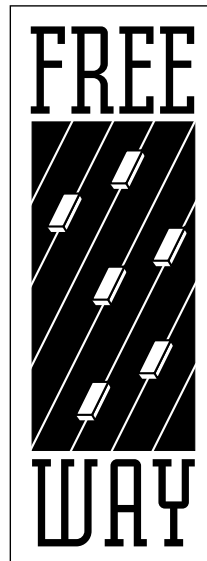


# **Freeway<sup>®</sup> OS/Impact Programmer's Guide**

DC 900-1030B



---

Simpact, Inc.  
9210 Sky Park Court  
San Diego, CA 92123  
January 1995

***SIMPACT***

Simpact, Inc.  
9210 Sky Park Court  
San Diego, CA 92123  
(619) 565-1865

Freeway OS/Impact Programmer's Guide  
© 1993, 1994, 1995 Simpact, Inc. All rights reserved  
Printed in the United States of America

This document can change without notice. Simpact Associates Inc. accepts no liability for any errors this document might contain.

Ethernet is a trademark of Xerox Corporation.  
Freeway is a registered trademark of Simpact, Incorporated.  
VMEbus is a trademark of Motorola Incorporated.  
VxWorks is a trademark of Wind River Systems Incorporated.

---

# Contents

<b>Preface</b>	<b>11</b>
<b>1 Overview</b>	<b>15</b>
1.1 Freeway Overview . . . . .	15
1.2 OS/Impact Overview . . . . .	18
<b>2 Principles of Operation</b>	<b>21</b>
2.1 System Scheduling. . . . .	21
2.2 Tasks . . . . .	22
2.3 Queues . . . . .	22
2.3.1 Singly Linked Queues . . . . .	22
2.3.2 Doubly Linked Queues. . . . .	23
2.3.3 Exchange Queues. . . . .	23
2.3.4 Queue Ordering . . . . .	24
2.4 Memory Allocation . . . . .	24
2.4.1 Stacks . . . . .	24
2.4.2 Partitions . . . . .	25
2.5 Timer Services . . . . .	25
2.5.1 Timer Interrupt Service Routine (ISR) . . . . .	26
2.5.2 Timer Task . . . . .	26
2.5.3 Alarms. . . . .	26
2.6 Resource Management . . . . .	27
<b>3 System Calls</b>	<b>31</b>
3.1 Task-Related Calls . . . . .	32
3.1.1 Create a Task (s_tcreat). . . . .	34
3.1.2 Delete Calling Task (s_tdelet) . . . . .	36

3.1.3	Disable Task Rescheduling (s_lock)	37
3.1.4	Enable Task Rescheduling (s_unlock)	38
3.1.5	Suspend Calling Task (s_susp)	39
3.1.6	Resume a Task (s_resum)	41
3.2	Queue-Related Calls	42
3.2.1	Create a Queue (s_qcreat)	43
3.2.2	Delete a Queue (s_qdelet)	45
3.2.3	Post a Message to a Queue (s_post)	46
3.2.4	Post a Message and Resume Queue Owner (s_postr)	48
3.2.5	Accept a Message from a Queue (s_acpt)	50
3.3	Resource-Related Calls	51
3.3.1	Create a Resource (s_rcreat)	52
3.3.2	Delete a Resource (s_rdelet)	54
3.3.3	Request a Resource (s_rreq)	55
3.3.4	Cancel a Resource Request (s_rcan)	57
3.3.5	Release a Resource (s_rrel)	59
3.4	Partition-Related Calls	61
3.4.1	Create a Partition (s_pcreat)	62
3.4.2	Delete a Partition (s_pdelet)	64
3.4.3	Request a Buffer (s_breq)	65
3.4.4	Release a Buffer (s_brel)	66
3.5	Alarm-Related Calls	67
3.5.1	Create an Alarm (s_acreat)	68
3.5.2	Delete an Alarm (s_adelet)	71
3.5.3	Set an Alarm (s_aset)	72
3.5.4	Cancel an Alarm (s_acan)	74
3.6	Miscellaneous Calls	75
3.6.1	Initialize OS (s_osinit)	76
3.6.2	Get System Address Table (s_getsat)	77
3.6.3	Return from ISR (s_iret)	78
3.6.4	Set Interrupt Level (s_iset)	79
<b>4</b>	<b>System Data Structures</b>	<b>81</b>
4.1	Task Control Block (TCB)	81
4.2	Queue Control Block (QCB)	83
4.3	Resource Control Block (RCB)	84

4.4	Partition Control Block (PCB)	85
4.5	Alarm Control Block (ACB)	86
4.6	Standard Alarm Queue	88
4.7	Special Alarm Queue	89
4.8	Dispatch Queues	90
4.9	Configuration Table	91
4.10	Task Initialization Structure (TIS)	94
4.11	Event Control Block (ECB)	95
4.12	System Buffer Header (SBH)	97
4.13	Resource Carrier Message	100
4.14	Stack Format	102
4.15	Global System Table (GST)	103
<b>A</b>	<b>Debugging Aids</b>	<b>105</b>
A.1	Global System Table	105
A.1.1	Panic Codes	106
A.2	System Variables	115
A.2.1	Pointers to Control Structures	115
A.2.2	Alarm Queues	116
A.2.3	Task Execution Variables	116
<b>B</b>	<b>Data Structure Field Offsets</b>	<b>117</b>
<b>C</b>	<b>System Call Summaries</b>	<b>123</b>
<b>D</b>	<b>Task Scheduling Examples</b>	<b>127</b>
	<b>Glossary of Acronyms</b>	<b>131</b>
	<b>Index</b>	<b>133</b>



---



# List of Figures

Figure 1-1: Freeway Configuration. . . . . 16





---

# List of Tables

Table A-1:	System Errors . . . . .	106
Table A-2:	Control Structures . . . . .	115
Table A-3:	Alarm Queues . . . . .	116
Table A-4:	Task Execution Variables . . . . .	116
Table B-1:	Task Control Block (TCB) . . . . .	118
Table B-2:	Queue Control Block (QCB) . . . . .	118
Table B-3:	Resource Control Block (RCB) . . . . .	118
Table B-4:	Partition Control Block (PCB) . . . . .	119
Table B-5:	Alarm Control Block (ACB) . . . . .	119
Table B-6:	Standard Alarm Queue . . . . .	119
Table B-7:	Special Alarm Queue . . . . .	120
Table B-8:	Dispatch Queues . . . . .	120
Table B-9:	Configuration Table . . . . .	120
Table B-10:	Task Initialization Structure (TIS) . . . . .	121
Table B-11:	Event Control Block (ECB) . . . . .	121
Table B-12:	System Buffer Header (SBH) . . . . .	121
Table B-13:	Resource Carrier Message . . . . .	122
Table B-14:	Global System Table (GST) . . . . .	122
Table C-1:	C Interface System Call Summary . . . . .	124
Table C-2:	Assembly Interface System Call Summary . . . . .	125



---



# Preface

## **Purpose of Document**

This document provides a complete description of the programmer's interface to OS/Impact, Simpact's real-time operating system kernel for the Motorola 68xxx family of processors.

## **Intended Audience**

This document should be read by programmers who are developing code that will be downloaded to a Simpact product and execute in the OS/Impact environment. You should be familiar with general operating-system concepts and with the fundamentals of developing programs in a real-time environment. Some familiarity with the C programming language is helpful because this document presents the system-call interface and data-structure definitions in C format.

## **Required Equipment**

You will need OS/Impact in executable form and a Freeway product based on Motorola's 68xxx family of processors. The software you develop and intend to execute in Simpact's OS/Impact environment must be assembled and linked into a file executable on the Motorola chip. You must also be able to download the software to the Simpact product.

## **Organization of Document**

[Chapter 1](#) provides an overview of Freeway and OS/Impact.

[Chapter 2](#) introduces the components of OS/Impact and provides a general description of its system services.

[Chapter 3](#) contains the OS/Impact system calls. Each section describes a system call in detail and provides input and output parameters for C and assembly language interfaces.

[Chapter 4](#) describes the system data structures used internally by OS/Impact and those used by the programmer to interface to OS/Impact.

[Appendix A](#) provides information for debugging software that executes in the OS/Impact environment, including system error codes and instructions for locating system data structures.

[Appendix B](#) explicitly defines the system data structures in terms of field sizes and byte offsets.

[Appendix C](#) provides one-page system-call summaries for C and assembly language interfaces.

[Appendix D](#) provides some examples of task scheduling.

The Glossary of Acronyms lists the acronyms used in this manual.

## References

### Freeway general support:

- *Freeway 1000 Hardware Installation Guide* DC 900-1315
- *Freeway 2000/4000 Hardware Installation Guide* DC 900-1331
- *Freeway 2000/4000 Hardware Maintenance Guide* DC 900-1332
- *Freeway Programmable Communications Servers Technical Overview* 25-000-0374
- *Freeway User's Guide* DC 900-1333
- *Getting Started with Freeway 1000* DC 900-1314
- *Getting Started with Freeway 2000/4000* DC 900-1330

### Freeway programming support:

- *Freeway Client-Server Interface Control Document* DC 900-1303
- *Freeway Data Link Interface Reference Guide* DC 900-1334
- *Freeway ICP6000 Hardware Description* DC 900-1020
- *Freeway OS/Impact Programmer's Guide* DC 900-1030
- *Freeway Transport Subsystem Interface Reference Guide* DC 900-1335
- *ICP2424 Hardware Description and Theory of Operation* DC 900-1328

### Freeway protocol support:

- *Freeway ADCCP NRM Programmer's Guide* DC 900-1317
- *Freeway Asynchronous Wire Service (AWS) Programmer's Guide* DC 900-1324
- *Freeway BSC Programmer's Guide* DC 900-1340
- *Freeway FMP Programmer's Guide* DC 900-1339
- *Freeway HDLC Low-Level Interface* DC 900-1352
- *Freeway Marketfeed 2000 Programmer's Guide* DC 900-1418
- *Freeway Protocol Software Toolkit Programmer's Guide* DC 900-1338
- *Freeway SWIFT and CHIPS Programmer's Guide* DC 900-1344
- *Freeway X.25 Low-Level Interface* DC 900-1307

## **Customer Support**

If you are having trouble with any Simpack product, call us at 1-800-275-3889 Monday through Friday between 8 a.m. and 5 p.m. Pacific time.

You can also fax your questions to us at (619) 560-2838 or (619) 560-2837 any time. Please include a cover sheet addressed to "Customer Service."

We are always interested in suggestions for improving our products. You can use the report form in the back of this manual to send us your recommendations.

## 1.1 Freeway Overview

Simpact's Freeway communications servers enable client systems on a local-area network (LAN) to access specialized wide-area networks (WANs). The Freeway server is user programmable and communicates in real time. It provides multiple data links and a variety of network services to LAN-based clients in financial, defense, telecommunications, and process-control applications. [Figure 1-1](#) shows the Freeway configuration.

To maintain high data throughput, Freeway uses a multi-processor architecture to support the LAN and WAN services. The LAN interface is managed by a single-board computer, called the server processor. It uses the commercially available VxWorks operating system to provide a full-featured base for the LAN interface and layered services needed by Freeway.

Freeway can be configured with multiple WAN interface processor boards, each of which is a Simpact-series intelligent communications processor (ICP). Each ICP runs the communication protocol software using Simpact's OS/Impact real-time operating system.

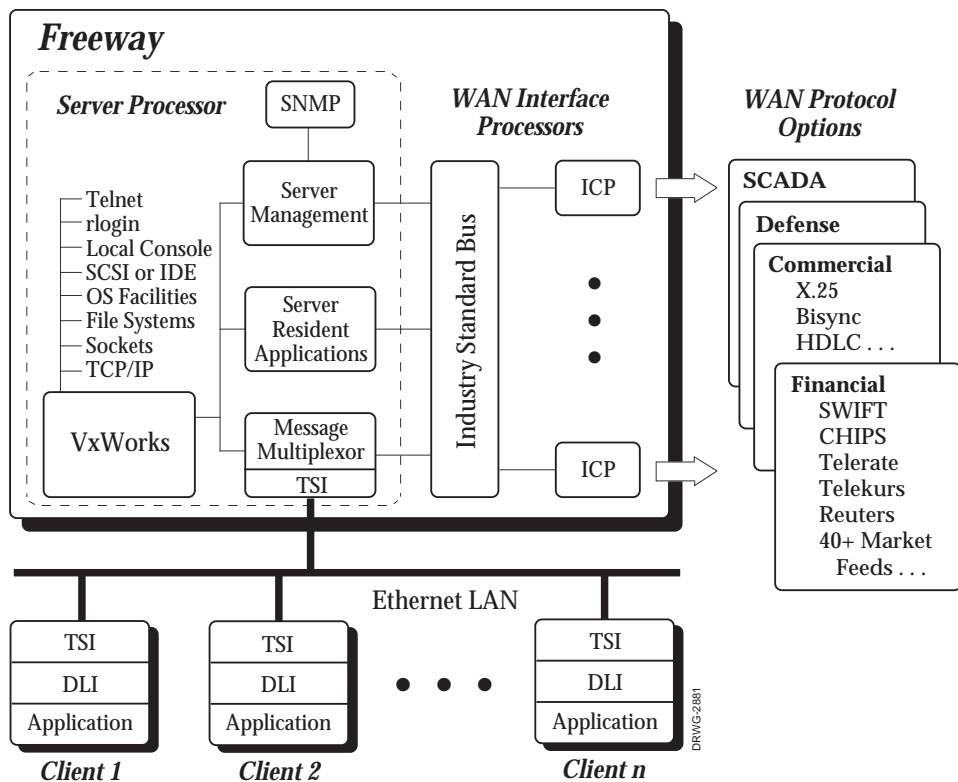


Figure 1-1: Freeway Configuration



Other Freeway features include:

- Standard support for Ethernet LANs running the transmission control protocol/internet protocol (TCP/IP)
- Standard support for FDDI LANs running the transmission control protocol/internet protocol (TCP/IP)
- Support for multiple ICPs (two, four, eight, or sixteen communication lines per ICP)
- Wide selection of electrical interfaces including EIA-232, EIA-449, EIA-485, EIA-530, EIA-562, V.35, ISO-4903 (V.11), and MIL-188
- Server management and performance monitoring with the simple network management protocol (SNMP), as well as interactive menus available through a web browser, local console, telnet, or rlogin
- Variety of off-the-shelf communication protocols available from Simpack
- Support for multiple WAN communication protocols simultaneously on a single Freeway server
- Elimination of difficult LAN and WAN programming and systems integration by providing a data link interface (DLI) to access Freeway services through the transport subsystem interface (TSI)
- Creation of customized server-resident and ICP-resident software, using Simpack's software development toolkits

## 1.2 OS/Impact Overview

The operating system that runs on the ICP is Simpack's OS/Impact real-time executive, which is optimized for communications applications. Customers of Simpack's commercial off-the-shelf protocols need no knowledge of OS/Impact, but system integrators developing customized Freeway applications to run on the ICP need to understand how to use OS/Impact, as explained in this manual.

OS/Impact is a real-time operating system kernel designed for Motorola's 68xxx family of processors. In the real-time environment, actions are generally interrupt-driven and many operations proceed logically in parallel. To support this environment, OS/Impact provides a multi-tasking system with priorities and optional time slicing.

The following system components are defined:

<b>Tasks</b>	Entities that execute logically in parallel
<b>Queues</b>	Linked lists used for intertask communication and for ordering events
<b>Alarms</b>	Used to schedule timed activities
<b>Partitions</b>	Blocks of memory divided into buffers
<b>Resources</b>	Used to allocate or control access to objects

The system environment is completely configurable and dynamic: the maximum number of each component is configurable at system initialization, and instances of each can be dynamically created (up to the maximum) and deleted during system operation.

OS/Impact includes 28 system service routines, which are accessed through the exception vector table using software interrupts (the 68xxx TRAP instruction). The addresses of the system routines, therefore, need not be available to application tasks, as they would if the routines were accessed with ordinary subroutine calls. This allows the ker-

nel to be maintained as a stand-alone module, with no link to application programs required.

As an operating system kernel, OS/Impact is only one part of what might be considered a complete operating system. For example, device drivers and host I/O services are not included in the kernel, but are provided with it as components of the software toolkit. Depending on the user's requirements, any additional application- or hardware-specific facilities can be added as operating system extensions (using additional software interrupt vectors) or at the task level.



## 2.1 System Scheduling

In a multi-tasking operating system, processes execute logically in parallel. In actuality, only one task executes at a time, but tasks are “rescheduled” periodically; that is, execution of the current task is suspended and another task is dispatched. The rules for system scheduling are as follows:

1. The task that is executing is suspended (preempted) when:
  - it is time sliced (has been executing for the maximum allowable time period);
  - it terminates;
  - it requests suspension;
  - a higher-priority task is scheduled for execution; or
  - if time slicing is enabled, a task at the same priority is scheduled for execution.
2. When one of these events occurs, tasks are rescheduled. At this time, the task to be dispatched (executed) is chosen from a group of tasks that are scheduled (waiting) to execute.
3. A task is scheduled to execute when it is created (by another task with the Create Task system call), when it is preempted (except when it has requested suspension

or has terminated), or as a result of a Resume or Post & Resume system call. A task can be both executing and scheduled for execution. This can occur as a result of a Resume or Post & Resume system call made by the task itself or by an interrupt service routine during the task's execution.

4. Of the tasks that are scheduled to execute, the system always dispatches the task with the highest priority. If more than one task is at that priority, the task that was scheduled first (has been waiting the longest) is dispatched.

## 2.2 Tasks

The maximum number of tasks in the system is configurable at system initialization. For each configured task, the system creates a task control block, which is defined in [Section 4.1 on page 81](#). Tasks are created and deleted dynamically through the Create Task and Delete Task system calls. Each task is identified by a task identification, supplied as an input parameter to the Create Task system call.

## 2.3 Queues

As defined for this operating system, a queue is a linked list which can contain any number of queue elements. A particular queue can be singly linked (forward pointers only), or doubly linked (forward and backward pointers). Any data structure containing a standard system buffer header, as defined in [Section 4.12 on page 97](#), can be used as a queue element.

### 2.3.1 Singly Linked Queues

A singly linked queue consists of a head pointer, a tail pointer and the “next element” field in the system buffer header of each queue element. The head pointer contains the address of the first queue element. The tail pointer contains the address of the last queue element. The “next element” field of each buffer header contains the address of the next

queue element, except for the last, which contains zero. If the queue is empty, the head pointer contains zero, and the value of the tail pointer is undefined.

### 2.3.2 Doubly Linked Queues

A doubly linked queue consists of a head pointer, a tail pointer, and the “next element” and “previous element” fields in the buffer header of each queue element. The head pointer contains the address of the first queue element. The tail pointer contains the address of the last queue element. The “next element” field of each buffer header contains the address of the next queue element, except for the last, which contains the address of the first queue element. The “previous element” field of each buffer header contains the address of the previous queue element, except for the first, which contains the address of the last queue element. If the queue is empty, the head pointer contains zero, and the value of the tail pointer is undefined.

### 2.3.3 Exchange Queues

The operating system uses both singly and doubly linked queues for various purposes internally and, in addition, you can define a configurable number of exchange queues for intertask communication. ([Chapter 4](#) includes descriptions of internal queues.)

The number of exchange queues created by the system during its initialization is a configurable parameter. A queue control block (QCB) is maintained for each configured exchange queue. Exchange queues are used for intertask communication, and a task can also use them for internal message storage or for communication with an interrupt service routine.

A task can own any number of exchange queues, which are obtained and released by means of the Create Queue and Delete Queue system calls. At creation, the ID of the task that will own the queue is specified, and also whether queue elements are to be singly or doubly linked. Each exchange queue is identified by a queue ID, which is also supplied as an input parameter to the Create Queue system call.

Any task can post a message (add an element) to or accept a message (remove an element) from an exchange queue. When a message is posted to a queue, its owner (and only its owner) can optionally be scheduled to execute. The Post Message, Post & Resume, and Accept Message system calls perform these functions.

### **2.3.4 Queue Ordering**

In general, exchange queues, whether singly or doubly linked, are intended to be managed as FIFOs; elements are added to the tail and removed from the head. However, the Post Message and Post & Resume system calls optionally allow a message to be added to the head of a queue.

Double links allow elements to be easily inserted or deleted at any position of a queue. This can be useful when, for example, the queue is ordered according to the value of a field within each queue element. The system uses internal doubly linked queues in this manner, but system calls are not provided to perform these application-specific operations on exchange queues.

## **2.4 Memory Allocation**

The operating system requires a fixed amount of memory for code, data, and stack, and also a variable amount for data structures, based on the system configuration. The memory used by tasks in the system is not controlled or even monitored by the operating system. The user is expected to coordinate the allocation of memory for code, data, stack space, and memory partitions (for system buffers) to prevent conflict with the operating system or among the various tasks.

### **2.4.1 Stacks**

The Motorola 68xxx family of processors define three stack pointers: interrupt, master, and user. A user stack space must be allocated for each task, and the initial stack pointer value must be supplied to the system when the task is created. The system allocates



space for a single interrupt stack and sets the interrupt stack pointer at system initialization. The master stack pointer is not used.

When a task is dispatched, its user stack pointer becomes active and is used for normal stack operations. When an interrupt occurs or a task makes a system call (enters supervisor state through a TRAP instruction), the interrupt stack is activated. Interrupt and system processing, therefore, do not add to user stack size requirements, except that space (66 bytes) must be available on each task's user stack to save its data and address registers, program counter, and status register while the task is not executing (see [Section 4.14 on page 102](#)).

### 2.4.2 Partitions

A partition is a block of memory that is subdivided into buffers of a particular size. The maximum number of partitions that can be created is configurable at system initialization. Partitions are created and deleted dynamically through the Create Partition and Delete Partition system calls. Each partition is identified by an ID, which is supplied as an input parameter to the Create Partition system call.

The system maintains a partition control block (PCB) for each configured partition. The format of the PCB is described in [Section 4.4 on page 85](#) and includes a linked list (a singly linked queue) of available buffers for that partition. A task requests a system buffer of a particular size by specifying (as an input parameter to the Request Buffer system call) the ID of a partition that contains buffers of that size.

## 2.5 Timer Services

Three basic timer services are supplied by the system:

1. A task can suspend with a timer set (sleep) for some number of ticks, after which it is scheduled for execution.

2. A task can set an alarm for some number of ticks; when the timer expires, it can set a flag and asynchronously call a signal routine to the task's execution.
3. If time slicing is enabled for a task, its execution is suspended after the time slice period has expired, and it is rescheduled immediately.

### **2.5.1 Timer Interrupt Service Routine (ISR)**

Timer services are supplied by a combination of interrupt- and task-level code. The Timer ISR is entered on interrupt from the system clock. The ISR decrements a system tick count set by the Timer task and, if it has expired, schedules the Timer task for execution. If time slicing is enabled for the currently executing task, the time slice tick count is decremented. If it has expired, the task is suspended, rescheduled, and another (or the same) task is dispatched.

### **2.5.2 Timer Task**

The Timer task is closely linked to the operating system in that it is scheduled for execution as part of the Set Timer and Suspend system calls (and also by the Timer ISR, as noted previously). It is also the only task in the system that has direct access to certain system data structures. It is responsible for decrementing the tick counts of running alarms and processing alarm expirations. The system tick count decremented by the Timer ISR is set by the Timer task to the minimum number of ticks remaining for a running alarm.

### **2.5.3 Alarms**

The maximum number of alarms in the system is a configurable parameter. A task can own any number of alarms, which it obtains and releases by means of the Create Alarm and Delete Alarm system calls. The system maintains an alarm control block (ACB) for each configured alarm, described in [Section 4.4 on page 85](#). Each alarm is identified by an alarm ID, supplied by the task as an input parameter to the Create Alarm system call.

When an alarm is created, the caller can specify a *standard* or *special* alarm type. The methods used by the Timer task to process the two types of alarms are slightly different. Depending on the intended use of a particular alarm and the number of alarms in the system, either a standard or a special alarm might be more efficient.

Less processing is required for a special alarm than for a standard alarm when it is started or canceled or when its tick count is adjusted; however, less processing is required to decrement the tick counts of running standard alarms than running special alarms. This is because a relative tick count is maintained for each alarm, and only the count for the alarm with the fewest remaining ticks must be decremented. An absolute tick count is maintained for each running special alarm, so that the count must be decremented for each alarm, and the processing required is proportional to the number of alarms running.

In general, therefore, the majority of alarms in the system should be standard; however, for a limited number of alarms that are started, canceled, or adjusted frequently (especially where these operations are performed from an interrupt service routine), the special alarm type might be more efficient.

A third *task* alarm type is used internally by the system. A task alarm is associated with each task in the system. When a task is suspended with a timer set, its alarm is set exactly as a standard alarm would be. When the alarm expires, the task is scheduled for execution. If the task is scheduled for some other reason, its alarm is canceled. Task and standard alarms are treated identically by the Timer task.

## 2.6 Resource Management

Tasks and interrupt service routines might require temporary exclusive access to or possession of certain resources. The operating system provides services to manage the allocation of these resources.

The simplest resource is a single entity — it can be obtained (locked) or released (unlocked). A more complex resource consists of a group of equivalent components; a request for the resource results in the allocation of one of its components, with the assumption that possession of any one of the components is acceptable to the requester. The resource management services do not distinguish between the two types of resources; a resource simply consists of one or more components.

Each component of a resource is represented by a “resource token.” Each resource token is a queue element and must contain a standard system buffer header. Aside from the header, the queue element contents, if any, are transparent to the system.

The maximum number of resources that can be created is configurable at system initialization. (Any number of components can be associated with each resource.) Resources are created and deleted dynamically through the Create Resource and Delete Resource system calls. Each resource is identified by an ID, which is supplied as an input parameter to the Create Resource system call. The system maintains a resource control block (RCB) for each configured resource. The format of the RCB is described in [Section 4.3 on page 84](#) and includes a linked list (a queue) of resource tokens.

After creating a resource, a task generally “stocks” the resource, by building and “releasing” (with a Release Resource system call) the appropriate number of resource tokens, which are then linked by the system to the RCB queue.

A task requests a resource by specifying (as an input parameter to the Request Resource system call) the resource identification. The system unlinks a resource token from the RCB queue and returns its address to the requester. If no tokens are available, an error is returned instead.

In anticipation of the resource being unavailable, the requester can supply the address of a “resource carrier” message on input to the Request Resource call. A resource carrier is a queue element that has a standard system buffer header followed by a number of

fields, including a return queue identification. The format of the carrier message is defined in [Section 4.13 on page 100](#).

If a resource carrier is supplied, and if no tokens are available, the carrier is linked to the RCB queue. (Note that the queue can contain tokens or carriers, but never both.) Depending on a parameter specified when the resource was created, carriers are linked to the queue in order of priority (based on a field in the carrier message) or to the tail of the queue.

When a resource token becomes available (on a Release Resource system call), the resource carrier at the head of the RCB queue is unlinked and the address of the resource token is stored in a field of the carrier message, which is then posted to the appropriate queue (as specified in the message).

If no carrier messages are queued when a resource is released, the resource token is linked to the RCB queue and becomes available to the next requester.



All requests for system services are made through a single vector in the exception vector table, using the TRAP instruction. A particular system call is specified by a function code passed in a register, which the system call trap handler converts to the address of the appropriate routine. The two exceptions to this rule are the Set Interrupt Level and the Return from ISR system calls which, for speed, are accessed directly through separate vectors. Input and output parameters are passed in registers. Programs written in a high-level language can access the system calls through a subroutine interface library.

Before returning control to the task when a system call (including those accessed through separate vectors) finishes, the calling task is preempted and another task is dispatched if all of the following are true:

- The call returns to user state (to the task level)
- Task rescheduling is not disabled
- Another task at a higher priority than the currently executing task is scheduled; or if time slicing is enabled, another task at the same priority is scheduled

Each of the following sections describes the function of a system call, the C and assembly language interfaces with their input and output parameters, and the access restrictions (in other words, whether the call can be made from a task, an interrupt service routine, or both). [Chapter 4](#) and [Appendix B](#) describe the data structures used by the system calls. [Appendix C](#) contains quick-reference tables.

### 3.1 Task-Related Calls

This section describes the system calls that are related to the management of tasks:

- Create a Task (`s_creat`)
- Delete Calling Task (`s_tdelete`)
- Disable Task Rescheduling (`s_lock`)
- Enable Task Rescheduling (`s_unlock`)
- Suspend Calling Task (`s_susp`)
- Resume a Task (`s_resum`)

Tasks can be created as part of system initialization, reinitialization (see [Section 3.6.1 on page 76](#)), or by another task with the Create Task system call. When a task is created, a task control block (TCB) is allocated according to its task ID, and the task is scheduled for execution.

After a task is executing, it can normally be preempted by a higher-priority task or by an interrupt service routine. A task can use the Disable Task Rescheduling and Enable Task Rescheduling system calls around critical sections of code to prevent processing being preempted by another task. If interrupts must also be blocked, a task can use the Set Interrupt Level system call instead (see [Section 3.6.4 on page 79](#)).

Tasks or interrupt service routines can use the Resume system call to schedule any task for execution.

Unless a task is executing at the lowest priority in the system, it must periodically suspend its execution so that the system can be rescheduled, giving lower-priority tasks an opportunity to execute. The Suspend system call is used for this purpose and always suspends the calling task; one task cannot suspend execution of another.



In some cases, a task may no longer be required after it has performed the specific function for which it was created. An example of this might be a “boot” task, which creates the tasks, partitions, queues, and so on that will be used during normal operations. In a case such as this, when the task has completed its processing, it can issue a Delete Task system call, which deallocates its task ID and associated TCB. This routine always deletes the calling task; one task cannot delete another.

### 3.1.1 Create a Task (s\_tcreat)

Given the address of a task initialization structure, the Create Task system call dynamically creates a task and schedules it for execution. The task initialization structure provides the task ID, priority, starting address, stack pointer and a time slice enable/disable flag. Its format is described in [Section 4.10 on page 94](#).

The TCB corresponding to the requested ID is allocated and initialized, and the task is added to the dispatch queue for its priority (in other words, is scheduled for execution). An error is returned if a task with the requested ID already exists (in other words, the TCB is allocated) or if the ID or the priority is out of range. Valid task IDs and priorities are dependent on the system configuration, as described in [Section 4.9 on page 91](#).

C Interface:

```
s_tcreat ( tis )  
struct TIS_TYPE *tis;
```

tis:            pointer to task initialization structure

return:        0x00 = good  
               0x01 = task ID is out of range  
               0x02 = task already exists  
               0x03 = priority is out of range

Assembly Interface:

TRAP            #0

input:            D0.L = 0x00  
                  A0.L = address of task initialization structure

output:           D0.L = completion status  
                  0x00 = good  
                  0x01 = task ID is out of range  
                  0x02 = task already exists  
                  0x03 = priority is out of range

Access: task only

### **3.1.2 Delete Calling Task (s\_tdelet)**

The Delete Task system call deactivates the calling task and frees the associated TCB. If task rescheduling is disabled, it is automatically re-enabled. If the task is currently scheduled for execution, it is removed from the dispatch queue. Control is passed to the dispatcher, and the routine does not return to the caller.

C Interface:

s\_tdelet ()

return: none; does not return

Assembly Interface:

TRAP #0

input: D0.L = 0x01

output: none; does not return

Access: task only

### 3.1.3 Disable Task Rescheduling (s\_lock)

The Lock Task system call disables task rescheduling, including time slicing. Until task rescheduling is re-enabled, the executing task will not be preempted by any other task, regardless of its priority. The Delete Task and Suspend calls automatically re-enable task rescheduling.

C Interface:

s\_lock ()

return: none

Assembly Interface:

TRAP #0

input: D0.L = 0x02

output: none

Access: task only

### 3.1.4 Enable Task Rescheduling (s\_unlock)

The Unlock Task system call enables task rescheduling. If a task makes a Suspend or Delete Task system call with task rescheduling disabled, rescheduling is automatically re-enabled and this call is not necessary.

C Interface:

```
s_unlock ()
```

```
return:      none
```

Assembly Interface:

```
TRAP        #0
```

```
input:      D0.L = 0x03
```

```
output:     none
```

Access: task only

### 3.1.5 Suspend Calling Task (`s_susp`)

The Suspend system call suspends execution of the calling task until an event occurs that causes the task to be rescheduled. By creating an event control block (ECB) and supplying its address as an input parameter to the routine, the calling task can selectively enable rescheduling on any or all of the following events: timer expiration, Resume system call, or a Post & Resume call specifying a queue owned by the task and listed in the ECB. In addition, the calling task can request to be removed from the dispatch queue if it is already scheduled at the time of suspension. The ECB format is defined in [Section 4.11 on page 95](#).

If the task does not supply an ECB address (in other words, the input parameter is zero), all events are enabled, except that no alarm is set. The task will be scheduled to execute on a Resume call or when a Post & Resume call is made with any of its queues specified. If the task is already scheduled at the time of suspension, it remains scheduled.

If task rescheduling is disabled, it is automatically re-enabled when a task suspends.

If an ECB is supplied, and it contains the ID of an exchange queue that the task does not own, an error is returned and the task is not suspended.

If an ECB is supplied, and it contains a nonzero alarm tick count, the alarm associated with the task's TCB is started by linking the ACB to the tail of the special alarm queue and scheduling the Timer task for execution. (The ACB will be moved to the standard alarm queue by the Timer task.) The signal routine address field of all task ACBs contains the address of a special system subroutine. When the alarm expires, the Timer task calls this subroutine, which schedules the associated (suspended) task for execution. If the task is scheduled for some other reason before this occurs, the alarm is canceled.

On return from suspension, the event code signifies the event that caused the task to be rescheduled. The high-order word of the event code signifies the event type: previous scheduling, timer expiration, Resume, or Post & Resume. If the event type signifies Post & Resume, the low-order word of the event code contains the queue ID; otherwise, the

low-order word is zero. Note that the event code signifies only the event that caused the task to be rescheduled; additional enabled events might have occurred since that time.

C Interface:

```
s_susp ( ecb, event_code )
```

```
struct ECB_TYPE *ecb;
```

```
int *event_code;
```

ecb:            pointer to ECB or zero to enable all events

event\_code:    if nonzero, event code is stored at this address on good return

0x010000 = previous scheduling

0x020000 = timer expiration

0x030000 = resume

0x04nnnn = post & resume (nnnn = queue ID)

Assembly Interface:

TRAP            #0

input:          D0.L = 0x04

A0.L = ECB address, or zero to enable all events

output:        D0.L = completion status

0x00 = good

0x01 = invalid ID in ECB

D1.L = event code (if good completion)

0x010000 = previous scheduling

0x020000 = timer expiration

0x030000 = resume

0x04nnnn = post & resume (nnnn = queue ID)

Access: task only



### 3.1.6 Resume a Task (s\_resum)

The Resume system call schedules the specified task for execution if it is currently executing or suspended with Resume enabled (see [Section 3.1.5 on page 39](#)).

An error is returned if the specified task ID is invalid; otherwise, the completion status is good, whether or not the call actually caused the task to be scheduled for execution.

C Interface:

```
s_resum ( task_id )
unsigned short task_id;

task_id:    task ID

return:     0x00 = good
            0x01 = invalid task ID
```

Assembly Interface:

```
TRAP      #0

input:    D0.L = 0x06
          D1.W = task ID

output:   D0.L = completion status
          0x00 = good
          0x01 = invalid task ID
```

Access: task or ISR

## 3.2 Queue-Related Calls

The system calls that are related to the management of exchange queues are as follows:

- Create a Queue (`s_qcreat`)
- Delete a Queue (`s_qdelete`)
- Post a Message to a Queue (`s_post`)
- Post a Message and Resume Queue Owner (`s_postr`)
- Accept a Message from a Queue (`s_accpt`)

An exchange queue can be created by a task with the Create Queue system call, which allocates a QCB according to the queue identification. When the queue is created, it must be designated as either singly or doubly linked. In addition, a task ID must be specified as the “owner” of the queue.

After a queue has been created, tasks and interrupt service routines can add messages to it with Post Message system calls and remove messages with Accept Message calls. Messages can be added either to the head or the tail of the queue but are always removed from the head. If messages are always added to the tail, then the queue acts as a FIFO (first in, first out). If messages are always added to the head, then the queue acts as a LIFO (last in, first out).

Because it is often convenient to notify the owner of a queue when a message has been added, the Post & Resume system call is provided, which is identical to the Post Message system call except that it also causes the task that owns the queue to be scheduled for execution.

If a queue is no longer required, the Delete Queue system call deallocates the queue ID and associated QCB.

### 3.2.1 Create a Queue (s\_qcreat)

The Create Queue system call causes the QCB associated with the specified queue ID to be allocated. As an input parameter, the caller must specify whether elements on the queue are to be singly or doubly linked (see [Section 2.3 on page 22](#)).

The caller must also specify the ID of a task that will become the “owner” of the queue. This task will be scheduled for execution when a Post & Resume call adds a message to the queue, unless it is suspended with that event disabled (see [Section 3.1.5 on page 39](#)).

An error is returned if the queue is already allocated or if the task or queue ID is out of range. Valid task and queue IDs are dependent on the system configuration, as described in [Section 4.9 on page 91](#). On a good completion, the QCB address is returned. Under controlled circumstances this address can be used to implement more sophisticated queue manipulation operations than those provided by the operating system kernel.

C Interface:

```
s_qcreat ( queue_id, q_type, task_id, qcb )  
unsigned short queue_id, q_type, task_id;  
struct QCB_TYPE *qcb;
```

queue\_id:      queue ID

q\_type:        0x00 = single links  
              0x01 = double links

task\_id:       task ID of queue owner

qcb:            if nonzero, pointer to QCB is stored at this address on good completion

return:       0x00 = good  
              0x01 = queue ID out of range  
              0x02 = queue not available  
              0x03 = task ID out of range

Assembly Interface:

TRAP         #0

input:       D0.L = 0x07  
              D1.W = queue ID  
              D2.W = single/double link flag  
                  0x00 = single links  
                  0x01 = double links  
              D3.W = task ID of queue owner

output:      D0.L = completion status  
                  0x00 = good  
                  0x01 = queue ID out of range  
                  0x02 = queue not available  
                  0x03 = task ID out of range  
              A0.L = QCB address if good completion, else unchanged

Access: task only

### 3.2.2 Delete a Queue (s\_qdelet)

The Delete Queue system call frees the QCB associated with the specified queue ID.

An error is returned and the QCB is not deallocated if the caller does not own the queue, if the queue is not empty, or if the queue ID is invalid.

C Interface:

```
s_qdelet ( queue_id )
unsigned short queue_id;

queue_id:    queue ID

return:      0x00 = good
             0x01 = invalid queue ID
             0x02 = queue is not empty
```

Assembly Interface:

```
TRAP        #0

input:      D0.L = 0x08
            D1.W = queue ID

output:     D0.L = completion status
            0x00 = good
            0x01 = invalid queue ID
            0x02 = queue is not empty
```

Access: task only

### 3.2.3 Post a Message to a Queue (s\_post)

The Post Message system call adds a message (a queue element) to the head or tail of the specified queue. The queue element must begin with a standard system buffer header, as defined in [Section 4.12 on page 97](#).

An error is returned if the specified queue is not currently allocated to any task or if the “this element” field of the system buffer header does not contain the queue element address.

C Interface:

```
s_post ( queue_id, head_tail, message )
```

```
unsigned short queue_id, head_tail;
```

```
struct SBH_TYPE *message;
```

queue\_id:      queue ID

head\_tail:     0x00 = tail  
                 0x01 = head

message:       pointer to message

return:        0x00 = good  
                 0x01 = invalid queue ID  
                 0x02 = “this element” invalid

Assembly Interface:

TRAP            #0

input:           D0.L = 0x09  
                  D1.W = queue ID  
                  D2.W = head/tail flag  
                          0x00 = tail  
                          0x01 = head  
                  A0.L = queue element address

output:           D0.L = completion status  
                          0x00 = good  
                          0x01 = invalid queue ID  
                          0x02 = "this element" invalid

Access: task or ISR

### 3.2.4 Post a Message and Resume Queue Owner (s\_postr)

The Post & Resume system call is identical to the Post Message call ([Section 3.2.3 on page 46](#)) except that the task that owns the queue will be scheduled for execution unless it is suspended with the event disabled (see [Section 3.1.5 on page 39](#)).

The completion status returned does not indicate whether the call actually caused the task to be scheduled.

C Interface:

```
s_postr ( queue_id, head_tail, message )  
unsigned short queue_id, head_tail;  
struct SBH_TYPE *message;
```

queue_id:	queue ID
head_tail:	0x00 = tail 0x01 = head
message:	pointer to message
return:	0x00 = good 0x01 = invalid queue ID 0x02 = "this element" invalid



Assembly Interface:

TRAP            #0

input:            D0.L = 0x09  
                  D1.W = queue ID  
                  D2.W = head/tail flag  
                          0x00 = tail  
                          0x01 = head  
                  A0.L = queue element address

output:            D0.L = completion status  
                          0x00 = good  
                          0x01 = invalid queue ID  
                          0x02 = "this element" invalid

Access: task or ISR

### 3.2.5 Accept a Message from a Queue (s\_accpt)

The Accept Message system call removes a message from the head of the specified queue and returns its address.

An error is returned if the queue ID is invalid or if the queue is empty.

C Interface:

```
s_accpt ( queue_id, message )
unsigned short queue_id;
struct SBH_TYPE *message;
```

queue\_id:      queue ID

message:      pointer to message is stored at this address on good return

return:        0x00 = good  
               0x01 = invalid queue ID  
               0xFF = queue is empty

Assembly Interface:

TRAP            #0

input:          D0.L = 0x0B  
                 D1.W = queue ID

output:         D0.L = completion status  
                 0x00 = good  
                 0x01 = invalid queue ID  
                 0xFF = queue is empty  
                 A0.L = queue element address if good completion,  
                 else unchanged

Access: task or ISR

### 3.3 Resource-Related Calls

The following system calls related to the management of resources are described in this section:

- Create a resource (`s_rcreat`)
- Delete a resource (`s_rdelete`)
- Request a resource (`s_rreq`)
- Cancel a resource request (`s_rcan`)
- Release a resource (`s_rrel`)

A resource can be created by a task with the Create Resource system call, which allocates an RCB according to the resource identification. The caller must specify either a priority or a FIFO resource, to determine the order in which resource tokens will be allocated. After creating a resource, the task will generally “stock” it with one or more resource tokens, each representing an available component of the resource.

Tasks and interrupt service routines can then obtain resource tokens with Request Resource system calls and release tokens with Release Resource calls. When requesting a resource, the caller can optionally provide the address of a resource carrier message. If no resource token is available, the carrier message is added to the RCB queue (in FIFO or priority order, depending on the type of resource). When a token becomes available, it is attached to the carrier, and the carrier is posted to a queue specified by the requester. A Cancel Resource Request system call can be cancel a resource request that is waiting to be processed. If this happens, the carrier message is returned with no token attached.

If a resource is no longer required, the Delete Resource system call deallocates the resource ID and associated RCB and posts all available tokens or queued carrier messages to the appropriate queues.

### 3.3.1 Create a Resource (s\_rcreat)

The Create a Resource system call causes allocation of the RCB associated with the specified resource identification. The “priority/FIFO flag” input parameter signifies the order in which carrier messages are to be queued when no resources are available. For a priority resource, the carrier messages are queued according to the priority field in the message (see [Section 4.12 on page 97](#)). For a FIFO resource, carrier messages are queued in the order that they are received.

An error is returned if the resource is already allocated or if the resource ID is out of range. Valid resource IDs are dependent on the system configuration, as described in [Section 4.9 on page 91](#).

To initialize the resource, this call should be followed with one or more calls to Release Resource (see [Section 3.3.5 on page 59](#)).

C Interface:

```
s_rcreat ( res_id, res_type )
unsigned short res_id, res_type;

res_id:      resource ID

res_type:    0x00 = FIFO
             0x01 = priority

return:      0x00 = good
             0x01 = resource ID out of range
             0x02 = resource not available
```

Assembly Interface:

TRAP            #0

input:           D0.L = 0x0C  
                  D1.W = resource ID  
                  D2.W = priority/FIFO flag  
                      0x00 = FIFO  
                      0x01 = priority

output:           D0.L = completion status  
                      0x00 = good  
                      0x01 = resource ID out of range  
                      0x02 = resource not available

Access: task only

### 3.3.2 Delete a Resource (s\_rdelet)

The Delete Resource system call frees the RCB associated with the specified resource identification. A queue ID must be specified as an input parameter, to which any resource tokens currently queued will be posted. If, instead, one or more resource carrier messages are queued, each message is posted to the appropriate queue, as specified in the message, with a completion code signifying that the requested resource has been deleted.

C Interface:

```
s_rdelet ( res_id, queue_id )
unsigned short res_id, queue_id;

res_id:      resource ID

queue_id:    queue ID

return:      0x00 = good
             0x01 = invalid resource ID
             0x02 = invalid queue ID
```

Assembly Interface:

```
TRAP        #0

input:      D0.L = 0x0D
            D1.W = resource ID
            D2.W = queue ID

output:     D0.L = completion status
            0x00 = good
            0x01 = invalid resource ID
            0x02 = invalid queue ID
```

Access: task only

### 3.3.3 Request a Resource (`s_rreq`)

The Request Resource system call unlinks a resource token from the appropriate RCB (as specified by the resource ID) and returns its address. If no resource tokens are available, an error is returned to the caller.

The caller can optionally supply the address of a resource carrier message. The format of this message (a queue element) is defined in [Section 4.13 on page 100](#). If a resource token is available, this parameter is not used; otherwise, if a carrier message address is supplied, the message is added to the RCB queue. For a priority resource, it is inserted into the queue according to the priority specified in the message (which does not necessarily correspond to task priority). For a FIFO resource, it is linked to the tail of the queue.

When a resource becomes available (in other words, is released with a Release Resource system call) and no higher-priority or longer-waiting message is queued (depending on the type of resource), the address of the resource token is stored in a field of the carrier message, the completion code in the message is set to signify that the resource has been allocated, and the message is posted to the specified queue.

C Interface:

```
s_rreq ( res_id, carrier, token )
unsigned short res_id;
struct RC_TYPE *carrier;
struct SBH_TYPE *token;
```

`res_id`: resource ID

`carrier`: pointer to resource carrier message, or zero

`token`: pointer to resource token is stored at this address on good return (token can be a pointer to a user-defined structure type)

return:        0x00 = good  
                 0x01 = invalid resource ID  
                 0x02 = invalid queue ID in carrier message  
                 0x03 = "this element" field of carrier message invalid  
                 0xFF = no resource tokens available

Assembly Interface:

TRAP            #0

input:          D0.L = 0x0E  
                 D1.W = resource ID  
                 A0.L = resource carrier message address, or zero

output:        D0.L = completion status  
                 0x00 = good  
                 0x01 = invalid resource ID  
                 0x02 = invalid queue ID  
                 0x03 = "this element" field of carrier message invalid  
                 0xFF = no resource tokens available  
                 A0.L = resource token address (if good completion)

Access: task or ISR



### 3.3.4 Cancel a Resource Request (s\_rcan)

The Cancel Resource Request system call searches the queue of the appropriate RCB (as signified by the resource ID) for the specified carrier message address. If the message is found, it is unlinked from the RCB queue, the completion code field is set to signify a canceled request, and the message is posted to the appropriate queue (as specified in the message). If the message is not found on the RCB queue, no action is taken.

Note that the completion code in the returned carrier message should be checked to verify that the request was actually canceled, because the resource might already have been allocated when the request was received. If the completion code signifies that the request was canceled, no further action is required, but if the resource was allocated, the token must be released with a Release Resource system call (see [Section 3.3.5 on page 59](#)).

C Interface:

```
s_rcan ( res_id, carrier )  
unsigned short res_id;  
struct RC_TYPE *carrier;
```

res\_id:        resource ID

carrier:       pointer to resource carrier message

return        0x00 = good

              0x01 = invalid resource ID

Assembly Interface:

TRAP           #0

input:           D0.L = 0x0F  
                  D1.W = resource ID  
                  A0.L = resource carrier message address

output:          D0.L = completion status  
                  0x00 = good  
                  0x01 = invalid resource ID

Access: task or ISR

### 3.3.5 Release a Resource (s\_rrel)

The Release Resource system call returns a resource token to the RCB associated with the specified resource identification. If one or more carrier messages are attached to the RCB queue, the following events occur:

- The message at the head of the queue is unlinked.
- The address of the resource token is stored in a field of the message.
- The completion code is set to signify that the resource has been allocated.
- The carrier message is posted to the appropriate queue, as specified in the message.

If the RCB queue is empty or already contains one or more resource tokens, the released token is linked to the tail of the queue.

This call is used not only to release a previously allocated resource, but also by the creator of a resource to initially “stock” the RCB queue with one or more resource tokens.

C Interface:

```
s_rrel ( res_id, token )  
signed short res_id;  
struct SBH_TYPE *token;
```

res\_id:           resource ID

token:            pointer to resource token message

return:           0x00 = good

                  0x01 = invalid resource ID

                  0x02 = “this element” field of token message invalid

Assembly Interface:

TRAP            #0

input:            D0.L = 0x10  
                  D1.W = resource ID  
                  A0.L = resource token address

output:           D0.L = completion status  
                  0x00 = good  
                  0x01 = invalid resource ID  
                  0x02 = "this element" field of token message invalid

Access: task or ISR

### 3.4 Partition-Related Calls

The following system calls are related to the management of partitions:

- Create a Partition (`s_pcreat`)
- Delete a Partition (`s_pdelet`)
- Request a Buffer (`s_breq`)
- Release a Buffer (`s_brel`)

By providing its address range and buffer size, a task can create a partition with the Create Partition system call. A PCB is allocated according to the partition ID, and a linked list of buffers is created.

After a partition has been created, tasks and interrupt service routines can obtain and release buffers with the Request Buffer and Release Buffer system calls. If the partition is empty when a buffer is requested, an error is returned. If a task must wait for a buffer (by suspending) rather than repeating the request until it is successful, then it might be more convenient to maintain free buffers on an exchange queue or as resource tokens.

If a partition is no longer required, the Delete Partition system call deallocates the partition ID and associated PCB. The memory included in the partition becomes available for use again.

### 3.4.1 Create a Partition (s\_pcreat)

The Create Partition system call allocates and initializes the PCB associated with the specified partition identification. The specified block of memory is divided into buffers of the requested size, and all buffers are linked to the partition's free list (see [Section 4.4 on page 85](#)).

An error is returned if the partition ID is already assigned, is out of range, or the size of the partition does not allow at least one buffer to be created. Valid partition IDs are dependent on the system configuration, as described in [Section 4.9 on page 91](#).

The buffer size specified must include the size of the system buffer header (see [Section 4.12 on page 97](#)). For example, a buffer size of 100 would include the 24-byte header and 76 bytes of user data space.

If the partition size is not an even multiple of the buffer size, a number of bytes (smaller than the buffer size) will be unused at the end of the partition. The system does not check the availability of the memory included in the partition. The user is responsible for assuring that the partition will not conflict with code, data, or the memory range of other partitions.

C Interface:

```
s_pcreat ( part_id, buffer_size, start_addr, end_addr, pcb )
unsigned short part_id;
int buffer_size, start_addr, end_addr;
struct PCB_TYPE *pcb;
```

part\_id:        partition ID

buffer\_size:    size of each buffer

start\_addr:    address of start of partition

end\_addr:       address of end of partition + 1

pcb: if nonzero, pointer to PCB is stored at this address on good completion

return: 0x00 = good  
0x01 = partition ID out of range  
0x02 = partition not available  
0x03 = invalid partition size  
0x04 = invalid buffer size

Assembly Interface:

TRAP #0

input: D0.L = 0x11  
D1.W = partition ID  
D2.L = buffer size  
A0.L = starting address  
A1.L = ending address + 1

output: D0.L = completion status  
0x00 = good  
0x01 = partition ID out of range  
0x02 = partition not available  
0x03 = invalid partition size  
0x04 = invalid buffer size  
A0.L = PCB address if good completion, else unchanged

Access: task only

### 3.4.2 Delete a Partition (s\_pdelet)

The Delete Partition system call frees the PCB associated with the specified partition identification.

An error is returned and the partition is not deleted if any buffers are currently allocated (not linked to the free list).

C Interface:

```
s_pdelet ( part_id )
```

```
unsigned short part_id;
```

```
part_id:    partition ID
```

```
return:     0x00 = good
```

```
            0x01 = invalid partition ID
```

```
            0x02 = one or more buffers currently allocated
```

Assembly Interface:

```
TRAP        #0
```

```
input:      D0.L = 0x12
```

```
            D1.W = partition ID
```

```
output:     D0.L = completion status
```

```
            0x00 = good
```

```
            0x01 = invalid partition ID
```

```
            0x02 = one or more buffers currently allocated
```

Access: task only



### 3.4.3 Request a Buffer (s\_breq)

The Request Buffer system call unlinks a system buffer from the free list of the specified partition and returns the address of the buffer. Other than the “partition ID” and “this element” fields of the system buffer header, the contents of the buffer are unspecified.

An error is returned if the partition is not allocated or if its free list is empty.

C Interface:

```
s_breq ( part_id, buffer )
unsigned short part_id;
struct SBH_TYPE *buffer;
```

part\_id:        partition ID

buffer:        pointer to buffer on good return

return:        0x00 = good  
               0x01 = invalid partition ID  
               0xFF = no buffers available

Assembly Interface:

TRAP        #0

input:        D0.L = 0x13  
               D1.W = partition ID

output:        D0.L = completion status  
               0x00 = good  
               0x01 = invalid partition ID  
               0xFF = no buffers available  
               A0.L = buffer address if good completion, else unchanged

Access: task or ISR

### 3.4.4 Release a Buffer (s\_brel)

The Release Buffer system call returns a system buffer to the free list of its partition.

An error is returned if the address of the buffer is out of range for the partition, if the partition ID in the system buffer header is invalid, or if the “this element” field of the system buffer header is not equal to the address of the buffer. The assumption is that the buffer has been allocated and is not already linked to the free list.

C Interface:

```
s_brel ( buffer )
struct SBH_TYPE *buffer;
```

buffer:        pointer to buffer

return:        0x00 = good  
              0x01 = invalid partition ID  
              0x02 = “this element” field invalid  
              0x03 = buffer address out of range for partition

Assembly Interface:

```
TRAP        #0
```

input:        D0.L = 0x14  
              A0.L = buffer address

output:        D0.L = completion status  
              0x00 = good  
              0x01 = invalid partition ID  
              0x02 = “this element” field invalid  
              0x03 = buffer address out of range for partition

Access: task or ISR

### 3.5 Alarm-Related Calls

This section describes the system calls related to the management of standard and special alarms:

- Create an Alarm (`s_acreat`)
- Delete an Alarm (`s_adel`)
- Set an Alarm (`s_aset`)
- Cancel an Alarm (`s_acan`)

A task can create an alarm using the Create Alarm system call, which allocates an ACB according to the alarm identification. The caller must specify the type (standard or special) and, as an option, might provide a flag or signal-routine address and a mask value for the flag.

After an alarm has been created, tasks and interrupt service routines can start the alarm, using the Set Alarm system call, for a specified number of ticks. After the alarm is started, it can be canceled with the Cancel Alarm system call.

When an alarm expires, the flag (if any) is set as specified by the mask, and the signal routine (if any) is called.

If an alarm is no longer required, the Delete Alarm system call deallocates the alarm ID and associated ACB.

### 3.5.1 Create an Alarm (s\_acreat)

The Create Alarm system call allocates and initializes the ACB associated with the specified alarm identification. On input, the calling task must specify either a standard or special alarm type (described in [Section 2.5.3 on page 26](#)).

The address of a signal routine is an optional input parameter. This routine is called by the Timer task after the alarm has been set and its ticks have expired. The Timer task passes the alarm ID as an input parameter to the signal routine in the low-order word of register D0. In addition, the address of a 32-bit flag and a bit mask can be specified. When the alarm expires, the Timer task sets the specified bits in the flag. If neither a signal routine address nor a flag address is specified, no action is taken on alarm expiration.

The environment of the signal routine is as follows:

- A signal routine is a subroutine (and is not an interrupt service routine). It is accessed by means of a subroutine call made directly from the Timer task; therefore, it executes in the context of the highest priority task in the system and will not be preempted by another task. The processor executes in user state and the interrupt mask level is zero (all interrupts enabled).
- The signal routine is called with the ID of the expired alarm in the low-order word of register D0. Registers D0 through D7 and A0 through A6 can be modified and need not be restored.
- With the obvious exception of Suspend, a signal routine can make any system call valid for task-level access, including Set Alarm to restart the expired alarm.

Create Alarm returns an error if the ACB is already assigned or the alarm ID is out of range. Valid alarm IDs are dependent on the system configuration, as described in [Section 4.9 on page 91](#). On good completion, the ACB address is returned, which can be used to modify the signal routine address, flag address, or bit mask before setting the alarm.

C Interface:

```
s_acreat ( alarm_id, alarm_type, mask, signal, flag, acb )  
unsigned short alarm_id, alarm_type;  
int mask;  
int *signal();  
int *flag;  
struct ACB_TYPE *acb;
```

alarm\_id: alarm ID

alarm\_type: 0x00 = standard  
0x01 = special

mask: bit mask for flag, or zero

signal: pointer to signal routine, or zero

flag: pointer to 32-bit flag, or zero

acb: if nonzero, pointer to ACB is stored at this address on good completion

return: 0x00 = good  
0x01 = alarm ID out of range  
0x02 = alarm not available

Assembly Interface:

TRAP            #0

input:            D0.L = 0x15  
                  D1.W = alarm ID  
                  D2.W = alarm type  
                      0x00 = standard  
                      0x01 = special  
                  D3.L = bit mask for flag, or zero  
                  A0.L = address of signal routine, or zero  
                  A1.L = address of flag, or zero

output:            D0.L = completion status  
                      0x00 = good  
                      0x01 = alarm ID out of range  
                      0x02 = alarm not available  
                  A0.L = ACB address if good completion, else unchanged

Access: task only

### 3.5.2 Delete an Alarm (s\_adelet)

The Delete Alarm system call frees the ACB associated with the specified alarm identification. If the alarm is running, it is canceled.

C Interface:

```
s_adelet ( alarm_id )  
unsigned short alarm_id;
```

alarm\_id:     alarm ID

return:       0x00 = good  
              0x01 = invalid alarm ID

Assembly Interface:

TRAP         #0

input:       D0.L = 0x16  
              D1.W = alarm ID

output:      D0.L = completion status  
              0x00 = good  
              0x01 = invalid alarm ID

Access: task only

### 3.5.3 Set an Alarm (s\_aset)

The Set Alarm system call starts an alarm or adjusts the tick count in an alarm that is already running. The ACB associated with the specified alarm ID is linked to the tail of the special alarm queue (if it is not already linked to the queue), the absolute requested tick count is stored in the ACB, and the Timer task is scheduled for execution. If a tick count greater than 32767 is specified, 32767 is used instead (no error is returned). The actual duration of the alarm varies between the number of ticks specified and that number plus one.

For a standard alarm, the Timer task, when it executes, moves the ACB to the appropriate position in the standard alarm queue based on the requested tick count, which is converted to a relative value at that time (see [Section 4.6 on page 88](#)).

When the alarm ticks have expired, the Timer task unlinks the ACB from the standard or special alarm queue. If a nonzero flag address and bit mask are stored in the ACB, the Timer task sets the specified bits in the flag.

If a nonzero signal routine address is stored in the ACB, the Timer task makes a subroutine call to that address, with the alarm ID in the low-order word of register D0.

C Interface:

```
s_aset ( alarm_id, ticks )
unsigned short alarm_id;
short ticks

alarm_id:    alarm ID

ticks:      number of ticks (1-0x7FFF)

return:     0x00 = good
            0x01 = invalid alarm ID
```



Assembly Interface:

TRAP	#0
input:	D0.L = 0x17 D1.W = alarm ID D2.W = number of ticks (1-0x7FFF)
output:	D0.L = completion status 0x00 = good 0x01 = invalid alarm ID

Access: task or ISR

### 3.5.4 Cancel an Alarm (s\_acan)

If the alarm associated with the specified alarm ID is running, the Cancel Alarm system call causes the ACB to be unlinked from the standard alarm queue or marked canceled on the special alarm queue. Canceled ACBs are removed from the special alarm queue by the Timer task.

If Cancel Alarm is called when the alarm is not running, no action is taken.

C Interface:

```
s_acan ( alarm_id )
unsigned short alarm_id;

alarm_id:    alarm ID

return:      0x00 = good
             0x01 = invalid alarm ID
```

Assembly Interface:

```
TRAP        #0

input:      D0.L = 0x18
            D1.W = alarm ID

output:     D0.L = completion status
            0x00 = good
            0x01 = invalid alarm ID
```

Access: task or ISR

### 3.6 Miscellaneous Calls

This section describes the remaining system calls:

- Initialize OS (`s_osinit`)
- Get System Address Table (`s_getsat`)
- Return from ISR (`s_i ret`)
- Set Interrupt Level (`s_i set`)

A task or interrupt service routine can reinitialize the operating system using the Initialize OS system call.

The Get System Address Table system call can be used by a task or interrupt service routine to obtain the addresses of the current configuration table, the exception vector table, and the global system table.

Interrupt service routines must complete their processing with a Return from ISR system call rather than executing an RTE instruction.

Because tasks execute in user state, they cannot directly modify the processor's interrupt mask level. The Set Interrupt Level system call is provided for this purpose.

### 3.6.1 Initialize OS (s\_osinit)

Given the address of a table containing the configurable system parameters (see [Section 4.9 on page 91](#)), the Initialize OS system call causes reinitialization of the operating system. If the input parameter is equal to zero, the current configuration table is used.

This call does not return.

C Interface:

```
s_osinit ( config )  
struct CFP_TYPE *config;
```

config: address of configuration table, or zero if current table is to be used

return: none; does not return

Assembly Interface:

TRAP #0

input: D0.L = 0x19  
A0.L = address of configuration table, or zero if current table is to be used

output: none; does not return

Access: task or ISR

### 3.6.2 Get System Address Table (`s_getsat`)

The Get System Address Table system call returns the address of the system address table. This table contains three addresses in the following format:

```
struct SAT_TYPE
{
    struct CFG_TYPE *sat_cfgp; /* pointer to configuration table */
    int *sat_evtp;           /* pointer to exception vector table */
    struct GST_TYPE *sat_gstp; /* pointer to global system table */
};
```

See [Section 4.9 on page 91](#) for the definition of the configuration table and [Section 4.15 on page 103](#) for the definition of the global system table. The exception vector table is as defined by the Motorola 68xxx family of processors.

C Interface:

```
s_getsat ()
```

```
return:      address of system address table
```

Assembly Interface:

```
TRAP        #0
```

```
input:      D0.L = 0x1A
```

```
output:     D0.L = address of system address table
```

Access: task or ISR

### 3.6.3 Return from ISR (s\_iret)

An interrupt service routine must issue an ISR Return system call at the completion of its processing rather than executing a return from exception (RTE) instruction. If the interrupted code was user state (task level), and if a task switch is waiting to be processed, control is transferred to the dispatcher for task rescheduling; otherwise, an RTE is executed and control is returned to the interrupted code. In the interest of efficiency, this call is made by a TRAP through a separate vector.

The ISR for the highest-priority interrupt in the system can execute an RTE directly if it has not made a system call that can have caused a task to be scheduled (Create Task, Resume, Post & Resume, or Set Alarm). Any other ISR must always issue an ISR Return system call, because it might be interrupted by a second, higher-priority interrupt. The ISR for that interrupt can cause a task to be scheduled, but the task switch will not be performed at its completion, because it returns to the first (lower-priority) ISR rather than to user state.

C Interface:

```
s_iret ()
```

return: none; does not return

Assembly Interface:

```
TRAP #1
```

input: none

output: none; does not return

Access: ISR only

### 3.6.4 Set Interrupt Level (`s_iset`)

The Set Interrupt Level system call sets the interrupt priority mask value in the 68xxx status register and returns its value before the change. This call is supplied because all tasks operate in user state and cannot directly modify the contents of the status register. In the interest of efficiency, it is made directly by a TRAP through a separate vector. The return value can be used by a task to restore the interrupt priority mask to its previous value with a second call to this routine.

#### C Interface:

```
s_iset ( mask )  
unsigned short mask;
```

mask:            interrupt mask value to set

return:          mask value before change

#### Assembly Interface:

TRAP            #2

input:           D0.W = interrupt priority mask value

output:          D0.W = priority mask value before change

Access: task only





# System Data Structures

This chapter describes the data structures used to control system operation. Except where noted, these structures are internal to the operating system and cannot be directly accessed by application tasks. Definitions of the internal data structures are provided here to aid in debugging and understanding the system's operation.

The data structures are defined in C language format. The data type `short` is 16 bits. Pointers and the data type `int` are 32 bits. No “pads” or “fillers” (sometimes automatically generated by C compilers for data alignment) are included unless explicitly defined. For additional clarity, [Appendix B](#) defines each structure by field sizes and offsets.

## 4.1 Task Control Block (TCB)

The system maintains a TCB for each configured task, which contains information related to the task's execution state. TCBs are arranged as an array of structures, with the task ID used as an index into the array. Each TCB contains the following information:

```
struct TCB_TYPE
{
    struct TCB_TYPE  *t_next;      /* link to next TCB on disp.q      */
    unsigned short   t_id;         /* task ID                          */
    unsigned short   t_pri;       /* task priority                    */
    unsigned short   t_slice;     /* 0 = time slice disabled         */
                                /* 1 = time slice enabled          */
    unsigned short   t_state;     /* task state (see below)          */
}
```

```
    unsigned int    t_stack;    /* user stack pointer    */
    struct ECB_TYPE *t_ecb;    /* ECB address          */
    struct ACB_TYPE *t_acb;    /* ACB address          */
};
```

The `t_state` field can contain the following values:

0x0000	terminated (TCB is not allocated)
0x0001	task is currently executing
0x0002	task is suspended
0xC00	task is currently executing and queued to execute again
0x8002	task is queued to execute after a suspend
0x8004	task is queued to execute after preemption
0xC004	task is queued to execute after preemption, with a resume pending (will move to 0xC001 rather than 0x0001 when executed)

A TCB is linked to the appropriate singly linked dispatch queue ([Section 4.8 on page 90](#)) when the associated task is scheduled for execution and is removed from the queue when it is dispatched (begins execution).

When a task is preempted, its registers, including its status register and its program counter, are saved on its user stack, and the current user stack pointer is saved in the TCB.

User code must not modify any of the TCB fields.

## 4.2 Queue Control Block (QCB)

The system maintains a QCB for each configured exchange queue. QCBs are arranged as an array of structures, with the queue ID used as an index into the array.

```
struct QCB_TYPE
{
    struct SBH_TYPE  *q_head;      /* head pointer          */
    struct SBH_TYPE  *q_tail;     /* tail pointer          */
    unsigned int     q_owner;     /* TCB address of owner */
    unsigned short   q_count;     /* count of queue elements */
    unsigned short   q_type;     /* 0 = single, 1 = double */
    unsigned short   q_id;       /* queue ID              */
};
```

The queue head and tail pointers are used to implement either a singly or doubly linked queue. The count of queue elements is provided as a debugging aid.

The address of the QCB is provided to the caller on return from the Create Queue system call. Under controlled circumstances, the information contained in the QCB can be used by a task or interrupt service routine to examine or manipulate the queue. User code can modify only the `q_head`, `q_tail`, and `q_count` fields.

### 4.3 Resource Control Block (RCB)

The system maintains an RCB for each configured resource. RCBs are arranged as an array of structures, with the resource ID used as an index into the array. Each RCB contains the following information:

```
struct RCB_TYPE
{
    struct SBH_TYPE  *r_head;      /* token/carrier head pointer    */
    struct SBH_TYPE  *r_tail;     /* token/carrier tail pointer    */
    unsigned short   r_type;      /* 0 = FIFO, 1 = priority        */
    short            r_count;     /* count of queue elements       */
    unsigned short   r_state;     /* resource state                 */
    unsigned short   r_id;       /* resource ID                    */
};
```

The token/carrier head and tail pointers implement a singly linked queue. When the count is greater than or equal to zero, it indicates the number of resource tokens available (linked to the queue). When the count is less than zero, it indicates the number of outstanding resource requests (the number of carrier messages linked to the queue).

User code must not modify any of the fields of the RCB.

## 4.4 Partition Control Block (PCB)

The system maintains a PCB for each configured partition. PCBs are arranged as an array of structures, with the partition ID used as an index into the array. Each PCB contains the following information:

```
struct PCB_TYPE
{
    struct SBH_TYPE  *p_head;      /* free list head pointer      */
    struct SBH_TYPE  *p_tail;     /* free list tail pointer     */
    unsigned int     p_start;     /* partition starting address  */
    unsigned int     p_end;       /* partition ending address + 1 */
    unsigned int     p_bsize;     /* buffer size                 */
    unsigned short   p_total;     /* total number of buffers     */
    unsigned short   p_count;     /* count of buffers on free list */
    unsigned short   p_id;       /* partition ID                */
};
```

The free list head and tail pointers implement a singly linked queue. The `sb_nxte` field of each buffer header (see [Section 4.12 on page 97](#)) is used as a link field while it is attached to the free list.

When a partition is created during system initialization, all buffers are linked to the free list. When a buffer is allocated, it is removed from the head of the list. When it is returned, it is appended to the tail.

User code must not modify any of the fields of the PCB.

## 4.5 Alarm Control Block (ACB)

The system maintains an ACB for each configured alarm. ACBs are arranged as an array of structures, with the alarm ID used as an index into the array. Each ACB contains the following information:

```
struct ACB_TYPE
{
    struct ACB_TYPE  *a_flink;    /* link to next ACB          */
    struct ACB_TYPE  *a_blink;    /* link to previous ACB     */
    int              (*a_sigad)(); /* address of signal routine */
    unsigned int     *a_flgad;    /* address of flag           */
    unsigned int     a_mask;      /* mask value for flag      */
    unsigned short   a_type;      /* alarm type (see below)   */
    short            a_tick;      /* tick count                */
    unsigned short   a_state;     /* alarm state (see below)  */
    unsigned short   a_id;        /* alarm ID                  */
};
```

The `a_type` field can contain the following values:

0x00	standard
0x01	special
0x80	task

The `a_state` field can contain the following values:

0x0000	not allocated
0x8000 or 0x8100	idle
0x80C1 or 0x81C1	just started
0x8081 or 0x8181	running
0x8001 or 0x8101	still queued after a cancel

An ACB is linked to the special alarm queue ([Section 4.7](#)) when it is started with a Set Alarm system call (standard or special alarm) or when its associated task is suspended with a timer set (task alarm). The tick count in an ACB attached to the special alarm queue is always absolute and is zero if the alarm has been canceled. A special alarm is removed from the special alarm queue only by the Timer task, either when it finds that the alarm has been canceled (any type) or when it expires (special alarms only).

Standard and task ACBs are moved to the standard alarm queue (see [Section 4.6](#)) from the special alarm queue by the Timer task. The tick count in an ACB attached to the standard alarm queue is always relative. An ACB is removed from the standard alarm queue by the Timer task when it expires or by the system when it is canceled.

If the signal routine address is zero, the Timer task will not make a subroutine call when the alarm expires. If the flag address or bit mask is zero, the Timer task will not set a flag when the alarm expires.

While a standard or special alarm is in the idle state (not running), user code can modify the `a_sigad`, `a_flagad`, and `a_mask` fields of the ACB. While a special alarm is running, `a_tick` can be modified, but the accuracy of the alarm's duration is unpredictable unless the Timer task is scheduled (with a Resume system call) after modifying the tick count. User code must not modify the `a_tick` field of a standard ACB at any time. User code must not modify any of the fields of a task ACB.

## **4.6 Standard Alarm Queue**

The system implements a doubly linked standard alarm queue with a head and tail pointer, which might contain a linked list of standard and task ACBs. The standard alarm queue is maintained in an order corresponding to ascending tick counts. The tick count in each ACB is converted to a relative value when it is inserted into the queue. That is, the number of ticks remaining for a particular alarm is equal to the sum of the tick count in its ACB and the tick counts in all preceding ACBs on the queue.

Because tick counts are relative, the Timer task decrements only the tick count in the ACB at the head of the queue and removes ACBs from the head of the queue as they expire. When an alarm is canceled, the corresponding ACB can occupy any position in the queue; therefore, when it is unlinked, the relative tick count in the following ACB must be adjusted accordingly.

User code must not modify the standard alarm queue head and tail pointers.



## 4.7 Special Alarm Queue

The system implements a singly linked special alarm queue with a head and tail pointer, which can contain a linked list of ACBs. An ACB is linked to the tail of the special alarm queue when its associated alarm is started (set) or when its associated task is suspended with a timer set. When the Timer task executes, it scans the entire special alarm queue. ACBs with tick counts of zero (canceled) are removed from the queue. Standard and task ACBs are moved to the standard alarm queue. The tick counts in the ACBs of running special alarms are decremented, and the ACBs are removed when they have expired.

User code must not modify the special alarm queue head and tail pointers.

## **4.8 Dispatch Queues**

The system maintains a singly linked dispatch queue for each task priority. (The number of priorities in the system is a configurable parameter.) Each queue consists of a head and a tail pointer and can contain a linked list of TCBs. When a task is scheduled to execute, its TCB is linked to the tail of the dispatch queue corresponding to its priority.

The system selects the next task to be dispatched by checking each dispatch queue in order of priority. (The queue corresponding to the highest priority in the system is always checked first.) The TCB at the head of the first non-empty queue is unlinked, and the associated task is dispatched.

User code must not modify the dispatch queue head and tail pointers.

## 4.9 Configuration Table

When the operating system is initialized at startup, and when it is re-initialized with the Initialize OS system call, it is configured according to the contents of a system configuration table, the address of which is an input parameter to the Initialize OS system call. The system does not modify the configuration table and accesses it only during system initialization.

The system configuration table consists of a list of parameters, described by the CFG\_TYPE structure. Also included is a variable-length array of task initialization structures, each containing the information necessary to create a task (see [Section 4.10](#)).

```
struct CFG_TYPE
{
    unsigned short    cf_ntsk;        /* number of tasks          */
    unsigned short    cf_npri;        /* number of priorities     */
    unsigned short    cf_nque;        /* number of queues         */
    unsigned short    cf_nlrn;        /* number of alarms        */
    unsigned short    cf_npar;        /* number of partitions     */
    unsigned short    cf_nrsc;        /* number of resources      */
    unsigned short    cf_ltik;        /* number of msec per tick  */
    unsigned short    cf_ltsl;        /* number of ticks per time slice */
    unsigned int      (*cf_cisr)();    /* user clock interrupt     */
                                     /* subroutine address       */
};
```

```
struct CONF_TYPE
{
    struct CFG_TYPE   cfp;           /* configurable parameters      */
    struct TIS_TYPE   cft[NUMTIS+1]; /* task initialization structures */
};
```

The final element in the array `cft` must specify a task ID (`ti_id`) of zero to terminate the list. `NUMTIS` refers to the number of task initialization structures. The configurable parameters are used by the system as follows:

`cf_ntsk` = number of tasks (1– 65535)

The system will create `cf_ntsk` TCBs and `cf_ntsk` corresponding task-type ACBs. Valid task IDs (specified on input to the Create Task system call) are 2 through `cf_ntsk`. (Task ID 1 is reserved for the Timer task.)

`cf_npri` = number of priorities (0 – 65535)

The system creates `cf_npri + 1` dispatch queues, corresponding to priorities 0 (highest) through `cf_npri` (lowest). The Timer task is created by the system at priority 0. Valid priorities (specified on input to the Create Task system call) are 1 through `cf_npri`. Any number of tasks can be created at a particular priority.

`cf_nque` = number of queues (0 – 65535)

The system creates `cf_nque` QCBs. Valid queue IDs (specified on input to the Create Queue system call) are 1 through `cf_nque`.

`cf_nlrm` = number of alarms (0 – 65535)

The system creates `cf_nlrm` ACBs for standard and special alarms. Valid alarm IDs (specified on input to the Create Alarm system call) are 1 through `cf_nlrm`.

`cf_npar` = number of partitions (0 – 65535)

The system creates `cf_npar` PCBs. Valid partition IDs (specified on input to the Create Partition system call) are 1 through `cf_npar`.

`cf_nrsc` = number of resources (0 – 65535)

The system creates `cf_nrsc` RCBs. Valid resource IDs (specified on input to the Create Resource system call) are 1 through `cf_nrsc`.

`cf_ltik` = length of tick (1 – 65535)

A *tick* is the unit of time used by the system for alarms and the time slice. The tick resolution is set to `cf_ltik` milliseconds.

`cf_ltsl` = length of time slice (0 – 65535)

The system sets the length of the time slice to `cf_ltsl` ticks. This parameter can be set to zero if time slicing will not be enabled for any of the tasks in the system.

`cf_cisr` = address of user clock interrupt subroutine

If `cf_cisr` is nonzero, a subroutine call is made to the address `cf_cisr` on each interrupt by the system clock (at completion of the system clock interrupt service routine described in [Section 2.5.1 on page 26](#)). This subroutine can perform hardware-specific functions (such as clearing the interrupt) or any special functions required by the user.

The format of the task initialization structure is defined in [Section 4.10](#). The last task initialization structure in the configuration table must contain a task ID of zero (invalid) to terminate the list.

## 4.10 Task Initialization Structure (TIS)

A TIS provides the system with the information necessary to create a task. The address of such a structure must be provided on input to the Create Task system call, and a list of the structures is included in the configuration table, as described in [Section 4.9](#). The system does not modify the contents of the structure.

Each task initialization structure contains the following fields:

```
struct TIS_TYPE
{
    unsigned short    ti_id;        /* task ID                */
    unsigned short    ti_pri;       /* task priority          */
    int               (*ti_strt)(); /* entry point address    */
    unsigned int      ti_usp;       /* initial user stack pointer */
    unsigned short    ti_tsen;     /* 0 = disable time slice */
                                /* 1 = enable time slice   */
    unsigned short    ti_fill;     /* fill to a longword boundary */
};
```

The field `ti_fill` is included so that, when a list of task initialization structures is included in the configuration table, each structure begins on a longword boundary.

## 4.11 Event Control Block (ECB)

An ECB can be created by a task and passed as input to the Suspend system call to specify the conditions under which it should be rescheduled. The system stores the address of the ECB in the task's TCB while the task is suspended but does not modify the ECB in any way.

The ECB is a structure containing three fixed-length fields and a variable-length array, as follows:

```
struct ECP_TYPE
{
    short          e_tick;          /* ticks for alarm          */
    unsigned short e_resm;         /* 0 = disable resume      */
                                /* 1 = enable              */
    unsigned short e_cps;         /* cancel previous scheduling: */
                                /* 0 = yes, 1 = no         */
    unsigned short e_filler;      /* fill to a longword boundary */
};
struct ECB_TYPE
{
    struct ECP_TYPE ecp;          /* ECB parameters          */
    unsigned short e_ques[NUMQID+1]; /* list of queue IDs      */
};
```

If `e_tick` is set to zero, no alarm is set. The `e_resum` flag identifies whether the suspended task will be scheduled for execution when a Resume system call is made by another task in the system. The `e_cps` flag identifies whether the task's TCB should be removed from the dispatch queue if the task is already scheduled to execute at the time it is suspended. The field `e_filler` is included to align the list of queue IDs on a longword boundary.

A task can own any number of exchange queues, each of which is identified with a queue identification. If a particular queue ID is included in the ECB list, a Post & Resume call specifying that queue causes the suspended task to be scheduled for execution. A Post & Resume call specifying a queue owned by the task but not included in the list will succeed but will not cause the task to be scheduled for execution. A queue cannot be included in the ECB list unless it is owned by the suspending task. `NUMQID` refers to the number of queue IDs in the list. The last queue ID is followed by a zero to terminate the list.



## 4.12 System Buffer Header (SBH)

Buffers allocated from partitions, and also elements posted to exchange queues, begin with a standard system buffer header. In general, a queue element can be said to consist of one or more linked buffers, but this does not imply that the buffers were obtained from a system partition; any data structure beginning with the required header can be posted to a queue. Likewise, a buffer allocated from a partition can be used for any purpose (it is not limited to use as a queue element).

```

struct SBH_TYPE
{
    struct SBH_TYPE  *sb_nxte;    /* address of next queue      */
                                /* element or buffer          */
    struct SBH_TYPE  *sb_pree;    /* address of previous queue  */
                                /* element                    */
    struct SBH_TYPE  *sb_thse;    /* address of this queue      */
                                /* element or buffer          */
    struct SBH_TYPE  *sb_nxtb;    /* address of next buffer in  */
                                /* queue element              */
    unsigned short   sb_pid;      /* partition ID                */
    unsigned short   sb_dlen;     /* data length                  */
    unsigned short   sb_disp;     /* disposition flag            */
    unsigned short   sb_dmod;     /* disposition modifier        */
};

```

Only certain fields of the buffer header are required or created by the system, and only during certain operations. Only the buffer header in the first buffer of a queue element is accessed by the system. The requirements for each of the fields of the system buffer header are described as follows:

`sb_nxte` The system uses this field under two circumstances: (1) while a buffer is attached to a partition free list, where it contains the address of the next buffer in

the list, or (2) while a queue element is attached to a singly or doubly linked queue, where it contains the address of the next queue element.

`sb_pree` The system uses this field when a queue element is attached to a doubly linked queue. It contains the address of the previous queue element.

`sb_thse` The system uses this field for consistency checks when a buffer or queue element is added to or removed from the free list of a partition, or to or from a queue (singly or doubly linked). It must contain the starting address of the buffer or queue element. The system sets this field in every buffer of a partition when the partition is created.

`sb_nxtb` The system does not access this field. It can be used by applications to link additional buffers to a queue element.

`sb_pid` The system sets this field in every buffer of a partition when the partition is created. When a buffer is released, this field must contain the ID of its partition.

`sb_dlen` The system does not access this field. Tasks can use it to specify the length of valid data in the buffer or the length of the buffer itself.

`sb_disp` The system does not access this field. In the buffers of queue elements that are used for intertask communication, it can be used by the sending task to specify the action that should be taken by the recipient on completion of processing (for example, release to partition or post to a queue).

`sb_dmod` The system does not access this field. Tasks can use it to specify further the action to be taken after processing a buffer (for example, a queue ID).

Tasks are required to maintain (or create) only certain fields of the system buffer header in a buffer or queue element and only in certain circumstances, as follows:

1. When a buffer is released to a partition, the `sb_thse` and `sb_pid` fields must be valid.

2. When a queue element is posted to a singly linked queue, the `sb_thse` field must be valid, and the `sb_nxte` field must be available for use by the system. (These requirements are for the first buffer of a queue element only.)
3. When a queue element is posted to a doubly linked queue, the `sb_thse` field must be valid, and the `sb_nxte` and `sb_pree` fields must be available for use by the system. (These requirements are for the first buffer of a queue element only.)

### 4.13 Resource Carrier Message

A task can create a resource carrier message and include its address as an optional input parameter to the Request Resource system call. If the requested resource is not available at the time the request is made, the system saves the address of the resource carrier message, and when the resource becomes available, stores the address of the resource token in the message and posts it to the specified queue.

The resource carrier message is a queue element and, therefore, must begin with the standard system buffer header (defined in [Section 4.12](#)). The format of the message is defined as follows:

```
struct RC_TYPE
{
    struct SBH_TYPE  rc_sbh;      /* system buffer header      */
    unsigned short  rc_qid;      /* return queue ID           */
    unsigned short  rc_res;      /* 0 = resume, 1 = post & resume */
    unsigned short  rc_pri;      /* priority                   */
    unsigned short  rc_cmp;      /* completion code (see below) */
    struct SBH_TYPE *rc_tkn;     /* address of resource token  */
};
```

When a carrier message is created, the return queue ID must be specified, and the resume flag must be set to signify whether the system should return the message with a Post Message or Post & Resume system call. If the requested resource is allocated on a priority basis (specified when the resource is created), a priority must also be provided. This priority does not necessarily correspond to the requester's task priority and is not limited to the range of task priorities.

Before posting the carrier message to the specified queue, the system sets the completion code. If the resource has been successfully allocated, the system also supplies the address of the associated resource token.

The `rc_cmp` field can contain the following values:

0x00	request succeeded (resource has been allocated)
0x01	request canceled
0x02	request failed (resource has been deleted)

## 4.14 Stack Format

When a task is suspended or is preempted, the 66 bytes immediately preceding the current user stack pointer are used to save the values of its registers. The stack pointer value is not updated and is saved in the `t_stack` field of the TCB. All registers are saved as longwords, except for the status register, which occupies a single word.

increasing	usp - 66	program counter (return address)
addresses	usp - 62	status register
	usp - 60	D0
		...
	usp - 32	D7
	usp - 28	A0
		...
	usp - 4	A6
	usp (t_stack)	

## 4.15 Global System Table (GST)

The GST contains various information that might be useful to application tasks and also during debugging. In addition, the unused fields of the table can be defined by applications to contain counts of errors, global variables, or for any other purpose. The address of the GST is included in the system address table (see [Section 3.6.2 on page 77](#)).

```
struct GST_TYPE
{
    int          (*gs_init) (); /* os initialization entry address */
    unsigned int gs_ticks;     /* clock tick count */
    unsigned int gs_idle;     /* idle task count */
    unsigned int gs_ramend;   /* first byte of RAM
                               /* available to applications */
    char         gs_version[4]; /* version number (ASCII) */
    unsigned int gs_debug;     /* debug code */
    unsigned int gs_unused[58]; /* unused entries */
};
```

The clock tick count is incremented by the clock interrupt service routine on each tick. The idle task count is incremented each time the idle task is executed, which occurs whenever the current task is suspended and no other tasks are scheduled for execution. The debug codes are defined in [Section A.1 on page 105](#).





# Debugging Aids

This appendix describes the facilities that have been incorporated into the operating system to assist programmers in debugging application programs.

## A.1 Global System Table

The GST is defined in [Section 4.15 on page 103](#). Several of the fields in the table might be of interest while debugging an application. If the clock tick count is not incremented, or increases more slowly than expected (depending on the tick length configured), interrupts might be erroneously disabled, or a higher-priority interrupt might be “hogging” processor time. The idle count can be monitored to give a rough estimate of system load. (However, if one or more tasks in the system execute continuously without suspending, the idle count will never be incremented.)

During normal system operation the debug code in the `gs_debug` field signifies the level of input parameter and consistency checking that will be performed by the system, as follows:

0x00000000	input parameter and consistency checking enabled
0x01000000	input parameter checking enabled; consistency checking disabled
0x02000000	input parameter and consistency checks disabled

When a consistency check fails, the system stores a code identifying the error in the `gs_debug` field and “panics” with an illegal instruction trap. These error codes are defined in [Section A.1.1](#).

When an input parameter check fails in a system call, an error is returned, as defined for each system call in [Chapter 3](#); however, when input parameter checking is disabled, the system routines return no errors except those listed in [Table A-1](#).

**Table A-1: System Errors**

System Call	Error Code	Description
<code>s_qdelet</code>	0x02	queue is not empty
<code>s_accpt</code>	0xFF	queue is empty
<code>s_rreq</code>	0xFF	no resource tokens available
<code>s_pdelet</code>	0x02	one or more buffers currently allocated
<code>s_breq</code>	0xFF	no buffers available

Input parameter and consistency checks are conditionally assembled in order to improve system performance; therefore, the level of checking enabled is determined when the operating system is assembled and cannot be changed dynamically or through system reconfiguration.

### A.1.1 Panic Codes

When consistency checking is enabled and an error is detected, the system routine detecting the error stores an identifying code in the `gs_debug` field of the GST and branches to a “panic” location, which contains an illegal instruction. Execution of this instruction generates an illegal instruction trap. Each of the error codes is listed as follows. Also included are the system calls during which it can occur, any register values that might be of assistance in evaluating the cause of the error, and the offset on the supervisor stack at which the caller’s return address is located.

As part of consistency checking, when a partition is created, the `sb_pree` field of each buffer is set to the value `0xF5F5F5F5`. When a buffer is allocated, the field is set to zero. When a buffer is released, the field is again set to `0xF5F5F5F5`. The `sb_pree` field of all buffers on the free list of a partition, therefore, is expected to contain this value, and the same field of allocated buffers is assumed to contain any other value. Error codes `0x09` and `0x10` are related to these consistency checks.

`gs_debug`            `0x01`  
`system calls:`      `s_tdelete, s_susp`  
`description:`        The TCB state (`t_state`) indicates that the TCB is scheduled for execution, but it was not found on the dispatch queue for its priority. This error suggests a corruption of the system.  
`registers:`           `A0.L = TCB address`  
`return address:`     `sp + 34`

`gs_debug`            `0x02`  
`system calls:`      `s_tdelete`  
`description:`        The ACB state (`a_state`) indicates that the task's ACB is queued (after being canceled), but it was not found on the special alarm queue. This error suggests a corruption of the ACB or the system in general.  
`registers:`           `A0.L = task's ACB address (the alarm ID in the ACB is equivalent to the task ID)`  
                       `A2.L = address of special alarm queue head`  
`return address`      `sp + 14`

gs\_debug: 0x03  
system calls: s\_osinit  
description: An error was returned from s\_tcreat when attempting to create the Timer task or a task listed in the configuration table. This error indicates a badly formatted task initialization structure or a duplicated or out-of-range task ID.  
registers: D0.L = error code returned from s\_tcreat  
A0.L = task initialization structure address  
return address: not available (stack has been reinitialized)

gs\_debug: 0x04  
system calls: s\_accpt  
description: The sb\_thse field of the element obtained from the queue is invalid. This error indicates that data within the queue element, or links between queue elements, might have been corrupted.  
registers: A0.L = queue element address  
A1.L = QCB address  
return address: sp + 22

gs\_debug: 0x05  
system calls: s\_breq  
description: The sb\_thse field of the element obtained from the partition is invalid. This error suggests that data within the buffer has been corrupted while the buffer was attached to the free list. For example, data might have been written past the end of the preceding (adjacent) buffer, or the links of the free list might have been corrupted.  
registers: A0.L = buffer address  
A1.L = PCB address  
return address: sp + 22

gs\_debug: 0x06  
system calls: s\_breq  
description: The partition ID in the buffer does not match the ID of the partition from which it was obtained. This error suggests that data within the buffer has been corrupted while the buffer was attached to the free list (see error code 0x05).  
registers: A0.L = buffer address  
A1.L = PCB address  
return address: sp + 22

gs\_debug: 0x07  
system calls: s\_tcreat, s\_resum, s\_postr  
description: The system is attempting to schedule a task that is in the terminated state. This error indicates corruption of the TCB or the system in general. It might also occur during a task switch or during processing of the expiration of a suspended task's alarm.  
registers: A3.L = TCB address  
return address: sp + 26 or sp + 30 (not applicable if the error is not related to a system call)

gs\_debug: 0x08  
system calls: s\_brel  
description: The number of buffers on the free list of the partition is already equal to the total number of buffers in the partition. This error suggests that the free list of the partition or the PCB itself has been corrupted.  
registers: A0.L = buffer address  
A1.L = PCB address  
D0.W = current count of buffers on free list  
return address: sp + 26

gs\_debug: 0x09  
system calls: s\_breq  
description: The value of the sb\_pree field of the buffer obtained from the partition is not equal to the value 0xF5F5F5F5. This error might suggest (1) that data within the buffer has been corrupted while the buffer was attached to the free list (see error code 0x05 above), (2) that the buffer remained in use by an application after being released to the partition's free list, or (3) that the links of the free list have been corrupted.  
registers: A0.L = buffer address  
A1.L = PCB address  
return address: sp + 22

gs\_debug: 0x0A  
system calls: s\_brel  
description: The value of the sb\_pree field of the buffer to be released to the partition is equal to the value 0xF5F5F5F5, signifying that it is already linked to the free list. This error indicates that the application might be attempting to release a buffer that has already been released.  
registers: A0.L = buffer address  
return address: sp + 26

gs\_debug: 0x0B  
system calls: s\_accpt  
description: The queue head pointer is equal to zero, indicating that the queue is empty, but the count of queue elements is nonzero. This error suggests corruption of the QCB.  
registers: A1.L = QCB address  
return address: sp + 22

gs\_debug: 0x0C  
system calls: s\_accept  
description: The queue head pointer is nonzero, signifying that the queue is not empty, but the count of queue elements was zero (and has now been decremented to -1). This error suggests corruption of the QCB.  
registers: A1.L = QCB address  
return address: sp + 22

gs\_debug: 0x0D  
system calls: s\_delete  
description: The ACB state (a\_state) signifies that the ACB is queued (running or canceled), but it was not found on the special alarm queue. This error suggests a corruption of the ACB or the system in general.  
registers: A0.L = ACB address  
A2.L = address of special alarm queue head  
return address: sp + 26

gs\_debug: 0x0E  
system calls: s\_rdelete  
description: The sb\_thse field of a token message obtained from the RCB queue is invalid. This error suggests corruption of the RCB queue.  
registers: A0.L = token message address  
A1.L = RCB address  
return address: sp + 26

gs\_debug: 0x0F  
system calls: s\_rdelet  
description: A call to s\_post failed (attempting to post a token message to the specified queue). This error suggests that an invalid queue ID was supplied.  
registers: D0.L = completion status returned from s\_post  
D1.L = queue  
A0.L = token message address  
A1.L = RCB address  
return address: sp + 26

gs\_debug: 0x10  
system calls: s\_rdelet  
description: The sb\_thse field of a carrier message obtained from the RCB queue is invalid. This error suggests corruption of the RCB queue.  
registers: A0.L = carrier message address  
A1.L = RCB address  
return address: sp + 26

gs\_debug: 0x11  
system calls: s\_rdelet, s\_rcan, s\_rrel  
description: A call to s\_post or s\_postr failed (attempting to post carrier message to the specified queue). This error suggests that the queue ID in the carrier message is no longer valid.  
registers: D0.L = completion status returned by s\_post or s\_postr  
D1.L = queue ID  
A0.L = carrier message address  
A1.L = RCB address  
return address: sp + 34 or sp + 38



gs\_debug: 0x12  
system calls: s\_rreq  
description: The sb\_thse field of the token message obtained from the RCB queue is invalid. This error suggests corruption of the RCB queue.  
registers: A0.L = token message address  
A1.L = RCB address  
return address: sp + 26

gs\_debug: 0x13  
system calls: s\_rcan  
description: The sb\_thse field of the canceled carrier message (obtained from the RCB queue) is invalid. This error suggests corruption of the RCB queue.  
registers: A0.L = carrier message address  
A1.L = RCB address  
return address: sp + 22

gs\_debug: 0x14  
system calls: s\_rrel  
description: The sb\_thse field of the carrier message obtained from the RCB queue is invalid. This error suggests corruption of the RCB queue.  
registers: A0.L = carrier message address  
A1.L = RCB address (Token message address is at top of stack)  
return address: sp + 30

gs\_debug: 0x15  
system calls: —  
description: The function code in register D0, which should identify the system call, is invalid.  
registers: D0.L = function code  
return address: sp + 6 or sp + 10

gs\_debug: 0x16  
system calls: s\_tcreat, s\_resum, s\_postr  
description: The system is attempting to add a TCB to a dispatch queue when the TCB is already linked to the queue. This error indicates corruption of the system. It might also occur during a task switch or during processing of the expiration of a suspended task's alarm.  
registers: A1.L = address of dispatch queue tail  
A3.L = TCB address  
return address: sp + 26 or sp + 30 (not applicable if the error is not related to a system call)

gs\_debug: 0x17  
system calls: s\_tdelet, s\_susp  
description: The call has been made from supervisor state (for example, an interrupt service routine). These calls are valid from the task level only.  
return address: sp + 14 (s\_tdelet) or sp + 10 (s\_susp)

## A.2 System Variables

System variables that might help you determine the state of a task, queue, alarm, partition, or resource immediately follow the GST. The GST contains 64 longwords, or 100 hexadecimal bytes. The system variables described in [Table A-2](#) and [Table A-3](#) begin at an address identified by the GST plus  $100_{16}$ . They are defined in terms of offsets from that location.

### A.2.1 Pointers to Control Structures

Each variable in [Table A-2](#) contains the address of the first control structure of its kind. To locate the control structure corresponding to a particular ID, subtract one from the ID, multiply it by the length of the structure, and add the result to the address of the first structure. To determine the dispatch queue head and tail pointers for a particular priority, multiply the priority by four and add the result to the addresses of the first head and tail pointers.

**Table A-2:** Control Structures

Offset (hex)	Variable	Size	Description
0	tcbs	L	Contains the address of the first TCB
4	qcbs	L	Contains the address of the first QCB
8	acbs	L	Contains the address of the first ACB
C	tacb	L	Contains the address of the first task ACB
10	pcbs	L	Contains the address of the first PCB
14	rcbs	L	Contains the address of the first RCB
18	dq_heads	L	Contains the address of the first dispatch queue head pointer
1C	dq_tails	L	Contains the address of the first dispatch queue tail pointer

## A.2.2 Alarm Queues

Table A-3 shows the alarm queues.

**Table A-3: Alarm Queues**

Offset (hex)	Variable	Size	Description
20	tq_head	L	Contains the address of the first ACB on the standard alarm queue, or zero if the queue is empty
24	tq_tail	L	Contains the address of the last ACB on the standard alarm queue, and is undefined if the queue is empty
28	aq_head	L	Contains the address of the first ACB on the special alarm queue, or zero if the queue is empty
2C	aq_tail	L	Contains the address of the last ACB on the special alarm queue, and is undefined if the queue is empty

## A.2.3 Task Execution Variables

Table A-4 shows the task execution variables.

**Table A-4: Task Execution Variables**

Offset (hex)	Variable	Size	Description
30	curtsk	L	Contains the TCB address of the currently executing task
34	curpri	W	Contains the priority of the currently executing task
36	tlock	W	Equal to zero if task rescheduling is enabled, equal to one if task rescheduling is disabled
38	tswitch	W	Two bits are defined, as follows: bit 15 (high order) is set if a task switch is pending; bit 0 (low order) is set if no state save is required at the next task switch

## Data Structure Field Offsets

The following tables show more precise definitions of the data structures defined with C structures in [Chapter 4](#). Here, each field is described by its offset from the start of the structure and its size in bytes. When C or any other high-level language is used to define these structures, the user should verify that the resulting field offsets correspond to those provided in this appendix.

The number of dispatch queues ([Table B–8 on page 120](#)) is dependent on the number of configured task priorities. This data structure is defined with an example showing the dispatch queues for a system with four priorities. The `end_marker` field contains a unique value that identifies the end of the list of dispatch queue heads. (Each queue head contains either a TCB address or zero if the queue is empty.)

The configuration table, defined in [Table B–9 on page 120](#), includes a variable number of TISs ([Table B–10 on page 121](#)). The structure is defined with an example showing a configuration table with two task initialization structures (TIS 1 and TIS 2), followed by a zero to terminate the list. (A configuration table with no task initialization structures would contain a zero immediately following the `cf_cisr` field.)

The event control block, defined in [Table B–11 on page 121](#), includes a variable-length list of queue IDs. As an example, the table shows an ECB with two queue IDs (QID 1 and QID 2), followed by a zero to terminate the list. (An ECB with no queue IDs would contain a zero immediately following the `e_fill` field.)

All other data structures are of fixed length.

**Table B-1: Task Control Block (TCB)**

Offset (hex)	Field Name	Size (hex bytes)
0	t_next	4
4	t_id	2
6	t_pri	2
8	t_slice	2
A	t_state	2
C	t_stack	4
10	t_ecb	4
14	t_acb	4

**Table B-2: Queue Control Block (QCB)**

Offset (hex)	Field Name	Size (hex bytes)
0	q_head	4
4	q_tail	4
8	q_owner	4
C	q_count	2
E	q_type	2
10	q_id	2

**Table B-3: Resource Control Block (RCB)**

Offset (hex)	Field Name	Size (hex bytes)
0	r_head	4
4	r_tail	4
8	r_type	2
A	r_count	2
C	r_state	2
E	r_id	2

**Table B-4: Partition Control Block (PCB)**

---

<b>Offset (hex)</b>	<b>Field Name</b>	<b>Size (hex bytes)</b>
0	p_head	4
4	p_tail	4
8	p_start	4
C	p_end	4
10	p_bsize	4
14	p_total	2
16	p_count	2
18	p_id	2

---

**Table B-5: Alarm Control Block (ACB)**

---

<b>Offset (hex)</b>	<b>Field Name</b>	<b>Size (hex bytes)</b>
0	a_flink	4
4	a_blink	4
8	a_sigad	4
C	a_flagad	4
0	a_mask	4
14	a_type	2
16	a_tick	2
18	a_state	2
1A	a_id	2

---

**Table B-6: Standard Alarm Queue**

---

<b>Offset (hex)</b>	<b>Field Name</b>	<b>Size (hex bytes)</b>
0	tq_head	4
4	tq_tail	4

---

**Table B-7: Special Alarm Queue**

Offset (hex)	Field Name	Size (hex bytes)
0	aq_head	4
4	aq_tail	4

**Table B-8: Dispatch Queues**

Offset (hex)	Field Name	Size (hex bytes)
0	dq_head 0	4
4	dq_head 1	4
8	dq_head 2	4
C	dq_head 3	4
10	(end_marker)	4
14	dq_tail 0	4
18	dq_tail 1	4
1C	dq_tail 2	4
20	dq_tail 3	4

**Table B-9: Configuration Table**

Offset (hex)	Field Name	Size (hex bytes)
0	cf_ntask	2
2	cf_nprior	2
4	cf_nque	2
6	cf_nalarm	2
8	cf_npart	2
A	cf_nresrc	2
C	cf_ltick	2
E	cf_lslice	2
10	cf_cisr	4
14	TIS 1	10



**Table B-10:** Task Initialization Structure (TIS)

Offset (hex)	Field Name	Size (hex bytes)
0	ti_id	2
2	ti_pri	2
4	ti_start	4
8	ti_usp	4
C	ti_tsen	2
E	ti_fill	2

**Table B-11:** Event Control Block (ECB)

Offset (hex)	Field Name	Size (hex bytes)
0	e_tick	2
2	e_resum	2
4	e_cps	2
6	e_fill	2
8	QID 1	2
A	QID 2	2
C	(zero)	2

**Table B-12:** System Buffer Header (SBH)

Offset (hex)	Field Name	Size (hex bytes)
0	sb_nxte	4
4	sb_pree	4
8	sb_thse	4
C	sb_nxtb	4
10	sb_pid	2
12	sb_dlen	2
14	sb_disp	2
16	sb_dmod	2

**Table B-13:** Resource Carrier Message

Offset (hex)	Field Name	Size (hex bytes)
0	SBH	18
18	rc_qid	2
1A	rc_resum	2
1C	rc_pri	2
1E	rc_comp	2
20	rc_token	4

**Table B-14:** Global System Table (GST)

Offset (hex)	Field Name	Size (hex bytes)
0	gs_init	4
4	gs_ticks	4
8	gs_idle	4
C	gs_ramend	4
10	gs_version	4
14	gs_debug	4
18	gs_unused	E8

---

Appendix

**C**

## System Call Summaries

This appendix provides quick-reference tables showing summaries of the system call parameters for C and assembly language interfaces. [Table C-1](#) shows the C interface. [Table C-2](#) shows the assembly language interface.

**Table C-1: C Interface System Call Summary**

Operation	C System Call
Create a task	s_tcreat ( tis )
Delete calling task	s_tdelet ( )
Disable task rescheduling	s_lock ( )
Enable task rescheduling	s_unlock ( )
Suspend calling task	s_susp ( ecb, event_code )
Resume a task	s_resum ( task_id )
Create a queue	s_qcreat ( queue_id, q_type, task_id, qcb )
Delete a queue	s_qdelet ( queue_id )
Post a message to a queue	s_post ( queue_id, head_tail, message )
Post a message and resume queue owner	s_postr ( queue_id, head_tail, message )
Accept a message from a queue	s_accpt ( queue_id, message )
Create a resource	s_rcreat ( res_id, res_type )
Delete a resource	s_rdelet ( res_id, queue_id )
Request a resource	s_rreq ( res_id, carrier, token )
Cancel a resource request	s_rcan ( res_id, carrier )
Release a resource	s_rrel ( res_id, token )
Create a partition	s_pcreat ( part_id, buffer_size, start_addr, end_addr, pcb )
Delete a partition	s_pdelet ( part_id )
Request a buffer	s_breq ( part_id, buffer )
Release a buffer	s_brel ( buffer )
Create an alarm	s_acreat ( alarm_id, alarm_type, mask, signal, flag, acb )
Delete an alarm	s_adele ( alarm_id )
Set an alarm	s_aset ( alarm_id, ticks )
Cancel an alarm	s_acan ( alarm_id )
Initialize OS	s_osinit ( config )
Get system address table	s_getsat ( )
Return from ISR	s_iRET ( )
Set interrupt level	s_iset ( mask )

Note: A completion code is returned from most routines, not shown in this table. See [Chapter 3](#).

**Table C-2: Assembly Interface System Call Summary**

Operation	Assembly System Call						
	I/O	Register					
		D0	D1	D2	D3	A0	A1
Create a task	input	0x00				tis	
Delete calling task	input	0x01					
Disable task rescheduling	input	0x02					
Enable task rescheduling	input	0x03					
Suspend calling task	input	0x04				ecb	
	output		event				
Resume a task	input	0x06	task_id				
Create a queue	input	0x07	que_id	type	tsk_id		
	output					qcb	
Delete a queue	input	0x08	que_id				
Post message to queue	input	0x09	que_id	hd_tail		msg	
Post message and resume queue owner	input	0x0A	que_id	hd_tail		msg	
Accept message from queue	input	0x0B	que_id				
	output					msg	
Create a resource	input	0x0C	res_id	type			
Delete a resource	input	0x0D	res_id	que_id			
Request a resource	input	0x0E	res_id			carrier	
	output					token	
Cancel resource request	input	0x0F	res_id			carrier	
Release a resource	input	0x10	res_id			token	
Create a partition	input	0x11	par_id	size		start	end
Delete a partition	input	0x12	par_id				
Request a buffer	input	0x13	par_id				
	output					buffer	
Release a buffer	input	0x14				buffer	
Create an alarm	input	0x15	alr_id	type	mask	signal	flag
	output					acb	

**Table C-2:** Assembly Interface System Call Summary (Cont'd)

Operation	Assembly System Call						
	I/O	Register					
		D0	D1	D2	D3	A0	A1
Delete an alarm	input	0x16	alr_id				
Set an alarm	input	0x17	alr_id	ticks			
Cancel an alarm	input	0x18	alr_id				
Initialize OS	input	0x19				config	
Get system address table	input	0x1A					
	output	table					
Return from ISR				-- TRAP #1 --			
Set interrupt level	input	new_mask		-- TRAP #2 --			
	output	old_mask					

Notes: All routines are accessed with TRAP #0 unless otherwise noted.  
 Most routines return completion status in register d0, not shown in this table.  
 See [Chapter 3](#) for details.

## Task Scheduling Examples

This appendix provides some examples of task scheduling. For these examples, assume that the system consists of the following tasks:

<b>Task</b>	<b>Priority</b>	<b>Time Slicing</b>
A	1	disabled
B	2	disabled
C	2	disabled

### *Example 1*

1. Task B is currently executing and no tasks are scheduled for execution.
2. Task B makes a Resume system call to schedule task C. Task C is added to the dispatch queue, but is not dispatched because it is at the same priority as task B. (If time slicing were enabled for task B, task C would be dispatched at this point.)
3. Task B suspends and task C is dispatched.

### *Example 2*

1. Task A is suspended, task B is currently executing and task C is scheduled for execution.
2. Task B makes a Post & Resume system call to a queue owned by task A, causing task A to be scheduled. Since task A is higher priority, task B is preempted at com-

pletion of the system call and task A is dispatched. The preempted task is added to the tail of the dispatch queue for its priority. (Task C is at the head of that queue.)

3. Task A suspends and task C is dispatched.
4. Task C suspends and task B is dispatched, continuing execution at the return from the Post & Resume call to task A's queue.

*Example 3*

1. Tasks A and C are suspended and task B is currently executing. Task B owns a queue. It has checked this queue, and finding it empty, is preparing to suspend until awakened by a Post & Resume call to the queue.
2. An interrupt occurs before task B suspends. The interrupt service routine (ISR) makes a Resume system call to schedule task A. At completion of the ISR, task B is preempted, added to the dispatch queue for its priority, and task A is dispatched.
3. During task A's execution, it makes a Post & Resume call to task B's queue. Task B is already scheduled for execution, but a special "resume pending" bit is set in its state. (See [Section 4.1 on page 81.](#))
4. When task A suspends, task B is dispatched, continuing execution at the point of interrupt ([Step 2](#)). At the time of dispatch, task B is again added to the tail of the dispatch queue for its priority because of the "resume pending" flag.
5. Not realizing that the interrupt and subsequent preemption occurred, task B proceeds to suspend, waiting for a post to its queue. Task B does not specify "cancel previous scheduling" in its event control block (see the note below).
6. Task B, having been rescheduled during dispatch ([Step 4](#)) is dispatched again and finds the queue element that was posted during task A's execution ([Step 3](#)).



---

**Note**

In most cases when a task suspends, it expects to be scheduled by an asynchronous event — by another task or from an interrupt service routine. In these cases, “cancel previous scheduling” should never be specified in the event control block (ECB). Use of this flag is recommended only in the case of a task that suspends for some number of ticks, with expiration of that alarm being the only event that is to cause the task to be scheduled. In this special case, “disable resume” should also be specified in the ECB.

---



---



# Glossary of Acronyms

<b>ACB</b>	alarm control block
<b>API</b>	application program interface
<b>ECB</b>	event control block
<b>FIFO</b>	first in, first out
<b>ICP</b>	intelligent communications processor
<b>ISR</b>	interrupt service routine
<b>GST</b>	global system table
<b>LAN</b>	local-area network
<b>LIFO</b>	last in, first out
<b>PCB</b>	partition control block
<b>QCB</b>	queue control block
<b>RCB</b>	resource control block
<b>RTE</b>	return from exception
<b>SBH</b>	system buffer header
<b>SNMP</b>	simple network management protocol

<b>TCB</b>	task control block
<b>TCP/IP</b>	transmission control protocol/internet protocol
<b>TIS</b>	task initialization structure
<b>WAN</b>	wide-area network

---

# Index

## A

Alarm control block (ACB) [26, 39, 67, 86, 119](#)  
Alarm queue [116](#)  
    special [89](#)  
    standard [88](#)  
Alarm-related calls [67](#)  
    cancel an alarm (s\_acan) [74](#)  
    create an alarm (s\_acreat) [68](#)  
    delete an alarm (s\_adelet) [71](#)  
    set an alarm (s\_aset) [72](#)  
Alarms [26](#)  
    special [27](#)  
    standard [27](#)  
    task [27](#)  
Audience [11](#)

## C

Configuration table [91, 120](#)  
Customer support [14](#)

## D

Data structures [81](#)  
    alarm control block (ACB) [86, 119](#)  
    configuration table [91, 120](#)  
    dispatch queues [90, 120](#)  
    event control block (ECB) [95, 121](#)  
    field offsets [117](#)  
    global system table (GST) [103, 122](#)  
    partition control block (PCB) [85, 119](#)  
    queue control block (QCB) [83, 118](#)  
    resource carrier message [100, 122](#)  
    resource control block (RCB) [84, 118](#)  
    special alarm queue [89, 120](#)  
    stack format [102](#)  
    standard alarm queue [88, 119](#)

    system buffer header (SBH) [97, 121](#)  
    task control block (TCB) [81, 118](#)  
    task initialization structure (TIS) [94, 121](#)

## Debugging aids [105](#)

    global system table (GST) [105](#)  
    panic codes [106](#)  
    system variables [115](#)

## Dispatch queues [90](#)

## Documents

    reference [13](#)

## E

Ethernet [17](#)  
Event control block (ECB) [39, 95, 121, 128, 129](#)  
Examples of task scheduling [127](#)

## F

FDDI [17](#)  
Features  
    Freeway [17](#)  
Freeway  
    features [17](#)  
    overview [15](#)

## G

Global system table (GST) [103, 105, 106, 122](#)  
Glossary of acronyms [131](#)

## I

Interrupt service routine (ISR) [26, 128](#)

## L

LAN interface processor [15](#)

## M

### Memory

- allocation 24
- partitions 25
- stacks 24

### Miscellaneous system calls 75

- get system address table (s\_getsat) 77
- initialize OS (s\_osinit) 76
- return from ISR (s\_iret) 78
- set interrupt level (s\_iset) 79

## O

### Operation, principles of 21

### OS/Impact 15

- overview 18

### Overview 15

- Freeway 15
- OS/Impact 18

## P

### Panic codes 106

### Partition control block (PCB) 25, 61, 85, 119

### Partition-related calls 61

- create a partition (s\_pcreat) 62
- delete a partition (s\_pdelet) 64
- release a buffer (s\_brel) 66
- request a buffer (s\_breq) 65

### Preface 11

### Product

- support 14

## Q

### Queue control block (QCB) 23, 42, 83, 118

### Queue-related calls 42

- accept message from queue (s\_accept) 50
- create a queue (s\_qcreat) 43
- delete a queue (s\_qdelet) 45
- post message and resume (s\_postr) 48
- post message to queue (s\_post) 46

### Queues

- alarm 116
- dispatch 39, 82, 90, 95, 120, 127, 128
- doubly linked 23
- exchange 23, 39, 42, 83, 96
- singly linked 22

- special alarm 89, 120

- standard alarm 88, 119

## R

### Reference documents 13

### Resource carrier message 28, 51, 100, 122

### Resource control block (RCB) 28, 51, 84, 118

### Resource management 27

### Resource token 28, 51

### Resource-related calls 51

- cancel a resource request (s\_rcan) 57
- create a resource (s\_rcreat) 52
- delete a resource (s\_rdelet) 54
- release a resource (s\_rrel) 59
- request a resource (s\_rreq) 55

### rlogin 17

## S

### Scheduling 21

### Scheduling tasks

- examples 127

### Server processor 15

### SNMP 17

### Special alarm queue 89

### Stacks 24

- format 102

### Standard alarm queue 88

### Support, product 14

### System buffer header (SBH) 22, 46, 97, 121

### System calls 31

- See also* Alarm-related calls, Miscellaneous system calls, Partition-related calls, Queue-related calls, and Resource-related calls

### summaries 123

- assembly interface 125

- C interface 124

### System data structures

- See* Data structures

### System errors 106

### System variables 115

- alarm queues 116
- control structures 115
- task execution 116

**T**

- Task control block (TCB) [22](#), [81](#), [118](#)
- Task execution variables [116](#)
- Task initialization structure (TIS) [94](#), [121](#)
- Task scheduling examples [127](#)
- Task-related calls [32](#)
  - create a task (s\_tcreat) [34](#)
  - delete calling task (s\_tdelet) [36](#)
  - disable task rescheduling (s\_lock) [37](#)
  - enable task rescheduling (s\_unlock) [38](#)
  - resume a task (s\_resum) [41](#)
  - suspend calling task (s\_susp) [39](#)
- Tasks [22](#)
- TCP/IP [17](#)
- Technical support [14](#)
- telnet [17](#)
- Timers [25](#)
  - alarms [26](#)
  - interrupt service routine [26](#)
  - timer task [26](#)

**V**

- VxWorks [15](#)

**W**

- WAN interface processor [15](#)





## Customer Report Form

We are constantly improving our products. If you have suggestions or problems you would like to report regarding the hardware, software or documentation, please complete this form and mail it to Simpack at 9210 Sky Park Court, San Diego, CA 92123, or fax it to (619) 560-2838.

If you are reporting errors in the documentation, please enter the section and page number.

Your Name: \_\_\_\_\_

Company: \_\_\_\_\_

Address: \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

Phone Number: \_\_\_\_\_

Product: \_\_\_\_\_

Problem or  
Suggestion: \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

Simpact, Inc.  
Customer Service  
9210 Sky Park Court  
San Diego, CA 92123