Introduction to VxWorks I/O System

Character I/O

Block I/O
11.1 Introduction

Character I/O

Block I/O

Overview of devices, drivers, and the I/O system.

Installing drivers

Creating devices
Many device drivers perform similar operations, namely:
- Write data to a device.
- Read data from a device.

VxWorks I/O system provides a simple device-independent and portable interface to these device drivers.

I/O system supports redirection.

`select()` function is available.

A device driver is a software module which manipulates some device.
Driver Installation

- A device driver must be installed in the I/O system.
- Then VxWorks can call driver-specific routines when generic routines are called: e.g. calling `xxRead()` when `read()` is called on an XX device.
- Installation done automatically for VxWorks drivers included in VxWorks image:
  - `ttyDrv()`.
  - `pipeDrv()`.
- Must call driver’s `xxDrv()` routine before using a third party driver.
Device Creation

<table>
<thead>
<tr>
<th>Device Driver</th>
<th>Device Creation Routine</th>
<th>Device Name Begins With</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipes</td>
<td>\texttt{pipeDevCreate()}</td>
<td><code>/pipe/</code></td>
</tr>
<tr>
<td>Serial</td>
<td>\texttt{ttyDevCreate()}</td>
<td><code>/tyCo/</code></td>
</tr>
<tr>
<td>remote</td>
<td>Networking chapter</td>
<td>End with <code>:</code></td>
</tr>
<tr>
<td>local</td>
<td>Block I/O section</td>
<td><code>/</code></td>
</tr>
<tr>
<td>third party</td>
<td>\texttt{xxDevCreate()}</td>
<td><code>/</code></td>
</tr>
</tbody>
</table>

- These routines each initialize instances of a particular device type.

- The names given above are recommended WRS conventions.

- Example:

  ```
  -> devs
  
  drv   name
  0     /null
  1     /tyCo/0
  1     /tyCo/1
  4     columbia:
  2     /pipe/dream
  2     /pipe/cleaner
  5     /vio
  ```
File Descriptors

- Integer that identifies a file (file or device).
- Assigned by `open()` or `creat()`.
- Used by `read()`, `write()`, `ioctl()` and `close()` to specify file.
- File descriptor table is global.
- Table size defined by the symbolic constant `NUM_FILES` (default 50). This is a parameter of the component /operating system components/IO system components/IO system.

- We will use the term file to refer to either a file on a device with a file system (e.g., a hard disk) or to a stream-oriented device (e.g., serial channel or pipe).
- `iosFdShow()` will display the file descriptor table. Example:

  -> **iosFdShow**

<table>
<thead>
<tr>
<th>fd</th>
<th>name</th>
<th>drv</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>/tyCo/0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>(socket)</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>/pipe/dream</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>/pipe/cleaner</td>
<td>2</td>
</tr>
</tbody>
</table>
I/O System

Introduction

11.2 Character I/O

Block I/O

Using file descriptors, select(), and I/O libraries
Standard Input, Standard Output, and Standard Error

- The first three file descriptors are predefined by the system and are never reassigned by the system.

<table>
<thead>
<tr>
<th>File Descriptor</th>
<th>0</th>
<th>STD_IN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>STD_OUT</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>STD_ERR</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- These three file descriptors (0 - 2) are never returned by `creat()` or `open()`.

- Mappings for the standard file descriptors 0, 1, and 2 are actually maintained outside of the file descriptor table. The file descriptor table starts at index 3 and extends to index `NUM_FILES + 2`.

- Global assignments may be changed after system start-up using

  ```c
  void ioGlobalStdSet (stdFd, newFd)
  ```

  Example:

  ```c
  fd = open ("columbia:/vx/errLog", O_RDWR, 0);
  ioGlobalStdSet (STD_ERR, fd);
  ```

- Redirect I/O on a per-task basis with

  ```c
  void ioTaskStdSet (taskId, stdFd, newFd)
  ```

- Use `ioGlobalStdGet()` to find the real file descriptor corresponding to one of the global standard descriptors; use `ioTaskStdGet()` to find the real file descriptor to which one of the standard descriptors is redirected for a specific task.
Basic I/O Routines

int open (name, flags, mode)

flags: O_RDONLY, O_WRONLY, O_RDWR, O_TRUNC, O_CREAT

mode: Permissions for NFS device

STATUS close (fd)

Tasks must close files when they are no longer needed. File descriptor table is fixed size.

int read (fd, buffer, nBytes)
int write (fd, buffer, nBytes)

May block; returns number of bytes read or written.

int ioctl (fd, command, arg)

Allows execution of a device specific command. Valid ioctl() commands are listed in driver help page.

Note: creat() and remove() are used for file systems.

Examples:

• Create a file on a remote host, if it doesn’t exist, and set write-only flag:
  
  fd = open ("columbia:/u/team5/myFile", O_CREAT | O_WRONLY, 0);

• Set the baud rate of a serial device:
  
  status = ioctl (fd, FIOBAUDRATE, baudRate);

• Get number of messages in a pipe:
  
  status = ioctl (fd, FIONMSGS, &nMsgs);

• Get current options for a serial device:
  
  options = ioctl (fd, FIOGETOPTIONS, 0);
• `select()` allows a task to wait for activity on a set of file descriptors.

• Requires driver support:
  • VxWorks pipes, sockets and serial device drivers support `select()`.
  • Third party drivers may also support `select()`.
  • Also used to pend with a timeout.

```
struct fd_set
```

• Used by `select()` to specify file descriptors.

• Conceptually an array of bits, with bit N corresponding to file descriptor N.

• Bits are manipulated via a collection of macros:

  ```
  FD_SET (fd, &fdSet)        Sets the bit
  FD_CLR (fd, &fdSet)        Clears the bit
  FD_ISSET (fd, &fdSet)      Returns TRUE if the fd bit is set, else FALSE.
  FD_ZERO (&fdSet)           Clears all bits.
  ```
The `select` function is used to wait for I/O activity on any of the file descriptors associated with the `pReadFds`, `pWriteFds`, or `pExceptFds` pointers.

**Syntax**

```c
int select (width, pReadFds, pWriteFds, pExceptFds, pTimeOut)
```

**Parameters**

- `width`: Number of bits to examine in `pReadFds` and `pWriteFds`.
- `pReadFds`: A `struct fd_set` pointer for the file descriptors we wish to read.
- `pWriteFds`: A `struct fd_set` pointer for the file descriptors we wish to write.
- `pExceptFds`: Not implemented.
- `pTimeOut`: Pointer to a `struct timeval`, or `NULL` to wait forever.

**Return Value**

- Returns the number of file descriptors with activity, 0 in the case of a timeout, or `ERROR` if a driver `select()` support routine failed for one of the file descriptors.

**Notes**

- `select()` modifies `pReadFds` and `pWriteFds`:
  - User sets bits corresponding to file descriptors before calling `select`.
  - Select blocks until one of the file descriptors is ready.
  - Select clears the bits in the `struct fd_set` for which the corresponding file descriptor is not ready.

- See the `select()` example in Appendix A.
VxWorks provides two libraries for higher level I/O functionality:

- **ansiStdio**: Buffered I/O package.
- **fioLib**: Formatted, non-buffered I/O.
Buffered I/O and File Pointers

- **ansiStdio** routines use *file pointers* (pointers to `FILE` structures) instead of file descriptors. The `FILE` data structure, typedef’ed in `stdio.h`, contains:
  - The underlying file descriptor.
  - Pointers, etc., for managing the file buffers.
- Stdio buffers are not protected with semaphores. Two tasks should not use the same fp at the same time.

- For a complete list of routines see **ansiStdio** man page.
- Include the `stdio.h` header file.
- `stdin`, `stdout`, and `stderr` are file pointers created using file descriptors 0, 1, and 2.

<table>
<thead>
<tr>
<th>File Descriptors (vxWorks.h)</th>
<th>FILE Pointers (stdio.h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STD_IN</td>
<td>stdin</td>
</tr>
<tr>
<td>STD_OUT</td>
<td>stdout</td>
</tr>
<tr>
<td>STD_ERR</td>
<td>stderr</td>
</tr>
</tbody>
</table>
Why Are the Buffers Used?

Private buffers minimize driver access:

- Stdio buffers are **not** protected with semaphores

- Driver access can be expensive because driver-controlled buffers are usually protected with semaphores.

- Standard I/O routines create private buffers, which reduce driver access.

- On VxWorks, driver access is fast. Standard I/O buffers may or may not increase performance, depending on the size of each data access and the driver implementation.
Formatted I/O

• `fioLib` contains routines for non-buffered formatted I/O.

• Includes `printf()` and `sprintf()` which are normally part of `stdio`.

• `fioLib` allows `ansiStdio` to be excluded without losing functionality.

• See `fioLib` for a complete list of routines.
Standard I/O and FIO Components

<table>
<thead>
<tr>
<th>Library</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffered I/O</td>
<td>/operating system components/ANSI C components (libc)/ANSI stdio</td>
</tr>
<tr>
<td>Formatted I/O</td>
<td>/operating system components/IO system components /formatted IO</td>
</tr>
</tbody>
</table>

- Other ANSI library components can be found in /operating system components/ANSI C components (libc)/. For example, `ansiCtype`, `ansiString`, `ansiAssert`, `ansiTime`, and `ansiMath`. These libraries conform to ANSI X3.159-1989.
I/O System

Introduction

Character I/O

11.3 Block I/O

Using RAM disks

Using a DOS File System or a Raw File System on a local disk
For local file storage, one needs:

- Block driver.
- File system routines.

Block driver:
- Reads/writes disk sectors.
- Formats disk (marks sector boundaries).

File system routines manage files on a disk. For example, dosFs uses a file allocation table to keep track of which disk sectors comprise a file.

SCSI sequential devices (e.g. tape drives) are also accessed via an intermediate “file system” layer provided by tapeFsLib.
Local File Systems

- Available block device drivers:
  - `ramDrv` Simulates a disk in memory.
  - `scsiLib` Supports SCSI random-access devices.
  - `tffsDrv` True Flash File System (TrueFFS) (optional product)
  - `xxDrv` Third party block drivers.

- Available file system libraries:
  - `dosFsLib` MS-DOS compatible.
  - `rawFsLib` Supports disk as a single file.
  - `rt11FsLib` For backward compatibility.
  - `cdromFsLib` ISO 9660 CD-ROM read-only file system.
  - `tapeFsLib` For SCSI sequential devices.

- `cdromFsLib` implements an ISO 9660 compliant read-only file system which can be used with any block driver. See the library help pages for an example of configuring a CD-ROM file system on a SCSI device.
- See the SCSI appendix for information on configuring the tape file system.
- The RT-11 file system can be considered obsolete, but is provided (for the moment) to support legacy systems.
Initialization and Use

- **To initialize a local file system:**
  1. Call a block driver’s device creation routine to initialize a `BLK_DEV` data structure describing how to access the device.
  2. Initialize file system configuration parameters if necessary.
  3. Call a file system initialization routine to initialize a local file system for the device corresponding to the `BLK_DEV` structure.
  4. Load configuration information onto physical medium if necessary.
- With the local file system initialized, use `creat()`, `read()`, `write()`, etc. to manipulate files on the file system.
Block Driver Example

BLK_DEV * ramDevCreate (ramAddr, bytesPerBlk, blksPerTrack, nBlocks, blkOffset)

- ramAddr: Memory location of ram disk (0 = malloc()).
- bytesPerBlk: Number of bytes per block.
- blksPerTrack: Number of blocks per track.
- nBlocks: Size of disk in blocks.
- blkOffset: Number of blocks to skip. Typically zero.

- Returns a pointer to a BLK_DEV structure describing the RAM disk, or NULL on error.

- ramDrv configured by adding the component /operating system components/IO system components/RAM disk driver.

Example: Create RAM disk and initialize DOS file system with default parameters

```c
-> pBlkDev = ramDevCreate (0, 512, 400, 400, 0)
-> dosFsMkfs ("/RAM1", pBlkDev)

/* Create and write to a file. Flush to RAM disk */
-> fd = creat("/RAM1/myFile", 2)
-> writeBuf = "This is a string.\n"
-> write(fd, writeBuf, strlen(writeBuf)+1)
-> close(fd)

/* Open file and read contents. */
-> readBuf = malloc(100)
-> fd = open("/RAM1/myFile", 2)
-> read(fd, readBuf, 100)
-> printf readBuf

This is a string.
```
# Configuring Local File Systems?

| dosFs | Include the component:  
|       | /operating system components/IO system components/DOS filesystem.  
|       | **dosFsDevInit** (name, pBlkDev, pVolConfig)  
|       | **dosFsMkfs** (name, pBlkDev) |
| rawFs | Add the component:  
|       | /operating system components/IO system components/raw filesystem.  
|       | Default NUM_RAWFS_FILES = 5. Fastest, least structured file system.  
|       | **rawFsDevInit** (devName, pBlkDev) |

- **target/src/usr/usrLib.c** implements a routine `diskFormat()`, which may be used to format a disk specified by name.
**Configuration: New DOS File System**

To create a new DOS file system with custom configuration parameters:

1. \( \text{pBlkDev} = \text{xxDevCreate}(...) \);
2. Initialize \text{DOS_VOL_CONFIG} structure.
3. \( \text{dosFsDevInit} \("/DOS", \text{pBlkDev}, \text{pConfig}) \);
4. \( \text{fd} = \text{open} \("/DOS", \text{O_WRONLY}, 0) \);
5. \( \text{ioctl} \:(\text{fd}, \text{FIODISKFORMAT}, 0); /* \text{if necessary} */ \)
6. \( \text{ioctl} \:(\text{fd}, \text{FIODISKINIT}, 0) \);
7. \( \text{close} \:(\text{fd}) \);

- \( \text{xxDevCreate} \) is the driver function used to create a logical partition on the appropriate block device. \( \text{ramDevCreate} \) and \( \text{scsiBlkDevCreate} \) are examples.
- \( \text{dosFsDevInit} \) initializes only the data structures (FAT, directory, etc.) used to manage the file system. Nothing is written to the disk.
- The \( \text{FIODISKFORMAT} \) command rewrites low-level sector marks on a block device.
- The \( \text{FIODISKINIT} \) command writes the new file system out to disk.
- To format a disk from the shell, you can use \( \text{diskFormat} \).
- To write the new file system to disk, you can use \( \text{diskInit} \).
- A simpler method of initializing an already formatted disk is \( \text{dosFsMkfs} \), which creates a file system with a default volume configuration.
Configuration: Using an Existing DOS File System

- To remount an existing file system:
  1. `pBlkDev = xxDevCreate(...);`
  2. `pDesc = dosFsDevInit("/DOS", pBlkDev, NULL);`
  3. `dosFsVolOptionsSet(pDesc, options); /*if needed*/`

- Typically, file systems are configured in startup code.

- Use `dosFsVolOptionsSet()` only if it is necessary to enable the `DOS_OPT_AUTOSYNC` or the `DOS_OPT_CHANGENOWARN` options.

- Two other options for the DOS file system are associated with the Network File System (NFS). `DOS_OPT_EXPORT` is specified to export a DOS file system via NFS, and `DOS_OPT_LOWERCASE` is used to force file names into a lowercase format (for PC-based NFS clients). See the Networking chapter for more information on the usage of `DOS_OPT_EXPORT`. 
Contiguous File Support

• To pre-allocate contiguous disk space for a file:
  
  ```c
  ioctl (fd, FIOCONTIG, numBytes)
  ```

  Must be called before anything is written to file.

• Example:
  
  ```c
  fd = creat ("/dos1/myDir/myFile", O_RDWR);
  status = ioctl (fd, FIOCONTIG, 0x10000);
  ```

• To obtain the largest contiguous block available, set `numBytes` to `CONTIG_MAX`.

• Pre-allocated space may be reclaimed with:
  
  ```c
  ioctl (fd, FIOTRUNC, newLength)
  ```

• When using `CONTIG_MAX`, the size actually allocated can be determined with `fstat()` or `ll()`.

• If the requested contiguous space is not available on the disk, `ioctl()` returns `ERROR`, and `errno` is set to `S_dosFsLib_NO_CONTIG_SPACE`.

• To find the size of the largest available contiguous block (without allocating it):
  
  ```c
  ioctl (fd, FIONCONTIG, &maxContigBytes)
  ```
Caveat: DOS Data Integrity

- To improve performance, FAT and directory changes are not flushed to disk immediately.

- To flush FAT and directory changes (before removing media), use `dosFsVolUnmount()`.

- Alternatively, set options in the `dosvc_options` field of a partition’s `DOS_VOL_CONFIG` structure:
  - `DOS_OPT_AUTOSYNC` Flush FAT and directory changes.
  - `DOS_OPT_CHANGENOWARN` Remount dosFs on `open()` or `creat()`.
  - Setting these options will degrade performance.

- Flush file changes with `ioctl(fd, FIOSYNC, 0)`.

- Ordinarily, changes to the FAT and directory are not written to disk until a file is closed, flushed, or deleted, or a new directory is created.

- Setting `DOS_OPT_AUTOSYNC` and `DOS_OPT_CHANGENOWARN` will help prevent data loss from unexpected events like power-failures and media removal, but will degrade performance. Avoid them if possible.

- `DOS_OPT_AUTOSYNC` and `DOS_OPT_CHANGENOWARN` options are not stored on the disk and must consequently be reset for disk remounts.

- `dosFsLib` will use JAN-01-1980 00:00:00 as a time stamp on all files unless you install a date/time function (or call `dosFsDateSet()` and `dosFsTimeSet()` at periodic intervals).

- To provide `dosFsLib` with a call-back routine for obtaining data-time information, use `dosFsDateTimeInstall()`. 
Caveat: DOS file names and PC Compatibility

• By default, standard 8+3 DOS file names are used.

• Enable UNIX-style file names by setting DOS_OPT_LONGNAMES option bit in DOS_VOL_CONFIG data structure.

• A disk initialized on a PC will work with VxWorks.

• For PC compatibility, a disk initialized with VxWorks
  • Must use correct media byte.
  • Cannot have UNIX-style file names enabled with DOS_OPT_LONGNAMES.

• The “UNIX-style” file names are up to 40 ASCII characters long, and case-sensitive.
Summary

- Basic I/O routines:
  
  \[
  \text{open( )} \quad \text{close( )} \\
  \text{read( )} \quad \text{write( )} \\
  \text{ioctl( )} \quad \text{creat( )} \\
  \text{remove( )}
  \]

- `select()` allows a task to pend, with timeout, waiting for read or write activity on a set of file descriptors.

- Buffered I/O (`ansiStdio`) built on top of basic I/O using file pointers. Not reentrant.

- Formatted Non-buffered I/O (`fioLib`). Smaller than `ansiStdio` and reentrant.
Summary

- DOS and RAW file systems for:
  - RAM disk.
  - SCSI devices.
  - Custom block devices.

- Initialization of a file system on a block device:
  - Create block device
  - Configure file system using `BLK_DEV` structure returned by `xxDevCreate()`.
  - Format disk (if necessary)