following reasons support this idea:

1. There is no per-task address space in RTTS and the address space is shared among all tasks. Therefore, a task can access any area in memory.

2. RTTS is a single user system. Since there is no malicious user, there is no need to protect the message buffer in one task.

3. The speed of sending and receiving messages is critical. To copy a message from user message buffer to system message buffer when sending, and to
copy from system message buffer to user message buffer when receiving, takes time.

4. There is no dynamic memory allocation in RTTS. This implies that in order to copy messages, a number of system message buffers must be allocated statically at system initialization time. Static allocation of buffers wastes memory if more than enough buffers are allocated, and slows the process down if fewer than enough buffers are allocated.

The disadvantage of not copying the message is that the sender of the message has to wait for the receiving task(s) before it can reuse the message buffer. There are two system calls provided for this purpose:

1. WAIT ON BUFFER: This system call checks the user buffer to see if it is free to be used or not. If the message is still in process the task could either wait for a period of time or return with the code indicating that the buffer is not free yet. If the task wishes to wait, it will be reactivated when the receiving task is done and signals the system, or when it times out.

2. SIGNAL BUFFER: This system call is invoked by the recipient(s) of a message. It could activate a suspended task. In the case of a broadcast message,
the task is activated after the last receiving task has signaled the freeing of the buffer.

The execution order of two system calls must be borne in mind to guarantee the proper message exchange. See section 3.1.8 (SEND MESSAGE).

3.1.11 SEMAPHORE PRIMITIVES

A semaphore entity is nothing but a mailbox. The same entity could be used as a mailbox or as a semaphore (i.e. a mailbox can be deleted by DELETE MAILBOX and be redefined as a semaphore by CREATE SEMAPHORE system call). In fact a semaphore is a mailbox with no message in it. A semaphore is identified by an internal flag. Two words in the exchange structure are used to indicate the current number of units for that resource, and the maximum number of units that resource could have. The primitives necessary for the semaphore operations are:

1. CREATE SEMAPHORE: allows a semaphore to be created. This is not a dynamic creation of the exchange. The system call returns the address of the semaphore to the task which will be used as a reference to the semaphore. The number of resources initially available is indicated as a parameter. The creator task will be the owner of the semaphore and will be
the only task privileged to destroy the semaphore.

2. DELETE SEMAPHORE: allows a semaphore to be deleted. This is a privileged function of the owner of the semaphore (i.e. the creator). The semaphore must be empty before it can be deleted. The owner task can purge the waiting tasks by a 'PURGE SEMAPHORE' system call. All the tasks waiting for the units of the semaphore will be signaled and are activated.

3. WAIT: allows a task to request units of a semaphore. A pointer to the semaphore is passed as a parameter. The task also specifies the number of units requested and the time to wait if the units of the resource are not available. If the task wishes to wait, in case the requested units are not available, it will be queued on the semaphore and the timer queue. The task is activated in the following three cases:

- when the units of the resource becomes available by a SIGNAL operation and its request could be satisfied.

- when a time out occurs and the task is activated without its request being satisfied.
- when the semaphore is deleted while the task was waiting.

4. SIGNAL: allows a task to return units of a semaphore to the system. If the number of returned units plus the available units satisfies the request of the first waiting task, it will be activated.

In RTTS there is no resource manager. Therefore, the SIGNAL operation does not validate that the sender task had actually acquired the units previously.
Chapter 4

SUMMARY

In a real time, multitask environment the tasks need to communicate and be synchronized. Intertask communication is one way to achieve this goal. All the currently existing operating systems have some way of permitting this synchronization -- in the simplest case using semaphores as permission to do a particular action and more elaborate message passing protocol for more efficient communication and synchronization. SES is basically a monitor designed and implemented for real time environments as a tool for intertask communication. Currently existing real time operating systems such as Thoth, that runs on Data General NOVA and TI 990, and iRMX86 are designed to host many, but small, tasks which is typical in real time systems. Both these operating systems have characteristics that makes them inefficient in terms of memory usage and CPU time (i.e. system throughput). For example in Thoth, any task sending a message is automatically blocked until a response is received from the receiving end. This feature may not be very desirable and in fact reduces the throughput, since:
1. Not all the messages require a response. In fact many messages are in the form of a signal and the sending task can continue execution after the message is sent.

2. The receiving end of a message has to respond back in order to unblock the sending task. Therefore an extra system message header and possibly a message buffer is consumed.

On the other hand, a message does not require a response in iRMX86, but there is the problem of message delivery and identification as follows: Any message is handled in a FIFO manner with respect to a particular mailbox and there is no message identification. In a preemptive real time system, where there are a large number of small tasks running, the flow of control may become unpredictable based on the external events. Therefore the user must be extremely careful about the proper delivery of messages among tasks.

SES increases the overall throughput:

1. By making the need for a response a user option. Any message can demand a response be sent back. Also the sender of a message can request the
response at any time. This allows the sender task to continue execution until the response from the receiving end is necessary to proceed. Therefore more work can get done by both tasks, resulting in an increase in the speed and throughput.

2. By processing messages on a FIFO basis with respect to tasks, and not to mailboxes, since the flow of control within tasks is more predictable than amongst tasks. At the other extreme, processing messages on a message ID basis with respect to two tasks would add extra overhead with little gain.

3. By allowing broadcast messages to be sent. This results in a great amount of processing time and memory resource being saved, since the message is copied only once as opposed to once for every recipient task.

4. By using the same data structure to represent mailboxes and semaphores. The ability to allocate and deallocate these entities and the fact that the same data structure can be used as both a semaphore and a mailbox saves memory space, with no extra overhead.

Notice that in SES messages are not copied to system message buffers when a message is sent. Instead they are exchanged via a pointer reference to the
message. Therefore two or more tasks share the same buffer. Copying of the message would require a memory manager which does not exist in RTTS, and a restriction on the length of the message, as in the case of Thoth. This implies that tasks must cooperate in using the shared buffer (i.e. the sender task must do a WAIT ON BUFFER before altering the contents of the buffer and the receiver must do a SIGNAL BUFFER after it has processed the message).

In conclusion, in SES as in other operating systems, there is a trade off between the speed and the efficiency of the system call executions. These decisions have been made based on the nature of the potential applications of the system. The applications in which SES would be used, must be borne in mind when analyzed.
BIBLIOGRAPHY


Appendix A

RTTS SYSTEM CALLS

DISABLE INTERRUPTS
Parameters:
  none
Return Codes:
  none

RESTORE INTERRUPTS
Parameters:
  none
Return Codes:
  none

SYNCHRONIZE
Parameters:
  none
Return Codes:
  none

GET_WORDS
Parameters:
  pointer to the local word to be fetched
  pointer to the word which will contain P0 data
  pointer to the word which will contain P2 data
  return code pointer
Return Codes:
  0 - no errors
  1 - no data could be fetched from P0
  2 - no data could be fetched from P2
STATUS = QUALIFY_VOTE
Parameters:
  pointer to the word to be voted
  consensus value of the word
Return Codes For STATUS:
  1 - total agreement on the consensus value
  2 - consensus reached between me and PO
  3 - consensus reached between me and P2
  4 - consensus reached between PO and P2
  5 - no agreement on the value

REQUEST_TASK_TO_RUN
Parameters:
  task pointer
  return code pointer
Return Codes:
  0 - no errors
  1 - invalid task pointer

DECREMENT_REQUEST
Parameters:
  task pointer
  return code pointer
Return Codes:
  0 - no errors
  1 - invalid task pointer
  2 - task had not been scheduled for execution

CHANGE_TASK_PRIORITY
Parameters:
  task pointer
  new priority
  return code pointer
Return Codes:
  0 - no errors
  1 - invalid task pointer
  3 - priority out of range

EXAMINE_REQUESTS
Parameters:
  task pointer
  pointer to returned request count value
  return code pointer
Return Codes:
  0 - no errors
  1 - invalid task pointer
REQUEST_RELATIVE_TASK
Parameters:
  task pointer
  low ticks
  high ticks
  return code pointer
Return Codes:
  0 - no errors
  1 - invalid task pointer
  2 - the task is already scheduled

REMOVE_RELATIVE_TASK
Parameters:
  task pointer
  return code pointer
Return Codes:
  0 - no errors
  1 - invalid task pointer
  2 - task is not scheduled relatively

ABORT_TASK
Parameters:
  task pointer
  return code pointer
Return Codes:
  0 - no errors
  1 - task is not executing

LINK_TO_EVENT
Parameters:
  task pointer
  event ID
  return code pointer
Return Codes:
  0 - no errors
  1 - invalid task pointer
  14H - invalid event ID

UNLINK_EVENT
Parameters:
  event ID
  return code pointer
Return Codes:
  0 - no errors
  14H - invalid event ID
MASK_EVENT
Parameters:
   event ID
   return code pointer
Return Codes:
   0 - no errors
   14H - invalid event ID

UNMASK_EVENT
Parameters:
   event ID
   return code pointer
Return Codes:
   0 - no errors
   14H - invalid event ID

RESET_EVENT
Parameters:
   event ID
   return code pointer
Return Codes:
   0 - no errors
   14H - invalid event ID

EXAMINE_EVENT_STATE
Parameters:
   event ID
   return code pointer
Return Codes:
   0 - no errors
   14H - invalid event ID

CAUSE_EVENT
Parameters:
   event ID
   return code pointer
Return Codes:
   0 - no errors
   14H - invalid event ID
   12H - event is masked
   13H - no task is linked
Appendix B

SES SYSTEM CALLS

CREATE_MAILBOX
Parameters:
  type of mailbox
    0 - private
    1 - semi-private
    2 - public
  for a private mailbox, pointer to the co-owner task
  returned mailbox pointer
  return code pointer

Return Codes:
  0 - no errors
  3 - no space for a new mailbox

DELETE_MAILBOX
Parameters:
  pointer to the mailbox
  return code pointer

Return Codes:
  0 - no errors
  4 - not privileged to delete the mailbox
  5 - mailbox is not empty
  8 - invalid mailbox pointer

PURGE_MAILBOX
Parameters:
  pointer to the mailbox
  purge code
    0 : purge messages only
    1 : purge tasks only
    2 : purge both messages and tasks
  return code pointer

Return Codes:
  0 - no errors
  4 - not privileged to purge the mailbox
  8 - invalid mailbox pointer
SEND MESSAGE
Parameters:
pointer to the destination mailbox
task pointer to whom the message is sent
pointer to the user message header
wait period for a message header
return code pointer
Return Codes:
0 - no errors
2 - the task pointer specified is invalid
3 - no message header and not willing to wait
4 - no message header, waited and timed out
8 - invalid mailbox pointer
9 - invalid request from the mailbox

RECEIVE MESSAGE
Parameters:
mailbox pointer from which the message is expected
the task pointer from whom the message is expected
buffer pointer which will have message information
wait period for a message
return code pointer
Return Codes:
0 - no errors
2 - the task pointer specified is invalid
3 - no message is pending and not willing to wait
4 - no pending message, waited and timed out
8 - invalid mailbox pointer
9 - invalid request from the mailbox
12H - mailbox was deleted during the wait period

BROADCAST
Parameters:
pointer to the recipient task pointer table
pointer to the mailbox to which the message is sent
wait period for a message header
pointer to the user message header
return code pointer
Return Codes:
0 - no errors
3 - no message header and not willing to wait
4 - no message header, waited and timed out
8 - invalid mailbox pointer
9 - the mailbox is not a public mailbox
10H - the number of recipient tasks is more than 64
WAIT_ON_BUFFER
Parameters:
  buffer address to wait on
  wait period for the release of the buffer
  return code word
Return Codes:
  0 - no errors
  3 - buffer not free, not waiting
  4 - buffer not free, waited, timed out
  13H - buffer already free

SIGNAL_BUFFER
Parameters:
  pointer to the owner of the buffer
  pointer to the buffer
  return code pointer
Return Codes:
  0 - no errors
  2 - invalid task pointer

CREATE_SEMAPHORE
Parameters:
  returned semaphore pointer
  initial units for the semaphore
  return code pointer
Return Codes:
  0 - no errors
  3 - no space for a new semaphore

DELETE_SEMAPHORE
Parameters:
  semaphore pointer
  return code pointer
Return Codes:
  0 - no errors
  4 - not privileged to delete the semaphore
  5 - semaphore is not empty
  20H - invalid semaphore pointer

PURGE_SEMAPHORE
Parameters:
  semaphore pointer
  return code pointer
Return Codes:
  0 - no errors
  4 - not privileged to purge the semaphore
  20H - invalid semaphore pointer
WAIT
Parameters:
    semaphore pointer
    units requested
    wait period for the units
    return code pointer
Return Codes:
    0 - no errors
    3 - no resource available, not waiting
    4 - no resource available, waited timed out
    9 - requested units is more than available
    20H - the semaphore pointer is invalid

SIGNAL
Parameters:
    semaphore pointer
    units of semaphore released
    return code pointer
Return Codes:
    0 - no errors
    9 - illegal number of units released
    20H - the semaphore pointer is invalid
Appendix C

SES COMMENTED SOURCE CODE LISTING

MBX DELETION CREATION:  DO;

/-----------------------------------------------*/
*
* NAME:  MBX DELETION CREATION
*
* SYSTEM:  S. E. S.
*
* DATE:  FEB 15, 1982
*
*-----------------------------------------------*/

PURPOSE: This module contains the procedures necessary to delete
* and create mailboxes and semaphores.
*
PROCEDURE INCLUDED:
* 1. CREATE MAILBOX: To find an unused exchange to be allocated.*
* 2. DELETE MAILBOX: To delete an existing mailbox.
* 3. CREATE SEMAPHORE: To find an unused exchange to be allocated.*
* 4. DELETE SEMAPHORE: To delete an existing semaphore.
* 5. FIND IT: To search the table of exchanges for a free*
* exchange.
*
EXTERNAL ROUTINES USED:
* 1. ADD TO PTR: Adds an offset to the value of a pointer.*
* 2. DISABLE INTERRUPTS: Disables interrupts simultaneously in all*
* three processors.
* 3. RESTORE INTERRUPTS: Restores interrupts state.
*
/-----------------------------------------------*/

ADD_TO_PTR:  PROCEDURE ( ptr_value, offset) POINTER EXTERNAL;
  DECLARE  ptr_value  POINTER;
  DECLARE  offset   WORD;
END ADD_TO_PTR;

DISABLE_INTERRUPTS: PROCEDURE EXTERNAL;
END DISABLE_INTERRUPTS;

RESTORE_INTERRUPTS: PROCEDURE EXTERNAL;
END RESTORE_INTERRUPTS;
$INCLUDE (:F1:MBXEXT.DEF)
$INCLUDE (:F1:PLMEXT.DEF)

DECLARE ini LITERALLY '0';
DECLARE cur LITERALLY '1';
FIND_IT: PROCEDURE ( ex$type, ex$name, sec$owner) PUBLIC;

/*
 * NAME: FIND IT
 * SYSTEM: S. E. S.
 * DATE: FEB 15, 1982
 *
 * PURPOSE: To find an unused exchange in the system to be allocated either as a semaphore or a mailbox.
 *
 * FUNCTION:
 * I. Set the pointer to the beginning of the exchange table.
 * II. While an unused exchange is not found:
 * A. If the current exchange’s owner is null:
 * 1. Set the found flag to TRUE.
 * 2. Add one to the current number of exchanges.
 * 3. Set the owner of the exchange to the tcb_base.
 * 4. Set the type of the mailbox to the type specified as parameter.
 * 5. Set the second owner if the mailbox is a private one.
 * 6. Set the check field to AAAA.
 * 7. Set the return name to the address of the exchange.
 * B. Otherwise:
 * 1. Get the next exchange.
 * III. Return.
 */

/*
 * PARAMETERS PASSED:
 * 1. ex$type: 0 __ private 1 __ semi-private
 * 2 __ public 4 __ semaphore
 * 2. ex$name: The return pointer to the exchange.
 * 3. sec$owner: The task pointer of the second owner if the mailbox is private.
 *
 * LOCAL VARIABLES:
 * 1. found: Flag indicating the exchange is found.
 * 2. return$name: The based pointer on the user defined pointer.
 */

DECLARE ex$type WORD;
DECLARE found WORD;
DECLARE sec$owner POINTER;
DECLARE ex$name POINTER;
DECLARE return$name BASED ex$name POINTER;

/*
 * Set the base pointer to the beginning of the exchange table and start a search through the table.
 */
found = FALSE;
mbx_base = ptr_mboxes_table;

DO WHILE NOT found;

/*-----------------------------------------------*/
* If the owner1 of the exchange is 0, it is unused. *
* Set the specified attributes in the exchange fields. *
* Set the return pointer to the address of the exchange. *
*-----------------------------------------------*/

IF mailbox.owner1 = 0 THEN DO;
    found = TRUE;
    num_exchanges = num_exchanges + 1;
    mailbox.owner1 = tcb_base;
    mailbox.type = ex$type;
    IF ex$type = PRIVATE THEN
        mailbox.owner2 = sec$owner;
    return$name = mbx_base;
    mailbox.check = CORRECT_USER_NAME;
END;
ELSE

/*-----------------------------------------------*/
* If not found yet, check the next exchange. *
*-----------------------------------------------*/

mbx_base = ADD_TO_PTR ( mbx_base, SIZE( mailbox) );
ENDIF
END;
END FIND_IT;
CREATE_MAILBOX: PROCEDURE (type, owner2, mbx$name, rt$ptr) PUBLIC;

* * *
* NAME: CREATE MAILBOX *
* SYSTEM: S. E. S. *
* DATE: MARCH 5, 1982 *
* *
* PURPOSE: To create a mailbox by defining an exchange to be one. *
* This is not dynamic creation of mailboxes. *
* *
* FUNCTION: *
* I. Disable interrupts for a user task. *
* II. If the maximum number of exchanges allowed is reached, set *
* the return code. *
* III. Otherwise: *
* A. If the task specified by the task as the second owner of *
* a private mailbox is invalid, set the return code. *
* B. Otherwise: *
* 1. Call the routine to find an unused exchange. *
* 2. Set the return code. *
* IV. Restore interrupts for a user task. *
* *
* PARAMETERS PASSED: *
* 1. type: 0 __ private mailbox. 1 __ semi-private *
* 2 __ public mailbox. 4 __ semaphore *
* 2. owner2: The second owner of a mailbox for a private *
* mailbox. *
* 3. mbx$name: The pointer to the mailbox returned to task. *
* 4. rt$ptr: The return code pointer. *
* *
* LOCAL VARIABLES USED: *
* 1. crt$result: The return code word. *
* *
DECLARE rt$ptr POINTER;
DECLARE type WORD;
DECLARE crt$result BASED rt$ptr WORD;
DECLARE (owner2, mbx$name) POINTER;
/* Disable interrupts and check if there is any room for a new mailbox by examining the number of current exchanges. */

IF system_state = user_state THEN
  CALL DISABLE_INTERRUPTS;

IF num_exchanges >= max_exchanges THEN
  crt$result = TOO_MANY_EXCHANGES;
ELSE DO;

  /*
  * If the co-owner task specified for a private mailbox is valid, call the routine to allocate the mailbox.
  */
  task_base = owner2;
  IF task.name_check <> CORRECT_USER_NAME AND type = PRIVATE THEN
    crt$result = INVALID_TSK;
  ELSE DO;
    CALL FIND_IT (type, mbx$name, owner2);
    crt$result = OK;
  END;
END;

IF system_state = user_state THEN
  CALL RESTORE_INTERRUPTS;

END CREATE_MAILBOX;
DELETE_MAILBOX:  PROCEDURE ( mbx$name, rt$ptr)  PUBLIC;

*---------------------------------------------------------------------*
*  *  NAME:       DELETE MAIBLOX  *
*  SYSTEM:      S. E. S.       *
*  DATE:        MARCH 5, 1982  *
*---------------------------------------------------------------------*
* PURPOSE: To delete an existing mailbox from the system. Note *  *    that a mailbox can be deleted only if there is no *  *    message pending in the mailbox and there is no task *  *    suspended on it.  *
* FUNCTION: *  I. Disable interrupts for a user task.  *
*         II. If the caller task is not the owner of the mailbox, set  *
*            the return code.  *
*         III. Otherwise:  *
*            A. If there is a task waiting or a message pending, set the  *
*               return code.  *
*            B. Otherwise,  *
*               1. Decrement the number of current exchanges.  *
*               2. Set the owner, second owner, and type to 0.  *
*               3. Set the return code to completed.  *
*         IV. Restore interrupts for a user task.  *
*---------------------------------------------------------------------*

/---------------------------------------------------------------------*
* PARAMETERS PASSED:  *
*  1. mbx$name:  The mailbox pointer to be deleted.  *
*  2. rt$ptr:  The return code pointer.  *
*---------------------------------------------------------------------*

* LOCAL VARIABLES USED:  *
*  1. del$result:  The return code word.  *
*---------------------------------------------------------------------*/

DECLARE mbx$name  POINTER;
DECLARE rt$ptr  POINTER;
DECLARE del$result  BASED rt$ptr  WORD;
/* Disable interrupts for a user task and check if the caller of * the system call is the owner of the mailbox. * Then check to see if there is a message pending or a task waiting * in the mailbox. Set the return code in any of those cases. */

IF system_state = user_state THEN
    CALL DISABLE_INTERRUPTS;

    mbx_base = mbx$name;

    IF mailbox.owner1 <> tcb_base THEN
        del$result = NOT_PRIVILEGED;
    ELSE DO;
        IF mailbox.mqhead <> mbx_base
            OR mailbox.tqhead <> ADD_TO_PTR ( mbx_base, 8 ) THEN
            del$result = NOT_EMPTY;
        ELSE DO;
            /* Decrement one from the number of current exchanges and * Set the owner, co-owner, and the type to 0. * Set the return code, restore interrupts and return. */
            num_exchanges = num_exchanges - 1;
            mailbox.owner1 = 0;
            mailbox.owner2 = 0;
            mailbox.type = 0;
            del$result = OK;
        END;
    END;

    IF system_state = user_state THEN
        CALL RESTORE_INTERRUPTS;

END DELETE_MAILBOX;

CREATE SEMAPHORE :PROCEDURE (sem$name, ini_units, rt) PUBLIC;

* PURPOSE: To create a semaphore by allocating an exchange.
* This is not dynamic creation of a semaphore.
*
* FUNCTION:
* I. Disable interrupts for a user task.
* II. If the maximum number of exchanges allowed is reached, set * the return code.
* III. Otherwise:
* A. Call the routine to find an unused exchange.
* B. Set the initial value of the semaphore in the exchange, * using the mailbox.mqtail as the initial units and current* units of the semaphore.
* C. Set the return code.
* IV. Restore interrupts for a user task.
*
* PARAMETERS PASSED:
* 1. ini_units: The initial units in a semaphore.
* 2. sem$name: The pointer to the semaphore returned to task.*
* 3. rt: The return code pointer.
*
* LOCAL VARIABLES USED:
* 1. sem$status: The return code word.
* 2. dummy: A dummy pointer to address 'mailbox.mqtail', * to set the initial and current units.
* 3. sem(2): An array based on dummy which is in turn the * same as mailbox.mqtail. The first element is * the initial value. The second element is the * current value of the units in the semaphore.
*
DECLARE (sem$name, rt) POINTER;
DECLARE ini_units WORD;
DECLARE sem$status BASED rt WORD;
DECLARE dummy POINTER;
DECLARE sem BASED dummy(2) WORD;
IF system_state = user_state THEN
    CALL DISABLE_INTERRUPTS;

IF num_exchanges >= max_exchanges THEN
    sem$status = TOO_MANY_EXCHANGES;
ELSE
    CALL FIND_AT (4, sem$name, 0);
END;

IF system_state = user_state THEN
    CALL RESTORE_INTERRUPTS;

END CREATE_SEMAPHORE;
DELETE_SEMAPHORE: PROCEDURE (sem$name, rt) PUBLIC;

/*------------------------------------------------------------*/

* NAME: DELETE_SEMAPHORE
* SYSTEM: S. E. S.
* DATE: MARCH 5, 1982

* PURPOSE: To delete an existing semaphore from the system. Note
* that a semaphore can be deleted only if there is no
* task suspended on it.
*
* FUNCTION:
* I. Disable interrupts for a user task.
* II. If the caller task is not the owner of the semaphore, set
* the return code.
* III. Otherwise:
* A. If there is a task waiting, set the return code.
* B. Otherwise:
* 1. Decrement the number of current exchanges.
* 2. Set the owner, initial and current value to 0.
* Note that these two fields are same as mailbox.mqtail.
* 3. Set the return code to completed.
* IV. Restore interrupts for a user task.

*------------------------------------------------------------*/

/ * PARAMETERS PASSED: */
* 1. sem$name: The semaphore pointer to be deleted.
* 2. rt: The return code pointer.

/ * LOCAL VARIABLES USED: */
* 1. del$status: The return code word.

*------------------------------------------------------------*/

DECLARE (sem$name, rt) POINTER;
DECLARE del$status BASED rt WORD;
/* Disable interrupts for a user task, then check: */
* 1. The caller task to see if it is the owner.
* 2. To see if there is any task suspended on the semaphore.
* If any of the above is true, set the return code.

IF system_state = user_state THEN
CALL DISABLE_INTERRUPTS;

mbx_base = sem$name;
IF mailbox.owner <> tcb_base THEN
del$status = NOT_PRIVILEGED;
ELSE DO;
IF mailbox.tqhead <> @mailbox.tqhead THEN
del$status = NOT_EMPTY;
ELSE DO;

/*
* Decrement the number of current exchanges by 1 and set *
* the owner, type, and mailbox.mqtail to 0.
*/
num_exchanges = num_exchanges - 1;
del$status = OK;
mailbox.owner = 0;
mailbox.type = 0;
mailbox.mqtail = 0;
END;
END;

IF system_state = user_state THEN
CALL RESTORE_INTERRUPTS;
END DELETE_SEMAPHORE;
END MBX_DELETION_CREATION;
SEND_MESSAGE: DO;

/* --------------------------------------------------------------- *
 * NAME: SEND MESSAGE
 * SYSTEM: S. E. S.
 * DATE: MARCH 5, 1982
 *
 * PURPOSE: To allow the user to send a message to any mailbox
 * if eligible.
 *
 * EXTERNAL ROUTINES USED:
 * 1. SUSPEND: To suspend the sender task. If the maximum number of
 * outstanding messages was reached and the task wishes
 * to wait.
 * 2. WAKE UP: To awaken a suspended task, if the sent message has
 * a task waiting for it, the message is delivered to
 * the waiting task.
 * 3. INSERT ENTRY: To enqueue the message in the mailbox, if there
 * was not any task waiting for it.
 * 4. GET HEADER FROM POOL: To copy the user provided information
 * to a system buffer, so that the user buffer can be
 * returned to it.
 * 5. ADD TO PTR: To add an offset to the value of a pointer.
 * 6. CHECK MBX ID: To check the mailbox pointer passed by the task
 * to see 1) if it is a mailbox pointer;
 * 2) if it is valid for the specific operation.
 * 7. DISABLE INTERRUPTS: To disable interrupts simultaneously in
 * all three processors.
 * 8. RESTORE INTERRUPTS: To restore the state of the interrupts to
 * the incoming state.
 * ---------------------------------------------------------------*/

SUSPEND: PROCEDURE ( queue, wait ) WORD EXTERNAL;
DECLARE queue POINTER;
DECLARE wait WORD;
END SUSPEND;

WAKE_UP: PROCEDURE ( task$name, queue, resource, preemption ) EXTERNAL;
DECLARE (task$name, queue) POINTER;
DECLARE resource POINTER;
DECLARE preemption WORD;
END WAKE_UP;

INSERT ENTRY : PROCEDURE (prev, header,status ) EXTERNAL;
DECLARE (prev, header,status) POINTER;
END INSERT_ENTRY;

GET_HEADER_FROM_POOL: PROCEDURE POINTER EXTERNAL;
END GET_HEADER_FROM_POOL;

ADD_TO_PTR : PROCEDURE ( ptr, offset ) POINTER EXTERNAL;
DECLARE ptr POINTER;
DECLARE offset WORD;
END ADD_TO_PTR;

CHECK_MBX_ID: PROCEDURE (mbx, code) WORD EXTERNAL;
DECLARE mbx POINTER;
DECLARE code WORD;
END CHECK_MBX_ID;

DISABLE_INTERRUPTS: PROCEDURE EXTERNAL;
END DISABLE_INTERRUPTS;

RESTORE_INTERRUPTS: PROCEDURE EXTERNAL;
END RESTORE_INTERRUPTS;

$INCLUDE (:F1:MBXEXT.DEF)
$INCLUDE (:F1:PLMEXT.DEF)
SEND_MESSAGE: PROCEDURE (mbx, to$who, user/head$base, wait, ret$ptr) PUBLIC;

*/---------------------------------------------------------------*/
*/
*/ NAME: SEND MESSAGE
*/ SYSTEM: S. E. S.
*/ DATE: MARCH 5, 1982
*/
*/---------------------------------------------------------------*/

PRUPOSE: To provide the user the facility to send messages.

FUNCTION:
I. Disable interrupts and synchronize for a user task.
II. Check the task ID on the receiving end. If not valid:
   A. Set the return code.
III. Otherwise:
   A. Check the ID of the mailbox for that task, for a send.
      1. Set the return code, if illegal mailbox.
   B. Otherwise,
      1. If max. number of outstanding messages is reached.
         a. If the task does not wait, set the return code.
         b. Otherwise:
            i. Suspend the task.
            ii. When return, if buffer granted, get the
                buffer address, and reset the flag. Otherwise
                the return code is set to NOT OK by suspend.
      2. Otherwise:
         a. Get the header from pool.
         b. < Note the return code is OK. >
      3. If the return code is OK:
         a. Copy the user information to system header.
         b. From the top of the waiting tasks queue:
            i. If the specific task is waiting for a mess-
               age from this task, set the found flag to
               true and awaken the task with preemption.
            ii. Otherwise, get the next task.
            c. If the task was not found by the end of queue,
               enqueue the message in mailbox.
   IV. Restore interrupts for a user task and return.
PARAMETERS PASSED:
1. mbx: The destination mailbox.
2. to$who: The destination task.
3. user$head$base: The pointer to user provided information.
4. wait: The time task is willing to wait for a header, if not available.
5. ret$ptr: The pointer to the return code.

LOCAL VARIABLES:
1. send$status: The return code word.
2. found: Indicates whether the recipient task was found.
3. mailbox_task_q: The pointer to the mailbox waiting tasks queue.
4. status: A dummy word for calling procedures.
5. SEND CODE: A literal '1' for CHECK MBX ID indicating the operation done on that mailbox is a 'send'.
6. RESOURCE$GRANTED: Indicates whether after activation the resource was granted.

DECLARE (user$head$base, ret$ptr) POINTER;
DECLARE wait WORD;
DECLARE to$who POINTER;
DECLARE send$status BASED ret$ptr WORD;
DECLARE found WORD;
DECLARE (mailbox_task_q, mbx) POINTER;
DECLARE status WORD;

DECLARE user$head BASED user$head$base STRUCTURE
LENGTH WORD,
response WORD,
data$ptr POINTER);

DECLARE SEND$CODE LITERALLY '1';
DECLARE RESOURCE$GRANTED LITERALLY '0';

IF system_state = user_state THEN
CALL DISABLE_INTERRUPTS;
task_base = to$who;

/* Check the parameters: 1) The destination task.*/
/* 2) The destination mailbox.*/

IF task.name_check <> CORRECT_USER_NAME THEN
send$status = INVALID_TSK;
ELSE DO;
send$status = CHECK_MBX_ID (mbx, SEND$CODE);
IF send$status = OK THEN DO;
/*---------------------------------------------------------------*/
  /* If no more message header available: */
  /*   ____ Return if no wait desired. */
  /*   ____ Suspend for the desired amount of time */
/*---------------------------------------------------------------*/
  IF num_messages >= max_messages THEN DO;
    IF wait = 0 THEN
      send$status = NOT_WAIT;
    ELSE DO;
      send$status = SUSPEND ( @buffer_queue, wait );
    /*---------------------------------------------------------------*/
    /* If after return from suspension the buffer */
    /* is available, get the buffer address. */
    /*---------------------------------------------------------------*/
    IF send$status = RESOURCE$GRANTED THEN DO;
      sys_head_base = tcb.timer_q_ptrs(2);
      tcb.timer_q_ptrs(2) = 0;
      tcb.period(1) = 2;
    END;
    END;
END;
/*---------------------------------------------------------------*/
/* Message header is available, get the buffer from pool. */
/*---------------------------------------------------------------*/
ELSE sys_head_base = GET_HEADER_FROM_POOL;
ENDIF
/*---------------------------------------------------------------*/
/* If status is OK, (i.e., either the buffer was available */
/* or it was granted after suspension). copy the message */
/* to system header. */
/*---------------------------------------------------------------*/
IF send$status = OK THEN DO;
  mbx_base = mbx;
  sys_head.length = user$head.length;
  sys_head.response = user$head.response;
  sys_head.sender = tcb_base;
  sys_head.receiver = to$who;
  sys_head.data_ptr = user$head.data$ptr;
  sys_head.bc = 0;
  found = FALSE;
  task_base = mailbox.tqhead;
mailbox_task_q = ADD_TO_PTR (mbx, 8);

/*---------------------------------------------------------------*
  * Check the waiting tasks queue in the mailbox for a *
  * task waiting for this message (i.e. from the caller to*  
  * the waiting task). If any task found, awaken the task,*
  * with possible preemption.                                  *
  *---------------------------------------------------------------*/

DO WHILE (task_base <> mailbox_task_q) AND NOT found;
    IF task.from who = tcb base 
       AND to$who = task_base THEN DO;
        task.period (1) = 5;
        CALL WAKE_UP (task_base, mailbox_task_q, 
        sys_head_base, OFFH);
        found = TRUE;
    END;
    ELSE
        task_base = task.rqlnk_for;
    ENDIF
END;

/*---------------------------------------------------------------*
  * If there was no task waiting for the message, enqueue it.*
  * Restore interrupts for a user task and return.               *
  *---------------------------------------------------------------*/

IF NOT found THEN
    CALL INSERT_ENTRY (mailbox.mqtail, sys_head_base, 
    @status);
ENDIF
END;

IF system_state = user_state THEN
    CALL RESTORE Interrupts;
END SEND_MESSAGE;
END SEND_MESSAGE;
BROADCAST MESSAGE: DO;

/*------------------------------------------*/

*  NAME:          BROAD CAST MESSAGE
*  SYSTEM:        S. E. S.
*  DATE:          MARCH 5, 1982

* PURPOSE: To allow a task to broadcast a message to any public mailbox.

* EXTERNAL ROUTINES USED:
* 1. SUSPEND: To suspend the sender task, if the maximum number of
   outstanding messages was reached and the task wishes to wait.
* 2. WAKE UP: To awaken a suspended task, if the sent message has a
   task waiting for it. The message is delivered to the waiting task.
* 3. INSERT ENTRY: To enqueue the message in the mailbox, if not
   all the recipient tasks were waiting for it.
* 4. GET HEADER FROM POOL: To copy the user provided information to
   a system buffer, so that the user buffer can be returned to it.
* 5. BUILD BC MAP: This procedure build a BCMap of the recipient
   tasks in the message header for an ID checking when to deliver a message.
* 6. VALID BC TASK: This procedure checks the task to see if it is
   one of the valid recipients of the broadcast message.
* 7. CHECK MBX ID: To check the mailbox pointer passed by the task
   to see 1) It is a mailbox pointer.
   2) It is valid for the specific operation.
* 8. DISABLE INTERRUPTS: To disable interrupts simultaneously in
   all three processors.
* 9. RESTORE INTERRUPTS: To restore the state of the interrupts to
   the incoming state.

------------------------------------------*/

SUSPEND: PROCEDURE ( qcb, wait ) WORD EXTERNAL;
DECLARE qcb POINTER;
DECLARE wait WORD;
END SUSPEND;

WAKE_UP: PROCEDURE ( task, qcb, resource, preemption ) EXTERNAL;
DECLARE ( task, qcb ) POINTER;
DECLARE preemption WORD;
DECLARE resource POINTER;
END WAKE_UP;

DISABLE_INTERRUPTS: PROCEDURE EXTERNAL;
END DISABLE_INTERRUPTS;

RESTORE_INTERRUPTS: PROCEDURE EXTERNAL;
END RESTORE_INTERRUPTS;

GET_HEADER_FROM_POOL: PROCEDURE POINTER EXTERNAL;
END GET_HEADER_FROM_POOL;

INSERT_ENTRY: PROCEDURE (prev, new, rt) EXTERNAL;
DECLARE (prev, new) POINTER;
DECLARE rt POINTER;
END INSERT_ENTRY;

BUILD_BC_MAP: PROCEDURE (map$ptr, t$no) EXTERNAL;
DECLARE map$ptr POINTER;
DECLARE t$no BYTE;
END BUILD_BC_MAP;

VALID_BC_TASK: PROCEDURE (task$num) WORD EXTERNAL;
DECLARE task$num WORD;
END VALID_BC_TASK;

CHECK_MBX_ID: PROCEDURE (mbx, code) WORD EXTERNAL;
DECLARE mbx POINTER;
DECLARE code WORD;
END CHECK_MBX_ID;

$INCLUDE (:F1:MBXEXT.DEF)
$INCLUDE (:F1:PLMEXT.DEF)
BROADCAST: PROCEDURE ( bc$table, mbx, wait, user$head$base, rt )
PUBLIC;

/*
 * NAME: BROADCAST
 * SYSTEM: S. E. S.
 * DATE: MARCH 9, 1982
 * PURPOSE: To allow a task to broadcast a message to several tasks. The
 * mailbox to which the broadcast message is sent must be public.
 * The maximum number of task to receive the message must not exceed 64.
 * FUNCTION:
 * I. Disable interrupts for a user task.
 * II. Set a dummy pointer to the message pointer, so that the number of the
 * recipient tasks could be extracted.
 * NOTE: The number of recipient tasks is the first byte of the message.
 * III. If the number of recipients exceeds 64, set the return code.
 * A. Otherwise, check the mailbox id.
 *   1. If mailbox pointer is valid AND is mailbox is public:
 *      a. If no message header available,
 *         1. If the task does not wish to wait, set the return code.
 *      2. Otherwise:
 *         i. Suspend the task.
 *            (** task activated **)  
 *            II. If the task could get a message header,
 *               get its address from the TCB.
 *      b. Otherwise:
 *         1. get a message header from the pool.
 * IV. If a message header is acquired:
 *        A. Copy the message header to the system buffer.
 *        B. Build the BCMap in the message header.
 *        C. While not end of the mailbox waiting task queue and not
 *           of the recipients task are found:
 *          1. If there is any task waiting for this message:
 *             a. Wake it up, without preemption.
 *             b. Set the FOUND flag to true.
 *          D. If not all the recipient tasks were found:
 *             1. Enqueue the message in the mailbox.
 *             E. If at least one recipient task was waiting, preempt.
 * V. Restore interrupts for a user task.
 * */
PARAMETERS PASSED:
1. bc$table: A pointer to the recipient tasks.
2. mbx: The mailbox to which the message is sent.
3. wait: The period of time the task is willing to wait for a message header, if necessary.
4. user$head$base: The pointer to the user message header.
5. rt: Pointer to the return code word.

LOCAL VARIABLES USED:
1. dummy: A dummy pointer, to be able to extract the number of recipient tasks from the message pointer.
2. no: Number of recipient tasks for the message.
3. tasks: An array of Task Control Block pointers.
4. bc$status: The return code word.
5. found: Flag indicates whether or not there is any task waiting for the message.
6. RESOURCE$GRANTED: Set to FFFFH if the message header was granted after the wait period.
7. BC$CODE: Broadcast code, to be used in CHECK MBX ID.
8. user head: User message header structure that contains the information about the message.
9. status: Dummy word for a call to INSERT ENTRY.

DECLARE ( bc$table, mbx) POINTER;
DECLARE user$head$base POINTER;
DECLARE wait WORD;
DECLARE rt POINTER;
DECLARE dummy POINTER;
DECLARE no BASED dummy BYTE;
DECLARE tasks BASED bc$table(1) POINTER;
DECLARE bc$status BASED rt WORD;
DECLARE user$head BASED user$head$base STRUCTURE
( length WORD,
  response WORD,
  data$ptr POINTER);
DECLARE found WORD;
DECLARE status WORD;
DECLARE RESOURCE$GRANTED LITERALLY '0';
DECLARE BC$CODE LITERALLY '2';

DISABLE interrupts and set a dummy pointer to the actual message.
Then check the number of recipient tasks. If more than 64, set the return code. Otherwise, check the mailbox pointer.
IF system_state = user_state THEN
  CALL DISABLE_INTERRUPTS;

  dummy = user$head.data$ptr;
  IF no > 64 OR no > NUM_TCBS THEN
    bc$status = TOO_MANY_BCAST;
    ELSE DO;
      bc$status = CHECK MBX_ID ( mbx, BC$CODE );
      IF bc$status = OK THEN DO;
        IF a message header is not available and the task is not willing to wait, set the return code. Otherwise, suspend it.
        * If a message header is available, get one from the pool. *
      END;
    END;
  END;

  IF num_messages >= max_messages THEN DO;
    IF wait = 0 THEN
      bc$status = NOT_WAIT;
    ELSE DO;
      IF system_state = user_state THEN DO;
        bc$status = SUSPEND ( @buffer_queue, wait );

        * If after return the message header is granted, *
        * get the address from the TCB and set the flag *
        * in the TCB to 2 (indicating task is not in the *
        * timer queue). *
      END;
    END;
  END;
ELSE
    bc$status = CANNOT_PREEMPT_RTTS;
ENDIF

END;
ELSE
  sys_head_base = GET_HEADER_FROM_POOL;
ENDIF

/* If the status is OK so far (i.e. either the message header was available or it was granted after preemption ), copy the message header into the system header. Build the bit map for the message. */
IF bc$status = OK THEN DO;
    mbx_base = mbx;
    sys_head.length = user$head.length;
    sys_head.response = user$head.response;
    sys_head.data_ptr = user$head.data$ptr;
    sys_head.sender = tcb_base;
    sys_head.receiver = 0;
    sys_head.bc = no;
    CALL BUILD_BC_MAP ( bc$table, no);
    found = FALSE;
    task_base = mailbox.tqhead;

    /*--------------------------------------------------------------*
    * While there is a task waiting in the queue, and not all       *
    * the tasks have received their message, check to see if       *
    * any waiting task is expecting a message from the caller,     *
    * and if so check to see if it is a valid recipient of the     *
    * message, if so wakeup the task, and set the found flag      *
    * to true.                                                    *
    *                                                          *
    * NOTE: In a call to wakeup, the preemption parameter is 0,    *
    * since a preemption should occur after all the               *
    * recipients of the message are activated (if any ).         *
    *----------------------------------------------------------------
    
    DO WHILE ( task_base <> @mailbox.tqhead )
        AND sys_head.bc <> 0;
        IF task.from who = tcb_base THEN DO;
            IF VALID_BC_TASK ( task.num ) THEN DO;
                sys_head.bc = sys_head.bc - 1;
                task.period(1) = 5;
                CALL WAKE UP ( task_base, @mailbox.tqhead,
                                sys_head_base,0);
                found = TRUE;
            END;
        END;
        ELSE
            task_base = task.rqlnk_for;
        ENDIF
    END;

    /*--------------------------------------------------------------*
    * If all the recipient tasks were already waiting, there       *
    * is no need to enqueue the message. Otherwise enqueue the     *
    * message. Then perform the preemption by a call to           *
    * DISABLE INTERRUPTS.                                         *
    *----------------------------------------------------------------
    
    IF sys_head.bc <> 0 THEN
        CALL INSERT_ENTRY ( mailbox.mqtail, sys_head_base, @status);
    ENDIF
IF found THEN DO;
   CALL DISABLE_INTERRUPTS;
   CALL RESTORE_INTERRUPTS;
   END;
END;
END;
END;
END;

IF system_state = user_state THEN
   CALL RESTORE_INTERRUPTS;
END BROADCAST;
END BROADCAST_MESSAGE;
RECEIVE_MESSAGE: DO;

/*-------------------------------------------------------------*/
/*
/* NAME: RECEIVE MESSAGE
/* SYSTEM: S. E. S.
/* DATE: MARCH 7, 1982
/*
/*-------------------------------------------------------------*/

* PURPOSE: To allow a task to request a message from a mailbox, *
* if the task is eligible for receipt.
* *
* EXTERNAL PROCEDURES USED:
* * 1. SUSPEND: To suspend the requesting task, if no message is in *
* the mailbox and it wishes to wait.
* * 2. WAKE UP: To awaken a suspended task, if a task is waiting on *
* a message header buffer.
* * 3. VALID BC TASK: To check to see if a task is a valid recipient *
* of a broadcast message.
* * 4. RETURN HEADER TO POOL: To return a message header buffer to *
* the pool of system header buffers.
* * 5. CHECK MBX ID: To check the mailbox pointer passed by the task *
* to see if it is valid.
* * 6. DISABLE INTERRUPTS: To disable interrupts simultaneously in *
* three processors.
* * 7. RESTORE INTERRUPTS: To restore interrupts in three processors *
* as the incoming state.
*-------------------------------------------------------------*/

SUSPEND: PROCEDURE (queue, wait ) WORD EXTERNAL;
   DECLARE queue POINTER;
   DECLARE wait WORD;
END SUSPEND;

WAKE_UP : PROCEDURE (task, queue, resource, preempt) EXTERNAL;
   DECLARE task POINTER;
   DECLARE queue POINTER;
   DECLARE resource POINTER;
   DECLARE preempt WORD;
END WAKE_UP;

VALID_BC_TASK: PROCEDURE (task$num ) WORD EXTERNAL;
   DECLARE task$num WORD;
END VALID_BC_TASK;

RETURN_HEADER_TO_POOL: PROCEDURE (header) EXTERNAL;
   DECLARE header POINTER;
END RETURN_HEADER_TO_POOL;

CHECK_MBX_ID: PROCEDURE (mbx, code) WORD EXTERNAL;
   DECLARE mbx POINTER;
   DECLARE code WORD;
END CHECK_MBX_ID;
DISABLE_INTERRUPTS: PROCEDURE
END DISABLE_INTERRUPTS;

RESTORE_INTERRUPTS: PROCEDURE
END RESTORE_INTERRUPTS;

$INCLUDE (:F1:MBXEXT.DEF)
$INCLUDE (:F1:PLMEXT.DEF)
RECEIVE_MESSAGE: PROCEDURE (mbx, from$who, user$head$base, wait, rt$code) PUBLIC;

/*--------------------------------------------------------------*/
* NAME: RECEIVED MESSAGE
* SYSTEM: S. E. S.
* DATE WRITTEN: MARCH 6, 1982
*--------------------------------------------------------------*/
* PURPOSE: to allow a task to receive a message from a mailbox if
* eligible.
*--------------------------------------------------------------*/
* FUNCTION:
* I. Disable interrupts for a user task
* that the system call is executed atomically.
* II. Check the mailbox pointer to see if it is valid and
* that the task is eligible to receive a message from it.
* III. If the mailbox pointer is valid then,
* A. Check the mailbox message queue for a pending message.
* 1. If the pending message is from the right task,
* a. Then if it is not broadcast and it is to the right
* task, the message is found.
* b. If the message is broadcast and the caller is a
* valid recipient task, the message is found.
* B. If the message is not found, and the task is not willing to
* wait, set the return code.
* C. Otherwise:
* 1. If the caller is a user task, suspend it for a while
* by putting the task in the timer queue and deleting it
* from the ready queue.
* 2. If after the activation, the message is not sent yet:
* a. Set the return code.
* 3. Otherwise,
* a. Get the address of the message.
* D. If the message is found:
* 1. Copy the message header to the user header area.
* 2. Set the return code.
* 3. If the message is not broadcast, or it is broadcast and
* this task was the last recipient task,
* a. Remove the message from the mailbox.
* b. If there is a task waiting for a message header,
* wake it up.
* c. Otherwise, return it to the pool.
* IV. Restore the interrupts for a user task.
*/
/* PARAMETERS PASSED: */  
1. mbx: Mailbox pointer from which the message is expected.  
2. from$who: Task from whom the message is expected.  
3. user$head$base: The pointer to user provided buffer area.  
4. wait: The time that the task is willing to wait, if there is no message pending in the mailbox.  
5. rt$code: The pointer to the return code word.  

/* LOCAL VARIABLES: */  
1. receive$status: The return code word.  
2. copy: The boolean flag indicating whether the message was found, either after a wait or the first time.  
3. found: The boolean flag indicating whether a message was found in the first pass.  
4. RESOURCE GRANTED: A literal indicating whether the message was sent during the wait period.  
5. user$head: The structure based on the user header:  
   length Length of the message.  
   response Response indication field.  
   data$ptr Pointer to the actual message.  

DECLARE (rt$code, from$who ) POINTER;  
DECLARE (mbx, user$head$base) POINTER;  
DECLARE wait WORD;  
DECLARE receive$status BASED rt$code WORD;  
DECLARE user$head BASED user$head$base STRUCTURE ( length WORD,  
   response WORD,  
   data$ptr POINTER);  
DECLARE copy WORD;  
DECLARE found WORD;  
DECLARE RESOURCE_GRANTED LITERALLY '0';  

/* Disable the interrupts for a user task. */  
/* Check the mailbox pointer to be valid, and that the task is eligible for receipt of the message from the mailbox. */  

IF system_state = user_state THEN  
   CALL DISABLE_INTERRUPTS;  
   mbx_base = mbx;  
   receive$status = CHECK_MBX_ID (mbx, 0 );  
   IF receive$status = OK THEN DO;  
      copy = TRUE;  
      found = FALSE;  
   END;  

END;
sys_head_base = mailbox.mqhead;

/*---------------------------------------------*
 * The parameters are valid. Now check the mailbox message queue *
 * for a message from the expected task. *
 * If a message is pending from the right task, check to see if *
 * the task is the right recipient for a non-broadcast message *
 * and is a valid recipient for a broadcast message. *
 *---------------------------------------------*/

DO WHILE ( sys_head_base <> mbx_base) AND NOT found;
   IF from$who = sys_head.sender THEN DO;
      IF sys_head.bc = 0 THEN DO;
         IF sys_head.receiver = tcb_base THEN
            found = TRUE;
      END;
   ELSE
      IF VALID_BC_TASK( tcb.num ) THEN DO;
         found = TRUE;
         sys_head.bc = sys_head.bc - 1;
      END;
      ENDIF
   END;
   IF NOT found THEN
      sys_head_base = sys_head.next_msg;
   ENDF
END;

/*---------------------------------------------*
 * If the message is not found, and the task is not willing to *
 * wait, set the return code. *
 * If the task is willing to wait, suspend it. *
 *---------------------------------------------*/

IF NOT found THEN DO;
   IF wait = 0 THEN DO;
      receive$status = NOT_WAIT;
      copy = FALSE;
   END;
ELSE DO;
   IF system_state = user_state THEN DO;
      tcb.from_who = from$who;
      receive$status = SUSPEND ( @mailbox.tqhead , wait);
   /*---------------------------------------------*
    * If the activation was due to time out, set the *
    * flag not to copy a message. *
    * Otherwise, get the address of the message and *
    * set the flag to copy the message to TRUE and *
    * set the flag 'from the mailbox' to FALSE. *
    *---------------------------------------------*/
   END;
IF receive$status <> RESOURCE_GRANTED THEN
  copy = FALSE;
ELSE DO;
  sys_head_base = tcb.timer_q_ptrs(2);
tcb.timer_q_ptrs(2) = 0;
tcb.period(1) = 2;
found = FALSE;
copy = TRUE;
END;
ELSE
  receive$status = CANNOT_PREEMPT_RTTS;
ENDIF
END;
END;

/**********************************************************
* If the copying of the message should be done (whether from  *
* the mailbox or from the message header address after the   *
* suspension), set the return code to 0 and copy the information*
* from the system buffer to user buffer.                      *
**********************************************************/

IF copy THEN DO;
  mbx_base = mbx;
  receive$status = OK;
  user/head.length = sys_head.length;
  user/head.response = sys_head.response;
  user/head.data$ptr = sys_head.data$ptr;
  IF sys_head.sender = SES THEN
    receive$status = FROM_SYSTEM;
  END;

/**********************************************************
* NOTE: If the message was not a broadcast message, or     *
* it was a broadcast message AND this task was the last     *
* recipient of the message, the message header should be    *
* returned to the pool.                                    *
**********************************************************/

IF found AND (sys_head.bc = 0) THEN DO;
  mailbox.mqhead = sys_head.next_msg;
  IF mailbox.mqhead = mbx_base THEN
    mailbox.mqtail = mbx_base;
END;
If the above NOTE is applicable, the message header is returned to the pool, if no task is waiting on it. Otherwise, the suspended task is awakened. By setting tcb.period to 5, it is indicated that the task is awakened not because of the time out.

IF sys_head.bc = 0 THEN DO;
    IF buffer_queue.qhead <> @buffer_queue THEN DO;
        task_base = buffer_queue.qhead;
        task.period(1) = 5;
        CALL WAKE_UP( buffer_queue.qhead, @buffer_queue ,sys_head_base, OFFH );
    END;
ELSE
    CALL RETURN_HEADER_TO_POOL( sys_head_base );
ENDIF
END;

IF system_state = user_state THEN
    CALL RESTORE_INTERRUPTS;
END RECEIVE MESSAGE;
END RECEIVE MESSAGE;
The module contains the utility programs to be used for message header pool controller.

**PROCEDURES INCLUDED:**
1. **RETURN HEADER TO POOL:** This procedure returns a free message header to the pool of message headers.
2. **GET HEADER FROM POOL:** This procedure gets a message header and returns the address to the caller.

**PROCEDURES USED BY 'MAIL BOX UTILITIES' MODULE:**
1. **INSERT IN DELAY Q:** Inserts a task in a timer queue.
2. **FIRST IS BIGGER OR EQUAL:** Performs a ' >= ' check for two double words.

```
INSERT_IN_DELAY_QUEUE: PROCEDURE (task$ptr, before$ptr, offset, qcb$ptr) EXTERNAL;
    DECLARE (task$ptr, before$ptr) POINTER;
    DECLARE qcb$ptr POINTER;
    DECLARE offset WORD;
END INSERT_IN_DELAY_QUEUE;

FIRST_IS_BIGGER_OR_EQUAL: PROCEDURE (dword1, dword2) WORD EXTERNAL;
    DECLARE (dword1, dword2) POINTER;
END FIRST_IS_BIGGER_OR_EQUAL;
```

$INCLUDE (:F1:MBXEXT.DEF)
$INCLUDE (:F1:PLMEXT.DEF)
RETURN_HEADER_TO_POOL: PROCEDURE ( free$header ) PUBLIC;

DECLARE free$header POINTER;
    sys_head_base = free$header;
    num_messages = num_messages - 1;

IF header_pool_qcb.qtail = @header_pool_qcb THEN
    header_pool_qcb.qtail = free$header;
    sys_head.next_msg = header_pool_qcb.qhead;
    header_pool_qcb.qhead = free$header;
END RETURN_HEADER_TO_POOL;
GET_HEADER_FROM_POOL: PROCEDURE POINTER PUBLIC;

*---------------------------------------------*
*   NAME: GET HEADER FROM POOL               *
*   SYSTEM: S. E. S.                        *
*   DATE: FEB 18, 1982                      *
*---------------------------------------------*

PURPOSE: This procedure gets a free message header and returns the address to the caller.

FUNCTION:
I. It first increments the number of outstanding messages.
II. It then modifies the forward link of the free header queue.
III. Lastly it modifies the backward link of the free header queue if this was the last message header in the queue.

*---------------------------------------------*
*   PARAMETERS PASSED:                       *
*   none                                      *
*---------------------------------------------*
*   LOCAL VARIABLES USED:                    *
*   none                                      *
*---------------------------------------------*

num_messages = num_messages + 1;

sys_head_base = header_pool_qcb.qhead;

header_pool_qcb.qhead = sys_head.next_msg;
IF sys_head.next_msg = @header_pool_qcb THEN
    header_pool_qcb.qtail = @header_pool_qcb.qcb;

RETURN(sys_head_base);
END GET_HEADER_FROM_POOL;
END MAIL_BOX_UTILITIES;
BROADCAST UTILITIES: DO;

/* ----------------------------------------------- *
 * * NAME: BROADCAST UTILITIES
 * * SYSTEM: S. E. S.
 * * DATE: MARCH 15, 1982
 * *
 * PURPOSE: This module includes the utility routines for a broadcast message and also to check a mailbox ID with regard to specific operation performed (i.e. send, receive or broadcast).

 * Routines included:
 * 1. CHECK MBX ID: This routine checks the mailbox pointer to see if it is valid. Also checks the callers ID with regard to the system call.
 * 2. BUILD BC MAP: This routine builds a bit map in the message header.
 * 3. VALID BC TASK: This routine checks the ID of a task against those in the bit map for a broadcast message.

 * EXTERNAL PROCEDURES USED:
 * none

* -----------------------------------------------*/

$INCLUDE (:F1:MBXEXT.DEF)
$INCLUDE (:F1:PLMEXT.DEF)

DECLARE bmap(16) WORD INITIAL(1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768);
CHECK_MBX_ID: PROCEDURE ( mbx, caller$code ) WORD PUBLIC;

/*
 * NAME: CHECK MBX ID
 * SYSTEM: S. E. S.
 * DATE: MARCH 15, 1982
 *
 * PURPOSE: To check a mailbox pointer. It also checks the mailbox
 * type against the caller task ID and the system call to
 * be performed.
 *
 * FUNCTION:
 * I. Check the mailbox check field to see if it is valid.
 * II. If yes, check the owner1 field of the mailbox to see if it
 *     is defined.
 * III. Check the mailbox type to see if it is a mailbox and not
 *     a semaphore.
 * IV. If any of the above cases is not true, set the return code
 *     to the proper value.
 * V. Otherwise:
 *     A. If the operation to be performed is receive:
 *        1. If the mailbox is private and the caller is none of
 *           the owners, set the return code to ILLEGAL REQUEST.
 *        2. If the mailbox is a semi-private mailbox and the
 *           caller is not the owner, set the return code to
 *           ILLEGAL REQUEST.
 *     B. If the operation to be performed is send:
 *        1. If the mailbox is private and the caller is none of
 *           the owners, set the return code to ILLEGAL REQUEST.
 *     C. If the operation to be performed is broadcast:
 *        1. If the mailbox is not public, set the return code to
 *           ILLEGAL REQUEST.
 * VI. Return with the return code.
 */

DECLARE mbx POINTER;
DECLARE caller$code WORD;
DECLARE mbx$status WORD;

PARAMETERS PASSED:
1. mbx: The pointer to the mailbox to be checked.
2. caller$code: Type of operation to be performed:
   0__ Receive operation.
   1__ Send operation.
   2__ Broadcast operation.

LOCAL VARIABLES USED:
1. mbx$status: Return code word.

---
mbx$status = OK;
mbx_base = mbx;

/* First check the mailbox to see if:
  * 1. It is a valid exchange pointer by the 'check' field.
  * 2. It has an owner by checking the owner field.
  * 3. It is a mailbox and not a semaphore.
  */

IF mailbox.check <> CORRECT_USER_NAME THEN
  mbx$status = INVALID_MAILBOX;
ELSE DO;
  IF mailbox.owner1 = 0 THEN
    mbx$status = MBX_NOT_DEFINED;
  ELSE DO;
    IF mailbox.type = 4 THEN
      mbx$status = MBX_IS_A_SEMAPHORE;
    ELSE DO CASE caller$code;
      DO;
        /* Check the mailbox against the operation to be performed.
        * 1. RECEIVE : If the mailbox is private, the task should be one of the owners,
        * if semi-private it must be the owner.
        * 2. SEND : If the mailbox is private, the task must be one of the owners.
        * 3. BROADCAST : The mailbox must be public.
        */

        IF mailbox.type = PRIVATE AND
          (tcb_base <> mailbox.owner1 AND
           tcb_base <> mailbox.owner2) THEN
          mbx$status = ILLEGAL_REQUEST;
        IF mailbox.type = SEMI_PRIVATE
          AND tcb_base <> mailbox.Owner1 THEN
          mbx$status = ILLEGAL_REQUEST;
      END;

      IF mailbox.type = PRIVATE AND
        (tcb_base <> mailbox.owner1 AND
         tcb_base <> mailbox.owner2) THEN
        mbx$status = ILLEGAL_REQUEST;
      IF mailbox.type <> GENERAL_DEL THEN
        mbx$status = ILLEGAL_REQUEST;
      END;
    END;
  END;
ELSE END;
END;
END;
RETURN (mbx$status);

END CHECK_MBX_ID;
BUILD_BC_MAP: PROCEDURE (map$ptr, no) PUBLIC;

*---------------------------------------------------------------------*

* NAME: BUILD_BC_MAP
* SYSTEM: S. E. S.
* DATE: MARCH 15, 1982
* *
* PURPOSE: To build a bit map for a broadcast message. This map is*
* part of the message header. The maximum number of the*
* recipient tasks for a broadcast message is 64.
* *
* FUNCTION:
* I. Repeat for each task:
* A. Get the task pointer and the task number.
* B. Get the index number for that task by dividing the task*
* number by 16.
* C. Get the offset within the word in the array.
* D. Set that bit by anding it to the proper number.
* II. Return.
* *
*---------------------------------------------------------------------*/

* PARAMETERS PASSED:
* 1. map$ptr: A pointer to the task pointers.
* 2. no: Number of broadcast message recipients.
* *
* LOCAL VARIABLES USED:
* 1. tasks(*): An array based on the tasks pointer, contains*
* a list of individual task pointers.
* 2. i: Index number for loop control.
* 3. num: The offset within the bit map word.
* 4. index: The index number within the bit map array.
* *
*---------------------------------------------------------------------*/

DECLARE map$ptr POINTER;
DECLARE no BYTE;
DECLARE tasks BASED map$ptr(1) POINTER;
DECLARE (i,num,index) WORD;

DO i = 0 TO no - 1;
    task_base = tasks(i);
    num = task.num;
    index = num / 16;
    num = num - index * 16;
    sys_head.bc_tasks(index) = sys_head.bc_tasks(index) OR bmap(num);
END;

END BUILD_BC_MAP;
VALID_BC_TASK: PROCEDURE (task$num) WORD PUBLIC;

/*-----------------------------------------------*/
/* NAME: VALID BC TASK */
/* SYSTEM: S. E. S. */
/* DATE: MARCH 15, 1982 */

/* PURPOSE: To check a task number to see if it is a valid task to */
/* receive a broadcast message. */
/* */
/* FUNCTION: */
/* I. Get the index within the bitmap array. */
/* II. Get the offset within the bitmap word. */
/* III. Check that bit to see if it is set. If set, reset it. */
/* If set, return a true value to the caller. */
/* IV. Otherwise, return false. */
/*-----------------------------------------------*/

DECLARE task$name WORD;
DECLARE (index,num) WORD;

num = task$name;
index = num / 16;
num = num - index * 16;

IF (sys_head.bc_tasks (index) AND bmap(num)) <> 0 THEN DO;
    sys_head.bc_tasks (index) = sys_head.bc_tasks (index) XOR bmap(num);
RETURN (TRUE);
END;
ELSE
RETURN (FALSE);
ENDIF
END VALID_BC_TASK;

END BROADCAST_UTILITIES;
PEREMPTION UTILITIES: DO;

/*-----------------------------------------------*/
*  
*   NAME:       PEREMPTION UTILITIES
*   SYSTEM:     S. E. S.
*   DATE:       MARCH 9, 1982
*  
*-----------------------------------------------*/

PURPOSE: This module provides the routines necessary to suspend
and awaken a task.

1. SUSPEND: Suspend a task for a period of time.
2. WAKE UP: Activates a task by inserting it in the
ready queue and deleting it from the timer queue
and the resource wait queue.

EXTERNAL PROCEDURES USED:
1. FIND PROPER PLACE IN Q: This routine inserts a task in the
   timer queue, which is ordered on the
time after which each of the tasks will
   be activated.
2. DELETE FROM DELAY Q: This routine deletes a task from the
timer queue.
3. INSERT TASK IN RDY Q: This routine inserts a task in the ready
   queue based on its priority.
4. INSERT ENTRY: This routine is a general insert in any
   queue. The place to be inserted is spec-
   ified by the parameters.
5. DELETE ENTRY: This is a general purpose delete from
   any queue routine. It adjusts the back-
   ward and forward pointers of the proper
   entries.
6. DISABLE INTERRUPTS: To disable interrupts in all three CCMs
   to perform the system call atomically.
7. RESTORE INTERRUPTS: To restore the interrupt state for the
caller task.

-----------------------------------------------*/

FIND PROPER PLACE IN Q: PROCEDURE (task, qcb, type )
DECLARE (task, qcb) POINTER;
DECLARE type WORD;
END FIND_PROPER_PLACE_IN_Q;

DELETE FROM DELAY QUEUE: PROCEDURE (task, qcb, offset)
DECLARE (task, qcb) POINTER;
DECLARE offset WORD;
END DELETE FROM DELAY QUEUE;

INSERT TASK IN RDY Q: PROCEDURE (task )
DECLARE task POINTER;
END INSERT_TASK_IN_RDY Q;
INSERT_ENTRY:  PROCEDURE (task, previous$task, status) EXTERNAL;
   DECLARE (task, previous$task) POINTER;
   DECLARE status POINTER;
END INSERT_ENTRY;

DELETE_ENTRY:  PROCEDURE (task, rt_ptr) EXTERNAL;
   DECLARE (task, rt_ptr) POINTER;
END DELETE_ENTRY;

DISABLE_INTERRUPTS:  PROCEDURE EXTERNAL;
END DISABLE_INTERRUPTS;

RESTORE_INTERRUPTS:  PROCEDURE EXTERNAL;
END RESTORE_INTERRUPTS;

$INCLUDE (:F1:PLMEXT.DEF)
$INCLUDE (:F1:MBXEXT.DEF)
SUSPEND:  PROCEDURE (qcb$ptr, wait)  WORD  PUBLIC;

/*-----------------------------------------------------------*/

* NAME:  SUSPEND
* SYSTEM:  S. E. S.
* DATE:  MARCH 9, 1982

*-----------------------------------------------------------*/

PURPOSE:  To suspend a task for a certain period of time.

FUNCTION:
I. Deletes the task from the ready queue.
II. Sets the wait time period in the Task Control Block.
III. Inserts the task in the timer queue based on its wait time.
IV. Inserts the task in the suspension queue related to the
resource on which it is suspended (e.g. mailbox queue).
V. Transfer to the dispatcher.
   By performing a DISABLE INTERRUPTS call.
VI. If after return the tcb.period is 2, it indicates time out.
VII. If tcb.period is a 5 it indicates the resource is granted.
VIII. If tcb.period is a 8 it indicates the resource won't be
available (e.g. the semaphore is deleted).
XI. Return with the proper return code.

PARAMETERS PASSED:
1. qcb$ptr:  The resource queue on which the task is suspended.
2. wait:  The time period that the task is willing to wait
for the resource.

LOCAL VARIABLES:
1. suspend$result:  The word contains the reason for activation
of the task after suspension (e.g. time out or
the resource was granted).
2. status:  A dummy word as a parameter for INSERT ENTRY.
3. qcb:  This structure is the head of the resource queue.
   Qcb.qhead being a pointer to the first task in
   queue, and qcb.qtail being the last task.

DECLARE  (status, wait)  WORD;
DECLARE  suspend$result  WORD;
DECLARE  qcb$ptr  POINTER;
DECLARE  qcb  BASED  qcb$ptr  STRUCTURE
   (qhead  POINTER,
   qtail  POINTER);
/* Delete the task from the ready queue, and setup the wait time in *
* Task Control Block. Then insert the task in the timer queue and *
* resource suspension queue. Preempt by calling DISABLE INTERRUPTS.*
*/

CALL DELETE_ENTRY( tcb_base, @status );
tcb.q_ptr = qcb$ptr;

BEGIN
CALL FIND_PROPER_PLACE_IN_Q ( tcb_base, @timer_qcb, 1);

CALL INSERT_ENTRY ( qcb.qtail, tcb_base, @status);
CALL DISABLE_INTERRUPTS;
CALL RESTORE_INTERRUPTS;

/* After activation, check the reason for activation. *
* 1. tcb.period = 2 : Time out occurred *
* 2. tcb.period = 5 : Resource was granted. *
* 3. tcb.period = 8 : All the related resources were deleted.*
* *
* Return with the proper return code word. *
*/

IF tcb.period(1) = 2 THEN
  suspend$result = TIME_OUT;
ELSE DO;
  IF tcb.period(1) = 5 THEN
    suspend$result = OK;
  ELSE DO;
    IF tcb.period(1) = 8 THEN
      suspend$result = MBX_DELETED;
    END;
  END;
END;
RETURN ( suspend$result);
END SUSPEND;
WAKE_UP:  PROCEDURE (task$ptr, qcb$ptr, resource, preemption ) PUBLIC;

/*------------------------------*/
/*
 * NAME:    WAKE UP
 * SYSTEM:  S. E. S.
 * DATE:    MARCH 9, 1982
 */
/*------------------------------*/

PURPOSE: To awaken a suspended task.

FUNCTION:
I. Sets the wait time in the Task Control Block to 0.
II. Deletes the task from the timer queue.
III. Deletes the task from the resource queue.
IV. Inserts the task in the ready queue.
V. If the resource is addressable, set the address in the TCB.
   A message header buffer is an example of an addressable
   resource and semaphore units are non-addressable resources.
VI. If the new task has a higher priority than the current
    task preempt by performing a DISABLE INTERRUPTS call.

/*------------------------------*/
/* PARAMETERS PASSED:*/
/* 1. task$ptr:       Pointer to the task to be awakened.*/
/* 2. qcb$ptr:       The queue on which the task is suspended.*/
/* 3. preemption:    Whether or not the preemption should be done.*/
/* 4. resource:      0 for non-addressable resource, the address*/
/*                   of the resource for addressable resource.*/
/* */
/* LOCAL VARIABLES:*/
/* 1. status:        Dummy word as a parameter in INSERT ENTRY.*/
/*------------------------------*/

DECLARE (task$ptr, resource ) POINTER;
DECLARE preemption     WORD;
DECLARE qcb$ptr        POINTER;
DECLARE status         WORD;
/** Reset the task wait time in its control block. Then delete the task from the resource queue and the timer queue and insert it in the ready queue. If resource is addressable, set the address of the resource in the Task Control Block. **/

task_base = task$ptr;
task.q_ptr = 0;
task.wait_time(2) = 0;
task.wait_time(3) = 0;

CALL DELETE_FROM_DELAY_QUEUE (task$ptr, @timer_qcb, 16);
CALL DELETE_ENTRY (task$ptr, @status);
CALL INSERT_TASK_IN_RDY_Q (task$ptr);

IF resource <> 0 THEN
    task.timer_q_ptrs(2) = resource;

/ ** Preempt by calling DISABLE INTERRUPTS if: **/
/ 1. Requested by the caller AND **
/ 2. New task has a higher priority than the running task. **/

IF task.priority > tcb.priority AND preemption THEN DO;
    CALL DISABLE_INTERRUPTS;
    CALL RESTORE_INTERRUPTS;
END;

END WAKE_UP;

END PREEMPTION_UTILITIES;
USER_BUFFER_ROUTINES: DO;

/* -----------------------------------------------*/
/*
* NAME: USER BUFFER ROUTINES
* SYSTEM: S. E. S.
* DATE: MARCH 5, 1982
*
* -----------------------------------------------*/

PURPOSE: To allow the user to reuse the message buffers after a
SEND OPERATION.

1. SIGNAL BUFFER: Called by the receiver of a message. Signals
that the message buffer is free and can be reused.
2. WAIT ON BUFFER: Called by the sender of a message. Waits for
the buffer that was already sent, to be freed.

EXTERNAL ROUTINES USED:
1. SUSPEND: To suspend the caller task. If the message buffer
is still in process and the task wishes to wait.
2. WAKE UP: To awaken a suspended task, used by SIGNAL BUFFER.
   When there is a task waiting on the buffer being
   released.
7. DISABLE INTERRUPTS: To disable interrupts simultaneously in
   all three processors.
8. RESTORE INTERRUPTS: To restore the state of the interrupts to
   the incoming state.

-----------------------------------------------*/

WAKE_UP: PROCEDURE (task, qcb, resource, preemption) EXTERNAL;
DECLARE (task, qcb) POINTER;
DECLARE resource POINTER;
DECLARE preemption WORD;
END WAKE_UP;

SUSPEND: PROCEDURE (qcb, wait) WORD EXTERNAL;
DECLARE qcb POINTER;
DECLARE wait WORD;
END SUSPEND;

DISABLE_INTERRUPTS: PROCEDURE EXTERNAL;
END DISABLE_INTERRUPTS;

RESTORE_INTERRUPTS: PROCEDURE EXTERNAL;
END RESTORE_INTERRUPTS;

$INCLUDE (:F1:MBXEXT.DEF)
$INCLUDE (:F1:PLMEXT.DEF)
WAIT_ON_BUFFER: PROCEDURE (buf$address, wait, rt$code) PUBLIC;

/*
* NAME:         WAIT ON BUFFER
* SYSTEM:       S. E. S.
* DATE:         MARCH 5, 1982
*
* PURPOSE: To allow the sender of a task to wait for its buffer
* if not already free.
*
* FUNCTION:
* I. Disable interrupts for a user task.
* II. If the buffer is already free, set the return code.
* III. Otherwise, if the task is not willing to wait, set the
*       return code.
* IV. Otherwise, set the address of the buffer in the Task Control
*      Block and suspend the task.
* V. If after activation the buffer was freed, reset the flag
*      in Task Control Block.
* VI. Restore interrupts for a user task and return.
*/

DECLARE (buf$address, rt$code) POINTER;
DECLARE wait WORD;
DECLARE wait$status BASED rt$code WORD;
DECLARE flag BASED buf$address BYTE;
DECLARE RESOURCE_GRANTED LITERALLY '0';
/* Disable interrupts, and if the buffer is already free, (indicated *
* by the first byte of the buffer being a 0 ), set the return code. *
* Otherwise, suspend the task if it wishes to wait for the buffer *
* to be freed. */

IF system_state = user_state THEN
    CALL DISABLE_INTERRUPTS;
ELSE
    IF flag = 0 THEN
        wait$status = CALL_COMPLETED;
    ELSE DO;
        IF wait = 0 THEN
            wait$status = NOT_WAIT;
        ELSE DO;
            tcb.buffer = buf$address;
            wait$status = SUSPEND ( @data_buffer_qcb, wait );
    END;
END;

IF wait$status = RESOURCE_GRANTED THEN
    tcb.period(1) = 2;
END;

END;

IF system_state = user_state THEN
    CALL RESTORE_INTERRUPTS;

END WAIT_ON_BUFFER;
SIGNAL BUFFER: PROCEDURE ( tname, buf$address, rt$code) PUBLIC;

NAME: SIGNAL BUFFER
SYSTEM: S. E. S.
DATE: MARCH 5, 1982

PURPOSE: This procedure is called by the receiver of a message. After the task is done processing the message, it calls this system call to return the buffer to the sender.

FUNCTION:
I. Disables interrupts for a user task.
II. Check the owner task specified as the parameter, return if not a valid task pointer.
III. Otherwise, if the buffer is already free (the first byte of the buffer is 0), set the return code.
IV. Otherwise, subtract one from the number of the users of the buffer (first byte of the buffer).
V. If the number of users is zero:
   1. If the owner task has requested the buffer,
   2. Call WAKE UP to activate the owner task and perform preemption.
VI. Restore interrupts and return.

PARAMETERS PASSED:
1. tname: The owner of the buffer.
2. buf$address: The pointer to the buffer.
3. rt$code: The return code pointer.

LOCAL VARIABLES USED:
1. signal$status: The return code word.
2. flag: The number of currently users of the buffer.
3. found: Indicates whether or not the owner task has been suspended on the buffer.

DECLARE (tname, rt$code ) POINTER;
DECLARE buf$address POINTER;
DECLARE signal$status BASED rt$code WORD;
DECLARE flag BASED buf$address BYTE;
DECLARE found WORD;
/* Disable interrupts. If the owner task specified by the parameter * 
* is not valid or if the buffer is already free, set the return  
* code. Otherwise, subtract one from the users of the buffer (the * 
* first byte of the buffer) and if 0 (means it is free) see if * 
* the owner task is waiting for the buffer to be freed. */

IF system_state = user_state THEN 
    CALL DISABLE_INTERRUPTS;

    task_base = tname;
    IF task.name_check <> CORRECT_USER_NAME THEN 
        signal$status = INVALID_TSK;
    ELSE 
        DO;
            IF flag = 0 THEN 
                signal$status = ALREADY_FREE;
            ELSE 
                DO;
                    flag = flag - 1;
                    signal$status = CALL_COMPLETED;
                IF flag = 0 THEN DO;
                    found = FALSE;
                    task_base = data_buffer_qcb.qhead;
                END;
            END;
        END;
    END;

/* Search the data_buffer queue to see if the owner* 
* task is waiting on it's buffer. If so, activate * 
* the task by a call to WAKE_UP. * 
* Restore the interrupts and return. */

DO WHILE task_base <> @data_buffer_qcb AND NOT found;
    IF task_base = tname AND 
        buf$address = task.buffer THEN DO;
        task.buffer = 0;
        found = TRUE;
        task.period(1) = 5;
        CALL WAKE_UP (task_base, @data_buffer_qcb, 
            0, OFFH );
    END;
    ELSE 
        task_base = task.rqlnk_for;
    END;
END;

END;
IF system_state = user_state THEN
    CALL RESTORE_INTERRUPTS;
END SIGNAL_BUFFER;
END USER_BUFFER_ROUTINES;
PURGE_EXCHANGE : DO;

/*---------------------------------------------*/
*
* NAME: PURGE EXCHANGE
* SYSTEM: S. E. S.
* DATE: MARCH 15, 1982
*
*---------------------------------------------*/
* PURPOSE: This module provides the routines necessary to purge a *
* mailbox or a semaphore.
*
* ROUTINES INCLUDED:
* 1. PURGE TASKS: Activates all the suspended task for a sema-
*     phone or message.
* 2. PURGE MESSAGE: Discards all the pending messages in a *
*     mailbox.
* 3. PURGE MAILBOX: Discards messages and/or activates all the *
*     waiting tasks as specified by calling the above routines.
* 4. PURGE SEMAPHORE: Activates all the suspended tasks on a sema-
*     phone.
*
* EXTERNAL ROUTINES USED:
* 1. WAKE UP: Activates a task by removing it from the resource and timer queue, and inserting it *
*     in the ready queue.
* 2. DISABLE INTERRUPTS: Disables interrupts simultaneously in all*
*     three processors.
* 3. RESTORE INTERRUPTS: Restores the state of interrupts to to *
*     the incoming state.
*
------------------------------------------------------------------------------------------------

WAKE_UP :PROCEDURE ( task, qcb, resource, pre ) EXTERNAL;
DECLARE ( task, qcb ) POINTER;
DECLARE resource POINTER;
DECLARE pre WORD;
END WAKE_UP;

RETURN_HEADER_TO_POOL: PROCEDURE ( header ) EXTERNAL;
DECLARE header POINTER;
END RETURN_HEADER_TO_POOL;

DISABLE_INTERRUPTS: PROCEDURE EXTERNAL;
END DISABLE_INTERRUPTS;

RESTORE_INTERRUPTS: PROCEDURE EXTERNAL;
END RESTORE_INTERRUPTS;

$INCLUDE (:F1:PLMEXT.DEF)
$INCLUDE (:F1:MBXEXT.DEF)
PURGE_MESSAGES: PROCEDURE;

/* ----------------------------------------------- */
/* */
/* NAME:          PURGE_MESSAGES */
/* SYSTEM:        S. E. S. */
/* DATE:          MARCH 15, 1982 */
/* */
/* ----------------------------------------------- */
/* PURPOSE: To discard all the messages pending in a mailbox by */
/* dequeuing them from the mailbox and returning them to */
/* the pool. */
/* */
/* FUNCTION: */
/* I. While not end of the queue: */
/* A. Get the next message address. */
/* B. Return the message header to the pool. */
/* II. Adjust the mailbox message pointers. */
/* ----------------------------------------------- */

DECLARE next_message POINTER;

sys_head_base = mailbox.mqhead;

/* Get the messages in the mailbox and return them to the pool of */
/* headers. Finally adjust the mailbox message pointers. */
/* ----------------------------------------------- */

DO WHILE sys_head_base <> mbx_base;
    next_message = sys_head.next_msg;
    CALL_RETURN_HEADER_TO_POOL (~sys_head_base);
    sys_head_base = next_message;
END;

mailbox.mqhead = mbx_base;
mailbox.mqtail = mbx_base;

END PURGE_MESSAGES;
PURGE_TASKS: PROCEDURE;

/*-----------------------------------------------*/
/* */
/* NAME: PURGE TASKS */
/* SYSTEM: S. E. S. */
/* DATE: MARCH 15, 1982 */
/* */
/*-----------------------------------------------*/
/* PURPOSE: To activate all the tasks waiting in a mailbox or a */
/* semaphore by calling the WAKE UP routine. */
/* */
/* FUNCTION: */
/* I. Until not end of the queue: */
/* A. Get the next task pointer. */
/* B. Signal that the exchange is deleted by setting the flag */
/* in the TCB. */
/* C. Wake up the task. */
/* II. Perform the preemption. */
/*-----------------------------------------------*/

DO WHILE mailbox.tqhead <> @mailbox.tqhead;
    task_base = mailbox.tqhead;
    task.period(1) = 8;
    CALL WAKE_UP (task_base, @mailbox.tqhead, 0,0); END;
CALL DISABLE_INTERRUPTS;
CALL RESTORE_INTERRUPTS;

END PURGE_TASKS;
PURGE MAILBOX:  PROCEDURE (mbx, code, rt)  PUBLIC;

/*---------------------------------------------*/
/*
* NAME:        PURGE MAILBOX
* SYSTEM:      S. E. S.
* DATE:        MARCH 15, 1982
*
* PURPOSE: To purge the messages and/or tasks in a mailbox.
*
* FUNCTION:
* I. Disable interrupts for a user task.
* II. If the mailbox pointer is invalid, set the return code.
* III. If the mailbox is not defined, set the return code.
* IV. If the caller of the system call is not the owner of the
*      mailbox, set the return code.
* V. If none of the above conditions were true:
*     A. If type specified is 0: purge messages.
*     B. If type specified is 1: purge tasks.
*     C. If type specified is 2: purge both tasks and messages.
* VI. Restore interrupts and return.
*---------------------------------------------*/

/*---------------------------------------------*/
/* PARAMETERS PASSED:*/
/* 1. mbx: The mailbox pointer which is to be purged.*/
/* 2. code: Indicates whether only message/only tasks*/
/* or both messages and tasks to be purged.*/
/* 3. rt: The return code pointer.*/
/*---------------------------------------------*/

LOCAL VARIABLES USED:
* 1. purge$status: The return code word.
*---------------------------------------------*/

DECLARE (mbx, rt)  POINTER;
DECLARE code  WORD;
DECLARE purge$status  BASED  rt  WORD;

IF system_state = user_state THEN
   CALL DISABLE_INTERRUPTS;
   purge$status = CALL_COMPLETED;
   mbx_base = mbx;

/* Check the mailbox pointer, to see if it is a valid pointer, then */
/* check to see if it is a mailbox, then check to see if the owner */
/* of the mailbox is the caller of this system call.*/

IF mailbox.check <> OAAAAH THEN
   purge$status = INVALID_MAILBOX;
ELSE DO;
   IF mailbox.type = 4 THEN
      purge$status = MBX_IS_A_SEMAPHORE;
   ELSE DO;
      IF mailbox.owner1 = 0 THEN
         purge$status = MBX_NOT_DEFINED;
      ELSE DO;
         IF mailbox.owner1 <> tcb_base THEN
            purge$status = NOT_DEFINED;
         ELSE DO;
            IF mailbox.owner1 <> tcb_base THEN
               purge$status = NOT_DEFINED;
            ELSE DO;
               /*
                * If code is illegal, set the return code.
                * If the mbx pointer is a valid mailbox pointer,
                * according to 'TYPE', purge messages and/or tasks.
                */
               IF code < 0 or code > 2 THEN
                  purge$status = ILLEGAL_REQUEST;
               ELSE DO CASE code;
                  CALL PURGE_MESSAGES;
                  CALL PURGE_TASKS;
                  DO;
                     CALL PURGE_MESSAGES;
                     CALL PURGE_TASKS;
                  END;
               END;
            END;
         ELSE DO;
            IF system state = user state THEN
               CALL RESTORE INTERRUPTS;
         END;
      ELSE DO;
      END;
   END;
ELSE DO;
END;
END;
END PURGE_MAILBOX;
PURGE_SEMAPHORE: PROCEDURE (sem$name, rt) PUBLIC;

/*-------------------------------------------------------------*/
/*                   NAME:            PURGE_SEMAPHORE          */
/*                   SYSTEM:            S. E. S.            */
/*                   DATE:            MARCH 15, 1982        */
/*-------------------------------------------------------------*/
/* PURPOSE: to purge the waiting tasks for a semaphore. */
/*-------------------------------------------------------------*/
/* FUNCTION: */
/* I. Disable interrupts for a user task. */
/* II. If the semaphore is an invalid semaphore pointer, set the */
/*      return code. */
/* III. If the caller of the system call is not the semaphore */
/*      owner, set the return code. */
/* IV. Otherwise, remove all the tasks, by calling WAKE UP. */
/* V. Restore interrupts and return. */
/*-------------------------------------------------------------*/
/
/* PARAMETERS PASSED: */
/* 1. sem$name: The semaphore pointer to be purged. */
/* 2. rt: The return code pointer. */
/*-------------------------------------------------------------*/
/
/* LOCAL VARIABLES USED: */
/* 1. purge$status: The return code word. */
/*-------------------------------------------------------------*/

DECLARE (sem$name, rt) POINTER;
DECLARE purge$status BASED rt WORD;

IF system_state = user_state THEN
    CALL DISABLE_INTERRUPTS;

purge$status = OK;
mbx_base = sem$name;

/* If the semaphore pointer is valid, then check to see if the */
/* caller is the owner of the semaphore, if true call PURGE_TASKS. */
/* Otherwise set the return code. */
/*-------------------------------------------------------------*/

IF mailbox.check <> CORRECT_USER_NAME OR mailbox.type <> 4 THEN
    purge$status = INVALID_SEMAPHORE;
ELSE DO;
    IF mailbox.owner1 <> tcb_base THEN
        purge$status = NOT_PRIVILEGED;
    ELSE
        CALL PURGE_TASKS;
END;
IF system_state = user_state THEN
CALL RESTORE_INTERRUPTS;
END PURGE_SEMAPHORE;
END PURGE_EXCHANGE;
SEMAPHORE_OPERATIONS: DO;

/*-----------------------------------------------*/
/*
* NAME: SEMAPHORE_OPERATIONS
* SYSTEM: S. E. S.
* DATE: MARCH 25, 1982
*-----------------------------------------------*/

PURPOSE: To provide the tasks to cooperate when sharing resources to prevent simultaneous access to a resource.

1. SIGNAL: To send semaphore units to a semaphore. This call awakens suspended tasks on the semaphore, if their request can be satisfied.

2. WAIT: Called when requested to access a shared resource. The task waits a period of time if desired, if the semaphore is not available if it wishes.

NOTE: THERE IS NO RESOURCE MANAGER CURRENTLY IMPLEMENTED IN RTTS. THEREFORE, THERE IS NO CHECK MADE TO SEE IF THE TASK RELEASING THE SEMAPHORE IS THE TASK THAT HAD ACQUIRED IT.

EXTERNAL ROUTINES USED:
1. SUSPEND: To suspend the caller task. If the semaphore is not available and the task wishes to wait.
2. WAKE UP: To awaken a suspended task used by SIGNAL, when there is a task waiting on the semaphore being released.
7. DISABLE INTERRUPTS: To disable interrupts simultaneously in all three processors.
8. RESTORE INTERRUPTS: To restore the state of the interrupts to the incoming state.

-----------------------------------------------*/

SUSPEND: PROCEDURE ( qcb, wait ) WORD EXTERNAL;
DECLARE qcb POINTER;
DECLARE wait WORD;
END SUSPEND;

DISABLE_INTERRUPTS: PROCEDURE EXTERNAL;
END DISABLE_INTERRUPTS;

RESTORE_INTERRUPTS: PROCEDURE EXTERNAL;
END RESTORE_INTERRUPTS;

WAKE_UP: PROCEDURE ( task, queue, resource, preempt ) EXTERNAL;
DECLARE (queue, task ) POINTER;
DECLARE preempt WORD;
DECLARE resource POINTER;
END WAKE_UP;

$INCLUDE (:F1:PLMEXT.DEF)
$INCLUDE (:F1:MBXEXT.DEF)

/ * ------------------------------- * /
/ * MODULE GLOBALS USED: *
/ * 1. dummy:  A dummy pointer. same as mailbox.mqtail. *
/ * 2. sem(2): An array used to indicate the maximum number of *
/ *    units and the current number of units for semaphore. *
/ * 3. ini:    Literal used to access the element of the array : '0'
/ * 4. cur:    Literal used to access the element of the array : '1'
/ *-------------------------------*/

DECLARE  dummy     POINTER;
DECLARE  sem       BASED dummy(2)  WORD;
DECLARE  cur       LITERALLY '1';
DECLARE  ini       LITERALLY '0';
**SIGNAL:**   

PROCEDURE ( sem$name, units, rt)   
PUBLIC;

/*
**
**   NAME:     SIGNAL
**   SYSTEM:   S. E. S.
**   DATE:     MARCH 25, 1982
**
**   PURPOSE: To release semaphore units previously acquired, a WAIT
**            system call should have been invoked previously.
**
**   FUNCTION:
**   I. Disable interrupts for a user task.
**   II. Check the semaphore pointer, and set the dummy pointer to
**        point to mailbox.mqtail.
**   III. If the number of sent units + the current units is greater
**        than the initial value, set the return code.
**   IV. Otherwise,
**        A. Set the return code to completed.
**        B. Add units released to the current number of units.
**        C. If there is any suspended task in the semaphore queue,
**           whose request can be satisfied by the released units:
**           1. Wake up the task.
**           NOTE: There is no preemption at this level, since more
**                  than one task may be awakened.
**           2. Decrement the current number of units by the amount
**                  requested by the waiting task.
**           3. Get the next task.
**        D. If there is at least one task awakened with a higher
**            priority than the running task, preempt.
**   V. Restore interrupts and return.
**
*/

/*
**   PARAMETERS PASSED:
**   1. sem$name:    The semaphore pointer.
**   2. units:       The number of released units.
**   3. rt:          The return code pointer.
**
**   LOCAL VARIABLES USED:
**   1. send$status: The return code word.
**
*/

DECLARE (sem$name, rt)   POINTER;
DECLARE units            WORD;
DECLARE send$status       BASED   rt       WORD;
DISABLE INTERRUPTS;
AND task.priority > tcb.priority THEN DO;

    CALL DISABLE_INTERRUPTS;
    CALL RESTORE_INTERRUPTS;
    END;
END;
END;
END;

IF system_state = user_state THEN
    CALL RESTORE_INTERRUPTS;
END SIGNAL;
WAIT: PROCEDURE (sem$name, units, wait, rt) PUBLIC;

/* -----------------------------------------------*/
/* */
/* NAME: WAIT */
/* SYSTEM: S. E. S. */
/* DATE: MARCH 25, 1982 */
/* */
/* PURPOSE: To request semaphore units. the task could suspend if */
/* there is no units available, if it wishes. */
/* FUNCTION: */
/* I. Disable interrupts for a user task. */
/* II. Check the semaphore pointer, and set a dummy pointer to */
/* point to mailbox.mqtail. */
/* III. Check the units requested, to see if it can be satisfied. */
/* (i.e. is less than the initial value). */
/* IV. If there is enough units available, then */
/* A. Decrement the number of units available by number of */
/* units requested. */
/* V. Otherwise: */
/* A. If the task does not wish to wait, set the return code. */
/* B. Otherwise: */
/* 1. Suspend a user task. */
/* NOTE: The return code is set by the SUSPEND call. */
/* VI. Restore interrupts for a user task. */
/* */
/* -----------------------------------------------*/

/ * PARAMETERS PASSED: * /
/* 1. sem$name: The semaphore pointer. */
/* 2. units: The number of requested units. */
/* 3. rt: The return code pointer. */
/* */
/* LOCAL VARIABLES USED: */
/* 1. send$status: The return code word. */
/* */
/* -----------------------------------------------*/

DECLARE (sem$name, rt) POINTER;
DECLARE (wait, units) WORD;
DECLARE rec$status BASED rt WORD;
/* Disable interrupts for a user task then check to see:
* 1. If the semaphore pointer is a valid exchange.
* 2. If the exchange is a semaphore and not a mailbox.
* 3. If the requested units are within range, given the maximum
*    number of units for the semaphore.
* If any of the above is false, set the return code.
*/

IF system_state = user_state THEN;
  CALL DISABLE_INTERRUPTS;

mbx_base = sem$name;
dummy = @mailbox.mqtail;
rec$status = OK;
IF mailbox.check <> CORRECT_USER_NAME THEN
  rec$status = INVALID_SEMAPHORE;
ELSE DO;
  IF mailbox.type <> 4 THEN
    rec$status = SEMAPHORE_IS_MBX;
  ELSE DO;
    IF units > sem(ini) THEN
      rec$status = INVALID_UNITS;
    ELSE DO;
      /* If enough units is available, decrement the number of
      * current units. Other wise, if the task wishes to wait
      * SUSPEND it. The return code is set by the SUSPEND.
      * If the task is not willing to wait, set the return code.
      * Restore interrupts and return.
      */
      IF sem(cur) >= units THEN
        sem(cur) = sem(cur) - units;
      ELSE DO;
        IF wait = 0 THEN
          rec$status = NOT_WAIT;
        ELSE DO;
        tcb.sem_request = units;
        rec$status = SUSPEND (@mailbox.tqhead, wait);
        END;
      END;
    END;
  END;
END;
IF system_state = user_state THEN
  CALL RESTORE_INTERRUPTS;
END WAIT;
END SEMAPHORE_OPERATIONS;