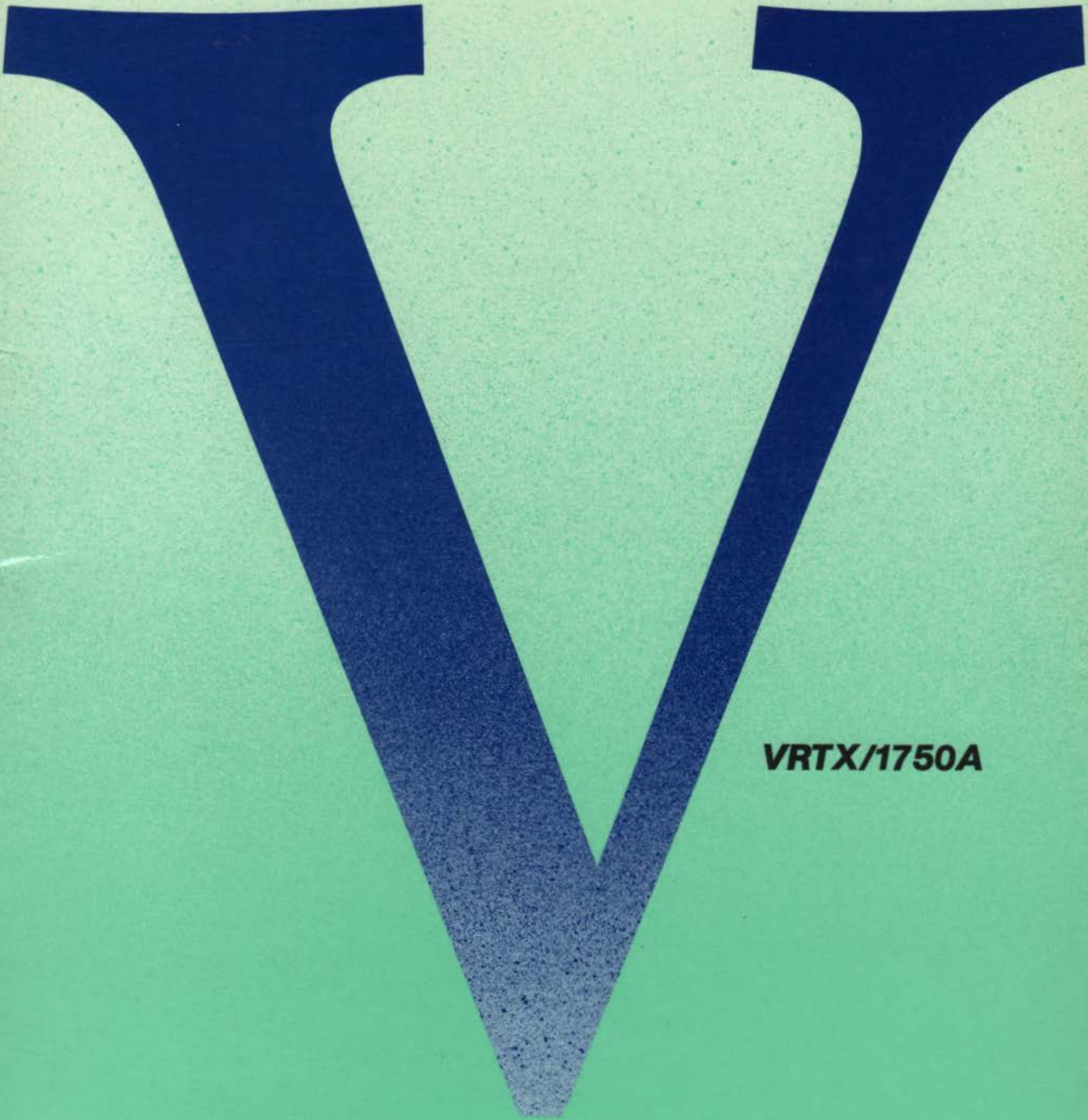


VRTX/1750A

HUNTER
READY

VRTX®/1750

Versatile Real-Time Executive
for the MIL-STD-1750A Computer

User's Guide

Version 3
Document Number 591613001
September 1984

REV.	REVISION HISTORY	PRINT DATE
— 001	Original Issue	9/84

Hunter & Ready, Inc. 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. Hunter & Ready, Inc. assumes no responsibility for any errors that may appear in this document. The information in this document is subject to change without notice.

Copyright © 1984
Hunter & Ready, Inc.
445 Sherman Avenue
P.O. Box 60803
Palo Alto, CA 94306-0803
415/326-2950
TELEX: 278835 (HRI UR)

All rights reserved
Printed in U.S.A.

Preface

VRTX is a silicon software component that provides operating system capabilities for an embedded microprocessor application. VRTX/1750 is designed for computers based on the MIL-STD-1750 Instruction Set Architecture.

This manual is intended for use by application programmers who are using VRTX/1750 to build a product. To fully implement the information discussed in this manual, the reader should also read *Getting Started with VRTX* and *How to Write a Board Support Package*. It is assumed that the reader has a complete understanding of the 1750 Instruction Set Architecture.

The first section of the manual presents an overview of the product discussing various features and defining terminology specific to VRTX and embedded applications. In the second chapter, descriptions of system calls begin with the basic system functions, task management, multitasking management, memory, memory allocation, intertask communication and synchronization and communication and synchronization calls. They are grouped by function so that all the calls relating to similar functions are grouped together. Chapter Three is devoted to interrupt support discussing the support available and the related system calls. Configuration and initialization are discussed in Chapter Four along with the initialization calls, VRTX_INIT and VRTX_GO. Chapter Five discusses support for user-defined extensions.

Table of Contents

HUNTER
READY

List of Illustrations

List of Tables

Chapter 1

OVERVIEW OF VRTX/1750

HUNTER
READY

1.1 Introduction

VRTX, the Versatile Real-Time Executive, is a silicon software component for embedded computers. VRTX/1750 is the implementation of VRTX designed for computers based on the MIL-STD-1750A (Notice 1) Instruction Set Architecture.[1] The following sections define terms basic to an understanding of VRTX/1750.

1.1.1 Silicon Software Components

A **silicon software component** is basically an executable version of a computer program that can operate within any environment that uses a particular type of computer. Because the source code of a silicon software component never needs to be modified to make it work with custom designs, a silicon software component 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 new concept introduced by silicon software components is the use of software as a building block, which can be connected to other pieces of software in a variety of designs, without ever being modified.

1.1.2 Embedded Applications

An **embedded** computer is a CPU buried inside some larger system, for example, inside an intelligent terminal, a communications system, an analytical instrument, an industrial robot, or a peripheral controller. Embedded computers are to be distinguished from **stand-alone** computers, such as small business systems, personal computers, and word processors.

The software that runs on embedded computers must meet a different set of requirements than software that runs on stand-alone systems. The most important of these is the need for **real-time** responsiveness. The system must be able to respond to unexpected events in the outside world rapidly enough to control some ongoing process. Another key requirement is **multitasking**, the ability of the software to handle a large number of tasks concurrently, since events in the real world usually overlap rather than occur in strict sequence.

[1] VRTX and VRTX/1750 are registered trademarks of Hunter & Ready, Inc. and may be used only in reference to Hunter & Ready products.

1.1.3 Real-Time Executive

Programmers have found that a common set of mechanisms is necessary to support real-time systems. Moreover, implementors of such systems spend more of their time on these basic mechanisms than on the application program itself. These mechanisms are usually contained in the operating system, which in embedded applications is called a **real-time operating system** or **real-time executive.** The real-time executive serves as the foundation upon which the rest of the application software is built. VRTX is a real-time executive.

1.2 VRTX Features

The features of VRTX/1750 can be divided into three categories: real-time features, silicon software component features, and features that specifically support the 1750A architecture.

1.2.1 Real-Time Features

VRTX provides a full range of real-time features, including:

* Multitasking support

* Interrupt-driven, priority-based scheduling

* Intertask communication and synchronization

* Dynamic memory allocation

* Real-time clock control, with optional time-slicing

* Real-time responsiveness

With these features, VRTX/1750 provides a strong foundation for real-time, multitasking application systems. It relieves designers and programmers of the problems of synchronizing multiple real-time tasks, thus allowing them to focus their efforts on the application itself.

1.2.2 Silicon Software Component Features

The silicon software component methodology used in the design of VRTX/1750 permits unparalleled flexibility of system design and implementation. The following key advantages should be noted:

* **Development environment independence.** Consisting entirely of one 2K word ROM component, VRTX configuration is not dependent on any particular assemblers, linkers, loaders or host environments.

* **Target environment independence.** Requiring only a CPU with a small amount of memory, VRTX provides support for the 1750A in a wide variety of embedded applications. No other system devices are required.

* **Extensibility.** Application-specific software is easily integrated with VRTX and its multitasking controller. This extended software can include user-defined system calls and device drivers.

* **Position independence.** VRTX is written entirely in position-independent code and can be positioned anywhere in the address space of the processor.

1.2.3 1750A Support Features

Several features of VRTX/1750 explicitly support features of the 1750A architecture:

* **Expanded Memory Addressing.** VRTX has been designed for complete compatibility with 1750A implementations that include the expanded memory addressing option. VRTX instruction and operand references use the privileged Address State 0 map; whereas individual tasks can be initiated with other, less privileged maps.

* **Memory Protection.** Integrity and security are further promoted in systems that include block-protection and/or access-locking features. Again, VRTX is entirely compatible with implementations that include these particular 1750A options.

* **Implementation Independence.** VRTX/1750 was designed to work with *all* computers based on MIL-STD-1750A, and is not dependent on the specific hardware details, operating characteristics, or range of options exhibited by any particular implementation.

1.3 VRTX Configuration

VRTX/1750 is specially designed to support 1750A processors in custom designs. Thus, the usual dependencies on particular environments (e.g., specific peripherals) have been removed. This support is provided by VRTX **object code**, which means that users with special requirements do not have to acquire and modify source code.

The user-supplied Configuration Table--along with simple, device-specific interrupt handlers--provides the interface between VRTX and its environment. Within this table, the user can specify all the parameters required by VRTX for a particular system environment.

The 1750A architecture reserves several low-address locations as a 'Vector Table,' providing linkage and service pointers for interrupt servicing. The data structure referenced by one of the vectors--the Interrupt 5 (Executive Call) service pointer--is used to route BEX executive calls to the VRTX entry location.

VRTX also reserves two additional low-address locations as pointers to data structures that it requires. The first of these, at location 0050 hex, points to the base of the Configuration Table. A second pointer, at location 0051 hex, points to a data area reserved for VRTX's private operand references.

Fields in the Configuration Table describe system-managed memory, multitasking controls, and the location of any user-supplied routines to be invoked by significant events, such as task-switching. Figure 1-1, VRTX Configuration, shows the relationships between VRTX, the Configuration Table and the reserved memory locations.

1.4 VRTX Architecture

A system based on VRTX is designed like a stack of bricks (see Figure 1-2, VRTX Architecture), with each level making use of the functions provided by the level below. The system's hardware occupies the bottom level, above which reside the simplest, most hardware-dependent operating system functions of software. On top are the user-defined application programs.

In more technical terminology, each level defines a **virtual machine** for the level above it. The functions provided by a software level are not distinguishable at higher levels from those provided by the hardware. Effectively, each software level adds several 'instructions' to the processor's instruction set. For application programs, VRTX adds a number of high-level instructions (the **system calls**) to the architecture of the 1750A.

The cross-hatched area in Figure 1-2, indicates the operating system mechanisms contained within the VRTX ROM. Notice that there are a few small pieces missing between VRTX and the hardware. These missing pieces are **interrupt service routines**, small hardware-dependent code segments that provide interrupt-handling for particular peripherals.

Other operating system mechanisms not provided within VRTX are shown to the right of VRTX in Figure 1-2. These include user-defined system call handlers and mechanisms for initializing and saving the state of special user devices (e.g., a Fourier transform co-processor in a signal processing application). Like interrupt service routines, these pieces are connected to VRTX via designed-in 'hooks' to form a unified operating system. Some of these hooks are made with certain entries in the Configuration Table (see Chapter 4, Configuration and Initialization, and Chapter 5, Support for User-Defined Extensions).

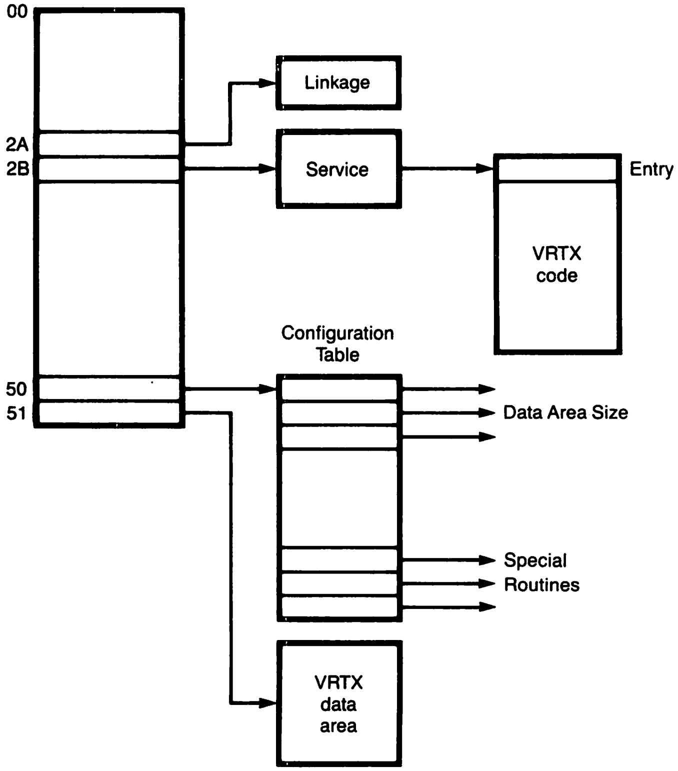


Figure 1-1. VRTX Configuration

The three horizontal bars shown below Figure 1-2 divide the overall system architecture into three vertical sections. Corresponding to these sections are three groups of VRTX mechanisms: the basic system call mechanisms, the mechanisms that support interrupts, and the mechanisms that support user-defined extensions. These groups are described in the chapters 2, 3 and 5. This document covers the assembly language format of VRTX/1750 calls. For a discussion of using VRTX/1750 with a high-level language, consult the*Jovial Interface Library User's Guide* (Document No. 592503).

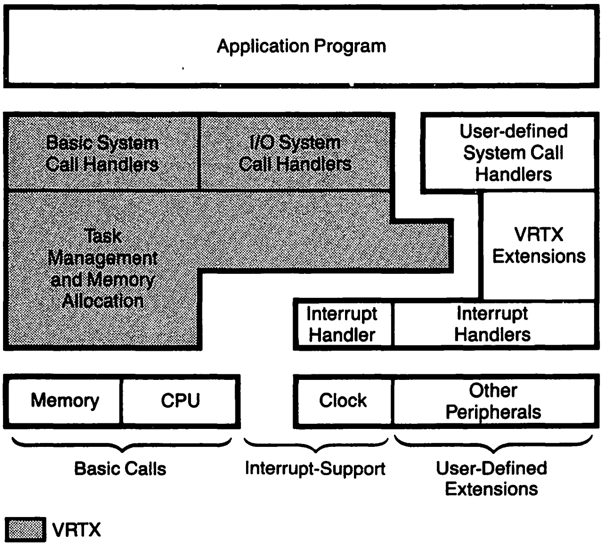


Figure 1-2. VRTX Architecture

Chapter 2

BASIC SYSTEM CALLS

HUNTER
READY

2.1 Introduction

This chapter describes basic operations that can be performed by VRTX (see the shaded portion of Figure 2-1, Basic Architecture). These functions are organized into three categories:

* Multitasking management

* Memory allocation

* Intertask communication and synchronization

2.1.1 System Call Format

Requests for VRTX services are made via the BEX instruction (Branch to Executive), which generates an Executive Call interrupt. Using MIL-STD mnemonics, the format of a call to VRTX is written as:

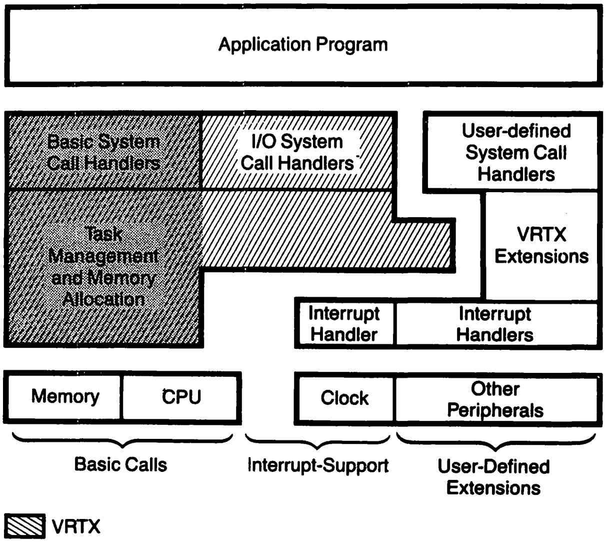


Figure 2-1. Basic Architecture

```
BEX n
```

Using IEEE mnemonics, the same call is written as:

```
BRK #n
```

In both cases, the value n specifies which of the 16 BEX executive entry points, numbered 0 through 15, has been designated for access to VRTX. BEX 0 is usually chosen. The remaining 15 entry points may be defined by the user (see Chapter 5, Support for User-Defined Extensions).

Along with the system call itself, a 16-bit function code is passed in register R0; this code specifies the requested VRTX service. Additional input parameters are passed to VRTX in registers R1 through R4. Some system calls return output results in these same registers; unless otherwise indicated, however, all input registers except R0 are left intact by a VRTX system call. Appendix A, System Call Summary, lists all VRTX calls along with their input parameters and output results.

When a call is completed, register R0 contains a 16-bit return code. If the call was successful, R0 returns a value of zero; otherwise, R0 returns one of the error codes listed in Appendix B, Return Codes.

Table 2.1 lists all VRTX system calls with their associated function codes (in hexadecimal format).

2.1.2 Executive Call Interrupt

The 1750A architecture uses a series of linkage and service pointers to control access to interrupt service routines. Whenever an interrupt is generated--either by external hardware or by execution of the current program instruction--a new machine state (consisting of a 'new interrupt mask,' 'new status word' and 'new instruction counter') is loaded from the status information block referenced by the appropriate service pointer. The new instruction counter (IC) points to the starting address of the service routine.

TABLE 2.1 VRTX SYSTEM CALLS

Mnemonic	Function Code	System Call
Task Management:		
SC_TCREATE	0000H	Task Create
SC_TDELETE	0001H	Task Delete
SC_TSUSPEND	0002H	Task Suspend
SC_TRESUME	0003H	Task Resume
SC_TPRIORITY	0004H	Task Priority Change
SC_TINQUIRY	0005H	Task Inquiry
SC_LOCK	0020H	Disable Task Rescheduling
SC_UNLOCK	0021H	Enable Task Rescheduling
Memory Allocation:		
SC_GBLOCK	0006H	Get Memory Block
SC_RBLOCK	0007H	Release Memory Block
SC_PCREATE	0022H	Create Memory Partition
SC_PEXTEND	0023H	Extend Memory Partition
Communication and Synchronization:		
SC_POST	0008H	Post Message
SC_PEND	0009H	Pend for Message
SC_ACCEPT	0025H	Accept Message
SC_QPOST	0026H	Post Message to Queue
SC_QPEND	0027H	Pend for Message from Queue
SC_QACCEPT	0028H	Accept Message from Queue
SC_QCREATE	0029H	Create Message Queue
SC_QINQUIRY	002AH	Queue Inquiry
Interrupt Support:		
UI_ENTER	0016H	Enter Interrupt Handler
UI_EXIT	0011H	Exit from Interrupt Handler
Real-Time Clock:		
SC_GTIME	000AH	Get Time
SC_STIME	000BH	Set Time
SC_TDELAY	000CH	Task Delay
SC_TSLICE	0015H	Enable Round-Robin Scheduling
UI_TIMER	0012H	Announce Timer Interrupt
Initialization:		
VRTX_INIT	—	Initialize VRTX
VRTX_GO	0031H	Start Multitasking

The BEX instruction generates an **Executive Call** interrupt (Interrupt 5), which references the pair of linkage and service pointers at hexadecimal addresses 002A and 002B. The status information block referenced by the Executive Call service pointer has the following format:

```
+---------------------+
|  New Interrupt Mask  |
|---------------------|
|  New Status Word     |
|---------------------|
|  New IC for BEX 0    |
|---------------------|
|  New IC for BEX 1    |
|---------------------|
|                     |
.                     .
.                     .
|                     |
|---------------------|
|  New IC for BEX 15   |
+---------------------+
```

In order to direct BEX 0 Executive Call traps to VRTX, the first IC value in the status block should be set to the beginning of VRTX. (Of course, if an entry point other than BEX 0 is chosen, then it is the 'New IC' corresponding to the chosen BEX that must be set; again, the value used to set the vector should be equal to the VRTX start address.)

The 'New Status Word' should be zero in order to grant VRTX a privileged processor state (PS=0), as well as to assure that VRTX operates with the appropriate address state (AS=0) and access key (AK=0) in implementations where these optional 1750A features are supported.

For example, if VRTX is positioned at physical address 1000 (hexadecimal), then the service pointer at location 002B points to a status information block of the following format, assuming that the BEX 0 vector has been chosen for accessing VRTX:

```
                         +---------+
New Interrupt Mask:      |  xxxx   |
                         |---------|
New Status Word:         |  0000   |
                         |---------|
New IC for BEX 0:        |  1000   |
                         |---------|
New IC for BEX 1:        |  yyyy   |
          .              +---------+
          .              .         .
          .              .         .
```

2.2 Tasks

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

VRTX is designed to support real-time systems by providing a set of basic mechanisms for implementing **multitasking**. The basic logical unit controlled by VRTX is the **task**, a logically complete execution path through user code that demands the use of system resources. In a multitask system several tasks appear to execute concurrently, although VRTX actually coordinates execution of the tasks in an interleaved fashion through very rapid reallocation of CPU time.

Under VRTX the program (or collection of programs) that defines the multitask environment can have as many as 255 logically distinct active tasks, each tagged with a unique identification number. (In addition, any number of untagged tasks can exist within the VRTX framework.) Each task performs a specified function asynchronously and in real time. Each task is assigned a priority level, and VRTX allocates control of the CPU to the highest priority task that is ready to execute. The kernel supports as many as 256 levels of priority with any number of active tasks at each level. Several tasks can operate asynchronously from a single piece of code, with each task assigned a priority and possibly an identification number. Tasks can create other tasks, and they can delete, suspend, and change the priority of themselves or of other tasks.

2.2.1 Task Priority and Scheduling

When a task is created, it may be given a unique identification (ID) number (from 1 to 255--ID number 0 indicates that no ID is assigned) and a priority level (from 0 to 255, with 0 being the highest priority). The ID number allows tasks to be readied, suspended or deleted on a selective basis. VRTX uses the priority level to implement its priority-based scheduling algorithm.

In multitask programs that run under VRTX, the first task is created by the user (see Chapter 4, Configuration and Initialization). Usually, this task creates other tasks in the system by issuing VRTX calls. Since this is the only task in the system, it receives control. As subsequent tasks are created, VRTX is called upon to initiate the **rescheduling procedure** and to put the highest priority task into execution. See Appendix E, Task Rescheduling, which describes rescheduling in detail.

VRTX is an **event-driven** operating system. This means that the user does not need to execute special system calls to accomplish task switching. VRTX maintains in execution the highest priority task capable of execution. This task continues to execute until one of the following events occurs:

* The task terminates its own operation.

* The task is suspended.

* A higher priority task is ready to execute.

The executing task can be **suspended** for a number of reasons, e.g., an explicit delay or a wait for a message. The task remains suspended until the required operation is completed, at which time it becomes available for continued execution.

Tasks that have been created, but are of lower priority than the executing task, are said to be **ready** to run. These lower priority ready tasks run when all higher priority tasks are completed or suspended. Ready tasks within the same priority level can receive CPU control on a time-sliced, round-robin basis if time-slicing has been explicitly enabled by the SC_TSLICE call. Any number of tasks can exist at the same priority level.

2.2.2 Task Control Block (TCB)

Due to the serial nature of a computer, tasks that appear to be executing in parallel are actually executing in short, interleaved segments. It is therefore necessary for VRTX to maintain status information (e.g., the contents of active registers) for all tasks that are not in control of the CPU. This information is retained in a data structure in system memory called the Task Control Block (TCB). One TCB is associated with each active task in the system. See Appendix C, IVT and TCB Formats, for a diagram of the TCB format.

A task is considered to be in an active state if it is executing, ready or suspended. If a task is in a dormant state (inactive), the system has no knowledge of its existence, even though its code remains in memory. (No TCB is defined for a dormant task.)

A task's TCB is frozen while the task is executing and is not altered until the task is completed or suspended, at which time the TCB is used to store status information about the task. The TCBs of ready and suspended tasks are linked together in order of decreasing task priority to form an active chain. Each TCB is connected to the next by a link word (see Figure 2-2, TCB Chain). VRTX executes the first ready task in this chain.

In the case where several tasks have the same priority, a TCB that is inserted into the chain is placed ahead of the TCBs of any task at that priority level. The task whose TCB was most recently inserted into the TCB chain is executed before any other tasks at the same priority level. Equal priority tasks are thus prioritized according to the chain modification history.

TCBs are inserted into the TCB chain as a result of task create calls, task priority change calls and time-slicing (see Section 3.3.5, Enable Round-Robin Scheduling). Among equal priority tasks, an optional round-robin scheduling can be enabled. At the end of a time-slice interval, the TCBs of tasks with the same priority level as the executing task are rotated. One of the other tasks then gets a chance to execute.

If a task is deleted, it becomes dormant and its TCB is not used. Unused TCBs are linked together to form an inactive chain of available TCBs. When VRTX is initialized, all the TCBs are located on the inactive chain. Except for the links themselves (TBNEXT in Figure C-2, in Appendix C, IVT and TCB Formats) and the initial stack pointer (TBSTACK), these TCBs are empty.

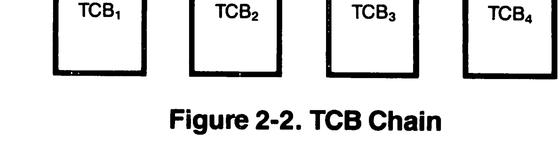


Figure 2-2. TCB Chain

Whenever a task is created, one of the TCBs is removed from the inactive chain and the interrupt mask save location is set to all ones (i.e., nothing masked). Parameters passed with the SC_TCREATE call determine the values for priority, task ID and Instruction Counter (TBPRI, TBID, TBIC). The location that contains the value for the new task (TBR14) is set to the TCB address value. The R15 Stack Pointer save location (TBSP) is set from TBSTACK. All remaining values in the TCB--including the saved values for the general registers (TBR0-TBR7, TBR8-TBR13) and for the SW special register (TBSW)--are copied from the caller's context.

> **Note:** When the first task is created, location TBSW (the stored value for the task's Status Word (SW) register) is reset. Thus, the first task runs with privileged processor state (PS=0) and address state (AS=0), unless these values in the TCB are overridden via the **sys-TCREATE-addr** hook described in Chapter 5, Support for User-Defined Extensions.

2.2.3 Task States and State Transitions

In a multitask environment, tasks exist in one of four states: **executing, ready** for execution, **suspended** or **dormant.**

Executing The task has control of the CPU and is executing its assigned instruction path.

Ready The task is ready for execution but cannot gain control of the CPU until (1) all higher priority tasks existing in the ready or executing state are either completed or suspended, and (2) it reaches the head of the queue of equal priority tasks.

Suspended The task has been suspended mid-execution and is waiting to be readied by a system call or an event (e.g., waiting for a certain number of ticks to expire).

Dormant The task has not been initiated, or its execution has been completed (i.e., task deleted) and it is now idle. No TCB is assigned to it.

A task may become suspended for any of the following reasons:

- * A Task Suspend call, SC_TSUSPEND, was issued specifying that task (either by priority or ID number).

* The task suspended itself for a specified time delay using the SC_TDELAY call.

* The task is waiting for a message from another task or interrupt handler (i.e., it issued an SC_PEND or SC_QPEND call but no message is posted yet).

The status word in each task's Task Control Block can be interrogated with the SC_TINQUIRY call. If the word returned is zero, the task is ready; if the word is nonzero, the task has been suspended for reasons indicated by the bit settings (see Appendix C, IVT and TCB Formats). Suspensions are independent and additive; for example, if 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 readied for execution.

WARNING

> Under no circumstances should a call be issued which can lead to the current task's suspension, if task switching has been disabled via an SC_LOCK call.

Just as a number of different events may suspend a task, several events and calls can place a suspended task back in the ready state.

* An SC_TRESUME call can be issued to ready a task that was suspended by an SC_TSUSPEND call.

* A time delay can expire, thus readying a task that was either suspended by an SC_TDELAY call or timed-out pending for a message.

* A message can be posted (via an SC_POST or SC_QPOST call) to a task that is waiting for a message.

Tasks are in the dormant state before they are created; they reenter the dormant state when they are deleted with an SC_TDELETE call. If all tasks are terminated, the entire system is placed in an idle state, essentially halting its activity, although the system remains capable of responding to interrupts. All task state transitions are shown in Figure 2-3, Task State Transitions.

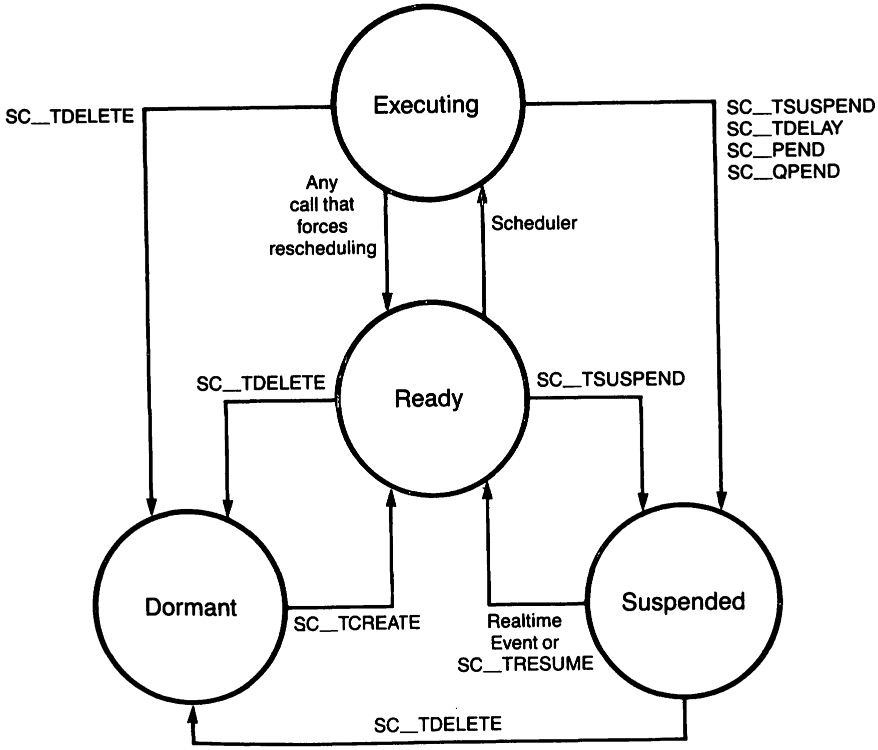


Figure 2-3. Task State Transitions

2.2.4 Configuration

The following entries in the Configuration Table are used to control multitasking operation.

user-stack-size specifies the amount of stack allocated to each task.

sys-TCREATE-addr, sys-TDELETE-addr, sys-TSWITCH-addr are extensions the user may utilize to control critical multitasking operations.

user-task-count specifies the maximum number of tasks allowed in the system.

Refer to Chapter 4, Configuration and Initialization, for a diagram of the Configuration Table and the location of these entries.

2.3 Multitasking Management Calls

The following calls allow the user to control the multitasking environment. All calls described in this section can lead to rescheduling which may result in a task switch, except SC_LOCK and SC_TINQUIRY.

2.3.1 SC_TCREATE - Create a Task

This call dynamically creates a task with a specified priority and ID number. Up to 256 priority levels may be specified; up to 255 unique ID numbers may be assigned. (Number 0 indicates that no ID is assigned.) The TCB of the newly created task is placed on the active chain immediately in front of the TCBs of all other tasks with the same priority.

The newly created task inherits all registers' contents, except R14 and R15, from the creator task's context. (R15 is the stack pointer for the task; R14 contains the new task's TCB address.) The newly created task also inherits the TCB extension pointer from the creating task. These values are stored in the newly created task's TCB and are available for the **sys-TCREATE-addr** user extension. This call results in a task switch if the new task's priority is higher than or equal to that of the calling task.

> **Note:** A task ID of 0, while legal, is a special case. A task with ID of 0 can be created but cannot be referenced by other tasks. The function calls SC_TDELETE, SC_TSUSPEND, SC_TRESUME, SC_TPRIORITY, and SC_TINQUIRY cannot reference a task with an ID of 0 if issued by a different task. These calls can be made for a task with ID of 0 only by the task itself. This anonymity is sometimes useful in systems for security applications.

```
+--------------------------------------------------------+
|                                                        |
|   INPUT:          R0 = SC_TCREATE (0000H)              |
|                                                        |
|             R1[0:7]  = priority for new task          |
|                                                        |
|             R1[8:15] = ID number for new task         |
|                                                        |
|                  R2 = address of new task             |
|                                                        |
|                                                        |
|   OUTPUT:         R0 = return code                     |
|                                                        |
+--------------------------------------------------------+
```

priority for new task is between 0 and 255, inclusive.

ID number for new task is between 1 and 255, inclusive, or 0 if no ID is to be assigned.

address of new task is the starting address of the user code for the new task.

RETURN CODES

0000H	RET_OK	Successful return.
0001H	ER_TID	Task ID error (ID number already assigned).
0002H	ER_TCB	No TCBs available.

2.3.2 SC_TDELETE - Delete a Task

This call removes one or more tasks from the active chain (including, possibly, the calling task itself). The affected task becomes dormant and its TCB is put on the inactive chain. This call results in a task switch if the current task is deleted.

```
+-----------------------------------------------------------+
|                                                           |
|    INPUT:          RO = SC_TDELETE   (0001H)               |
|                                                           |
|                    R1 = priority_or_ID                    |
|                                                           |
|                                                           |
|    OUTPUT:         RO = return code                       |
|                                                           |
+-----------------------------------------------------------+
```

priority_or_ID parameter in R1 can be formatted in any of three ways:

```
FORMAT 1:   Delete all tasks of a specified priority.
                R1[0: 7] = 'A' (ASCII character 'A')
                R1[8:15] = priority
            The return code will have the value RET_OK even
            if there are no tasks with the specified priority.

FORMAT 2:   Delete a task with a specified ID number.
                R1[0: 7] = 0
                R1[8:15] = task ID

FORMAT 3:   Delete self (i.e., delete calling task).
                R1 = 0
```

RETURN CODES

0000H	RET_OK	Successful return.
0001H	ER_TID	Task ID error (no task with specified ID number, Format 2 only).

2.3.3 SC_TSUSPEND - Task Suspend

This call suspends one or more tasks. The TCB of each affected task remains on the active chain, but bit 15 of the status word is set. A task suspended in this manner does not resume execution until an SC_TRESUME call is issued. This call results in a task switch if the current task is suspended.

> **Note:** If a task has disabled interrupts and subsequently becomes suspended, VRTX reenables interrupts for itself and other tasks. However, as soon as this task resumes execution, interrupts are again disabled.

```
+--------------------------------------------------------+
|                                                        |
|   INPUT:          R0 = SC_TSUSPEND  (0002H)            |
|                                                        |
|                   R1 = priority_or_ID                  |
|                                                        |
|                                                        |
|   OUTPUT:         R0 = return code                     |
|                                                        |
+--------------------------------------------------------+
```

priority_or_ID parameter in R1 can be formatted in any of three ways:

```
FORMAT 1:   Suspend all tasks of a specified priority.
                    R1[0: 7] = 'A' (ASCII character 'A')
                    R1[8:15] = priority
            The return code will have the value RET_OK even
            if there are no tasks with the specified priority.

FORMAT 2:   Suspend a task with a specified ID number.
                    R1[0: 7] = 0
                    R1[8:15] = task ID

FORMAT 3:   Suspend self (i.e., suspend calling task).
                    R1 = 0
```

RETURN CODES

0000H	RET_OK	Successful return.
0001H	ER_TID	Task ID error (no task with specified ID number, Format 2 only).

2.3.4 SC_TRESUME - Task Resume

This call resumes the execution of one or more tasks previously suspended by an SC_TSUSPEND call. This call results in a task switch if the resumed task has a higher priority, or if its priority is equal to that of the calling task but its TCB is ahead of the others on the active chain.

> **Note:** A task with ID equal to 0 cannot be explicitly resumed as an ID of 0 cannot be specified in this call.

```
+-------------------------------------------------------------+
|                                                             |
|   INPUT:            R0 = SC_TRESUME   (0003H)               |
|                                                             |
|                     R1 = priority_or_ID                     |
|                                                             |
|                                                             |
|   OUTPUT:           R0 = return code                        |
|                                                             |
+-------------------------------------------------------------+
```

priority_or_ID parameter in R1 can be formatted in two ways:

```
FORMAT 1:   Resume all tasks of a specified priority.
                    R1[0: 7] = 'A' (ASCII character 'A')
                    R1[8:15] = priority
            The return code will have the value RET_OK even
            if there are no tasks with the specified priority.

FORMAT 2:   Resume a task with a specified ID number.
                    R1[0: 7] = 0
                    R1[8:15] = task ID
```

RETURN CODES

0000H	RET_OK	Successful return.
0001H	ER_TID	Task ID error (no task with specified ID number, Format 2 only).

2.3.5 SC_TPRIORITY - Task Priority Change

This call changes the priority of a task. The TCB of the affected task is placed on the active chain immediately in front of the TCBs of all other tasks with its new priority. This call results in a task switch if the new priority of the affected task is higher than or equal to that of the calling task.

```
+---------------------------------------------------------+
|                                                         |
|    INPUT:           R0 = SC_TPRIORITY (0004H)           |
|                                                         |
|                     R1 = task ID                        |
|                                                         |
|                     R2 = priority                       |
|                                                         |
|                                                         |
|    OUTPUT:          R0 = return code                    |
|                                                         |
+---------------------------------------------------------+
```

task ID parameter in R1 and **priority** parameter in R2 can be formatted in two ways:

```
FORMAT 1:   Change the priority of a task with a specified
               ID number.
                      R1[0: 7] = 0
                      R1[8:15] = task ID

                      R2[0: 7] = 0
                      R2[8:15] = new priority

FORMAT 2:   Change the priority of the calling task.
                      R1 = 0

                      R2[0: 7] = 0
                      R2[8:15] = new priority
```

RETURN CODES

0000H	RET_OK	Successful return.
0001H	ER_TID	Task ID error (no task with specified ID number, Format 1 only).

2.3.6 SC_TINQUIRY - Task Inquiry

This system call is used to obtain priority and status information about a particular task (specified by ID number along with a pointer to the task's TCB. Alternatively, the SC_TINQUIRY call can be used by a task to determine its own task ID number and priority level, and the TCB's address. This call never goes through the rescheduling procedure and thus never results in a task switch.

This call can be made from a task or an interrupt service routine (ISR). If the call is made from an ISR, and no task ID is specified (i.e., task ID=0), the information describes the current interrupted task.

```
+------------------------------------------------------------+
|                                                            |
|    INPUT:          R0 = SC_TINQUIRY  (0005H)               |
|                                                            |
|                    R1 = task ID                            |
|                                                            |
|                                                            |
|    OUTPUT:         R0 = return code                        |
|                                                            |
|                    R1 = task ID                            |
|                                                            |
|                    R2 = priority                           |
|                                                            |
|                    R3 = status word                        |
|                                                            |
|                    R4 = TCB address                        |
|                                                            |
+------------------------------------------------------------+
```

task ID parameter in R1 can be formatted in two ways:

```
FORMAT 1:   Get information about a task with a specified
            ID number.
                    R1[0: 7] = 0
                    R1[8:15] = task ID

FORMAT 2:   Get information about one's self (i.e.,
            the calling task).
                    R1[0: 7] = 0
                    R1[8:15] = 0
```

RETURN CODES

0000H	RET_OK	Successful return.
0001H	ER_TID	Task ID error (no task with specified ID number, Format 1 only).

Note that if the return code is nonzero, all registers other than R0 remain unmodified. The status word returned by register R3 is the same as that at TCB offset 05. If the value of this word is zero, the associated task is ready to run. If the status word is nonzero, the task has been suspended for one or more of the following reasons, as indicated by the bit settings:

```
bit: 0                    8  9 10 11 12 13 14 15
     +-----------------------------------------+
R3 = |         0          |       status       |
     +-----------------------------------------+
```

Bit	Reason for Suspension	Suspending Call
15	Explicitly suspended	SC_TSUSPEND
14	Suspended for message	SC_PEND
10	Suspended for task delay	SC_TDELAY*
9	Suspended on message queue	SC_QPEND

*Also set for SC_PEND and SC_QPEND when a time-out is in effect.

For further information on TCBs refer to Appendix C, IVT and TCB Formats.

2.3.7 SC_LOCK - Disable Task Rescheduling

This call prevents task rescheduling until an explicit SC_UNLOCK call is issued. The task that issues SC_LOCK retains processor control, even though other higher priority tasks may become ready to run.

SC_LOCK and SC_UNLOCK are used in pairs. A count of locks and unlocks is kept so that nested instances of these calls do not prematurely end a scheduling lock. For example, nested subroutines and procedures may need to run locked for short periods of time. When they issue an SC_UNLOCK, it cancels the effect of the previous SC_LOCK *only*. This nesting count supports up to 255 levels of SC_LOCKs and SC_UNLOCKs.

This call should be used with caution, since it disrupts the ordinary management of the multitask environment. Interrupt-handling, however, is unaffected by disabled rescheduling.

WARNING

After this call has been issued, care should be taken not to issue any of the VRTX calls that can lead to the current task becoming suspended. This event is treated abnormally and causes unpredictable results.

```
+-------------------------------------------------------+
|                                                       |
|    INPUT:         R0 = SC_LOCK (0020H)                |
|                                                       |
|                                                       |
|    OUTPUT:        R0 = return code                    |
|                                                       |
+-------------------------------------------------------+
```

RETURN CODES

0000H	RET_OK	Successful return.

2.3.8 SC_UNLOCK - Enable Task Rescheduling

This call reenables normal VRTX task rescheduling, cancelling the effect of a single previously issued SC_LOCK call. If rescheduling is already enabled, this call has no effect. This call always goes through the rescheduling procedure which may result in a task switch.

A count of lock/unlock nesting is maintained by VRTX; refer to the SC_LOCK function for more information.

```
+------------------------------------------------------+
|                                                      |
|    INPUT:          R0 = SC_UNLOCK (0021H)            |
|                                                      |
|                                                      |
|    OUTPUT:         R0 = return code                  |
|                                                      |
+------------------------------------------------------+
```

RETURN CODES

0000H	RET_OK	Successful return.

2.4 Memory

The 1750A's 16-bit Instruction Counter defines an address space of 64K words. With the expanded memory addressing option, however, physical memory can be expanded up to a maximum of 2M words, organized as 16 pairs of separate 64K instruction and operand spaces. VRTX is entirely compatible with implementations that include the memory mapping feature, but this feature is not required.

In a system that makes use of the mapping option, memory is organized in the fashion shown in Figure 2-4, Separation of 1750A Address Spaces.

Conceptually, the memory of a VRTX-based system consists of the following modules:

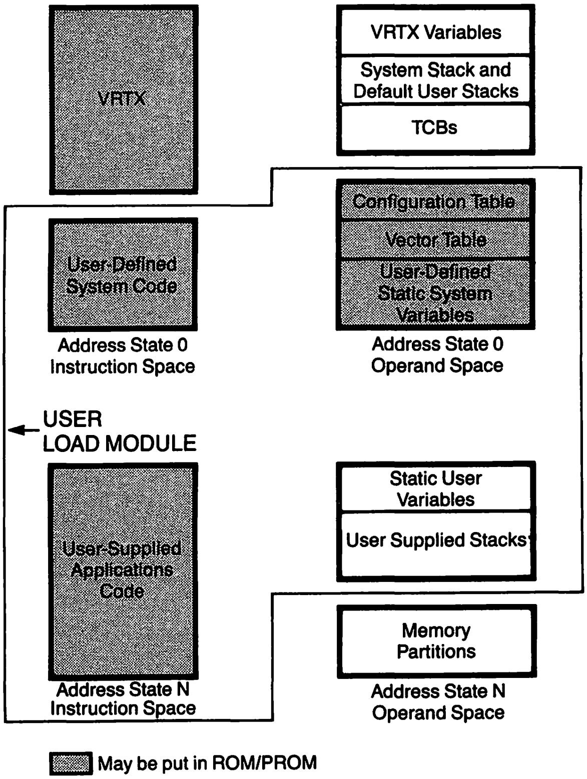

VRTX
VRTX Variables
System Stack and Default User Stacks
TCBs
User-Defined System Code
Configuration Table
Vector Table
User-Defined Static System Variables
Address State 0 Instruction Space
Address State 0 Operand Space
USER LOAD MODULE
User-Supplied Applications Code
Static User Variables
User Supplied Stacks
Memory Partitions
Address State N Instruction Space
Address State N Operand Space
▨ May be put in ROM/PROM

Figure 2-4. Separation of 1750A Address Spaces

- **VRTX code**, i.e., the VRTX PROM set.
- **The user load module**, i.e., the software package that the user is responsible for developing, assembling, linking and placing in the execution environment.
- **The VRTX Workspace**, which contains system variables, TCBs and stacks.
- **VRTX-managed user memory**, comprising one or more partitions, that is, pools of memory blocks that can be dynamically acquired and released by the user.

Figure 2-5, Memory Organization, is a schematic view of the overall memory organization of a VRTX system.

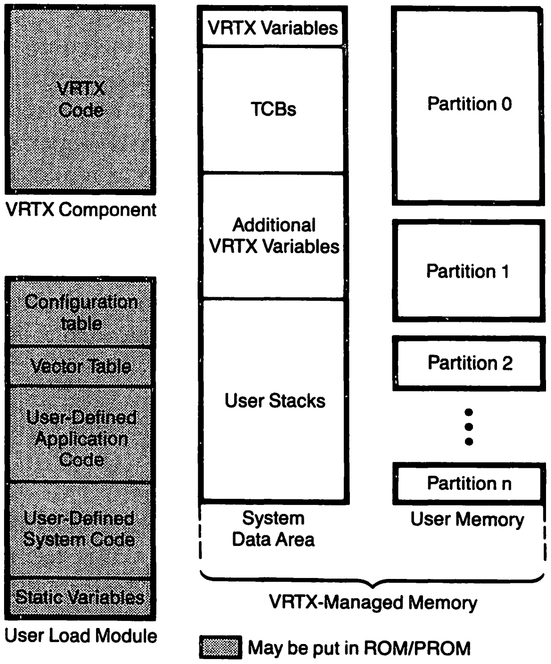


Figure 2-5. Memory Organization

The user load module holds the user's application code and any user-defined system-level code (see Chapter 3, Interrupt Support, and Chapter 5, Support for User-Defined Extensions). In addition, the load module contains the interrupt vectoring data (i.e., linkage and service pointers, as well as the status information blocks referenced by these pointers), the Configuration Table, and any status variables associated with the user application or with system code (see Section 2.6.1, Mailboxes). The shading in Figure 2-4 indicates what can be burned into ROM; everything else must exist in dynamic read/write memory.

The VRTX Workspace contains the VRTX variables, the TCBs, a system stack, a stack for each task in the system, and message queues. VRTX is responsible for setting up and managing the stacks and for initializing and managing the TCB chain. The user memory managed by VRTX consists of a number of partitions (i.e., chunks of memory which may be discontiguous). Each partition is subdivided into several fixed-size blocks of memory that can be allocated dynamically to tasks. The following section describes how VRTX manages user memory and its own VRTX Workspace.

2.4.1 Memory Allocation

A task's demand for memory varies over the course of its execution, and different tasks usually have different requirements. Consequently, a memory allocation policy must be established, and mechanisms in the operating system must exist to implement that policy. 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.

Two main approaches to memory allocation have been used by multitasking executives in the past: static allocation of fixed-size memory blocks and dynamic allocation of variable-sized 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-sized memory blocks, available memory eventually becomes fragmented as tasks allocate and release memory blocks from the available pool.

One technique for allocating variable-sized blocks is the 'buddy' system, which is widely used in non-real-time systems. In this technique, the system attempts to match the size of the allocated memory block to the size of the requested unit by repeatedly splitting existing units in half (starting with one single chunk of memory). When the request is for a unit larger than any of those currently available, the system attempts to combine a smaller unit with its twin (or 'buddy') into a large compact unit. This scheme suffers from a serious flaw for real-time applications: indeterminacy.

As memory grows progressively more fragmented, occasions inevitably arise when a request cannot be met. Even though there is enough total free memory, it may be so fragmented that a large enough contiguous block cannot be found. These occasions cannot be predicted in advance and compensated for, since they do not depend on the number of memory requests (which can be anticipated), but rather on the order of the requests (which usually cannot be anticipated). The design of this kind of operating system introduces an element of unpredictability into the total system behavior, above and beyond the unpredictability of the environment. This sort of additional unpredictability is usually unsatisfactory in real-time systems. Real-time systems cannot tolerate a memory system that *usually* works.

The designers of VRTX felt that static allocation was too restrictive, but that variable-sized blocks imposed too high a system overhead. Thus, the VRTX memory allocation mechanism is a compromise between these two memory allocation schemes. VRTX gives every task a fixed-size stack in system memory and dynamically allocates partitions of user memory in blocks. Users are able to dynamically create memory partitions to mirror the often discontiguous chunks that make up the actual physical organization of memory. Each partition of user memory has blocks of a fixed size, which is set when that partition is created. The **user-stack-size** is set by the user via a parameter in the Configuration Table.

VRTX dynamic memory allocation memory uses the following process:

1. At system initialization, parameters in the Configuration Table indicate the starting address and the size of the VRTX Workspace, how many tasks can exist at any one time, and how large each task's user stack should be. The VRTX Workspace must be large enough to contain the VRTX system stack, the VRTX system variables, one TCB for each active task, a stack for every task in the system, and additional memory for every allocatable block of user memory. In addition, the VRTX Workspace must be large enough to accommodate a control block for each memory partition and a control block for each defined message queue. For details on the calculation of the VRTX Workspace requirements, consult Chapter 4, Configuration and Initialization.

2. Whenever a task is created, VRTX automatically allocates a stack to the task. This stack can be used to store local variables and is allocated in the VRTX Workspace. In mapped systems, this default stack allocation should

be overridden, so as to provide each task with a stack in the operand space of its own Address State. (See Appendix D, Implementation Notes, for details.)

3. A task can execute the SC_PCREATE call to create a partition of user memory. Parameters passed with this call specify the starting address, size and standard block size of the partition. The calls SC_GBLOCK and SC_RBLOCK can then be used to acquire and release blocks of memory from the new partition.

4. The call SC_PEXTEND can be used to enlarge a previously defined partition to include an additional range of memory locations. The extension need not be contiguous with the originally defined partition.

5. A task can execute the call SC_GBLOCK to obtain a memory block from the pool of unallocated partition blocks of a partition previously created by the user. A task can execute this call repeatedly until all blocks in this partition are allocated.

6. A task can execute the SC_RBLOCK call to release a block of memory back to the partition previously created by the user. If a task is deleted, its blocks are not automatically released, so its blocks should be released before deleting the task.

Since all memory blocks within a partition are the same size, no fragmentation results from dynamic memory allocation; consequently, no memory compaction is required. Figure 2-6, VRTX Workspace and Figure 2-7, User Memory Managed by VRTX, show how memory is subdivided.

The VRTX partition/block system has several key features that give it extraordinary flexibility and most of the advantages of a variable-sized block scheme, without the indeterminacy and system overhead. First, partitions can be defined within other partitions. For example, one partition may be entirely within a single block of another partition; thus, blocks can easily be broken down into sub-blocks. Second, two partitions with differently sized blocks can be defined to cover the same area of memory; thus, blocks of different sizes can be allocated from the same memory region (the only requirement is that all blocks of one size be released before any blocks of the other size are allocated).

2.4.2 Configuration

The following parameters in the Configuration Table are used to set up memory allocation. Refer to Chapter 4, Configuration and Initialization, for more information.

VRTX-workspace-addr and **VRTX-workspace-size** assist in overall VRTX memory allocation.

user-stack-size and **user-task-count** are used to allocate application-related control structures.

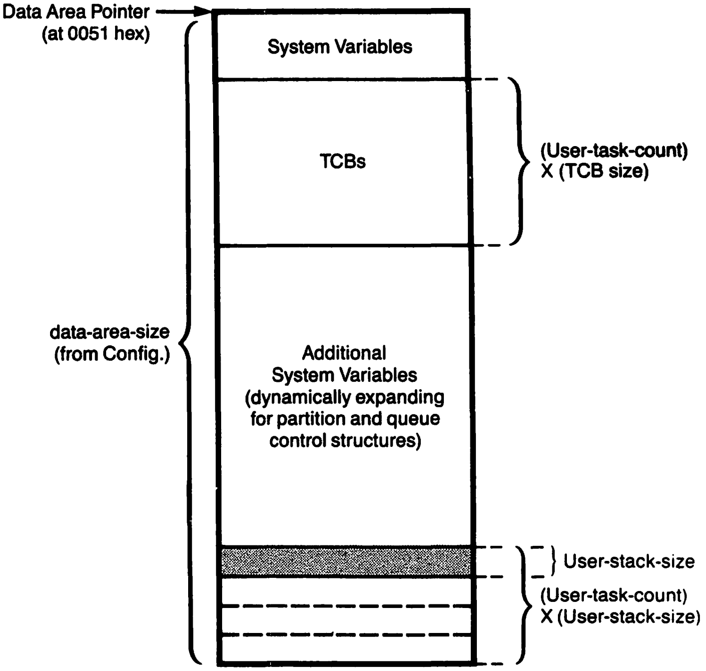


Figure 2-6. VRTX Workspace

2.5 Memory Allocation Calls

Two memory allocation calls (Get Memory Block and Release Memory Block) allow user tasks to obtain from a specified partition--and subsequently to release back to that partition--individual blocks of memory. All the blocks that comprise a partition are of fixed size, as determined by the SC_PCREATE call. None of the memory allocation calls go through the rescheduling procedure, therefore none of them result in a task switch.

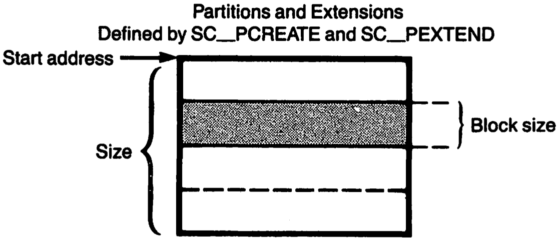

Partitions and Extensions
Defined by SC__PCREATE and SC__PEXTEND
Start address
Size
Block size

Figure 2-7. User Memory Managed by VRTX

2.5.1 SC_GBLOCK - Get Memory Block

This call obtains a memory block from one of the pools (or ‘partitions’) of memory blocks managed by VRTX.

```
+--------------------------------------------------------+
|                                                        |
|    INPUT:          R0 = SC_GBLOCK (0006H)              |
|                                                        |
|                    R2 = partition ID number            |
|                                                        |
|                                                        |
|    OUTPUT:         R0 = return code                    |
|                                                        |
|                    R1 = address of memory block        |
|                                                        |
+--------------------------------------------------------+
```

partition ID number is 16 bits

RETURN CODES

0000H	RET_OK	Successful return.
0003H	ER_MEM	No memory blocks available.
000EH	ER_PID	Partition ID error (no such partition).

2.5.2 SC_RBLOCK - Release Memory Block

This call returns a previously allocated memory block to the partition from which it was originally allocated. Blocks are not automatically released when the task is deleted.

```
+-----------------------------------------------------------+
|                                                           |
|    INPUT:          RO = SC_RBLOCK  (0007H)                |
|                                                           |
|                    Rl = address of memory block           |
|                                                           |
|                    R2 = partition ID number               |
|                                                           |
|                                                           |
|    OUTPUT:         RO = return code                       |
|                                                           |
+-----------------------------------------------------------+
```

partition ID number is 16 bits.

RETURN CODES

OOOOH	RET_OK	Successful return.
OOO4H	ER_NMB	Not a memory block (specified address does not reference a block previously allocated from the specified partition).
OOOEH	ER_PID	Partition ID error (no such partition).

2.5.3 SC_PCREATE - Create Memory Partition

This call is used to define the characteristics of a memory partition that is to be managed by the VRTX kernel. Associated with each such partition is an ID number and a default block size; successive SC_GBLOCK requests use this ID number to obtain blocks of memory of default size from the memory partition.

```
+-----------------------------------------------------------+
|                                                           |
|    INPUT:          R0 = SC_PCREATE (0022H)                |
|                                                           |
|                    R1 = partition ID number              |
|                                                           |
|                    R2 = partition start address          |
|                                                           |
|                    R3 = partition size                   |
|                                                           |
|                    R4 = block size                       |
|                                                           |
|                                                           |
|    OUTPUT:         R0 = return code                       |
|                                                           |
+-----------------------------------------------------------+
```

partition ID number is 16 bits.

partition size is the total size of the partition specified in words.

block size is specified in words.

RETURN CODES

0000H	RET_OK	Successful return.
0003H	ER_MEM	No memory available (insufficient system memory for VRTX control structures).
000EH	ER_PID	Partition ID error (ID number already assigned).

2.5.4 SC_PEXTEND - Extend Memory Partition

This command extends a previously defined memory partition to encompass an additional range of memory locations. In conjunction with SC_PCREATE, this command defines memory partitions that span multiple discontiguous ranges within an address space.

The block size for a partition extension is identical to that originally defined by SC_PCREATE.

```
+----------------------------------------------------+
|                                                    |
|   INPUT:          R0 = SC_PEXTEND  (0023H)         |
|                                                    |
|                   R1 = partition ID number         |
|                                                    |
|                   R2 = extension start address     |
|                                                    |
|                   R3 = extension size              |
|                                                    |
|                                                    |
|   OUTPUT:         R0 = return code                 |
|                                                    |
+----------------------------------------------------+
```

partition ID number is 16 bits.

extension size is the total size of the extension specified in words.

RETURN CODES

0000H	RET_OK	Successful return.
0003H	ER_MEM	No memory available (insufficient system memory for VRTX control structures).
000EH	ER_PID	Partition ID error (no such partition).

2.6 Intertask Communication and Synchronization

Even though tasks operate asynchronously, it is often desirable for one task to talk to another task. In VRTX, tasks communicate with one another by sending and receiving two-word (32-bit), nonzero messages via VRTX-controlled structures known as **mailboxes** and **queues**. These messages can, of course, be pointers to larger messages if the communicating tasks are designed to use such a technique.

2.6.1 Mailboxes

Synchronization and communication between multiple tasks in a VRTX system can be accomplished with three simple, yet powerful, commands:

SC_POST	Post a Message
SC_PEND	Pend for Message
SC_ACCEPT	Accept a Message

A transmitting task deposits the message in an agreed-upon location using the SC_POST call. To receive the message, another task issues an SC_PEND call. If the message has already been sent, the receiving task accepts it and is placed in the ready state. The message location is reset to zero when the message is received. If a location is empty (holds no message), a task attempting to receive a message with an SC_PEND call suspends execution until a message arrives. Conversely, if a location is full (message is present), a task attempting to send a message with SC_POST continues execution, but an error code is returned. A task using SC_ACCEPT to receive does not suspend if no message is present; instead an error return is taken. (A message location must be initialized to zero when it is first declared by the user.)

More than one task can wait for the same message by issuing SC_PEND calls with the same message address. The highest priority task is placed in the ready state when another task sends a message to that location. If a task pended at a mailbox is explicitly suspended, it continues to pick up messages, although it does not resume execution until it is explicitly readied.

Using these calls, it is easy to implement mutual exclusion and resource locking, as well as standard intertask communication. Resource locking is implemented simply when all the tasks attempting to use the resource pend at the same location; as each task finishes with the resource, it sends a message to that location, enabling the next task.

Synchronization between tasks can also be implemented with the two basic calls. Task A posts a message to one location, then immediately pends at another location.

Task B simply does the reverse: it receives the message, then immediately posts a message back to enable Task A. The two tasks are then synchronized.

2.6.2 Queues

VRTX also provides five additional calls, which implement message queueing. Message queues are fixed-length buffers, and enqueued messages are managed in a first-in/first-out (FIFO) manner. Unlike mailboxes, queues are not part of the user's set of variables, instead they are system-managed structures. Queues can be created dynamically by VRTX. Tasks can post messages to, pend at, or accept messages from these queues. If the queue is full, a task or interrupt handler attempting to post a message receives an error return. On the other hand, if the queue is empty, a pending task is suspended, but a task attempting to accept messages from an empty queue is not suspended. Tasks pended at a queue are readied by incoming messages in priority order, not in the order they were pended. Thus, a low priority task that arrived early at a queue cannot be activated in preference to a higher priority task.

The following VRTX calls manipulate queues:

SC_QCREATE	Create a Message Queue
SC_QPOST	Post a Message to a Queue
SC_QPEND	Pend for a Message from a Queue
SC_QACCEPT	Accept a Message from a Queue
SC_QINQUIRY	Queue Inquiry

Queues can be used to implement a generalized version of the Dijkstra primitives SIGNAL and WAIT, which are useful in establishing resource-locking mechanisms for multiple resources of the same type. Each type of resource (e.g., line printers) is assigned a specific queue, the length of which is determined by the number of resources included in that type (i.e., the number of printers on the system.) All tasks attempting to use a resource of a specified type pend at the resource's queue in a procedure similar to that described for mailboxes. The length of the queue governs how many tasks can use the resource at the same time. VRTX's priority-ordered readying of tasks ensures that several tasks waiting to use a resource receive that resource in order of their priority.

2.7 Communication and Synchronization Calls

Three of these system calls are used for the exchange of two-word (32-bit) messages via simple mailboxes. The remaining five calls in this section are used for more elaborate exchanges via message queues. Only the posting and pending calls (SC_POST, SC_PEND, SC_QPOST, and SC_QPEND) go through the rescheduling procedure which may result in a task switch.

2.7.1 SC_POST - Post a Message

This call is used by a task to post a two-word (32-bit), nonzero message to a specified message location (mailbox). Mailbox addresses must be logical addresses relative to the Address State 0 operand space if extended memory addressing is used. This call results in a task switch if a task whose priority is higher than that of the calling task was pended on that mailbox.

```
+------------------------------------------------------+
|                                                      |
|    INPUT:           R0 = SC_POST (0008H)             |
|                                                      |
|                     R1 = mailbox address             |
|                                                      |
|                  R2/R3 = message                     |
|                                                      |
|                                                      |
|    OUTPUT:          R0 = return code                 |
|                                                      |
+------------------------------------------------------+
```

RETURN CODES

0000H	RET_OK	Successful return.
0005H	ER_MIU	Mailbox already in use.
0006H	ER_ZMW	Zero message.

2.7.2 SC_PEND - Pend for Message

This call is used to obtain a two-word message from a specified message location (mailbox). If no message has yet been posted to the specified mailbox, the calling task is suspended until such posting does occur.

An optional time-out value can be specified with this call. In this case, the error code ER_TMO is returned to the calling task if no message is received within the specified number of clock ticks (See Section 3.3.1, Real-Time Clock Support). A task switch occurs if the mailbox is empty.

```
+----------------------------------------------------------+
|                                                          |
|    INPUT:            R0 = SC_PEND (0009H)                |
|                                                          |
|                      Rl = mailbox value                  |
|                                                          |
|                   R2/R3 = time-out value                 |
|                                                          |
|                                                          |
|    OUTPUT:           R0 = return code                    |
|                                                          |
|                   R2/R3 = message                        |
|                                                          |
+----------------------------------------------------------+
```

time-out value is a 31-bit value, with the high-order bit of R2 ignored. A zero value indicates no time-out.

RETURN CODES

0000H	RET_OK	Successful return.
000AH	ER_TMO	Time-out.

2.7.3 SC_ACCEPT - Accept A Message

This call is used to obtain a two-word message from a specified message location (mailbox). Unlike SC_PEND, however, this call does not suspend the calling task when no message is present; instead, the error code ER_NMP is returned immediately. This call does not go through the rescheduling procedure.

```
+---------------------------------------------------------+
|                                                         |
|    INPUT:            R0 = SC_ACCEPT   (0025H)           |
|                                                         |
|                      R1 = mailbox address              |
|                                                         |
|                                                         |
|    OUTPUT:           R0 = return code                  |
|                                                         |
|                   R2/R3 = message                      |
|                                                         |
+---------------------------------------------------------+
```

RETURN CODES

0000H	RET_OK	Successful return.
000BH	ER_NMP	No message present.

2.7.4 SC_QPOST - Post Message to Queue

This call is used by a task to post a two-word, nonzero message to a specified queue. This call results in a task switch if a task whose priority is higher than that of the calling task was pended on that queue.

```
+--------------------------------------------------------+
|                                                        |
|   INPUT:           R0 = SC_QPOST     (0026H)           |
|                                                        |
|                    R1 = queue ID number               |
|                                                        |
|                 R2/R3 = message                       |
|                                                        |
|                                                        |
|   OUTPUT:          R0 = return code                   |
|                                                        |
+--------------------------------------------------------+
```

queue ID number is 16 bits.

RETURN CODES

0000H	RET_OK	Successful return.
0006H	ER_ZMW	Zero message.
000CH	ER_QID	Queue ID error (no such queue).
000DH	ER_QFL	Queue full.

2.7.5 SC_QPEND - Pend for Message from Queue

This call is used to obtain a two-word message from a specified queue. If the specified queue is currently empty, the calling task is suspended until a message is posted at that queue.

An optional time-out value can be specified with this call. In this case, the error code ER_TMO is returned to the calling task if no message is received within the specified number of clock ticks (see Section 3.3.1, Real-Time Clock Support). A task switch occurs if the queue is empty.

```
+----------------------------------------------------------+
|                                                          |
|    INPUT:            R0 = SC_QPEND    (0027H)            |
|                                                          |
|                      Rl = queue ID number               |
|                                                          |
|                   R2/R3 = time-out value                 |
|                                                          |
|                                                          |
|    OUTPUT:            R0 = return code                   |
|                                                          |
|                   R2/R3 = message (if call successful)   |
|                                                          |
+----------------------------------------------------------+
```

queue ID number is 16 bits.

time-out value of 0 indicates that no time-out is requested.

RETURN CODES

0000H	RET_OK	Successful return.
000AH	ER_TMO	Time-out.
000CH	ER_QID	Queue ID error (no such queue).

2.7.6 SC_QACCEPT - Accept Message from Queue

This call is used to obtain a two-word message from a specified queue. Unlike SC_QPEND, however, this call does not suspend the calling task when no message is present; instead, the error code ER_NMP is returned immediately. This call does not go through the rescheduling procedure.

```
+-------------------------------------------------------+
|                                                       |
|   INPUT:           R0 = SC_QACCEPT  (0028H)           |
|                                                       |
|                    R1 = queue ID number              |
|                                                       |
|                                                       |
|   OUTPUT:          R0 = return code                  |
|                                                       |
|                 R2/R3 = message (if call successful). |
|                                                       |
+-------------------------------------------------------+
```

RETURN CODES

0000H	RET_OK	Successful return.
000BH	ER_NMP	No message present.
000CH	ER_QID	Queue ID error (no such queue).

2.7.7 SC_QCREATE - Create Message Queue

This call is used to create a message queue. This queue is of a fixed size; there is an upper limit on the number of messages that can be enqueued at any given time. The queue is managed by VRTX in a 'first-in/first-out' (FIFO) manner. This call does not go through the rescheduling procedure.

```
+------------------------------------------------------+
|                                                      |
|    INPUT:          R0 = SC_QCREATE  (0029H)          |
|                                                      |
|                    R1 = queue ID number              |
|                                                      |
|                    R2 = number of entries in queue   |
|                                                      |
|                                                      |
|    OUTPUT:         R0 = return code                  |
|                                                      |
+------------------------------------------------------+
```

queue ID number of 0 is allowed.

number of entries in queue cannot exceed 4095 (12 bits of significance).

RETURN CODES

0000H	RET_OK	Successful return.
0003H	ER_MEM	No memory available (insufficient system memory).
000CH	ER_QID	Queue ID error (ID number already assigned).

2.7.8 SC_QINQUIRY - Queue Inquiry

This call is used to obtain a count of the number of messages waiting in a queue. If the count is nonzero, the actual contents of the head-of-queue message (i.e., that message which will be given to the next SC_QPEND or SC_QACCEPT request) is also returned to the caller, without being extracted from the queue. This call does not go through the rescheduling procedure.

It should be noted that although the caller is given a copy of the first message, the message remains queued. The calling program will still need to QPEND or QACCEPT the message to remove it from the queue.

This function can be used at both task and interrupt service routine levels.

> **Note:** If the return code is nonzero, all registers other than R0 remain unmodified.

```
+--------------------------------------------------------+
|                                                        |
|   INPUT:            R0 = SC_QINQUIRY (002AH)           |
|                                                        |
|                     R1 = queue ID number               |
|                                                        |
|                                                        |
|   OUTPUT:           R0 = return code                   |
|                                                        |
|                  R2/R3 = message                       |
|                                                        |
|                     R4 = count of messages in queue    |
|                                                        |
+--------------------------------------------------------+
```

count of messages in queue is invalid if ER_QID returned.

message is returned as zero if there are no messages in the queue (i.e., count is zero.)

RETURN CODES

0000H	RET_OK	Successful return.
000CH	ER_QID	Queue ID error (no such queue).

Chapter 3

INTERRUPT SUPPORT

3.1 Introduction

VRTX/1750 makes no assumptions about its target environment, other than the existence of a 1750A computer with some random access (read/write) memory. Instead, it is the responsibility of the user to supply such hardware-dependent service routines as may be required to initialize devices and to handle interrupts. The integration of these routines into a VRTX system is the subject of this chapter. The shaded portion of Figure 3-1, Interrupt Architecture, indicates the functions that are covered.

3.2 Interrupt Service Routines

A real-time system must be able to respond quickly to externally generated interrupts. VRTX provides the means by which user-supplied interrupt service routines (also called interrupt handlers) can influence the scheduling of critical tasks. In contrast to application tasks, which are scheduled synchronously by VRTX, an interrupt handler routine is executed asynchronously to the rest of the system software whenever its hardware interrupt is generated.

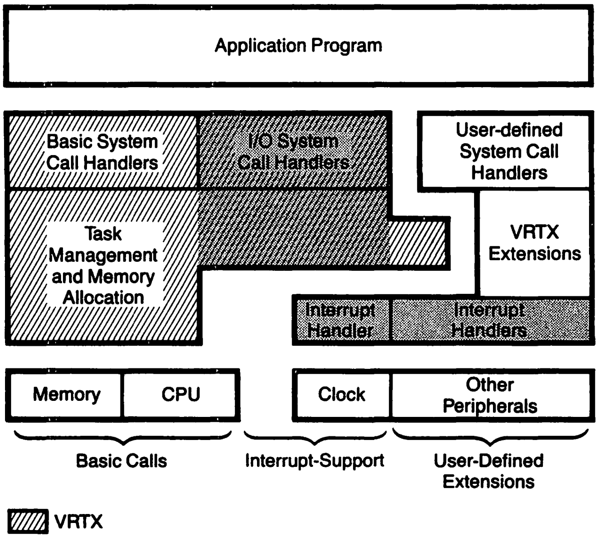


Figure 3-1. Interrupt Architecture

Upon detection of an interrupt request, the 1750A hardware automatically saves the current program status information in a linkage block and executes a jump through a vector in the corresponding service block. The user supplies the linkage and service blocks, as well as the reserved-address pointers to these blocks, for each interrupt and trap that may occur.

Because interrupt handler routines are entered directly, without intervention by VRTX, it is the responsibility of each such routine to save and restore registers as required. Moreover, since interrupt-handling is entirely distinct from the multitask environment, a simple Load Status ('LST' in MIL-STD mnemonics) instruction can be used to exit such a routine.

In most cases a close coordination is required between interrupt servicing and its effect on the multitask environment. Two special calls, UI_ENTER and UI_EXIT, allow interrupt handlers to interface with VRTX. A typical interrupt handler is begun with the UI_ENTER system call and is ended with the UI_EXIT call (rather than an LST instruction). So long as the handler is 'bracketed' in this manner, it can make use of a variety of VRTX services. In particular, an interrupt handler can issue SC_ACCEPT and SC_QACCEPT calls to obtain messages from tasks; and, conversely, it can use SC_POST and SC_QPOST to send messages to tasks.

In fact, an interrupt handler can issue all VRTX calls except the following:

```
SC_LOCK                    SC_QPEND
SC_PCREATE                 SC_QCREATE
SC_PEND                    SC_TDELAY
SC_TCREATE
SC_TDELETE
SC_TSUSPEND (when time-slicing enabled)
SC_TPRIORITY
```

The user should always take care when issuing, from an interrupt service routine, any SC-prefix call that could suspend the interrupted task, since unexpected consequences could result.

3.2.1 Interrupt Service Routines and VRTX

All multitask activity ceases the moment an interrupt is detected, and control passes to the designated interrupt service routine. All tasks resume their former states when control returns, unless the interrupt service routine signals the occurrence of a significant event with a VRTX call (such as UI_TIMER, SC_POST or SC_QPOST). Rescheduling is initiated if the UI_EXIT (and matching UI_ENTER) call is used instead of LST to return from the interrupt service routine. Rescheduling occurs after control returns to the task environment (i.e., after all nested interrupts have been handled and any system calls in progress are completed). See Appendix E, Task Rescheduling, for more details.

As mentioned earlier, most interrupt service routines begin with UI_ENTER and end with UI_EXIT. This pair of calls guarantees that VRTX performs the rescheduling procedure upon exit from the interrupt handler. Both calls must always be used together. If task switching is not needed and the slight amount of CPU time that UI_ENTER and UI_EXIT consume must be saved, then the LST instruction may be used instead of UI_EXIT (in this case, UI_ENTER is not to be coded either). When using LST, the user must *guarantee* that no nested interrupt whose handler uses UI_ENTER/UI_EXIT occurs. Were such a nested interrupt handler to run, VRTX would *not* perform the rescheduling procedure upon exit from the handler ending with LST.

It should be noted that VRTX, in its internal processing of system calls, makes extensive use of the 'disable interrupts' and 'enable interrupts' XIO instructions. Thus, although an interrupt service routine is always entered with interrupts automatically disabled, a call to VRTX may result in interrupts being reenabled. For this reason, interrupt service routines should use the Interrupt Mask Register--rather than the DSBL and ENBL signals--as the primary mechanisms for controlling their own interruptibility.

In other words, if an interrupt service routine that uses VRTX services wishes not to be interrupted, it should ensure that the interrupt mask register (MK) is set to all zeroes. In order to accomplish this, a value of zero for New Interrupt Mask can be specified within the service block that initially gives control to the routine; or alternatively, the routine itself can reset the mask by means of the I/O set mask instruction (SMK).

3.2.2 Format of an Interrupt Service Routine

The interrupt service routine is responsible for saving and restoring the contents of the registers it modifies in the course of its execution, including register R0 used for the VRTX function code and R1 used for the linkage pointer parameter to UI_EXIT. Note also that the contents of register R0 and R1 remain in the saved area where they are restored by VRTX during the course of the UI_EXIT call. VRTX performs no register saves on an interrupt (in fact, VRTX doesn't handle interrupts at all). Thus, an interrupt service routine usually has the following format (using MIL-STD assembler format):

```
;
; Interrupt Handler for Clock
;
int
        ST     R0,save+3       ; save registers used
        ST     R1,save+4
        LIM    R0,16H          ; load function code
        BEX    4               ; make ui_enter call

        LIM    R0,0FFFFH
        XIO    R0,0D           ; put ack line lo
        LIM    R0,0
        XIO    R0,0D           ; put ack line hi
        LIM    R0,0FH
        XIO    R0,RPI          ; reset pending int

        LIM    R0,12H          ; load function code
        BEX    4               ; make ui_timer call

        LIM    R0,11H          ; load function code
        LIM    R1,save         ; set up save area
        BEX    4               ; make ui_exit call

save    DATA   0,0,0,0,0       ; save area
npsa    DATA   0,0,int         ; new program status area

        END
```

3.2.3 Communication from an Interrupt Service Routine

Besides the standard VRTX system calls, the following special calls are used to interface an interrupt service routine to VRTX.

UI_ENTER	Enter an Interrupt Handler
UI_EXIT	Exit from Interrupt Handler
UI_TIMER	Announce Timer Interrupt

In conjunction with UI_ENTER and UI_EXIT, an interrupt service routine may use standard VRTX services--such as SC_POST and SC_QPOST--to activate user tasks and transmit messages to them. SC_POST deposits a two-word message in a specified location. The contents of the location must be zero at the time SC_POST is invoked; otherwise, the location is deemed already in use. If the task for which the message is intended has issued an SC_PEND call at the appropriate location, its task state is changed from suspended to ready, even though the task is currently not active due to the interrupt.

If more than one task is awaiting a message at this location, only the highest priority task receives the message and is readied. If a task does not issue an SC_PEND call for the message, the SC_POST simply posts the message so that it can be retrieved when a task issues the SC_PEND call. The UI_EXIT call can be used to exit the interrupt service routine and initiate the rescheduling procedure. Thus, the newly readied task receives control if it has highest priority. The SC_QPOST call has much the same effect as the SC_POST call, except the message is directed to a queue instead of to a mailbox. Again, the UI_EXIT call can be used to exit the interrupt service routine and initiate the rescheduling procedure.

Interrupt service routines that do not end with UI_EXIT should disable (or mask out) interrupts; otherwise, all interrupt handlers that are nested deeper are unable to initiate the rescheduling procedure. Remember, in VRTX/1750, the UI_EXIT call must be matched with a UI_ENTER call.

The UI_TIMER call is used to integrate a real-time clock into VRTX. The user must define a minimal clock service routine, which merely handles the mechanics of dealing with a specific clock device (such as the optional Timer A or Timer B, if these features are implemented). On a periodic basis, the clock handler issues the UI_TIMER call, informing VRTX that another time interval (or 'tick') has expired. Even in target environments without a real-time clock device, a timer of sorts (basic, but sufficient for task delay and round-robin scheduling) can be implemented by issuing the UI_TIMER command on a regular basis from other interrupt handlers.

3.2.4 Interrupt Management Calls

The following interrupt management calls, Enter Interrupt and Exit Interrupt, provide an interface from interrupt handlers to the VRTX multitasking environment. The Exit Interrupt call (UI_EXIT), when paired with a corresponding UI_ENTER call, allows the rescheduling procedure to occur upon completion of interrupt servicing.

It is recommended that Interrupt Service Routines use UI_ENTER and UI_EXIT, rather than LST instructions. These VRTX function calls perform a number of activities beneficial to the application that are not available with LSTs. If mailboxes and/or queues are used to communicate with tasks, a task switch may be needed. This need is checked in UI_EXIT by initiating the rescheduling procedure. If an ISR can be interrupted by higher priority interrupts that cause POSTing or QPOSTing, then all nestable ISRs must use UI_ENTER and UI_EXIT to take advantage of the rescheduling procedure.

These function calls are optimized for fast performance to assist in short interrupt service routine durations.

3.2.5 UI_ENTER - Enter Interrupt Handler

This call is used to enter an interrupt handler that uses UI_EXIT for termination. UI_ENTER and UI_EXIT form a matched pair; whenever one is used, the other must also be used.

Note that register R0 must have been saved prior to invoking UI_ENTER, since this register is used to pass a function code to VRTX. Thus, the call to UI_ENTER should not be the very first instruction in the interrupt service routine.

Instead, since the 1750A architecture disables interrupts automatically prior to transferring control to an interrupt service routine, it is sufficient for an interrupt handler to defer calling UI_ENTER until reaching the point where it wishes to reenable interrupts.

```
+--------------------------------------------------+
|                                                  |
|   INPUT:          R0 = UI_ENTER    (0016H)       |
|                                                  |
|                                                  |
|   OUTPUT:         R0 = return code               |
|                                                  |
+--------------------------------------------------+
```

RETURN CODES

0000H	RET_OK	Successful return.

3.2.6 UI_EXIT - Exit from Interrupt Handler

This call is used to exit an interrupt handler. Unlike the LST instruction, the UI_EXIT call interfaces with the VRTX kernel, thereby allowing the rescheduling procedure to occur upon return to the multitask environment.

The UI_EXIT call should be used in preference to the LST instruction whenever a previous posting call--either from the current handler or from the handler of an intervening nested interrupt -- may have readied a pended task. In general, always use UI_EXIT if the interrupt handler itself can be interrupted; use LST only if interrupts have been disabled or masked. Unless this is done, an intervening nested interrupt cannot use UI_EXIT to initiate the rescheduling procedure.

Always pair UI_EXIT with a matching UI_ENTER at the beginning of the interrupt.

UI_ENTER and UI_EXIT are optimized for fast performance to assist in short ISR durations.

```
+------------------------------------------------------------+
|                                                            |
|   INPUT:          R0 = UI_EXIT      (0011H)                |
|                                                            |
|                   R1 = linkage pointer                     |
|                                                            |
|                                                            |
|   OUTPUT:         No return is possible                    |
|                                                            |
+------------------------------------------------------------+
```

linkage pointer in register R1 is similar in function to the address operand ordinarily supplied to the LST instruction. Thus, it must point to a block of data from which the previous Interrupt Mask, Status Word, and Instruction Counter values can be restored. In addition, however, the saved values of registers R0 and R1--which are used in making the UI_EXIT call--must also be restored in order to fully return to the interrupted program's context.

Thus, the linkage pointer in R1 points to a five-word context block, formatted as follows:

```
+--------------------------+
|  Old Interrupt Mask      |
|--------------------------|
|  Old Status Word         |
|--------------------------|
|  Old Instruction Counter |
|--------------------------|
|  Saved R0 Contents       |
|--------------------------|
|  Saved Rl Contents       |
+--------------------------+
```

3.2.7 Other System Calls from ISRs

The previous function calls are used only by interrupt service routines, but are not the only functions available at ISR level. Generally, a system call is available for use at ISR level if it meets the following criteria:

1. the system call does not cause suspension of the caller; or

2. the system call does not cause a rearrangement of the task, queue, or partition control block chains.

The following system calls are useful for communication between ISRs and tasks, and also support ISR control over multitasking priorities and scheduling.

```
Task status:  SC_TINQUIRY and
              SC_TSUSPEND (when time-slicing not invoked)

Scheduling management:  SC_POST, SC_ACCEPT, SC_QPOST and
                        SC_QACCEPT

Timer management:  SC_GTIME, SC_STIME and UI_TIMER
```

3.3 Integrated Support for Special Devices

Many VRTX applications require a real-time clock device. Support for such a device is fully integrated into VRTX. The user need only supply a short hardware-dependent interrupt service routine for the clock device. VRTX, in turn, manages all the logical operations needed to provide user application tasks with a full repertoire of associated clock management commands. It is important to realize, however, that VRTX has been designed to operate quite satisfactorily without the existence of these devices; even the clock is not required.

The VRTX commands that support a clock fall into two categories: calls from user tasks and calls from interrupt handlers. VRTX recognizes four user calls for timer services (SC_GTIME, SC_STIME, SC_TDELAY and SC_TSLICE) and one call from the clock service routine (UI_TIMER). The first two calls permit user tasks to obtain the value from the clock counter and to set a new value for the counter. The remaining user calls implement task delays and round-robin scheduling. The UI_TIMER call, issued from an interrupt handler, simply notifies VRTX that another clock interval (or 'tick') has expired.

3.3.1 Real-Time Clock Support

The system calls in this section allow tasks to obtain the value of a VRTX-maintained 31-bit timer, to set that timer to a selected value, to delay a task for a specified period, and to enable time-slicing. The UI_ prefix call UI_TIMER is used by the clock handler to inform VRTX that a clock 'tick' has occurred. Only the task delay call results in a task switch.

The timer maintained by VRTX is a 31-bit value. It is set to zero at VRTX_INIT time. This timer is incremented by 1 for each UI_TIMER call, which is used to signal 1 clock tick. Since this timer is 31 bits wide, it rolls over from 07FFFFFFF Hexadecimal to 0. Once past VRTX_INIT, this timer is only modified by UI_TIMER and SC_STIME function calls.

3.3.2 SC_GTIME - Get Time

This call is used to obtain the current value, in 'ticks' of the clock counter; it does not go through the rescheduling procedure.

```
+--------------------------------------------------------+
|                                                        |
|    INPUT:            R0 = SC_GTIME    (000AH)          |
|                                                        |
|                                                        |
|    OUTPUT:           R0 = return code                  |
|                                                        |
|                   R2/R3 = clock counter value          |
|                                                        |
+--------------------------------------------------------+
```

RETURN CODES

0000H	RET_OK	Successful return.

3.3.3 SC_STIME - Set Time

This call sets the current value, in number of 'ticks,' of the system clock counter; the system defaults this value to zero at initialization. This call does not go through the rescheduling procedure.

```
+----------------------------------------------------+
|                                                    |
|   INPUT:          R0 = SC_STIME     (000BH)        |
|                                                    |
|                R2/R3 = new value                   |
|                                                    |
|                                                    |
|   OUTPUT:         R0 = return code                 |
|                                                    |
+----------------------------------------------------+
```

new value is a 31-bit value.

RETURN CODES

0000H	RET_OK	Successful return.

3.3.4 SC_TDELAY - Task Delay

This call suspends execution of the calling task for a specified number of clock 'ticks.' VRTX maintains the TCBs of delayed tasks in a queue that is arranged in order of the expiration of their delay periods. For example, if task 1 was delayed for 10 ticks at time 5, and task 2 was delayed for 5 ticks at time 7, then in the delay chain task 2 will be ahead of task 1. This scheme reduces the overhead in processing clock ticks. The delay value stored in the TCB is not an absolute delay, but a relative increment from the delay value of its predecessor. The rescheduling procedure is always initiated and generally results in a task switch.

```
+------------------------------------------------------------+
|                                                            |
|   INPUT:            R0 = SC_TDELAY   (000CH)               |
|                                                            |
|                  R2/R3 = delay interval                    |
|                                                            |
|                                                            |
|   OUTPUT:           R0 = return code                       |
|                                                            |
+------------------------------------------------------------+
```

delay interval is the number of clock 'ticks' for which the calling task is to be suspended

RETURN CODES

0000H	RET_OK	Successful return.

3.3.5 SC_TSLICE - Enable Round-Robin Scheduling

This call is used to enable and disable time-sliced, round-robin scheduling of equal-priority tasks under VRTX. Every time VRTX is notified of a clock 'tick' (i.e., it receives a UI_TIMER call--see Section, 3.3.6, UI_TIMER call), it records which task is in control. If the same task is continuously in control during the time-slicing interval, the task is suspended when the interval elapses, and its TCB is put at the end of its priority group on the ready chain.

The rescheduling procedure is not initiated by this call. All groups of equal priority tasks are subject to time-slicing (e.g., three tasks at priority 5 and six tasks at priority 10 will all undergo time-slicing).

If time-slicing is in effect and a task suspends for any reason, it is put at the end of its priority group on the active chain. Round-robin scheduling is disabled when the SC_TSLICE call has a 0 interval specified.

```
+-----------------------------------------------------+
|                                                     |
|    INPUT:          R0 = SC_TSLICE   (0015H)         |
|                                                     |
|                    R3 = time-slicing interval       |
|                                                     |
|                                                     |
|    OUTPUT:         R0 = return code                 |
|                                                     |
+-----------------------------------------------------+
```

time-slicing interval is the number of clock 'ticks' which are to comprise the time-slice interval for round-robin scheduling. A value of 0 disables time-slicing.

RETURN CODES

0000H	RET_OK	Successful return.

3.3.6 UI_TIMER - Post Time Increment from Interrupt

This call is used by an interrupt handler to inform VRTX that a time interval (or 'tick' has expired. A task switch may occur after a subsequent UI_EXIT call if a delayed task is readied by the UI_TIMER call and its priority is higher than the interrupted task.

```
+------------------------------------------------------+
|                                                      |
|    INPUT:          R0 = UI_TIMER    (0012H)          |
|                                                      |
|                                                      |
|    OUTPUT:         R0 = return code                  |
|                                                      |
+------------------------------------------------------+
```

RETURN CODES

0000H	RET_OK	Successful return.

Chapter 4

CONFIGURATION AND INITIALIZATION

HUNTER ⁘ READY

This chapter describes the VRTX/1750 data structure in a user-supplied Configuration Table, and the two different actions that make up system initialization: VRTX initialization and user-supplied initialization.

4.1 Configuration Table

VRTX/1750 is a single, indivisible PROM product, rather than a collection of individual modules requiring a particular linker and host for its assembly. VRTX is designed to be as independent as possible from individual development systems and target configurations.

VRTX does not exist in isolation and must be connected to its surrounding environment. The Configuration Table supplies this vital link, and is VRTX's window to its environment. Within this table, the user specifies the parameters VRTX needs to define a particular configuration. The word at location 0050 hex is reserved by VRTX to point to the Configuration Table. Also, VRTX reserves the word at location 0051 hex to point to a data area reserved for VRTX's private operand references.

At system initialization, these two pointers at the reserved locations point to the Configuration Table and to the VRTX Workspace (i.e., to the area of system memory reserved for VRTX). Parameters in the Configuration Table indicate the size of the VRTX Workspace, how many tasks can exist at any one time, and how large each task's user stack should be. The VRTX Workspace must be large enough to contain the VRTX system stack (with a minimum size of 64 words), a 48-word area for VRTX system variables, a 29-word TCB for each active task, a stack for every task in the system, and a two-word area for every allocatable block of user memory. In addition, the VRTX Workspace must be large enough to accommodate a 5-word control block for each memory partition and a control block of length 2n+7 (where n is the number of queue elements) for each defined message queue.

The following describes in detail each parameter in the Configuration Table.

reserved represents a parameter reserved for future use of VRTX. The user should always supply a value of zero (0) here.

VRTX-workspace-size specifies the total size, in words, of memory area available to the kernel. See Section 4.2, Determining VRTX-workspace-size, for more information.

VRTX-stack-size specifies, in words, the amount of memory from VRTX-workspace-size to be dedicated to the VRTX stack (minimum = 64 words).

user-stack-size specifies the length, in words, of stacks assigned to user tasks. During system initialization, VRTX automatically allocates stacks of this size. There is another option. To bypass VRTX and explicitly allocate stack size by using the TCREATE extension, the user supplies a value of zero (0) here. See Appendix D, Implementation Notes, for more information on user-allocated stacks.

user-task-count specifies the maximum number of tasks that can be simultaneously active in the system. VRTX uses this value to allocate Task Control Blocks (TCBs) and stack space.

VRTX-codespace-addr specifies the starting address of the VRTX code.

sys-TCREATE-addr and **sys-TDELETE-addr** are optional parameters that allow the user to perform special processing whenever tasks are created or deleted. If these parameters are not used, a value of zero (0) is supplied here. For additional information, see Chapter 5, Support for User-Defined Extensions.

sys-TSWITCH-addr is another optional parameter that allows a user-defined routine to be given control whenever a context-switch is made from one task to another. If no special context switching code is required, the user supplies a value of zero (0) here. For more information, see Chapter 5, Support for User-Defined Extensions.

4.2 Determining VRTX-workspace-size

VRTX-workspace-size specifies the total size of memory available to the VRTX kernel. The amount of memory required by VRTX to operate an application is determined by a calculation formula using the following elements.

VRTX/1750

0	(reserved, must = 0)
1	VRTX-workspace-size
2	VRTX-stack-size
3	(reserved, must = 0)
4	(reserved, must = 0)
5	(reserved, must = 0)
6	(reserved, must = 0)
7	user–stack–size
8	(reserved, must = 0)
9	user–task–count
A	VRTX-workspace-addr
B	(reserved, must = 0)
C	*sys–TCREATE–addr
D	*sys–TDELETE–addr
E	*sys-TSWITCH-addr
F	(reserved, must = 0)

* indicates optional parameter; if omitted or unused, must be set to 0.

Figure 4-1. VRTX Configuration Table

Symbol	Meaning	Workspace size formula	Notes
--	VRTX system variables	48	allocated by system
t	Max. number of tasks in system	29*t	each task allocated a 29-word TCB
us	user stack size	us*t	total stack size area.
p	Memory partitions in system	5*p	5 words allocated for each partition control block
b	Memory blocks in system	b=Plsize/ Pblksize + P2size/ P2blksize ...	Total blocks in system determined by each partition size divided by each partition's block size
q	Queues in system	7*q	7 words allocated for each queue control block
qe	queue elements in system	qe=Qlsize + Q2size	Size of each queue
s	system stack	s	Minimum size is 64 words

Figure 4-2. Determining VRTX-workspace-size

VRTX-workspace-size is expressed in words. The formula for determining VRTX-workspace-size is presented below and its result expressed in words.

```
VRTX–workspace–size:
= 48 + 29t + (us*t) + 5p + 2b + 7q + 2qe +  s
```

For example, a system using VRTX/1750 has the following user-specified configuration:

```
10 tasks
user stack size of 64 words
1 partition of size 128 words with 4 word blocks = 32 blocks
1 partition of size 256 words with 32 word blocks = 8 blocks
8 queues each of size 10 = 80 queue elements
2 queues each of size 20 = 40 queue elements
```

This formula determines its VRTX-workspace-size:

```
= 48 + 29t + (us*t) + 5p + 2b + 7q + 2qe + s
= 48 + (29*10) + (64*10) + (5*2) + (2*(32+8)) + (7*10) +
   (2*(80+40)) + 64
= 1442 words
```

4.3 Support for System Initialization

System initialization comprises those preliminary activities necessary to have the system in a predicted state prior to application execution. Examples of system initialization include timer initialization, initial task priorities, device states, and general software state variables.

System initialization depends on the overall board environment. Because VRTX makes only minimal assumptions about its environment, it performs only that part of initialization dependent upon the microcomputer and memory. The user defines the rest of initialization.

Overall system initialization is defined in two different groups of actions: VRTX initialization and user-supplied initialization.

4.3.1 VRTX Initialization

Two separate calls, VRTX_INIT and VRTX_GO, initialize VRTX and start application processing.

The VRTX_INIT function call performs the following:

* Sizes and clears the VRTX Workspace.

* Saves in the VRTX Workspace the addresses to user-provided extensions in the Configuration Table, including the Task Create, Task Delete, and Task Switch hooks.

* Sets up and reserves Task Control Blocks for SC_TCREATE calls.

* Sets up the user-specified stacks for each task.

* Initializes other internal VRTX variables.

* Returns control to the caller.

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

A subsequent VRTX_GO call, from which there is no return, begins executing the highest priority task created in the user's initialization code. Multitasking is now underway.

When Reset is activated on the 1750A, the processor aborts its current operation, sets a variety of system registers to a known state, and fetches the instruction located at address 0. This instruction, typically encoded in PROM or ROM, should be a branch or jump to a user-supplied pre-initialization routine. The pre-initialization routine must:

1. Load addresses 0050 and 0051 (hexadecimal) with pointers to the VRTX Configuration Table and the VRTX Workspace, respectively.

2. Load addresses 002A and 002B (hexadecimal) with pointers to the Executive Call linkage and service blocks.

3. Initialize the Executive Call service block, as described in Section 2.1.2, Executive Call Interrupt, and Appendix C, IVT and TCB Formats.

4. Perform a VRTX_INIT call.

5. Perform any user required initialization.

6. Perform a VRTX_GO call.

Even if Reset is not used to activate VRTX, the above steps must be performed in order to initialize VRTX properly.

4.3.2 User-Supplied Initialization

User-supplied initialization consists of initialization specific to hardware devices within the system, and initialization specific to the function that the application performs.

Device initialization depends on the board-level environment. Usually the real-time clock, and any special user devices (e.g., a memory management unit) are initialized at overall system initialization time. Initialization of these devices is provided by user-supplied code external to VRTX.

Initialization specific to the application's function usually consists of setting up software control structures and variables related to the application code, such as mailboxes, queues, and boolean variables as well as creating the initial set of tasks with which the system begins.

User-supplied device initialization can be performed in either of two different places:

1. in code that precedes the VRTX_INIT call; or

2. in code immediately after the VRTX_INIT call and before the first task begins with the VRTX_GO call.

Users can perform initialization in either, both, or neither of these places. Some devices should be initialized before VRTX, because their successful operation is a prerequisite for VRTX initialization. For example, if the extended memory addressing option exists in the system, the map registers should be initialized before VRTX, because system memory allocation depends on these registers. Another example is the stack; VRTX_INIT needs a stack, so the stack pointer must be set up prior to that call. Devices without system-wide ramification can be initialized after VRTX.

Initialization of constructs that require VRTX services, such as queues and memory partitions, must be performed after the call to VRTX_INIT.

4.3.3 Use of System Calls during Initialization

Using any VRTX system calls prior to the call to VRTX_INIT causes unpredictable results.

The following system calls can be used after the call to VRTX_INIT:

```
SC_TCREATE  SC_TDELETE  SC_TPRIORITY  SC_TINQUIRY
SC_PCREATE  SC_PEXTEND  SC_GBLOCK     SC_RBLOCK
SC_STIME    SC_GTIME    UI_TIMER      SC_TSLICE
SC_QCREATE  SC_POST     SC_QPOST      SC_QINQUIRY
```

The three construct-creation calls, SC_TCREATE, SC_PCREATE, and SC_QCREATE, are typically used at this location.

4.4 Initialization Calls

Two special calls (VRTX_INIT and VRTX_GO) are used during initialization. The many activities in VRTX_INIT are illustrated in Section 4.3.1, VRTX Initialization. VRTX_GO forces the scheduling of the highest priority task.

4.4.1 VRTX_INIT - Initialize VRTX

This call, issued from user-supplied code executed upon system reset, causes VRTX to perform its initialization activities.

Unlike other VRTX calls, this call is not issued by means of a BEX instruction. Instead, the call is made as follows (using MIL-STD syntax):

```
SJS  R15,vrtx
```

where `vrtx` specifies the starting location plus 2 (offset 2) of VRTX itself. (In IEEE mnemonics, the same instruction is written as '`CALL @-R15,vrtx`'.)

```
+---------------------------------------------------------+
|                                                         |
|    INPUT:          R15 = stack-pointer                  |
|                                                         |
|                   vrtx = VRTX-address                   |
|                                                         |
|                                                         |
|    OUTPUT:          R0 = return code                    |
|                                                         |
+---------------------------------------------------------+
```

RETURN CODES

0000H	RET_OK	Successful return.
000FH	ER_INI	Fatal initialization error.

4.4.2 VRTX_GO - Start Multitasking

This call, issued only after VRTX_INIT, causes VRTX to gain control. VRTX begins multitasking by starting the highest priority task. No return is made to the caller.

```
+-----------------------------------------------------------+
|                                                           |
|    INPUT:          R0 = VRTX_GO      (0031H)              |
|                                                           |
|                                                           |
|    OUTPUT:         No output values                       |
|                                                           |
+-----------------------------------------------------------+
```

Chapter 5

SUPPORT FOR USER-DEFINED EXTENSIONS

5.1 Introduction

VRTX/1750 can supply much of the system software requirements for most embedded applications. However, as a component designed for use in many different applications with many different hardware configurations, it does not contain all the code that might be found in an operating system tailored to a specific environment.

VRTX and other Hunter & Ready silicon software components are designed to be connected to build a large complete system, just as stereo components are hooked up to build a complete audio system. As with stereo components, silicon software components must provide the 'plugs' and 'cables' that allow them to be hooked up. There must exist mechanisms in the components that make it easy to interface them to external software and other components.

There are four kinds of interfaces that must be defined between the VRTX component and user-supplied code:

* The interface between VRTX and user-supplied application code (i.e., user tasks). This interface is defined by the basic VRTX system calls and key entries in the Configuration Table (see Chapter 2, Basic System Calls).

* The interface between VRTX and user-supplied interrupt service routines and initialization code. This interface is defined in Chapter 3, Interrupt Support.

* The interface between VRTX and additional system call handlers. These handlers are supplied by the user and designed to support special hardware devices--for example, file-handling calls (open, close, etc.) for supporting a disk.

* The interface between VRTX and user-supplied VRTX extensions, that is, code that extends the basic VRTX mechanisms so that each task can support additional information. (For example, if a floating-point unit exists in the system, the values of the FPU registers must be saved and restored with every task switch. User code must be used to perform this operation, since not all systems have an FPU; but the user code must be activated by VRTX on every task switch.)

Most packaged operating systems fail to provide an adequate definition of the last three interfaces (i.e., they do not provide enough 'hooks' for the user to interface non-application code to the package). Therefore, unless the user's application is exactly that envisioned by the package designers, the source code of the operating system requires modification--an expensive and risky job.

This chapter describes the last two interfaces. (The interrupt service routine and initialization code interface is covered in Chapter 3, Interrupt Support and in Chapter 4, Configuration and Initialization.) Figure 5-1, Extensions Architecture, shows the functions involved.

5.2 User-Defined System Call Handlers

User-defined system call handlers serve the same basic purpose as the system call handlers provided by VRTX itself; they provide system services that can be accessed with a program-generated interrupt instruction. They are used to implement functions that are not appropriate for VRTX itself to provide. For example, if the system includes a local area network (e.g., Ethernet), certain communications primitives (e.g., Send and Receive) could be implemented with user-defined system calls.

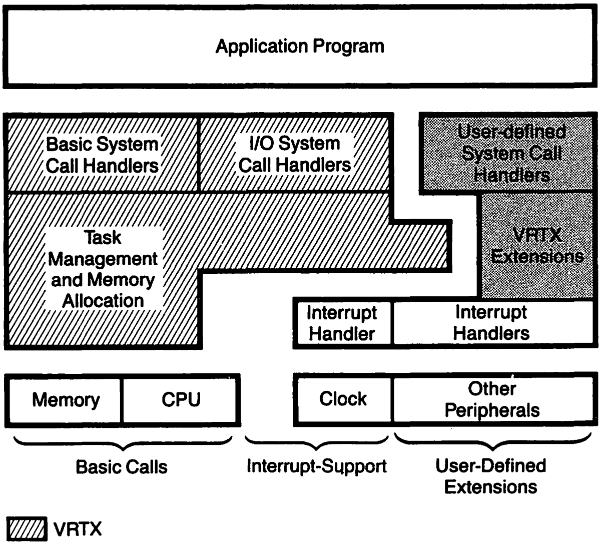


Figure 5-1. Extensions Architecture

5.2.1 Overview

The Branch to Executive (BEX) instruction of the 1750A CPU allows program-generated traps to access as many as 16 individual system call handlers, identified as 'BEX 0' through 'BEX 15'. One of these, ordinarily 'BEX 0', must be reserved as the entry point for making VRTX system calls. The remaining 15 entry points, however, may be serviced by user-defined system call handlers.

Unlike application programs, these system call handlers are not part of the multitask environment (i.e., they have no priority and do not undergo scheduling; instead, they execute when BEX instructions are encountered). Because they execute with privileged status (PS=0 in the Executive Call 'New Status Word'), they have complete access to all the resources of the 1750A architecture (e.g., they can execute I/O instructions). Most VRTX calls can be made from a user-supplied system call handler (all code outside the multitask environment can execute the calls an interrupt service routine can execute; see Section 3.2, Interrupt Service Routines).

5.2.2 Interfacing System Call Handlers to VRTX

User-defined system call handlers are interfaced to VRTX in a manner that is identical to the way interrupt service routines are interfaced to VRTX. As many as 16 'New IC' vectors can be included in the status block referenced by the Executive Call (Interrupt 5) service pointer, just as 'New IC' vectors are included in the status blocks for normal interrupt servicing. When the BEX n instruction is executed, the processor transfers control to the address specified in the (n+1)st location past the 'New Status Word' in the EXEC new status block. (See Figure C-1 in Appendix C, IVT and TCB Formats.)

As with interrupt handlers, a user-defined system call handler is responsible for saving and restoring registers; it can exit with a simple Load Status (LST) instruction. If bracketed with UI_ENTER and UI_EXIT, however, a user-defined system call handler can employ such VRTX services as SC_POST, SC_ACCEPT, SC_QPOST and SC_QACCEPT; in fact, it can issue all calls that are permitted from interrupt handlers.

5.3 VRTX Extensions

Occasionally, it is critical for users to be able to attach general system-mode software to VRTX, software that is not necessarily activated by an BEX instruction. For example, if the system includes the extended memory addressing option, the mapping registers have to be handled by user-defined system software. When each task is created, an array of values for the map registers must also be created; and when a task switch occurs, the contents of the registers may be switched as well (e.g., if there are more than 16 tasks and thus more tasks than maps).

To do this, the user needs to create an extension to the VRTX-managed Task Control Block (TCB) where the values of each task's instruction page registers and operand page registers can be stored. User-supplied code must also get control from VRTX on every task switch, so that the map registers can be updated.

5.3.1 Mechanisms for Extending VRTX

Three optional vectors in the Configuration Table provide the hooks that allow users to interface general system software to VRTX. The vectors **sys-TCREATE-addr** and **sys-TDELETE-addr** point to user-supplied routines that are given control whenever a task is created or deleted by VRTX. When the routine receives control, register R1 contains a pointer to the TCB of the newly created (or deleted) task. Register R2 contains a pointer to the TCB of the creator's (or deletor's) task. The optional vector **sys-TSWITCH-addr** points to a routine that is given control whenever a task switch occurs. (In this case, register R1 contains a pointer to the old TCB; register R2 contains a pointer to the new TCB.)

Thus, the user-supplied code has immediate access to all the data stored in the TCB (see Appendix C, IVT and TCB Formats) of the affected task. Moreover, each task's TCB contains a location (offset 02) that can hold a pointer to a TCB extension in user-defined space. This extension can be used to store additional information about the task (e.g., the values of map registers or floating point registers).

As with system call handlers, VRTX extensions can execute all system calls that can be issued from interrupt service routines. Such extension routines are responsible, of course, for saving and restoring all affected registers. Unlike system calls handlers, however, they should not use the UI_ENTER/UI_EXIT pair. Instead, an extension routine returns to VRTX by means of an unconditional branch to the address contained in register R14. In MIL-STD mnemonics, the instruction that returns to VRTX is coded as:

```
JC  7,0,R14
```

The same instruction, in IEEE format, is:

```
BR 0(R14)
```

To reiterate: any registers modified by the extension must be saved or VRTX may be seriously affected.

5.3.2 Examples of VRTX Extensions

Two examples of VRTX extensions have already been mentioned: routines to control a floating-point unit (FPU), or routines to manage page registers for extended memory addressing. These are both examples that require **extended state** information for each task in the system. Another example is a Fast Fourier Transform (FFT) device for signal-processing applications. In every case, the extended state can be saved in a TCB extension and manipulated by user code that is activated on task create, delete or switch instructions.

Another example of a user-defined VRTX extension is special code run at task create time to set up the run-time environment of a high-level language (e.g., to allocate stack frames of a specified size). Tracing and debugging can also be implemented by means of VRTX extensions. User-supplied code can be activated at every task switch to record the ID of each task as it runs and thus generate a usage profile.

5.4 Configuration

The final entries in the Configuration Table are used to support user-defined system calls and VRTX extensions. Refer to Chapter 4, Configuration and Initialization, for a diagram.

sys-TCREATE-addr and **sys-TDELETE-addr** are optional parameters that allow the user to perform special processing whenever tasks are created or deleted. If not required, supply a value of 0 for these parameters. Whenever such a routine is invoked, R1 contains a pointer to the TCB of the created (or deleted) task.

sys-TSWITCH-addr similarly, is an optional parameter that gives a user-supplied routine control whenever a context-switch is made from one task to another. If no special context-switching code is required, supply a value of 0. If a routine is specified, whenever it is invoked, R1 contains a pointer to the TCB of the old task and R2 contains a pointer to the TCB of the new task.

Appendix A

SYSTEM CALL SUMMARY

HUNTER
READY

The set of VRTX system calls, the input data (including the hexadecimal value of the function code to be supplied in R0), and the return data (shown in brackets) are illustrated in the following table.

Because the error code is always returned in R0 (except for the UI_EXIT and VRTX_GO calls where nothing is returned), this value is not shown in the table.

Task Management:

	input/[returned] data			
Mnemonic	R0	R1	R2	R3
SC_TCREATE	0000H	priority_&_ID	address	
SC_TDELETE	0001H	priority_or_ID		
SC_TSUSPEND	0002H	priority_or_ID		
SC_TRESUME	0003H	priority_or_ID		
SC_TPRIORITY	0004H	ID	priority	
SC_TINQUIRY	0005H	ID/[ID]	[priority]	R3:[status] R4:[TCB addr]
SC_LOCK	0020H			
SC_UNLOCK	0021H			

Memory Allocation:

	input/[returned] data			
Mnemonic	R0	R1	R2	R3
SC_GBLOCK	0006H	[address]	partition ID	
SC_RBLOCK	0007H	address	partition ID	
SC_PCREATE	0022H	partition ID	address	R3:size R4:blocksize
SC_PEXTEND	0023H	partition ID	address	size

Communication and Synchronization:

	input/[returned] data			
Mnemonic	R0	R1	R2	R3
SC_POST	0008H	address	message	
SC_PEND	0009H	address	time-out/ [message]	
SC_ACCEPT	0025H	address	[message]	
SC_QPOST	0026H	queue ID	message	
SC_QPEND	0027H	queue ID	time-out/ [message]	
SC_QACCEPT	0028H	queue ID	[message]	
SC_QCREATE	0029H	queue ID	count	
SC_QINQUIRY	002AH	queue ID	[message]	R4:[count]

Interrupt Support:

	input/[returned] data			
Mnemonic	R0	R1	R2	R3
UI_ENTER	0016H			
UI_EXIT	0011H	linkage pointer		

Real-Time Clock:

	input/[returned] data			
Mnemonic	R0	R1	R2	R3
SC_GTIME	000AH		[clock counter]	
SC_STIME	000BH		clock counter	
SC_TDELAY	000CH		delay interval	
SC_TSLICE	0015H			slice interval
UI_TIMER	0012H			

Initialization:

	input/[returned] data			
Mnemonic	R0	R1	R2	R3
VRTX_INIT	—			
VRTX_GO	0031H			

Appendix B

RETURN CODES

Upon return from a VRTX system call, a return code is generally returned in register R0. The following table lists the mnemonics, values (in hexadecimal notation), and meanings of all possible return codes.

R0	Mnemonic	Meaning	Affected Commands
0000H	RET_OK	Successful return	[All valid commands]
0001H	ER_TID	Task ID error	TCREATE, TDELETE, TSUSPEND, TRESUME, TPRIORITY, TINQUIRY
0002H	ER_TCB	No TCBS available	TCREATE
0003H	ER_MEM	No memory available	GBLOCK, PCREATE, PEXTEND, QCREATE
0004H	ER_NMB	Not a memory block	RBLOCK
0005H	ER_MIU	Mailbox in use	POST
0006H	ER_ZMW	Zero message	POST, QPOST
0009H	ER_ISC	Invalid system call	[Invalid commands]
000AH	ER_TMO	Time-out	PEND, QPEND
000BH	ER_NMP	No message present	ACCEPT, QACCEPT
000CH	ER_QID	Queue ID error	QPOST, QPEND, QACCEPT QCREATE, QINQUIRY
000DH	ER_QFL	Queue full	QPOST
000EH	ER_PID	Partition ID error	GBLOCK, RBLOCK, PCREATE, PEXTEND
000FH	ER_INI	Fatal init error	VRTX_INIT
0021H	ER_COM	Invalid component call	[Invalid commands] High byte does not equal zero

Appendix C

VECTOR TABLE AND TCB FORMATS

HUNTER
READY

VRTX uses two standard data structures for run-time information storage: the Vector Table defined by the 1750A architecture and the Task Control Blocks (TCBs) assigned to each task in the system. These data structures can be accessed by users, so for convenient reference, their formats are collected into this appendix.

C.1 Vector Table

The 1750A architecture defines a mechanism known as an 'interrupt vector table,' at reserved addresses 0020 through 003F (hexadecimal), that controls access to interrupt and trap handler routines. To avoid complicated system generation and configuration procedures, VRTX gives the user full control of this well-defined structure. Thus, the specification of user-defined interrupt and trap handlers is accomplished in an easy and straightforward manner.

Figure C-1, 1750A Vector Table and VRTX Pointers, is a simplified rendition of the 1750A vector table. Complete details are given in the MIL-STD-1750A (Notice 1) document.

Note that one of the BEX system call traps must be routed into VRTX by the user-defined vector table. The Instruction Counter for this purpose is at an offset of exactly 4 words into the kernel. Thus, if BEX 0 is chosen and the VRTX kernel is positioned at physical address 1000, the VRTX IC is specified as hex 1004 in the third word of the block referenced by the Executive Call service pointer.

Note also that the two words at addresses 0050 and 0051 are pointers, respectively, to the user-defined Configuration Table and to the area of data memory reserved for VRTX's storage requirements.

C.2 TCB Format

For each task in an ongoing VRTX environment, the system maintains a data structure known as a Task Control Block (TCB). Each TCB records all relevant context and state information for its associated task. The format of the 29-word (001DH) TCB block is shown in Figure C-2, Task Control Block.

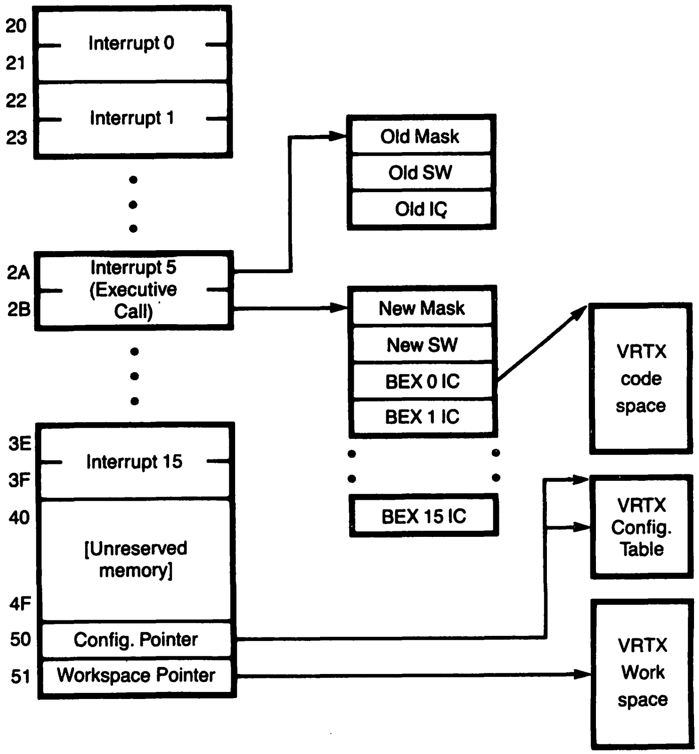


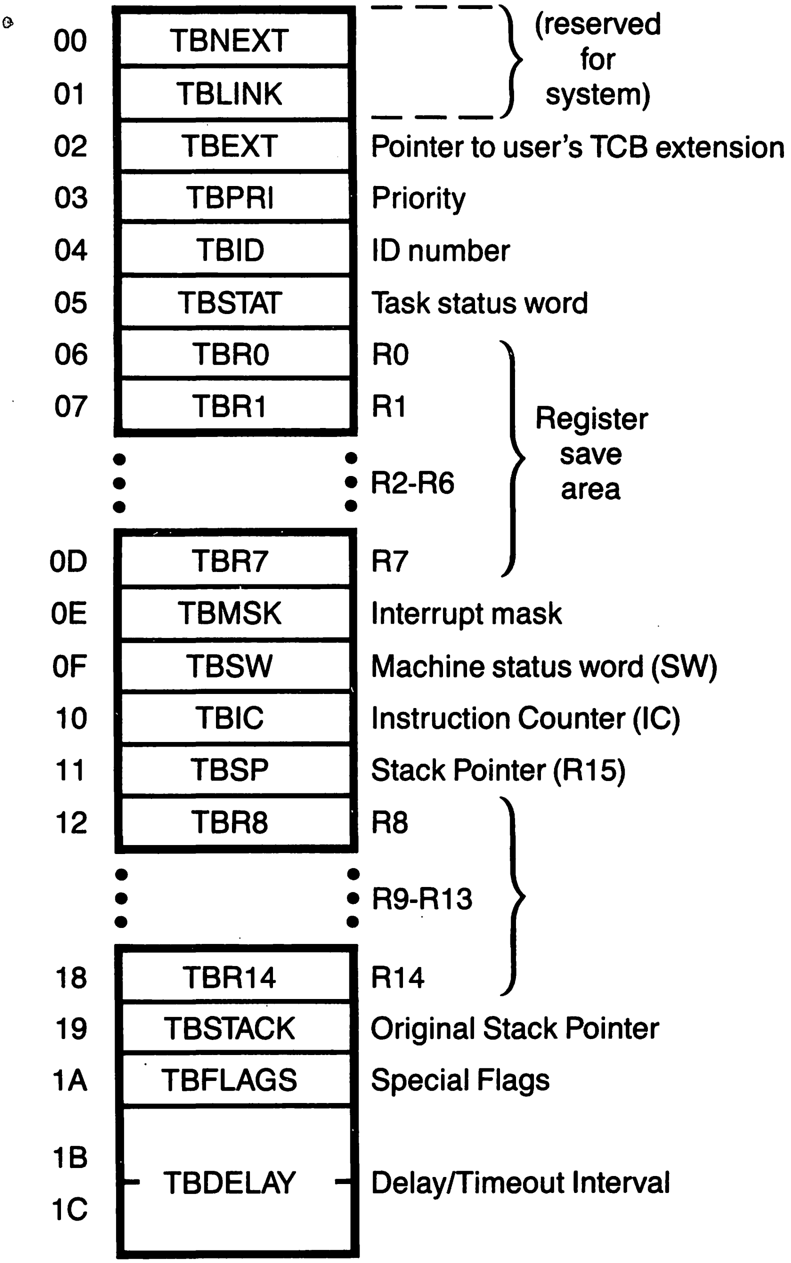
Figure C-1. 1750A Vector Table and VRTX Pointers

The status word at TCB offset 05 is the value returned by a Task Inquiry (SC_TINQUIRY) call. If the value of this word is zero, the associated task is ready to run. If the value is nonzero, the task has been suspended for one or more of the following reasons, as indicated by the bit settings:

```
bit: 0                    8  9 10 11 12 13 14 15
     +------------------------------------------+
R3 = |           0          |      status       |
     +------------------------------------------+
```

Bit	Reason for Suspension	Suspending Call
15	Explicitly suspended	SC_TSUSPEND
14	Suspended for message	SC_PEND
10	Suspended for task delay	SC_TDELAY*
9	Suspended on message queue	SC_QPEND

*Also set for SC_PEND and SC_QPEND when a time-out is in effect.



00	TBNEXT		(reserved for system)
01	TBLINK		
02	TBEXT	Pointer to user's TCB extension	
03	TBPRI	Priority	
04	TBID	ID number	
05	TBSTAT	Task status word	
06	TBR0	R0	Register save area
07	TBR1	R1	
•		R2-R6	
0D	TBR7	R7	
0E	TBMSK	Interrupt mask	
0F	TBSW	Machine status word (SW)	
10	TBIC	Instruction Counter (IC)	
11	TBSP	Stack Pointer (R15)	
12	TBR8	R8	
•		R9-R13	
18	TBR14	R14	
19	TBSTACK	Original Stack Pointer	
1A	TBFLAGS	Special Flags	
1B 1C	TBDELAY	Delay/Timeout Interval	

Figure C-2. Task Control Block

Appendix D

IMPLEMENTATION NOTES

HUNTER READY

D.1 Implementing Variable-Sized Stacks

Under some circumstances, users may wish to circumvent the usual allocation of fixed-size stacks to tasks, choosing instead to assign stacks of different sizes to different tasks. This can be accomplished when the task is created using the **sys-TCREATE-addr** vector described in Chapter 5, Support for User-Defined Extensions. This Configuration Table vector is set up to point to a short piece of user-supplied code that runs whenever a task is created. The code simply overwrites the two stack pointer values in the task's TCB (TBSP, TBSTACK) with new values that point to the top of an arbitrarily sized area of user-managed memory. VRTX manages the stack pointers in exactly the same way it normally handles them.

The Configuration Table parameter **user-stack-size** can be set to zero if user stacks are to be handled in this manner.

Appendix E

TASK RESCHEDULING

The **rescheduling procedure** involves traversing the active TCB chain to find the highest priority ready task. A **task switch** occurs if the task found has 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 state of the new task. Rescheduling does not always result in a task switch. Checking the active TCB chain for the highest priority task occurs after some VRTX system calls just before they return to the task level.

Since VRTX code is interruptible, the rescheduling procedure can be interrupted. Because interrupt service routines often affect the states of tasks, checking for the highest priority task may have to be restarted when interrupted to insure the highest priority task gets control.

If a task switch occurs, the **sys-TSWITCH-hook** is used to activate the TSWITCH routine. Occasionally the rescheduling procedure does not result in a task switch, in which case the TSWITCH routine is *not* used. Because the rescheduling procedure is interruptible, the TSWITCH routine may be activated more than once.

The VRTX calls listed below do not activate the rescheduling procedure, therefore none of them result in a task switch:

```
SC_TINQUIRY, SC_LOCK,
SC_GBLOCK,   SC_RBLOCK,   SC_PCREATE, SC_PEXTEND,
SC_ACCEPT,   SC_QACCEPT,  SC_QCREATE, SC_QINQUIRY,
SC_GTIME,    SC_STIME,    SC_TSLICE,
UI_ENTER
```

One call always causes a task switch to occur:

```
SC_TDELAY
```

The remaining calls go through the rescheduling procedure under certain circumstances (usually when the call suspends or deletes the current task, or readies a higher-priority task or an equal priority task that appears on the TCB chain before the current task).

Additionally, the following task management calls cause a task switch if the current task is affected or if any task with a higher priority is affected. For example, if a task switch with the same priority as the current task is created or resumed, the rescheduling procedure results in a task switch. Similarly, a suspend of the current task results in a task switch.

```
SC_TCREATE, SC_TDELETE, SC_TRESUME, SC_TSUSPEND,
  SC_TPRIORITY, SC_UNLOCK
```

The four communication calls cause a task switch if the current task suspends on an empty mailbox or queue, or if the posted message readies a higher priority task:

```
SC_POST, SC_PEND, SC_QPOST, SC_QPEND
```

Calls made from interrupt service routines (system level code) are special cases. The calls below cause VRTX to go through the rescheduling procedure but only if the interrupt service routine ends with the UI_EXIT call. In the case of nested interrupts, rescheduling occurs only when the outer nest interrupt handler completes and executes its UI_EXIT call.

The calls below cause a task switch if they ready a suspended task with higher priority than the task that was interrupted. This also applies to tasks of equal priority but whose TCB is on the TCB chain ahead of the TCB of the interrupted task.

```
UI_TIMER, SC_POST, SC_QPOST
```

INDEX

The following list contains only the most important references to each item. System calls are not listed specifically; consult pages 2-3, Appendix A, or the Table of Contents for these calls.

We'd like your comments

Hunter & Ready, Inc. attempts to provide documents that meet the needs of all VRTX users. 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 Hunter & Ready, Inc.

VRTX/1750 User's Guide #591613001

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 ______________________ Phone ____________

State ______________________ Zip ____________

HUNTER
READY

Thanks for your help!

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO.265 PALO ALTO,CA

POSTAGE WILL BE PAID BY ADDRESSEE

HUNTER & READY, Inc.
P.O. Box 60803
445 Sherman Avenue
Palo Alto, CA 94306-0803

HUNTER
READY