VRTX32/68000 USER'S GUIDE

READY SYSTEMS

V

VRTX32/68000

READY
SYSTEMS

VRTX32/68000

Versatile Real-Time Executive for the M68000 Microprocessor

USER'S GUIDE

Software Release 1
Document Number 541311001
April 1987

REV.	MANUAL REVISION HISTORY	PRINT DATE
	Beta site edition	2/87
-001	First edition; release 1.04	2/89

Ready Systems makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Ready Systems assumes no responsibility for any errors that may appear in this document. The information in this document is subject to change without notice.

Copyright © 1989
Ready Systems
470 Potrero Avenue
P.O. Box 60217
Sunnyvale, California 94086
408/736-2600
FAX:408/736-3400
TELEX: 711510608 (domestic)
0231510608 (international)

All rights reserved
Printed in U.S.A.

Table of Contents

How to Use This Manual

Chapter 1 Overview

Chapter 2 Basic System Calls

Chapter 3 Interrupt Support

Chapter 4 Configuration and Initialization

Chapter 5 Support for User-Defined Extensions

Chapter 6 Interfacing Software Components

Chapter 7 System Call Reference

List of Illustrations

List of Tables

List of Examples

How to Use This Manual

HUNTER
READY
A Division of Ready Systems

Purpose of This Manual

This manual describes VRTX32, the high-performance Versatile Real-Time Executive. VRTX32 is a silicon software component that provides real-time, multitasking operating system functions for embedded microprocessor applications.

VRTX32/68000 is the implementation of VRTX32 designed for the Motorola MC68000, MC68008, and MC68010 microprocessors.

Intended Audience

This manual is for the application programmer who requires VRTX32's real-time executive functions to build a product. The programmer should be familiar with standard real-time operating system functions and the Motorola M68000 architecture.

How This Manual is Organized

The rest of this manual is organized as follows.

- Chapter 1 is an overview of the VRTX32 software component and its M68000 microprocessor family support.

- Chapters 2 through 4 describe the basic VRTX32 services.

- Chapters 5 through 6 describe integrating VRTX32 with user-supplied code and other components.

- Chapter 7 provides an alphabetical reference to the VRTX32 system calls describing each call's operation, input and output values, possible return codes, and possible environments.

- The appendices contain supporting material in quick reference format. Appendix D is an example of a VRTX32 application and a board support package.

Where to Start

This manual serves as an introduction and as a reference guide.

For an introduction to VRTX32/68000, read Chapter 1, Overview. Then read Chapters 2 through 4 for information about VRTX32's system calls.

Read Chapter 5, Support for User-Defined Extensions, and Chapter 6, Interfacing Software Components, for information about user-defined system call handlers, VRTX32 extensions, and integrating other components into a VRTX32 system.

To use this manual as a reference guide, look up any given VRTX32 system call in Chapter 7, System Call Reference. When you need additional information, refer to the earlier chapters.

Conventions

There are several conventions you should be aware of as you read the *VRTX32/68000 User's Guide.*

- Numbers preceded by the dollar sign ($) character are hexadecimal numbers; otherwise, numbers are decimal numbers.

- A notation such as D1[7:0] stands for register D1, bits 7 through 0; bit 0 is the least significant bit.

- In figures that show memory, low memory is at the top of the figure.

- In some figures, there are fields labeled "Reserved, must = 0". These fields *must* be zero.

- All code in this manual is in Motorola assembler format.

Related Documents

We recommend the following documents for additional information.

- *Getting Started With Silicon Software Components* provides details on the real-time software development process.

- *How to Write a Board Support Package for VRTX* describes the process of writing an interface between VRTX32 and your microprocessor board.

- Motorola's *M68000 16/32-bit Microprocessor Programmer's Reference Manual* provides specific Motorola M68000 references.
- For a discussion of using VRTX32/68000 with the C language, consult Ready Systems' *VRTX32 C User's Guide.*
- *Interfacing a Language to Silicon Software Components* provides guidelines for writing an interface that allows a high-level language to make system calls to Ready Systems' software components.

Questions/Suggestions

If you have questions about VRTX32/68000 that are not answered by this manual, contact the Ready Systems Service and Support Group. To give us suggestions about this manual, use the reader comment card at the back of the manual. If the card is missing, send the suggestions to Ready Systems Technical Publications. Contact us at this address:

Ready Systems
449 Sherman Avenue
P.O. Box 61029
Palo Alto, CA 94306-9991
415/326-2950
TELEX: 711510608 (domestic)
0231510608 (international)

Chapter 1

Overview

1.1 Introduction

VRTX32, the high-performance Versatile Real-Time Executive, is a silicon software component for embedded microprocessors. VRTX32 is designed to take advantage of the power and features typically available on 32-bit microprocessors. VRTX32/68000 is the implementation of VRTX32 designed for the Motorola MC68000, MC68008, and MC68010 microprocessors.

The following sections define terms basic to an understanding of VRTX32. These terms include silicon software components, embedded applications, and real-time executive. This chapter also gives an overview of VRTX32's features, configuration, and architecture.

1.1.1 Silicon Software Components

A **silicon software component** is an executable version of a microprocessor program that operates on all board-level microcomputers using the same type of microprocessor. Because a silicon software component does not have to be modified to make it work with custom board designs, it can be delivered in Read-Only Memory (ROM). In fact, a silicon software component is more like a hardware component than a traditional piece of software.

The critical concept introduced by silicon software components is the use of software as a building block to connect other pieces of software in a variety of designs, without modification.

1.1.2 Embedded Applications

An **embedded microprocessor** is buried inside a larger system, such as an intelligent terminal, a communications system, an analytical instrument, an industrial robot, or a peripheral controller. Embedded microprocessors are to be distinguished from stand-alone microcomputers, such as small business systems or word processors.

The software that runs on embedded microprocessors must meet a different set of requirements than software that runs on stand-alone systems. The most important requirement for embedded software systems is **real-time** responsiveness. The system must respond to unexpected events in the outside world rapidly enough to control ongoing processes. Another key requirement is **multitasking**. Multitasking is the ability of the software to handle many tasks concurrently, because events in the real world usually overlap rather than occur in strict sequence.

1.1.3 Real-Time Executive

A common set of mechanisms is necessary to support real-time systems. These mechanisms include such things as multitasking support, CPU scheduling, communication, and memory allocation. Programmers and designers of real-time systems frequently spend more time on these basic mechanisms than on the application program itself. In embedded applications, this set of mechanisms is called a **real-time operating system** or a **real-time executive**. Programmers build the application using the real-time executive as the foundation. VRTX32 is a real-time executive.

1.2 VRTX32 Features

There are three categories of VRTX32 features: real-time executive features, silicon software component features, and M68000 support.

1.2.1 Real-Time Executive Features

VRTX32 provides all the features required in a real-time executive:

- Multitasking support
- Event-driven, priority-based scheduling
- Intertask communication and synchronization
- Dynamic memory allocation
- Real-time clock control, with optional time-slicing
- Character I/O support
- Real-time responsiveness

With these features, VRTX32 provides a strong foundation for real-time, multitasking applications. VRTX32 frees designers and programmers from the problems of synchronizing multiple real-time tasks and allows them to focus their efforts on the application.

1.2.2 Silicon Software Component Features

VRTX32 provides these advantages:

- **Development environment independence**. VRTX32 consists entirely of an indivisible ROM component; its configuration is not dependent on any assemblers, linkers, loaders, or host environments.

- **Target environment independence**. VRTX32 requires only a CPU with a small amount of memory. This allows VRTX32 to provide true chip-level support for the M68000 family in a wide variety of embedded applications.

- **Extensibility**. You can easily integrate application-specific system-level software with VRTX32. This extended software can include user-defined system call handlers and user-supplied routines.

- **Position independence**. VRTX32 is written entirely in position-independent code. This means it can be positioned anywhere in the address space of the processor.

You can easily integrate additional silicon software components into the system. Other Hunter & Ready components include TRACER, a debugger; IOX, an input/output executive; and FMXs, file management executives. You can also supply your own components.

1.2.3 M68000 Support

VRTX32/68000 fully supports the Supervisor and User modes of the MC68000, MC68008, and MC68010 processors. VRTX32 (as well as user-defined extensions and interrupt handlers) executes entirely in Supervisor mode. User application tasks can execute in Supervisor mode or in User mode. Thus, target applications can support access-protection, data security, and memory management (through the use of a memory management unit).

1.3 VRTX32 Configuration

The M68000 architecture uses a data structure called the Exception Vector Table (EVT) to define the addresses of user-supplied interrupt and trap service routines. There are two EVT entries that link VRTX32 to its board environment. The first of these is a vector that points to the VRTX32 entry point, which is the starting address of VRTX32. A second EVT entry is a vector that points to the base of the VRTX32 Configuration Table.

The user-supplied VRTX32 Configuration Table and simple, device-specific interrupt handlers provide the interface between VRTX32 and its environment. With this configuration table, you specify all the parameters required by VRTX32 for a particular system environment.

Values in the configuration table specify the beginning and extent of system-managed memory, multitasking parameters, interrupt support, and linkage to other silicon software components. This table also describes the location of any user-supplied routines invoked by significant events such as task switching. See Figure 1-1, VRTX32 Configuration.

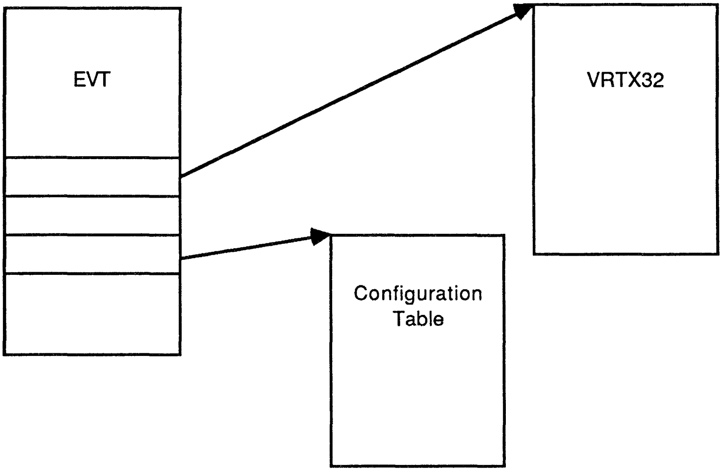


Figure 1-1 VRTX32 Configuration

1.4 VRTX32 Architecture

A system based on VRTX32 is layered according to function, with each level making use of the functions provided by the level below. See Figure 1-2, VRTX32 Architecture. The system hardware occupies the lowest level. The next level contains the simplest, most hardware-dependent operating system functions. On top are user-defined application programs.

In more technical terminology, each level defines a **virtual machine** for the level above it. At higher levels, the functions provided by a software level are not distinguishable from those provided by the hardware. Each software level adds several instructions to the processor's instruction set. For application programs, VRTX32 adds high-level instructions (system calls) to the architecture of the M68000 microprocessor.

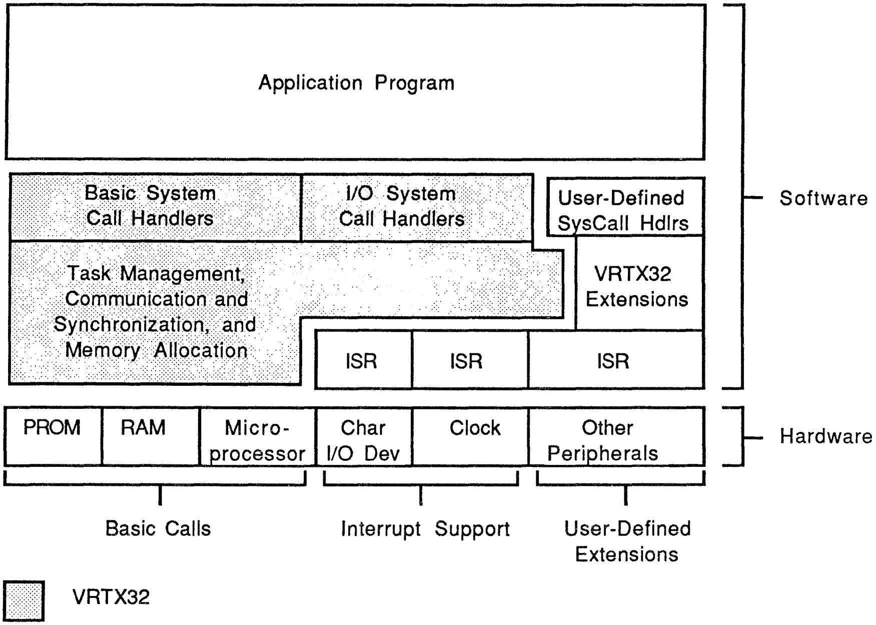


Figure 1-2 VRTX32 Architecture

The shaded area in Figure 1-2 shows VRTX32's operating system mechanisms. A few small pieces are missing between VRTX32 and the hardware. These missing pieces are interrupt service routines (ISRs), small hardware-dependent code segments that provide interrupt handling for particular peripherals. These are not supplied with VRTX32, but Hunter & Ready offers supplementary packages that contain the ISRs for several widely used peripheral devices such as counter-timers and serial I/O chips. Consult *How to Write a Board Support Package for VRTX* for more information.

Other operating system mechanisms that VRTX32 does not provide are shown to the right of VRTX32 in Figure 1-2. These include user-defined system call handlers and VRTX32 extensions. These mechanisms can, for example, initialize and save the state of special devices, such as a Fourier transform chip in a signal-processing application. Like ISRs, these pieces are connected to VRTX32 with software hooks to form a unified operating system. Some hooks are defined by entries in the configuration table. Refer to Chapter 4, Configuration and Initialization, and Chapter 5, Support for User-Defined Extensions.

The three horizontal braces shown at the bottom of Figure 1-2 divide the overall system architecture into three vertical sections, corresponding to three groups of VRTX32 mechanisms: mechanisms that support basic system calls, mechanisms that support interrupts, and mechanisms that support user-defined extensions. The following chapters discuss these mechanisms.

Chapter 2
Basic System Calls

HUNTER
READY
A Division of Ready Systems

2.1 Introduction

This chapter describes the process of making a VRTX32 system call and VRTX32's basic operations. These operations are organized into three categories:

- Multitasking management
- Memory allocation
- Intertask communication and synchronization

Figure 2-1, Basic Architecture, shows these three operations.

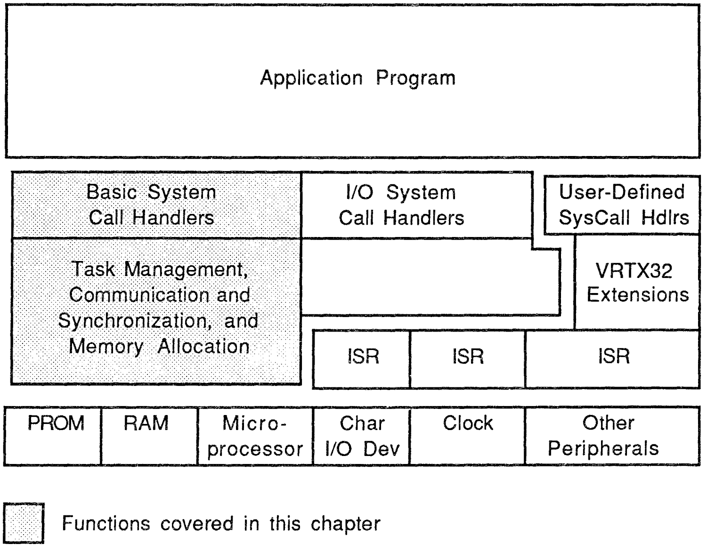

Application Program
Basic System Call Handlers
I/O System Call Handlers
User-Defined SysCall Hdlrs
Task Management, Communication and Synchronization, and Memory Allocation
VRTX32 Extensions
ISR
ISR
ISR
PROM
RAM
Micro-processor
Char I/O Dev
Clock
Other Peripherals
▨ Functions covered in this chapter

Figure 2-1 Basic Architecture

2.1.1 Accessing VRTX32

The M68000 architecture uses a data structure known as the **Exception Vector Table (EVT)** to control access to service routines for hardware-generated interrupts and software-generated traps. Your application accesses VRTX32 as a trap service routine through this same data structure.

The EVT is usually based at physical address 0. However, the MC68010 architecture allows you to locate the EVT at any address by setting the **Vector Base Register (VBR)**. There are 16 TRAP vectors in the EVT, numbered 0 through 15. See Figure C-1, Exception Vector Table, for the location of the TRAP vectors.

The application calls VRTX32 by issuing a **TRAP** instruction. The TRAP #n instruction generates a trap exception; **n** is the TRAP vector number. When this trap exception occurs, the CPU loads a new **Program Counter (PC)** value from the EVT. For the trap to VRTX32, this new PC value contains the address of the VRTX32 entry point.

You can choose any of the 16 TRAP numbers for VRTX32 access. Figure 2-2, VRTX32 TRAP Vector, shows the format of vector 32, the vector corresponding to TRAP #0, in the case where VRTX32 resides at physical address $1000. Vector 32 is located at EVT offset $080.

EVT Offset	
$80	$0000
$82	$1000

Figure 2-2 VRTX32 TRAP Vector

2.1.2 System Call Format

All VRTX32 system calls are made with the TRAP instruction. When calling VRTX32, register D0 must contain a 32-bit **function code** that specifies the desired VRTX32 system call. When a call completes, VRTX32 returns a 32-bit **return code** in register D0. When the call is successful, VRTX32 returns a value of zero; otherwise VRTX32 returns an **error code** (refer to Appendix B, Return Codes).

Additional parameters can be passed to VRTX32 in other registers. Nonaddress parameters are passed in data registers D1 through D4, and address parameters are passed in address register A0. Unless otherwise indicated, VRTX32 system calls leave all input registers except D0 intact. Refer to Appendix A, System Call Summary, which lists all VRTX32 calls with their input parameters and output results.

Table 2-1, VRTX32 System Calls, lists all VRTX32 system calls and their function codes.

2.2 Tasks

Real-time systems are designed to perform seemingly unrelated functions in a nonsequential way, using the processor and I/O devices as efficiently as possible. Several common processing situations lend themselves to this control philosophy. Examples include listening for input from several devices at the same time, reading or writing a block of data while concurrently performing arithmetic computations, and implementing communications applications.

VRTX32 supports real-time systems with a set of basic **multitasking mechanisms**. The basic unit controlled by VRTX32 is the **task**, a logically complete path of user code. The task is a collection of actions that deals with one issue asynchronously and in real time. Several tasks can operate autonomously from the same piece of code, or tasks can be located in separate code modules. In a multitasking system, several tasks appear to execute simultaneously, although VRTX32 actually allocates CPU control among tasks in an interleaved fashion.

Tasks are **active** or **inactive**. Inactive tasks are dormant tasks, while active tasks have executing, suspended, and ready **task states**. There can be as many active tasks as the application requires. All active tasks have priority levels, and 255 of the active tasks can have unique identification numbers. VRTX32 moves tasks from one task state to another based on the task priority level and as the result of system calls.

The task's Task Control Block (TCB) and the task stack maintain task status information for each active task not in control of the CPU.

Tasks can create other tasks and they can delete, suspend, resume, inquire about the status, and change the priority of themselves or of other tasks. Tasks can also "lock" critical sections of their code so that they are not preempted by other tasks. You can create Supervisor mode tasks or User mode tasks. In Supervisor mode, tasks can use the full instruction set of the M68000 microprocessor.

Table 2-1 VRTX32 System Calls

Mnemonic	Function Code	System Call
Task Management		
SC_TCREATE	$0000	Create Task
SC_TDELETE	$0001	Delete Task
SC_TSUSPEND	$0002	Suspend Task
SC_TRESUME	$0003	Resume Task
SC_TPRIORITY	$0004	Change Task Priority
SC_TINQUIRY	$0005	Task Status Inquiry
SC_LOCK	$0020	Disable Task Rescheduling
SC_UNLOCK	$0021	Enable Task Rescheduling
Memory Allocation		
SC_GBLOCK	$0006	Get Memory Block
SC_RBLOCK	$0007	Release Memory Block
SC_PCREATE	$0022	Create Memory Partition
SC_PEXTEND	$0023	Extend Memory Partition
Communication and Synchronization		
SC_POST	$0008	Post Message to Mailbox
SC_PEND	$0009	Pend for Message from Mailbox
SC_ACCEPT	$0025	Accept Message from Mailbox
SC_QPOST	$0026	Post Message to Queue
SC_QJAM	$001E	Jam Message to Queue
SC_QPEND	$0027	Pend for Message from Queue
SC_QACCEPT	$0028	Accept Message from Queue
SC_QCREATE	$0029	Create Message Queue
SC_QECREATE	$001F	Create FIFO Message Queue
SC_QINQUIRY	$002A	Queue Status Inquiry
SC_FCREATE	$0017	Create Event Flag Group
SC_FDELETE	$0018	Delete Event Flag Group
SC_FPOST	$001A	Post Event to Event Flag Group
SC_FPEND	$0019	Pend on Event Flag Group
SC_FCLEAR	$001B	Clear Event
SC_FINQUIRY	$001C	Event Flag Group Inquiry

(continued on next page.)

Table 2-1, continued

Mnemonic	Function Code	System Call
Communication and Synchronization, continued		
SC_SCREATE	$002B	Create Semaphore
SC_SDELETE	$002C	Delete Semaphore
SC_SPOST	$002E	Post Unit to Semaphore
SC_SPEND	$002D	Pend on Semaphore
SC_SINQUIRY	$002F	Semaphore Inquiry
Interrupt Support		
UI_ENTER	$0016	Enter Interrupt Handler
UI_EXIT	$0011	Exit Interrupt Handler
Real-Time Clock		
SC_GTIME	$000A	Get Time
SC_STIME	$000B	Set Time
SC_TDELAY	$000C	Delay Task
SC_TSLICE	$0015	Enable Round-Robin Scheduling
UI_TIMER	$0012	Announce Timer Interrupt
Character I/O		
SC_GETC	$000D	Get Character
SC_PUTC	$000E	Put Character
SC_WAITC	$000F	Wait for Special Character
UI_RXCHR	$0013	Received-Character Interrupt
UI_TXRDY	$0014	Transmit-Ready Interrupt
Initialization		
VRTX_INIT	$0030	Initialize VRTX32
VRTX_GO	$0031	Start Application Execution

The rest of this section talks about these concepts in more detail and discusses VRTX32's multitasking management calls.

2.2.1 Task States and State Transitions

In a multitasking environment, tasks exist in and are moved between one of four states: executing, ready for execution, suspended, or dormant.

This section discusses the VRTX32 system calls that affect a task's state. Refer to the following sections and chapters for detailed explanations of these calls.

Executing Task State. An **executing** task has control of the CPU and is executing its instruction path. Only one task executes at a time.

Suspended Task State. A **suspended** task is suspended in mid-execution, and is waiting to be readied by a system call or an event.

A task can suspend for any of these reasons:

- A task suspend call, SC_TSUSPEND, specifies that task either by priority or by ID number. A task can suspend itself.

- The task issues an SC_TDELAY call and suspends for a specified time interval.

- The task issues an SC_PEND or SC_QPEND call, but no message from a task or an interrupt handler is waiting at the mailbox or queue.

- The task issues an SC_FPEND call, but the correct event flag(s) are not set.

- The task issues an SC_SPEND call, but the resource's semaphore has a zero value (the resource is not available).

- The task issues an SC_WAITC call and waits for an I/O device to send a special character.

- The task issues an SC_GETC call, but the input buffer maintained by VRTX32 is empty. The task waits for a character to be put into the buffer.

- The task issues an SC_PUTC call, but the output buffer is full. The task waits for a character to be removed from the buffer.

To find out why a task is suspended, you can issue the SC_TINQUIRY call to read the Status field in the task's TCB.

It is important to note that suspensions are independent and additive. For example, when a task is suspended while waiting for a message and it is also explicitly suspended by another task, both suspending conditions must be removed before the task is ready for execution.

A final point about the suspended task state is that when a task suspends, VRTX32 notes the current interrupt level. When this task is resumed, interrupts are enabled to the level they were enabled when the task was suspended.

Ready Task State. A **ready** task is one that is ready for execution; for example, a task that has just been created is ready for execution. However, a ready task cannot gain control of the CPU until all higher-priority tasks in the ready or executing state either complete, suspend, or become dormant.

A task can move from the suspended state to the ready state for any of these reasons:

- An SC_TRESUME call readies a task suspended by an SC_TSUSPEND call.
- A time delay expires, which can ready a task suspended by an SC_TDELAY call, or a task that timed out pending for a message.
- An SC_POST or SC_QPOST call posts a message to a task that is waiting on a mailbox or queue.
- An SC_FPOST call posts an event(s) to an event flag group and a task was waiting for that event(s).
- An SC_SPOST call indicates that the resource is available and a task was waiting on that resource's semaphore.
- An interrupt service routine (ISR) sends a special character (with the UI_RXCHR call) to VRTX32. VRTX32 then transfers the character to a task suspended by an SC_WAITC call.
- An ISR sends a character to the input buffer with the UI_RXCHR call. The tasks suspended on an empty buffer are readied in the order they were suspended, one at a time, with each succeeding UI_RXCHR call.
- An ISR retrieves a character from the output buffer with the UI_TXRDY call. The tasks suspended on a full buffer are readied in the order they were suspended, one at a time, with each succeeding UI_TXRDY call.

A task moves from the executing state to the ready state when a higher-priority task becomes ready to execute. CPU control then passes to the higher-priority task.

Because the task that loses control is not suspended, no bits are set in the TBSTAT field of its TCB. The task remains in the ready state until all higher-priority tasks complete, suspend, or become dormant, at which point the task moves back to the executing state.

For example, when a task is executing and a higher-priority task's SC_TDELAY interval expires, the higher-priority task gains control. This moves the lower-priority task from the executing state to the ready state.

This transfer of control from one task to another is called a **task switch**. Refer to Section 2.2.2, Task Scheduling, for more information about task switching, and Appendix E, The Rescheduling Procedure, for information about the process that leads to task switches.

Dormant Task State. A **dormant** task is a task that is not initialized, or a task whose execution is terminated (task deleted). No TCB is assigned to it.

Tasks are in the dormant state before they are created; they reenter the dormant state when they are deleted with an SC_TDELETE call. When all user tasks are deleted or suspended, the system switches to the idle task until an external event occurs.

Figure 2-3, Task State Transitions, shows these task states.

2.2.2 Task Scheduling

VRTX32 schedules and manipulates tasks based on each task's identification number and priority.

Each task has a unique **identification (ID) number** that allows it to be selectively readied, suspended, or deleted. You specify the task ID number when you create the task: either a unique ID number from 1 to 255, or an ID of zero, which indicates that no ID is assigned. (Any number of tasks with an ID of zero can exist.)

VRTX32 schedules control of the CPU based on the highest-priority task that is ready to execute. A task's priority is determined by two things:

- the priority level you assign to the task when it is created
- the order tasks are made ready among equal-priority tasks

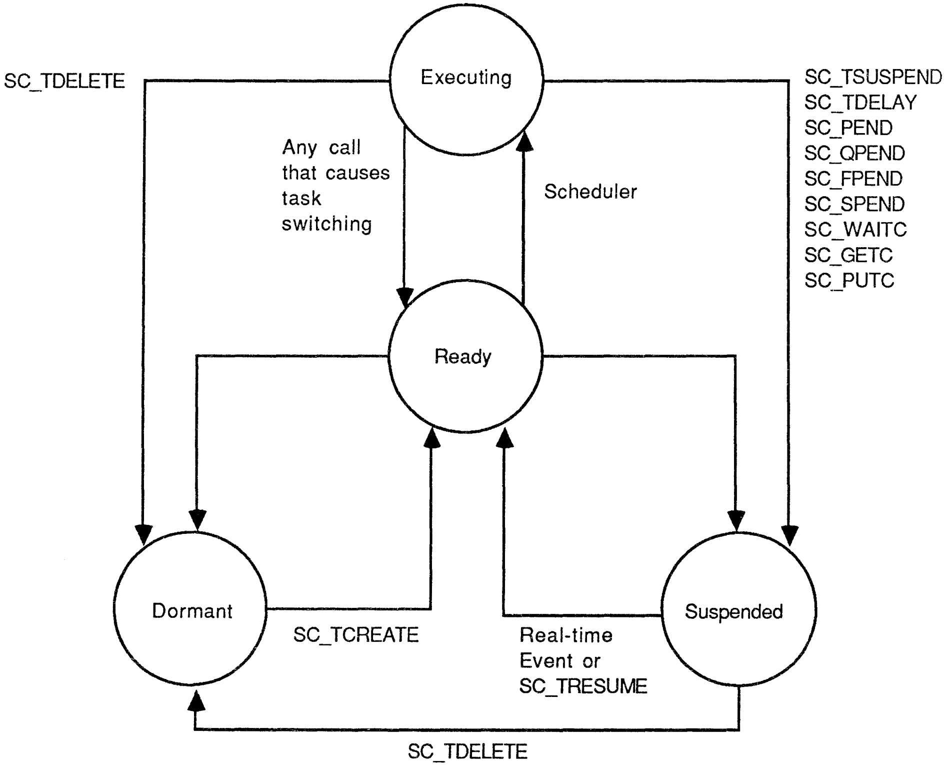


Figure 2-3 Task State Transitions

You must specify a **priority level** for each task when it is created. There are 256 priority levels ranging from zero to 255; zero is the highest-priority level. Any number of active tasks can exist at each priority level.

In a group of equal-priority tasks, tasks execute in the order that they become ready (first-in/first-out (FIFO) order). In other words, the "oldest" ready task is the highest-priority task in its priority group. (Refer to Section 2.2.1, Task States and State Transitions, for a discussion of ready tasks.) For example, the task that was created first executes before newer ready tasks of the same priority.

VRTX32 initiates the **rescheduling procedure** to ensure that the highest-priority task is executing. (Refer to Appendix E, The Rescheduling Procedure, for more information about this procedure.) You do not execute special system calls to accomplish task switching once initialization is complete and system execution is under way.

The highest-priority task executes until the task terminates its own operation, the task suspends, or a higher-priority task is ready to execute. When one of these events occurs, the rescheduling procedure determines the next task to move from the ready state to the executing state.

The rescheduling procedure can be disabled with the SC_LOCK call, and reenabled with the SC_UNLOCK call.

The way tasks are scheduled can be altered with the change task priority (SC_TPRIORITY) call, the delay task (SC_TDELAY) call, and the enable time-slice (SC_TSLICE) call. The SC_TPRIORITY call changes the priority of a task. A task switch can occur if the new priority of the affected task is higher than that of the calling task, or if the task lowers its own priority and there is a ready task with a higher priority. The current task can also change the execution order of equal-priority ready tasks by issuing SC_TPRIORITY with the "new" priority equal to the "old" priority. This makes the affected task ready *after* the other members of its priority group. Note that the calling task can voluntarily preempt itself using this technique if there are equal-priority tasks that are ready to execute.

The SC_TDELAY call delays the calling task's execution for a specified number of VRTX32 clock ticks. A task can also use this call to voluntarily preempt itself; if the task specifies a zero delay value, it moves to the end of its priority group. The next equal-priority ready task executes.

The SC_TSLICE call enables optional round-robin scheduling among equal-priority tasks. At the end of a time-slice interval, or when a task in the priority group suspends, the tasks in the priority group rotate. The next ready task is given a chance to execute.

2.2.3 Task Control Block (TCB)

Because microprocessors can execute only one instruction at a time, tasks that appear to be executing in parallel are really executing in short, interleaved bursts. VRTX32 must therefore maintain status information about the contents of active registers for all tasks not in control of the CPU.

This information is stored in the task's **Task Control Block (TCB)** and on the task's stack. The TCB is a data structure in the VRTX32 Workspace. Each active task has a TCB, but no TCB is defined for a dormant task. See Figure C-2, Task Control Block, for a diagram of the TCB.

A task's TCB is frozen while the task is executing and is not altered until the task moves to the ready, suspended, or dormant state. When the task moves to the ready or suspended state, the TCB saves the contents of registers D0 through D5, A0 through A3, SSP and USP, as well as other status information about the task. The task's stack saves registers D6, D7, A4 through A6, PC, and the **Status Register (SR)**. When the task moves to the suspended state, the TBSTAT field in the TCB indicates the reason for suspension. When the task moves to the dormant state, its TCB is no longer associated with that task.

2.2.4 Task Management Support

VRTX32 manages tasks with these calls:

SC_TCREATE	Task Create
SC_TDELETE	Task Delete
SC_TSUSPEND	Task Suspend
SC_TRESUME	Task Resume
SC_TPRIORITY	Task Priority Change
SC_TINQUIRY	Task Status Inquiry
SC_LOCK	Disable Task Rescheduling
SC_UNLOCK	Enable Task Rescheduling

The SC_TCREATE call creates a task with a priority level, an ID number, a mode (Supervisor or User), and a specified start address. The task begins execution with interrupts enabled (interrupt level 0). The creator task's environment determines the new task's TCB and stack values.

It is possible to create a task with an ID of zero. However, it is a special case because other tasks cannot reference it. The SC_TDELETE, SC_TSUSPEND, SC_TPRIORITY, and SC_TINQUIRY calls reference the calling task when a zero ID is specified.

A task can remove one or more tasks, including itself, with the SC_TDELETE call. You can specify the task to be deleted by priority or ID number. The deleted task becomes dormant and the TCB is available for reuse.

The SC_TSUSPEND call suspends one or more tasks by priority or ID number. A task can suspend itself. When a task is suspended, a flag in the TCB's TBSTAT field is set to indicate the reason for suspension. An explicitly suspended task is not resumed until an SC_TRESUME call is issued.

The SC_TRESUME call resumes the execution of one or more tasks previously suspended by an SC_TSUSPEND call. You can specify the tasks to be resumed by priority or ID number.

A task can change the priority of another task or of itself with the SC_TPRIORITY call. You specify the task by ID number. (You can change the priority of an explicitly suspended task, but it remains explicitly suspended until you issue an SC_TRESUME call.)

The SC_TINQUIRY call obtains the task ID number, priority level, and status information from the TCB of any task, including itself. We recommend that ISRs use this call only for general performance or statistical monitoring. When this call is made before any tasks are created, the data returned is invalid.

You can disable the rescheduling procedure with the SC_LOCK call. This can be useful when, for example, there is a critical section of code that higher-priority tasks must not preempt. The task that issues the SC_LOCK call retains processor control, even though higher-priority tasks may be ready to run. The SC_UNLOCK call reenables the rescheduling procedure. However, it cancels only the last SC_LOCK call. (The maximum lock/unlock nest count supported is 65,535.)

> **CAUTION**
>
> Any call that suspends the current task causes unpredictable results when task rescheduling is disabled with the SC_LOCK call.

2.2.5 Multitasking Management Calls

Table 2-2 contains a summary of the system calls that control the multitasking environment. All calls described in this section can lead to the rescheduling procedure, except for SC_LOCK and SC_TINQUIRY; refer to Appendix E, The Rescheduling Procedure, for more information. For detailed information on each of the calls, refer to Chapter 7, System Call Reference.

Table 2-2 Task Management Call Summary

SC_TCREATE	Creates a task with a specified priority, ID number, mode, and address.
SC_TDELETE	Deletes one or more tasks specified by priority or ID number.
SC_TSUSPEND	Suspends one or more tasks specified by priority or ID number.
SC_TRESUME	Resumes one or more tasks specified by priority or ID number.
SC_TPRIORITY	Changes the priority of a task specified by ID number.
SC_TINQUIRY	Obtains the ID number, priority, TCB address, and status of a task specified by ID number.
SC_LOCK	Disables task rescheduling until SC_UNLOCK is issued.
SC_UNLOCK	Enables task rescheduling.

2.3 Memory

The M68000 microprocessor's 24-bit PC and its 24 address lines define an address space of 16 megabytes (2^{24} bytes), although the actual amount of memory in the system is often considerably less. The memory map of a VRTX32-based system consists of these modules:

- **VRTX32 code**: the VRTX32 PROM set. Note that VRTX32 can be placed in random access read/write memory, and can be loaded into memory from disk, if you wish.

- **User load module**: the software package you are responsible for developing, assembling, linking, and placing in the execution environment.

- **VRTX32 Workspace**: contains system variables, TCBs, and stacks.

- **VRTX32-managed user memory**: one or more partitions, or pools, of memory blocks that tasks and ISRs dynamically acquire and release.

- Optional Hunter & Ready components, such as IOX, FMX, or TRACER.
- Optional user-supplied components.

Figure 2-4, Memory Organization, is an overview of the entire memory organization of a VRTX32 system. The shading shows what can be burned into ROM; everything else must be in dynamic read/write memory.

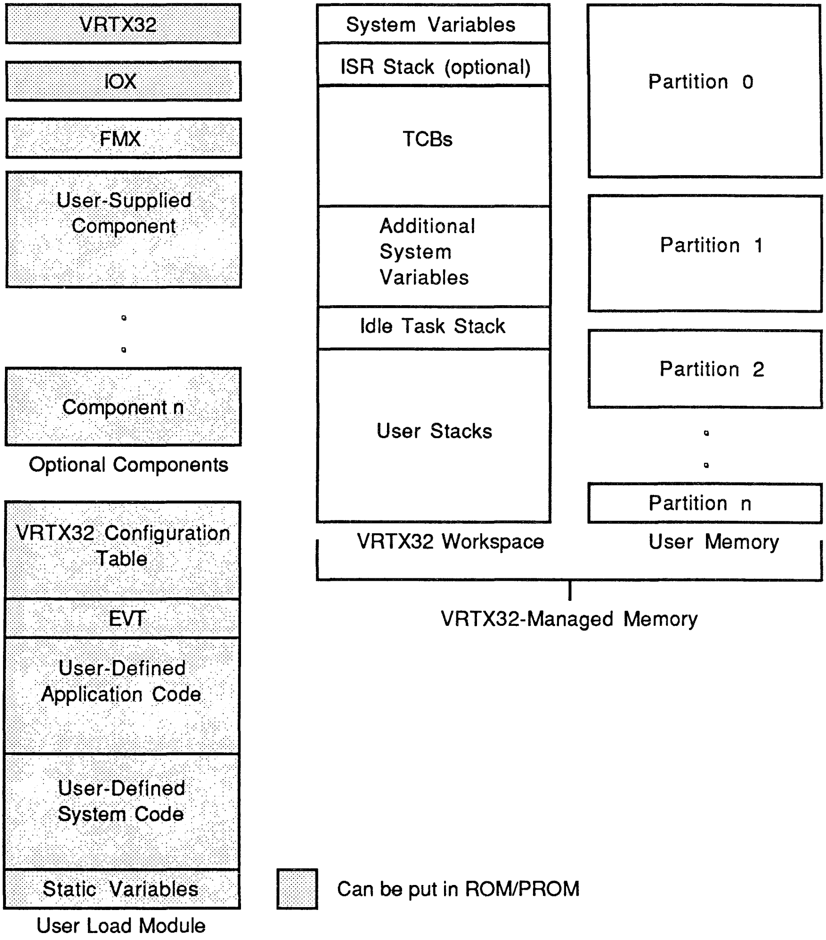


Figure 2-4 Memory Organization

The **user load module** holds your application code and any user-defined, system-level code, as described in Chapters 3 through 6. In addition, the user load module contains the EVT, the configuration table, and any static variables associated with your application or with system code. Refer to Appendix D, An Example, for an example of a user load module.

The **VRTX32 Workspace** contains the system variables, the optional interrupt stack, the TCBs, control structures for queues, event flag groups, and semaphores, control structures for VRTX32-managed user memory, the idle task stack, and stack areas for each task in the system. VRTX32 is responsible for setting up and managing the stacks and for initializing and managing the TCBs.

The **VRTX32-managed user memory** consists of one or more **partitions**, or chunks, of memory. These partitions can be noncontiguous. Each partition is subdivided into one or more fixed-size blocks of memory that can be allocated dynamically. The rest of this section describes how VRTX32 manages user memory and its own workspace.

2.3.1 Memory Allocation

A task's demand for memory varies over the course of its execution, and different tasks usually have different requirements. The operating system treats memory as a resource and allocates that resource among competing tasks, just as it allocates control of the CPU among competing tasks.

Multitasking executives generally use one of two approaches to memory allocation: **static allocation** of fixed-size memory blocks or **dynamic allocation** of variable-size blocks. In static allocation, each task is assigned a block of memory at system initialization. This block is dedicated to that one task and cannot be used by any other task. In dynamic allocation of variable-size memory blocks, available memory eventually becomes fragmented and unusable as tasks allocate and release memory blocks from the available pool.

One technique for allocating variable-size blocks is the buddy system, widely used in non-real-time systems. In this technique, a single piece of memory is split in half. Each half is the buddy of the other half. The piece of memory is repeatedly split in half until a block size appropriate to the request is created. When the memory request is for a unit larger than any of those currently available, the system attempts to combine a smaller unit with its buddy into a larger contiguous unit. This scheme suffers from indeterminacy, a serious flaw for real-time applications.

A problem with the buddy system and other variable-size memory allocation systems is that as memory grows progressively more fragmented, occasions inevitably arise when a request cannot be met. Even though there can be enough total free memory, it can be so fragmented that a large enough contiguous block cannot be found. These occasions cannot be predicted in advance and compensated for, because the order of memory requests usually cannot be anticipated in a real-time system. This design introduces an element of unpredictability into the total system behavior beyond that of the external environment. This additional unpredictability is unsatisfactory in real-time systems, because real-time systems cannot tolerate a memory system that works only some of the time.

2.3.2 Memory Allocation Support

The designers of VRTX32 felt static allocation was too restrictive, but the memory compaction resulting from dynamic allocation led to unacceptable indeterminacy and imposed too much system overhead. Thus, the VRTX32 memory allocation mechanism is a compromise between the two schemes. The VRTX32 memory management schemes are determinate, yet they allow flexibility in the sizes of the stacks allocated to each task and in the sizes of the partitions that divide user memory.

In the VRTX32 Configuration Table, you specify the starting address and size of VRTX32 Workspace, the optional interrupt stack's size, the maximum number of tasks that can exist at any one time, and each task's stack area size. The VRTX32 Workspace must be large enough to contain VRTX32 system variables, the optional interrupt stack, one TCB for each task, and stack areas for every task in the system. In addition, the VRTX32 Workspace must be large enough to accommodate a control block for each memory partition and extension, and a control block for each defined message queue. Refer to Section 4.3, Determining VRTX32 Workspace Size, for details.

When a task is created, VRTX32 automatically allocates the task's fixed-size stack (or stacks) in the VRTX32 Workspace. The stack size is specified by the User-Stack-Size and Sys-Stack-Size parameters in the configuration table. (Refer to Chapter 4, Configuration and Initialization.)

You can bypass this allocation when you want to manage stacks with a user-supplied routine invoked at task create time; for example, when you want every task to have a different stack size. Refer to Section 5.3, VRTX32 Extensions, for information about user-supplied routines.

A User mode task is given two separate stacks, referenced by the **User Stack Pointer (USP)** and **Supervisor Stack Pointer (SSP)** (the TBUSP and TBSSP fields in the task's TCB). A Supervisor mode task is given a single stack, referenced by the SSP (the TBSSP field in the task's TCB).

VRTX32 also dynamically allocates partitions of user memory. You can dynamically define partitions to match the often noncontiguous chunks of memory that make up the actual physical organization of memory. Each partition of user memory has blocks of a fixed size set when that partition is created.

VRTX32 manages its memory allocation system with these system calls:

SC_GBLOCK	Get Memory Block
SC_RBLOCK	Release Memory Block
SC_PCREATE	Create Memory Partition
SC_PEXTEND	Extend Memory Partition

The SC_PCREATE call defines a contiguous area of user memory as a partition. Parameters passed with the call specify the partition start address, the partition size, the partition ID number, and the block size. There cannot be more than 32K blocks in the partition as first defined by the SC_PCREATE call. However, you can extend the partition with the SC_PEXTEND call. The block size must not equal zero and must be less than or equal to the partition size. To avoid wasted space, the partition size should be an integer multiple of the block size.

The SC_GBLOCK call acquires a block of memory from the partition. You can repeat this call until all blocks in the partition are allocated.

The SC_RBLOCK call releases a block of memory back to the partition. A task's blocks are not automatically released when the task is deleted, because blocks can be passed to other tasks for data exchange. Therefore, you should use the SC_RBLOCK call to release all blocks before you delete a task.

The SC_PEXTEND call enlarges a previously-defined partition to include an additional range of memory locations. There cannot be more than 32K blocks in an extension. However, you can issue multiple SC_PEXTEND calls to define more blocks. The extension and the original partition do not have to be contiguous with each other.

Since all memory blocks in a partition are the same size, no fragmentation results from dynamic memory allocation; consequently, no memory compaction is required.

The VRTX32 partition/block system has several key features. These features give VRTX32's memory allocation system great flexibility and most of the advantages of a variable-size block system, without the indeterminacy and excessive system overhead. First, you can define partitions in other partitions to achieve different block sizes. For example, one partition can be entirely in a single block of another partition. This means that blocks can easily be divided into sub-blocks. Second, you can define two partitions to cover the same area of memory, allowing you to allocate blocks of two different sizes from the same memory region. The only requirement here is that you must release all blocks of one size before you allocate any blocks of the other size.

Figures 2-5, VRTX32 Workspace, and 2-6, User Memory Managed by VRTX32, show how memory is subdivided.

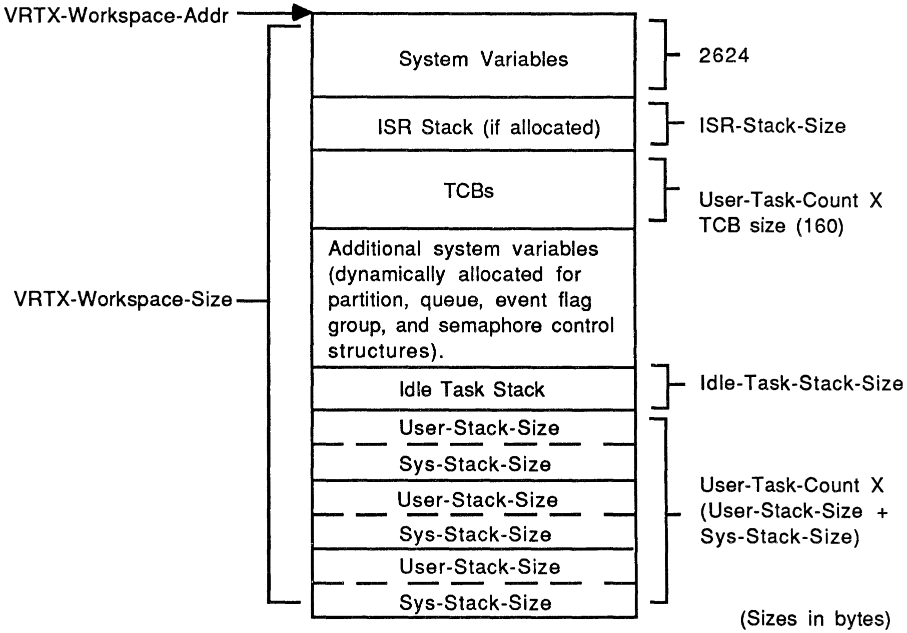


Figure 2-5 VRTX32 Workspace

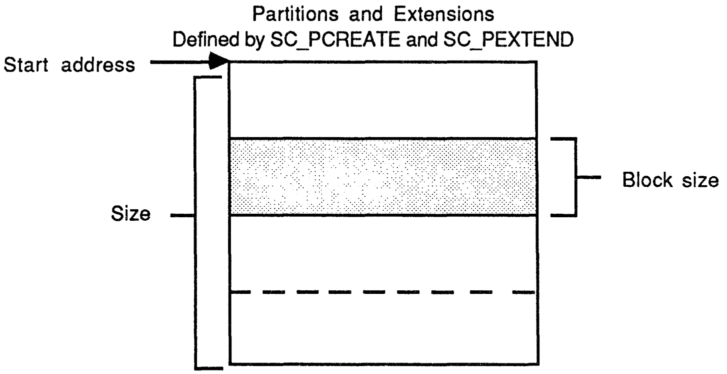

Partitions and Extensions
Defined by SC_PCREATE and SC_PEXTEND
Start address
Size
Block size

Figure 2-6 User Memory Managed by VRTX32

2.3.3 Memory Allocation Calls

Table 2-3 contains a summary of the system calls that allow programs to obtain and return blocks of memory from a specified partition. The summary also includes the VRTX32 calls that create and extend partitions. These calls do not initiate the rescheduling procedure, and so cannot cause a task switch. For detailed information on each of the calls, refer to Chapter 7, System Call Reference.

Table 2-3 Memory Allocation Call Summary

SC_GBLOCK	Gets a memory block from a specified partition.
SC_RBLOCK	Releases a memory block back to the specified partition.
SC_PCREATE	Creates a memory partition of a specified size, ID number, block size, and address.
SC_PEXTEND	Extends a specified partition with a specified extension size and address.

2.4 Intertask Communication and Synchronization

A real-time multitasking system has several communication and synchronization needs:

- A task must be able to exchange data with other tasks and ISRs.
- A task must be able to synchronize with other tasks and ISRs, in these ways:
  - Unilateral synchronization: a task synchronizes with another task or an ISR.
  - Bilateral synchronization: two tasks synchronize with each other.
  - Conjunctive synchronization: a task synchronizes with several events.
  - Disjunctive synchronization: a task synchronizes with the first of several possible events.
- Tasks must occasionally be able to mutually exclude each other so that each is guaranteed exclusive control of a protected resource.

VRTX32 provides several mechanisms to meet these needs. Tasks and ISRs can pass long-word (32-bit) messages using mailboxes and queues to meet all the above needs. Messages can be significant in themselves, or pointers to larger messages. Tasks and ISRs can also use event flags for synchronization, and semaphores for mutual exclusion.

2.4.1 Mailboxes

A **mailbox** is a user-defined long-word variable in user read/write memory. Mailboxes allow tasks to pass long-word (32-bit) nonzero messages. VRTX32 does not create mailboxes; instead, you set up the memory for the mailbox. The application should initialize the mailbox to the appropriate value: zero when the mailbox is immediately available; nonzero when the mailbox is used for mutual exclusion.

These are VRTX32's mailbox system calls:

SC_POST	Post Message to Mailbox
SC_PEND	Pend for Message from Mailbox
SC_ACCEPT	Accept Message from Mailbox

A task or an ISR sends a message to a specified mailbox with the SC_POST call. If a message is already in the mailbox (mailbox value is nonzero), VRTX32 returns an error code.

To receive the message, another task issues an SC_PEND call. If there is a message in the mailbox (mailbox value is nonzero), the task receives the message and continues execution. VRTX32 resets the mailbox to zero when the message is received.

If there is no message in the mailbox (mailbox value is zero), the task attempting to receive a message with SC_PEND suspends until the message arrives. You can specify a nonzero timeout value that allows the task to resume execution if no message arrives during that time period.

When the task attempts to receive the message with an SC_ACCEPT call and no message is present, the task does not suspend. Instead, VRTX32 returns an error code and the task continues execution. To avoid suspension, ISRs must use SC_ACCEPT rather than SC_PEND to receive messages.

When a task pending at a mailbox is suspended with the SC_TSUSPEND call, it can receive a message. However, the task remains suspended until it is resumed with the SC_TRESUME call.

More than one task can wait at the same mailbox if each task issues an SC_PEND call with the same mailbox address. When a message is sent to that mailbox, the highest-priority task receives the message and is placed in the ready state. (Tasks receive messages according to their priority level at the time they pend on the mailbox. Changing a pended task's priority does not affect the order in which messages are allocated.)

You can use VRTX32's mailbox calls for data transfer, synchronization, and mutual exclusion. To synchronize two tasks with each other, Task A posts a message to one mailbox, then immediately pends at another mailbox. Task B simply does the reverse: it receives Task A's message, then immediately posts a message to enable Task A.

To perform mutual exclusion of a protected resource, you "lock" the resource by initializing a mailbox to any nonzero "key" value. Every task that needs to use that resource pends at the mailbox for the key. As each task finished with the resource, it posts the key back to the mailbox to enable the next task.

2.4.2 Queues

Message queues are fixed-length buffers that you create dynamically. Queues are not part of your set of variables; they are VRTX32-managed structures referenced by a queue ID number.

Queues allow tasks to pass long-word (32-bit) messages. (A queue of length 1 behaves logically like a mailbox.)

These are VRTX32's queue system calls:

SC_QCREATE	Create Message Queue
SC_QECREATE	Create FIFO Message Queue
SC_QPOST	Post Message to Queue
SC_QJAM	Jam Message to Queue
SC_QPEND	Pend for Message from Queue
SC_QACCEPT	Accept Message from Queue
SC_QINQUIRY	Queue Status Inquiry

You create a queue in VRTX32's Workspace with the SC_QCREATE or SC_QECREATE call, specifying the queue ID number and the queue size. Tasks pend on a queue created with SC_QCREATE in priority order. With the SC_QECREATE call, you specify whether they pend in priority or FIFO order.

Tasks and ISRs send messages to queues with the SC_QPOST and SC_QJAM calls. The SC_QPOST call puts the messages at the end of the queue; messages are handled in first-in/first-out (FIFO) order. If the queue is full, an error code is returned.

SC_QJAM puts the message at the beginning of the queue. When a queue is created with SC_QCREATE or SC_QECREATE, VRTX32 adds one queue entry to the number you specify. This additional entry is reserved at the beginning of the queue for a message posted with the SC_QJAM call when the queue is otherwise full. If the queue is full and a message has already been "jammed", an error code is returned. As an alternative to mixing SC_QJAMs and SC_QPOSTs, you can use the SC_QJAM call to post all messages to the queue. In this case, you can use the full size of the queue (including the reserved entry), and messages are handled in last-in/first-out (LIFO) order.

Tasks receive messages with SC_QPEND or SC_QACCEPT. If a task attempts to receive a message with SC_QPEND and the queue is empty, the task suspends. You can specify a nonzero timeout value that allows the task to resume execution if no message arrives during that time period. If a task attempts to receive a message from an empty queue with SC_QACCEPT, it is not suspended. Instead, VRTX32 returns an error code and the task continues execution. To avoid suspension, ISRs must use SC_QACCEPT.

When two or more tasks pend at an empty priority-order queue, the task with the highest priority receives the first message sent to the queue. (Tasks receive messages according to their priority level at the time they pend on the queue. Changing a pended task's priority does not affect the order in which messages are allocated.) When two or more tasks pend at an empty FIFO-order queue, the task that pended first receives the first message sent to the queue.

The SC_QINQUIRY call obtains information about a queue. This call returns the number of messages in the queue and the message at the head of the queue. This message is returned to the caller but is not removed from the queue.

You can use queues for mutual exclusion of several resources of the same type. Assign each type of resource, such as a line printer, a specific queue. The length of this queue should be equal to the number of resources in that resource type, such as the number of line printers on the system. This length determines how many tasks can use the resource type at the same time.

For example, suppose there are five line printers in the system. A priority-order line printer queue of length five locks these printers. This line printer queue is initialized with printer ID numbers. All tasks attempting to use a line printer pend at the line printer queue. When a printer becomes available, the highest-priority pended task receives that printer's ID number and uses the printer. When the task finishes with the line printer, it posts the printer ID number back to the line printer queue. This enables another task to use that printer.

2.4.3 Event Flags

An **event flag group** is a global, long-word (32-bit) structure in VRTX32 Workspace. Each of the 32 bits in the event flag group is an **event flag**. Event flags have two states: set (one) and cleared (zero). When a flag is set, the associated event has occurred. This means that tasks and ISRs can use event flags to signal the occurrence of events to other tasks.

Event flags provide these synchronization features:

- A task can wait for a **disjunctive** (OR) set of events to occur. In other words, a task specifies a set of events to wait for. When the first one occurs, the task is readied.
- A task can wait for a **conjunctive** (AND) set of events to occur. This means that a task specifies a set of events to wait for, and is not readied until all the events have occurred.
- Many tasks can be waiting for the same event. This means that a task or an ISR "broadcasts" the event to all the tasks waiting for it to occur.

These are VRTX32's event flag system calls:

SC_FCREATE	Create Event Flag Group
SC_FDELETE	Delete Event Flag Group
SC_FPEND	Pend on Event Flag Group
SC_FPOST	Post Event to Event Flag Group
SC_FCLEAR	Clear Event
SC_FINQUIRY	Event Flag Group Inquiry

The SC_FCREATE call creates a 32-bit event flag group in VRTX32 Workspace, and returns the event flag group ID number. Each event flag group and semaphore is associated with a **control block**. You specify the maximum number of control blocks in the VRTX32 Configuration Table (refer to Section 4.2, VRTX32 Configuration Table). If you try to create more event flag groups and/or semaphores than you've specified in the configuration table, VRTX32 returns an error code.

The SC_FDELETE call deletes an event flag group, making its control block available for reuse. There may be tasks pending on the event flag group; you can specify whether to delete only if there are no tasks pending, or to force a delete. In the latter case, all pending tasks are readied.

Tasks wait for one or more events with the SC_FPEND call. The task specifies whether it is an AND pend or an OR pend. If the specified event flags are set, the task continues execution (the SC_FPEND call does not clear the event flags). If the specified event flags are not set, the task suspends. You can specify a nonzero timeout value that allows the task to resume execution if the event does not occur during that time period. If the task is suspended on an event flag group and the group is deleted, the task is readied and VRTX32 returns an error code.

To satisfy an AND pend, all specified event flags must have a value of one simultaneously. For example, suppose a task is waiting for both Flag 1 and Flag 2. Flag 1 is set, but is immediately cleared. Next, Flag 2 is set. The task continues to pend, because Flag 1 and Flag 2 have not had a value of one at the same time.

Tasks and ISRs signal one or more events with the SC_FPOST call. Tasks suspended on the event flag group are readied if the SC_FPOST call satisfies their AND or OR pend. If an event flag is already set (one), and SC_FPOST tries to set it again, VRTX32 returns an error code. However, if SC_FPOST specifies several event flags, and some of them are already set, VRTX32 returns an error code *and* sets any event flags that were not previously set.

The SC_FCLEAR call clears event flags. An event flag should be cleared before an attempt is made to post to it again.

A task or an ISR can check the status of event flags by issuing the SC_FINQUIRY call. The entire 32-bit event flag group is returned to the caller.

2.4.4 Semaphores

VRTX32 provides **counting semaphores** for mutual exclusion. A counting semaphore is a word (16-bit) variable in VRTX32 Workspace that has an initial value from 0 to 65,535. An initial value of zero indicates that the resource starts in a locked state. A nonzero value indicates how many tasks can access the resource at one time.

These are VRTX32's semaphore system calls:

SC_SCREATE	Create Semaphore
SC_SDELETE	Delete Semaphore
SC_SPEND	Pend on Semaphore
SC_SPOST	Post Unit to Semaphore
SC_SINQUIRY	Semaphore Inquiry

The SC_SCREATE call creates a semaphore in VRTX32 Workspace, and returns the semaphore ID number. You specify the initial value of the semaphore and whether tasks pend on the semaphore in priority order or FIFO order.

Each semaphore and event flag group is associated with a **control block**. You specify the maximum number of control blocks in the VRTX32 Configuration Table (refer to Section 4.2, VRTX32 Configuration Table). If you try to create more semaphores and/or event flag groups than you've specified in the configuration table, VRTX32 returns an error code.

The SC_SDELETE call deletes a semaphore, making its control block available for reuse. There may be tasks pending on the semaphore; you can specify whether to delete only if there are no tasks pending, or to force a delete. In the latter case, all pending tasks are readied.

To wait for the restricted resource, a task issues the SC_SPEND call. If the semaphore has a nonzero value, the semaphore is decremented and the task continues execution. If the semaphore is zero, the task suspends. You can specify a nonzero timeout value that allows the task to resume execution if the resource does not become available during that time period. If the task is suspended on the semaphore and the semaphore is deleted, the task is readied and VRTX32 returns an error code.

A task or an ISR signals that the resource is available with the SC_SPOST call. This call increments the semaphore; however, if a task is waiting, it is readied immediately and the semaphore is not incremented. If a semaphore receives an SC_SPOST when its value is already at the maximum of 65,535, an overflow occurs and VRTX32 returns an error code.

When two or more tasks pend at a priority-order semaphore, the task with the highest priority is readied with the next SC_SPOST call. (Tasks are readied according to their priority level at the time they pend on the semaphore. Changing a pended task's priority does not affect the order in which tasks are readied.) When two or more tasks pend at a FIFO-order semaphore, the task that pended first is readied with the next SC_SPOST call.

Tasks and ISRs can check the value of a semaphore with the SC_SINQUIRY call.

2.4.5 Communication and Synchronization Calls

Table 2-4 contains a summary of the system calls used for message exchange, synchronization, and mutual exclusion. The pending, posting (including SC_QJAM), and deleting calls initiate the rescheduling procedure; refer to Appendix E, The Rescheduling Procedure, for more information. For detailed information on each of the calls, refer to Chapter 7, System Call Reference.

Table 2-4 Communication and Synchronization Call Summary

SC_POST	Posts a message to a specified mailbox.
SC_PEND	Pends for a message from a specified mailbox. You can specify an optional time limit.
SC_ACCEPT	Accepts a message from a specified mailbox, but does not suspend the caller if no message is present.
SC_QPOST	Posts a message to a queue specified by ID number.
SC_QJAM	Jams a message to the beginning of a queue specified by ID number.
SC_QPEND	Pends for a message from a queue specified by ID number. You can specify an optional time limit.
SC_QACCEPT	Accepts a message from a queue specified by ID number, but does not suspend the caller if no message is present.
SC_QCREATE	Creates a queue with a specified ID number and a specified number of queue entries.
SC_QECREATE	Creates a queue with a specified ID number and a specified number of queue entries. You specify whether tasks pend in priority or FIFO order.
SC_QINQUIRY	Obtains the number of messages and the contents of the first message in a specified queue. The message is not extracted from the queue.
SC_FCREATE	Creates an event flag group and returns the event flag group ID number to the caller.
SC_FDELETE	Deletes an event flag group specified by ID number. You can specify to delete only if there are no tasks pending on the event flag group, or to force a delete and ready all pending tasks.

(continued on next page.)

Table 2-4, continued

SC_FPOST	Posts one or more events to an event flag group specified by ID number.
SC_FPEND	Pends for one or more events (AND or OR) from a specified event flag group. You can specify an optional time limit.
SC_FCLEAR	Clears one or more event flags in a specified event flag group.
SC_FINQUIRY	Obtains the specified event flag group.
SC_SCREATE	Creates a semaphore with an initial value and returns the semaphore ID number to the caller. You specify whether tasks pend in priority or FIFO order.
SC_SDELETE	Deletes a semaphore specified by ID number. You can specify to delete only if there are no tasks pending on the semaphore, or to force a delete and ready all pending tasks.
SC_SPOST	Posts a unit to a semaphore specified by ID number.
SC_SPEND	Pends for a unit from a specified semaphore. You can specify an optional time limit.
SC_SINQUIRY	Obtains the current value of the specified semaphore.

Chapter 3

Interrupt Support

3.1 Introduction

The only assumption VRTX32/68000 makes about its target environment is that an M68000 microprocessor with some random access (read/write) memory is present. You supply any hardware-dependent service routines required to initialize special devices and to service interrupts.

This chapter discusses **interrupt service routines (ISRs)** and VRTX32's support of these routines. In addition, this chapter discusses VRTX32's support for optional counter-timer and console character I/O devices.

Hunter & Ready offers documentation containing ISRs for many widely used devices. Consult *How to Write a Board Support Package for VRTX* for more information.

Figure 3-1, Interrupt Architecture, shows the functions covered in this chapter.

3.2 Interrupt Service Routines (ISRs)

A real-time system must respond quickly to externally generated interrupts to successfully interact with the external environment. VRTX32 provides the means for user-supplied ISRs, also called **interrupt handlers**, to communicate with and influence the scheduling of critical tasks. In contrast to application tasks, which are scheduled synchronously by VRTX32, an interrupt handler is executed asynchronously and is not scheduled by VRTX32. An interrupt handler executes when its hardware interrupt is generated.

ISRs typically do only the necessary actions required to service the interrupt. Input data, output data, or control information is passed to task level for further processing.

The following sections describe the Exception Vector Table, entering and exiting ISRs, ISR format, the system calls allowed from ISRs, stack allocation considerations, and M68000 interrupt levels.

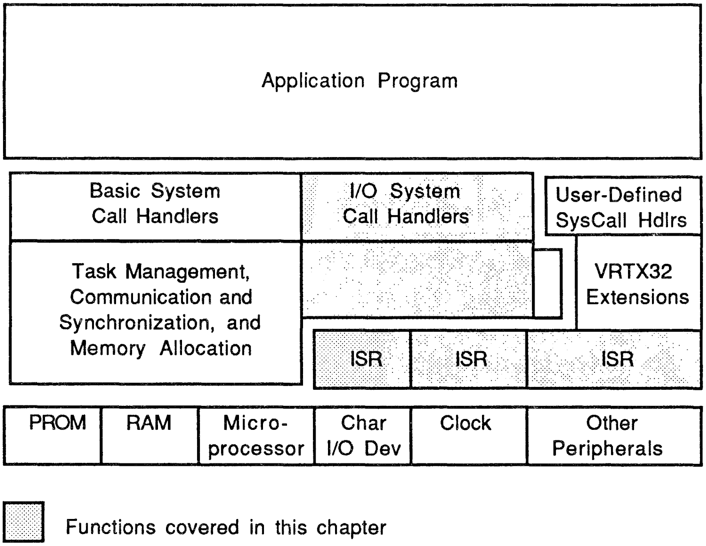

Application Program
Basic System Call Handlers
I/O System Call Handlers
User-Defined SysCall Hdlrs
Task Management, Communication and Synchronization, and Memory Allocation
VRTX32 Extensions
ISR
ISR
ISR
PROM
RAM
Micro-processor
Char I/O Dev
Clock
Other Peripherals
▨ Functions covered in this chapter

Figure 3-1 Interrupt Architecture

3.2.1 The Exception Vector Table

The M68000 architecture uses the Exception Vector Table (EVT) to control access to all service routines for hardware-generated interrupts and software-generated traps.

When a device interrupts the microprocessor, the M68000 hardware automatically pushes the current program status information (the SR and PC registers, as well as the format/ID word for the MC68010 microprocessor) onto the Supervisor stack, switches to the Supervisor stack, and jumps to the appropriate vector in the EVT. You supply the EVT vectors and the service routines associated with each interrupt and trap that can occur in the system. See Figure C-1, Exception Vector Table.

3.2.2 Entering and Exiting an ISR

Interrupt handling is separate from the multitasking environment managed by VRTX32. ISRs are entered directly without intervention from VRTX32. When the hardware detects an interrupt, all multitasking activity ceases and control passes to the designated ISR. This switching of control from tasks to ISRs does not result in any VRTX32 overhead.

Each ISR must include code that saves and restores any registers used during its execution. This guarantees that the interrupted code, whether VRTX32, the application, or another ISR, does not have its environment disturbed.

Both ISRs and tasks can make system calls. VRTX32 must distinguish between system calls made from ISRs and system calls made from application tasks. This is important because tasks can preempt other tasks, but a task must not preempt an ISR. Therefore, VRTX32 requires mechanisms to identify user-supplied ISR code and to defer task rescheduling until completion of all ISR activity. The UI_EXIT system call and hardware **interrupt levels** provide these mechanisms. Refer to Section 3.2.7, M68000 Interrupt Levels and VRTX32.

The UI_EXIT call signals the end of an ISR that makes system calls. UI_EXIT guarantees that any significant event communicated to the tasking environment from the ISR results in the execution of the highest-priority task on return to the task level.

Hunter & Ready has optimized UI_EXIT's performance to minimize ISR execution time. A system call made from the ISR can ready a higher-priority task than the one interrupted (for example, by posting a message to a mailbox or a queue where the higher-priority task is pended). In this case, the UI_EXIT call initiates the rescheduling procedure, which results in a task switch. When a system call from an ISR does not ready a higher-priority task, VRTX32 immediately returns control to the interrupted task.

Nested interrupts are interrupts that occur during the execution of an ISR. With nested interrupts, VRTX32 returns control to the task environment only after all the ISRs have completed. When one nested ISR readies a higher-priority task with a system call, a task switch occurs after the last nested UI_EXIT call. Sometimes an ISR that communicates with tasks using VRTX32 calls interrupts another ISR. In this case, both ISRs must use UI_EXIT to ensure correct functioning of the VRTX32 rescheduling procedure.

As described in Section 3.2.6, Interrupt Stack Switching, VRTX32 provides an optional stack-switching feature, allowing the stack memory requirements for interrupt servicing to be consolidated into a single area. If this option is enabled, every ISR that concludes with a UI_EXIT call must also begin with a corresponding UI_ENTER call. The UI_ENTER call signals the start of an ISR, just as the UI_EXIT call signals its completion. VRTX32 uses the UI_ENTER/UI_EXIT pair to keep track of interrupt nesting levels to determine when to switch stacks.

If interrupt stack switching is not enabled, the UI_ENTER call is optional. Like UI_EXIT, the UI_ENTER call is optimized for fast performance. You do not have to use UI_EXIT (or UI_ENTER) if the ISR does not make component calls and if it is not interrupted by another ISR that makes component calls.

3.2.3 Format of an Interrupt Service Routine

The format of a typical ISR is shown in Example 3-1.

VRTX32 is not invoked when an interrupt occurs; therefore, VRTX32 itself cannot save any registers on an interrupt. This means the ISR must include code to save and restore the contents of any registers it modifies.

The example saves registers D1 and D2 on the stack at the beginning of the ISR, and restores the registers just before returning to the task environment.

Interrupt stack switching is enabled in this example, so the UI_ENTER call must be used to signal the beginning of the ISR. This means that D0 must be saved before the UI_ENTER call is made. (Refer to Section 3.2.6, Interrupt Stack Switching, for more information.) The UI_EXIT call automatically restores the D0 register, using the value on the top of the stack.

CAUTION

If interrupt stack switching is enabled, save only D0 on the stack before issuing the UI_ENTER call. If additional registers are saved before the UI_ENTER call, they are saved on the wrong stack and are not restored properly.

Example 3-1 Format of a Typical Interrupt Service Routine (ISR)

```
UIFENTER    EQU $16         * UI_ENTER function code
UIFEXIT     EQU $11         * UI_EXIT function code
VRTX        EQU $00         * VRTX32 trap number

INTERRUPT_NUM:
   MOVE.L  D0,-(SP)         * save D0
   MOVEQ.L #UIFENTER,D0     * call UI_ENTER
   TRAP    #VRTX
   MOVEM.L D1-D2,-(SP)      * save registers
*    .
*    .                        interrupt servicing
*    .
   MOVEM.L (SP)+,D1-D2   * restore registers
   MOVEQ.L #UIFEXIT,D0   * call UI_EXIT to restore
   TRAP    #VRTX         *   D0 and return (with
*                            possible rescheduling
*                            if at task level)
```

3.2.4 VRTX32 Calls Allowed from ISRs

ISRs coordinate with tasks by conveying significant events to the multitasking environment. Usually, ISRs use the post calls to communicate messages or events to tasks; they use the accept calls to receive messages from tasks. However, an ISR can make most VRTX32 calls, as Table 3-1 shows.

In general, VRTX32 calls issued from ISRs have the same effect as calls issued from task level. To illustrate, consider the use of SC_POST from an ISR.

The SC_POST call allows the ISR to ready tasks and transmit messages to them. SC_POST deposits a double-word (32-bit) message in a specified mailbox. The contents of the mailbox must be zero at the time SC_POST is invoked, or the system considers the mailbox already in use. When the task for which the message is intended has issued an SC_PEND call at the appropriate mailbox, the task state changes from suspended to ready. In other words, the task status changes as a result of the SC_POST call and not as a result of the UI_EXIT executed at the end of the ISR. However, if a task switch is to occur, it occurs after the last UI_EXIT call in a series of nested ISRs.

When more than one task is waiting for a message at this mailbox, only the highest-priority task receives the message and is readied. When no task has issued an SC_PEND call for the message, the SC_POST simply posts the message to the mailbox so it can be retrieved later. The UI_EXIT call exits the ISR and a task switch occurs only when the newly readied task has higher priority than the interrupted task.

Table 3-1 VRTX32 Calls Allowed from ISRs

Task Management

SC_TSUSPEND
SC_TRESUME
SC_TINQUIRY
SC_LOCK
SC_UNLOCK

Memory Management

SC_GBLOCK
SC_RBLOCK

Communication and Synchronization

SC_POST
SC_ACCEPT
SC_QPOST
SC_QJAM
SC_QACCEPT
SC_QINQUIRY
SC_FPOST
SC_FCLEAR
SC_FINQUIRY
SC_SPOST
SC_SINQUIRY

Interrupt Support

UI_ENTER
UI_EXIT

Real-Time Clock

SC_GTIME
SC_STIME
UI_TIMER

Character I/O

UI_RXCHR
UI_TXRDY

3.2.5 VRTX32 Calls Not Allowed from ISRs

ISRs must not make system calls that affect the integrity of the task, queue, or partition control block chains. Also, ISRs cannot make calls that can cause suspension of the currently executing environment. These restrictions include several VRTX32 calls, as Table 3-2 shows.

> **CAUTION**
>
> ISRs for Level 7 interrupts and ISRs for interrupts greater than Component-Disable-Level cannot make any system calls, except for UI_ENTER and UI_EXIT calls if ISR stack switching is enabled. Refer to Section 3.2.7, M68000 Interrupt Levels and VRTX32, for more information.

3.2.6 Interrupt Stack Switching

When an interrupt occurs with an interrupt level greater than the current level (refer to Section 3.2.7, M68000 Interrupt Levels and VRTX32), the corresponding ISR gains control of the system. This ISR uses the stack of the interrupted task. When a User mode task is interrupted, the ISR uses that task's Supervisor mode stack.

This event can present a problem in applications with limited memory, since enough stack space must be allocated to all tasks to accommodate the Supervisor mode stack requirements of ISRs. This includes all possible nested ISRs; a common design error in real-time applications is allowing too little stack space for nested interrupts. To eliminate this inefficient use of stack memory, VRTX32 allows the dedication of a single stack for use by ISRs only. However, systems with high performance needs or a high frequency of interrupts can avoid the stack switching overhead of a dedicated interrupt stack and run ISRs off the current task's Supervisor mode stack.

The ISR-Stack-Size parameter in the configuration table specifies the size of the interrupt stack. If a special interrupt stack is desired, this parameter should be set to the size of the interrupt stack in bytes. If no special interrupt stack is needed, and ISRs should use the current task's Supervisor mode stack, then this parameter should be set to zero. See Figure 4-1, VRTX32/68000 Configuration Table.

VRTX32 allocates the interrupt stack at VRTX_INIT time in the VRTX32 Workspace. If an interrupt stack is specified, ISRs that are to use this stack should be bracketed with the UI_ENTER and UI_EXIT system calls, whether or not they make system calls.

Table 3-2 VRTX32 Calls Not Allowed from ISRs

Task Management

SC_TCREATE
SC_TDELETE
SC_TPRIORITY

Memory Management

SC_PCREATE
SC_PEXTEND

Communication and Synchronization

SC_PEND
SC_QPEND
SC_QCREATE
SC_QECREATE
SC_FCREATE
SC_FDELETE
SC_FPEND
SC_SCREATE
SC_SDELETE
SC_SPEND

Real-Time Clock

SC_TDELAY
SC_TSLICE

Character I/O

SC_GETC
SC_PUTC
SC_WAITC

Initialization

VRTX_INIT
VRTX_GO

When UI_ENTER is invoked, it switches to the interrupt stack. UI_EXIT restores the stack pointer of the interrupted task, or of the newly readied task in the case where a task switch occurs. VRTX32 performs interrupt stack switching only for the first interrupt in a series of nested ISRs. The rest of the nested ISRs then use the interrupt stack without any additional switching.

ISRs with interrupt levels greater than Component-Disable-Level and ISRs with interrupt level 7 do not have to use the interrupt stack, even if interrupt stack switching is enabled. If these ISRs use the interrupt stack, bracket them with the UI_ENTER and UI_EXIT calls. However, ISRs with interrupt levels *less* than Component-Disable-Leve! must always use the interrupt stack if interrupt stack switching is enabled. In other words, these ISRs must be bracketed with UI_ENTER and UI_EXIT if interrupt stack switching is enabled. (Refer to Section 3.2.7, M68000 Interrupt Levels and VRTX32, for information about interrupt levels and the Component-Disable-Level.)

3.2.7 M68000 Interrupt Levels and VRTX32

The M68000 microprocessor provides seven levels of interrupt priorities numbered from 1 to 7; 7 is the highest priority. There is also level 0, which indicates that the code is not an ISR.

The SR contains a three-bit **interrupt mask** field. This field indicates the **current interrupt level** of the processor. Tasks usually run at interrupt level 0, and ISRs usually run at higher levels.

When an interrupt occurs with an interrupt level greater than the current level, the M68000 sets the interrupt mask field in the SR to the level of the acknowledged interrupt. When the ISR completes execution, the CPU restores the interrupt level to the level that existed before the interrupt.

While the ISR is executing, interrupts from external devices are disabled for all interrupt levels less than or equal to the current level. Level 7 interrupts, however, are nonmaskable interrupts that cannot be disabled. (VRTX32 calls cannot be made from level 7.)

Interrupt Disabling. VRTX32 must prevent certain kinds of events from occurring while it is engaged in executing critical code. In particular, it must prevent ISRs from executing system calls during these critical intervals. VRTX32 does this by disabling interrupts during critical regions.

VRTX32 disables interrupts by raising the current interrupt level to the value specified by the **Component-Disable-Level** parameter in the configuration table. All interrupts less than or equal to Component-Disable-Level are disabled. (However, if Component-Disable-Level is 7, interrupts of level 7 are acknowledged.) Interrupts *greater* than Component-Disable-Level are not disabled. Thus, the interrupt latency is essentially eliminated for high-priority interrupts.

Determine Component-Disable-Level by the interrupt level of the highest-priority ISR that makes VRTX32/OS system calls, *plus one.* The default is a Component-Disable-Level of 7.

ISRs greater than Component-Disable-Level and ISRs with interrupt level 7 must not issue VRTX32/OS system calls (except the UI_ENTER and UI_EXIT calls if interrupt stack switching is enabled), because they can occur during a critical VRTX32 region. ISRs that make system calls must run at a level greater than 0 and less than Component-Disable-Level. ISRs that make no system calls can run at any level greater than 0.

Normally, tasks should run at level 0. When a task needs to disable interrupts, it must raise its level to the Component-Disable-Level by changing the SR (only Supervisor mode tasks can do this).

Changing the ISR Interrupt Level. The hardware that activates the interrupt, usually a jumper setting on the processor board, determines the initial interrupt level of an ISR. Normally, ISRs run at this level to completion.

Because the M68000 disables all interrupts with a level less than or equal to the current level, it is sometimes desirable for an ISR to lower its own level. This allows the CPU to process other interrupts from devices with the same interrupt level. If you use this tactic, we recommend that the level of the hardware interrupt be at least 2, so that the ISR can lower its level without reducing it all the way to 0 (which leads to confusion with task code).

3.2.8 Interrupt Support Calls

Table 3-3 contains a summary of the interrupt management system calls. The UI_EXIT call can initiate the rescheduling procedure, which results in a task switch. Refer to Appendix E, The Rescheduling Procedure, for more information. For detailed information on each of the calls, refer to Chapter 7, System Call Reference.

Table 3-3 Interrupt Support Call Summary

UI_ENTER	Enters an ISR if interrupt stack switching is enabled.
UI_EXIT	Exits an ISR. The rescheduling procedure can be initiated when rescheduling is called for and when the ISR is not nested. No return is made to the caller.

3.3 Integrated Support for Special Devices

Although VRTX32 operates without these devices, many VRTX32 applications require a counter-timer and a single-channel character I/O device. VRTX32 supports complete integration of these devices; you supply only a short hardware-dependent ISR for each device. In turn, VRTX32 manages all the logical operations required to provide application tasks with a repertoire of associated real-time clock management and character I/O commands.

Two categories of VRTX32 commands support character I/O and real-time clock management: calls from tasks that use the devices, and calls from ISRs that manage the devices. Table 3-4 shows these categories.

3.3.1 Real-Time Clock Support

VRTX32 maintains a 32-bit **VRTX32 clock**. This VRTX32 clock interfaces with the counter-timer device to provide a real-time clock for your system.

The VRTX32 clock supports the real-time clock calls discussed in this section. It also supports timeout in the SC_PEND, SC_QPEND, SC_FPEND, SC_SPEND, and SC_TDELAY calls.

The VRTX_INIT call sets the VRTX32 clock to zero. The UI_TIMER call, usually issued by a counter-timer ISR, then increments this clock each time it is issued. The 32-bit VRTX32 clock rolls over from $0FFFFFFFF to 0. After VRTX_INIT, only the UI_TIMER and SC_STIME calls modify the VRTX32 clock.

The UI_TIMER call integrates the counter-timer device with VRTX32. You define a short ISR to service the specific device, such as an 8253 Interval Timer or a 9513 Timing Controller. The counter-timer ISR periodically issues the UI_TIMER call to

Table 3-4 VRTX32 Calls for Special Devices

For real-time clock support, VRTX32 recognizes these calls:

From tasks:

SC_GTIME
SC_STIME
SC_TDELAY
UI_TIMER
SC_TSLICE

From interrupt handlers:

SC_GTIME
SC_STIME
UI_TIMER

For character I/O, VRTX32 recognizes these calls:

From tasks:

SC_GETC
SC_PUTC
SC_WAITC

From interrupt handlers:

UI_RXCHR
UI_TXRDY

inform VRTX32 that another time interval, or **VRTX32 clock tick**, has expired. (VRTX32 processes the timer tick at the last UI_EXIT call in a group of nested interrupts.) Even in target environments without a counter-timer device, issuing the UI_TIMER from other ISRs or a low-priority task on a regular basis can fulfill task delay and round-robin scheduling needs.

The SC_GTIME and SC_STIME calls access the VRTX32 clock from task level. The SC_TDELAY call suspends a task's execution for the specified number of VRTX32 clock ticks. (A task can also issue an SC_TDELAY call with a zero value to voluntarily preempt itself.)

The SC_TSLICE call allows equal-priority tasks to take turns executing; each executes for a specified number of VRTX32 clock ticks. Note that higher-priority tasks can preempt tasks that undergo time-slicing. When control returns to the preempted task, it completes its time interval.

3.3.2 Real-Time Clock Calls

Table 3-5 contains a summary of the real-time clock system calls. The SC_TDELAY call always results in a task switch. The UI_TIMER call can initiate the rescheduling procedure; refer to Appendix E, The Rescheduling Procedure, for more information. For detailed information on each of the calls, refer to Chapter 7, System Call Reference.

Table 3-5 Real-Time Clock Call Summary

SC_GTIME	Gets the current value, in VRTX32 clock ticks, of the VRTX32 clock.
SC_STIME	Sets the current value, in VRTX32 clock ticks, of the VRTX32 clock.
SC_TDELAY	Delays execution of the calling task for a number of VRTX32 clock ticks (or preempts the task if a zero value is specified).
SC_TSLICE	Enables round-robin scheduling of equal-priority tasks, specifying the amount of time each task can be in control.
UI_TIMER	Posts time increment from an ISR or a task to VRTX32.

3.3.3 Character I/O Support

The system calls in this section allow a task to perform general character I/O. VRTX32 manages separate 64-character FIFO buffers for reads and writes from an I/O port. The SC_PUTC call places a single character into the output buffer. When the buffer is full, calling tasks suspend until there is room in the buffer. The SC_GETC call retrieves a single character from the input buffer. When buffers are empty, calling tasks suspend until there is a character in the buffer. An additional call, SC_WAITC, suspends a task until a specified character is received. VRTX32 manages only one SC_WAITC request at a time.

Simple, user-supplied ISRs use the UI_TXRDY and UI_RXCHR system calls to pass characters one at a time to and from VRTX32. An ISR uses the UI_RXCHR call to transfer each character to VRTX32's SC_GETC buffer as the character is received from the input device (such as a USART or a parallel I/O device).

When the output device emits a transmit-ready interrupt, the device's ISR is invoked. This ISR uses UI_TXRDY to tell VRTX32 that the device is ready. If there are characters in the SC_PUTC buffer, UI_TXRDY returns the next character to the ISR. The ISR can output the character directly, or it can call the **TXRDY driver routine** to output the character. (The ISR does not have to use the TXRDY driver routine, but it is recommended for well-structured programs.) The TXRDY driver routine transmits the character to the output device and returns.

If the SC_PUTC buffer is empty, UI_TXRDY returns the ER_NCP error code to the ISR. VRTX32 notes the ready status of the output device. The ISR does not call the TXRDY routine; it exits.

When SC_PUTC puts a character to an empty buffer and the output device is ready, VRTX32 calls the TXRDY driver routine directly to transmit the character.

When the TXRDY routine receives control from VRTX32, register D1[7:0] contains the character, register D0 contains the RET_OK ($0000) return code, and interrupts are disabled to Component-Disable-Level.

Although VRTX32 has only single-port I/O calls, you can create your own system calls or use IOX for multiport I/O. Refer to Chapter 5, Support for User-Defined Extensions, for more information on user-defined system calls. Consult the *IOX/68000 User's Guide* and the *IOX/68000 Installation Guide* for information on expanded I/O support.

3.3.4 Character I/O Support Calls

Table 3-6 contains a summary of the character I/O system calls. The SC_WAITC call always results in a task switch. The other calls can initiate the rescheduling procedure; refer to Appendix E, The Rescheduling Procedure, for more information. For detailed information on each of the calls, refer to Chapter 7, System Call Reference.

Table 3-6 Character I/O Call Summary

SC_GETC	Gets the next character from the supported I/O device.
SC_PUTC	Puts a character to the supported I/O device.
SC_WAITC	Waits for a special character from the supported I/O device.
UI_RXCHR	Posts a received character from an ISR to VRTX32.
UI_TXRDY	Posts a transmit ready signal from an ISR to VRTX32, informing VRTX32 that the ISR is ready to transmit another character to the supported I/O device. When a character is present, this call retrieves it for transmission.

Chapter 4

Configuration and Initialization

HUNTER ◆ READY
A Division of Ready Systems

4.1 Introduction

System initialization depends on the overall board environment. It includes all preliminary activities that place the system in a known state before application execution. Examples of system initialization are device initialization, initial task creation, and initialization of general software state variables. In a VRTX32 system, initialization consists of VRTX32 initialization and user-supplied initialization.

VRTX32 is designed to be as independent as possible from individual development systems and target configurations. Therefore, VRTX32 is a single, indivisible PROM product, rather than a collection of individual modules requiring a particular linker and host for its assembly. Because of this, VRTX32 performs only that part of initialization dependent on the M68000 microprocessor and system memory.

The board support package completes the rest of the initialization process. One element of a board support package is the user-supplied VRTX32 Configuration Table, which is VRTX32's window to its surrounding environment. Consult *How to Write a Board Support Package for VRTX* for more information.

This chapter describes the VRTX32 Configuration Table and the initialization process in detail.

4.2 VRTX32 Configuration Table

The **VRTX32 Configuration Table** contains parameters that define a particular system configuration. A vector in the Exception Vector Table (EVT) points to the configuration table. You can specify any relative location, but the default is vector 64 of the EVT at address $100. Refer to Section 4.4.1, User-Supplied Initialization, for information on changing this default.

At system initialization, the parameters in the configuration table provide the following information to VRTX32:

- The location and size of the VRTX32 Workspace
- The size of each task's stacks
- The size of the optional interrupt stack
- The number of control blocks for event flag groups and semaphores
- The size of the idle task's stack
- The number of tasks that can exist in the system at any one time
- The address of the optional TXRDY driver routine
- The addresses of optional user extensions
- The address of the optional Component Vector Table (CVT)

Figure 4-1 shows the configuration table's format.

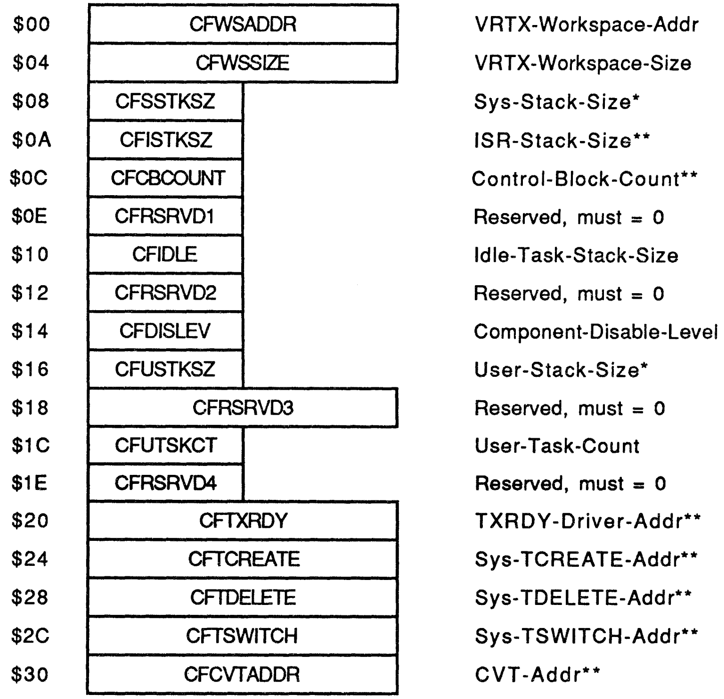
The following describes each parameter of the VRTX32 Configuration Table. The mnemonic following each parameter name is defined in the **vrtxvisi.inc** file. Refer to Section D.3, VRTX32 Definitions File, for a listing of this file.

- **VRTX-Workspace-Address** (CFWSADDR) specifies the beginning address of the VRTX32 Workspace available for VRTX32's data requirements. VRTX32 Workspace includes an area for VRTX32 system variables, a TCB and stacks for each task in the system, the idle task's stack, an optional interrupt stack, and all control structures for queues, partitions, event flag groups, and semaphores.

- **VRTX-Workspace-Size** (CFWSSIZE) specifies the total size of memory available to VRTX32. The VRTX32 Workspace size is specified in bytes. Section 4.3, Determining VRTX32 Workspace Size, presents the formula for determining VRTX32 Workspace size.

- **Sys-Stack-Size** (CFSSTKSZ), when added to **User-Stack-Size**, specifies the total amount of stack space to be given to each VRTX32 task.

  VRTX32 automatically sets up an area of memory for the stack of each task. The sum of Sys-Stack-Size and User-Stack-Size determines the total size of this area in bytes. When a Supervisor mode task is created, it is given a single stack (referenced by SSP) whose total size equals the sum of the two parameters. When a User mode task is created, however, it is given two separate stacks, one of size User-Stack-Size (referenced by USP) and one of size Sys-Stack-Size (referenced by SSP).



$00	CFWSADDR	VRTX-Workspace-Addr
$04	CFWSSIZE	VRTX-Workspace-Size
$08	CFSSTKSZ	Sys-Stack-Size*
$0A	CFISTKSZ	ISR-Stack-Size**
$0C	CFCBCOUNT	Control-Block-Count**
$0E	CFRSRVD1	Reserved, must = 0
$10	CFIDLE	Idle-Task-Stack-Size
$12	CFRSRVD2	Reserved, must = 0
$14	CFDISLEV	Component-Disable-Level
$16	CFUSTKSZ	User-Stack-Size*
$18	CFRSRVD3	Reserved, must = 0
$1C	CFUTSKCT	User-Task-Count
$1E	CFRSRVD4	Reserved, must = 0
$20	CFTXRDY	TXRDY-Driver-Addr**
$24	CFTCREATE	Sys-TCREATE-Addr**
$28	CFTDELETE	Sys-TDELETE-Addr**
$2C	CFTSWITCH	Sys-TSWITCH-Addr**
$30	CFCVTADDR	CVT-Addr**

* If stacks are explicitly allocated with the TCREATE routine, set these parameters to zero.

** Indicates optional parameter; if omitted or unused, set to zero.

Figure 4-1 VRTX32/68000 Configuration Table

The Sys-Stack-Size parameter specifies the maximum amount of stack space required by a VRTX32 task, whether User mode or Supervisor mode, when it invokes system services. The stack size depends on the stack needs of a task's system call, user-supplied extension, and user-supplied system call activities. The stack must also be large enough to accommodate all ISR activity, if interrupt stack switching is not enabled. Also, tasks that call other components require additional stack space. (Consult the respective component's user's guide for information on component stack requirements.) When determining Sys-Stack-Size, you must allow 100 bytes in each stack for VRTX32 requirements. For more information about each task's stack, refer to the User-Stack-Size description and Section 4.3.1, Determining Task Stack Requirements.

- **ISR-Stack-Size** (CFISTKSZ) specifies the size of the optional interrupt stack in bytes, or zero if this stack is not used. If ISRs use the interrupted task's stack (interrupt stack switching is not enabled), you must supply a zero value. If all ISRs use a single dedicated stack separate from task stacks (interrupt stack switching is enabled), you supply a nonzero value specifying this stack size. During VRTX_INIT, VRTX32 allocates this stack in VRTX32 Workspace. For more information on interrupt stack support, refer to Section 3.2.6, Interrupt Stack Switching, and Section 4.3.2, Determining Interrupt Stack Requirements.

- **Control-Block-Count** (CFCBCOUNT) is an optional parameter that specifies the maximum number of event flag groups and semaphores that can exist in the system at one time. Each event flag group or semaphore is associated with one control block. If you do not use event flag groups or semaphores in your application, supply a zero value.

- **Idle-Task-Stack-Size** (CFIDLE) specifies the size of the idle task's stack, in bytes. If interrupt stack switching is enabled, the default idle task stack size of 128 bytes is sufficient; supply a value of zero. If interrupt stack switching is *not* enabled, 128 bytes may not be enough to cover your system's interrupt stack switching needs; supply a nonzero value specifying the stack size. (Refer to Section 4.3.2, Determining Interrupt Stack Requirements.) During VRTX_INIT, VRTX32 allocates the idle task's stack in VRTX32 Workspace. For more information on interrupt stack support, refer to Section 3.2.6, Interrupt Stack Switching.

- **Component-Disable-Level** (CFDISLEV) is a word value ranging from 0 to 7, inclusive. This value specifies the interrupt level that is loaded into the SR when VRTX32 disables interrupts. The Component-Disable-Level is determined by the interrupt level of the highest-priority ISR that makes VRTX32/OS system calls, *plus one*. A value of zero for this parameter indicates the default interrupt level of 7, the highest level.

  You can assign interrupt levels greater than Component-Disable-Level to critical devices as long as their associated ISRs do not make VRTX32 or component system calls. Any ISR with an interrupt level greater than or equal to Component-Disable-Level (or with interrupt level 7) that makes VRTX32 or component system calls causes indeterminate behavior. For more information about this parameter, refer to Section 3.2.7, M68000 Interrupt Levels and VRTX32.

- **User-Stack-Size** (CFUSTKSZ), when added with **Sys-Stack-Size**, specifies the total amount of stack space to be given to each VRTX32 task.

  VRTX32 automatically sets up an area of memory for the stack of each task. The sum of User-Stack-Size and Sys-Stack-Size determines the total size of this area in bytes. When a Supervisor mode task is created, it is given a single stack (referenced by SSP) whose total size equals the sum of the two parameters. When a User mode task is created, however, it is given two separate stacks, one of size User-Stack-Size (referenced by USP) and one of size Sys-Stack-Size (referenced by SSP).

The User-Stack-Size parameter specifies the size, in bytes, of the area that handles a task's subroutine calls and that is used for temporary variable storage. For User mode tasks, this area is the stack pointed to by USP. For Supervisor mode tasks, this area is a portion of the total stack pointed to by SSP.

You may also bypass VRTX32 and explicitly allocate stacks with the TCREATE routine. Refer to Section 5.3.1, TCREATE Routine, for information about this user-supplied extension. In this case, supply a value of zero in both the User-Stack-Size and Sys-Stack-Size parameters.

- **User-Task-Count** (CFUTSKCT) specifies the maximum number of tasks that can be simultaneously active in the system. VRTX32 uses this value to allocate TCBs and stack space.

- **TXRDY-Driver-Address** (CFTXRDY) is an optional parameter that specifies the address of the TXRDY driver routine called by VRTX32. VRTX32 calls this routine when the output device is ready and SC_PUTC places a character in the output buffer. Refer to Section 7.46, UI_TXRDY-Transmit-Ready Interrupt. If your application does not use VRTX32 I/O, you do not have to provide the TXRDY driver routine. In this case, supply a null pointer (zero).

- **Sys-TCREATE-Address** (CFTCREATE) and **Sys-TDELETE-Address** (CFTDELETE) are optional parameters that specify the address of user-defined routines that perform special processing when tasks are created or deleted. If you do not use these routines, supply a null pointer (zero). For additional information, refer to Sections 5.3.1, TCREATE Routine, and 5.3.2, TDELETE Routine.

- **Sys-TSWITCH-Address** (CFTSWITCH) is an optional parameter that specifies the address of the user-defined routine that is given control when a task switch occurs. When no special handling is required when making a context switch, supply a null pointer (zero). For more information, refer to Section 5.3.3, TSWITCH Routine.

- **CVT-Address** (CFCVTADDR) is an optional parameter that specifies the address of a Component Vector Table (CVT). The CVT routes execution control to components other than VRTX32. If you are not using any other components, supply a null pointer (zero). For more information, refer to Section 6.3, Component Vectoring.

- **Reserved** (CFRSRVD) represents a parameter reserved for future versions of VRTX32. You must always supply a value of zero.

4.3 Determining VRTX32 Workspace Size

The configuration table parameter **VRTX-Workspace-Size** specifies the total size of memory available to VRTX32. The required workspace size is determined by stack requirements, control structure requirements, and internal variable requirements.

This section provides guidelines for determining task and interrupt stack requirements. It also presents the VRTX32 Workspace size formula that combines the control structure and internal variable requirements with the stack requirements.

4.3.1 Determining Task Stack Requirements

The task stack requirements of a system affect the **User-Stack-Size** and **Sys-Stack-Size** parameters of the VRTX32 Configuration Table. (They also affect the sizes of stacks allocated with the TCREATE routine.) These parameters in turn affect the VRTX-Workspace-Size parameter.

If you allocate task stacks explicitly with the TCREATE routine, set the User-Stack-Size and Sys-Stack-Size parameters to zero. However, if VRTX32 allocates task stacks, User-Stack-Size and Sys-Stack-Size must contain specific values.

Two factors determine User-Stack-Size: the number of subroutine calls made, and the amount of variable storage needed by each task.

To determine Sys-Stack-Size, you must examine the following factors: VRTX32 stack requirements, the existence of user-defined VRTX32 extensions, the existence of user-defined system call handlers, the amount of space needed for ISR activity if interrupt stack switching is not enabled, and the existence of other components. The formula shown in Table 4-1 combines these factors.

4.3.2 Determining Interrupt Stack Requirements

The interrupt stack requirements of a system affect the **ISR-Stack-Size** parameter of the VRTX32 Configuration Table if interrupt stack switching is enabled. If interrupt stack switching is *not* enabled, the interrupt stack requirements affect the **Idle-Task-Stack-Size** parameter and the **Sys-Stack-Size** parameter formula. These parameters in turn affect the VRTX-Workspace-Size parameter.

Table 4-1 Determining Task Stack Requirements

Formula

Sys-Stack-Size = 100 + UX + USC + ISW + ISR + C

Symbols

--	100 bytes needed for VRTX32 stack requirements
UX	User-supplied VRTX32 extensions requirements
USC	User-defined system call handler requirements
ISW	Interrupt stack switch requirements before switch: if interrupt stack switching enabled, ISW = 32; else ISW = 0.
ISR	Total interrupt stack requirements if interrupt stack switching not enabled; if interrupt stack switching enabled = 0 (refer to Section 4.3.2)
C	Total component call stack requirements, specified in bytes (consult component's user's guide)

The M68000 architecture supports different levels of interrupts and nesting of interrupts across levels. When all ISRs enable interrupts to allow nesting, then all six levels must be taken into consideration when determining stack requirements.

This is the formula for an individual interrupt level:

n = interrupt level 1 through 6
i_n = (number of ISR invocations responding to level n) *
(maximum interrupt stack requirements at level n)

The worst case is when each ISR enables interrupts. For example, a system has the following ISR activity:

Two ISRs responding to interrupt level 2 with maximum stack requirements = 32 bytes
One ISR responding to interrupt level 4 with maximum stack requirements = 64 bytes

Three ISRs responding to interrupt level 5 with maximum stack requirements = 100 bytes

This is the formula to determine the example system's interrupt stack requirements:

$ISR = i_1 + i_2 + \dots i_6$

$i_1 = 0$
$i_2 = (2 * 32) \quad = 64$
$i_3 = 0$
$i_4 = (1 * 64) \quad = 64$
$i_5 = (3 * 100) \quad = 300$
$i_6 = 0$

$ISR = \quad 0 + 64 + 0 + 64 + 300 + 0$
ISR = 428 bytes (decimal)

4.3.3 Determining Control Structure Requirements

VRTX32 uses control structures and internal variables during multitasking management. Table 4-2 combines the control structure and internal variable requirements with the stack requirements to determine the total amount of workspace required by VRTX32 to support an application. Numbers are shown in decimal.

For example, a system has the following user-supplied configuration:

10 tasks
1 partition of size 2048 bytes with 64-byte blocks
= 32 blocks
1 partition of size 4096 bytes with 512-byte blocks
= 8 blocks
1 extension to this partition, also of size 4096 bytes
= 8 blocks
8 queues each of size 10
= 80 queue elements
2 queues each of size 20
= 40 queue elements
1 event flag group
1 semaphore

no user-defined extensions
no user-supplied system calls
no interrupt stack (interrupt stack switching not enabled)
VRTX32 is the only component in the system

Table 4-2 Determining VRTX32 Workspace Size

Formula

VRTX-Workspace-Size = 2624 + ((160+s+us)*t) + 64p + 2b + 32e + 36q + 4qe + 36cb + idle + istk

Symbols

--	VRTX32 system variables need 2624 bytes.
s	Sys-Stack-Size. See Table 4-1.
us	User-Stack-Size, determined by subroutines and variable storage.
t	Maximum number of tasks. VRTX32 allocates 160 bytes for the TCB frame and other overhead for each task.
p	Maximum number of memory partitions. Each partition needs a 64-byte control block.
b	Maximum number of memory blocks. Determine this for each partition (or extension) by dividing the partition's size by the partition's block size. Each block needs 2 bytes.
e	Maximum number of partition extensions, determined by the number of SC_PEXTEND calls. Each extension needs 32 bytes.
q	Maximum number of queues in system. Each queue needs a 36-byte control block.
qe	Maximum number of queue elements. Each queue element needs 4 bytes.
cb	Maximum number of control blocks for event flag groups and semaphores. Each control block needs 36 bytes.
idle	Idle-Task-Stack-Size. If interrupt-stack-switching *not* enabled, determine idle with the formula in Section 4.3.2. If enabled, idle = 128.
istk	ISR-Stack-Size. If interrupt stack switching *not* enabled, istk = 0. If enabled, determine istk with the formula in Section 4.3.2.

The first step is to determine the stack requirements **us**, **s**, **idle**, and **istk**. Assuming no task in the system requires more than 128 bytes for its own subroutine calling and local variables, **us** = 128.

The following is the calculation for **s** (see Table 4-1, Determining Task Stack Requirements). Because interrupt stack switching is *not* enabled, the **ISW** term is zero and the **ISR** term is nonzero. Using the interrupt requirements calculated in Section 4.3.2, Determining Interrupt Stack Requirements, ISR = 428.

s = 100 + UX + USC + ISW + ISR + C
= 100 + 0 + 0 + 0 + 428 + 0
= 528 bytes (decimal)

Because interrupt stack switching is *not* enabled, **idle** is calculated using the formula in Section 4.3.2, and **istk** = 0. Here is the calculation for the example system's VRTX-Workspace-Size parameter:

=2624 + ((160 + s + us) * t) + 64p + 2b + 32e + 36q + 4qe +
36cb + idle + istk
=2624 + ((160 + 528 + 128) * 10) + 64*2 + 2*(32 + 8 + 8) + 32*1 +
36*10 + 4*(80 + 40) + 36*2 + 428 + 0
=2624 + 8160 + 128 + 96 + 32 + 360 + 480 + 72 + 428 + 0
=12380 bytes (decimal)

4.4 Support for System Initialization

System initialization includes user-supplied initialization and VRTX32 initialization. User-supplied code performs user initialization and initializes VRTX32 pointers. This code also issues the VRTX_INIT call to initialize VRTX32 and the VRTX_GO call to start multitasking.

4.4.1 User-Supplied Initialization

User-supplied initialization consists of initialization specific to the hardware devices and the application program. Initialization usually begins when the system is reset. In a VRTX32 system, you must initialize the VRTX32 pointers before VRTX32 initialization begins. All user initialization code must execute in Supervisor mode. Interrupt activity should not occur before VRTX_INIT.

System Reset. The M68000 architecture provides a special signal for system reset, and the CPU performs a specified set of actions when this signal is asserted.

On any M68000 system, the first eight bytes of the EVT (in other words, the first two entries in the EVT) must have the organization shown in Figure 4-2, EVT Reset Format.

EVT Byte Offset	
$00	Reset SSP
$04	Reset PC

Figure 4-2 EVT Reset Format

In Figure 4-2, **Reset PC** is a pointer to a reset routine that is executed when the system reset signal is sent. **Reset SSP** is the value of the Supervisor Stack Pointer that is appropriate for the reset routine.

When the system reset line is activated, the M68000 hardware automatically stops current execution, loads the new value of SSP from offset $00 and the new value of PC from offset $04, and begins execution at the new PC location.

The pointer at offset $04 addresses user-supplied code that performs user initialization and initializes VRTX32 pointers. This initialization code must set up a small (100-byte) stack before issuing the VRTX_INIT call. The VRTX_INIT call initializes VRTX32, and the VRTX_GO call starts multitasking.

If the reset hardware is not used to start up the VRTX32 initialization code, SSP must be set to provide the initial stack for VRTX32's use during the VRTX_INIT call.

Device Initialization. Device initialization depends on the board-level environment. Usually the counter-timer device, character I/O device, and any special user devices (for example, a memory management unit) are initialized at system initialization time.

User-supplied device initialization can occur in code that precedes the VRTX_INIT call, or in code after the VRTX_INIT call and before the VRTX_GO call. You should initialize some devices before VRTX32, because their successful operation is a prerequisite for VRTX32 initialization. For example, when a memory management unit (MMU) exists in the system, initialize it before VRTX32, because system memory allocation depends on the MMU. You can initialize devices without system-wide ramification after VRTX32 initialization.

Application Initialization. Initialization specific to the application program usually consists of setting up software control structures and variables related to the application code, such as mailboxes, queues, and boolean variables. Also, the application initialization code usually creates the task or tasks that the system starts executing after VRTX_GO.

You must initialize constructs that require VRTX32 services, such as queues and memory partitions, after VRTX_INIT.

User Initialization of VRTX32 Pointers. To connect VRTX32 to the user-supplied VRTX32 Configuration Table and to the application, you must initialize two pointers. It is important that you initialize them before the application issues the VRTX_INIT call. These pointers are the **configuration table pointer** and the **VRTX32 entry pointer**.

The initialization of the configuration table pointer connects the configuration table to VRTX32. VRTX32 assumes this pointer resides at vector 64, offset $100 from the base of the EVT. However, you can choose any location.

If you use a location other than the default, you must change VRTX32's internal pointer value. Offset $22 from the base of VRTX32 contains this value's default, the 32-bit address $100. If you define a new vector, you must load it with the base of the configuration table.

The base of VRTX32 is known as the **VRTX32 entry point**. You must initialize the TRAP vector used by the application to make VRTX32 calls with the VRTX32 entry point. This manual uses TRAP #0 as an example. However, you can use any of the 16 TRAP vectors.

4.4.2 VRTX32 Initialization

Two VRTX32 system calls initialize VRTX32 and start application processing. All user initialization code, including the code that issues these calls, must execute in Supervisor mode.

VRTX_INIT	Initialize VRTX32
VRTX_GO	Start Application Execution

The VRTX_INIT system call performs these functions:

1. Delimits and zeros the VRTX32 Workspace.
2. Sets up and reserves TCBs for SC_TCREATE calls.
3. Sets up the user-specified stacks for each task (optional).
4. Sets up the interrupt stack (optional).
5. Initializes other internal VRTX32 variables.
6. Returns control to the caller.

The return code indicates whether the VRTX_INIT operation encountered any errors.

The VRTX_INIT call requires the use of a temporary stack. Set up a small (100-byte) stack before calling VRTX_INIT.

VRTX_GO is called after VRTX_INIT. This call begins executing the highest-priority task created in the user initialization code and does not return to initialization code. Multitasking is now underway.

4.4.3 Use of System Calls During Initialization

Use of VRTX32 system calls (by user initialization code or by ISRs) before VRTX_INIT causes unpredictable results. User initialization code can use the system calls in Table 4-3 after the VRTX_INIT call and before the VRTX_GO call.

You must prevent interrupt activity before VRTX_INIT; we recommend that interrupts remain disabled until VRTX_GO. However, ISRs can use many of the calls in Table 4-3 during initialization (see Table 3-2, VRTX32 Calls Not Allowed from ISRs, for restrictions). When an ISR makes these calls, UI_ENTER and UI_EXIT must bracket the ISR. In addition, if interrupt stack switching is enabled, the ISR must begin with a UI_ENTER call.

User initialization code typically uses the SC_TCREATE, SC_FCREATE, SC_SCREATE, SC_PCREATE, and SC_QCREATE calls.

Use the SC_PUTC call with caution during initialization. The call can be useful for displaying a sign-on message and the starting prompt character on system consoles. However, an attempt to put more than 64 characters in the buffer before VRTX_GO causes unpredictable results.

Table 4-3 Calls Permitted between VRTX_INIT and VRTX_GO

Task Management

SC_TCREATE
SC_TDELETE
SC_TSUSPEND
SC_TRESUME
SC_TPRIORITY
SC_TINQUIRY

Memory Mangement

SC_GBLOCK
SC_RBLOCK
SC_PCREATE
SC_PEXTEND

Communication and Synchronization

SC_POST
SC_ACCEPT
SC_QPOST
SC_QJAM
SC_QACCEPT
SC_QCREATE
SC_QECREATE
SC_QINQUIRY
SC_FCREATE
SC_FDELETE
SC_FPOST
SC_FCLEAR
SC_FINQUIRY
SC_SCREATE
SC_SDELETE
SC_SPOST
SC_SINQUIRY

Real-Time Clock

SC_GTIME
SC_STIME
SC_TSLICE
UI_TIMER

Initialization of other Hunter & Ready components that use VRTX32 component routing should occur only after VRTX_INIT.

4.4.4 Initialization Calls

Table 4-4 contains a summary of the initialization system calls. For detailed information on each of the calls, refer to Chapter 7, System Call Reference.

Table 4-4 Initialization Call Summary

VRTX_INIT	Initializes VRTX32.
VRTX_GO	Starts application execution after VRTX_INIT. No return is made to the caller.

Chapter 5

Support for User-Defined Extensions

HUNTER
READY
A Division of Ready Systems

5.1 Introduction

VRTX32 supplies much of the system software requirements for embedded microprocessor applications. However, as a component designed for use in different applications with different hardware configurations, VRTX32 is generalized and does not contain all the functions that might be found in an operating system tailored to a specific microcomputer board or to a specific application.

Hunter & Ready has designed VRTX32 and its other silicon software components as elements for building a larger computer system. This design is similar to that of stereo components that you connect with one another to build a complete audio system. Like the audio plugs and cables, linkage mechanisms in the silicon software components permit you to interface the components with other software and hardware.

There are four types of interfaces defined between VRTX32 and user-supplied code.

- The interface between VRTX32 and user-supplied application code (tasks) is defined by the basic VRTX32 system calls (refer to Chapter 2, Basic System Calls). This interface is also defined by entries in the configuration table (refer to Chapter 4, Configuration and Initialization).

- The interface between VRTX32 and user-supplied ISRs is defined in Chapter 3, Interrupt Support.

- The interface between VRTX32 and any additional system call handlers you supply is defined in this chapter. These handlers are designed to support special hardware devices or functions.

- The interface between VRTX32 and user-supplied VRTX32 extensions is defined in this chapter. A VRTX32 extension is code that adds additional functions to the basic VRTX32 mechanisms at specific times during VRTX32 processing.

Most packaged operating systems fail to provide an adequate definition of the last two interfaces; they do not provide enough hooks to link the operating system to the application code. Unless the application is exactly what the operating system's designers envisioned, the source code of the operating system must be modified. This can be an expensive and risky job.

This chapter discusses the interface between VRTX32 and user-defined system call handlers, and the interface between VRTX32 and user-supplied VRTX32 extensions. Figure 5-1, Extensions Architecture, shows the functions involved.

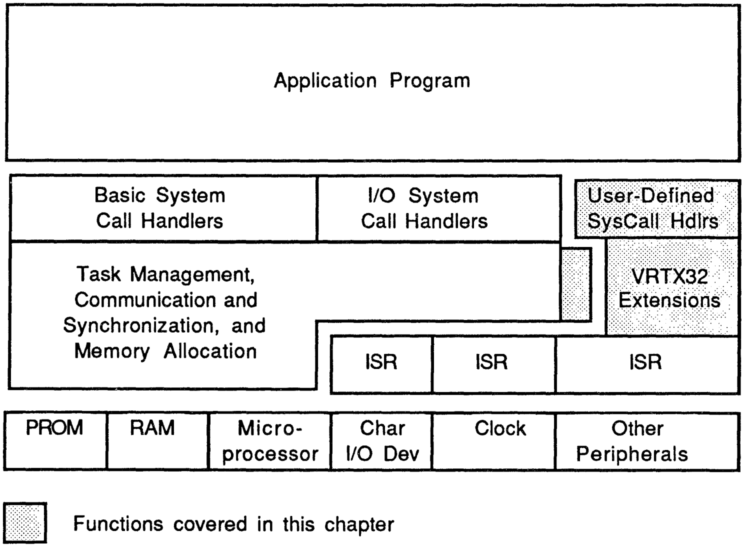

Application Program
Basic System Call Handlers
I/O System Call Handlers
User-Defined SysCall Hdlrs
Task Management, Communication and Synchronization, and Memory Allocation
VRTX32 Extensions
ISR
ISR
ISR
PROM
RAM
Micro-processor
Char I/O Dev
Clock
Other Peripherals
Functions covered in this chapter

Figure 5-1 Extensions Architecture

5.2 User-Defined System Call Handlers

User-defined system call handlers perform system services, just as VRTX32 system calls do. Both are invoked in the same way, using a software-generated interrupt (TRAP) instruction. The difference is that user-defined system call handlers perform functions specific to the application.

In fact, user-defined system call handlers are similar to subroutines, as the following section shows. However, a call handler has the advantage of being independent from all linkers or loaders because it is accessed through the software-generated interrupt instruction.

To implement a user-defined call, reserve a software interrupt vector not already used by VRTX32 or by interrupt or error handlers. The **TRAP instruction** specifies the vector used. The entry point of the user-supplied system call is loaded into the vector table in the same way that the VRTX32 entry point is loaded into the vector table. Refer to Section 2.1.1, Accessing VRTX32, for information about the VRTX32 entry point.

5.2.1 Writing a System Call Handler

In effect, a user-defined call handler routine is not much different from an ordinary subroutine; as far as VRTX32 is concerned, the rules for coding both types of routines are nearly identical. Subroutines and call handlers have these similarities:

- Parameters for both types of routines can be communicated to and from the caller according to any convention. For example, parameters can be passed in registers or on a stack.
- Both types of routines must ordinarily save and restore registers.
- Both call handlers and ordinary subroutines, like any user-supplied code, can use the SC_LOCK and SC_UNLOCK calls to prevent VRTX32 rescheduling. Code that is bracketed in this way is not preempted by a task.
- Both call handlers and subroutines are sometimes subject to restrictions imposed on them by their callers. Thus, if ISRs as well as tasks might call a routine, the routine must limit itself to the VRTX32 calls permissible from ISR level; refer to Section 3.2.4, VRTX32 Calls Allowed from ISRs. Conversely, a routine called only from task level can use the full range of VRTX32 services.

However, there are a few noteworthy differences between system call handlers and ordinary subroutines. A subroutine ends with a return-from-subroutine instruction (RTS). But a system call handler, because it is entered by a program-generated exception, ends with the return-from-exception instruction (RTE). The TRAP instruction that invokes a system call handler causes an **exception frame** of additional information (the SR and PC registers, as well as the format/ID word for the MC68010 microprocessor) to be placed on the stack. Thus, the call handler must conclude with the RTE instruction to remove this frame and return correctly to the caller.

Because system call handlers are accessed by a TRAP instruction, system interdependencies are reduced. Relocating a call handler to a new address requires only that a single location in the vector table be changed. Relocating a subroutine, however, can mean that all modules that make use of the routine must be relinked.

The most significant difference between call handlers and ordinary subroutines concerns their interaction with the M68000 execution modes. Because it gains control with a TRAP instruction, a system call handler always executes in Supervisor mode. A subroutine, on the other hand, inherits its caller's mode. Thus, when a User mode task invokes a system call handler, that handler routine can perform services for the caller (such as disabling interrupts or executing privileged I/O instructions) that the caller could not ordinarily perform.

Changing from a User mode caller to a Supervisor mode handler routine can pose special problems, though, if parameters are passed on stacks. In this case, the system call handler must use the **MOVE USP** to access parameters that are on a User mode stack. If a call handler is invoked by both User mode tasks and by Supervisor mode code (ISRs and/or Supervisor mode tasks), it can locate its stacked parameters only by first checking the saved SR word in the exception frame to determine the caller's mode.

5.2.2 An Example System Call Handler

As an example of a user-defined system call handler, consider a routine that extends the functions of SC_FPOST and SC_FCLEAR. The **Broadcast (BRDCST)** call posts the time of day to all tasks currently waiting at the specified event flag group, then clears the flags. This means that none of the readied tasks have to issue the SC_FCLEAR call.

You must initialize an unused interrupt vector with the address of the BRDCST routine. When TRAP #1 is chosen, the initialization code looks like Example 5-1. For MC68010, this example assumes that the VBR is set to zero. The BRDCST initialization code would be included in the board support initialization code and executed before the VRTX_GO call.

Example 5-1 BRDCST Initialization Code

```
MOVE.L   #BRDCST,$84    * point trap #1 to BRDCST code
```

Before a task calls BRDCST, an event flag group must be created with SC_FCREATE. The event flag group ID number is passed to BRDCST in variable EFID, with register D1. The SC_GTIME call is issued to get the time of day, which is passed to BRDCST in register D2. The task then makes the BRDCST call using the TRAP instruction, as Example 5-2 shows. When the call completes, the return code is in register D0.

Example 5-2 Making the BRDCST Call

```
SCFGTIME     EQU  $0A       * SC_GTIME function code
VRTX         EQU  $00       * VRTX32 trap number
BROADCST     EQU  $01       * BRDCST trap number

   XDEF EFID
EFID:
   DS.L    1                * event flag group ID number

BRDCST_TIME:
   MOVE.L  #SCFGTIME,D0     * D0 = function code
   TRAP    #VRTX            * VRTX32 SC_GTIME call
   MOVE.L  D1,D2            * move time from D1 to D2
   BEQ.S   BRDCST_TIME      * don't post a zero time
   MOVE.L  EFID,D1          * event flag group ID number
   TRAP    #BROADCST        * call BRDCST
   TST.L   D0               * error during BRDCST?
   BNE.S   ERROR            * if so, branch to error handler
   RTS
```

Example 5-3 contains the code for the BRDCST call itself. The BRDCST call sends the time of day, with SC_FPOST, to all tasks pended on the event flag group. After the message is sent, BRDCST uses the SC_FCLEAR call to clear the event flags it set with the SC_FPOST call.

Note that the body of the BRDCST routine is bracketed by the SC_LOCK and SC_UNLOCK calls. This ensures that event flags are cleared before BRDCST is called again on the same event flag group.

Example 5-3 The BRDCST Call in Action

```
SCFLOCK     EQU $20        * SC_LOCK function code
SCFUNLOCK   EQU $21        * SC_UNLOCK function code
SCFFPOST    EQU $1A        * SC_FPOST function code
SCFFCLEAR   EQU $1B        * SC_FCLEAR function code
VRTX        EQU $00        * VRTX32 trap number

BRDCST:
   CLR.L    -(SP)          * reset our return code
   MOVE.L   #SCFLOCK,D0    * D0 = function code
   TRAP     #VRTX          * VRTX32 SC_LOCK call
   MOVE.L   D0,(SP)        * error during lock?
   BNE.S    BEXIT          * if so, exit

   MOVE.L   #SCFFPOST,D0   * D0 = function code
   TRAP     #VRTX          * VRTX32 SC_FPOST call
   MOVE.L   D0,(SP)        * error during post?
```

```
      BNE.S    BRDX           * if so, exit
      MOVE.L   #SCFCLEAR,D0   * D0 = function code; D1 and D2
*                               have same values as they did
*                               for SC_FPOST call
      TRAP     #VRTX          * VRTX32 SC_FCLEAR call
      MOVE.L   D0,(SP)        * save return code

BRDX:
      MOVE.L   #SCFUNLOCK,D0  * D0 = function code
      TRAP     #VRTX          * VRTX32 SC_UNLOCK call
      TST.L    D0             * error during unlock?
      BEQ.S    BEXIT          * if no, don't change return code
      MOVE.L   D0,(SP)        * return unlock error

BEXIT:
      MOVE.L   (SP)+,D0       * caller's D0 = return code
      RTE                     * return from call handler
```

To pend for a BRDCST message, tasks issue SC_FPEND on the same event flag group. These tasks should pend for all event flags with an OR pend. See Example 5-4.

Example 5-4 Pending for BRDCST

```
SCFFPEND   EQU $19          * SC_FPEND function code
VRTX       EQU $00          * VRTX32 trap number

FPEND:
      MOVE.L   EFID,D1        * event flag group ID
      MOVE.L   #0,D2          * no timeout
      MOVE.L   #$FFFFFFFF,D3  * pend on all bits in group
      MOVE.L   #0,D4          * use OR option
      MOVE.L   #SCFFPEND,D0   * SC_FPEND function code
      TRAP     #VRTX          * call VRTX32
      TST.L    D0             * error during pend?
      BNE.S    ERROR          * if so, branch to error handler
```

5.3 VRTX32 Extensions

VRTX32 extensions are user-supplied routines that extend the basic VRTX32 mechanisms. These extensions are activated by VRTX32 system events (task creation, task deletion, task switching), rather than by TRAP instructions.

For example, when the system contains a memory management unit (MMU), user-defined system software must handle the MMU registers. As each task is created, the system must create values for the map registers. When a task switch occurs, the system must also switch the contents of the MMU registers.

For MMU support, create an extension to the VRTX32-managed TCB to store the values of each task's MMU register values. The application code sets the TCB extension pointer (TBEXT), which points to additional storage space. The VRTX32 extension routines use this user-supplied TCB extension at every task switch to update the MMU registers.

Three optional parameters in the VRTX32 Configuration Table provide hooks that interface such general system-level software to VRTX32. The Sys-TCREATE-Address and Sys-TDELETE-Address parameters point to the addresses of the routines that perform special processing when tasks are created or deleted. Sys-TSWITCH-Address points to a user-supplied routine that is called when a task switch is made.

When these three parameters are not null in the configuration table, VRTX32 gives these user-supplied routines control when the appropriate events occur. At that time, information regarding the state of the current task is passed to these routines.

The TCREATE, TDELETE, and TSWITCH routines are called in Supervisor mode and use the current Supervisor mode stack. These routines must save and restore any registers used during their execution and must end with the RTS instruction. In addition, the same VRTX32 calls that are allowed from ISRs are allowed from these routines. Refer to Section 3.2.4, VRTX32 Calls Allowed from ISRs, for details.

5.3.1 TCREATE Routine

The optional VRTX32 Configuration Table parameter **Sys-TCREATE-Address** points to a user-supplied routine that is given control when VRTX32 creates a new task.

When the **TCREATE routine** receives control, register A1 contains a pointer to the TCB of the newly created task. Register A2 contains a pointer to the TCB of the calling task. This pointer is not valid before the execution of the VRTX_GO call; nor is it valid for tasks created during system initialization.

The TCREATE routine is not called when the idle task is created. This means that if you use the TCREATE routine to allocate an additional data area to TCBs, or to alter TCBs in any way, the idle task's TCB is not affected.

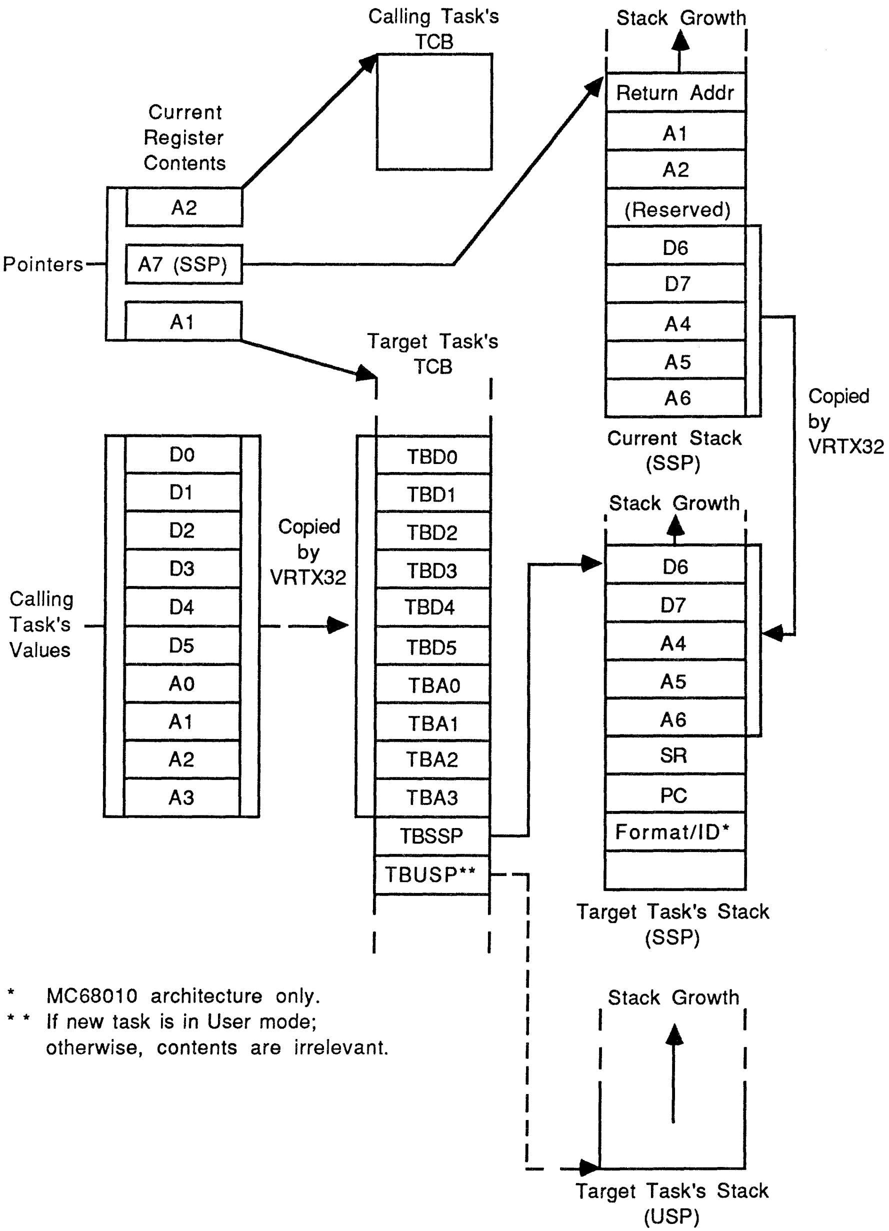
The register values in the newly created task's TCB are derived from the values of the creating task's registers at the moment the SC_TCREATE call is executed. The TCB of a task does not, however, contain all the task's state information. For performance reasons, VRTX32 stores some register values on the task's stack instead of its TCB.

For the newly created task's stack, these values are copied from the current stack just before entry to the TCREATE routine.

Figure 5-2, Environment on Entry to TCREATE Routine, shows the layout of the newly created task's TCB and stacks on entry to the TCREATE routine. Note that when the SC_TCREATE call executes, VRTX32 copies the contents of registers D0 through D5 and A0 through A3 directly into the appropriate locations of the new task's TCB. Register A1 contains a pointer to this TCB. VRTX32 copies registers D6, D7 and A4 through A6 from the calling task's stack onto the new task's stack, and sets up the SR and PC registers on this stack from the SC_TCREATE call's values. VRTX32 stores the new task's USP and SSP registers in the TCB; these registers are set up to point as shown in the figure. Notice that only User mode tasks are created with a starting value for both SSP and USP; Supervisor mode tasks set up only SSP.

Example 5-5 uses a TCREATE hook to set up variable-size stacks, and Figure 5-3 shows the process. Using this hook, you can replace VRTX32's fixed-size allocations and allocate stacks of sizes dependent on each task's requirements (refer to Section 4.3.1, Determining Task Stack Requirements).

To use the code in Example 5-5, the configuration table must have the address of the TCREATE routine, and the User-Stack-Size and Sys-Stack-Size parameters must be zero. This signals VRTX32 to bypass allocation of task stacks in the VRTX32 Workspace. VRTX32 then builds the initial stack frame in a temporary storage area in the VRTX32 Workspace. The TCREATE routine must copy this stack frame to the newly allocated stack. This is shown in Figure 5-3. Note that the example is for the MC68000 only. An implementation for the MC68010 processor would have to save the format word, as shown in the commented code.



* MC68010 architecture only.
** If new task is in User mode; otherwise, contents are irrelevant.

Figure 5-2 Environment on Entry to TCREATE Routine

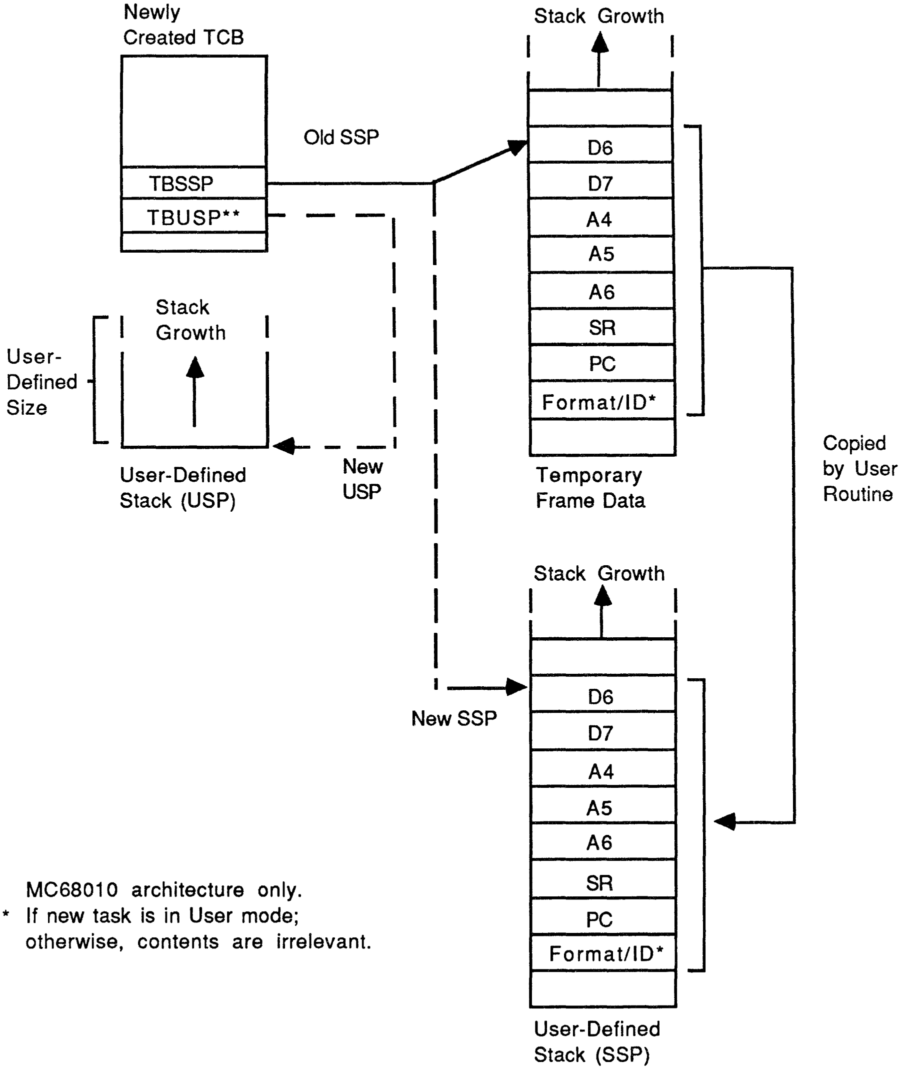


Figure 5-3 User-Defined Stacks

Example 5-5 Implementing Variable-Size Stacks with TCREATE

```
* Offsets into new task's TCB --
TBSSP       EQU  $38          * SSP for task
TBUSP       EQU  $3C          * USP for task

* Offsets into initial VRTX32 stack frame --
STKFI       EQU  $1A          * fmt/ID word-68010 only
STKPC       EQU  $16          * PC value for new task
STKSR       EQU  $14          * SR value for new task
STKA6       EQU  $10          * A6 value for new task
STKA5       EQU  $0C          * A5 value for new task
STKA4       EQU  $08          * A4 value for new task
STKD7       EQU  $04          * D7 value for new task
STKD6       EQU  $00          * D6 value for new task

TCR_HOOK:
   MOVEM.L A3-A4/D0-D1,-(SP) * save all registers used
   MOVEQ.L #0,D0
   MOVEA.L STKPOS,A4          * get current alloc position
   MOVE.W  STKNDX,D1          * get index into size tables
   ADDQ.W  #2,STKNDX          * update index for next time
   MOVE.W  USIZE(PC,D1.W),D0 * get USP size for this task
   ADDA.L  D0,A4              * allocate stack space
   MOVE.L  A4,TBUSP(A1)       * set USP stack ptr for task
   MOVE.W  SSIZE(PC,D1.W),D0 * get SSP size for this task
   ADDA.L  D0,A4              * allocate stack space
   MOVE.L  A4,STKPOS          * save position for next alloc

   MOVE.L  TBSSP(A1),A3       * point to old VRTX32 stack

* Copy values to new SSP stack.
** MOVE.W  STKFI(A3),-(A4)    * fmt/ID word-68010 only
   MOVE.L  STKPC(A3),-(A4)    * PC from old to new stack
   MOVE.W  STKSR(A3),-(A4)    * SR from old to new stack
   MOVE.L  STKA6(A3),-(A4)    * A6 from old to new stack
   MOVE.L  STKA5(A3),-(A4)    * A5 from old to new stack
   MOVE.L  STKA4(A3),-(A4)    * A4 from old to new stack
   MOVE.L  STKD7(A3),-(A4)    * D7 from old to new stack
   MOVE.L  STKD6(A3),-(A4)    * D6 from old to new stack
   MOVE.L  A4,TBSSP(A1)       * set SSP stack ptr for task

   MOVEM.L (SP)+,A3-A4/D0-D1 * restore registers
   RTS                        * return to VRTX32

USIZE:
   DC.W    $100,$0E0,$0C0,$0A0,$080,$100,$0E0,$0C0,$0A0,$080
   DC.W    $100,$0E0,$0C0,$0A0,$080

SSIZE:
   DC.W    $300,$2E0,$2C0,$2A0,$280,$300,$2E0,$2C0,$2A0,$280
   DC.W    $300,$2E0,$2C0,$2A0,$280

   SECTION 14
STK:
        DS.B     $4600

    STKPOS:
        DC.L     STK

    STKNDX:
        DC.W     0
        END
```

5.3.2 TDELETE Routine

The optional VRTX32 Configuration Table parameter **Sys-TDELETE-Address** points to a user-supplied routine that is given control when VRTX32 deletes a task. For example, you might use this routine in a program that tracks tasks according to their creation and deletion times. If the task deletes itself, the TSWITCH routine is called after the **TDELETE routine**.

When the TDELETE routine receives control, register A2 contains a pointer to the TCB of the deletor's task. This pointer is not valid before the execution of the VRTX_GO call; it is not valid for tasks deleted during system initialization. Register A1 contains a pointer to the TCB of the task that is being deleted; see Figure 5-4, Environment on Entry to TDELETE Routine.

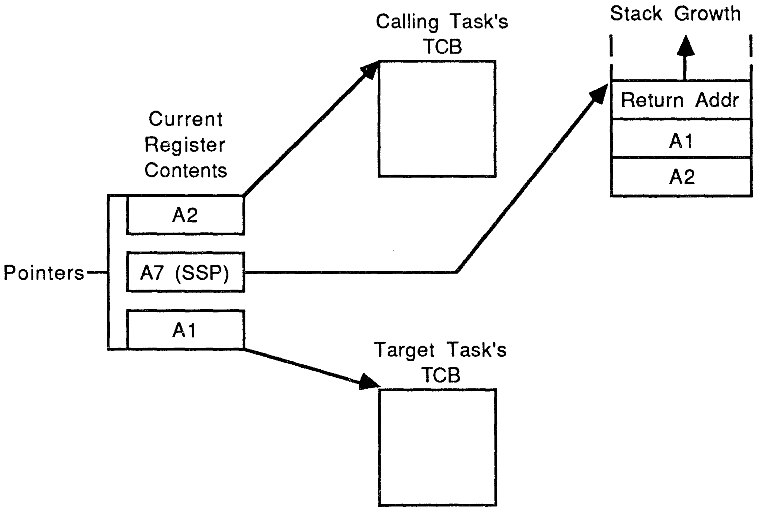


Figure 5-4 Environment on Entry to TDELETE Routine

5.3.3 TSWITCH Routine

The optional VRTX32 Configuration Table parameter **Sys-TSWITCH-Address** points to a user-supplied routine that is given control when VRTX32 switches to another task. Since interrupts can restart the rescheduling procedure, this routine can be called several times before the next task is started.

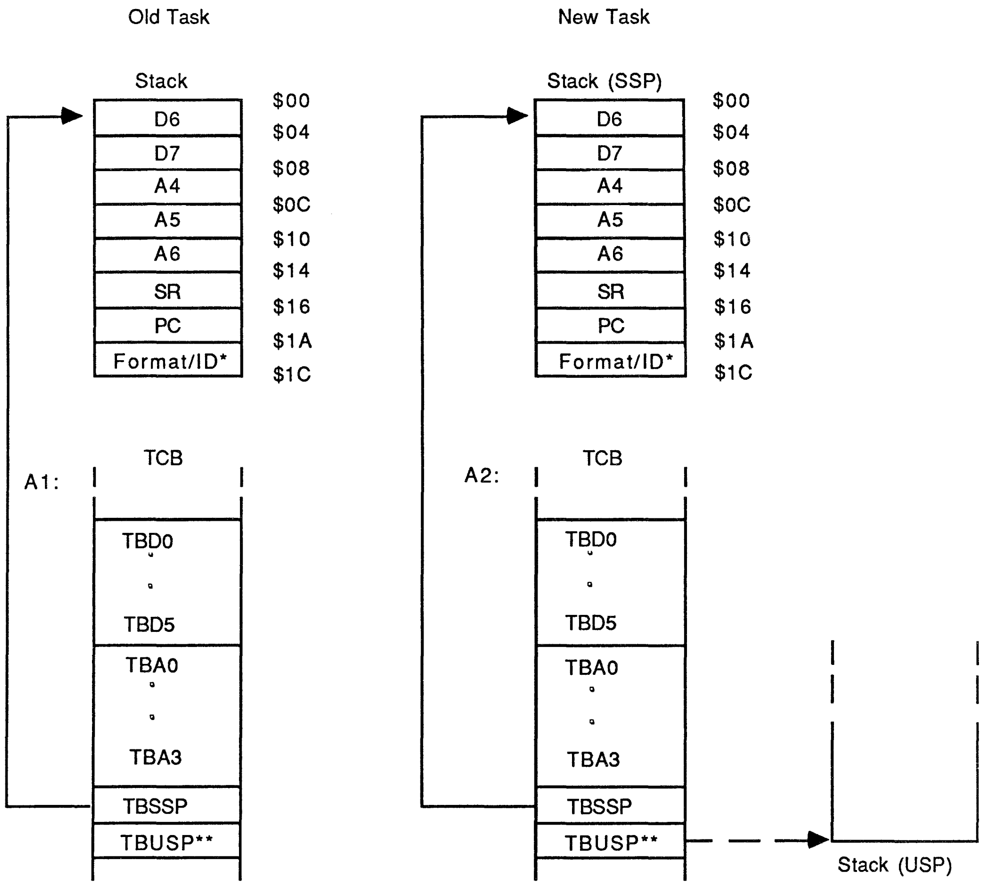
The **TSWITCH routine** is called with a pointer to the old task's TCB in register A1; register A2 contains a pointer to the new task's TCB. For the first task that executes, register A1 contains a pointer to the idle task, and register A2 contains a pointer to the new task's TCB.

The TCREATE routine is not called when the idle task is created. This means that if you use the TCREATE routine to allocate an additional data area to TCBs, or to alter TCBs in any way, the idle task's TCB is not affected. For example, if additional memory is allocated to the TCBs to save special registers, the idle task's TCB will not have this area. If the TSWITCH routine is called to use memory or other items set up during the TCREATE routine, it must check for the idle task. Alternatively, you can create your own low-priority idle task to ensure that the TCREATE sets up all TCBs identically, and that the TSWITCH routine does not need special processing for the VRTX32-created idle task.

Figure 5-5, Environment on Entry to TSWITCH Routine, shows the state of the TCBs and the user stacks at the time the TSWITCH routine gets control.

Note that VRTX32 does not save all registers in the TCB. VRTX32 improves performance by leaving the values of some registers saved on the task stack. Figure 5-5 shows this organization. The registers for both tasks are stored at identical offsets from TBSSP.

Figure 5-6, Complete VRTX32 System, shows how user-defined extensions fit into a VRTX32-based system.



* MC68010 architecture only.

** If new task is in User mode; otherwise, contents are irrelevant.

Figure 5-5 Environment on Entry to TSWITCH Routine

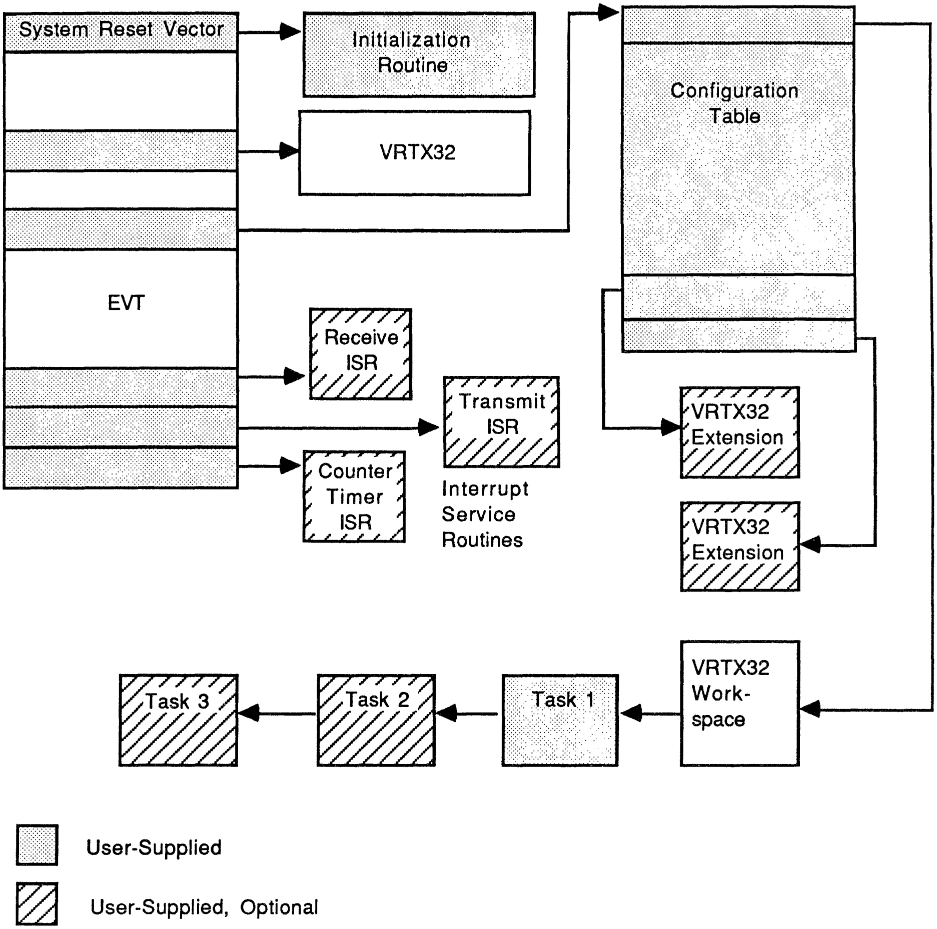


Figure 5-6 Complete VRTX32 System

Chapter 6

Interfacing Software Components

HUNTER ◆ READY
A Division of Ready Systems

6.1 Introduction

Silicon software components are well-designed pieces of software that are used as building blocks. Software components can connect with other software components in a variety of designs, without change to the components themselves. Silicon software components developed by Hunter & Ready include VRTX32, the high-performance versatile real-time executive, IOX, the input/output executive, and FMX, the file management executive.

This chapter discusses integrating VRTX32 with other silicon software components. It includes sections on the component calling conventions of all Hunter & Ready silicon software components, the Component Vector Table (CVT), and the internal structure of software components.

6.2 Component Calling Conventions

This section describes the Hunter & Ready conventions that apply when calling a silicon software component. These conventions include the component call format convention, the component call trap vector convention, and the parameter passing conventions. Using calling conventions ensures a consistent and easy-to-use interface to all silicon software components.

6.2.1 Component Call Format

All requests for silicon software component services are made with a single architecture-dependent instruction. In the instruction set of each architecture supported by Hunter & Ready, there is at least one instruction that causes what is variously known as a **software-generated interrupt**, an exception, or a trap. The M68000 architecture defines a series of TRAP instructions for this purpose.

These instructions make possible a type of run-time linkage. Ordinary subroutine calls require a calling program with built-in knowledge of the address of its called subroutine. However, a software interrupt depends only on the contents of a hardware-defined software interrupt vector to reach its intended destination. Thus, the calling program does not have to be compiled, assembled, linked, or otherwise bound up with the called routine.

By using software interrupt instructions in this way, Hunter & Ready provides software components that are entirely independent from your development system: since a software component need not be bound up with your application code, there is no special reliance on individual linkers, loaders, or assemblers.

6.2.2 Component Call Trap Vector

Corresponding to each software interrupt instruction is an interrupt vector location defined by the computer architecture. Executing a software interrupt instruction causes the hardware to save its current program address and to fetch its new program address from the appropriate vector location. This new program address is the address of the interrupt service routine (ISR).

The M68000 architecture defines 16 distinct TRAP vectors, any one of which can be used to access VRTX32 system calls. Execution of the TRAP instruction causes the new program address to be loaded from a vector at address $080+4*n offset from the base of the EVT.

In VRTX32, a routing mechanism directs all requests for system services to the appropriate silicon software component. The entry point for this service is the start of VRTX32 code. After the TRAP instruction for component access is selected, be sure the corresponding TRAP vector points to the VRTX32's starting address. All components are then accessed through a single software interrupt instruction.

6.2.3 Parameter Passing

All calls to silicon software components are routed to a single entry point. Therefore, a component call must specify the component being called and the desired operation.

In addition, a component must always return a return code to the caller. The return code indicates the success or failure of the operation.

The rest of this section describes these three parameters, as well as additional input and output parameters. These parameters are passed using registers.

Function Code. All silicon software component calls designate their target with a **function code** passed in register D0. This code is a 32-bit number, divided into three fields. The lower two bytes of the function code are sign extended.

Sign Extension	Component ID	Opcode
31 16	15 8	7 0

The **Component ID** field (bits 15-8) specifies a component identification number. Each Hunter & Ready component has a unique component number that is the same for all releases of the component. This identification number does not conflict with other component number assignments. For example, VRTX32 is designated as component number 0, IOX is component number 2, and FMX/DOS is component number 3.

Hunter & Ready reserves the full range of nonnegative component numbers from 0 to 127 for its current and future silicon software components. Negative numbers ranging from -1 to -128 are unreserved and available for integrating user-supplied components. This procedure is described in Section 6.3, Component Vectoring, and Section 6.4, Component Internals.

The **Operation Code** field (bits 7-0) of the function code specifies an operation code, or **Opcode**. The Opcode selects the operation to be invoked in the selected component. Opcode values are defined locally for each component; they do not necessarily have consistent meanings across different components.

Return Code. All silicon software components return an output value known as the **return code** in register D0. (D0 specifies the function code on input and the return code on output.)

The 32-bit return code indicates the success or failure of the requested operation. A return code of $0000 (mnemonic RET_OK) indicates a successful completion for all components. Nonzero return codes indicate unsuccessful completion of an operation.

All nonzero return codes have the Component ID in bits 8 through 15 and an Error Code in bits 0 through 7. Hunter & Ready uses only positive values in the error code to indicate error conditions. This 8-bit error code is specific to the component ID, except for the RET_OK return code. The component ID ensures unique return codes. Hunter & Ready guarantees that its own return codes are unique across all components.

The 32-bit return code has this format:

Sign Extension	Component ID	Error Code
31 16	15 8	7 0

The following are descriptions of error conditions and return codes made by VRTX32/68000 in response to errors with either the Component Vector Table (CVT) or the Opcode Vector Table (OVT) of a specific component.

$0020 CVT pointer in the configuration table is null; no CVT is defined.

$0021 Illegal Component ID. Component ID in D0 is either beyond the range of the maximum component number in the CVT, or it references a CVT entry that is a null pointer (in other words, undefined in the CVT).

$0022 Illegal Opcode ID. Opcode in D0 is either beyond the range of the maximum opcode number in the OVT, or it references an OVT entry that is a null pointer (in other words, undefined in the OVT).

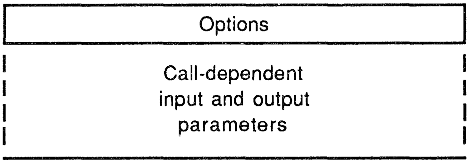
Appendix B, Return Codes, lists the mnemonics, values, and meanings of all VRTX32 return codes assigned by Hunter & Ready.

Other Parameters. Besides the function code and return code parameters, almost all operations performed by silicon software components require additional input and output parameters. By convention, operations accept and return all necessary parameters using machine registers. VRTX32 operations follow this convention.

Other components frequently define a data structure known as a **parameter packet** for particular operations. This structure contains all additional parameters required by a specified operation. Only the packet's address must be passed, typically in register A0.

When values other than the error code must be returned, or when input values cannot be determined at compile time, locate the packet in read/write memory. (However, you can locate packets used only for reference in PROM.) In a multitasking environment, you must ensure that the packet is allocated on a per-task basis. You can accomplish this by allocating a task's packet on the stack or in an area of memory dedicated to that task.

The parameter packet typically has this format:



The first field in Hunter & Ready parameter packets is a 16-bit **Options** word. The format of the rest of the packet can vary from one function to another. The parameters in the packet can be 8-, 16-, or 32-bit integers, pointers, or arbitrary bit fields. We recommend that user-supplied components use the same structure.

Parameter packets can contain input values and output values. Hunter & Ready components do not overwrite input values. For example, a Set File Position call might accept an input parameter that specifies the desired file position and return the actual file position. Do not overwrite the original input value; supply a separate parameter in the parameter packet for the returned file position. This allows packets to be used again with minimum change.

Generally, some parameters in a parameter packet are filled during program execution. For example, file channels, stack addresses, and task-dependent states are not known until the program runs. Thus, for efficiency reasons, parameter packets should be kept small. Large parameters (greater than 32 bits), structures, and varying length parameters (byte strings) should be passed by reference (pointer).

Variable-length character strings are passed by reference. One parameter points to the string, and another parameter specifies the maximum length of the string. When the string is terminated with a null (zero) byte, the string length parameter is not required. Although technically unnecessary, this method of passing strings conforms to the C language convention.

All pointer values should be capable of holding the largest address accessible by the target microprocessor. A special value, all zeros, is reserved to indicate a **null pointer**. The null pointer indicates the absence of the parameter referenced by the pointer value.

6.3 Component Vectoring

Section 6.2, Component Calling Conventions, describes how a single software interrupt instruction accesses all Hunter & Ready silicon software components. The VRTX32 component intercepts each software-generated interrupt, analyzes the function code passed by the caller, and routes the call to the appropriate component.

To accomplish **component vectoring**, you must tell VRTX32 which components are present in the system and where they are located. The **Component Vector Table (CVT)** supplies this information.

The optional CVT-Address parameter in the VRTX32 Configuration Table points to the CVT. When there are no components other than VRTX32 in the system, the CVT is not required, and zero is supplied in the CVT-Address parameter. Figure 6-1, Component Vector Table, shows the CVT's format.

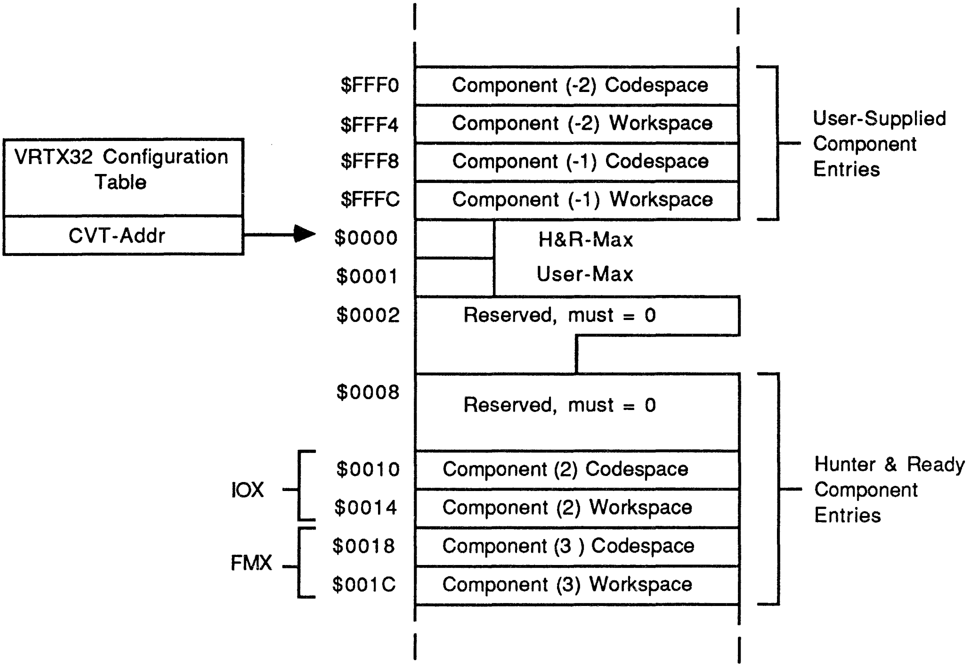


Figure 6-1 Component Vector Table

The first byte in the CVT, **H&R-Max**, specifies the component number of the highest-numbered Hunter & Ready component present in the system.

The second byte in the CVT, **User-Max**, should be set to zero unless there are user-supplied components in the system. These other components must be written according to the guidelines presented in Section 6.4, Component Internals. When such components are present, whether supplied by third-party vendors or by yourself, you must assign them negative component numbers. The value of User-Max must reflect the absolute value of the greatest extent component number. For example, when components numbered -1, -2, and -7 are present in the system, User-Max is specified as 7.

Corresponding to each possible component number, within the limits described by H&R-Max and User-Max, is an 8-byte entry. Entries for Hunter & Ready components are at positive displacements from the start of the CVT. Entries for user-supplied components are at negative displacements. There can be gaps in the enumeration of components present in the system (for instance, components 4 and 5 are present, but not components 2 or 3). You must set all entries corresponding to absent components to zero.

For components present in the system, the corresponding entry contains two pointers. The first specifies the **Codespace Entry Point Address**. The entry point of all Hunter & Ready components is offset zero of the component; thus, the value of this field is equal to the starting address of the component.

The second pointer in a component entry specifies the **Workspace Address**, the address of the primary data area for that component. Ordinarily a silicon software component requires some amount of private memory for its own purposes; the workspace vectors in the CVT are generally used to point to such areas. VRTX32 passes a workspace pointer to a component every time that component is called.

You must initialize the entire CVT before any attempt is made to access components other than VRTX32.

6.4 Component Internals

A user-defined system call handler is intended to be an extension of VRTX32. It increases the repertoire of VRTX32 system calls available to your application. In fact, user-defined system call handlers frequently combine several VRTX32 system calls into one routine. Refer to Section 5.2, User-Defined System Call Handlers.

On the other hand, a user-supplied component usually performs functions outside the scope of VRTX32's executive functions. One example of a user-supplied component might be a database management component. A component that performed data base management would have system calls independent from VRTX32's system calls.

This section describes the rules that must be followed when writing a software component. These rules ensure correct operation with VRTX32 and consistency with the Hunter & Ready calling conventions. Each component referenced in the CVT, including user-supplied components indicated by negative component numbers, must obey the structural constraints outlined here.

These rules include opcode handling rules, register contents rules, stack structure rules, and multitasking considerations. If you do not intend to write your own software components, you can skip the remainder of this chapter.

6.4.1 Opcode Handling

Just as the CVT specifies the location of components in the system, an **Opcode Vector Table (OVT)** specifies the location of **opcode handler routines** (component calls) in a specific component. Figure 6-2, Opcode Vector Table, shows this data structure's format.

Each software component contains an OVT starting at offset zero of the component. The table's address is entered in the CVT as the component's codespace vector.

The first 64 bytes ($40) of the OVT are reserved for use by Hunter & Ready.

At location $40, **Opcode-Max** specifies the highest-numbered operation code valid for the specific component. The designer of the component assigns the opcodes; they are unsigned numbers ranging from 0 through 255.

Corresponding to each possible opcode value, from 0 to the specified maximum, is a long-word entry. These entries start at $50. There can be gaps in the enumeration of valid opcodes (for instance, when 13 is a valid opcode but 11 and 12 are not). The entries corresponding to invalid opcodes should be set to zero.

Each opcode entry contains a 16-bit **Offset** value that specifies the address of the opcode handler routine. This value is the offset from the start of the component.

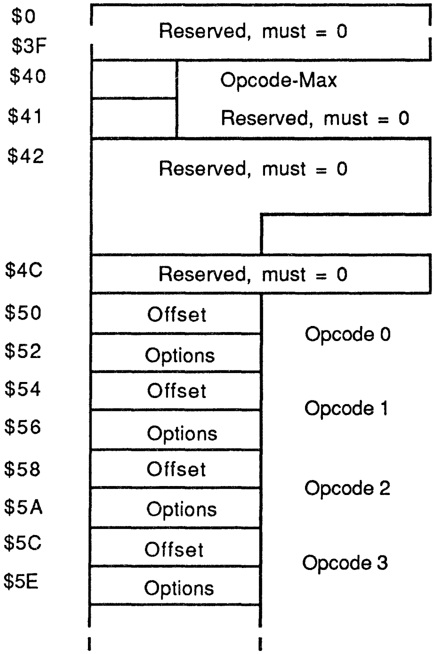


Figure 6-2 Opcode Vector Table

The **Options** value for each entry in the table allows you to specify the mode of operation for an opcode handler. Only the least significant two bits are currently defined; the remaining bits of each Options word are reserved and should always be set to zero. Bit positions are defined as used in the M68000 bit-manipulation instructions (modulo-32).

Table 6-1, Opcode Handler Options, shows the options supported by VRTX32/68000 for use by opcode handler routines.

6.4.2 Register Contents

When entering an opcode handler routine, the six data registers (D0 through D5) and four address registers (A0 through A3) are left intact. This means that the contents of these registers are exactly as they were when the component call was made. For many operations, register A0 specifies the address of a parameter packet. When the call is routed to the component, register D0 contains the caller's 32-bit function code.

Table 6-1 Opcode Handler Options

Bit	Value	Meaning
00	0	Handler is effectively a subroutine, executing with the same preemptibility as the caller.
	1	Handler is nonpreemptible.
01	0	Handler is effectively a system service, executing in Supervisor mode.
	1	Reserved

When returning to the caller, the handler routine should set D0 to indicate a return code value.

The remaining registers (SR, PC, D6, D7, and address registers A4 through A7) are not left intact. Their original values at the time of the call are saved on the stack. An exception is A7, the stack pointer, whose original value is simply its current value adjusted by the size of the registers pushed onto the stack. Address register A7 is set up to point to the top of the stack. Address register A5 points to the component's workspace, as determined by the entry in the CVT.

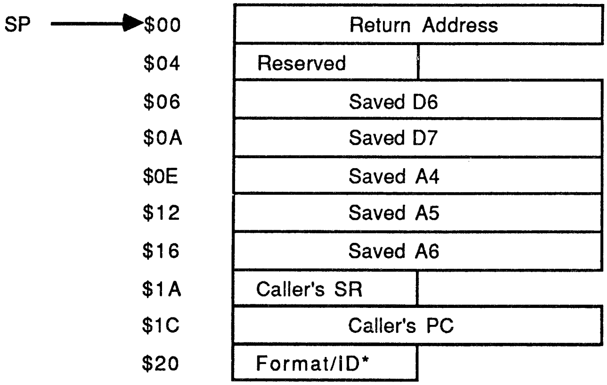
6.4.3 Stack Structure

The format of the stack on entry to an opcode handler routine is shown in Figure 6-3, M68000 Stack Format.

The handler routine returns with an RTS instruction. The return address at the top of the stack is set up for a return into VRTX32, which in turn restores registers from the stack and returns to the caller.

6.4.4 Multitasking Considerations

In an embedded system with some degree of concurrency, a resident multitasking executive must coordinate all operations. Therefore, VRTX32 is the focal point for Hunter & Ready component interface mechanisms.



* MC68010 architecture only.

Figure 6-3 M68000 Stack Format

Every opcode handler routine must be aware of its interaction with the surrounding multitasking environment. When a component is called from the task level, unlike calls from ISRs, the handler routine is presumed reschedulable and preemptible, just like the calling task. The handler can use SC_LOCK and SC_UNLOCK, just as tasks do, to override this assumption and bracket nonpreemptible, critical regions of code. When the nonpreemptible flag is set in the OVT, the nonpreemptibility holds for the entire component call.

When a component handler routine is first invoked, interrupts are enabled to the level they were enabled by the caller. The handler can also disable interrupts for brief sections of critical code, as an alternative to SC_LOCK and SC_UNLOCK.

Chapter 7

System Call Reference

HUNTER
READY
A Division of Ready Systems

7.1 Introduction

This chapter lists all VRTX32 system calls in alphabetic order. The following information is provided for each call:

- The mnemonic name of the call.
- A brief statement of the call's function and operation, including input and output values. The function code is always an input value in register D0 and is shown as a mnemonic followed by the hexadecimal value.
- A list of the call's possible return codes in register D0.
- A list of the environments from which the call can be made:
  - **User initialization code** is system code that precedes the VRTX_GO call.
  - **Interrupt handler code** includes device ISRs, user-supplied system call handlers, and extensions.
  - **Task code** is any task level code.

Conventions followed in this chapter include:

- Numbers preceded by the dollar sign ($) character are hexadecimal numbers; otherwise, numbers are decimal numbers.
- A notation such as D1[7:0] stands for register D1, bits 7 through 0; bit 0 is the least significant bit.
- Unless otherwise noted, all parameters use the full 32 bits of the register. When the parameters use less than the full 32 bits of the register, they use the least significant bits of the register.
- A given parameter's range is restricted only by the register size, unless otherwise noted.

7.2 SC_ACCEPT — Accept Message from Mailbox

This call obtains a long-word (32-bit) nonzero message from a specified mailbox. Unlike SC_PEND, this call does not suspend the caller if no message is present. Instead, VRTX32 returns the error code ER_NMP immediately and the calling task continues execution.

To avoid suspension, ISRs must use SC_ACCEPT rather than SC_PEND to receive messages.

The SC_ACCEPT call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_ACCEPT ($0025)
	A0	=	mailbox address
OUTPUT:	D0	=	return code
	D1	=	message

mailbox address is a 32-bit pointer to the mailbox.

message is a nonzero 32-bit data value. Register D1 remains unchanged if VRTX32 returns ER_NMP.

RETURN CODES

$0000	RET_OK	Successful return
$000B	ER_NMP	No message present

ENVIRONMENTS

This call can be made from task, interrupt handler, and user initialization code.

7.3 SC_FCLEAR — Clear Event

This call clears one or more event flags in the specified event flag group, and returns the event flag group *before* the flags were cleared. An event flag should be cleared with SC_FCLEAR before an attempt is made to post to it again.

This call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_FCLEAR ($001B)
	D1	=	event flag group ID number
	D2	=	event flags
OUTPUT:	D0	=	return code
	D2	=	event flag group

event flag group ID number is a 32-bit value that references the event flag group. VRTX32 returns this number in the SC_FCREATE call.

event flags is a 32-bit mask. Each bit corresponds to one event flag in the event flag group.

event flag group is the state of the 32-bit event flag group before the flags were cleared.

RETURN CODES

$0000	RET_OK	Successful return
$0031	ER_ID	Event flag group ID error

ENVIRONMENTS

This call can be made from task, interrupt handler, and user initialization code.

7.4 SC_FCREATE — Create Event Flag Group

This call creates a long-word (32-bit) event flag group in VRTX32 Workspace, and returns the event flag group ID number.

Each event flag group and semaphore is associated with a control block. You specify the maximum number of control blocks in the VRTX32 Configuration Table (refer to Section 4.2, VRTX32 Configuration Table). If you try to create more event flag groups and/or semaphores than you've specified in the configuration table, VRTX32 returns the ER_NOCB error code.

The SC_FCREATE call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_FCREATE ($0017)
OUTPUT:	D0	=	return code
	D1	=	event flag group ID number

event flag group ID number is a 32-bit value that references the event flag group.

RETURN CODES

$0000	RET_OK	Successful return
$0030	ER_NOCB	No control blocks available

ENVIRONMENTS

This call can be made from task and user initialization code.

7.5 SC_FDELETE — Delete Event Flag Group

This call deletes the specified event flag group, making the event flag group's control block available for reuse.

The SC_FDELETE call initiates the rescheduling procedure if the forced delete option is specified and there are tasks pending on the event flag group. In this case, all pending tasks are readied.

If this option is not specified, VRTX32 returns the ER_PND error code if you try to delete an event flag group with pending tasks.

INPUT:	D0	=	SC_FDELETE ($0018)
	D1	=	event flag group ID number
	D2	=	force delete option
OUTPUT:	D0	=	return code

event flag group ID number is a 32-bit value that references the event flag group. VRTX32 returns this number in the SC_FCREATE call.

force delete option is a 32-bit value that can be formatted in two ways:

FORMAT 1: Delete event flag group only if no tasks are pending.
D2 = 0

FORMAT 2: Delete event flag group and ready all pending tasks.
D2 = 1

RETURN CODES

$0000	RET_OK	Successful return
$0031	ER_ID	Event flag group ID error
$0032	ER_PND	Tasks pending on event flag group; Format 1 only

ENVIRONMENTS

This call can be made from task and user initialization code.

7.6 SC_FINQUIRY — Event Flag Group Inquiry

This call obtains the specified event flag group. The SC_FINQUIRY call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_FINQUIRY ($001C)
	D1	=	event flag group ID number
OUTPUT:	D0	=	return code
	D2	=	event flag group

event flag group ID number is a 32-bit value that references the event flag group. VRTX32 returns this number in the SC_FCREATE call.

event flag group is the current 32-bit event flag group.

RETURN CODES

$0000	RET_OK	Successful return
$0031	ER_ID	Event flag group ID error

ENVIRONMENTS

This call can be made from task, interrupt handler, and user initialization code.

7.7 SC_FPEND — Pend on Event Flag Group

This call pends for one or more events on the specified event flag group, and returns the event flag group that readied the caller. You specify whether it is an AND pend or an OR pend. If the specified event flags are not set, the task suspends and a task switch occurs. The task is not readied until the appropriate flag(s) are set.

To satisfy an AND pend, all specified event flags must have a value of one simultaneously. For example, suppose a task is waiting for both Flag 1 and Flag 2. Flag 1 is set, but is immediately cleared. Next, Flag 2 is set. The task continues to pend, because Flag 1 and Flag 2 have not had a value of one at the same time.

You can issue SC_FPEND with a nonzero timeout value. In this case, VRTX32 returns the ER_TMO error code to the calling task if the event flag(s) are not set in the specified number of VRTX32 clock ticks; refer to Section 3.3.1, Real-Time Clock Support. (To pend without a timeout, set the timeout parameter to zero.) The timeout is not synchronized with the VRTX32 clock. Thus, a timeout value of one VRTX32 clock tick results in the task's timeout period ending on the next occurrence of any VRTX32 clock tick. The actual elapsed time the task times out could be less than one VRTX32 clock tick in this example.

If the task is suspended on an event flag group and the group is deleted, the task is readied and VRTX32 returns the ER_DEL error code.

A pending task that has been explicitly suspended with SC_TSUSPEND can be notified of an event. However, it remains suspended until it is explicitly resumed with SC_TRESUME.

INPUT:	D0	=	SC_FPEND ($0019)
	D1	=	event flag group ID number
	D2	=	timeout value
	D3	=	event flags
	D4	=	mask option
OUTPUT:	D0	=	return code
	D2	=	event flag group

event flag group ID number is a 32-bit value that references the event flag group. VRTX32 returns this number in the SC_FCREATE call.

timeout value is a 32-bit number of VRTX32 clock ticks. A value of zero indicates no timeout is requested.

event flags is a 32-bit mask. Each bit corresponds to one event flag in the event flag group.

mask option is a 32-bit value that can be formatted in two ways:

FORMAT 1:	Pend on an OR mask; any of the specified event flags ready the task. D4 = 0
FORMAT 2:	Pend on an AND mask; all specified event flags must be set to ready the task. D4 = 1

event flag group is the 32-bit event flag group when the task is readied. Register D2 is invalid if VRTX32 returns ER_ID or ER_TMO.

RETURN CODES

$0000	RET_OK	Successful return
$000A	ER_TMO	Timeout
$0031	ER_ID	Event flag group ID error
$0033	ER_DEL	Event flag group is deleted

ENVIRONMENTS

This call can be made from task code only.

7.8 SC_FPOST — Post Event to Event Flag Group

This call posts one or more events to the specified event flag group. Every task that is pending for this event (and has its OR or AND mask satisfied) is readied, and the rescheduling procedure occurs.

If an event flag is already set (one), and SC_FPOST tries to set it again, VRTX32 returns the ER_OVF error code. However, if SC_FPOST specifies several event flags, and some of them are already set, VRTX32 returns ER_OVF *and* sets any event flags that were not previously set.

INPUT:	D0	=	SC_FPOST ($001A)
	D1	=	event flag group ID number
	D2	=	event flags
OUTPUT:	D0	=	return code

event flag group ID number is a 32-bit value that references the event flag group. VRTX32 returns this number in the SC_FCREATE call.

event flags is a 32-bit mask. Each bit corresponds to one event flag in the event flag group.

RETURN CODES

$0000	RET_OK	Successful return
$0031	ER_ID	Event flag group ID error
$0034	ER_OVF	Event flag already set

ENVIRONMENTS

This call can be made from task, interrupt handler, and user initialization code.

7.9 SC_GBLOCK — Get Memory Block

This call obtains a memory block from a partition of memory blocks managed by VRTX32. The SC_PCREATE call, which creates memory partitions, specifies the block size. You can repeat SC_GBLOCK until all blocks in a partition are allocated.

This call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_GBLOCK ($0006)
	D1[15:0]	=	partition ID number
OUTPUT:	D0	=	return code
	A0	=	memory block address

partition ID number is a 16-bit value that references the partition. A value of zero is allowed.

memory block address is a 32-bit pointer to the start of a memory block.

RETURN CODES

$0000	RET_OK	Successful return
$0003	ER_MEM	No memory blocks available
$000E	ER_PID	Partition ID error; no such partition

ENVIRONMENTS

This call can be made from task, interrupt handler, and user initialization code.

7.10 SC_GETC — Get Character

This call obtains the next character from the supported I/O device. When the 64-byte buffer of received characters is empty, the calling task suspends until a character is received. This call does not echo the character onto the output device. However, the ISR that supports the device can echo each character it receives.

The SC_GETC call initiates the rescheduling procedure if no character is present.

INPUT:	D0	=	SC_GETC ($000D)
OUTPUT:	D0	=	return code
	D1[7:0]	=	next received character

next received character is an 8-bit value.

RETURN CODES

$0000	RET_OK	Successful return

ENVIRONMENTS

This call can be made from task code only.

7.11 SC_GTIME — Get Time

This call obtains the current value of the VRTX32 clock, specified as a number of VRTX32 clock ticks. The SC_GTIME call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_GTIME ($000A)
OUTPUT:	D0	=	return code
	D1	=	VRTX32 clock value

VRTX32 clock value is a 32-bit value. This is the current value of the VRTX32 clock, specified as a number of VRTX32 clock ticks.

RETURN CODES

$0000	RET_OK	Successful return

ENVIRONMENTS

This call can be made from task, interrupt handler, and user initialization code.

7.12 SC_LOCK — Disable Task Rescheduling

This call disables the VRTX32 rescheduling procedure until an explicit SC_UNLOCK call is issued. The task that issues the SC_LOCK call retains processor control, even when higher-priority tasks are ready to run.

The SC_LOCK and SC_UNLOCK calls are used in pairs. VRTX32 keeps an internal count of locks and unlocks so that nested instances of these calls do not prematurely end a scheduling lock. (The maximum lock/unlock nest count supported is 65,535.) For example, nested subroutines and procedures can contain critical code that other tasks cannot interrupt. When a nested routine issues an SC_UNLOCK, the SC_UNLOCK cancels the last SC_LOCK call only.

Use the SC_LOCK call with caution, since it disrupts VRTX32's normal scheduling of the multitasking environment. However, SC_LOCK does not affect interrupt handling.

CAUTION

After issuing SC_LOCK, the program should not make any system calls that could lead to the suspension of the current task. This event causes unpredictable results.

INPUT:	D0	=	SC_LOCK ($0020)
OUTPUT:	D0	=	return code

RETURN CODES

$0000	RET_OK	Successful return

ENVIRONMENTS

This call can be made from task and interrupt handler code.

7.13 SC_PCREATE — Create Memory Partition

This call creates a partition of contiguous memory managed by VRTX32. The SC_PCREATE call specifies the partition ID number and the block size that successive SC_GBLOCK calls use to obtain memory blocks.

This call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_PCREATE ($0022)
	D1[15:0]	=	partition ID number
	D2	=	partition size
	D3	=	block size
	A0	=	partition address
OUTPUT:	D0	=	return code

partition ID number is a 16-bit value that references the partition. A value of zero is allowed.

partition size is a 32-bit value that specifies the total size of the partition, specified as a count of bytes. This size must be greater than or equal to block size. In addition, the partition cannot contain more than 32K blocks, although the partition can be extended with the SC_PEXTEND call. To avoid wasted space, the partition size should be an integer multiple of the block size.

block size is a 32-bit count of bytes. This value must not equal zero.

partition address is a 32-bit pointer to the partition.

RETURN CODES

$0000	RET_OK	Successful return
$0003	ER_MEM	No memory available; insufficient system memory for VRTX32 control structures
$000E	ER_PID	Partition ID error; ID number already assigned
$0012	ER_IIP	Invalid input parameter; returned in these cases: • When the block size is specified as zero • When the partition size is less than the block size • When the partition contains more than 32K blocks

ENVIRONMENTS

This call can be made from task and user initialization code.

7.14 SC_PEND — Pend for Message from Mailbox

This call obtains a long-word (32-bit) nonzero message from a specified mailbox. If a message is present, the task receives it and continues execution. VRTX32 resets the mailbox to zero.

If the mailbox is empty, the task suspends and a task switch occurs. The task is not readied until it receives a message.

You can issue SC_PEND with a nonzero timeout value. In this case, VRTX32 returns the ER_TMO error code to the calling task if it does not receive a message in the specified number of VRTX32 clock ticks; refer to Section 3.3.1, Real-Time Clock Support. (To pend without a timeout, set the timeout parameter to zero.) The timeout is not synchronized with the VRTX32 clock. Thus, a timeout value of one VRTX32 clock tick results in the task's timeout period ending on the next occurrence of any VRTX32 clock tick. The actual elapsed time the task times out could be less than one VRTX32 clock tick in this example.

A pending task that has been explicitly suspended with SC_TSUSPEND can receive a message. However, it remains suspended until it is explicitly resumed with SC_TRESUME.

When several tasks are pending on the same mailbox, the highest-priority task receives the message.

To avoid suspension, ISRs must use SC_ACCEPT rather than SC_PEND to receive messages.

INPUT:	D0	=	SC_PEND ($0009)
	D1	=	timeout value
	A0	=	mailbox address
OUTPUT:	D0	=	return code
	D1	=	message

timeout value is a 32-bit number of VRTX32 clock ticks. A value of zero indicates no timeout is requested.

mailbox address is a 32-bit pointer to the mailbox.

message is a nonzero 32-bit data value. Register D1 remains unchanged if VRTX32 returns ER_TMO.

RETURN CODES

$0000	RET_OK	Successful return
$000A	ER_TMO	Timeout

ENVIRONMENTS

This call can be made from task code only.

7.15 SC_PEXTEND — Extend Memory Partition

This call extends a memory partition previously defined by SC_PCREATE. The extension encompasses an additional range of memory locations. This extension does not have to be contiguous with the memory location of the original partition. However, the block size in the extension is the same as defined in the original partition.

The SC_PEXTEND call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_PEXTEND ($0023)
	D1[15:0]	=	partition ID number
	D2	=	extension size
	A0	=	extension address
OUTPUT:	D0	=	return code

partition ID number is a 16-bit value that references the partition. A value of zero is allowed.

extension size is a 32-bit value that specifies the total size of the extension in bytes. The extension size must be greater than or equal to block size. An extension cannot contain more than 32K blocks, although multiple extensions can be used to define more blocks. To avoid wasted space, the extension size should be an integer multiple of the block size.

extension address is a 32-bit pointer to the extension.

RETURN CODES

$0000	RET_OK	Successful return
$0003	ER_MEM	No memory available; insufficient system memory for VRTX32 control structures
$000E	ER_PID	Partition ID error; no such partition
$0012	ER_IIP	Invalid input parameter; returned in these cases: • When the extension size is less than the block size • When the extension contains more than 32K blocks

ENVIRONMENTS

This call can be made from task and user initialization code.

7.16 SC_POST — Post Message to Mailbox

This call posts a long-word (32-bit) nonzero message to a specified mailbox. Do not specify a zero message; zero indicates an empty mailbox.

When there is already a message in the mailbox (mailbox value is nonzero), VRTX32 returns the error code ER_MIU to the posting task. The posting task continues execution.

A message posted to a mailbox with SC_POST is immediately allocated to any task pending for the message; the message is not saved in the mailbox itself in this case.

When a higher-priority task is pending on the mailbox, a task switch occurs.

INPUT:	D0	=	SC_POST ($0008)
	D1	=	message
	A0	=	mailbox address
OUTPUT:	D0	=	return code

message is a nonzero 32-bit data value.

mailbox address is a 32-bit pointer to the mailbox.

RETURN CODES

$0000	RET_OK	Successful return
$0005	ER_MIU	Mailbox in use
$0006	ER_ZMW	Zero message

ENVIRONMENTS

This call can be made from task, interrupt handler, and user initialization code.

7.17 SC_PUTC — Put Character

This call specifies the next character to transmit to the supported I/O device. When the 64-byte buffer of characters to transmit is full, the calling task suspends until the buffer is available (that is, one character is transmitted).

The SC_PUTC call must be used with caution during initialization. An attempt to put more than 64 characters in the buffer before VRTX_GO causes unpredictable results.

The SC_PUTC call initiates the rescheduling procedure when the transmit buffer is full.

INPUT:	D0	=	SC_PUTC ($000E)
	D1[7:0]	=	character
OUTPUT:	D0	=	return code

character is an 8-bit value.

RETURN CODES

$0000	RET_OK	Successful return

ENVIRONMENTS

This call can be made from task code only.

7.18 SC_QACCEPT — Accept Message from Queue

This call obtains a long-word (32-bit) message from a specified queue. Unlike SC_QPEND, this call does not suspend the caller if no message is present. Instead, VRTX32 returns the error code ER_NMP immediately and the calling task continues execution. VRTX32 does not return a message when the call is unsuccessful.

To avoid suspension, ISRs must use SC_QACCEPT rather than SC_QPEND to receive messages.

The SC_QACCEPT call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_QACCEPT ($0028)
	D1[15:0]	=	queue ID number
OUTPUT:	D0	=	return code
	D2	=	message

queue ID number is a 16-bit value that references the queue. A value of zero is allowed.

message is a 32-bit data value. A value of zero is allowed.

RETURN CODES

$0000	RET_OK	Successful return
$000B	ER_NMP	No message present
$000C	ER_QID	Queue ID error; no such queue

ENVIRONMENTS

This call can be made from task, interrupt handler, and user initialization code.

7.19 SC_QCREATE — Create Message Queue

This call creates a message queue from available VRTX32 Workspace (refer to Section 4.3, Determining VRTX32 Workspace Size). You specify a queue ID number and the number of queue entries (the queue size). Tasks pend on the queue in priority order.

When the queue is created, VRTX32 adds one queue entry to the number you specify. This additional entry is reserved at the beginning of the queue for a message posted with the SC_QJAM call when the queue is otherwise full (refer to Section 7.22, SC_QJAM—Jam Message to Queue).

The SC_QCREATE call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_QCREATE ($0029)
	D1[15:0]	=	queue ID number
	D2[15:0]	=	number of queue entries
OUTPUT:	D0	=	return code

queue ID number is a 16-bit value that references the queue. A value of zero is allowed.

number of queue entries is a 16-bit value representing the maximum number of 32-bit messages supported by this queue.

RETURN CODES

$0000	RET_OK	Successful return
$0003	ER_MEM	No memory available; insufficient system memory for VRTX32 control structures
$000C	ER_QID	Queue ID error; ID number already assigned

ENVIRONMENTS

This call can be made from task and user initialization code.

7.20 SC_QECREATE — Create FIFO Message Queue

This call creates a message queue from available VRTX32 Workspace (refer to Section 4.3, Determining VRTX32 Workspace Size). You specify a queue ID number, the number of queue entries (the queue size), and whether tasks should pend in priority order or FIFO order.

When the queue is created, VRTX32 adds one queue entry to the number you specify. This additional entry is reserved at the beginning of the queue for a message posted with the SC_QJAM call when the queue is otherwise full (refer to Section 7.22, SC_QJAM—Jam Message to Queue).

The SC_QECREATE call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_QECREATE ($001F)
	D1[15:0]	=	queue ID number
	D2[15:0]	=	number of queue entries
	D3	=	task pend order option
OUTPUT:	D0	=	return code

queue ID number is a 16-bit value that references the queue. A value of zero is allowed.

number of queue entries is a 16-bit value representing the maximum number of 32-bit messages supported by this queue. (VRTX32 adds one to this number for a message posted with SC_QJAM.)

task pend order option is a 32-bit value that can be formatted in two ways:

FORMAT 1: Pend tasks in priority order.
D3 = 0

FORMAT 2: Pend tasks in FIFO order.
D3 = 1

RETURN CODES

$0000	RET_OK	Successful return
$0003	ER_MEM	No memory available; insufficient system memory for VRTX32 control structures
$000C	ER_QID	Queue ID error; ID number already assigned

ENVIRONMENTS

This call can be made from task and user initialization code.

7.21 SC_QINQUIRY — Queue Status Inquiry

This call obtains the current count of messages waiting in a queue. When the count is nonzero, the actual contents of the head-of-queue message is returned to the caller. Although the caller is given a copy of the first message, the message remains queued. The calling program must make the SC_QPEND or SC_QACCEPT call to remove the message.

If the queue is full and messages have been posted with the SC_QJAM call, the count is one higher than that specified in the SC_QCREATE or SC_QECREATE call.

The SC_QINQUIRY call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_QINQUIRY ($002A)
	D1[15:0]	=	queue ID number
OUTPUT:	D0	=	return code
	D2	=	message
	D3	=	count of messages in queue

queue ID number is a 16-bit value that references the queue. A value of zero is allowed.

message is a 32-bit data value. A value of zero is allowed. Register D2 remains unchanged if there are no messages in the queue (count is zero).

count of messages in queue is a 32-bit value representing the current number of messages in the queue. Register D3 remains unchanged if VRTX32 returns ER_QID.

RETURN CODES

$0000	RET_OK	Successful return
$000C	ER_QID	Queue ID error; no such queue

ENVIRONMENTS

This call can be made from task, interrupt handler, and user initialization code.

7.22 SC_QJAM — Jam Message to Queue

This call posts a long-word (32-bit) message to the beginning of a specified queue. The next task to pend for a message receives the most recently "jammed" message before any messages posted with the SC_QPOST call. If a task is pending on the queue when the message is posted, the message is immediately allocated to the task and is not first saved in the queue.

When a queue is created with SC_QCREATE or SC_QECREATE, VRTX32 adds one queue entry to the number you specify. This additional entry is reserved at the beginning of the queue for a message posted with the SC_QJAM call when the queue is otherwise full. This ensures that at least one SC_QJAM call is successful. If the queue is full and a message has already been "jammed", VRTX32 returns the ER_QFL return code.

As an alternative to mixing SC_QJAMs and SC_QPOSTs, you can use the SC_QJAM call to post all messages to the queue. In this case, you can use the full size of the queue (including the reserved entry), and messages are handled in last-in/first-out (LIFO) order.

When a higher-priority task is pending on the queue, a task switch occurs.

INPUT:	D0	=	SC_QJAM ($001E)
	D1[15:0]	=	queue ID number
	D2	=	message
OUTPUT:	D0	=	return code

queue ID number is a 16-bit value that references the queue. A value of zero is allowed.

message is a 32-bit data value. A value of zero is allowed.

RETURN CODES

$0000	RET_OK	Successful return
$000C	ER_QID	Queue ID error; no such queue
$000D	ER_QFL	Queue full

ENVIRONMENTS

This call can be made from task, interrupt handler, and user initialization code.

7.23 SC_QPEND — Pend for Message from Queue

This call obtains a long-word (32-bit) message from a specified queue. If a message is present, the task receives it and continues execution. If the queue is empty, the task suspends and a task switch occurs. The task is not readied until it receives a message.

You can issue SC_QPEND with a nonzero timeout value. In this case, VRTX32 returns the ER_TMO error code to the calling task if no message is received in the specified number of VRTX32 clock ticks; refer to Section 3.3.1, Real-Time Clock Support. (To pend without a timeout, set the timeout parameter to zero.) The timeout is not synchronized with the VRTX32 clock. Thus, a timeout value of one VRTX32 clock tick results in the task's timeout period ending on the next occurrence of any VRTX32 clock tick. The actual elapsed time the task times out could be less than one VRTX32 clock tick in this example.

When several tasks pend on the same queue, the tasks receive the message in either priority or FIFO order. The order is determined by the queue create call used; refer to Section 7.19, SC_QCREATE—Create Message Queue, and Section 7.20, SC_QECREATE—Create FIFO Message Queue.

A pending task that has been explicitly suspended with SC_TSUSPEND can receive a message. However, it remains suspended until it is explicitly resumed with SC_TRESUME.

To avoid suspension, ISRs must use SC_QACCEPT rather than SC_QPEND to receive messages.

INPUT:	D0	=	SC_QPEND ($0027)
	D1[15:0]	=	queue ID number
	D2	=	timeout value
OUTPUT:	D0	=	return code
	D2	=	message

queue ID number is a 16-bit value that references the queue. A value of zero is allowed.

timeout value is a 32-bit number of VRTX32 clock ticks. A value of zero indicates no timeout is requested.

message is a 32-bit data value. A value of zero is allowed. Register D2 remains unchanged if VRTX32 returns ER_TMO.

RETURN CODES

$0000	RET_OK	Successful return
$000A	ER_TMO	Timeout
$000C	ER_QID	Queue ID error; no such queue

ENVIRONMENTS

This call can be made from task code only.

7.24 SC_QPOST — Post Message to Queue

This call posts a long-word (32-bit) message to the end of a specified queue. (To post the message to the beginning of the queue, refer to Section 7.22, SC_QJAM—Jam Message to Queue.) A message posted to a queue with SC_QPOST is immediately allocated to any task pending for the message; the message is not saved in the queue itself in this case.

When a higher-priority task is pending on the queue, a task switch occurs.

INPUT:	D0	=	SC_QPOST ($0026)
	D1[15:0]	=	queue ID number
	D2	=	message
OUTPUT:	D0	=	return code

queue ID number is a 16-bit value that references the queue. A value of zero is allowed.

message is a 32-bit data value. A value of zero is allowed.

RETURN CODES

$0000	RET_OK	Successful return
$000C	ER_QID	Queue ID error; no such queue
$000D	ER_QFL	Queue full

ENVIRONMENTS

This call can be made from task, interrupt handler, and user initialization code.

7.25 SC_RBLOCK — Release Memory Block

This call releases a memory block back to the partition it came from. The SC_RBLOCK call should be used to release all blocks back to the partition before SC_TDELETE is issued, because blocks are not automatically released when a task is deleted.

The SC_RBLOCK call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_RBLOCK ($0007)
	D1[15:0]	=	partition ID number
	A0	=	memory block address
OUTPUT:	D0	=	return code

partition ID number is a 16-bit value that references the partition. A value of zero is allowed.

memory block address is a 32-bit pointer to the memory block.

RETURN CODES

$0000	RET_OK	Successful return
$0004	ER_NMB	Not a memory block; specified address does not reference a block previously allocated from the specified partition
$000E	ER_PID	Partition ID error; no such partition

ENVIRONMENTS

This call can be made from task, interrupt handler, and user initialization code.

7.26 SC_SCREATE — Create Semaphore

This call creates a word (16-bit) counting semaphore in VRTX32 Workspace, and returns the semaphore ID number. You specify the initial value of the semaphore and whether tasks pend on the semaphore in priority order or FIFO order.

An initial value of zero indicates that the resource the semaphore is "locking" starts in a locked state. A nonzero value indicates how many tasks can access the resource at one time.

Each semaphore and event flag group is associated with a control block. You specify the maximum number of control blocks in the VRTX32 Configuration Table (refer to Section 4.2, VRTX32 Configuration Table). If you try to create more semaphores and/or event flag groups than you've specified in the configuration table, VRTX32 returns the ER_NOCB error code.

This call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_SCREATE ($002B)
	D1	=	initial value
	D2	=	task pend order option
OUTPUT:	D0	=	return code
	D1	=	semaphore ID number

initial value is a 32-bit value that specifies the initial value of the semaphore from 0 to 65,535. A zero value indicates that the resource starts in a locked state.

task pend order option is a 32-bit value that can be formatted in two ways:

FORMAT 1: Pend tasks in priority order.
D2 = 0

FORMAT 2: Pend tasks in FIFO order.
D2=1

semaphore ID number is a 32-bit value that references the semaphore.

RETURN CODES

$0000	RET_OK	Successful return
$0030	ER_NOCB	No control blocks available

ENVIRONMENTS

This call can be made from task and user initialization code.

7.27 SC_SDELETE — Delete Semaphore

This call deletes the specified semaphore, making the semaphore's control block available for reuse.

The SC_SDELETE call initiates the rescheduling procedure if the forced delete option is specified and there are tasks pending on the semaphore. In this case, all pending tasks are readied.

If this option is not specified, VRTX32 returns the ER_PND error code if you try to delete a semaphore with pending tasks.

INPUT:	D0	=	SC_SDELETE ($002C)
	D1	=	semaphore ID number
	D2	=	force delete option
OUTPUT:	D0	=	return code

semaphore ID number is a 32-bit value that references the semaphore. VRTX32 returns this number in the SC_SCREATE call.

force delete option is a 32-bit value that can be formatted in two ways:

FORMAT 1: Delete semaphore only if no tasks are pending.
D2 = 0

FORMAT 2: Delete semaphore and ready all pending tasks.
D2 = 1

RETURN CODES

$0000	RET_OK	Successful return
$0031	ER_ID	Semaphore ID error
$0032	ER_PND	Tasks pending on semaphore; Format 1 only

ENVIRONMENTS

This call can be made from task and user initialization code.

7.28 SC_SINQUIRY — Semaphore Inquiry

This call obtains the specified semaphore's value. The SC_SINQUIRY call does not initiate the rescheduling procedure.

INPUT:	D0	=	SC_SINQUIRY ($002F)
	D1	=	semaphore ID number
OUTPUT:	D0	=	return code
	D2	=	semaphore value

semaphore ID number is a 32-bit value that references the semaphore. VRTX32 returns this number in the SC_SCREATE call.

semaphore value is the current 16-bit semaphore value.

RETURN CODES

$0000	RET_OK	Successful return
$0031	ER_ID	Semaphore ID error

ENVIRONMENTS

This call can be made from task, interrupt handler, and user initialization code.

7.29 SC_SPEND — Pend on Semaphore

This call pends on a restricted resource's semaphore. If the semaphore has a nonzero value, the semaphore is decremented and the task continues execution.

If the specified semaphore has a zero value, the task suspends and a task switch occurs. The task is not readied until the resource becomes available and the SC_SPOST call increments the semaphore.

You can issue SC_SPEND with a nonzero timeout value. In this case, VRTX32 returns the ER_TMO error code to the calling task if the resource does not become available in the specified number of VRTX32 clock ticks; refer to Section 3.3.1, Real-Time Clock Support. (To pend without a timeout, set the timeout parameter to zero.) The timeout is not synchronized with the VRTX32 clock. Thus, a timeout value of one VRTX32 clock tick results in the task's timeout period ending on the next occurrence of any VRTX32 clock tick. The actual elapsed time the task times out could be less than one VRTX32 clock tick in this example.

When several tasks pend on the same semaphore, the tasks are readied in either priority or FIFO order. The order is determined by the SC_SCREATE call; refer to Section 7.26, SC_SCREATE—Create Semaphore.

If the task is suspended on a semaphore and the semaphore is deleted, the task is readied and VRTX32 returns the ER_DEL error code.

A pending task that has been explicitly suspended with SC_TSUSPEND can receive access to a resource. However, it remains suspended until it is explicitly resumed with SC_TRESUME.

INPUT:	D0	=	SC_SPEND ($002D)
	D1	=	semaphore ID number
	D2	=	timeout value
OUTPUT:	D0	=	return code

semaphore ID number is a 32-bit value that references the semaphore. VRTX32 returns this number in the SC_SCREATE call.

timeout value is a 32-bit number of VRTX32 clock ticks. A value of zero indicates no timeout is requested.

RETURN CODES

$0000	RET_OK	Successful return
$000A	ER_TMO	Timeout
$0031	ER_ID	Semaphore ID error
$0033	ER_DEL	Semaphore is deleted

ENVIRONMENTS

This call can be made from task code only.

7.42 UI_ENTER — Enter Interrupt Handler

This call starts an ISR if interrupt stack switching is enabled; the ISR must use UI_EXIT for termination. If interrupt stack switching is enabled, UI_ENTER switches from the stack of the interrupted task to the interrupt stack.

UI_ENTER can also be used if interrupt stack switching is not enabled. However, this is effectively a null operation. UI_ENTER and UI_EXIT are optimized for fast performance.

> **NOTE:** If interrupt stack switching is enabled, save only register D0 on the stack before issuing UI_ENTER. Other registers must be saved after UI_ENTER. Otherwise, the registers are saved on the wrong stack and are not restored properly. See Example 7-1.

Example 7-1 The UI_ENTER call

```
UIFENTER    EQU   $16    * UI_ENTER function code
VRTX        EQU   $00    * VRTX32 trap number

   MOVE.L   D0,-(SP)     * save D0
   MOVEQ.L  #UIFENTER,D0 * UI_ENTER function code
   TRAP     #VRTX        * trap into VRTX32
   MOVE.L   D1,-(SP)     * save additional registers
*   .
*   .                      interrupt servicing
*   .
```

INPUT:	D0	=	UI_ENTER ($0016)
OUTPUT:	D0	=	return code

RETURN CODES

$0000	RET_OK	Successful return

ENVIRONMENTS

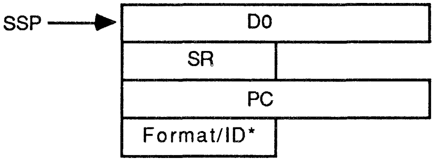
This call can be made from device ISRs only.

7.43 UI_EXIT — Exit Interrupt Handler

This call exits an ISR. Unlike the RTE instruction, the UI_EXIT call interfaces with VRTX32. This means that task rescheduling can occur in response to ISR activity, such as VRTX32 clock updates, posted messages, or character I/O. When any VRTX32 call is made from the ISR, this call must be used.

When ISRs are nested, rescheduling occurs only after the last UI_EXIT. Refer to Chapter 3, Interrupt Support, for more information.

Because the UI_EXIT call is made with register D0, the ISR must save the original value of D0 at the top of the stack. The stack must have this format when calling UI_EXIT:



* MC68010 architecture only.

In addition, UI_ENTER and UI_EXIT must be used if interrupt stack switching is enabled. When UI_ENTER starts the ISR, UI_EXIT must end the ISR. UI_ENTER and UI_EXIT are optimized for fast performance.

INPUT:	D0 = UI_EXIT ($0011)
OUTPUT:	No return is possible

ENVIRONMENTS

This call can be made from device ISRs only.

7.44 UI_RXCHR — Received-Character Interrupt

This call is used by an ISR to transfer each character to VRTX32 as it is received from the supported input device. VRTX32 manages the buffering of such characters and their transfer to tasks that issue SC_GETC and SC_WAITC calls. This buffer is 64 bytes in length. After the ISR that calls UI_RXCHR exits (with UI_EXIT), the rescheduling procedure can occur.

INPUT:	D0	=	UI_RXCHR ($0013)
	D1[7:0]	=	character
OUTPUT:	D0	=	return code

character is an 8-bit value.

RETURN CODES

$0000	RET_OK	Successful return
$0007	ER_BUF	Buffer full

ENVIRONMENTS

This call can be made from interrupt handler code only.

Example 7-2 is an example of the UI_RXCHR call. Note that this example shows only the relevant VRTX32 calls; the ISR may require additional instructions to support proper device operation.

Example 7-2 The UI_RXCHR Call

```
U_DATA      EQU   $D8     * 8251A USART data port

UIFRXCHR    EQU   $13     * UI_RXCHR function code
UIFEXIT     EQU   $11     * UI_EXIT function code
VRTX        EQU   $00     * VRTX32 trap number

INT_RXCHR:
   MOVE.L   D0,-(SP)      * save D0
```

```
      MOVE.L    D1,-(SP)       * save D1
      MOVE.B    U_DATA,D1      * input character from USART
      MOVEQ.L   #UIFRXCHR,D0   * call UI_RXCHR to give
      TRAP      #VRTX          *   character to VRTX32
   *                             should check return code here
      MOVE.L    (SP)+,D1       * restore D1
      MOVEQ.L   #UIFEXIT,D0    * call UI_EXIT to restore D0
      TRAP      #VRTX          *   with possible rescheduling
```

7.45 UI_TIMER — Announce Timer Interrupt

This call is used by an ISR to inform VRTX32 that a time interval, or VRTX32 clock tick, has occurred. (VRTX32 processes the timer tick at the last UI_EXIT call in a group of nested interrupts.)

A task switch can occur after a UI_EXIT call when UI_TIMER readies a delayed or pended task whose priority is higher than the interrupted task.

INPUT:	D0	=	UI_TIMER ($0012)
OUTPUT:	D0	=	return code

RETURN CODES

$0000	RET_OK	Successful return

ENVIRONMENTS

This call can be made from task (to simulate VRTX32 clock ticks) and interrupt handler code only.

Example 7-3 is a counter-timer ISR. It ends with UI_EXIT instead of an RTE instruction; this allows VRTX32 to reschedule priorities when the call results in the expiration of a time-slicing interval, a delay interval, or a pend timeout. Note that this example shows only the relevant VRTX32 calls; the ISR may require additional instructions to support proper device operation.

Example 7-3 Counter-Timer ISR

```
UIFTIMER    EQU  $12        * UI_TIMER function code
UIFEXIT     EQU  $11        * UI_EXIT function code
VRTX        EQU  $00        * VRTX32 trap number

INTERRUPT_CLK:
   MOVE.L   D0,-(SP)        * save D0
   MOVEQ.L #UIFTIMER,D0     * call UI_TIMER to inform
   TRAP     #VRTX           *   VRTX32 that a tick has expired
   MOVEQ.L #UIFEXIT,D0      * call UI_EXIT, restoring D0,
   TRAP     #VRTX           *   with possible rescheduling
```

7.46 UI_TXRDY — Transmit-Ready Interrupt

This call is used by an ISR to inform VRTX32 that it is ready to transmit another character to the supported character output device. VRTX32 returns the next character from its accumulated buffer of SC_PUTC requests to the ISR.

The ISR and VRTX32 use the TXRDY driver routine to transmit characters. Refer to Section 3.3.3, Character I/O Support, for more information about this user-supplied routine.

When there are no outstanding SC_PUTC requests (the output buffer is empty), UI_TXRDY returns with the ER_NCP error code.

When the output buffer is full and one or more tasks have suspended on an SC_PUTC, the rescheduling procedure is initiated at the UI_EXIT call.

INPUT:	D0	=	UI_TXRDY ($0014)
OUTPUT:	D0	=	return code
	D1[7:0]	=	character

character is an 8-bit value. Register D1 remains unchanged if VRTX32 returns ER_NCP.

RETURN CODES

$0000	RET_OK	Successful return
$0010	ER_NCP	No character present

ENVIRONMENTS

This call can be made from interrupt handler code only.

7.47 VRTX_GO — Start Application Execution

VRTX_GO causes VRTX32 to gain control; VRTX32 then begins multitasking by starting the highest-priority task. This call should be issued only after VRTX_INIT. No return is made to the caller.

> **NOTE:** All user initialization code, including the VRTX_GO call, must execute in Supervisor mode.

INPUT:	D0 = VRTX_GO ($0031)
OUTPUT:	No output values

ENVIRONMENTS

This call can be made from user initialization code only.

7.48 VRTX_INIT — Initialize VRTX32

This call, issued from user-supplied code executed by system reset, causes VRTX32 to perform its initialization activities. User initialization code must execute in Supervisor mode.

VRTX_INIT requires the use of a temporary stack. You must set up a small (100-byte) stack before calling VRTX_INIT. Refer to Chapter 4, Configuration and Initialization, for more information.

INPUT:	D0	=	VRTX_INIT ($0030)
OUTPUT:	D0	=	return code

RETURN CODES

$0000	RET_OK	Successful return
$000F	ER_INI	Insufficient VRTX32 Workspace
$0011	ER_ICP	Invalid configuration table parameter

ENVIRONMENTS

This call can be made from user initialization code only.

Appendix A

System Call Summary

HUNTER
READY
A Division of Ready Systems

This appendix contains the set of VRTX32 system calls and the input data, including the hexadecimal value of the function code supplied in D0. The return data is shown in square brackets [ ].

The error code is always returned in D0 except for the UI_EXIT and VRTX_GO calls. In these cases, no return is possible. This error code value is not shown in this appendix.

Table A-1 Task Management

input/[returned] data					
Mnemonic	*D0*	*D1*	*D2*	*D3*	*A0*
SC_TCREATE	$0000	priority	task ID	mode	address
SC_TDELETE	$0001	priority or ID			
SC_TSUSPEND	$0002	priority or ID			
SC_TRESUME	$0003	priority or ID			
SC_TPRIORITY	$0004	ID	new priority		
SC_TINQUIRY	$0005	ID/[ID]	[priority]	[status]	[TCB addr]
SC_LOCK	$0020				
SC_UNLOCK	$0021				

Table A-2 Memory Allocation

input/[returned] data					
Mnemonic	*D0*	*D1*	*D2*	*D3*	*A0*
SC_GBLOCK	$0006	partition ID			[address]
SC_RBLOCK	$0007	partition ID			address
SC_PCREATE	$0022	partition ID	partition size	block size	address
SC_PEXTEND	$0023	partition ID	extension size		address

Table A-3 Communication and Synchronization

input/[returned] data					
Mnemonic	*D0*	*D1*	*D2*	*D3*	*D4/A0*
SC_POST	$0008	message			A0: address
SC_PEND	$0009	timeout/ [message]			A0: address
SC_ACCEPT	$0025	[message]			A0: address
SC_QPOST	$0026	queue ID	message		
SC_QJAM	$001E	queue ID	message		
SC_QPEND	$0027	queue ID	timeout/ [message]		
SC_QACCEPT	$0028	queue ID	[message]		
SC_QCREATE	$0029	queue ID	count		

(continued on next page.)

Table A-3, continued

input/[returned] data					
Mnemonic	*D0*	*D1*	*D2*	*D3*	*D4/A0*
SC_QECREATE	$001F	queue ID	count	pend option	
SC_QINQUIRY	$002A	queue ID	[message]	[count]	
SC_FCREATE	$0017	[event flag group ID]			
SC_FDELETE	$0018	event flag group ID	force option		
SC_FPOST	$001A	event flag group ID	event flags		
SC_FPEND	$0019	event flag group ID	timeout/ [event flag group]	event flags	D4: mask option
SC_FCLEAR	$001B	event flag group ID	event flags/ [event flag group]		
SC_FINQUIRY	$001C	event flag group ID	[event flag group]		
SC_SCREATE	$002B	initial value/ [semaphore ID]	pend option		
SC_SDELETE	$002C	semaphore ID	force option		
SC_SPOST	$002E	semaphore ID			
SC_SPEND	$002D	semaphore ID	timeout		
SC_SINQUIRY	$002F	semaphore ID	[semaphore value]		

Table A-4 Interrupt Support

input/[returned] data					
Mnemonic	*D0*	*D1*	*D2*	*D3*	*A0*
UI_ENTER	$0016				
UI_EXIT	$0011				

Table A-5 Real-Time Clock

input/[returned] data					
Mnemonic	*D0*	*D1*	*D2*	*D3*	*A0*
SC_GTIME	$000A	[VRTX32 clock value]			
SC_STIME	$000B	VRTX32 clock value			
SC_TDELAY	$000C	ticks			
SC_TSLICE	$0015	ticks			
UI_TIMER	$0012				

Table A-6 Character I/O

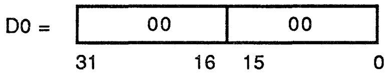
input/[returned] data					
Mnemonic	*D0*	*D1*	*D2*	*D3*	*A0*
SC_GETC	$000D	[character]			
SC_PUTC	$000E	character			
SC_WAITC	$000F	character			
UI_RXCHR	$0013	character			
UI_TXRDY	$0014	[character]			

Table A-7 Initialization

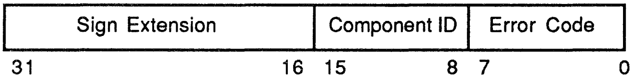
input/[returned] data					
Mnemonic	*D0*	*D1*	*D2*	*D3*	*A0*
VRTX_INIT	$0030				
VRTX_GO	$0031				

Appendix B
Return Codes

Other components can return completion status codes that are specific to the individual components. Consult the appropriate user's guide for information about IOX and FMX completion status codes. All Hunter & Ready components return the same RET_OK value for successful completions:



All error returns from Hunter & Ready components have this format:



After a VRTX32 system call executes, a 32-bit return code is returned in register D0. Table B-1, Return Codes, lists the values, mnemonics, meanings, and affected commands of all possible return codes (specified in hexadecimal).

When using VRTX32 to access other components, VRTX32 can reject invalid component calls with the return codes shown in Table B-2.

Table B-1 Return Codes

D0	Mnemonic	Meaning	Affected Commands
$0000	RET_OK	Successful return	[All valid commands]
$0001	ER_TID	Task ID error	SC_TCREATE, SC_TDELETE, SC_TINQUIRY, SC_TPRIORITY SC_TRESUME, SC_TSUSPEND
$0002	ER_TCB	No TCBs available	SC_TCREATE
$0003	ER_MEM	No memory available	SC_GBLOCK, SC_PCREATE, SC_PEXTEND, SC_QCREATE, SC_QECREATE
$0004	ER_NMB	Not a memory block	SC_RBLOCK
$0005	ER_MIU	Mailbox in use	SC_POST
$0006	ER_ZMW	Zero message	SC_POST
$0007	ER_BUF	Buffer full	UI_RXCHR
$0008	ER_WTC	Previous SC_WAITC already in progress	SC_WAITC
$000A	ER_TMO	Timeout	SC_FPEND, SC_PEND, SC_QPEND, SC_SPEND
$000B	ER_NMP	No message present	SC_ACCEPT, SC_QACCEPT
$000C	ER_QID	Queue ID error	SC_QACCEPT, SC_QCREATE, SC_QECREATE, SC_QJAM, SC_QINQUIRY, SC_QPEND, SC_QPOST
$000D	ER_QFL	Queue full	SC_QJAM, SC_QPOST
$000E	ER_PID	Partition ID error	SC_GBLOCK, SC_PCREATE, SC_PEXTEND, SC_RBLOCK
$000F	ER_INI	Insufficient VRTX32 Workspace	VRTX_INIT
$0010	ER_NCP	No character present	UI_TXRDY
$0011	ER_ICP	Invalid configuration table parameter	VRTX_INIT

(continued on next page.)

Table B-1, continued

D0	Mnemonic	Meaning	Affected Commands
$0012	ER_IIP	Invalid input parameter	SC_PCREATE, SC_PEXTEND, SC_TDELETE, SC_TRESUME, SC_TSUSPEND
$0030	ER_NOCB	No control blocks available	SC_FCREATE, SC_SCREATE
$0031	ER_ID	Event flag group or semaphore ID error	SC_FCLEAR, SC_FDELETE, SC_FINQUIRY, SC_FPEND, SC_FPOST, SC_SDELETE, SC_SINQUIRY, SC_SPEND, SC_SPOST
$0032	ER_PND	Tasks pending on event flag group or semaphore	SC_FDELETE, SC_SDELETE
$0033	ER_DEL	Event flag group or semaphore is deleted	SC_FPEND, SC_SPEND
$0034	ER_OVF	Event flag already set or semaphore overflow	SC_FPOST, SC_SPOST

Table B-2 Return Codes Indicating Invalid Component Calls

D0	Mnemonic	Meaning
$0009	ER_ISC	Invalid system call (VRTX32 opcode invalid)
$0020	ER_CVT	Component Vector Table not present
$0021	ER_COM	Undefined component
$0022	ER_OPC	Undefined opcode for component

Appendix C

EVT and TCB Formats

HUNTER
READY
A Division of Ready Systems

C.1 Introduction

VRTX32 uses several standard data structures for run-time information storage. This appendix describes the EVT and the TCB. Another standard data structure VRTX32 uses is the configuration table, which is described in Chapter 4, Configuration and Initialization.

C.2 Exception Vector Table

The M68000 architecture defines a **Exception Vector Table (EVT)** to control access to service routines for hardware-generated interrupts and software-generated traps. You specify your own interrupt and trap handlers.

Figure C-1, Exception Vector Table, is a simplified example of an EVT. Motorola technical publications supply complete details.

In this figure, the TRAP #0 instruction accesses VRTX32. However, any software interrupt vector can be used. The PC value specifies the base address of VRTX32. Also shown in this figure is the pointer to the VRTX32 Configuration Table at vector 64 (offset $100).

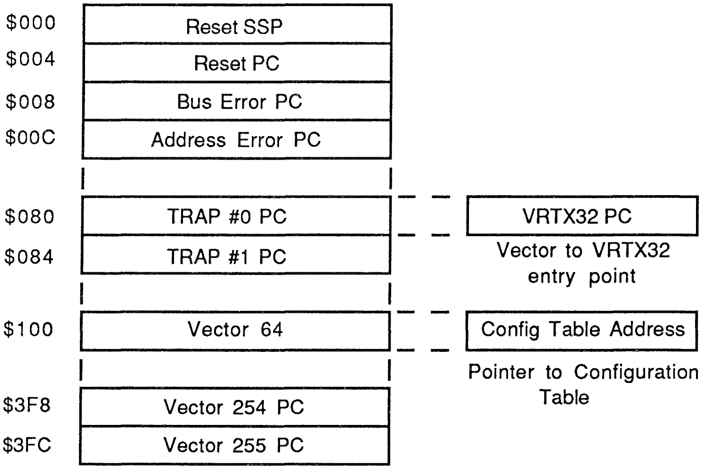
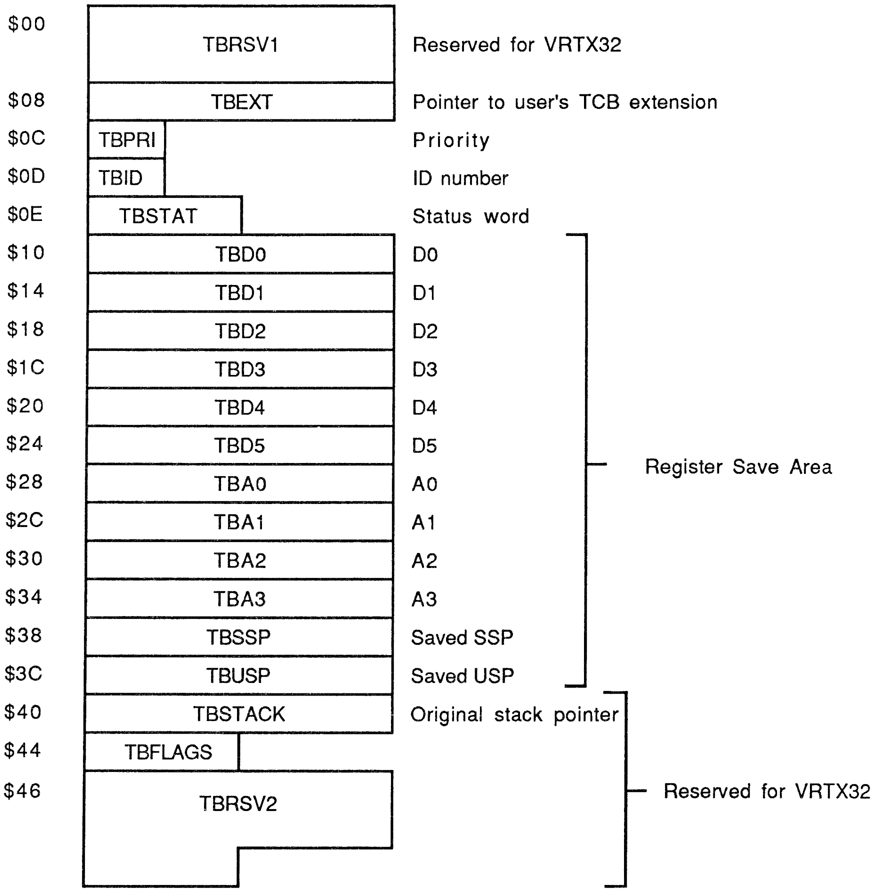


Figure C-1 Exception Vector Table

C.3 Task Control Block Format

For each task created, VRTX32 maintains in its workspace a **Task Control Block (TCB)**. Each TCB records the contents of registers D0 through D5, A0 through A3, SSP, USP, and other status information for its task. The format of the TCB is shown in Figure C-2, Task Control Block.



$00	TBRSV1	Reserved for VRTX32	
$08	TBEXT	Pointer to user's TCB extension	
$0C	TBPRI	Priority	
$0D	TBID	ID number	
$0E	TBSTAT	Status word	
$10	TBD0	D0	Register Save Area
$14	TBD1	D1	
$18	TBD2	D2	
$1C	TBD3	D3	
$20	TBD4	D4	
$24	TBD5	D5	
$28	TBA0	A0	
$2C	TBA1	A1	
$30	TBA2	A2	
$34	TBA3	A3	
$38	TBSSP	Saved SSP	
$3C	TBUSP	Saved USP	
$40	TBSTACK	Original stack pointer	Reserved for VRTX32
$44	TBFLAGS		
$46	TBRSV2		

Figure C-2 Task Control Block

The SC_TINQUIRY call returns the TBSTAT (status word) value from the TCB. When the status word is zero, the associated task is ready to run. When the status word is nonzero, the idle task is running or the task is suspended for one or more reasons. The bit settings of the status word indicate the reasons for suspension; see Table C-1.

Table C-1 Status Word Bit Values

Mnemonic	Bit Value	Reason for Suspension	Suspending Call
TBSSUSP	$01	Explicitly suspended	SC_TSUSPEND
TBSMBOX	$02	Suspended for mailbox message	SC_PEND
TBSGETC	$04	Suspended for character input	SC_GETC
TBSPUTC	$08	Suspended for character output	SC_PUTC
TBSWAITC	$10	Suspended for special character	SC_WAITC
TBSDELAY	$20	Suspended for task delay*	SC_TDELAY
TBSQUEUE	$40	Suspended for queue message	SC_QPEND
	$80	Reserved	-not applicable-
TBSIDLE	$100	Idle task	-not applicable-
TBSFLAG	$200	Suspended on event flag group	SC_FPEND
TBSSEMA	$400	Suspended on semaphore	SC_SPEND

* Also set for pend calls when a timeout is in effect.

Appendix D

An Example

This appendix provides an introduction to running an application program with VRTX32. This example is written specifically for the Microbar DBC68K2 68000 board with the Microbar DBUG68K monitor.

The board support package included in this appendix initializes VRTX32, creates a single task, initializes the EVT vectors, the interrupt controller device, the counter-timer device, and the USART device. It then starts multitasking.

The interactive example included in this appendix contains two tasks. The first task repeatedly reads characters from the I/O port and posts them to a mailbox. The second task is created by the first task and pends at the mailbox waiting for messages. As each character is received, the second task outputs the character to the I/O port and pends again at the mailbox. In other words, this example echoes characters.

Also included in this appendix is the **vrtxvisi.inc** file. This file contains the standard equates used to define VRTX32 function codes, error codes, and the TCB and configuration table structures.

Figure D-1, Example Configuration, shows the configuration of the example.

Figure D-2, Example Memory Organization, shows the memory organization of the example. Both VRTX32 and the optional monitor are in PROM, and the remaining code is in read/write memory. The mailbox is included in the area labeled **Variables**.

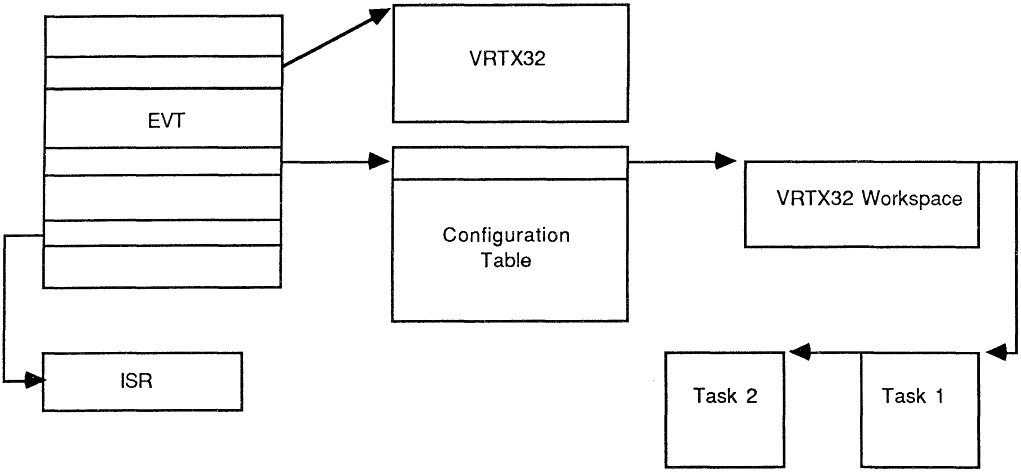


Figure D-1 Example Configuration

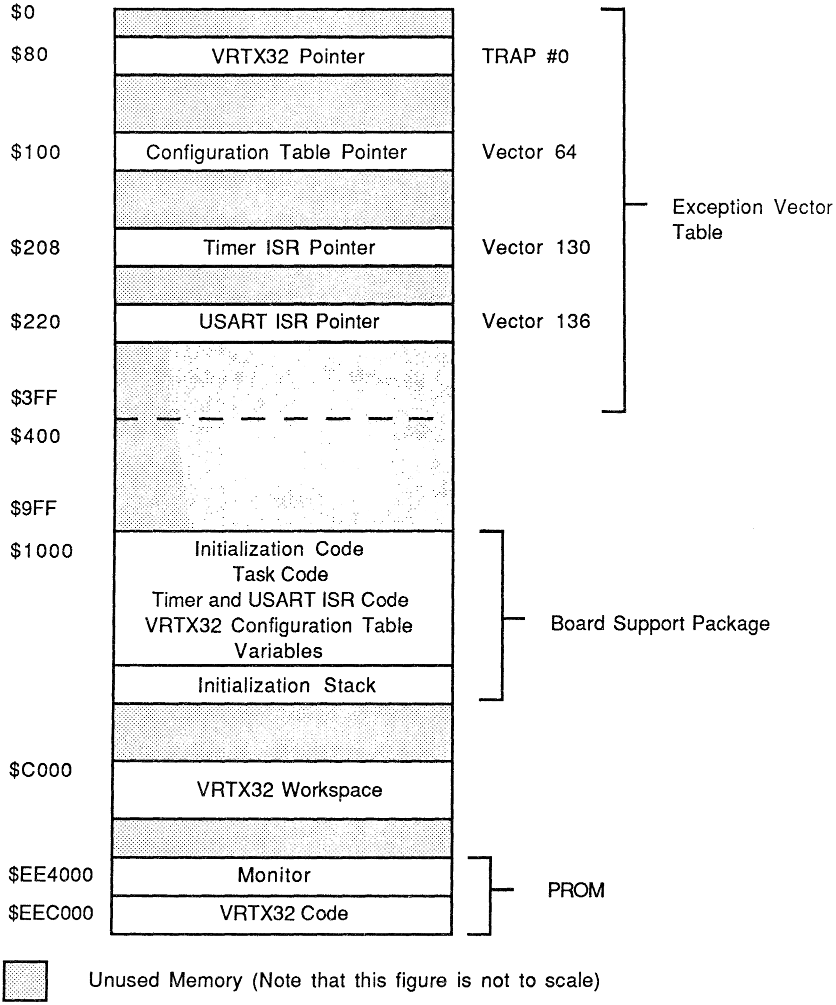


Figure D-2 Example Memory Organization

D.1 Example Board Support Package

```
*********************************************************************
*                                                                   *
*                    VRTX32 BOARD SUPPORT PACKAGE                   *
*                                                                   *
*  FILE:    bsp.asm                                                 *
*  VERSION: 1.04                          DATE: APRIL 1987          *
*                                                                   *
*  COPYRIGHT 1987, HUNTER & READY, INC.                             *
*                                                                   *
*********************************************************************
*                                                                   *
*  This is an example board support package for the Microbar        *
*  DBC68K2 68000 CPU board.                                         *
*                                                                   *
*********************************************************************

* Include VRTX32 definitions

   NOLIST
   INCLUDE 'vrtxvisi.inc'      * VRTX32 equates
   LIST

* Codes put in register D7 at breakpoints

MONITOR    EQU   $0FF00        * monitor return code
VXINERR    EQU   $0FF01        * VRTX_INIT error
TCRTERR    EQU   $0FF04        * SC_TCREATE error
RXCHERR    EQU   $0FF06        * UI_RXCHR error
BREAKPT    EQU   3             * breakpoint TRAP value

* Memory allocation

VRTXCODE   EQU   $0EEC000      * VRTX32 codespace address

* Interrupt vectors

VRTX_VCT   EQU   $80           * VRTX32 vector (trap 0)
CFTBL_VCT  EQU   $100          * configuration table vector
INT53_VCT  EQU   $208          * Intel 8253 vector
INT74_VCT  EQU   $220          * Intel 8274 vector

* External references

   XREF MAIN                   * application program

   SECTION 13

* VRTX32 Configuration Table

   XDEF CFTBL                  * VRTX32 configuration table
CFTBL:

   DC.L     $C000              * VRTX32 workspace start address
   DC.L     3988                  * VRTX32 workspace size
   DC.W     126                   * system stack size
   DC.W     0                     * no interrupt stack
   DC.W     0                     * no control blocks
   DC.W     0                     * reserved, must be 0
   DC.W     0                     * use default idle stack size (128)
   DC.W     0                     * reserved, must be 0
   DC.W     7                     * component-disable-level
   DC.W     126                   * user stack size
   DC.L     0                     * reserved, must be 0
   DC.W     3                     * user task count
   DC.W     0                     * reserved, must be 0
   DC.L     VTXRDY                * VRTX32 TXRDY driver address
   DC.L     0                     * no TCREATE routine
   DC.L     0                     * no TDELETE routine
   DC.L     0                     * no TSWITCH routine
   DC.L     0                     * component vector table

   SECTION 14

* Initialization stack

   XDEF STACKBAS
STACKBAS:

   DS.B    100

   XDEF STACKTOP
STACKTOP:

   SECTION 0

*********************************************************************
*                                                                   *
*                 ENTRY: Start actual execution here                *
*                                                                   *
*********************************************************************

* Initialize VRTX32 and stack

   XDEF ENTRY
ENTRY:

   MOVE.W   #$2700,SR             * status register; interrupts disabled
   MOVE.L   #CFTBL,CFTBL_VCT      * VRTX32 Configuration Table vector
   MOVE.L   #VRTXCODE,VRTX_VCT    * load VRTX32 vector
   MOVE.L   #STACKTOP,SP          * set up init stack
   MOVE.L   #VRTXFINIT,D0         * VRTX_INIT function code
   TRAP     #VRTX                 * call VRTX32
   TST.L    D0                    * any errors?
   BEQ      INITASK               * no, continue

   MOVE.L   #VXINERR,D7           * VRTX_INIT error code
   TRAP     #BREAKPT              * back to monitor
```

```
* Create application task

INITASK:

   MOVEQ.L  #0,D1                * priority = 0
   MOVEQ.L  #0,D2                * ID = 0
   MOVEQ.L  #0,D3                * user mode
   MOVE.L   #MAIN,A0             * load task address
   MOVE.L   #SCFTCREATE,D0       * SC_TCREATE function code
   TRAP     #VRTX                * call VRTX32
   TST.L    D0                   * any errors?
   BEQ      INITCHIP             * no, continue

   MOVE.L   #TCRTERR,D7          * SC_TCREATE error code
   TRAP     #BREAKPT             * back to monitor

* Initialize chips and start VRTX32

INITCHIP:

   JSR      S8259                * initialize the 8259
   JSR      S8253                * initialize the 8253
   JSR      S8274                * initialize the 8274

   MOVE.L   #VRTXFGO,D0          * VRTX_GO function code
   TRAP     #VRTX                * call VRTX32

*******************************************************************
*                                                                 *
*      Intel 8259A Interrupt Controller Initialization Routine    *
*                                                                 *
*******************************************************************

D8259      EQU   $EFFF80         * 8259 device address
D59CNTRL   EQU   D8259+1         * 8259 control port
D59DATA    EQU   D8259+3         * 8259 data port

   XDEF S8259                    * initialization routine entry point
S8259:

* Initial steps

   MOVE.B   #$00,D59DATA         * sync up
   MOVE.B   #$00,D59DATA         * sync up
   MOVE.B   #$00,D59DATA         * sync up
   MOVE.B   #$00,D59DATA         * sync up

* Set Initialization Command Word 1 (ICW1) to edge-triggered mode,
* single 8259 (ICW3 not needed), ICW4 needed

   MOVE.B   #$13,D59CNTRL        * ICW1 (ICW4, no ICW3)

* Set ICW2 to EVT addresses $200 to $21C

   MOVE.B   #$80,D59DATA         * ICW2 (vector)
```

```
* Set ICW4 to not special fully nested mode, buffered mode/master,
* auto EOI, 8086/8088 mode.  Then return to caller.

   MOVE.B   #$0F,D59DATA        * ICW4
   RTS                          * return to caller

*********************************************************************
*                                                                   *
*                      Intel 8253 Counter-Timer                     *
*                                                                   *
*********************************************************************

D8253       EQU   $EFFF40       * 8253 device address
D53CNT0     EQU   D8253+1       * 8253 counter 0
D53CNT1     EQU   D8253+3       * 8253 counter 1
D53CNT2     EQU   D8253+5       * 8253 counter 2
D53CNTRL    EQU   D8253+7       * 8253 control port

* Initialization routine

   XDEF S8253                   * initialization routine entry point
S8253:

* Initialize counter 2 (baud rate) 9600 baud

   MOVE.B   #$B6,D53CNTRL       * set control word (mode 3)
   MOVE.B   #$08,D53CNT2        * set least sig byte (lsb) of count
   MOVE.B   #$00,D53CNT2        * set most sig byte (msb) of count

* Initialize counter 0 (clock) 6.66ms;
* set delay to $2000 (6.66ms with 1.25MHz clock)

   MOVE.B   #$36,D53CNTRL       * set control word (mode 3)
   MOVE.B   #$00,D53CNT0        * set lsb of count
   MOVE.B   #$20,D53CNT0        * set msb of count

* Load the Exception Vector Table (EVT) and return

   MOVE.L   #I8253,INT53_VCT    * vector 130 (decimal)
   RTS                          * return to caller

* Interrupt service routine (ISR)

   XDEF I8253                   * ISR entry point
I8253:

   MOVE.L   D0,-(SP)            * save D0
   MOVE.L   #UIFTIMER,D0        * UI_TIMER function code
   TRAP     #VRTX               * call VRTX32
   MOVE.L   #UIFEXIT,D0         * UI_EXIT function code
   TRAP     #VRTX               * call VRTX32
```

```
**********************************************************************
*                                                                    *
*                         Intel 8274 USART                           *
*                                                                    *
**********************************************************************

D8274       EQU   $EFFF20        * Intel 8274 USART address
ADATA74     EQU   D8274+1        * 8274 data channel A
BDATA74     EQU   D8274+3        * 8274 data channel B
ACNTRL74    EQU   D8274+5        * 8274 control/status channel A
BCNTRL74    EQU   D8274+7        * 8274 control/status channel B
RXMASK      EQU   $7F            * received character mask
EOI         EQU   $38            * 8274 end-of-interrupt

* Initialization routine

   XDEF S8274                    * initialization routine entry point
S8274:

* Initialize channel A

   MOVE.B   #$10,ACNTRL74        * reset pointer to WR0A
   MOVE.B   #$10,ACNTRL74        * point to command register WR0A
   MOVE.B   #$18,ACNTRL74        * channel A reset
   NOP                           * delay #3 to process reset channel A
   NOP                           * delay #2 to process reset channel A
   NOP                           * delay #1 to process reset channel A
   NOP                           * delay #0 to process reset channel A
   MOVE.B   #$14,ACNTRL74        * point to command register WR4A
   MOVE.B   #$44,ACNTRL74        * clock rate x16; 1 stop bit; no parity
   MOVE.B   #$11,ACNTRL74        * point to command register WR1A
   MOVE.B   #$02,ACNTRL74        * interrupt on all Rx & ignore parity

* Fixed vector; Tx interrupt enable

   MOVE.B   #$13,ACNTRL74        * point to command register WR3A
   MOVE.B   #$C1,ACNTRL74        * Rx 8 bits/char; Rx enable
   MOVE.B   #$15,ACNTRL74        * point to command register WR5A
   MOVE.B   #$68,ACNTRL74        * Tx 8 bits/char; Tx enable

* Initialize channel B

   MOVE.B   #$10,BCNTRL74        * reset pointer to WR0B
   MOVE.B   #$10,BCNTRL74        * point to command register WR0B
   MOVE.B   #$18,BCNTRL74        * channel B reset
   NOP                           * delay #3 to process reset channel B
   NOP                           * delay #2 to process reset channel B
   NOP                           * delay #1 to process reset channel B
   NOP                           * delay #0 to process reset channel B
   MOVE.B   #$14,BCNTRL74        * point to command register WR4B
   MOVE.B   #$44,BCNTRL74        * clock rate x16; 1 stop bit; no parity
   MOVE.B   #$11,BCNTRL74        * point to command register WR1B
   MOVE.B   #$1A,BCNTRL74        * interrupt on all Rx & ignore parity

* Fixed vector; Tx interrupt enable
```

```
   MOVE.B   #$13,BCNTRL74       * point to command register WR3B
   MOVE.B   #$C1,BCNTRL74       * Rx 8 bits/char; Rx enable
   MOVE.B   #$15,BCNTRL74       * point to command register WR5B
   MOVE.B   #$68,BCNTRL74       * Tx 8 bits/char; Tx enable
   MOVE.B   #$12,BCNTRL74       * point to command register WR2B
   MOVE.B   #$88,BCNTRL74       * vector for interrupt acknowledge

* Initialize channel B to interrupt vector mode

   MOVE.B   #$12,ACNTRL74       * point to command register WR2A
   MOVE.B   #$30,ACNTRL74       * nonvectored interrupt mode for ch B

* Put ISR vector in Exception Vector Table (EVT) and return

   MOVE.L   #I8274,INT74_VCT    * vector 136 (decimal)
   RTS                          * return to caller

* Interrupt service routine (ISR)

   XDEF I8274                   * ISR entry point
I8274:

   MOVE.L   D0,-(SP)            * save D0
   MOVE.L   D1,-(SP)            * save D1

* USART only provides one interrupt output, so ISR must
* decide if it's a transmitter or receiver interrupt

   BTST     #0,BCNTRL74         * Tx interrupt?
   BEQ      XMIT                * yes, handle it

* Receiver interrupt handled here

   MOVE.B   BDATA74,D1          * get the character
   ANDI.B   #RXMASK,D1          * strip off high bit
   MOVE.L   #UIFRXCHR,D0        * UI_RXCHR function code
   TRAP     #VRTX               * call VRTX32
   TST.L    D0                  * any errors?
   BEQ      RTN                 * exit if none

   MOVE.L   #RXCHERR,D7         * UI_RXCHR error code
   TRAP     #BREAKPT            * back to monitor

* Transmitter interrupt handled here

XMIT:

   MOVE.B   #$28,ACNTRL74       * set disable channel A
   MOVE.B   #$28,BCNTRL74       * set disable channel B

   MOVE.L   #UIFTXRDY,D0        * UI_TXRDY function code
   TRAP     #VRTX               * call VRTX32
   TST.L    D0                  * character available?
   BNE      RTN                 * exit if no character
   JSR      VTXRDY              * call TXRDY driver routine
```

```
* Common return for transmit and receive.  Call UI_EXIT
* and return.

RTN:

   MOVE.L   (SP)+,D1           * restore D1
   MOVE.B   #EOI,BCNTRL74      * send EOI command chB
   MOVE.B   #EOI,ACNTRL74      * send EOI command chA
   MOVE.L   #UIFEXIT,D0        * UI_EXIT function code
   TRAP     #VRTX              * call VRTX32

*********************************************************************
*                                                                   *
*                     VRTX32 TXRDY Driver Routine                   *
*                                                                   *
*********************************************************************

   XDEF VTXRDY
VTXRDY:

   ANDI.W   #$F8FF,SR          * enable interrupts
   MOVE.B   D1,BDATA74         * output the character
   RTS                         * return to VRTX32

   END
```

D.2 Example Application Program

```
*******************************************************************
*                                                                 *
*                     THE APPLICATION PROGRAM                     *
*                                                                 *
*  FILE:    demo.asm                                              *
*  VERSION: 1.04                           DATE: APRIL 1987       *
*                                                                 *
*  COPYRIGHT 1987, HUNTER & READY, INC.                           *
*                                                                 *
*******************************************************************
*                                                                 *
*  Task MAIN clears the mailbox, lowers its own priority, creates *
*  TASK2, outputs a prompt character, then enters a loop to get a *
*  character from the terminal and post it to the mailbox.        *
*                                                                 *
*******************************************************************

BREAKPT     EQU   3             * trap vector for return to monitor

* Include VRTX32 definitions

   NOLIST
   INCLUDE 'vrtxvisi.inc'       * VRTX32 equates
   LIST

   SECTION 9

   XDEF MAIN
MAIN:

   CLR      BOX                 * clear the mailbox

* Task MAIN lowers its own priority to 2

   MOVE.L   #0,D1               * self
   MOVE.L   #2,D2               * pri = 2
   MOVE.L   #SCFTPRIORITY,D0    * SC_TPRIORITY function code
   TRAP     #VRTX               * call VRTX32
   CMPI.L   #RET_OK,D0          * check return code
   BEQ.S    TCREATE             * branch if no error
   MOVE.L   #SCFTPRIORITY,D7    * indicate which VRTX32 call
   JMP      ERRORX              * jump to error handler

* Task MAIN creates TASK2

TCREATE:

   MOVE.L   #1,D1               * pri = 1
   MOVE.L   #2,D2               * ID = 2
   MOVE.L   #0,D3               * user mode task
   MOVEA.L  #TASK2,A0           * address of task
   MOVE.L   #SCFTCREATE,D0      * SC_TCREATE function code
   TRAP     #VRTX               * call VRTX32
```

```
    CMPI.L    #RET_OK,D0          * check return code
    BEQ.S     PROMPT              * branch if no error
    MOVE.L    #SCFTCREATE,D7      * indicate which VRTX32 call
    JMP       ERRORX              * jump to error handler

* Task MAIN outputs the '>' prompt

PROMPT:

    MOVE.L    #$3E,D1             * char = '>'
    MOVE.L    #SCFPUTC,D0         * SC_PUTC function code
    TRAP      #VRTX               * call VRTX32
    CMPI.L    #RET_OK,D0          * check return code
    BEQ.S     GETC                * branch if no error
    MOVE.L    #SCFPUTC,D7         * indicate which VRTX32 call
    JMP       ERRORX              * jump to error handler

* Task MAIN gets a character from the CRT

GETC:

    MOVE.L    #SCFGETC,D0         * SC_GETC function code
    TRAP      #VRTX               * call VRTX32
    CMPI.L    #RET_OK,D0          * check return code
    BEQ.S     POST                * branch if no error
    MOVE.L    #SCFGETC,D7         * indicate which VRTX32 call
    JMP       ERRORX              * jump to error handler

* Character is now in register D1; post the character to the
* mailbox

POST:

    MOVEA.L   #BOX,A0             * address of mailbox
    MOVE.L    #SCFPOST,D0         * SC_POST function code
    TRAP      #VRTX               * call VRTX32
    CMPI.L    #RET_OK,D0          * check return code
    BEQ.S     GETC                * loop if no error, get next char
    MOVE.L    #SCFPOST,D7         * indicate which VRTX32 call
    JMP       ERRORX              * jump to error handler

*******************************************************************
*                                                                 *
*  Task2 picks up the character from the mailbox, then outputs    *
*  the character to the terminal.                                 *
*                                                                 *
*******************************************************************

TASK2:

* Task2 pends on mailbox for a character

PENDC:

    MOVE.L    #0,D1               * no timeout
```

```
   MOVEA.L  #BOX,A0            * address of mailbox
   MOVE.L   #SCFPEND,D0        * SC_PEND function code
   TRAP     #VRTX              * call VRTX32
   CMPI.L   #RET_OK,D0         * check return code
   BEQ.S    PUTC               * branch if no error
   MOVE.L   #SCFPEND,D7        * indicate which VRTX32 call
   JMP      ERRORX             * jump to error handler

* Character is now in D1; put the character to the CRT

PUTC:

   MOVE.L   #SCFPUTC,D0        * SC_PUTC function code
   TRAP     #VRTX              * call VRTX32
   CMPI.L   #RET_OK,D0         * check return code
   BEQ.S    PENDC              * loop if no error, get next char
   MOVE.L   #SCFPUTC,D7        * indicate which VRTX32 call
   JMP      ERRORX             * jump to error handler

*******************************************************************
*                                                                 *
*                         Error Handler                           *
*                                                                 *
*******************************************************************

ERRORX:

   TRAP     #BREAKPT           * back to monitor

   SECTION 14

*******************************************************************
*                                                                 *
*                           The Mailbox                           *
*                                                                 *
*******************************************************************

BOX:

   DS.L     1
END
```

D.3 VRTX32 Definitions File

```
VRTX       EQU   $00   * VRTX32 trap number

* VRTX32 task-level function codes

SCFTCREATE   EQU $0000   * create task
SCFTDELETE   EQU $0001   * delete task
SCFTSUSPEND  EQU $0002   * suspend task
SCFTRESUME   EQU $0003   * resume task
SCFTPRIORITY EQU $0004   * change task priority
SCFTINQUIRY  EQU $0005   * task status inquiry
SCFGBLOCK    EQU $0006   * get memory block
SCFRBLOCK    EQU $0007   * release memory block
SCFPOST      EQU $0008   * post message to mailbox
SCFPEND      EQU $0009   * pend for message from mailbox
SCFGTIME     EQU $000A   * get time
SCFSTIME     EQU $000B   * set time
SCFTDELAY    EQU $000C   * task delay
SCFGETC      EQU $000D   * get character
SCFPUTC      EQU $000E   * put character
SCFWAITC     EQU $000F   * wait for special character
SCFTSLICE    EQU $0015   * enable round-robin scheduling
SCFFCREATE   EQU $0017   * create event flag group
SCFFDELETE   EQU $0018   * delete event flag group
SCFFPEND     EQU $0019   * pend on event flag group
SCFFPOST     EQU $001A   * post event to event flag group
SCFFCLEAR    EQU $001B   * clear event
SCFFINQUIRY  EQU $001C   * event flag group inquiry
SCFQJAM      EQU $001E   * jam message to queue
SCFQECREATE  EQU $001F   * create FIFO message queue
SCFLOCK      EQU $0020   * disable task rescheduling
SCFUNLOCK    EQU $0021   * enable task rescheduling
SCFPCREATE   EQU $0022   * create memory partition
SCFPEXTEND   EQU $0023   * extend memory partition
SCFACCEPT    EQU $0025   * accept message from mailbox
SCFQPOST     EQU $0026   * post message to queue
SCFQPEND     EQU $0027   * pend for message from queue
SCFQACCEPT   EQU $0028   * accept message from queue
SCFQCREATE   EQU $0029   * create message queue
SCFQINQUIRY  EQU $002A   * queue status inquiry
SCFSCREATE   EQU $002B   * create semaphore
SCFSDELETE   EQU $002C   * delete semaphore
SCFSPEND     EQU $002D   * pend on semaphore
SCFSPOST     EQU $002E   * post unit to semaphore
SCFSINQUIRY  EQU $002F   * semaphore inquiry

* VRTX32 interrupt-level function codes

UIFEXIT    EQU $0011   * exit interrupt handler
UIFTIMER   EQU $0012   * announce timer interrupt
UIFRXCHR   EQU $0013   * received-character interrupt
UIFTXRDY   EQU $0014   * transmit-ready interrupt
UIFENTER   EQU $0016   * enter interrupt handler
```

```
* VRTX32 initialization function codes

VRTXFINIT  EQU $0030   * initialize VRTX32
VRTXFGO    EQU $0031   * start application execution

* System-wide error codes

RET_OK     EQU $0000   * successful return

* VRTX32 error codes

ER_TID     EQU $0001   * task ID error
ER_TCB     EQU $0002   * no TCBs available
ER_MEM     EQU $0003   * no memory available
ER_NMB     EQU $0004   * not a memory block
ER_MIU     EQU $0005   * mailbox in use
ER_ZMW     EQU $0006   * zero message
ER_BUF     EQU $0007   * buffer full
ER_WTC     EQU $0008   * WAITC in progress
ER_ISC     EQU $0009   * invalid system call
ER_TMO     EQU $000A   * timeout
ER_NMP     EQU $000B   * no message present
ER_QID     EQU $000C   * queue ID error
ER_QFL     EQU $000D   * queue full
ER_PID     EQU $000E   * partition ID error
ER_INI     EQU $000F   * fatal initialization error
ER_NCP     EQU $0010   * no character present
ER_ICP     EQU $0011   * invalid config parameter during INIT
ER_IIP     EQU $0012   * invalid input parameter, PCREATE/PEXTEND
ER_CVT     EQU $0020   * component vector table not present
ER_COM     EQU $0021   * undefined component
ER_OPC     EQU $0022   * undefined opcode for component
ER_NOCB    EQU $0030   * no control blocks available
ER_ID      EQU $0031   * event flag group or semaphore ID error
ER_PND     EQU $0032   * tasks pending on ev flag group/semaphore
ER_DEL     EQU $0033   * event flag group or semaphore is deleted
ER_OVF     EQU $0034   * event flag already set or semaphore overflow

* TCB -- Task Control Block

           OFFSET  0
TBRSV1     DS.L    2   * $00 reserved
TBEXT      DS.L    1   * $08 pointer to user's TCB extension
TBPRI      DS.B    1   * $0C priority
TBID       DS.B    1   * $0D task ID number
TBSTAT     DS.W    1   * $0E status
TBD0       DS.L    1   * $10 saved D0
TBD1       DS.L    1   * $14 saved D1
TBD2       DS.L    1   * $18 saved D2
TBD3       DS.L    1   * $1C saved D3
TBD4       DS.L    1   * $20 saved D4
TBD5       DS.L    1   * $24 saved D5
TBA0       DS.L    1   * $28 saved A0
TBA1       DS.L    1   * $2C saved A1
TBA2       DS.L    1   * $30 saved A2
```

```
TBA3        DS.L     1    * $34 saved A3
TBSSP       DS.L     1    * $38 supervisor-mode stack pointer
TBUSP       DS.L     1    * $3C user-mode stack pointer
TBSTACK     DS.L     1    * $40 reserved (original stack pointer)
TBFLAGS     DS.W     1    * $44 reserved
TBRSV2      DS.W     5    * $46 reserved
TCBB        EQU    $50    * total size in bytes

* TBSTAT task status flags

TBSSUSP     EQU $0001    * explicitly suspended
TBSSUSPT    EQU     0
TBSMBOX     EQU $0002    * suspended for mailbox message
TBSMBOXT    EQU     1
TBSGETC     EQU $0004    * suspended for character input
TBSGETCT    EQU     2
TBSPUTC     EQU $0008    * suspended for character output
TBSPUTCT    EQU     3
TBSWAITC    EQU $0010    * suspended for special character
TBSWAITCT   EQU     4
TBSDELAY    EQU $0020    * suspended for task delay/timeout
TBSDELAYT   EQU     5
TBSQUEUE    EQU $0040    * suspended for queue message
TBSQUEUET   EQU     6
TBSRSV1     EQU $0080    * reserved
TBSRSV1T    EQU     7
TBSIDLE     EQU $0100    * idle task
TBSIDLET    EQU     8
TBSFLAG     EQU $0200    * suspended for event
TBSFLAGT    EQU     9
TBSSEMA     EQU $0400    * suspended for semaphore unit
TBSSEMAT    EQU    10

* CFTBL -- Configuration Table

            OFFSET  0
CFWSADDR    DS.L    1    * $00 VRTX32 workspace pointer
CFWSSIZE    DS.L    1    * $04 VRTX32 workspace size
CFSSTKSZ    DS.W    1    * $08 supervisor-mode stack size per task
CFISTKSZ    DS.W    1    * $0A ISR stack size, if enabled
CFCBCOUNT   DS.W    1    * $0C ev flag & semaphore control block count
CFRSRVD1    DS.W    1    * $0E reserved, must be zero
CFIDLE      DS.W    1    * $10 idle task stack size
CFRSRVD2    DS.W    1    * $12 reserved, must be zero
CFDISLEV    DS.W    1    * $14 component disable level
CFUSTKSZ    DS.W    1    * $16 user-mode stack size per task
CFRSRVD3    DS.L    1    * $18 reserved, must be zero
CFUTSKCT    DS.W    1    * $1C user task count
CFRSRVD4    DS.W    1    * $1E reserved, must be zero
CFTXRDY     DS.L    1    * $20 TXRDY driver routine pointer
CFTCREATE   DS.L    1    * $24 TCREATE routine pointer
CFTDELETE   DS.L    1    * $28 TDELETE routine pointer
CFTSWITCH   DS.L    1    * $2C TSWITCH routine pointer
CFCVTADDR   DS.L    1    * $30 component vector table pointer
CFTBLB      EQU   $34    * total size in bytes
```

Appendix E

The Rescheduling Procedure

This manual uses the term **rescheduling procedure** to describe the process that evaluates task priorities and execution states. A **task switch** occurs when a ready task is found that has a higher priority than the currently executing task. A task switch entails saving the current task's state in its TCB and loading the registers with the context of the higher-priority ready task.

The rescheduling procedure occurs after some VRTX32 system calls immediately before they return to the task level. However, the rescheduling procedure does not always result in a task switch.

ISRs can interrupt the rescheduling procedure. VRTX32 restarts the rescheduling procedure every time it is interrupted and a VRTX32 call has been made; this ensures that the correct task gets control.

The optional Sys-TSWITCH-Address parameter in the configuration table points to the TSWITCH routine. When the rescheduling procedure results in a task switch, VRTX32 calls this routine. (Occasionally the rescheduling procedure does not result in a task switch; in this case, VRTX32 does not call the TSWITCH routine.) Because the rescheduling procedure is interruptible, VRTX32 can call the TSWITCH routine more than once before the next task is run.

Table E-1 lists VRTX32 calls that do not initiate the rescheduling procedure and so do not result in a task switch.

The SC_TDELAY and SC_WAITC calls always initiate the rescheduling procedure, which usually results in a task switch. If there are no ready user tasks, VRTX32 switches to the idle task.

The remaining calls initiate the rescheduling procedure under certain circumstances, usually when the call suspends or deletes the current task or readies another task.

For example, when you create a task with a higher priority than the current task, the rescheduling procedure results in a task switch. A suspension of the current task also results in a task switch. The following task management calls cause a task switch when they affect the current task or a task with a higher priority:

SC_TCREATE
SC_TDELETE
SC_TSUSPEND
SC_TRESUME
SC_TPRIORITY
SC_UNLOCK

These intertask communication and synchronization calls cause a task switch when the current task suspends on a mailbox, queue, event flag group, or semaphore; when a post call readies a higher-priority task; or when a delete call readies a higher-priority task.

SC_POST
SC_PEND
SC_QPOST
SC_QJAM
SC_QPEND
SC_FDELETE
SC_FPOST
SC_FPEND
SC_SDELETE
SC_SPOST
SC_SPEND

Two of the character I/O calls cause a task switch when they try to put a character to a full buffer or try to get a character from an empty buffer:

SC_GETC
SC_PUTC

ISRs that make VRTX32 system calls must end with the UI_EXIT call. If the ISR readies a higher-priority task than the interrupted task, a task switch occurs after the UI_EXIT call. For nested interrupts, the task switch occurs only when the initial interrupt handler completes and executes its UI_EXIT call.

The system calls below, when made from an ISR, can cause a task switch:

SC_POST
SC_QPOST
SC_QJAM
SC_FPOST
SC_SPOST
UI_TIMER
UI_RXCHR
UI_TXRDY

When considering rescheduling and possible task switches, you must understand the operation of system calls. Some calls, such as the post calls, complete their action and do not necessarily return immediately to the caller. This can happen when a higher-priority task pends on the target queue, mailbox, event flag group, or semaphore. The higher-priority task unpends and gains control, and VRTX32 does not return to the original caller until all higher-priority tasks complete, suspend, or become dormant.

Table E-1 Calls that Do Not Initiate the Rescheduling Procedure

Task Management

SC_TINQUIRY
SC_LOCK

Memory Allocation

SC_GBLOCK
SC_RBLOCK
SC_PCREATE
SC_PEXTEND

Communication and Synchronization

SC_ACCEPT
SC_QACCEPT
SC_QCREATE
SC_QECREATE
SC_QINQUIRY
SC_FCREATE
SC_FCLEAR
SC_FINQUIRY
SC_SCREATE
SC_SINQUIRY

Interrupt Support

UI_ENTER

Real-Time Clock

SC_GTIME
SC_STIME
SC_TSLICE

Initialization

VRTX_INIT
VRTX_GO

Index

HUNTER
READY
A Division of Ready Systems

We'd like your comments

Ready Systems attempts to provide documents that meet the needs of all our customers. We can improve our documentation if you help us by commenting on the usability, accuracy, readability, and organization of this manual. All comments and suggestions become the property of Ready Systems.

VRTX32/68000 User's Guide #541311001

1. Please specify by page any errors you found in this manual.

____

____

2. Is this document comprehensive enough? Please suggest any missing topics or information that is not covered.

____

____

3. Did you have any difficulty understanding this document? Please identify the unclear sections.

____

____

4. Please rate this document on a scale from 1 to 10, with 10 the best rating. ______________

Your Name ______________________________

Title ______________________________

Company Name ______________________________

Address ______________________________

City

READY SYSTEMS

Thanks for your help!

NO POSTAG
NECESSAR
IF MAILEC
IN THE
UNITED STAT

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1854 SUNNYVALE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

READY SYSTEMS
P.O. Box 60217
Sunnyvale, CA 94088-0217

READY
SYSTEMS