RAMDISK: A Sample User-Defined C I/O Driver

Brett Huber *Software Development Systems*

ABSTRACT

The run-time support library (RTS) supplied with each TI DSP C/C++ compiler provides C language I/O (CIO) in the form of the high–level functions such as `fopen` and `printf`, and the low–level functions such as `open` and `write`, which use device drivers to complete CIO requests. The RTS provides a way for the user to define a new device driver. This provides an easy way to use the sophisticated buffering of the high–level CIO functions on an arbitrary device. This application report presents a sample implementation of a user–defined device driver.

Contents

Trademarks are the property of their respective owners.

1 Introduction

The run-time support library (RTS) supplied with each TI DSP C/C++ compiler provides C language I/O (CIO) in the form of high-level functions such as `fopen` and `printf`, and low-level functions such as `open` and `write`. These functions behave exactly as their familiar counterparts in other C implementations. However, the default implementation relies on the presence of a host machine to actually perform the operations; once the DSP is operated standalone, these functions simply don't work. The RTS provides a way for the user to define a new device driver, supplied by the user, which can drive a peripheral. CIO can be directed to this device, and external communication is restored.

1.1 Low-level Functions

Internally, CIO is based on seven functions: `open, close, read, write, lseek, unlink`, and `rename`. These are called the low-level functions; all high-level operations are broken down into one or more low-level function calls. On Unix systems, these seven are system calls, whose functionality is provided by the operating system. Internal to the operating system, these system calls are directed to one of several devices based on the name of the file. The name might not represent a disk file; it need only make sense to the driver intended to handle it. For instance, a request to open a file named "/dev/sound" might be directed to the sound card device, and a file named "regular" might be directed to the hard disk device. The file abstraction is an important concept in Unix programming; most objects are treated as, and manipulated like, disk files. Not everything supports all the operations that files do, but the file interface provides a convenient abstraction.

Each distinct device has its own device driver which handles all requests made to that device. Although the TI RTS does not place these seven functions in an operating system, they follow the same model by calling a device driver to handle all reqeusts. By default, requests go to the HOST device, which simply passes all requests through to the host via special handshaking with the debugger.

Accessing a peripheral through CIO is simply a matter of writing a device driver for it and installing it using the RTS function, `add_device`. The user must define the seven basic functions (`open, close, read, write, lseek, unlink, rename`). These are expected to behave just like the Unix functions, but every detail does not need to be supported. Further, not all of them need actually do anything; for instance, it may not make any sense to `lseek` on an antenna, so `lseek` might always just return a failure code. A device driver to toggle an output pin might only support `open, close` and `write`, with the other functions implemented as stubs which just return an error code.

2 Implementing the RAMDISK Device Driver

This application note and the accompanying source code present a sample implementation of a user-defined device driver. The RAMDISK device attempts to follow the Unix devices as closely as possible, and even supports file descriptors which point to unlinked files; however, since the RTS low-level functions don't use the value of errno from the device driver, RAMDISK does not set errno. The RTS low-level interface also does not support setting the file access mode.

The call to add the RAMDISK device driver looks like this.

The `flag _MSA` means that multiple streams (files) are allowed to be open at once. If only one stream at a time is allowed, use _SSA.

```
add_device(”ramdisk”, _MSA, RAM_open, RAM_close, RAM_read, RAM_write,
            RAM_lseek, RAM_unlink, RAM_rename);
```

CIO requests are directed to a specific device by calling fopen or open with a filename of device:filename.

```
FILE *file = fopen(”ramdisk:samplefile”, ”w”);
```

Each of the functions should return –1 on failure, or if the operation isn't implemented on that device.

2.1 open and close Functions

```
int RAM_open(const char *path, unsigned flags, int llv_fd);
int RAM_close(int dev_fd);
```

The device driver interface is based on file descriptors. File descriptors are integers which are handles representing distinct streams open to a file. There may be more than one descriptor open to each file. The device driver must keep track of its own file descriptors, which will be used by the low-level functions to read and write to the device. Each open stream must have a unique device file descriptor, but since the RTS low-level functions map this to a distinct low-level file descriptor, you don't need to worry about any other device's file descriptors.

The function `open` creates a new stream to a file and returns a new file descriptor to represent that stream. For a device that can have multiple files open at the same time, the device driver must maintain a mapping of file descriptors to actual files. If there can only be one instance of a file, the file descriptor is mostly ignored, but `open` must return a non-negative integer such as zero.

One difference between the interface required by TI RTS and the Unix function `open` is that the third argument to open is the RTS's file descriptor rather than the file access mode. A device driver may choose to use this file descriptor as its own file descriptor rather than generating its own; however, it must still map this file descriptor to the file. (NOTE: There is a bug [SDSsq24995] in RTS versions before Code Composer Studio 2.2; device drivers which must work with older versions must use the file descriptor provided by `open`). The TI RTS does not support file access mode.

When a file is newly opened, the file position pointer is set to the beginning of the file. There can be multiple streams open to the same file, so each stream will have its own file position pointer. Reads and writes occur at the file position pointer, and advance the pointer by as many bytes as are read or written. The function `lseek` sets this pointer to an arbitrary value. For devices which do not support `lseek`, an actual file position pointer is unnecessary, but RAMDISK does support `lseek`, so we must provide this pointer.

There are two flags to `open` that need special support: O_CREAT and O_TRUNC. Other flags such as O_RDONLY may simply be stored for later testing by `read` and `write`. O_CREAT means the file should be created if it doesn't exist, and O_TRUNC means the file should be truncated to zero length before any other operation. Depending on the specific devices, these flags might be ignored.

The `close` function marks the file descriptor as unused (so that it is available for future calls to `open`), and may need to delete the file if it has been unlinked. On Unix, an unlinked file remains on disk until its last file descriptor is closed; thus, it is possible to open a file, unlink it, and still be able to read and write from the now nameless file. If this functionality is not implemented, `close` need not attempt to delete the file.

Remember that each stream must have its own copy of a file position pointer and open flags.

2.2 `read` and `write` Functions

```
int RAM_read(int dev_fd, char *buf, unsigned count);
int RAM_write(int dev_fd, const char *buf, unsigned count);
```

The function `read` copies into `buf` at most `count` bytes from the file represented by `dev_fd`, starting at the file position pointer. It will not go beyond the end of the file; if a `read` request would do so, it is truncated to the number of available bytes. At the end of the file, then, any `read` request will be truncated to zero bytes, and `read` will return zero.

The function `write` copies from `buf` `count` bytes into file `dev_fd`, starting at the file position pointer. The function `write` will go beyond the end of the file; in this case, it will increase the size of the file to be just large enough to hold the request.

Both functions advance the file position pointer by the number of bytes read or written. They also return the number of bytes read or written, or –1, on error.

A file can be open in one of three read/write modes: O_RDONLY, O_WRONLY, or O_RDRW; these allow reads only, writes only, or reads and writes respectively. If none is specified, the default is O_RDONLY. Again, the device driver may choose to ignore these values. RAMDISK respects these modes, and will not allow writes to a read–only device or vice versa.

The flag O_APPEND needs special support. For a file opened with O_APPEND, the file position pointer must be set to the end of the file at the beginning of every call to write.

2.3 `lseek` Function

```
off_t RAM_lseek(int dev_fd, off_t offset, int origin);
```

The function `lseek` should check to make sure the user isn't attempting to seek to a point before the beginning of the file, and should return –1. However, it is perfectly legitimate to seek beyond the end of the file. Such seeks do not change the size of the file; they merely set the pointer.

If the `origin` argument is SEEK_SET, `lseek` sets the file position pointer to the value `offset`. If the origin argument is SEEK_CUR, `lseek` advances the file position pointer by the value `offset`. If the `origin` argument is SEEK_END, `lseek` sets the file position pointer to the value of the current size of the file plus `offset`.

If a `write` occurs when the pointer is past the end of the file, `write` must increase the size of the file and pad with zeros. On Unix, these zeros are typically not actually written to disk; there may be a gap of nothing between valid data in the file. A user-defined device driver may do this too, but RAMDISK actually stores the zeros for simplicity.

2.4 `unlink` Function

```
int RAM_unlink(const char *path);
```

In essense, `unlink` simply deletes the named file. However, if the device allows continued access to an open file which has been unlinked, `unlink` must only set a flag in the file indicating it has been unlinked; the actual deletion will be deferred to the function `close` when its last open file descriptor is closed.

2.5 `rename` Function

```
int RAM_rename(const char *old_name, const char *new_name);
```

The function `rename` changes the name of a file, but does not affect the contents or any open file descriptors. This is a straightforward operation; the only tricky thing is that if a file is renamed to a name for which a file already exists, the file with the new name must be unlinked before performing the `rename`.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2002, Texas Instruments Incorporated