VRTX C USER'S GUIDE

HUNTER
READY

VRTX C

HUNTER
READY

VRTX

C Interface Library

User's Guide

Version 3
Document Number 592103001
November 1984

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

Table of Contents

HUNTER READY

List of Illustrations

List of Tables

Chapter 1

OVERVIEW OF THE VRTX C INTERFACE LIBRARY

HUNTER READY

1.1 Introduction

The VRTX C Interface Library offers C language users a convenient means of interfacing with VRTX, the Versatile Real-Time Executive. With this library, VRTX system calls available to the assembly language user are also available to the user implementing in C. (The only exceptions are the calls provided for user-supplied interrupt handlers and the calls provided for initialization.) The actual operations performed by VRTX are identical in assembly language and in C. By using the C Interface Library with VRTX, the C language can be extended to include multitasking without having to modify the C compiler. Since VRTX is available for use with all widely used 16-bit microprocessors, a programmer using C and a VRTX C Interface Library can write multitasking programs that are completely independent of particular processor architectures.

Hunter & Ready, Inc. provides an interface library for each supported compiler. Currently supported compilers are listed in the Hunter & Ready, Inc. Product Catalog. Since the C language is virtually the same for all compilers, this manual can serve as a reference for all versions of the C Interface Library. For specific C language references, refer to *The C Programming Language* by Kernighan and Ritchie.

The interface library contains a collection of routines that can be invoked as high-level language functions or procedures. A routine exists for each VRTX system call, and making the call is simply a matter of calling the corresponding function or procedure. Although they have the format of high-level language functions or procedures, these library routines are actually written in assembly language so they can reference particular registers and execute the software trap instruction (TRAP or INT) that is used to call VRTX. Their basic function is to transfer parameters from the stack, where the compiler puts them, to registers, where VRTX expects them; then they make the assembly language VRTX call described in the *VRTX User's Guide*. At execution time, calls to functions in the library are automatically translated into VRTX system calls. Documentation provided with the library explains the start-up procedures in more detail.

1.2 VRTX Functions

Each compiler has a different assembly language format for functions (which can be studied by examining the compiled version of a function), and all functions in that compiler's library follow this unique format. Each function in the library is responsible for one VRTX system call. Thus, each function performs the following operations:

* Saves registers.

* Takes parameters off the stack (where the compiler puts them) and puts them in registers (where VRTX expects them).

* Makes a VRTX call (thus the function **sc_tcreate** makes the VRTX call SC_TCREATE).

* Restores registers and exits from the function call.

The functions provided by VRTX can be organized into five categories:

* Task Management

* Memory Allocation

* Communication and Synchronization

* Real-Time Clock

* Character I/O

Along with the function name itself, input parameters are passed to VRTX as function arguments. This manual describes the specific format and meaning of each argument. There are no optional arguments in any of the VRTX calls. Therefore, omitting or adding an argument may cause serious problems. The order of the arguments, shown in Table 1-1, 'VRTX Functions,' must be strictly followed. In some cases, the argument is actually a pointer, specifying the address of a variable or an array. For example, the error argument is always passed as an address to VRTX.

VRTX error codes returned to the C user are identical to those returned by VRTX to the assembly language user. A return code, where applicable, is returned as the contents of an integer variable (**err**) specified by an input pointer argument (**&err**). If the call is successful, err returns a value of zero; otherwise, err returns one of the error codes. Some additional VRTX functions also return output results. Table 1-2, 'VRTX Arguments,' defines all input arguments used with VRTX.

VRTX may also call other silicon software components through the use of sc_call(). For further details, refer to the chapter titled 'Interfacing Software Components' in the *VRTX User's Guide* and Section 5.2, 'Call a Component,' in this manual.

Table 1-1. VRTX Functions

Task Management:

sc_tcreate(task,tid,pri,&err)	Task Create
sc_tdelete(tid/pri,code,&err)	Task Delete
sc_tsuspend(tid/pri,code,&err)	Task Suspend
sc_tresume(tid/pri,code,&err)	Task Resume
sc_tpriority(tid,pri,&err)	Task Priority Change
tcb = sc_tinquiry(info,tid,&err)	Task Inquiry
sc_lock()	Disable Rescheduling
sc_unlock()	Enable Rescheduling

Memory Allocation:

block = sc_gblock(pid,&err)	Get Memory Block
sc_rblock(pid,block,&err)	Release Memory Block
sc_pcreate(pid,paddr,psize,bsize,&err)	Create Memory Partition
sc_pextend(pid,paddr,psize,&err)	Extend Memory Partition

Communication and Synchronization:

sc_post(&mbox,msg,&err)	Post Message
msg = sc_pend(&mbox,timeout,&err)	Pend for Message
msg = sc_accept(&mbox,&err)	Accept a Message
sc_qpost(qid,msg,&err)	Post Message to Queue
msg = sc_qpend(qid,timeout,&err)	Pend for Message from Queue
msg = sc_qaccept(qid,&err)	Accept Message from Queue
sc_qcreate(qid,qsize,&err)	Create Message Queue
msg = sc_qinquiry(qid,&count,&err)	Queue Inquiry

Real-Time Clock:

`time = sc_gtime()`	Get Time
`sc_stime(time)`	Set Time
`sc_delay(timeout)`	Task Delay
`sc_tslice(ticks)`	Enable Round-Robin Scheduling

Character I/O:

`char = sc_getc()`	Get Character
`sc_putc(char)`	Put Character
`sc_waitc(char,&err)`	Wait Character

Component Management:

`sc_call(fcode,&pkt,&err)`	Call a Component

Table 1-2. VRTX Arguments

block	Pointer variable that holds the starting address of a block of memory.
bsize	Integer variable that holds the block size for a partition.
char	Integer variable that holds a character.
code	Integer variable to select either tid or pri as the first argument.
count	Pointer to an integer variable that holds the count of messages in a queue.
err	Integer variable that holds the return code.
fcode	Integer function code specified for silicon software components.
info	A three-element integer array declared as int info[3]; used by sc_tinquiry to return the task status information.

mbox	Pointer variable to be used as a mailbox.
msg	Pointer variable that receives a message.
paddr	Pointer variable that holds the starting address of a partition or partition extension.
pid	Integer variable that holds the ID number of a partition.
pkt	Pointer to a parameter packet defined for silicon software components.
pri	Integer variable that holds the priority of the task.
psize	Pointer variable that holds the size of the partition or partition extension specified in bytes.
qid	Integer variable that holds the ID of a queue.
qsize	Integer variable that holds the size of a queue.
task	Name of the C function to be run as a task.
tcb	Pointer variable that holds the address of the TCB.
ticks	Integer variable that holds the number of ticks required.
tid	Integer variable that holds the ID number for the task.
tid/pri	Either tid or pri (integer variable).
time	Long integer variable that holds the value of the clock.
timeout	Long integer variable that holds the time-out value.

1.3 Program Portability

C language programs written for VRTX can be compiled for the 68000 or 8086 and run without modification, provided that a few simple rules are followed in the declaration and use of variables.

The declaration of variables is important, because different implementations of the C language use different lengths for the integer and pointer variables. For example, when implemented for the 8086, C uses 16 bits for both integers and pointers. However, some implementations of C for the 68000 use 32 bits for both integers and pointers, while others use 16 bits for integers and 32 bits for pointers. Therefore, it is never assumed that pointers and integers are the same size. Rather, variables used as pointers are declared pointers (not integers).

The general rule for pointers is that they are guaranteed to be equal in length to the logical addressing range of the target machine; in practice, this value is then rounded up to the nearest 16 bits. Unfortunately, the choice between 16 and 32 bits for the size of integers is more arbitrary, and no general rule can be made.

1.4 Declarations of Functions

Functions can return a variety of data types. Unless indicated otherwise, C assumes that a function is returning an integer. Therefore, if a function is returning a pointer it must be explicitly declared:

```
char *sc_pend();
```

This declaration states that **sc_pend** is a function returning a pointer that points to a character. In the second example,

```
int *sc_pend();
```

sc_pend is declared as returning a pointer to an integer. The difference between these two becomes apparent when the object they point to is referenced.

1.5 Address Parameters

VRTX uses addresses extensively as input and output parameters for system calls. However, addresses are usually more difficult to work with in high-level languages than in assembly language. Consequently, users need to become familiar with the way address parameters are handled by the compiler.

In a **call-by-value** language like C a copy of the actual value of the parameter is passed to a function (for single-element parameters, that is; multi-element parameters, such as arrays and structures, are passed via their starting addresses). Consequently, address parameters must be explicitly indicated. In the C language, the pointer data type is used for address parameters.

Five input address parameters are used in VRTX and three addresses are returned as outputs:

INPUT ADDRESS PARAMETER	PARAMETER USED IN
task code address	sc_tcreate()
mailbox address	sc_post(), sc_pend(), sc_accept()
block address	sc_rblock()
partition address	sc_pcreate(), sc_pextend()
partition size	sc_pcreate(), sc_pextend()

OUTPUT ADDRESS PARAMETER	RETURNED BY
TCB and status address	sc_tinquiry()
block address	sc_gblock()
message	sc_pend(), sc_accept(), sc_qpend(),sc_qaccept(), sc_qinquiry()

In the following sections, we examine how each of these addresses is handled in C.

1.5.1 Task Code Addresses

In C, the address of a function can be passed implicitly. If the name of the function is used as a parameter, the compiler actually passes the address of the function (this implicit address convention is used with all multi-element structures). For example:

```
sc_tcreate(task,tid,pri,&err)
```

where **task** is the name of the function that is to run as a task.

1.5.2 Mailboxes

The VRTX intertask communication calls, sc_post(), sc_pend() and sc_accept() employ the **mailbox address** parameter. Mailboxes themselves are address-sized memory locations that are used to hold **messages** (see Section 1.5.3, 'Messages').

Mailboxes are defined as pointer variables. When making the VRTX call the user should pass the address of the mailbox; in C this means a pointer to a pointer:

```
sc_post(&mbox,msg,&err);
```

where **mbox** is the pointer-sized mailbox and &mbox is a pointer to the mailbox.

1.5.3 Messages

Messages are address-length parameters that are passed to the calls sc_post() and sc_qpost() and are returned from the calls sc_pend(), sc_accept(), sc_qpend(), sc_qaccept() and sc_qinquiry(). Of course, messages can actually be the addresses of larger data structures that contain the information being exchanged. For the purpose of this discussion, the message is the address of such a structure.

The fundamental problem with messages for high-level languages is type checking. The parameters of a VRTX system call must be declared as a certain type (integer, character, pointer to an array, pointer to a structure), and only variables of that type can be used as parameters. Unfortunately, in a typical application the message parameter may at various times be a pointer, an integer or a character. Moreover, even if a message is a pointer, it may be a pointer to an array, or to a structure, or to almost anything.

In C, however, messages are easy to work with, since type conversion (**casting**) can be performed at any time. Messages may be declared generally as pointer variables, but at times some messages may be assigned the value of a character or integer, simply by casting the character as a pointer. Thus, the user can declare the message parameter in a system call as a pointer, but can use integer or character messages at any time. For example:

```
char *msg;               /* declare msg a char ptr */

msg = (char *)7;         /* assign msg the integer */
                         /* value 7 */
```

1.5.4 TCB and Status Addresses

TCB and Status Block addresses are handled exactly like messages, except that the type is well defined. For example, in C the SC_TINQUIRY call has the following form:

```
tcb = sc_tinquiry(info,tid,&err)
```

After the call returns, the variable **tcb** holds the address of the task's TCB. The variable **tcb** is declared as a pointer to a structure that is organized as a set of fields corresponding to the TCB entries. The status information is returned in a 3-element integer array, info[3], pointed to by **info**. Although this is actually returned data, the parameter is passed as an input parameter and modified within the system call, because in C, functions can return only one value.

1.5.5 Block Addresses

Block addresses are easy to handle in C. VRTX calculates the block address and passes it back to the user when the sc_gblock() call is executed. The user must then pass this address to VRTX when executing the sc_rblock() call. In C, one simply assigns a pointer variable to the output of the function that implements the sc_gblock() call, for example:

```
block = sc_gblock(pid,&err);
```

1.5.6 Partition Addresses

Partitions are regions of memory from which blocks are dynamically allocated to tasks. Partitions themselves may be dynamically created and extended. In the two VRTX calls sc_pcreate() and sc_pextend(), one parameter is the absolute starting address of the partition.

In C this parameter is easy to set up. Define it as a pointer, then assign it an absolute address (i.e., an integer), using the built-in type conversion primitives. For example:

```
/*******************************************************/

char *paddr;            /* declare paddr to be a character */
                        /* pointer                         */
   .
   .
   .
paddr = (char *) 0xAE00; /* assign paddr the value        */
                         /* AE00 (hex), first             */
                         /* converting that integer       */
                         /* to a pointer                  */

sc_pcreate(pid,paddr,psize,bsize,&err); /* make the call */

/*******************************************************/
```

1.5.7 Partition Sizes

The two VRTX calls sc_pcreate() and sc_pextend() each take a parameter that specifies the size of the partition. In order that C programs can be executed on any implementation of VRTX, the interface library requires that the partition size be specified in byte units. The type of this parameter must span the entire addressing range of the machine. On virtually all C implementations, the type **char *** covers the whole addressing range.

The partition size should be cast type **char *** before calling the interface library. This will allow the program to run correctly on any CPU. The C code fragment below shows how this is done:

```
/************************************************************/

#define PARTID 1      /* partition ID number */
typedef struct {      /* type of each element in partition */
        int xxx;
        int yyy;
} elem;
elem part[100];       /* partition contains 100 elements */

crepart()
{
  int err;            /* error code returned by VRTX */
  sc_pcreate(PARTID, part, (char *)sizeof(part),
             sizeof(elem), &err);
}

/************************************************************/
```

Chapter 2
TASK MANAGEMENT

HUNTER
READY

2.1 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 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-grouped execution path through user code that demands the use of system resources. A task can be viewed as a thread or path of code that deals with one issue or system need. The act of dealing with that issue or need is collected or assembled into one task. In a multitasking 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 define the multitasking environment can have as many as 255 logically distinct active tasks, each tagged with a unique identification number. Any number of untagged tasks can also 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 autonomously 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.

In multitasking VRTX programs written in C, tasks are implemented as functions. Therefore, VRTX tasks have access to all the facilities of the C language that apply to functions, such as separate compilation and the ability to gather functions into libraries. Thus, a VRTX multitasking program written in C consists of a set of functions that are used as tasks and run concurrently, as well as a set of functions that are used as subroutines that execute sequentially.

If written reentrantly, a single C function may be run as any number of independent tasks, all executing from one copy of the C function. Also, if a C function is written reentrantly, it can be shared by any number of tasks as a common function (for example, a trig function).

2.1.1 Task Priority and Scheduling

When a task is created, it may be given a unique identification (ID) number from 1 to 255 (an ID number of zero indicates that no ID is assigned) and a priority level from zero to 255 (a priority level of zero indicates 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 programs that run under VRTX, the first task is created by the application initialization code. This task can create the other tasks in the system. As subsequent tasks are created, VRTX is called upon to initiate the **rescheduling procedure** and to put the highest priority task into execution. If another task is created that has higher priority than the initial task, it preempts the creator. For this reason, it is more straightforward to create all the tasks at initialization time.

VRTX is an event-driven operating system. 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 suspends.

* A higher priority task is ready to execute.

The executing task can explicitly suspend itself or can suspend waiting for an event. Events for which tasks wait include posting of a message, time elapse or special characters to arrive. The task remains suspended until the required event has occurred, at which time it is available for continued execution.

Tasks created of lower priority than the executing task are ready to run. Ready tasks run when all higher priority tasks are complete or when they suspend. Ready tasks with the same priority level can receive CPU control on a time-sliced, round-robin basis if time-slicing is explicitly enabled by the SC_TSLICE call. Any number of tasks can exist at the same priority level.

2.1.2 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 bursts. It is necessary for VRTX to maintain status information about the contents of active registers for all tasks 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.

A task is in an active state if it is executing, ready or suspended. If a task is dormant or inactive, the system has no knowledge of it even though its code may remain 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 completes or suspends, 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 pointer (see Figure 2-1, 'TCB Chain'). VRTX executes the first ready task in this chain.

In the case where several tasks have the same priority, a TCB inserted into the chain is placed ahead of the TCBs of any tasks at that priority level. The task whose TCB was most recently inserted into the TCB chain is therefore executed before any other tasks at the same priority level. Equal priority tasks are thus prioritized by 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 5.25, 'Enable Round-Robin Scheduling'). Among equal priority tasks, an optional round-robin scheduling may be enabled. At the end of a time-slice interval, TCBs of tasks with the same priority level as the executing task are rotated in the TCB chain. The next task in the chain is thus given a chance to execute.

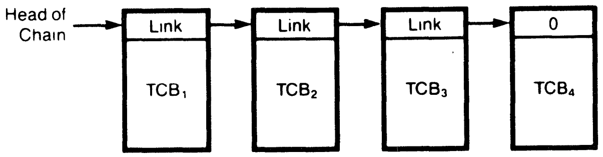

Figure 2-1. TCB Chain

When a task is deleted, it becomes dormant and its TCB is no longer associated with that task. Unused TCBs are linked together to form an inactive chain of available TCBs. When VRTX is initialized, all TCBs are located on the inactive chain. Except for the links themselves (TBNEXT in the TCB structure) and the initial stack pointer (TBSTACK), these TCBs are empty.

2.1.3 Task States and State Transitions

In a multitasking 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. (Only one task is executing at a time.)
Ready	The task is ready for execution but cannot gain control of the CPU until all higher priority tasks existing in the ready or executing state are either completed or suspended.
Suspended	The task suspends in mid-execution and is waiting to be readied by a system call or an event, such as waiting for a certain number of ticks to expire or a special character to arrive.
Dormant	The task is not initialized or its execution is complete (task deleted) and it is now idle. No TCB is assigned to it.

A task may suspend for any of the following reasons:

* A Task Suspend call, sc_tsuspend(), was issued specifying that task either by priority, ID number or via a self-suspend.

* The task suspends itself for a specified time interval using the sc_delay() call.

* The task is waiting for a message from another task or interrupt handler. It issues an sc_pend() or sc_qpend() call but no message is posted yet.

* The task issues an sc_waitc() call and is waiting for a special character to be sent from an I/O device.

* The task issues an sc_getc() call, but the input buffer maintained by VRTX is empty, so it is waiting for input from an I/O device.

* The task issues an sc_putc() call, but the output buffer is full, so it is waiting for output to an I/O device.

The status field in each TCB 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 is suspended for reasons indicated by the bit settings (see Section 5.22, 'Task Inquiry'). 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

Any call issued that leads to the current task's suspension causes unpredictable results 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, which readies a task that was either suspended by an sc_delay() call or timed out pending for a message.
* A message can be posted, with an sc_post() or sc_qpost() call, to a task that is waiting for a message.
* A special character can be sent from an Interrupt Service Routine (ISR) to a task that is suspended by an sc_waitc() call.
* Characters can be sent to the input buffer from an ISR. The tasks that suspend on an empty buffer are readied in the order in which they suspended.
* Characters can be retrieved from the output buffer by an ISR. The tasks that suspend on a full buffer are readied in the order in which they suspended.

Note: If a task has disabled interrupts and subsequently suspends, VRTX reenables interrupts for itself and other tasks. As soon as this task resumes execution, interrupts are again disabled.

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 deleted or suspended, the system has no tasks to run and is placed in an idle state, essentially halting its activity. Interrupts are enabled for this idle state, so the system remains capable of responding to interrupts. All task state transitions are diagrammed in Figure 2-2, 'Task State Transitions.'

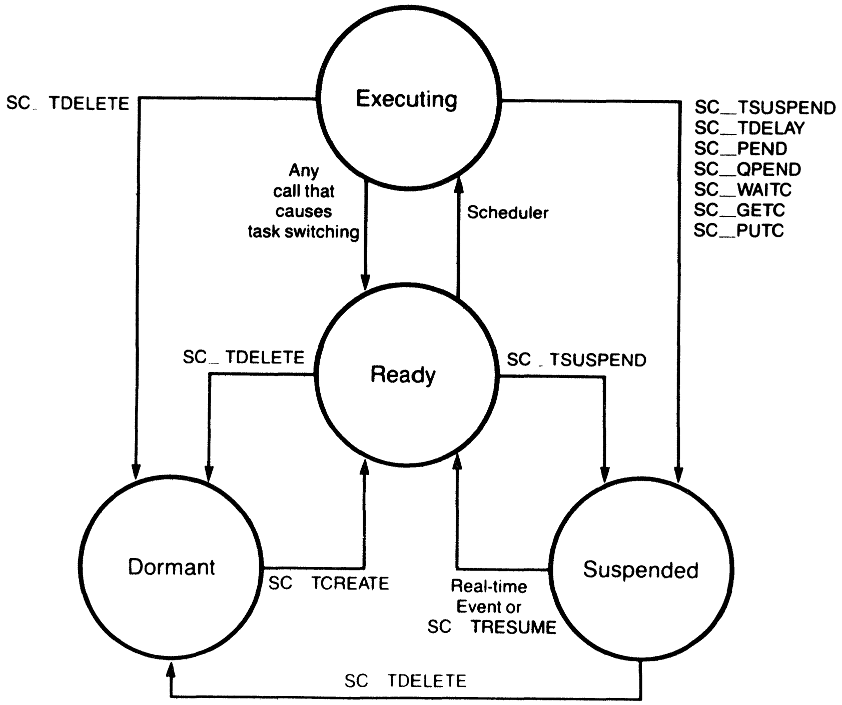

Figure 2-2. Task State Transitions

2.2 Multitasking Management Call Summary

The table below contains a summary of the system calls that control the multitasking environment. The set of VRTX calls with their arguments is illustrated. For detailed information on each of the calls, see Chapter 5, 'System Call Reference.'

The return code is always returned in err. This value is not shown in the table.

TABLE 2-1. SYSTEM CALL SUMMARY

TASK MANAGEMENT:	Ret	Arg1	Arg2	Arg3	Arg4	Arg5
sc_tcreate(task,tid,pri,&err)		ptr	int	int	&int	
sc_tdelete(tid/pri,code,&err)		int	int	&int		
sc_tsuspend(tid/pri,code,&err)		int	int	&int		
sc_tresume(tid/pri,code,&err)		int	int	&int		
sc_tpriority(tid,pri,&err)		int	int	&int		
tcb = sc_tinquiry(info,tid,&err)	ptr	ptr	int	&int		
sc_lock()						
sc_unlock()						

The following example uses all eight of the task management calls:

```
/***********************************************************/

main()  {
int taskl(),err;                  /* must declare task 1 */

sc_tcreate(taskl,1,2,&err);       /* ID = 1   pri = 2 */
if (err != 0)   _error("tcreate",err);

sc_tsuspend(1,0,&err);            /* by ID, ID = 1    */
if (err != 0)   _error("tsuspend",err);

sc_tpriority(1,3,&err);           /* change pri to 3  */
if (err != 0)   _error("tpriority",err);

sc_tresume(3,'A',&err);           /* by pri , pri = 3 */
if (err != 0)   _error("tresume",err);

sc_lock();                        /* disable rescheduling */

sc_unlock();                      /* enable rescheduling */

sc_tdelete(0,0,&err);             /* delete self      */
}

taskl()  {
char *tcb;
int info[3],err,own_id,own_pri,status;

tcb = sc_tinquiry(info,0,&err); /* inquire about self */
if (err != 0)   _error("tinquiry",err);
own_id  = info[0];
own_pri = info[1];
status  = info[2];

}

/***********************************************************/
```

Chapter 3

MEMORY MANAGEMENT AND INTERTASK COMMUNICATION

HUNTER & READY

3.1 Memory

The memory map of a VRTX-based system consists of the following modules:

* **VRTX code**: the VRTX PROM set. (Note that VRTX may be placed in dynamic memory, perhaps even loaded into memory from disk.)

* **The VRTX Workspace**: contains system variables, TCBs and stacks.

* **The user load module**: the software package the user is responsible for developing, assembling, linking and placing in the execution environment.

* **VRTX-managed user memory**: one or more partitions or pools of memory blocks that can be dynamically acquired and released by the user.

* Optional Hunter & Ready components such as IOX, FMX or TRACER.

* Optional user-supplied components.

Figure 3-1, 'Memory Organization,' is an overview of the entire memory organization of a VRTX system.

The **user load module** holds the user's application code and any user-defined, system-level code. In addition, the user load module contains the Interrupt or Exception Vector Table, the Configuration Table, and any static variables associated with the user application or with system code. See Section 3.3.1, 'Mailboxes'. The shading in Figure 3-1 indicates what can be burned into ROM; everything else must exist in dynamic read/write memory.

The **VRTX Workspace** contains the system variables, the TCBs, a stack for each task in the system, control structures for message queues and control structures for **VRTX-managed user memory**. VRTX is responsible for setting up and managing the stacks and for initializing and managing the TCB chain. The VRTX-managed user memory consists of a number of partitions or chunks of memory which may be noncontiguous. Each partition is subdivided into one or more fixed-sized blocks of memory that can be allocated dynamically to tasks. The following section describes how VRTX manages user memory and its own Workspace.

3.1.1 Memory Allocation

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

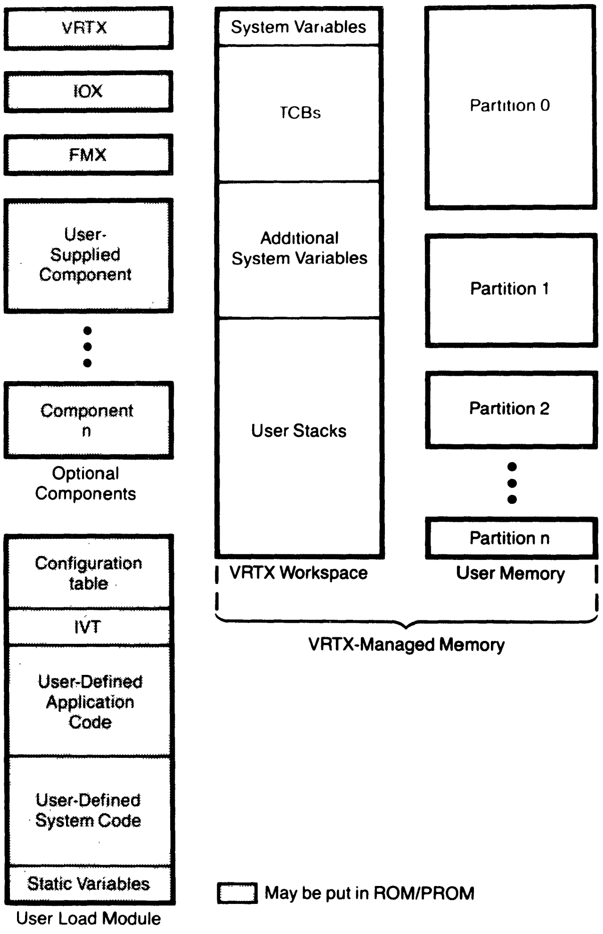

Figure 3-1. Memory Organization

Two main approaches to memory allocation have been used by multitasking executives: **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, 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 starting with one single chunk of memory and repeatedly splitting existing units in half. When the request is for a unit larger than any of those currently available, the system attempts to combine a smaller unit with its buddy into a 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 is 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 on the order of the requests. This usually cannot be anticipated in a real-time system. This design introduces an element of unpredictability into the total system behavior beyond that of the external environment. This additional unpredictability is unsatisfactory in real-time systems. Real-time systems cannot tolerate a memory system that works only some of the time.

The designers of VRTX felt static allocation was too restrictive, but memory compaction led to unacceptable indeterminacy and imposed too much system overhead. Thus, the VRTX memory allocation mechanism is a compromise between the two memory allocation schemes. VRTX gives every task a fixed-sized 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 noncontiguous chunks that make up the actual physical organization of memory. Each partition of user memory has blocks of a fixed size set when that partition is created. The **user-stack-size** is set via a parameter in the Configuration Table.

Dynamic memory allocation for user 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 stack should be. The VRTX Workspace must be large enough to contain VRTX system variables, one TCB for each active task and a stack for every task in the system. 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.

2. Whenever a task is created, VRTX automatically allocates a stack to the task. This stack can store local variables and is allocated in the VRTX Workspace. This allocation may be bypassed if the user wants to manage stacks via a hook invoked at task switch time.

3. The call sc_pcreate() is used to define a contiguous partition of user memory. Parameters passed with the call specify the starting address, size and standard block size of the partition. The sc_gblock() call can then be used to acquire blocks of memory from the new partition. This call can be repeated until all blocks in this partition are allocated. The sc_rblock() call is executed in order to release a block of memory back to the partition. A task's blocks are not automatically released when the task is deleted, therefore an sc_rblock() call should be made to release all blocks before the task is deleted.

4. The call sc_pextend() is 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.

Since all memory blocks are the same size within a partition, no fragmentation results from dynamic memory allocation; consequently, no memory compaction is required. Figures 3-2, 'System Memory Managed by VRTX' and 3-3, 'User Memory Managed by VRTX,' show how memory is subdivided.

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

3.2 Memory Allocation Call Summary

The table below contains a summary of the system calls that allow applications to obtain and return blocks of memory from a specified partition. The summary also includes the VRTX calls that create and extend partitions. These calls do not go through the rescheduling procedure. The set of VRTX calls with their arguments is illustrated. For detailed information on each of the calls, see Chapter 5, 'System Call Reference.'

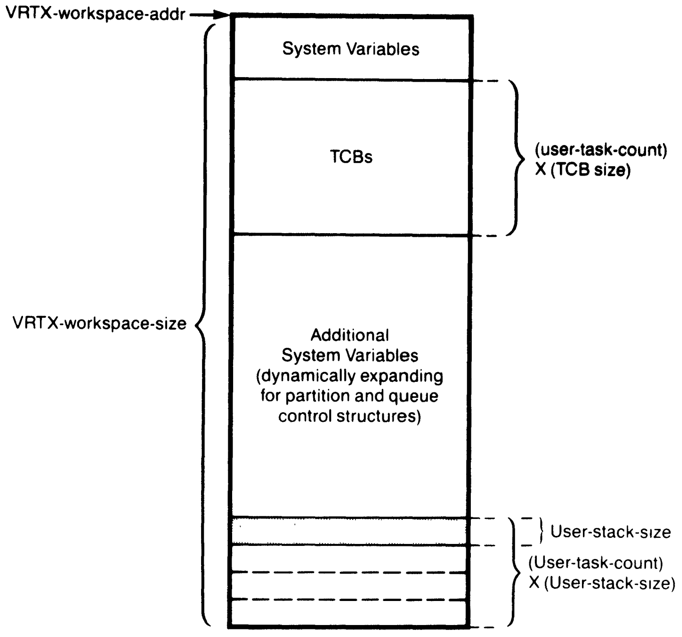
VRTX-workspace-addr
System Variables
TCBs
(user-task-count) X (TCB size)
VRTX-workspace-size
Additional System Variables (dynamically expanding for partition and queue control structures)
User-stack-size
(User-task-count) X (User-stack-size)

Figure 3-2. System Memory Managed by VRTX

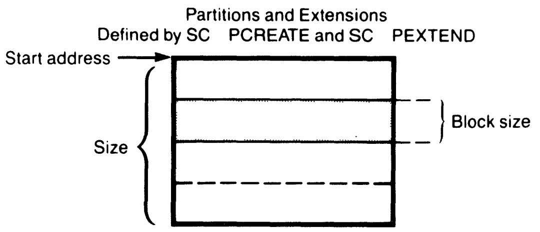
Partitions and Extensions Defined by SC PCREATE and SC PEXTEND
Start address
Size
Block size

Figure 3-3. User Memory Managed by VRTX

The return code is always returned in err. This value is not shown in the table.

TABLE 3-1. SYSTEM CALL SUMMARY

MEMORY ALLOCATION:	Ret	Arg1	Arg2	Arg3	Arg4	Arg5
block = sc_gblock(pid,&err)	ptr	int	&int			
sc_rblock(pid,block,&err)		int	ptr	&int		
sc_pcreate(pid,paddr,psize,bsize,&err)		int	ptr	ptr	int	&int
sc_pextend(pid,paddr,psize,&err)		int	ptr	ptr	&int	

The following example uses the four memory allocation calls:

```
/**********************************************************/

main()  {
char *sc_gblock(),*block,*paddr,*psize;
int pid,err,bsize;

pid = 1;
psize = (char*)0x1000;
bsize = 64;

paddr = (char*)0x30000;
sc_pcreate (pid,paddr,psize,bsize,&err);
paddr = (char*)0x50000;
sc_pextend(pid,paddr,psize,&err);

block = sc_gblock(pid,&err);
if(err != 0)  error("gblock",err);

block[2] = 'B';

sc_rblock(pid,block,&err);
if(err != 0)  error("rblock",err);

}

/**********************************************************/
```

3.3 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 pointer-sized, nonzero messages via VRTX-controlled structures known as **mailboxes** and **queues**. These messages can be pointers to larger messages if the communicating tasks are so designed.

3.3.1 Mailboxes

A mailbox is a user-defined location residing in user read/write memory that allows tasks to pass pointer-sized, nonzero messages. A mailbox is simply a pointer-sized variable that the user should allocate in memory. VRTX does not create mailboxes.

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

```
sc_post()        Post a Message
sc_pend()        Pend for a Message
sc_accept()      Accept a Message
```

A transmitting task deposits the message in a specified mailbox 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 receives the message and remains in the ready state. The message location is reset to zero by VRTX when the message is received. The application should empty the mailbox at initialization time by initializing it to zero when allocated by the user. If the mailbox value is zero (holds no message), a task attempting to receive a message with an sc_pend() call suspends until a message arrives. Additionally, a nonzero time-out value may be specified that allows the task to resume execution if no message arrives during that time period. Conversely, if the mailbox value is nonzero (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() does not suspend if there is no message present; instead an error code is returned.

More than one task can wait at the same mailbox by issuing sc_pend() calls with the same mailbox address. The highest priority task receives the message and is placed in the ready state when a message is sent to that mailbox. If a task pending at a mailbox is explicitly suspended it may still receive a message, although it does not resume execution until it is explicitly resumed.

With these calls, the user can easily implement mutual exclusion and **resource locking**, as well as standard intertask communication. Resource locking is implemented when all the tasks attempting to use a resource pend at the same mailbox. As each task finishes with the resource, it sends a message to that mailbox to enable the next task.

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

3.3.2 Queues

VRTX provides five additional calls to 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. Queues are system-managed structures referenced by a queue ID (qid) assigned by the user with the sc_qcreate() call and are 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 suspends. As with mailboxes, tasks attempting to accept messages from an empty queue are not suspended; an error code is returned instead. Tasks pended at a queue are readied by incoming messages in priority order, not in the order they were pended. A high-priority task that pends at an empty queue after a low priority task has pended, receives the first message sent to that queue in accordance with VRTX's priority management philosophy. Information about the queues can be obtained by issuing the sc_qinquiry() call which returns the number of messages in the queue and, without removing it from the queue, the message at the head of the queue. Note that a queue of length 1 behaves in a similar manner to a mailbox.

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 Status Inquiry`

Queues can 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 (such as a line printer) is assigned a specific queue, the length of which is determined by the number of resources included in that type (such as the number of printers on the system). All tasks attempting to use a resource of a given 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 readying of tasks ensures that several tasks waiting to use a resource receive the resource in order of priority.

3.4 Communication and Synchronization Call Summary

The table below contains a summary of the system calls used to exchange pointer-sized, nonzero messages via mailboxes, and the calls used for more elaborate exchanges via message queues. Only the posting and pending calls go through the rescheduling procedure and may result in a task switch. The set of VRTX calls with their arguments is illustrated. For detailed information on each of the calls, see Chapter 5, 'System Call Reference.'

The return code is always returned in err. This value is not shown in the table.

TABLE 3-2. SYSTEM CALL SUMMARY

COMMUNICATION AND SYNCHRONIZATION:	Ret	Arg1	Arg2	Arg3	Arg4	Arg5
sc_post(&mbox,msg,&err)		&ptr	ptr	&int		
msg = sc_pend(&mbox,timeout,&err)	ptr	&ptr	long	&int		
msg = sc_accept(&mbox,&err)	ptr	&ptr	&int			
sc_qpost(qid,msg,&err)		int	ptr	&int		
msg = sc_qpend(qid,timeout,&err)	ptr	int	long	&int		
msg = sc_qaccept(qid,&err)	ptr	int	&int			
sc_qcreate(qid,qsize,&err)		int	int	&int		
msg = sc_qinquiry(qid,&count,&err)	ptr	int	&int	&int		

The following example uses all three of the intertask communication calls:

```
/*********************************************************/

char  *mbox[10] = 0;  /* note that mbox is declared */
                      /* external and as a pointer */

task6()  {
char *sc_pend(),*sc_accept(),*msg;
long timeout;  int err;
timeout = (long) 0;
msg = sc_accept(&mbox[2],&err);

msg = sc_pend(&mbox[2],timeout,&err);

}


task7()  {
char *msg;
int err;

msg = (char *)'A';
sc_post(&mbox[2],msg,&err);
if(err != 0)  error("post",err);

}

/*********************************************************/
```

> **Note:** Use of sc_post(&mbox[2],'A',&err) is not portable. Because msg must be the size of a pointer, 'A' may not necessarily be of pointer size.

The following example uses all five of the message queuing calls:

```
/************************************************************/

main() {
int qid,qsize,err;
qid = 1;
qsize = 10;
sc_qcreate(qid,qsize,&err);
if(err != 0) error("Qcreate",err);

}

task 1 ()  {
char *msg,*sc_qaccept(),*sc_qpend();
long timeout; int qid, count, err;
timeout = (long) 0;
qid = 1;
msg = sc_qinquiry(qid,&count,&err)
msg = sc_qaccept(qid,&err);

msg = sc_qpend(qid,timeout,&err)

}

task 2 ()  {
char  *msg;
int qid,err;
qid = 1;
msg = (char *) 64;
sc_qpost(qid,msg,&err);

}

/************************************************************/
```

3.5 Communication with Other Components

A special system call, known as sc_call(), provides a general mechanism for communicating service requests to components other than VRTX. See Section 5.2, 'Call a Component,' for further details on this particular call.

TABLE 3-3. SYSTEM CALL SUMMARY

COMPONENT MANAGEMENT:

sc_call(fcode,&pkt,&err)

Ret	Arg1	Arg2	Arg3	Arg4	Arg5
	int	&pkt	&int		

```
/*******************************************************/

task_IO(){
int fcode, *err;
IODEFPK disk;           /* structure defined in IOX */

fcode = IOFRMDEV;       /* function code for remove */
                        /* device - 0022H           */
disk.IODOPTS = 0;       /* reserved, must = 0       */
disk.IODRMBZ = 0;
disk.IODDVID = 10;      /* specify device ID        */

sc_call(fcode,&disk,&err);
}

/*******************************************************/
```

Chapter 4

INTERRUPT SUPPORT

HUNTER READY

4.1 Real-Time Clock Support

VRTX operates quite satisfactorily without a real-time clock. Nevertheless, support for a real-time clock is fully integrated into VRTX, adding to user application code the following collection of system calls:

`sc_gtime()`	Get Time
`sc_stime()`	Set Time
`sc_delay()`	Task Delay
`sc_tslice()`	Enable Round-Robin Scheduling

At the interrupt level, the user must define a minimal clock service routine, which merely handles the mechanics of dealing with a particular clock device (such as an 8253 Interval Timer or a 9513 Timing Controller) and, on a periodic basis, issues a UI_TIMER system call to VRTX. This command informs VRTX that a time interval (or tick) has occurred. Even in target environments without a real-time clock device, a timer of sorts (basic, but sufficient for task delay and round-robin scheduling) may be implemented by issuing the UI_TIMER command on a somewhat regular basis from other interrupt handlers.

4.2 Real-Time Clock Support Calls

The table below is a summary of the system calls used to support a real-time clock. The set of VRTX functions with their arguments is illustrated in the following table. For detailed information on each of the calls, see Chapter 5, 'System Call Reference.'

TABLE 4-1. SYSTEM CALL SUMMARY

```
REAL-TIME CLOCK:            +-----+-----+-----+-----+-----+-----+
                            |Ret  |Arg1 |Arg2 |Arg3 |Arg4 |Arg5 |
                            |-----|-----|-----|-----|-----|-----|
time = sc_gtime()           |long |     |     |     |     |     |
sc_stime(time)              |     |long |     |     |     |     |
sc_delay(timeout)           |     |long |     |     |     |     |
sc_tslice(ticks)            |     |int  |     |     |     |     |
                            +-----+-----+-----+-----+-----+-----+
```

The following example uses all four of the real-time clock support functions:

```
/*********************************************************/

main()  {
long int sc_gtime(),time;

sc_stime((long)  0);
time = sc_gtime();
/*  'time' should now have the value zero */

time = (long) 100;
sc_stime(time);
sc_delay(time);

time = sc_gtime();
/* 'time' should now have the value 200  */

sc_tslice(10); /* enable time slicing, slice size=10 ticks */

sc_tslice(0);  /* disable time slicing */

}

/*********************************************************/
```

4.3 Character I/O Support

VRTX also provides fully integrated support for a single character-oriented input/output device. The following calls allow a task to read a character from a single character-oriented I/O device, to write a character to that device and to suspend itself until a particular character is received.

`sc_getc()`	Get Character
`sc_putc()`	Put Character
`sc_waitc()`	Wait for Special Character

At the interrupt level, the user must define interrupt service routines to handle the mechanics of communicating with a particular device (such as a USART or parallel I/O device).

4.4 Character I/O Support Calls

The table below is a summary of the system calls used to provide character I/O support. The set of VRTX functions with their arguments is illustrated in the following table. For detailed information on each of the calls, see Chapter 5, 'System Call Reference.'

TABLE 4-2. SYSTEM CALL SUMMARY

CHARACTER I/O:	Ret	Argl	Arg2	Arg3	Arg4	Arg5
char = sc_getc()	char					
sc_putc(char)		char				
sc_waitc(char,&err)		char	&int			

The following example uses all three of the character I/O functions:

```
/************************************************************/

main()  {
int err,c;

sc_waitc(0x03,&err);  /* wait for control C */

if(err != 0)  error("waitc",err);

c = getchar();

}

getchar()  {
int  c;
c = sc_getc();
sc_putc(c);  /*  echo the character  */
return(c);
}

/************************************************************/
```

Chapter 5

SYSTEM CALL REFERENCE

This chapter contains a complete description of all the system calls used in VRTX. They are given in alphabetical order for easy reference. For each call, the following items are listed:

* **Mnemonic name** of the call,

* A brief **Description** of the call's function and operation,

* The **Calling Sequence** of the system call,

* An **Example**,

* A description of the **Arguments**,

* A list of the possible **Return Codes** returned in integer variable err.

5.1 sc_accept - Accept a Message

This call obtains a pointer-sized, nonzero message from a specified mailbox. Unlike sc_pend(), this call does not suspend the caller if no message is present; the error code ER_NMP is returned immediately. This call does not go through the rescheduling procedure.

```
CALLING
SEQUENCE:    msg = sc_accept(&mbox,&err)

EXAMPLE:     msg = sc_accept(&mbox[3],&err)
```

```
+------------------------------------------------------------+
|                                                            |
|  INPUT:    mbox      Pointer variable to be used as a       |
|                      mailbox.                              |
|                                                            |
|                                                            |
|  OUTPUT:   err       Integer variable that holds the        |
|                      return code.                          |
|                                                            |
|            msg       Pointer variable that receives a       |
|                      message.                              |
|                                                            |
+------------------------------------------------------------+
```

RETURN CODES

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

5.2 sc_call - Call a Component

This call allows a silicon software component to be called via VRTX. The parameter **fcode** specifies the component ID in its upper byte and a command code in its lower byte. The parameter **pkt** is defined for a specific call by the specified software silicon component. For further details consult the chapter titled 'Interfacing Software Components' in the *VRTX User's Guide*.

```
CALLING
SEQUENCE:    sc_call(fcode,&pkt,&err)

EXAMPLE:     sc_call(IOXXXX,&iopkt,&err)
```

```
+---------------------------------------------------+
|                                                   |
|  INPUT:    fcode    Integer function code specified for  |
|                     silicon software components.  |
|                                                   |
|            pkt      Pointer to a parameter packet |
|                     defined for silicon software  |
|                     components.                   |
|                                                   |
|                                                   |
|  OUTPUT:   err      Integer variable that holds the  |
|                     return code.                  |
|                                                   |
+---------------------------------------------------+
```

RETURN CODES

0000H	RET_OK	Successful return.
0020H	ER_CVT	Component Vector Table not defined in Configuration Table.
0021H	ER_COM	Undefined component.
0022H	ER_OPC	Undefined opcode for this component.

5.3 sc_delay - Task Delay

This call suspends execution of the calling task for a specified number of clock ticks. The delay value stored in the TCB is not an absolute delay, but a relative increment from the delay value of tasks already delayed. This call results in a task switch.

> **Note:** Many compilers do not carry more than six characters of significance in their labels. Therefore, the VRTX function name SC_TDELAY has been renamed at the C level to sc_delay in order to avoid possible conflicts with sc_tdelete.

```
CALLING
SEQUENCE:   sc_delay(timeout)

EXAMPLE:    sc_delay(8)
```

```
+----------------------------------------------------------+
|                                                          |
|  INPUT:    timeout  Long integer variable that holds     |
|                     the number of ticks required.        |
|                                                          |
|                                                          |
|  OUTPUT:            None.                                |
|                                                          |
+----------------------------------------------------------+
```

5.4 sc_gblock - Get Memory Block

This call obtains a memory block from one of the partitions of memory blocks managed by VRTX.

```
CALLING
SEQUENCE:   block = sc_gblock(pid,&err)

EXAMPLE:    my_heap = sc_gblock(part_id,&err)
```

```
+-----------------------------------------------------------+
|                                                           |
|   INPUT:    pid       Integer variable that holds the     |
|                       ID number of a partition.           |
|                                                           |
|                                                           |
|   OUTPUT:   block     Pointer variable that holds the     |
|                       starting address of a block of      |
|                       memory.                             |
|                                                           |
|             err       Integer variable that holds the     |
|                       return code.                        |
|                                                           |
+-----------------------------------------------------------+
```

RETURN CODES

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

5.5 sc_getc - Get Character

With this call, a task obtains the next sequential character from the supported I/O device. If the 64-byte buffer of received characters is empty, the calling task suspends until a character is received. It does not echo the character onto the output device. This call initiates the rescheduling procedure if no character is present.

```
CALLING
SEQUENCE:   char = sc_getc()

EXAMPLE:    lnbuf[i] = sc_getc()

+----------------------------------------------------------+
|                                                          |
|  INPUT:             None.                                |
|                                                          |
|                                                          |
|  OUTPUT:   char     Integer variable that holds a        |
|                     character.                           |
|                                                          |
+----------------------------------------------------------+
```

5.6 sc_gtime - Get Time

This call obtains the current value, as a count of ticks, of the system clock counter. It does not go through the rescheduling procedure.

```
CALLING
SEQUENCE:   time = sc_gtime()

EXAMPLE:    clock_val = sc_gtime()
```

```
+----------------------------------------------------+
|                                                    |
|  INPUT:              None.                         |
|                                                    |
|                                                    |
|  OUTPUT:  time       Long integer variable that holds the |
|                      value of the clock.           |
|                                                    |
+----------------------------------------------------+
```

5.7 sc_lock - Disable Task Rescheduling

This call prevents task rescheduling until an explicit sc_unlock() call is issued. The task that issues the sc_lock() function call retains processor control even though other higher priority tasks may be ready to run.

The calls sc_lock() and sc_unlock() are used in pairs. An internal 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.

The maximum nest count supported is 255 minus the maximum number of nested ISRs. For example, if a system has prioritized interrupts with a maximum depth of three ISRs, the maximum lock/unlock nesting is 255-3=252.

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

WARNING

After this call has been issued, the user should not issue any VRTX calls that could lead to suspending the current task. This event causes unpredictable results.

```
CALLING
SEQUENCE:   sc_lock()

+--------------------------------------------------+
|                                                  |
|  INPUT:            None.                         |
|                                                  |
|                                                  |
|  OUTPUT:           None.                         |
|                                                  |
+--------------------------------------------------+
```

5.8 sc_pcreate - Create Memory Partition

This call defines the characteristics of a partition of contiguous memory that is managed by the VRTX kernel. Associated with each such partition is an ID number and a default block size. Successive sc_gblock() requests then use this ID number to obtain blocks of memory of default size from this partition.

```
CALLING
SEQUENCE:   sc_pcreate(pid,paddr,psize,bsize,&err)

EXAMPLE:    sc_pcreate(part_id,part_adr,part_size,256,&err)
```

```
+--------------------------------------------------------+
|                                                        |
|  INPUT:    pid      Integer variable that holds the    |
|                     ID number of a partition.          |
|                                                        |
|            paddr    Pointer variable that holds the    |
|                     starting address of a partition    |
|                     or partition extension.            |
|                                                        |
|            psize    Pointer variable that holds the    |
|                     size of the partition or partition |
|                     extension specified in bytes.  psize |
|                     must be greater than or equal to   |
|                     block size.                        |
|                                                        |
|            bsize    Integer variable that holds the    |
|                     block size for a partition.  bsize |
|                     cannot equal zero.                 |
|                                                        |
|                                                        |
|  OUTPUT:   err      Integer variable that holds the    |
|                     return code.                       |
|                                                        |
+--------------------------------------------------------+
```

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.
```

5.9 sc_pend - Pend for Message

This call obtains a pointer-sized, nonzero message from a specified mailbox. If no message is posted at the specified mailbox, the calling task suspends until a message becomes available to the calling task. When several tasks are waiting on the same mailbox, the task highest (in priority on the active TCB chain) will receive the message.

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 4.1, 'Real-Time Clock Support.') A task switch occurs if the mailbox is empty.

```
CALLING
SEQUENCE:   msg = sc_pend(&mbox,timeout,&err)

EXAMPLE:    msg = sc_pend(&mbox[3],100L,&err)
```

INPUT:	mbox	Pointer variable to be used as a mailbox.
	timeout	Long integer variable that holds the time-out value.
OUTPUT:	err	Integer variable that holds the return code.
	msg	Pointer variable that receives a message.

RETURN CODES

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

5.10 sc_pextend - Extend Memory Partition

This call extends a previously defined memory partition to encompass an additional range of memory locations. In conjunction with sc_pcreate(), this call defines memory partitions that span noncontiguous chunks of memory within an address space.

The block size used for a partition extension is identical to that originally defined by sc_pcreate().

```
CALLING
SEQUENCE:   sc_pextend(pid,paddr,psize,&err)

EXAMPLE:    sc_pextend(part_id,xpart_adr,xpart_size,&err)
```

```
+--------------------------------------------------------+
|                                                        |
|  INPUT:    pid      Integer variable that holds the   |
|                     ID number for the partition.       |
|                                                        |
|            paddr    Pointer variable that holds the   |
|                     starting address of a partition    |
|                     or partition extension.            |
|                                                        |
|            psize    Pointer variable that holds the   |
|                     size of the partition or partition |
|                     extension specified in bytes.  psize |
|                     must be greater than or equal to   |
|                     block size.                        |
|                                                        |
|                                                        |
|  OUTPUT:   err      Integer variable that holds the   |
|                     return code.                       |
|                                                        |
+--------------------------------------------------------+
```

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.
```

5.11 sc_post - Post a Message

This call is used to post a pointer-sized, nonzero message to a specified mailbox. Since zero is used to indicate the mailbox is empty, messages of zero are not allowed. This call results in a task switch when a task with a priority higher than the calling task was pended on that mailbox.

If a task is pended on a mailbox the next message posted to that mailbox is immediately allocated to the task, bypassing being saved in the mailbox itself.

```
CALLING
SEQUENCE:   sc_post(&mbox,msg,&err)

EXAMPLE:    sc_post(&mbox[3],msg,&err)
```

```
+------------------------------------------------------------+
|                                                            |
|  INPUT:    mbox      Pointer variable to be used as        |
|                      a mailbox.                            |
|                                                            |
|            msg       Pointer variable that receives a      |
|                      message.                              |
|                                                            |
|                                                            |
|  OUTPUT:   err       Integer variable that holds the       |
|                      return code.                          |
|                                                            |
+------------------------------------------------------------+
```

RETURN CODES

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

5.12 sc_putc - Put Character

With this call, a task specifies the next character to transmit to the supported I/O device. If the 64-byte buffer of characters to transmit is full, the calling task suspends until the buffer is available (i.e., one character is transmitted). This call initiates the rescheduling procedure if the transmit buffer is full.

```
CALLING
SEQUENCE:   sc_putc(char)

EXAMPLE:    sc_putc('V')

+----------------------------------------------------------+
|                                                          |
|  INPUT:    char     Integer variable that holds a        |
|                     character.                           |
|                                                          |
|                                                          |
|  OUTPUT:            None.                                |
|                                                          |
+----------------------------------------------------------+
```

5.13 sc_qaccept - Accept Message from Queue

This call obtains a pointer-sized, nonzero message from a specified queue. Unlike sc_qpend(), this call does not suspend the caller if no message is present, but returns the error code ER_NMP immediately. No message is returned if the call is unsuccessful. This call does not go through the rescheduling procedure.

```
CALLING
SEQUENCE:   msg = sc_qaccept(qid,&err)

EXAMPLE:    msg = sc_qaccept(que_id,&err)
```

```
+---------------------------------------------------------+
|                                                         |
|  INPUT:   qid      Integer variable that holds the      |
|                    ID of a queue.                       |
|                                                         |
|                                                         |
|  OUTPUT:  err      Integer variable that holds the      |
|                    return code.                         |
|                                                         |
|           msg      Pointer variable that receives a     |
|                    message.                             |
|                                                         |
+---------------------------------------------------------+
```

RETURN CODES

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

5.14 sc_qcreate - Create Message Queue

This call creates a message queue whose size cannot exceed available VRTX Workspace. There is an upper limit on the number of messages that can be enqueued at any given time, as specified by the size count. The queue is managed by VRTX in a 'first-in/first-out' (FIFO) manner. This call does not initiate the rescheduling procedure.

```
CALLING
SEQUENCE:   sc_qcreate(qid,qsize,&err)

EXAMPLE:    sc_qcreate(que_id,24,&err)
```

```
+------------------------------------------------------+
|                                                      |
|  INPUT:   qid      Integer variable that holds the   |
|                    ID of a queue.                    |
|                                                      |
|           qsize    Integer variable that holds the   |
|                    size of a queue (1 to 255         |
|                    inclusive).                       |
|                                                      |
|                                                      |
|  OUTPUT:  err      Integer variable that holds the   |
|                    return code.                      |
|                                                      |
+------------------------------------------------------+
```

RETURN CODES

0000H	RET_OK	Successful return.
0003H	ER_MEM	No memory available; insufficient VRTX Workspace.
000CH	ER_QID	Queue ID error; ID number already assigned.

5.15 sc_qinquiry - Queue Inquiry

This call obtains a count of messages waiting in a queue. If the count is nonzero, the actual contents of the head-of-queue message (the message to be given to the next sc_qpend() or sc_qaccept() request) is returned to the caller, without being extracted from the queue.

Although the caller is given a copy of the first message, the message remains queued. The calling program still needs to make the sc_qpend() or sc_qaccept() call to remove the message.

This function can be used at both task and ISR levels. This call does not go through the rescheduling procedure.

Note: If the return code is nonzero, the message returned is invalid.

```
CALLING
SEQUENCE:   msg = sc_qinquiry(qid,&count,&err)

EXAMPLE:    msg = sc_qinquiry(que_id,&cnt,&err)
```

```
+---------------------------------------------------------+
|                                                         |
|  INPUT:   qid      Integer variable that holds the      |
|                    ID of a queue.                       |
|                                                         |
|                                                         |
|  OUTPUT:  err      Integer variable that holds the      |
|                    return code.                         |
|                                                         |
|           msg      Pointer variable that receives a     |
|                    message.                             |
|                                                         |
|           count    Pointer to an integer variable       |
|                    that holds the count of messages     |
|                    in a queue.                          |
|                                                         |
+---------------------------------------------------------+
```

RETURN CODES

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

5.16 sc_qpend - Pend for Message from Queue

This call obtains a pointer-sized, nonzero message from a specified queue. If the specified queue is currently empty, the calling task suspends until a message is posted at that queue. Care should be taken to have fewer than 256 tasks pended on a single queue simultaneously; this will cause unpredictable results.

This call can specify an optional time-out value. 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 4.1, 'Real-Time Clock Support.' No message is returned if the call is unsuccessful. A task switch occurs if the queue is empty.

```
CALLING
SEQUENCE:   msg = sc_qpend(qid,timeout,&err)

EXAMPLE:    msg = sc_qpend(que_id,100L,&err)
```

```
+--------------------------------------------------------+
|                                                        |
|  INPUT:    qid       Integer variable that holds the   |
|                      ID of a queue.                    |
|                                                        |
|            timeout   Long integer variable that holds  |
|                      the time-out value.               |
|                                                        |
|                                                        |
|  OUTPUT:   err       Integer variable that holds the   |
|                      return code.                      |
|                                                        |
|            msg       Pointer variable that receives a  |
|                      message.                          |
|                                                        |
+--------------------------------------------------------+
```

RETURN CODES

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

5.17 sc_qpost - Post Message to Queue

This call posts a pointer-sized, nonzero message to a specified queue. This call results in a task switch when a task with a priority higher than the calling task was pended on that queue.

If a task is pended on the queue, the message is immediately posted to that task, bypassing being saved in the queue.

```
CALLING
SEQUENCE:   sc_qpost(qid,msg,&err)

EXAMPLE:    sc_qpost(que_id,my_msg,&err)
```

INPUT:	qid	Integer variable that holds the ID of a queue.
	msg	Pointer variable that receives a message.
OUTPUT:	err	Integer variable that holds the return code.

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.

5.18 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 a task is deleted.

```
CALLING
SEQUENCE:   sc_rblock(pid,block,&err)

EXAMPLE:    sc_rblock(part_id,my_heap,&err)
```

```
+------------------------------------------------------+
|                                                      |
|  INPUT:   pid      Integer variable that holds the   |
|                    ID number of a partition.         |
|                                                      |
|           block    Pointer variable that holds the   |
|                    starting address of a block of    |
|                    memory.                           |
|                                                      |
|                                                      |
|  OUTPUT:  err      Integer variable that holds the   |
|                    return code.                      |
|                                                      |
+------------------------------------------------------+
```

RETURN CODES

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

5.19 sc_stime - Set Time

This call sets the current value, as a count of ticks, of the system clock. The system resets this value to zero at initialization. This call does not go through the rescheduling procedure.

```
CALLING
SEQUENCE:   sc_stime(time)

EXAMPLE:    sc_stime(clock_val)
```

```
+--------------------------------------------------------+
|                                                        |
|  INPUT:    time      Long integer variable that holds  |
|                      the value of the clock.           |
|                                                        |
|                                                        |
|  OUTPUT:             None.                             |
|                                                        |
+--------------------------------------------------------+
```

5.20 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. A value of zero 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.

This call results in a task switch if the new task's priority is higher than or equal to that of the calling task.

```
CALLING
SEQUENCE:   sc_tcreate(task,tid,pri,&err)

EXAMPLE:    sc_tcreate(task4,23,7,&err)
```

```
+-----------------------------------------------------------+
|                                                           |
|  INPUT:    task     Name of the C function to be run      |
|                     as a task.                            |
|                                                           |
|            tid      Integer variable that holds the ID    |
|                     number for the task (1 to 255         |
|                     inclusive, or 0 if no ID is to be     |
|                     assigned).                            |
|                                                           |
|            pri      Integer variable that holds the       |
|                     priority of the task (0 to 255        |
|                     inclusive, with 0 being highest       |
|                     priority).                            |
|                                                           |
|                                                           |
|  OUTPUT:   err      Integer variable that holds the       |
|                     return code.                          |
|                                                           |
+-----------------------------------------------------------+
```

Note: A task ID of zero, while legal, is a special case. A task with ID of zero 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 zero when issued by a different task. These calls can be made for a task with an ID of zero only by the task itself. This anonymity is sometimes useful in systems for security applications.

RETURN CODES

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

5.21 sc_tdelete - Task Delete

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 becomes available for reuse. This call results in a task switch if the current task is deleted.

```
CALLING
SEQUENCE:   sc_tdelete(tid/pri,code,&err)

FORMAT 1:   Delete all tasks of a specified priority.
                     sc_tdelete(pri_1,'A',&err)
            The return code has the value RET_OK even
            if there are no tasks with the specified
            priority.

FORMAT 2:   Delete a task with a specified ID number.
                     sc_tdelete(your_tid,0,&err)

FORMAT 3:   Delete self (i.e., delete calling task).
                     sc_tdelete(0,0,&err)
```

```
+------------------------------------------------------------+
|                                                            |
|  INPUT:     tid/pri   Either tid or pri (integer           |
|                       variable).                           |
|                                                            |
|             code      Integer variable used to select      |
|                       either tid or pri as the first       |
|                       argument.  0 indicates tid; 'A'      |
|                       indicates pri.                       |
|                                                            |
|                                                            |
|  OUTPUT:    err       Integer variable that holds the      |
|                       return code.                         |
|                                                            |
+------------------------------------------------------------+
```

RETURN CODES

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

5.22 sc_tinquiry - Task Inquiry

This system call obtains priority and status information about a particular task, specified by ID number, along with a pointer to the task's TCB. A task can use the sc_tinquiry() call to determine its own task ID number, priority level and TCB address. This call never goes through the rescheduling procedure and therefore never results in a task switch.

```
CALLING
SEQUENCE:   tcb = sc_tinquiry(info,tid,&err)

FORMAT 1:   Get information about a task with a given
            ID number.
                     tcb = sc_tinquiry(info,your_tid,&err)

FORMAT 2:   Get information about one's self (i.e.,
            the calling task).
                     tcb = sc_tinquiry(info,0,&err)

+----------------------------------------------------------+
|                                                          |
|  INPUT:    tid      Integer variable that holds the ID   |
|                     number for the task (1 to 255        |
|                     inclusive, or 0 if no ID is to be    |
|                     assigned).                           |
|                                                          |
|  OUTPUT:   tcb      Pointer variable that holds the      |
|                     address of the TCB.                  |
|                                                          |
|            info     A three-element integer array        |
|                     declared as int info[3]; used to     |
|                     return the task status information.  |
|                        info[0] = ID number               |
|                        info[1] = priority                |
|                        info[2] = status variable of TCB  |
|                                                          |
|            err      Integer variable that holds the      |
|                     return code.                         |
+----------------------------------------------------------+
```

This call can be made from a task or an ISR level. If the call is made from an ISR, and no task ID is specified (task ID of zero), the information returned describes the current interrupted task. If the sc_tinquiry() is made during user-initialization before any tasks are created, the data returned is invalid.

RETURN CODES

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

If the return code is nonzero, the values in info are not valid.

If the value of the status variable is zero, the associated task is ready to run. If the status variable is nonzero, the task has been suspended for one or more of the following reasons, as indicated by the bit setting:

```
          bit: 15   . . .    6 5 4 3 2 1 0
              ---------------------------
      info[2] = |      0      |   status  |
              ---------------------------
```

Bit	Reason for Suspension	Suspending Call
0	Explicitly suspended	sc_tsuspend()
1	Suspended for message	sc_pend()
2	Suspended for input	sc_getc()
3	Suspended for output	sc_putc()
4	Awaiting special character	sc_waitc()
5	Suspended for task delay	sc_delay()*
6	Suspended on message queue	sc_qpend()

*Also set for sc_pend() and sc_qpend() when a time-out is in effect.

5.23 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 the same priority. This call results in a task switch when the new priority of the affected task is higher than or equal to that of the calling task. Note that the sc_tpriority() call does not affect the status of a task; in other words, a suspended task remains suspended even if its priority is changed.

```
CALLING
SEQUENCE:   sc_tpriority(tid,pri,&err)

FORMAT 1:   Change the priority of a task with a specified
            ID number.
                     sc_tpriority(your_tid,new_pri,&err)

FORMAT 2:   Change the priority of the calling task.
                     sc_tpriority(0,new_pri,&err)
```

INPUT:	tid	Integer variable that holds the ID number for the task (1 to 255 inclusive, or 0 if no ID is to be assigned).
	pri	Integer variable that holds the priority of the task (0 to 255 inclusive, with 0 being highest priority).
OUTPUT:	err	Integer variable that holds the return code.

RETURN CODES

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

5.24 sc_tresume - Task Resume

This call resumes the execution of one or more tasks previously suspended by an sc_tsuspend() call. This call initiates the rescheduling procedure.

> **Note:** A task with ID equal to zero cannot be explicitly resumed. (An ID of zero cannot be specified in this call.)

```
CALLING
SEQUENCE:   sc_tresume(tid/pri,code,&err)

FORMAT 1:   Resume all tasks of a specified priority.
                     sc_tresume(pri_1,'A',&err)
            The return code has the value RET_OK even
            if there are no tasks with the specified
            priority.

FORMAT 2:   Resume a task with a specified ID number.
                     sc_tresume(your_tid,0,&err)
```

```
+-------------------------------------------------------------+
|                                                             |
|  INPUT:     tid/pri   Either tid or pri (integer            |
|                       variable).                            |
|                                                             |
|             code      Integer variable used to select       |
|                       either tid or pri as the first        |
|                       argument.  0 indicates tid; 'A'       |
|                       indicates pri.                        |
|                                                             |
|                                                             |
|  OUTPUT:    err       Integer variable that holds the       |
|                       return code.                          |
|                                                             |
+-------------------------------------------------------------+
```

RETURN CODES

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

5.25 sc_tslice - Enable Round-Robin Scheduling

This call enables round-robin scheduling of equal priority tasks under VRTX. When time-slicing is in effect and VRTX is notified of a clock tick via receipt of a UI_TIMER call, VRTX records which task is in control. If the same task is in control when the time-slicing interval elapses, then the task suspends. Its TCB is put at the end of its priority group on the ready chain. Round-robin scheduling becomes disabled if the sc_tslice() call specifies a zero interval.

All groups of equal-priority tasks are subject to time-slicing. For example, three tasks at priority 5 and six tasks at priority 10 all undergo time-slicing.

When time-slicing is in effect, a task which suspends for any reason is put at the end of its priority group on the active chain.

```
CALLING
SEQUENCE:   sc_tslice(ticks)

EXAMPLE:    sc_tslice(7)
```

INPUT:	ticks	Integer variable that holds the number of ticks to comprise the time-slicing interval for round-robin scheduling. Use 0 as the time-slicing interval to disable round-robin scheduling.
OUTPUT:		None.

5.26 sc_tsuspend - Task Suspend

This call suspends one or more tasks. The TCB of each affected task remains on the active chain, but the explicitly-suspended flag in the TCB is set. A task that suspends in this manner will not resume execution until an sc_tresume() call is issued. This call initiates the rescheduling procedure if the current task suspends.

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

```
CALLING
SEQUENCE:   sc_tsuspend(tid/pri,code,&err)

FORMAT 1:   Suspend all tasks of a specified priority.
                      sc_tsuspend(pri_1,'A',&err)
            The return code has the value RET_OK even
            if there are no tasks with the specified
            priority.

FORMAT 2:   Suspend a task with a specified ID number.
                      sc_tsuspend(your_tid,0,&err)

FORMAT 3:   Suspend self (i.e., suspend calling task).
                      sc_tsuspend(0,0,&err)
```

```
+-----------------------------------------------------------+
|                                                           |
|  INPUT:     tid/pri  Either tid or pri (integer           |
|                      variable).                           |
|                                                           |
|             code     Integer variable used to select      |
|                      either tid or pri as the first       |
|                      argument.  0 indicates tid; 'A'      |
|                      indicates pri.                       |
|                                                           |
|                                                           |
|  OUTPUT:    err      Integer variable that holds the      |
|                      return code.                         |
|                                                           |
+-----------------------------------------------------------+
```

RETURN CODES

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

5.27 sc_unlock - Enable Task Rescheduling

This call reenables normal VRTX task rescheduling, cancelling the effect of a single previously issued sc_lock() call. If scheduling is already enabled, this call has no effect. The calls sc_lock() and sc_unlock() are used in pairs. An internal 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 call goes through the rescheduling procedure when the nesting count becomes zero.

The maximum nest count supported is 255 minus the maximum number of nested ISRs. For example, if a system has prioritized interrupts with a maximum depth of three ISRs, the maximum lock/unlock nesting is 255-3=252.

```
CALLING
SEQUENCE:   sc_unlock()
```

```
+--------------------------------------------------+
|                                                  |
|  INPUT:            None.                         |
|                                                  |
|                                                  |
|  OUTPUT:           None.                         |
|                                                  |
+--------------------------------------------------+
```

5.28 sc_waitc - Wait for Special Character

With this call, a user task can act as a watchdog for a particular character, such as a break or a CONTROL-C, that might need special processing. The calling task suspends until the specified character is received from the supported I/O device . The character does not get placed in the VRTX input buffer. VRTX permits only one sc_waitc() request to be active in the system. The rescheduling procedure is always initiated.

```
CALLING
SEQUENCE:    sc_waitc(char,&err)

EXAMPLE:     sc_waitc(0x03,&err)

             (03 is the Hex code for CONTROL-C)
```

```
+----------------------------------------------------------+
|                                                          |
|  INPUT:    char     Integer variable that holds a        |
|                     character.                           |
|                                                          |
|                                                          |
|  OUTPUT:   err      Integer variable that holds the      |
|                     return code.                         |
|                                                          |
+----------------------------------------------------------+
```

RETURN CODES

0000H	RET_OK	Successful return.
0008H	ER_WTC	Previous sc_waitc() request already in progress.

Appendix A

SYSTEM CALL SUMMARY

HUNTER READY

The set of VRTX functions with their arguments is illustrated in the following table.

TASK MANAGEMENT:

	Ret	Arg1	Arg2	Arg3	Arg4	Arg5
`sc_tcreate(task,tid,pri,&err)`		ptr	int	int	&int	
`sc_tdelete(tid/pri,code,&err)`		int	int	&int		
`sc_tsuspend(tid/pri,code,&err)`		int	int	&int		
`sc_tresume(tid/pri,code,&err)`		int	int	&int		
`sc_tpriority(tid,pri,&err)`		int	int	&int		
`tcb = sc_tinquiry(info,tid,&err)`	ptr	ptr	int	&int		
`sc_lock()`						
`sc_unlock()`						

MEMORY ALLOCATION:

	Ret	Arg1	Arg2	Arg3	Arg4	Arg5
`block = sc_gblock(pid,&err)`	ptr	int	&int			
`sc_rblock(pid,block,&err)`		int	ptr	&int		
`sc_pcreate(pid,paddr,psize,bsize,&err)`		int	ptr	ptr	int	&int
`sc_pextend(pid,paddr,psize,&err)`		int	ptr	ptr	&int	

COMMUNICATION AND SYNCHRONIZATION:

	Ret	Arg1	Arg2	Arg3	Arg4	Arg5
`sc_post(&mbox,msg,&err)`		&ptr	ptr	&int		
`msg = sc_pend(&mbox,timeout,&err)`	ptr	&ptr	long	&int		
`msg = sc_accept(&mbox,&err)`	ptr	&ptr	&int			
`sc_qpost(qid,msg,&err)`		int	ptr	&int		
`msg = sc_qpend(qid,timeout,&err)`	ptr	int	long	&int		
`msg = sc_qaccept(qid,&err)`	ptr	int	&int			
`sc_qcreate(qid,qsize,&err)`		int	int	&int		
`msg = sc_qinquiry(qid,&count,&err)`	ptr	int	&int	&int		

REAL-TIME CLOCK:

	Ret	Arg1	Arg2	Arg3	Arg4	Arg5
time = sc_gtime()	long					
sc_stime(time)		long				
sc_delay(timeout)		long				
sc_tslice(ticks)		int				

CHARACTER I/O:

	Ret	Arg1	Arg2	Arg3	Arg4	Arg5
char = sc_getc()	char					
sc_putc(char)		char				
sc_waitc(char,&err)		char	&int			

COMPONENT MANAGEMENT:

	Ret	Arg1	Arg2	Arg3	Arg4	Arg5
sc_call(fcode,&pkt,&err)		int	&pkt	&int		

Appendix B

RETURN CODES

HUNTER
READY

Upon return from a VRTX function, integer variable err contains the return code. The following table lists the mnemonics, values and meanings of all possible return codes.

err	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
0007H	ER_BUF	Buffer full	rxchr
0008H	ER_WTC	WAITC in progress	waitc
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
0020H	ER_CVT	Component Vector Table not defined in Configuration Table	call
0021H	ER_COM	Undefined component	call
0022H	ER_OPC	Undefined opcode for this component	call

INDEX

HUNTER
READY

The following list does not contain references to specific system calls. To find these, refer to Chapter 5, Appendix A or the Table of Contents.

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 C User's Guide #592103001

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