Vx960™

5.0.3 TRANSITION GUIDE

Intel Corporation

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

Order No. 517913

Contents

Vx960 4.0.2 to 5.0.3 Transition Guide

1. INTRODUCTION

Release 5.0.3 of the Vx960 real-time operating system and development environment offers major enhancements over the previous release, 4.0.2. Changes include the following:

- an extensively rewritten kernel providing a significant increase in performance and functionality
- a highly optimized network
- a new file system
- SCSI support
- new booting options
- additional support for the 8960SA/SB processor
- additional support for the 8960KA/KB processor

The beginning section of this guide discusses 5.0.3 systems application compatibility and conversion. Be sure to read the following section:

2. PORTING APPLICATIONS TO 5.0.3

Applications developed on 4.0.2 Vx960 will require minimal changes to be compatible with Vx960 5.0.3. This section itemizes these minimal yet important changes and should be reviewed carefully for their potential effect on pre-existing applications.

Subsequent sections expand on the information covered in these sections and discuss Vx960's new features and optimizations:

3. **KERNEL ENHANCEMENTS**
   - Improved context switching time
   - New scheduling algorithms
   - New semaphore types and options
   - Improved error status handling
   - Message queues
4. **I/O SYSTEM**
   - Improvements in *select*( )
   - SCSI support
5. **LOCAL FILE SYSTEMS**
   - New DOS-compatible file system
   - New raw disk file system
   - Changes in block-device driver implementation
6. **NETWORK ENHANCEMENTS**
   - Optimizations
   - Serial-Line Interface Protocol (SLIP)
   - Security Feature
7. **SHELL CHANGES**
   - Changes in task referencing
   - Changes in information reporting
8. **DEVELOPMENT ENVIRONMENT**
   - ANSI header support
   - Support for source-level debugging with VxGDB
9. **CONFIGURATION AND BOOTING**
   - Booting from Vx960
   - Boot ROM compression
   - New INCLUDE options
10. **DIRECTORY AND FILE REORGANIZATION**
   - Directory changes
   - Renamed libraries
   - New libraries
   - Unsupported material
11. **NEW MODULES AND ROUTINES**
   - Summary listing of new modules and routines

2. PORTING APPLICATIONS TO 5.0.3

2.1 Compiling

2.1.1 Applications Must Be Recompiled

Before loading applications developed under 4.0.2, review the changes outlined below in **2.2 Summary of User-Interface Changes** which may affect the compatibility of your application.

2.1.1.1 Object Compatibility

Regardless of whether code adjustments are necessary, 5.0.3 and 4.0.2 are not object-compatible; applications developed under 4.0.2 must be recompiled with 5.0.3 header files.

2.1.1.2 GNU/960 R1.3

This release was built with Release 1.3 of the GNU/960 toolset from Intel. Development of code to run on top of Vx960 must be done with the same release. Since Vx960 5.0.3 uses GNU1.3, all applications must be recompiled with GNU1.3. If you do not have GNU/960 R1.3, contact your local Intel sales office.

Install the GNU/960 R1.3 toolset before you install or use the Vx960 Release 5.0.3. Vx960 does not install correctly if your GNU/960 R1.3 tape has not been installed. Your G960BASE environment variable must point to the GNU/960 directory for your host. For complete details on installing GNU/960 R1.3 and setting your environment variables, see your GNU/960 R1.3 documentation.

2.1.2 Host Development Tree

With Vx960 4.0.2 it was necessary to maintain separate and redundant development trees for use from host to host. This was required because the makefiles differed in the invocation of the compiler, assembler, and loader, using the native tools for different hosts. Also, certain Vx960 host tools in /usr/vx/bin (e.g., hex and xsym) were different object modules for different hosts.

At the time of installation, a host is selected and appropriate makefiles are created for that host.

The directory /usr/vx/bin has been reorganized into /usr/vx/bin/sun3, /usr/vx/bin/-sun4, etc., with the appropriate versions of tools in each. The makefiles have been changed to invoke Vx960 tools out of the appropriate bin directory. You may also want to put the appropriate bin directory in your UNIX shell search path. On Sun systems the tool arch is useful for this. For example, you might add the following to your .cshrc:

```
set arch=`arch`
setenv PATH .:/usr/vx/bin/$arch:...
```

2.2 Summary of User-Interface Changes

In implementing the many changes included in release 5.0.3, a major concern was to keep the user interface as compatible as possible with the previous release, so that applications developed under 4.0.2 could be up and running under 5.0.3 with few, if any, modifications to existing code. Incompatibilities have been confined to situations where no other recourse was feasible without sacrificing functionality and efficient and consistent design.

Interface changes reflect a wide range of issues: from the major new features, enhancements, and methodologies discussed elsewhere in this document to the minor changes necessary to comply with ANSI requirements. Wherever possible, routines and options which are now outmoded are still included with the system to maintain backward compatibility.

This section serves as a guide to converting applications developed under the previous release to work with Vx960 5.0.3. Of the following tables, Table 1 lists routines that have been deleted and Table 2 lists routines that are now obsolete. Table 3 is a quick reference to the routines that have changed, and includes cross-references to subsequent pages that discuss these changes in greater detail.

Table 1. Modules and Routines Deleted in 5.0.3

Module/Routine	Reason/Substitute
fioFormat( )	Use *fioFormatV*( ) with *varargs*.
fioStdIn( )	Use STD_IN (0).
fioStdOut( )	Use STD_OUT (1).
fioStdErr( )	Use STD_ERR (2).
nfsAllUnmount( )	Obsolete
nfsMntDump( )	Obsolete
pathLib	Internal use only.
ramMkfs( )	Use *ramDevCreate*( ) + *rt11FsMkfs*( ) or *dosFsMkfs*( ) (see page 13).
selectDelaySet( )	Select no longer polls.
sysMemProbe( )	Use *vxMemProbe*( ).
taskTcbX( )	TCB extension (tcbX) no longer exists.
wdTick( )	Subsumed by *tickAnnounce*( ).

Table 2. Obsolete Routines in 5.0.3 *

Routine	Reason/Substitute
semCreate( )	Use *semBCreate*( ), *semCCreate*( ), *semMCreate*( ).
semClear( )	Use *semTake*( ) with NO_WAIT.
taskACWGet( )	Obsolete. Use *taskRegsGet*( ).
taskPCWGet( )	Obsolete. Use *taskRegsGet*( ).
taskTCWGet( )	Obsolete. Use *taskRegsGet*( ).
taskACWSet( )	Obsolete. Use *taskRegsGet*( ).
taskCreate( )	Renamed *taskInit*( ).
taskPCWSet( )	Obsolete. Use *taskRegsGet*( ).
taskTCWSet( )	Obsolete. Use *taskRegsGet*( ).

* These routines remain in the standard system for backward compatibility.

Table 3. **Routines Changed in 5.0.3**

Routine	Reason	Transparent[†]	Page
scanf( )	Floating-point field specs ANSI compatible.	no	7
sscanf( )	Floating-point field specs ANSI compatible.	no	7
fscanf( )	Floating-point field specs ANSI compatible.	no	7
printf( )	Returns character count instead of VOID.	yes	7
fprintf( )	Returns character count instead of OK.	probably	7
fdprintf( )	Returns character count instead of VOID.	yes	7
printErr( )	Returns character count instead of VOID.	yes	7
sprintf( )	Return value excludes null termination.	no	7
symTblCreate( )	Symbol access time decreased by hashing.	no	8
symTblInit( )	Symbol access time decreased by hashing.	no	8
kernelInit( )	No longer uses trap number.	no	8
taskRegsGet( )	Takes pointer to REG_SET structure.	no	9
taskRegsSet( )	Takes pointer to REG_SET structure.	no	9
semTake( )	Takes new timeout parameter.	yes, w/ *semCreate*	9
semGive( )	Returns STATUS instead of VOID.	yes	10
semInit( )	Returns STATUS instead of VOID.	yes	10
semDelete( )	Returns STATUS instead of VOID.	yes	10
calloc( )	Returns VOID* instead of *char**.	yes	10
malloc( )	Returns VOID* instead of *char**.	yes	10
realloc( )	Returns VOID* instead of *char**.	yes	10
memInit( )	Returns STATUS instead of VOID.	yes	10
memPartAddToPool( )	Returns STATUS instead of VOID.	yes	10
memPartAlloc( )	Returns VOID* instead of *char**.	yes	10
memPartRealloc( )	Returns VOID* instead of *char**.	yes	10
memPartOptionsSet( )	Returns STATUS instead of VOID.	yes	10
memPartShow( )	Returns STATUS instead of VOID.	yes	10
wdLibInit( )	Returns STATUS instead of VOID.	yes	10
ld( )	Takes new parameter for filename.	yes, from shell	10
ls( )	Takes new parameter for long format.	yes, from shell	11
read( )	Number of bytes read conforms to POSIX.	yes	11
period( )	Returns task ID instead of STATUS.	probably	11
repeat( )	Returns task ID instead of STATUS.	probably	11
RT-11 routines	Name prefix is "rt11Fs" instead of "rt11"	no	12
rt11FsDevInit( )	Takes pointer to a BLK_DEV structure.	no	12
ramDevCreate( )	Creates BLK_DEV structure.	no	12

† "Transparent" means no code changes should be required.

2.2.1 Scan-Routine Formatting: *scanf*( ), etc.

For ANSI compliance, changes were made to the formatting specifications used by *scanf*( ), *sscanf*( ), and *fscanf*( ). These changes are not backward compatible.

In 4.0.2, the uppercase floating-point field specifications, %E, %F, and %G, implied that the corresponding argument was a *double*, while the lowercase specifications, %e, %f, and %g, implied that the corresponding argument was a *float*.

In 5.0.3, as specified by ANSI, the uppercase specifications are identical to their lowercase counterparts. To indicate that the corresponding argument is a *double*, use the "l" modifier, e.g., %le, %lf, %lg.

2.2.2 Print-Routine Return Values: *printf*( ), etc.

OLD:

```
VOID printf (fmt, ...)
VOID fdprintf (fd, fmt, ...)
VOID printErr (fmt, ...)
int sprintf (buffer, fmt, ...)
int fprintf (fp, fmt, ...)
```

NEW:

```
int printf (fmt, ...)
int fdprintf (fd, fmt, ...)
int printErr (fmt, ...)
int sprintf (buffer, fmt, ...)
int fprintf (fp, fmt, ...)
```

In 4.0.2, *printf*( ), *fdprintf*( ), and *printErr*( ) returned no value (VOID) and the routine *fprintf*( ) returned OK if the output was successful.

In 5.0.3, all formatted "print" routines return a count of the characters successfully output, or ERROR if the output is unsuccessful.

In 4.0.2, *sprintf*( ) returned the number of characters put into the specified buffer *including* the null termination. In 5.0.3, as specified by ANSI, it returns the number of characters put into the specified buffer *excluding* the null termination. Thus the count returned by *sprintf*( ) in 5.0.3 is one less than the count returned in 4.0.2.

2.2.3 Symbol Table Creation and Initialization: *symTblCreate( )*, *symTblInit( )*

```
OLD:    SYMTAB_ID symTblCreate (maxSymbols, maxSymLen)
        STATUS symTblInit (pTbl, maxSymbols, maxSymLen)

NEW:    SYMTAB_ID symTblCreate (hashSizeLog2, sameNameOk, symPartId)
        STATUS symTblInit (pSymTbl, sameNameOk, symPartId, symHashTblId)
```

In 4.0.2, the routines *symTblCreate( )* and *symTblInit( )* used parameters for the maximum number of symbols that could be entered in the table, and the maximum symbol length. The symbol table was structured as a fixed linear list.

In 5.0.3, symbol tables are hash tables that can grow dynamically and indefinitely (see **3.8 Symbol Table and Hashing Libraries**). The parameters for *symTblCreate( )* are:

- the size of the hash table as a power of two,
- a flag indicating whether duplicate symbols are to be allowed,
- the ID of a memory partition from which to allocate space for symbols as they are added.

The main parameter to *symTblInit( )* is the ID of the hash table to use for the symbol table.

2.2.4 Kernel Initialization Routine: *kernelInit( )*

```
OLD:    VOID kernelInit (trapNum, rootRtn, rootStackSize, memPoolStart,
                         memPoolEnd, intStackSize, lockOutLevel)

NEW:    VOID kernelInit (rootRtn, rootMemSize, pMemPoolStart, pMemPoolEnd,
                         intStackSize, lockOutLevel)
```

In 4.0.2, the kernel initialization routine, *kernelInit( )*, defined a *trapNum* parameter, which was the fault number to use for kernel traps.

In 5.0.3, this parameter has been removed since Vx960 does not use kernel traps. Its historical justification no longer exists.

2.2.5 Take-Semaphore Routine: *semTake( )*

OLD: `VOID semTake (semId)`

NEW: `STATUS semTake (semId, timeout)`

In 5.0.3, *semTake( )* offers a new, optional parameter, *timeout*, that can be one of the following:

Symbol	Value	Meaning
WAIT_FOREVER	-1	no timeout; equivalent to 4.0.2 *semTake*
NO_WAIT	0	return immediately
	>0	return ERROR if not available after *timeout* ticks

However, for backward compatibility, this parameter is only used if the semaphore was created with one of the new 5.0.3 semaphore-creation calls: *semBCreate( )*, *semCCreate( )*, or *semMCreate( )* (see **3.3 Semaphores**). A semaphore created with the 4.0.2-compatible *semCreate( )* does not use the *timeout* parameter. Thus applications that used semaphores in 4.0.2 do not have to be changed immediately in order to run under 5.0.3. Eventually, however, all semaphore calls should be converted to the 5.0.3 preferred calls.

If you do not use the *timeout* parameter, you will see the following warning:

```
too few arguments to function 'semTake'
```

In spite of the warning, the current implementation of the compiler will generate the correct code.

2.2.6 Memory Allocation: *malloc( )*, etc.

NEW:
```
VOID *calloc (elemNum, elemSize)
VOID *malloc (nBytes)
VOID *realloc (pBlock, newSize)
VOID *memPartAlloc (partId, nBytes)
VOID *memPartRealloc (partId, pBlock, nBytes)
```

In 4.0.2, memory allocation routines returned type *char**. For ANSI compliance, they now return a value of type VOID*. The actual value they return is unchanged. In ANSI C, *void** is used as a "generic object pointer type."

2.2.7 Routines Changed from VOID to STATUS

NEW:
```
STATUS memInit (pPool, poolSize)
STATUS memPartAddToPool (partId, pPool, poolSize)
STATUS memPartOptionsSet (partId, options)
STATUS memPartShow (partId, type)
STATUS semInit (pSemaphore)
STATUS semDelete (semId)
STATUS semGive (semId)
STATUS wdLibInit ()
```

In 4.0.2, the above routines returned no value (VOID). In 5.0.3, they return the status of the invocation (STATUS), i.e., OK or ERROR.

2.2.8 Load Routine: *ld*( )

OLD: `STATUS ld (syms, noAbort)`

NEW: `STATUS ld (syms, noAbort, name)`

In 5.0.3, the shell routine *ld*( ) takes an additional parameter *name*. If *name* is non-NULL, it is used as the file name to be opened and loaded from. If *name* is NULL, standard input is used. Thus, in the following example, omitting *name* calls the routine with the default value of NULL and still loads from standard input:

```
-> ld <foo
```

2.2.9 List-Files Routine: *ls*( )

OLD: `STATUS ls (dirName)`

NEW: `STATUS ls (dirName, doLong)`

In 5.0.3, the shell routine *ls*( ) takes an additional flag *doLong*. If TRUE, it causes the directory listing to be in long format, as in "ls -l" in UNIX. If FALSE, e.g., if omitted, it outputs the list in short form. Note that there is also a new **usrLib** routine *ll*( ) (long list), which is equivalent to calling *ls*( ) with *doLong* set to TRUE.

> **NOTE:** The *ls*( ) routine no longer works with netDrv and no longer displays RT-11 "empty" entries. Use *lsOld*( ) in such cases.

2.2.10 Read Routine: *read*( )

In 4.0.2, the *read*( ) routine was specified as reading any number of bytes from 1 to the number of bytes specified in the call, even if there were more bytes available. Each driver could read a convenient number of bytes. The number of bytes read with NFS and RT-11 was often less than the specified number, typically corresponding to a sector or other buffer-size variable.

In 5.0.3, the number of bytes read by *read*( ) is always the full number of bytes specified, or the number of bytes remaining in the file, whichever is less. This behavior conforms to the POSIX definition of *read*( ).

2.2.11 Task Repetition Routines: *period*( ), *repeat*( )

OLD:
```
STATUS period (secs, func, arg1, arg2, arg3, ... )
STATUS repeat (n, func, arg1, arg2, arg3, ... )
```

NEW:
```
int period (secs, func, arg1, arg2, arg3, ... )
int repeat (n, func, arg1, arg2, arg3, ... )
```

In 4.0.2, the routines *period*( ) and *repeat*( ) returned the status value OK if the call was successful, or ERROR if unable to spawn the task.

In 5.0.3, both routines return the ID of the task that was spawned, or ERROR if unable to spawn the task.

2.2.12 RT-11 Module and Routine Names

OLD: rt11Lib

NEW: rt11FsLib

The RT-11 file system library has been renamed **rt11FsLib** for consistency. Accordingly, names for all RT-11 file system routines are now prefixed with "rt11Fs" instead of "rt11". These names are typically called by disk drivers rather than directly by users, and due to changes in disk-driver implementation, the drivers must be rewritten anyway (see **5.3 Changes in Block Device Drivers**).

2.2.13 RT-11 Device Initialization: *rt11FsDevInit*( )

```
OLD:    STATUS rt11DevInit (vdptr, bytesPerSec, secPerTrack, nSectors,
                            rt11Fmt, nEntries, rdSec_func, wrtSec_func,
                            reset_func)

NEW:    RT_VOL_DESC *rt11FsDevInit (devName, pBlkDev, rt11Fmt, nEntries,
                                    changeNoWarn)
```

In 5.0.3, *rt11FsDevInit*( ) has changed to take a *pBlkDev* parameter that is the block I/O device to use for the RT-11 device. See **5.3 Changes in Block Device Drivers.**

2.2.14 RAM Device Creation: *ramDevCreate*( )

```
OLD:    STATUS ramDevCreate (name, where, bytesSec, secTrack, nSectors,
                             rtFmt, secOffset)

NEW:    BLK_DEV *ramDevCreate (ramAddr, bytesPerSec, secPerTrack,
                               nSectors, secOffset)
```

In 5.0.3, *ramDevCreate*( ) has been changed to create a "block I/O device," which is a new type of object that can be connected to a file system (i.e., either RT-11 or Vx960-DOS) to make a file system I/O device. Some of the information previously supplied to the *ramDevCreate*( ) call is now specified in *rt11FsDevInit*( ) or *dosFsDevInit*( ). See **5.3 Changes in Block Device Drivers.**

2.2.15 RAM Disk File System Initialization: *ramMkfs( )* Deleted

```
OLD:     STATUS ramMkfs (name, nbytes, where, dontInit)
             char     *name;          /* Device name */
             int      nbytes;         /* No. of bytes total for RAM disk  */
             char     *where;         /* Where it is in memory (0=malloc) */
             BOOL     dontInit;       /* FALSE=init RT-11 dir, TRUE=don't */

NEW:     BLK_DEV *ramDevCreate (ramAddr, bytesPerSec, secPerTrack,
                                nSectors, secOffset)
             char     *ramAddr;       /* Address of ram disk (0 = malloc) */
             int      bytesPerSec;    /* Number of bytes per sector       */
             int      secPerTrack;    /* Number of sectors per track      */
             int      nSectors;       /* Number of sectors on this device */
             int      secOffset;      /* Number of sectors to skip at     */
                                       * beginning of physical device     */

         RT_VOL_DESC *rt11FsMkfs (volName, pBlkDev)
             char     *volName;       /* volume name to use */
             BLK_DEV  *pBlkDev;       /* pointer to block device struct */

         DOS_VOL_DESC *dosFsMkfs (volName, pBlkDev)
             char     *volName;       /* volume name to use */
             BLK_DEV  *pBlkDev;       /* pointer to block device struct */
```

In 5.0.3, as described elsewhere, the standard interface between block I/O devices and file systems is the BLK_DEV structure (see **5.3 Changes in Block Device Drivers**). The old call *ramMkfs( )* was not sufficient to support this structure. Instead, use *ramDevCreate( )* to create a RAM block I/O device and then call either *rt11FsMkfs( )* or *dosFsMkfs( )* to create either an RT-11 or DOS file system on the device.

For example, to create a 200Kb RAM disk with memory allocated using *malloc( )*, and to create either an RT-11 or DOS file system on it, the following could be invoked from the shell:

```
-> pBlkDev = ramDevCreate (0, 512, 400, 400, 0)

-> rt11FsMkfs ("/rt11/", pBlkDev)
      or
-> dosFsMkfs ("/dos/", pBlkDev)
```

In 4.0.2, the *ramMkfs( )* call took a parameter *dontInit*, which, if TRUE, would inhibit writing a new directory, etc. on the disk. In 5.0.3, if a new directory should not be written, use *dosFsDevInit( )* or *rt11FsDevInit( )* in place of *dosFsMkfs( )* or *rt11Mkfs( )*, which always write a new directory on the disk.

2.2.16 Task Stack Space

In 4.0.2, several extended task facilities dynamically allocated memory at task creation and freed that memory at task deletion. This was done for optional or variable-length extensions to a task's context.

In 5.0.3, this mechanism was changed to carve the necessary optional or variable length space from the specified task stack size. This eliminates the need for task creation and deletion hooks to do dynamic memory allocation. However, the actual stack space available to the task is reduced from the amount specified in the ***taskSpawn*( )** or ***taskInit*( )** call, by the amount required by these facilities.

It may be necessary to increase stack sizes of applications which ran under 4.0.2. The routine *checkStack*( ) can be used to determine a task's stack usage.

The following is a list of facilities that carve space from the task stack space:

Facility	Size	
task name	**strlen** (*taskname*) + 1	
signals	**sizeof** (TASK_SIGNAL_INFO)	(592 bytes)
floating point support	**sizeof** (FP_CONTEXT)	(64 bytes)

3. KERNEL ENHANCEMENTS

The internal implementation of the Vx960 kernel has changed considerably in 5.0.3. It requires less memory and operates at higher speed. The give/take timings of binary semaphores are much faster; context switching time is now extremely fast and performs in constant time; the implementation of floating point processing is much improved; and interrupt latency has been greatly reduced.

3.1 Queue Implementation

New queuing algorithms have been added to Vx960 and have been implemented separately from the operation system facilities that use them, thus providing flexibility for the future implementation of new queuing methods.

There are a variety of kernel queues in Vx960: the *ready queue* is a priority-sorted queue of all tasks eligible for scheduling; the tick queue is used by functions involving timing; the *semaphore queue* is the list of blocked tasks waiting on a semaphore. The *active queue* is a FIFO (first-in-first-out) list of all tasks in the system. Each of these specialized queues requires a different queuing algorithm. Instead of hiding the algorithms in the libraries that use them, they have been extracted into autonomous queuing libraries that are interchangeable. These libraries have a common interface and data structure, allowing Vx960 kernel queues to be configurable and extendable.

None of the changes in queue implementation have significantly altered the user interface; however, they form the basis for many of the optimizations, scheduling algorithms, and semaphore options now available with release 5.0.3.

3.2 Tasking

Release 5.0.3 introduces a number of enhancements to the kernel's tasking facility. Fundamental features of task scheduling have been tuned for better performance. Routines have been added, providing new functionality, and some existing routines have been made more flexible.

3.2.1 Context Switching Time

When one task stops executing and another begins, the context of the old task is saved in its associated task control block (TCB) and the context of the new task is restored. This is known as a *task context switch*. Context switching time is greatly reduced in release 5.0.3 due, in part, to a new queuing mechanism for the ready queue. During a context switch, the previously executing task must be inserted. Context switching time is therefore directly affected by the performance of the ready queue.

Prior versions of Vx960 use a prioritized, doubly-linked list as the basis of the ready queue. While nodes in such a list can be removed in fixed time, the insertion time degrades proportionally with the number of nodes in the queue; thus context switching time was degraded as the ready queue grew in length. The 5.0.3 ready queue uses a newly created bitmap priority queue, which exhibits fixed-time performance for both node insertion and node removal. The result of this enhancement is that context switching time is of higher performance and in constant time.

Context switching time has been further reduced by eliminating the use of task variables in Vx960 facilities. Since task variables must be saved and restored during a context switch, rewriting facilities to avoid their use has resulted in a much faster context switching time for the basic system.

The libraries which previously depended on task variables are **rpcLib**, **stdioLib**, **errnoLib**, and **dbgLib**. Instead of task variables, they now use macros referenced through the global variables ***taskIdCurrent,*** which is always a pointer to the TCB of the current executing task. All information needed for standard I/O, RPC, and *errno* is contained within the TCB. The use of switch hooks by these libraries has also been cut back or eliminated, further contributing to the reduction of context switching time in the basic system.

Context switching time has also been improve by no longer saving the floating-point registers with every context switch involving a floating point task. See **3.7 Floating Point.**

3.2.2 Basic Routines

New tasks are created in Vx960 with the ***taskSpawn*( )** routine. This routine allocates memory required for the task's data structures and stack, initializes the task's data structures, and then activates the task. Vx960 also includes lower-level routines to perform these functions independently. In 4.0.2, the routine ***taskCreate*( )** performed the data structure initialization only but not the memory allocation, allowing the memory for the task's data structures to be determined by the caller. In 5.0.3, this routine has been renamed ***taskInit*( )** to maintain consistency with overall Vx960 naming conventions. For backwards compatibility, the old ***taskCreate*( )** call is still included but will be deleted in a future release.

3.2.3 Deletion Safety: *taskSafe( )*, *taskUnsafe( )*

Two new routines, ***taskSafe( )*** and ***taskUnsafe( )***, address problems stemming from unexpected deletion of tasks. The basic function of ***taskSafe( )*** is to protect a task from deletion by other tasks. Frequently, a task executing in a critical region or engaging a critical resource is particularly important to protect. Consider the following scenario. A task takes a semaphore for exclusive access to some data structure. While executing inside the critical region, the task is deleted by another task. Because the task was unable to complete the critical region, the data structure may have been left in a corrupt or inconsistent state. Furthermore, the semaphore can never be released by the task. Hence, the critical resource is now unavailable for use by any other task; it is essentially frozen.

Using ***taskSafe( )*** to protect the task that took the semaphore prevents such an outcome. Any task that tries to delete a task thus protected will block. When finished with its critical resource, the protected task can make itself available for deletion by calling ***taskUnsafe( )***, which unblocks any of the deleting tasks. To support nested deletion-safe regions, a count is kept of the number of times ***taskSafe( )*** and ***taskUnsafe( )*** are called. The safe count is checked by ***taskDelete( )*** whenever it tries to delete. Protection operates only on the calling task; i.e., a task cannot make another task safe or unsafe from deletion.

The following code fragment shows how a critical region of code can be protected using ***taskSafe( )*** and ***taskUnsafe( )***:

```
taskSafe ();
semTake (semId, WAIT_FOREVER);
.
.   (critical region)
.
semGive (semId);
taskUnsafe ();
```

However, for convenience and efficiency, deletion safety can be handled with a new semaphore option that works with mutual exclusion semaphores, a new class of semaphore available in Release 5.0.3. If the option is enabled, each ***semTake( )*** includes the task-safe operation and each ***semGive( )*** includes the task-unsafe operation. See **3.3.2 Mutual-Exclusion Semaphores** for more information.

The 5.0.3 operating system facilities have been made more robust with the addition of deletion safety in most places where mutual exclusion has been used. Users of earlier versions of Vx960 noticed that if they spawned a task that sent serial output to the console and then deleted the task, console output would occasionally be lost and could only be regained by restarting the shell. This happened because the task performing serial output was deleted while it had the console semaphore, which left the console device with an empty semaphore. This and similar scenarios are resolved in 5.0.3.

3.3 Semaphores

Semaphores are the fundamental synchronization/mutual exclusion mechanism at the foundation of all intertask communications in Vx960. Release 5.0.3 now offers two new types of semaphores in addition to the standard binary semaphore: the *counting semaphore* and the *mutual-exclusion semaphore*. The counting semaphore is useful for coordinating access to resources of which there are multiple copies. The mutual-exclusion semaphore is a specialized binary semaphore designed to handle problems inherent in mutual exclusion.

All semaphores have been highly optimized. Two major new features now offered with all semaphores are time-outs and queue options.

3.3.1 Binary Semaphores

The general-purpose binary semaphore is capable of addressing the requirements of the two fundamental forms of task coordination: mutual exclusion and synchronization. The binary semaphore has the least overhead associated with it, making it particularly applicable to high-performance requirements. The mutual-exclusion semaphore described below in **3.3.2 Mutual-Exclusion Semaphores** is also a binary semaphore, but it has been tailored to address problems inherent to mutual exclusion. Alternatively, the binary semaphore may be utilized for mutual exclusion if the advanced features of the mutual-exclusion semaphore are deemed unnecessary.

3.3.1.1 Mutual Exclusion

Binary semaphores interlock access to a shared resource efficiently. Unlike disabling interrupts or preemptive locks, binary semaphores limit the scope of the mutual exclusion to only the associated resource. In this technique, a semaphore is created to guard the resource. Initially the semaphore is full.

```
SEM_ID semMutex;

/* create a binary semaphore that is initially full */
semMutex = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
```

When a task wants to access the resource, it must first take that semaphore. So long as the task keeps the semaphore, all other tasks seeking access to the resource will be blocked from execution. When the task is finished with the resource, it gives back the semaphore allowing another task to use to the resource.

Thus all accesses to a resource requiring mutual exclusion are bracketed with *semTake*( ) and *semGive*( ) pairs:

```
semTake (semMutex, WAIT_FOREVER);
.
.   (critical region, only accessible by a single task at a time)
.
semGive (semMutex);
```

3.3.1.2 Synchronization

When used for task synchronization, a semaphore may represent a condition or event that a task is waiting for. Initially the semaphore is empty. A task or interrupt service routine signals the occurrence of the event by giving the semaphore. Another task waits for the semaphore by calling *semTake*( ). The waiting task will be blocked until the event occurs and the semaphore is given.

Note the difference in sequence when semaphores are used for mutual exclusion and when they are used for synchronization. For mutual exclusion, the semaphore is initially full, and each task first takes and then gives back the semaphore. For synchronization, the semaphore is initially empty, and one task waits to take the semaphore, which will be given by another task.

The following is an example of using semaphores for task synchronization:

```
SEM_ID syncSem; /* ID of sync semaphore */

init ()
    {
    intConnect (..., eventInterruptSvcRout, ...);
    syncSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
    taskSpawn (..., task1);
    }

task1 ()
    {
    ...
    semTake (syncSem, WAIT_FOREVER); /* wait for event to occur */
    .../* process event */
    }

eventInterruptSvcRout ()
    {
    ...
    semGive (syncSem); /* let task 1 process event */
    ...
    }
```

In the above example, when the *init*( ) routine is called, the binary semaphore is created, an interrupt service routine is attached to an event, and a task is spawned to process the event. The routine *task1*( ) will run until it calls *semTake*( ). It will remain blocked at that point until an event causes the interrupt service routine to call *semGive*( ). When the interrupt service routine completes, *task1*( ) will execute to process the event. There is an advantage of handling event processing within the context of a dedicated task: Less processing takes place at interrupt level, thereby reducing interrupt latency. This model of event processing is recommended for real-time applications. time applications. time applications.

3.3.2 Mutual-Exclusion Semaphores

The mutual exclusion semaphore is a specialized version of the binary semaphore designed to address issues inherent in mutual exclusion, including priority inversion, deletion safety, and recursive access to resources.

3.3.2.1 Priority Inversion

Priority inversion arises when a higher-priority task is forced to wait an indefinite period of time for the completion of a lower-priority task. Consider the following scenario: *t1*, *t2*, and *t3* are tasks of high, medium, and low priority, respectively. *t3* has acquired some resource by taking its associated semaphore. When *t1* preempts *t3* and contends for the resource by taking the same semaphore, it becomes blocked. If we could be assured that *t1* would be blocked no longer than the time it takes *t3* to finish with the resource, the situation would not be particularly problematic. After all the resource is non-preemptible. However, the low-priority task is vulnerable to preemption by medium-priority tasks; a preempting task, *t2*, will inhibit *t3* from relinquishing the resource. This condition could persist, blocking *t1* for an indefinite period of time. See the diagram in Figure 1.

The mutual exclusion semaphore has an additional option SEM_INVERSION_SAFE, which enables a *priority inheritance* algorithm. Priority inheritance solves the problem of priority inversion by elevating the priority of *t3* to the priority of *t1* during the time *t1* is blocked on *t3*. This protects *t3*, and indirectly *t1*, from preemption by *t2*. Stated more generally, the priority inheritance protocol assures that a task which owns a resource will execute at the priority of the highest priority task blocked on that resource. When execution is complete, the task gives up the resource and returns to its normal or standard priority. Hence, the "inheriting" task is protected from preemption by any intermediate-priority tasks. See the diagram in Figure 2.

3.3.2.2 Deletion Safety

Another problem of mutual exclusion involves task deletion. Within a critical region guarded by semaphores, it is often desirable to protect the executing task from unexpected deletion. Deleting a task executing in a critical region can be catastrophic. The resource could be left in a corrupted state and the semaphore guarding the resource left unavailable, effectively shutting off all access to the resource.

As discussed previously, the primitives *taskSafe*( ) and *taskUnsafe*( ) provide the desired protection, but as this protection goes hand in hand with mutual exclusion, the mutual-exclusion semaphore offers the option SEM_DELETE_SAFE, which enables an implicit *taskSafe*( ) with each *semTake*( ), and a *taskUnsafe*( ) with each *semGive*( ). This convenience is also more efficient as the resulting code requires fewer entrances to the kernel.

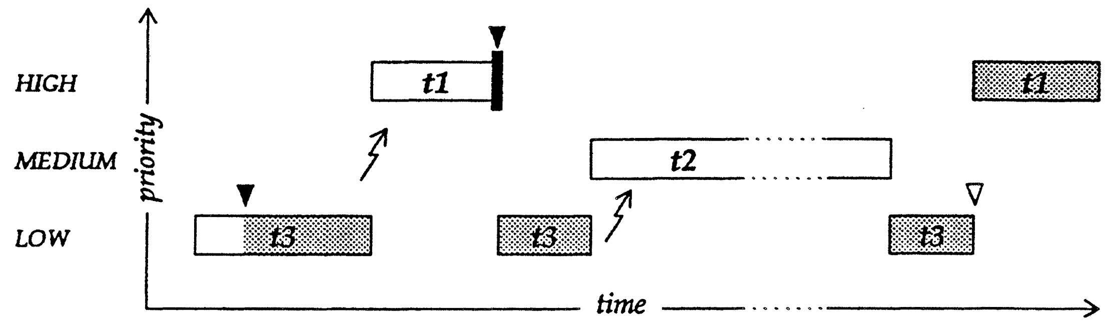

Figure 1. Priority Inversion

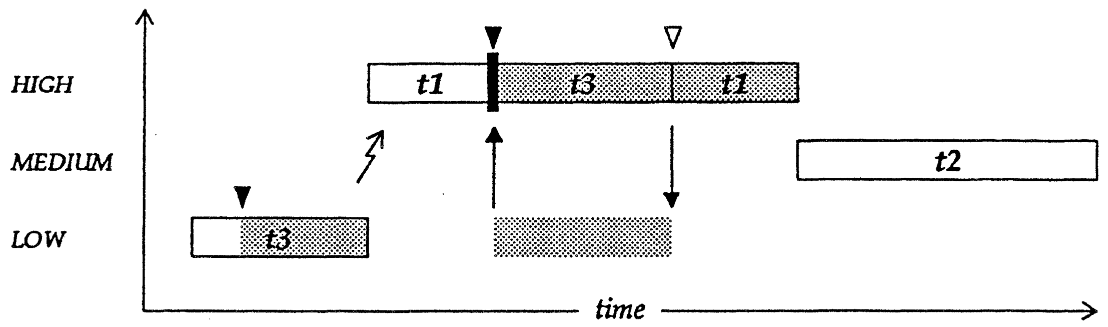

Figure 2. Priority Inheritance

Key:		
▼ - take semaphore	↯ - preemption	
▽ - give semaphore	↑↓ - priority inheritance/release	
▨ - own semaphore	▮ - block	

3.3.2.3 Recursion

One last feature of mutual exclusion semaphores is that they may be taken "recursively," i.e., a mutual exclusion semaphore can be taken more than once by the task that owns it before finally being released. Recursion is useful for a set of routines that need mutually exclusive access to a resource, but may need to call each other. It is made possible by the fact that with mutual exclusion semaphores, the system keeps track of which task currently owns the semaphore.

Before being released, a mutual exclusion semaphore taken recursively must be given the same number of times it has been taken; this is tracked by means of a count which is incremented with each *semTake( )* and decremented with each *semGive( )*.

The example below illustrates recursive use of a mutual exclusion semaphore. Function A requires access to a resource which it acquires by taking *mySem*; function A may also need to call function B, which also requires *mySem*:

```
SEM_ID mySem;

mySem = semMCreate (...);

funcA ()
    {
    semTake (mySem, WAIT_FOREVER);
    ...
    funcB ();
    ...
    semGive (mySem);
    }

funcB ()
    {
    semTake (mySem, WAIT_FOREVER);
    ...
    semGive (mySem);
    }
```

3.3.3 Counting Semaphores

The counting semaphore works like the binary semaphore except that it keeps track of the number of times a semaphore has been given. Every time a semaphore is given, the count is incremented; every time a semaphore is taken, the count is decremented. When the count reaches zero, a task that tries to take the semaphore is blocked. As with the binary semaphore, if a semaphore is given and a task is blocked, it becomes unblocked. However, if a semaphore is given and no tasks are blocked, then the count is incremented. This means that a semaphore that has been given twice can be taken a second time without being blocked, unlike the binary semaphore.

Counting semaphores are useful for guarding resources of which there are multiple copies. For example, the use of five tape drives could be coordinated using a counting semaphore with an initial count of five, or a ring buffer with 256 entries could be implemented using a counting semaphore with an initial count of 256.

3.3.4 Unblocking Tasks: *semFlush*( )

Counting and binary semaphores support a new function, ***semFlush*( )**. This routine automatically unblocks all tasks pended on the specified semaphore. (When a ***semFlush*( )** has unblocked all pending tasks specific to that semaphore, the ***semTake*( )** routine returns OK.) Unlike ***semGive*( )**, ***semFlush*( )** never changes the state of a semaphore. The ***semFlush*( )** routine is used primarily as a broadcast mechanism for synchronization. This new routine also works with the old-version binary semaphores (see **3.3.7 Vx960 4.0.2-Compatible Semaphores**).

3.3.5 Timeouts

All three new semaphore types include the ability to time out, which is managed by a parameter to ***semTake*( )**. A task that tries to take a semaphore supplies a value that specifies the amount of time in ticks that the task can wait in a blocked state. If the task succeeds in taking the semaphore in the allotted amount of time, ***semTake*( )** returns OK; if it times out before taking the semaphore, it returns ERROR. A value of WAIT_FOREVER means do not time out, wait indefinitely; a value of NO_WAIT means do not wait at all.

3.3.6 Semaphore Wait Queue: FIFO vs. Priority

Tasks blocked on a semaphore can now be queued based on either of two criteria: first-in-first-out (FIFO) order or priority order. Previously, tasks were queued only on the basis of assigned priority.

This option is specified with the semaphore-creation routine. Semaphores using priority inheritance (SEM_INVERSION_SAFE option) can use priority-order queuing only.

3.3.7 Vx960 4.0.2-Compatible Semaphores

While the new semaphore types and features have meant changes in the user interface, the syntax for the binary semaphores used in Vx960 4.0.2 remains unchanged, providing backward compatibility for applications developed under the previous release. This means that there are now two versions of binary semaphores. Semaphores compatible with 4.0.2 can be created with *semCreate*( ). The interface for these old-version binary semaphores differs from the new only in that it has no options for time-outs or queues; however, it has been implemented to use almost all the performance optimizations achieved with the new binary semaphores. In future releases of Vx960, the 4.0.2 semaphore interface will become obsolete in favor of the more flexible 5.0.3 interface.

```
oldSem = semCreate()
semTake (oldSem)              /* no timeout specified */
```

3.3.8 User Interface

Instead of creating a complete set of function calls specific to each semaphore type, the 5.0.3 semaphore interface has been designed to be as uniform as possible. As shown in Figure 3, the primitives for the delete, give, and take functions work on all the semaphore types; only the creation phase requires special calls. This is particularly powerful because it means an entire set of semaphore calls can easily be changed from one type to another just by changing the initial create call.

In the new semaphores, the functionality of *semClear*( ) has been subsumed by *semTake*( ): a value of NO_WAIT in the time-out field means do not wait for a semaphore to become available. For additional information, see **2.2.6 Take-Semaphore Routine: semTake( ).**

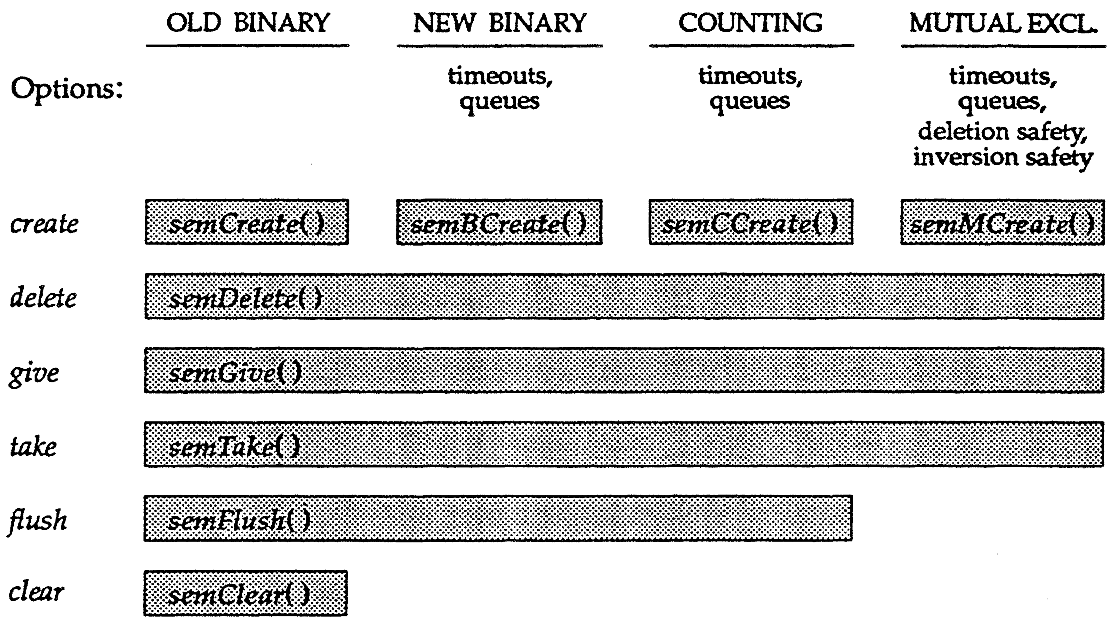

Figure 3. Semaphore Routines in Vx960 5.0.3

3.4 Message Queues

Modern real-time applications are constructed as a set of independent but cooperating tasks. While semaphores provide a high-speed mechanism for the synchronization and interlocking of tasks, often a higher-level mechanism is necessary to allow cooperating tasks to communicate with each other. In Vx960, the primary intertask communication mechanism within a single CPU is *message queues*. Message queues allow a variable number of messages, each of variable length, to be queued in first-in-first-out order. Any task or interrupt service routine can send messages to a message queue. Any task can receive messages from a message queue. Multiple tasks can send to and receive from the same message queue. Full-duplex communication between two tasks generally requires two message queues, one for each direction.

3.4.1 Creating and Using Message Queues

A message queue is created with *msgQCreate( )*. Its parameters specify the maximum number of messages that can be queued to the message queue and the maximum length in bytes of each message. Enough buffer space is preallocated for the specified number and length of messages.

A task or interrupt service routine sends a message to a message queue with *msgQSend( )*. If no tasks are waiting for messages on that queue, the message is added to the queue's buffer of messages. If any tasks are already waiting for a message from that message queue, the message is immediately delivered to the first waiting task.

A task receives a message from a message queue with *msgQReceive( )*. If any messages are already available in the message queue's buffer, the first message is immediately dequeued and returned to the caller. If no messages are available, then the calling task will block and be added to a queue of tasks waiting for messages. This queue of waiting tasks can be ordered either by task priority or FIFO, as specified in an option parameter when the queue is created. The number of messages available in a message queue can be obtained using *msgQ*NumMesgs( ) or *msgQ*InfoGet( ).

3.4.2 Message Queues and Pipes

Pipes provide an alternative interface to the message queue facility that goes through the Vx960 I/O system. Pipes are virtual I/O devices managed by the **pipeDrv** driver. Pipes are created and given names with *pipeDevCreate( )* and can then be read and written with normal *read( )* and *write( )* calls.

The **pipeDrv** driver uses the Vx960 message queue facility to buffer and deliver the messages. The pipe driver gives access to the message queue facility through the I/O system. The main differences between using pipes and using message queues directly are:

- Pipes have I/O device names.
- Pipes use the standard I/O functions *open( ), close( ), read( ), write( )* while message queues use the specific functions *msgQSend( )* and *msgQReceive( )*.
- Pipes respond to standard *ioctl( )* functions.
- Pipes can be used in a *select( )* call.

- Message queues have more flexible options for time-outs and message priorities.
- Pipes are less efficient than message queues because of the additional overhead of the I/O system.

3.5 Watchdog Timers

The watchdog timer facility was previously implemented with its own separate queue. In 5.0.3 it is included as part of the kernel, since it is now based on the optimized tick queue, shared by all other kernel timing functions. The use of this queue significantly improves watchdog performance.

The 5.0.3 watchdog timer facility is fully compatible with 4.0.2, and also includes the new routine *wdDelete*( ). However, the time-out feature of the new semaphore routines obviates the need for some watchdogs that may have been used by applications developed under 4.0.2.

3.6 Alternative Kernels and the TCB Extension

The pSOS and VRTX kernels are not supported in Vx960. For this reason, the TCB extension has been eliminated. Useful information in the TCB extension has been moved into the TCB proper. Perhaps the most subtle change resulting from this modification is that task create/switch/delete hooks no longer take a pointer to the TCB extension, but a pointer to the task TCB.

3.7 Floating Point

In 5.0.3, saving the floating-point register state during context switching has been significantly optimized. The floating-point state is not necessarily saved when a task using floating point is switched out; instead, saving is delayed until another floating-point task is switched in. Therefore, in switching between floating-point and non-floating-point tasks, the kernel may not need to save and restore the floating-point state at all.

3.8 Symbol Table and Hashing Libraries

The library symLib, which manages Vx960 symbol tables, has been changed to use a hash table, with the result that symbol lookup in 5.0.3 is hundreds of times faster. Due to the efficiency of hashing, the facilities that use the symbol table can now run much faster. This includes the Vx960 loader, shell, and source-level debugger.

The switch to hashing has necessitated changes to the routines ***symTblInit*( )** and ***symTblCreate*( )**. Both take arguments specifying whether or not symbols of the same name should be allowed, and the memory partition for symbol allocation. An additional argument to ***symTblCreate*( )** specifies the hash table size. For additional information, see **2.2.3 Symbol Table Creation and Initialization: symTblCreate( ), symTblInit( )**.

3.9 Error Status Handling

ANSI C requires a global integer *errno* to be set to an appropriate error number whenever a library function is to return information about an error. Previously, Vx960 used ***errnoSet*( )** and ***errnoGet*( )** to fulfill this function. However, in 5.0.3, Vx960 applications can directly reference the global variable *errno*, which is saved and restored as part of a task's context. Interrupt service routines may also utilize *errno* since it is saved and restored off the interrupt stack as part of the functionality of the code generated by ***intConnect*( )**. For backward compatibility, ***errnoSet*( )** and ***errnoGet*( )** are still fully functional.

4. I/O SYSTEM

4.1 Select

The implementation of *select*( ) in release 5.0.3 has been modified extensively, providing wider functionality and a substantial increase in performance.

The function of *select*( ) is to wait for activity on a set of file descriptors. Previous implementations polled the requested file descriptors periodically, with the polling period specified as a number of clock ticks. This technique resulted in a worst-case latency equal to the polling period, thereby causing applications that use *select*( ), such as RPC, to perform poorly.

The new implementation of *select*( ) uses a set of functions that permits tasks pended in *select*( ) and waiting for I/O on a device to be detected by the device's driver. Thus, the driver's interrupt service routine directly wakes up tasks that are pended on device activity, eliminating the need for polling. These functions, along with *select*( ), are contained in the new module selectLib.

Applications may now use *select*( ) with pipes and serial devices in addition to sockets. Also, *select*( ) now examines *write* file descriptors in addition to *read* file descriptors; however, exception file descriptors are still unsupported.

Driver implementors should review the new documentation for selectLib if they wish to support *select*( ).

4.2 SCSI Support

Vx960 now offers support for SCSI (Small Computer System Interface), an increasingly popular interface standard for a wide-ranging variety of computer peripherals. Release 5.0.3 supports the SCSI protocols for Winchester disks, optical disks, floppy disks, and tape drives. Both file systems currently used by Vx960 – RT-11 and Vx960 DOS – can be used on SCSI disks.

Drivers are supplied for the Fujitsu MB87030 and Western Digital WD33C93 SCSI devices. The Western Digital device is used by the Heurikon HK80/V960 target board.

The device-independent interface for handling the basic SCSI protocol is supplied in the module **scsiLib**. The device-dependent interfaces are **mb87030Lib** and **wd33c93Lib**. The SCSI interface is included in the Vx960 system image by defining INCLUDE_SCSI in /usr/config/*target*/config.h for relevant targets.

The SCSI disconnect/reconnect feature, which is used only by systems using multiple peripherals, is not yet implemented. The option of using the DMA (Direct Memory Access) controller with SCSI may be available on a per-target basis.

4.3 POSIX Standard Directory Handling

Vx960 5.0.3 routines for manipulating and searching directories follow the POSIX standard. The functions ***mkdir*( )**, ***rmdir*( )**, ***opendir*( )**, ***closedir*( )**, ***readdir*( )**, ***rewinddir*( )**, ***stat*( )**, and ***fstat*( )** are provided in dirLib(1). Code developed with these functions on other systems can easily be converted to Vx960.

5. LOCAL FILE SYSTEMS

Vx960 5.0.3 now offers a new file system compatible with the file systems of MS-DOS and other versions of DOS for PCs. The performance of the RT-11 file system, the only file system offered previously, has also been improved. Changes to Vx960 file system support and implementation are discussed below. A fuller discussion of Vx960 file systems is provided in **Chapter 5** of the *Vx960 Programmer's Guide*. See also the discussion of changes in RT-11 routines and RAM disk routines under **2.2 Summary of User-Interface Changes.**

5.1 New DOS-Compatible File System

The new Vx960 DOS file system is compatible with versions of MS-DOS up to and including 4.0, as well as other versions of DOS for PCs.

The Vx960 DOS file system is considerably more flexible than RT-11:

- It offers the advantage of allowing files and directories to be arranged hierarchically, while RT-11 allows only a "flat" arrangement using a single directory. This provides the added advantage of permitting an indefinite number of DOS files to be created on a volume, while RT-11 is limited to a fixed number of files.
- Unlike RT-11, blocks of files need not be contiguous. While such fragmentation may cause some increase in physical I/O time, it ultimately results in more efficient use of available disk space. To provide enhanced performance, the Vx960 DOS file system offers an I/O control option which can be used to preallocate contiguous disk space to files when required by application demands.
- DOS files can be appended, while RT-11 files generally cannot; the size of RT-11 files is usually fixed once they are initially created and closed.
- Multiple DOS files can be created without being closed; in some circumstances, an open RT-11 file must be closed before another file can be created.
- The Vx960 DOS file system supports disk volumes larger than 32 megabytes.
- Using the Vx960 DOS file system provides compatibility with widely available storage and retrieval media. Disks created under Vx960 and on MS-DOS personal computers and other systems may be freely interchanged.

Services for file-oriented device drivers using Vx960 DOS are provided in **dosFsLib**. The file system is included in the Vx960 system image by defining INCLUDE_DOSFS in the configuration file **/usr/vx/config/all/configAll.h** or in **/usr/vx/config/<target>/config.h**.

NOTE: **/usr/vx/config/all/configAll.h** and **/usr/vx/config/<target>/config.h** will be referred to as **configAll.h** or **<target>/config.h** respectively throughout the rest of this document.

5.2 New Raw Disk File System

Vx960 5.0.3 also offers a simple "raw file system" for use with disk devices. The raw file system treats the entire disk much like a single large file. Portions of the disk can be read and written, specified by byte offset, and simple buffering is performed. The raw file system offers advantages of size and speed when only very simple, low-level disk I/O is required.

This functionality was previously available by using the RT-11 file system and specifying the device name as a file name for file operations. Services for file-oriented device drivers using the raw file system are now provided in **rawFsLib**. The file system can be included in the Vx960 system image by defining **INCLUDE_RAWFS** in the configuration files **configAll.h** or **<target>/config.h**.

5.3 Changes in Block Device Drivers

5.3.1 Summary

Moving Vx960 from an I/O structure based on a single-file-system (RT-11) to one supporting multiple-file-systems has resulted in some fundamental changes in how I/O requests are routed to block device controllers[1]. Previously, the Vx960 I/O system issued I/O calls directly to block device drivers, which in turn made RT-11 file system calls. However, to allow a block device driver to be used with more than one type of file system, this sequence has been rearranged. Entries in the Vx960 driver table now refer to an RT-11, Vx960 DOS, or raw disk function library, rather than the actual device driver. I/O calls are routed directly to the appropriate file system, which in turn calls the device driver to perform physical I/O and other disk controller functions. Figure 4 shows a comparison of I/O routing for block devices in Vx960 releases 5.0.3 and 4.0.2.

In general, block device drivers for Vx960 5.0.3 are simpler than their earlier counterparts, since they no longer must route high-level I/O requests to the file system. The device driver, during its device initialization, describes the block device in a

1. A *block device* is a device that is organized as a sequence of individually accessible blocks of data. The most common type of block device is a disk. The term *block* refers to the smallest addressable unit on the device. For most disk devices, a Vx960 block corresponds to a *sector*, although terminology varies.

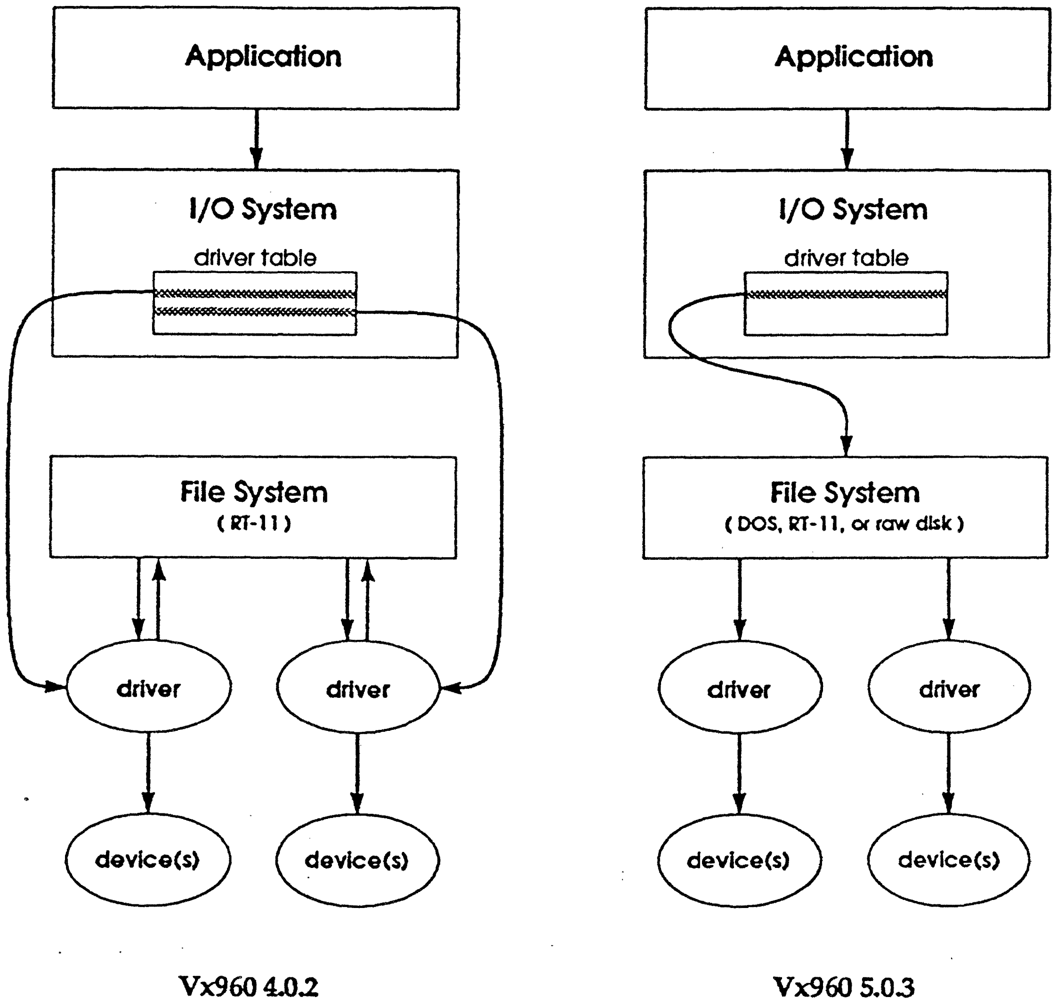

Figure 4. Implementation Changes in Block Device Drivers

standard data structure (BLK_DEV). The application program then calls a function (e.g., *dosFsDevInit*( ) or *rt11FsDevInit*( )) specifying the name to be used for the device, the address of the BLK_DEV structure describing the device, and other configuration data specific to the file system. This process associates the generic block device with the specific file system (e.g., Vx960 DOS or RT-11).

Existing device drivers for disk controller boards (Vx960 4.0.2 and earlier) require modification to remove code specific to RT-11, to implement the BLK_DEV description structure, and to no longer be installed in the I/O system driver table. The Vx960 RAM disk driver, ramDrv, has been modified to reflect these changes and can serve as an example for creating or modifying other disk device drivers.

The sections below outline the differences between 5.0.3 block device drivers and their previous counterparts, and should be useful for converting existing drivers to the new implementation.

5.3.2 Initializing the Driver

In previous versions of Vx960, a block device driver's initialization routine would install the driver in the I/O system's driver table using *iosDrvInstall( )*. Now, the file system is installed in the driver table instead, allowing the file system to be called directly by the I/O system in response to the basic I/O calls (*open( )*, *read( )*, etc.). As a result, *iosDrvInstall( )* should not be called by the driver's initialization routine.

5.3.3 The BLK_DEV Structure

The BLK_DEV structure, defined in the include file **blkIo.h**, serves as a standard interface between the device driver and the file system. Several parameters previously passed to the file system via the device initialization call, *rt11FsDevInit( )*, are now specified in this structure.

The BLK_DEV structure contains fields which describe both the physical configuration of the device (e.g., number of blocks on the device) and the addresses of routines within the driver.

Three of the five routines whose addresses are passed to the file system via the BLK_DEV structure were previously parameters to *rt11FsDevInit( )*. These are the routines to read a block, write a block, and reset the device. The fourth routine is the device driver's I/O control routine. The fifth routine is an optional status-check routine that is called by the file system at the start of every open or create operation.

The BLK_DEV structure must be the first element in the device descriptor structure used by the device driver. (In previous versions of Vx960, the required first element was an RT-11 volume descriptor.) The file system passes the pointer to the BLK_DEV structure when calling the driver's routines. If the BLK_DEV structure is the first element, this is equivalent to a pointer to the device descriptor.

5.3.4 Creating Devices

The BLK_DEV structure should be initialized by the driver's device creation routine. A pointer to the BLK_DEV structure must be returned to the caller, so that it may then be used during *dosFsDevInit( )*, *rt11FsDevInit( )*, or *rawFsDevInit( )*.

In previous Vx960 device drivers, the device name was assigned by the driver when the device was created. This was done with a call to *iosDevAdd( )*, which installs the device in the I/O system device table. The device name was a parameter to the driver's device creation routine.

In current device drivers, *iosDevAdd( )* is not called by the device driver. Instead, the device is installed in the device table by the file system, via *dosFsDevInit( )*, *rt11FsDevInit( )*, or *rawFsDevInit( )*. Therefore, no name is assigned to the device during its creation, and the device name is no longer a parameter to the driver's device creation routine. Immediately after it is created, the device does not have a particular file system associated with it (this is done by the file system's device initialization routine).

5.3.5 Block Read and Write Routines

The routines provided by the device driver to read and write blocks must now support reads and writes of multiple blocks. This takes advantage of current disk controller hardware that supports such calls directly.

The new call interfaces for these routines are:

```
STATUS xxBlkRd (pDev, startBlk, numBlks, pBuf)
    DEVICE  *pDev;      /* pointer to device descriptor */
    int     startBlk;   /* starting block to read */
    int     numBlks;    /* number of blocks to read */
    char    *pBuf;      /* pointer to buffer of data to read */
```

and

```
STATUS xxBlkWrt (pDev, startBlk, numBlks, pBuf)
    DEVICE  *pDev;      /* pointer to device descriptor */
    int     startBlk;   /* starting block for write */
    int     numBlks;    /* number of blocks to write */
    char    *pBuf;      /* pointer to buffer to write data to */
```

NOTE: In the above examples, the subroutine names *xxBlkRd* and *xxBlkWrt* are user-supplied names, and this convention is used throughout the remainder of this section. The symbol *DEVICE* is used here to indicate the device descriptor used by the particular device driver.

The BLK_DEV structure must be the first element in the device descriptor used by the device driver. The file system passes the pointer to the BLK_DEV structure when calling the driver routine. As long as the BLK_DEV structure is the first element, these pointers are equivalent.

If your hardware does not directly support multiple-block reads and writes, the routines in your device driver must loop to perform the necessary number of transfers. For example:

```
int i;
for (i = 0;  i < numSecs;  i++)
    {
    /* perform single-block read or write */

    /* increment block number */

    /* advance buffer pointer by number of bytes per block */
    }
```

Be aware that if your device driver returns an error to the file system from a block-read or block-write routine, the file system retries the transfer, based upon the bd_retry field specified in the BLK_DEV structure. The entire transfer is retried, even if the error occurred after some of the blocks were correctly transferred.

5.3.6 I/O Control Routine

The call interface for the I/O control routine is:

```
STATUS xxIoctl (pDev, funcCode, arg)
    DEVICE  *pDev;      /* pointer to device descriptor */
    int     funcCode;   /* ioctl function code */
    int     arg;        /* function-specific argument */
```

This is the same as in previous versions of Vx960. However, in previous versions, the driver's I/O control routine was called directly by the I/O system. If an I/O control function could not be handled locally by the driver, it would then be passed to the file system. For new block device drivers, this arrangement is reversed. The I/O system now calls the file system directly for I/O control functions. If the file system is unable to service a request, the driver's I/O control routine is called.

As a result, it is now the responsibility of the device driver's I/O control routine to declare an error if it cannot handle a request. This is typically done in the *default:* case for a C-language *switch* statement. The driver's I/O control routine should set the task's error number to S_ioLib_UNKNOWN_REQUEST, and should return ERROR.

For example:

```
switch (funcCode)
    {
    case FIODISKFORMAT:
        /* do disk formatting */
        break;
    . . .
    default:
        errnoSet (S_ioLib_UNKNOWN_REQUEST);
        return (ERROR);
    }
```

5.3.7 Device-Reset Routine

The reset routine is unchanged from previous versions of Vx960. If no reset function is required by your device driver, the bd_reset field in the BLK_DEV structure should be set to NULL by the device creation routine.

5.3.8 Status-Check Routine

This is an optional routine that allows the device driver to perform any necessary functions at the beginning of open or create operations on the device. It is particularly useful for calling routines to check the status of the disk.

The call interface for the status-check routine is:

```
STATUS xxStatusChk (pDev)
    DEVICE   *pDev;          /* pointer to device descriptor */
```

This routine, if present, is called by the file system at the beginning of each ***open*()** or *creat*( ) on the device. It should return OK if the open or create may continue. If it detects a problem with the device, the routine should set *errno* to some value indicating the problem, and return ERROR. If ERROR is returned, the open or create will not be completed.

A primary use of the status-check routine is to check for a disk change on devices that do not detect the change until after a new disk has been inserted. For such devices, the routine should determine whether a new disk has been inserted. If a new disk is present, the routine should set the **bd_readyChanged** field in the BLK_DEV structure to TRUE and then return OK, allowing the open or create to continue. The new disk is mounted automatically by the file system.

If no status-check routine is needed in your device driver, the **bd_statusChk** field in the BLK_DEV structure should be set to NULL by the device creation routine.

5.3.9 Obsolete Functions

Several functions that were previously required in Vx960 block device drivers are no longer necessary. These are the routines which called the file system to perform the actual services:

- *xxCreate*( )
- *xxDelete*( )
- *xxOpen*( )
- *xxClose*( )
- *xxRead*( )
- *xxWrite*( )

Now, the I/O system calls the file system directly for such functions in response to high-level system calls (e.g., *open*( )).

5.3.10 Change in Ready Status

In previous device drivers, if a change in the device's ready status was detected (e.g., if a disk change was recognized), the device driver would call the file system's ready-change routine, *rt11FsReadyChange*( ).

To allow the driver to notify the file system in a more general way, the BLK_DEV structure contains a boolean field, **bd_readyChanged**. The device driver should set this field to TRUE whenever a change in ready status is observed, or any other condition that implies the volume should be remounted. This field is cleared by the file system; it should never be cleared by the device driver.

5.3.11 Write-Protected Media

In previous device drivers, if the device driver recognized that the disk had been write-protected, it would call the file system's mode-change routine, *rt11FsModeChange*( ), to mark the volume as read-only. This was required because the volume mode was kept in the file system's volume descriptor.

To allow the driver to notify the file system in a more general way, the volume mode is now kept in the BLK_DEV structure. The **bd_mode** field should initially be set to UPDATE when the device is created. If the disk has been write-protected, the driver should change the field's value to READ. This field may also be modified by the file system's mode-change routine.

5.4 Buffering

The Vx960 DOS file system uses a combination of internal buffering and direct use of the caller's buffer to handle large I/O transfers efficiently. The buffering mechanism for the RT-11 file system has also been improved, and speed is now comparable to the DOS file system. While RT-11 previously could read or write only a single block at a time, it is now capable of efficiently handling much larger reads and writes. In tests of the new buffering scheme, RT-11 performed up to three times a fast as in Vx960 4.0.2.

6. NETWORK ENHANCEMENTS

6.1 Optimizations

Vx960 networking facilities have been significantly improved in release 5.0.3 Depending on network configuration, users will see as much as 50 to 100 percen increase in throughput.

The major optimization that made this possible was to redesign the network buffering mechanism to use UNIX-style clusters in addition to *mbufs*. Speed was alsc increased by optimizing the socket code and by basing network code on the 4.3 BSD Tahoe release of TCP/IP (see **6.3.1 TCP/IP**).

6.2 Transmission Media

6.2.1 SLIP

Vx960 can now be networked with the host operating system using the Serial Line Interface Protocol (SLIP).

In Vx960, SLIP has been implemented as a network interface driver that can be attached to the IP software in the same manner as other network interface drivers. For more information on the use of SLIP, see the manual entry for if_sl(1).

SLIP is particularly useful for communications setups where Ethernet or other high-speed networks cannot be used, such as long-distance connections requiring phone lines. However, such communications have two fundamental restrictions. Since they rely on serial line connections, throughput is inherently slower: a maximum of 38.4Kbits/sec as compared to Ethernet's 10Mbits/sec. SLIP communications are also point-to-point, as opposed to broadcast. This means that packets can only be routed between two endpoints of a serial connection. However, if the host at either end of the serial connection has other network interfaces, such as Ethernet, and can forward packets to additional machines, it can act as a gateway between SLIP and other systems on the network.

6.3 Communications Protocols

6.3.1 TCP/IP

Vx960 TCP/IP has been upgraded to the Tahoe maintenance release of 4.3 BSD UNIX. This upgrade includes the Van Jacobsen optimizations for TCP/IP.

6.3.2 FTP Service

Vx960 can now function as an *ftp* server. In previous releases Vx960 was an *ftp* client only; all *ftp* communications relied on the UNIX host acting as server. The *ftp* daemon running on Vx960 allows *ftp* calls from a UNIX client to a Vx960 server, and from a Vx960 client to a Vx960 server. In particular, this makes it possible to boot another Vx960 system directly from a Vx960 server. On the Vx960 server, calling *ftpdInit( )* initializes the *ftp* server task. For more information see the manual entry for ftpdLib(1).

6.3.3 RPC

Vx960 RPC has been upgraded to RPC release 4.0.

6.4 Security

Release 5.0.3 offers a new security feature so that users can be required to enter a user name and password when accessing the Vx960 shell via the network with telnet or rlogin.

This facility is activated by defining INCLUDE_SECURITY in the Vx960 configuration files configAll.h or <target>/config.h. It can also be disabled at boot time by specifying the flag SYSFLG_NO_SECURITY flag bit (value = 0x20) in the flags parameter on the boot line.

The password table is initialized with a single entry with the following user name and password:

user name:	`target`
password:	`password`

Passwords are created using the supplied tool vxencrypt, which is run on the UNIX host. See the manual entry for vxencrypt(3). Users can also supply their own encryption algorithm in place of vxencrypt(3).

The Vx960 login prompt string can also be changed using *loginStringSet*( ). This can be useful to reflect system identity where there are multiple targets on a network.

6.5 New NFS Option

Vx960 offers a new configuration option for automatically mounting all exportable NFS file systems from the host system. Each file system is NFS mounted if INCLUDE_NFS_MOUNT_ALL is defined in the Vx960 configuration file. If only INCLUDE_NFS is defined, Vx960 mounts the file system from which Vx960 was booted, as in the previous release.

Certain precautions must be observed if the host file system "/" is mounted in Vx960. If you try to reference a file whose name begins with a slash, Vx960 confuses the slash with the NFS-mounted file system "/". For example, if your current directory is the DOS device DEV1: and you try to open a file on this device called /myfile, Vx960 will look only for myfile on the file system "/", unless the full device name is specified.

If "/" has been mounted because INCLUDE_NFS_MOUNT_ALL is defined, it may be more convenient to unmount "/" using *nfsUnmount*( ) in your startup script.

7. SHELL CHANGES

7.1 Task Referencing

Most Vx960 routines that take a task parameter require a task ID. However, when invoking routines interactively, specifying a task ID can be cumbersome since the ID is an arbitrary and possibly lengthy number.

To accommodate interactive use, all Vx960 shell expressions can now reference a task by either task ID or task name. The Vx960 shell attempts to resolve a task argument to a task ID; if no match is found in the system symbol table, it searches for the argument in the list of active tasks. When it finds a match, it substitutes the task name with its matching task ID. In symbol lookup, symbol names take precedence over task names.

By convention, task names are now prefixed with a "t". This avoids name conflicts with entries in the system symbol table. The names of system tasks and the default task names assigned when tasks are spawned use this convention. For example, tasks spawned by *sp( )* will be given names such as *t1*, *t2*, and *t3*. Users are encouraged to adopt this convention.

7.2 Idle Task

In 5.0.3 the function of the idle task is now internal to the kernel; it no longer appears with system tasks displayed by the *i( )* command. In previous versions, the idle task merely looped to itself using all remaining CPU cycles in an application. Removing it as a system task represents an optimization because there is no idle context to be saved and restored when an application enters or exits the idle state. The time spent taking the system out of the idle state is therefore much faster.

7.3 Task Activity Reporting: *spy*( )

The task activity report displayed by *spy*( ) now shows the time spent at kernel state, in addition to the time spent by each task. This is useful for understanding where processor time is being spent in the system.

7.4 Information Commands: *i*( ), *ti*( )

If a task's priority is inherited, this appears as "+I" in the status field when displaying task information with *i*( ). In addition, the *ti*( ) command now summarizes the task's stack and options in a more presentable form.

8. DEVELOPMENT ENVIRONMENT

8.1 ANSI Support

Vx960 now supplies ANSI C function prototype declarations for all global routines in the Vx960 header files. The ANSI prototypes are conditionally compiled; to use them, define the macro __STDC__ either in **vxWorks.h** or in the application code before any include statements. Vx960 5.0.3 provides the ANSI-specified header files **stddef.h**, **stdlib.h**, and **string.h**.

8.2 X Windows Based Source-Level Debugging Support

VxGDB 3.2 extends the line-oriented mode of the GNU Source-Level Debugger (GDB), Version 3.2, with a graphical user-interface based on the X Window System. The graphical user-interface lets you enter commands either by selecting a command button with the mouse or typing the command directly into a dialogue window. The *User's Guide to VxGDB* describes how to use the new VxGDB interface.

9. CONFIGURATION AND BOOTING

Release 5.0.3 of Vx960 adds a number of new features that improve booting procedures, including the ability to boot from another Vx960 system. New configuration and boot-line options have been added as well.

9.1 Booting from Vx960

With release 5.0.3, it is possible to boot Vx960 from another Vx960 system running an *ftp* daemon, since Vx960 can now function as an *ftp* server, as explained earlier. Calling *ftpdInit*( ) on the Vx960 server initializes the *ftp* server task. To boot from the Vx960 server, just specify the Internet address of the Vx960 server in the "host" field of the boot parameters and supply a password in the "ftp password" field. For more information see the manual entry for ftpdLib(1).

9.2 New Boot Line Fields

The Vx960 boot line now accepts the following additional fields:

Field	Meaning
tn=*targetname*	add *targetname* to host table as name of target system
s=*startupScript*	execute *startupScript* on system startup
o=*other*	unused – available for application use

The addition of a target name allows the target to know a "hostname" for itself.

In 4.0.2, a start-up script was sought with a fixed name (*host*:*bootDir*/startup.cmd) and executed if found. This was enabled at build time by the INCLUDE_STARTUP_SCRIPT option in configAll.h, or <target>/config.h and could be disabled at boot time by setting the SYSFLG_NO_STARTUP_SCRIPT option bit in the "flags" boot parameter.

In 5.0.3, the start-up script filename is specified by the "s=" boot parameter. If the parameter is missing, no start-up script is executed. As in 4.0.2, this is enabled at build time by the INCLUDE_STARTUP_SCRIPT option in configAll.h or <target>/config.h. The SYSFLG_NO_STARTUP_SCRIPT option bit is now ignored.

In 5.0.3, an optional "other" field can be specified. This field is ignored by Vx960 other than to enter it in the structure containing the system boot parameters, *sysBootParams*. This field is thus available for application use.

9.3 New INCLUDE Options

The INCLUDE options, INCLUDE_GNU_CLIB and INCLUDE_GNU_FLOATLIB, can be defined in configAll.h or <target>/config.h to force all of the GNU libraries to be loaded into the kernel. When an application is downloaded, the loader resolves the library references.

To obtain the standard C library functionality or the floating point library functionality, you can do one of the following:

- If you want the standard C library as part of your application, you can specify the -lcg option on the command line or you can specify INCLUDE_GNU_960_CLIB as a preprocessor option to the kernel configuration.
- If you want the floating point library as part of your application, you can specify the -lfpg option on the command line or you can specify INCLUDE_GNU_960_FLOATLIB as a preprocessor option to the kernel configuration.

The following table illustrates the choices described above.

	Command-line Option	Preprocessor Option
Standard C Library	-lcg	INCLUDE_GNU_CLIB
Floating Point Library	-lfpg	INCLUDE_GNU_FLOATLIB

When you define **INCLUDE_GNU_CLIB** or **INCLUDE_GNU_FLOATLIB**, do not specify the options **-lcg** or **-lfpg** respectively on the **gld960 -r** command line. If you specify these INCLUDE options, the application runs, but memory is wasted since some routines will be loaded twice -- the first time, since they already exist in the kernel and the second time, when the application is downloaded.

The following table outlines what will occur if you do or do not specify the INCLUDE options.

	Preprocessor Option Specified	**Preprocessor Not-Specified**
Command-line Option Specified	The application runs but memory is wasted	The application runs.
Command-line Option Not Specified	The application runs.	The application downloads with unresolved symbols.

For additional information on other INCLUDE options see **Table 8-1. Selectable Vx960 Options** in the *Vx960 Programmer's Guide.*

9.4 Boot ROM Compression

With the addition of new functionality in the Vx960 boot ROMs, the memory required exceeded the maximum on-board ROM capacity of some systems. In 5.0.3, the Vx960 boot ROMs are compressed to about 50% of their actual size using a binary compression algorithm. When control is passed to the ROMs on system reset or reboot, a small (8 Kbytes) decompression routine, which is *not* compressed, is executed. It then decompresses the remainder of the ROM into RAM and jumps to the start of the decompressed image in RAM. There is a short delay during the decompression (about 1 second) before the Vx960 prompt appears.

Vx960 4.0.2 runs with the Vx960 5.0.3 boot ROMs even though these ROMs contain the decompression routines. However, Vx960 4.0.2 cannot use Vx960 5.0.3 extensions on the boot line.

The compression mechanism is available to application builders who wish to compress a ROMable Vx960 application into ROM. For more information, see the manual entries for bootInit(1) and compress(3).

10. DIRECTORY AND FILE REORGANIZATION

10.1 Directory Changes

Several directories in the vx tree have been reorganized:

/usr/vx/h/drv	contains hardware-specific headers for drivers, etc.
/usr/vx/h/i960	contains versions of architecture-dependent header files[2]
/usr/vx/bin/*host*	contains shell scripts and host-specific executables of Vx960 tools

10.2 Hosts Supported

The following is a list of the hosts supported by Vx960. If the operating system for your host is upwardly compatible, the existing GNU/960 tools may be supported. The source for all Vx960 binaries are provided to allow other hosts to be supported. If the GNU/960 tools are supported, Vx960 supports that host.

Processor	Operating System	Host Type
Sun-4	SunOs4.0.3c	Sun4
Sun-3	SunOs4.0.3c	Sun3
Sun-386i	SunOs4.0.1	386i
VAX 8600	Ultrix-32 V3.1 (Rev.9)	
HP9000/300	HP/UX V7.0	hp300
Compaq Deskpro 386/33	Intel System V R3.2	i386v
DECStation 2100	ULTRIX V2.0 (Rev 7)	
IBM RS/6000	AIX Version 3.1	rs6000
HP/Apollo Series 400	Domain/OS SR 10.3	ap400
i386 systems	System V R3.2	i386v

2. Note that /usr/vx/h still contains architecture-independent versions, which include appropriate architecture-dependent versions (e.g., /usr/vx/h/dsmLib.h includes /usr/vx/h/i960/dsm960-Lib.h depending on the definition of the CPU variable).

10.3 Renamed Libraries

The Vx960 libraries in /usr/vx/lib/*arch* were renamed to be more consistent:

all.a	-	formerly **vxWorks.a**
config.a	-	formerly **vxConfig.a**
net.a	-	formerly **network.a**
rpc.a	-	formerly **rpclib.a**

10.4 New Libraries

The following libraries were added:

drv.a	-	drivers
netif.a	-	network interface drivers
i960.a	-	i960 microprocessor specific modules

10.5 Unsupported

Several hardware support packages that are no longer supported by Intel have been moved to **/usr/vx/unsupported**. These include several board support packages, network drivers, and device drivers. Also, some software developed for hardware that is not suited to an i960 environment is no longer available.

Some tools developed for in-house use are also supplied "as is" in **/usr/vx/unsupported/tools**. They are:

mangen	- generate the manual entries for a specified library
mofset	- make structure member offset definitions header
mkmk	- generate a makefile from makefile skeletons and source files
shadow	- create a private workspace with links to shared files

Manual entries for these tools can be generated using **mangen**:

```
    mangen tool filename
```

These tools can be copied (minus the .sh extension) to a directory in your executable search path as desired. You are free to use these tools, but Intel does not provide support for them.

11. NEW MODULES AND ROUTINES

The following list is a summary of new libraries, drivers, routines, and tools introduced with Vx960 5.0.3:

New Libraries

dirLib.c	- POSIX directory handling library
opendir( )	- open a directory for searching
readdir( )	- read one entry from a directory
rewinddir( )	- reset position to start of directory
closedir( )	- close directory
fstat( )	- get file status information
stat( )	- get file status information (using pathname)
dosFsLib.c	- MS-DOS media-compatible file system library
dosFsConfigInit( )	- initialize *dosFs* volume configuration structure
dosFsDateSet( )	- set current date
dosFsDateTimeInstall( )	- install user-supplied date/time function
dosFsDevInit( )	- associate block device with dosFs file system functions
dosFsInit( )	- prepare to use *dosFs* library
dosFsMkfs( )	- initialize device and create *dosFs* file system
dosFsModeChange( )	- modify mode of *dosFs* volume
dosFsReadyChange( )	- notify *dosFsLib* of a change in ready status
dosFsTimeSet( )	- set current time
dosFsVolUnmount( )	- unmount a *dosFs* volume

ftpdLib.c	- File Transfer Protocol server
ftpdTask( )	- FTP server daemon task
ftpdInit( )	- initialize the FTP server task
ftpdDelete( )	- clean up and finalize FTP server task
loginLib.c	- user login/password subroutine library
loginInit( )	- initialize the login table
loginUserAdd( )	- add a user to the login table
loginUserDelete( )	- delete a user entry from the login table
loginUserVerify( )	- verify a user name and password in the login table
loginUserShow( )	- display the user login table
loginPrompt( )	- display a login prompt and validate user entry
loginStringSet( )	- change the login string
loginEncryptInstall( )	- installation of *encryption* routine
loginDefaultEncrypt( )	- password *encryption* routine
msgQLib.c	- message queue library
msgQCreate( )	- create and initialize a message queue
msgQDelete( )	- delete message queue
msgQInfoGet( )	- get list of task IDs that are blocked on message queue
msgQShow( )	- show information about a message queue
msgQSend( )	- send a message to a message queue
msgQReceive( )	- receive a message from a message queue
netShow.c	- network related information display routines
ifShow( )	- display available network interfaces
icmpstatShow( )	- print out statistics for ICMP
inetstatShow( )	- show all active connections for Internet protocol sockets
ipstatShow( )	- display IP statistics
mbufShow( )	- report *mbuf* statistics
netShowInit( )	- initialize network show routines.
tcpstatShow( )	- show all statistics for TCP protocol
udpstatShow( )	- print out statistics for UDP
arptabShow( )	- show a list of known ARP entries
rawFsLib.c	- raw disk file system library
rawFsDevInit( )	- associate block device with raw volume functions
rawFsInit( )	- prepare to use raw volume library

rawFsModeChange( )	- modify mode of raw device volume
rawFsReadyChange( )	- notify *rawFsLib* of a change in ready status
rawFsVolUnmount( )	- disable a raw device volume
scsiLib.c	- Small Computer System Interface (SCSI) library
scsiPhysDevDelete( )	- delete a SCSI physical device structure
scsiPhysDevCreate( )	- create a SCSI physical device structure
scsiShow( )	- list the physical devices attached to a SCSI controller
scsiBlkDevCreate( )	- define a logical partition on a SCSI block device
scsiBlkDevInit( )	- initialize fields in a SCSI logical partition
scsiBusReset( )	- pulse the reset signal on the SCSI bus
scsiTestUnitRdy( )	- issue a TEST_UNIT_READY command to a SCSI device
scsiFormatUnit( )	- issue a FORMAT_UNIT command to a SCSI device
scsiInquiry( )	- issue an INQUIRY command to a SCSI device
scsiModeSelect( )	- issue a MODE SELECT command to a SCSI device
scsiModeSense( )	- issue a MODE SENSE command to a SCSI device
scsiReadCapacity( )	- issue a READ_CAPACITY command to a SCSI device
scsiRdSecs( )	- read sector(s) from an SCSI block device
scsiWrtSecs( )	- write sector(s) to an SCSI block device
scsiReqSense( )	- issue a REQUEST SENSE command to a device and read results
scsiIoctl( )	- do a device-specific control function
selectLib.c	- UNIX BSD 4.3 select library
selectInit( )	- initialize the select library
select( )	- pend on a set of file descriptors
selWakeup( )	- wake up a task pended in select
selWakeupAll( )	- wake up everyone on a select wake up list
selNodeAdd( )	- add a copy of this wake up node to the wake up list
selNodeDelete( )	- find a node on a wake up list and delete it
selWakeupListInit( )	- initialize a select wake up list
selWakeupListLen( )	- return number of nodes currently on wake up list
selWakeupType( )	- return the type of the given SEL_WAKEUP_NODE
semBLib.c	- Vx960 binary semaphore library
semBCreate( )	- create and initialize a binary semaphore

semCLib.c	- Vx960 counting semaphore library
semCCreate( )	- create and initialize a counting semaphore
semMLib.c	- Vx960 mutual exclusion semaphore library
semMCreate( )	- create and initialize a mutual exclusion semaphore
semOLib.c	- Vx960 4.0 binary semaphore library
semCreate( )	- create and initialize a binary semaphore
semClear( )	- take binary semaphore if semaphore is available
taskArchLib.c	- architecture specific task management routines for kernel
taskRegsShow( )	- print contents of a task's registers

New Routines in Existing Libraries

bindresvport( )	- bind a socket to a privileged IP port
bootStringToStruct( )	- interpret the boot parameters from the boot line
bootStructToString( )	- construct a boot line
fioFormatV( )	- format processor
getsockopt( )	- get socket options
getwd( )	- get current default path
ifAddrGet( )	- get Internet address of a network interface
ifBroadcastGet( )	- get broadcast address for network interface
ifDstAddrGet( )	- get point-to-point peer's Internet address
ifDstAddrSet( )	- define address for the opposite end of a point-to-point link
ifFlagChange( )	- change network interface flags
ifFlagGet( )	- get network interface flags
ifFlagSet( )	- specify flags for a network interface
ifMaskGet( )	- get the subnet mask for a network interface
ifMetricGet( )	- get the metric for a network interface
ifMetricSet( )	- specify a network interface hop count
ifRouteDelete( )	- delete routes associated with a network interface
intLockLevelGet( )	- get the current interrupt lock-out level
intLockLevelSet( )	- set the current interrupt lock-out level
ll( )	- do long listing of directory contents
lsOld( )	- list contents of RT-11 directory

nfsMount402( )	- mount NFS directory as in 4.0.2
nfsMountAll( )	- mount all file systems exported by specified host
rt11FsMkfs( )	- initialize device and create RT-11 file system
semFlush( )	- unblock every task pended on a semaphore
semInfo( )	- get list of task IDs that are blocked on semaphore
symFindByValueAndType( )	- find a symbol in a symbol table, given the value
symRemove( )	- remove and delete a symbol from a symbol table
symTblDelete( )	- delete a symbol table
taskCreateHookShow( )	- show create routines
taskDeleteHookShow( )	- show delete routines
taskInit( )	- initialize a task with stack at specified address
taskSafe( )	- make calling task safe from deletion
taskSwitchHookShow( )	- show switch routines
taskTerminate( )	- terminate a task
taskUnsafe( )	- make calling task unsafe from deletion
unlink( )	- delete a file
usrStartupScript( )	- make shell read initial startup script file
usrStartupScript402( )	- execute startup script from network
vfdprintf( )	- print formatted string with variable argument list to specified field
vprintf( )	- print formatted string with variable argument list to standard output
vsprintf( )	- put formatted string with variable argument list in specified buffer
wdDelete( )	- delete a watchdog timer

New Drivers

memDrv	- install memory driver
memDevCreate( )	- create a memory device
memDrv( )	- install memory driver

New Tools

aoutToBin.c	- strip text and data segments from a.out file

compress.c	-	general purpose file compression utility
picLib.c	-	jump program support routines
vxencrypt.c	-	*encryption* program for *loginLib*