

Writing a Flash driver

The QNX Neutrino OS is a POSIX-compliant real-time operating system designed on a true microkernel philosophy. Non-core services that most OS designs would place in the kernel, such as file systems, are implemented as MMU-protected applications. The resulting small size, customisability, and reliability have made QNX Neutrino a common choice for vendors of systems requiring flash-based file systems. Because standard interfaces do not exist for implementing a flash-based file system, developers must write their own drivers. While QNX provides toolkits that shield the developer from many of the implementation details, customisation is still required. This paper describes some of the issues and steps required to customise a QNX flash driver.

WHY CUSTOMISE?

Flash drivers need to be customized because there is no standard flash interface. In fact, there are two flash programming algorithms, one from Intel and one from AMD, each having its own performance-enhancing characteristics, buffers on Intel flash for instance, and unlock bypass on AMD. All other flash manufacturers follow one of these two standards. While some effort has been made at standardization via the Common Flash Interface (CFI), CFI is not an algorithm, but a table that describes a flash's characteristics, such as geometry. Also key is that multiple geometries are possible. Flash chips come in different sizes, and are divided into differing numbers of erasable blocks, typically of 64 or 128 K. On some flash the top or bottom block is subdivided to provide a boot block.

Because flash is memory, it plugs straight into the CPU, and as such there is no such thing as a standard flash bus. There are 8 and 16-bit data bus widths on flash chips, which can be interleaved to suit bus widths of 8, 16, 32 or 64 bits. Flash can also be paged to fit address space where necessary. Finally, because flash is used to run code and store data, it must be partitioned. Multiple partition configurations are possible to suit the needs of a given scenario.

FLASH FILE SYSTEM OVERVIEW

The driver organises the flash as sockets and partitions. A socket is a logical flash drive and consists of a contiguous and homogeneous region of flash memory. Each driver manages one or more sockets, and each socket can be divided into a number of partitions, which can be raw, or file-system based. A raw partition does not contain a flash file system (FFS) and can be read-only, read/write or read/write/erase. Common uses include image file systems and storage of application-specific data. A file system partition contains an FFS but can also be accessed as a raw partition. The three major types of FFS supported are read-only, read/write, and read/write/reclaim. A read/write FFS does not reclaim deleted space and will eventually fill up. On the other hand, a read/write/reclaim FFS will search for deleted space to reclaim when it runs out of free space.

Since the flash under QNX is presented to the system as a POSIX-compliant file system, the full POSIX

semantics are supported with the exceptions of hard links and ATIME. Mount points for flash sockets and raw partitions are under the "/dev" directory, while file system partitions can be mounted anywhere, as with familiar disk-based partitions.

The QNX FFS supports several features of interest to flash driver implementers. For instance, the FFS supports on-the-fly decompression, which can provide performance enhancements as well as cost cutting. It also supports wear-levelling, and decent end of life by rendering the flash read-only for longer usability. Background reclaiming and fault tolerance provide real-time performance and high reliability.

KNOW YOUR BOARD

Before coding the flash driver, certain information must be gathered, such as the interleave, bus width, geometry, and specific features such as unlock bypass. However, embedded boards can be difficult to document for a number of reasons. Sometimes the specifications are available only through NDA, sometimes the specifications are not up to date or include errors, sometimes specifications don't even exist and one must read schematics. As a general rule however, the best place to start is the manufacturer's web site.

CODING THE DRIVER

A useful starting point when coding the driver is to make a copy of the sample code supplied with the toolkit that most closely matches the flash you are working with, and making the required modifications. The developer needs to write several functions and properly select the appropriate MTD files to be used with the type of flash they are developing for.

A *main()* function must be written containing two arrays, *flash[]* and *socket[]*, which declare the socket and flash-related services (MTD) respectively. The *flash[]* array will have one entry per type of flash to be supported. The *socket[]* array will have one entry per socket. The *main()* function also initialises and starts the io-flash libraries. The socket services declared in *main()* are supplied by the developer. These include:

- The *open()* function, which initialises the socket services environment, applies command line options, syspage options or default options, and allocates a flash memory region. If you are using bank-switched flash, this function initialises a window on

the socket rather than the entire socket.

- The *page()* function, which is called to access a window at a given address offset from the start of the device and of size *socket->window_size* in the context of bank-switched flash. The *page()* function returns a memory pointer for a referenced offset and truncates the buffer size. It does not do much for linearly mapped flash, but it must be provided in either case.
- The *status()* function, which returns information such as that the media has been removed from the socket. For onboard flash, *status()* should simply return EOK.
- The *close()* function, which frees memory allocated by *open()*.

The flash services functions that make up the MTD are supplied by QNX. It is up to the developer to select and declare the appropriate object files for their flash. The following flash services are provided by the MTD:

- *ident()*
- *reset()*
- *write()*
- *erase()*
- *suspend()*
- *resume()*
- *sync()*

Making use of MTD services involves selecting the correct flash library file and mapping the correct functions into the *flash[]* array. The appropriate entry must be made for each type of flash that will be used. When selecting the library, remember that multiple libraries are available for each vendor. The file naming convention for MTD files codifies important information. For example:

```
libmtd-flash-amd-w8-i2a
```

is an MTD library for AMD flash with a bus width of 8 bits and an interleave of 2.

Sample main() function:

```
int main(int argc, char **argv)
{
    int error;
    static f3s_service_t service[] = {
        { sizeof(f3s_service_t), f3s_800fads_open,
          f3s_800fads_page, f3s_800fads_status,
          f3s_800fads_close },
        { 0, 0, 0, 0, 0 /* mandatory last entry */ }
    };
    static f3s_flash_t flash[] = {
        { sizeof(f3s_flash_t), f3s_a29f040_ident, f3s_a29f040_reset,
          NULL, f3s_a29f040_write, f3s_a29f040_erase,
          f3s_a29f040_suspend, f3s_a29f040_resume,
          f3s_a29f040_sync },
        { 0, 0, 0, 0, 0, 0, 0 /* mandatory last entry */ }
    };
    f3s_init(argc, argv); // parse arguments, initialize
    error=f3s_start(service, flash);
    return error;
}
```

COMPILING THE DRIVER

The QNX toolchain includes several tools. *mkifs* is used to create a bootable or non-bootable image file from a given build file. *mkefs* is used to create a FFS image file from a given build file. The build file typically contains a set of attributes along with the name of a directory to be used as the contents of the flash file system.

```
[block_size=256k spare_blocks=1 mount=/ min_size=500k
max_size=6M]
/ffs_dir
```

block_size depends on which memory devices are being used and how they're arranged, for instance, two interleaved 64k*8-bit devices configured for a 16-bit interface have a block size of 128k. The default is 64k. **spare_blocks** is used to reclaim space taken up by deleted files. The default is 1. **mount** is the mountpoint for the embedded FFS. If you don't specify a mountpoint, the default mountpoint is **fsXpY**, where X is the number of the socket, and Y is the number of the partition on that socket. **min_size** is the minimum size of the embedded file system. If the size is less than this after all specified files have been added, the FFS is padded to this minimum size. Omitting **min_size** results in no padding. **max_size** specifies the maximum size of the embedded FFS. If **max_size** is exceeded **mkefs** will issue a warning. The default is 4G. **mkimage** builds a socket image file from a number of individual files by combining multiple image files into one. It is needed for flash devices that will accept only one file.

```
mkimage bootimage.image efs.image -b 256k -o singleimage
```

The order of files specified on the command line is not important; **mkimage** will automatically put them in the order:

1. Bootable image file
2. Embedded FFS image files
3. Any other files

You could use the common *cat* utility, but you would have to manually pad the bootable image so that the FFS image starts on a block boundary. Using **mkimage**, the **-b blocksize** option will do this for you.

DEBUGGING THE DRIVER

After compiling, block special devices should appear in the */dev* directory as */dev/fs0* and */dev/fs0p0*. If not, try using the verbose option **-v**. You should see a line starting with 'socket' and the socket name as set in the open service function. You should also see a line starting with 'array' and the flash MTD name with information on the size and number of units. If you don't see a 'socket' line, set a breakpoint in the *open()* function and step through, looking for errors. If you see an array line with SRAM, make sure the mapped memory is actually flash. If you see an array line with ROM, compile the MTD libraries in debug mode, put a breakpoint in the *ident* function, and step. If you get an array line with CFI but an incorrect size, compile the flash/lib library in debug mode, then put a breakpoint in

`f3s_flash_probe` and step through the function.

ISSUES TO BE AWARE OF

File compression can be an effective way to increase storage utility and reduce costs. However, there are some issues to be aware of when implementing compression. A compressed file will have two sizes:

- **virtual size** - This is, for the end user, the real size of the decompressed data.
- **media size** - The size that the file actually occupies on the media.

The QNX flash file system offers a second namespace that replicates the regular flash file's namespace, but provides media sizes when the files are *stat()*'ed rather than the virtual or effective size. This namespace is accessible by default through the `.cmp` mountpoint directory under the regular flash partition mountpoint. Using this namespace, files are never decompressed and read operations will yield raw compressed data. While reading and getting the size of files under `.cmp` is simple, writing to those files is more complicated.

1. When you write to a file created under the `.cmp` mountpoint, the data must be compressed first. The FFS will never transparently compress data.
2. Only appends are permitted when writing to a file created from the `.cmp` mountpoint. The FFS will reject random writes to compressed files.
3. You can write **uncompressed** data to a **compressed** file, but only from the regular mountpoint. The append-only rule applies for this file as well. However, writing uncompressed data to a compressed file can be wasteful because the uncompressed data is still encapsulated into compressed

headers, adding a layer of unneeded code. At system design time, files that are meant to be writable during the product life should not be compressed, and compressed files should remain read-only.

4. The compression algorithms need a minimum data set to be effective, so the result should be good enough to justify the header abstraction overhead.
5. Although it's possible to write uncompressed data without the header overhead to a compressed file provided if done from the `.cmp` namespace, the file will be readable, but will lose the POSIX capability of rendering virtual uncompressed size and will become unseekable to positions after the first chunk of uncompressed data.

Truncation is a special case. If a compressed file is opened with `O_TRUNC` from the regular virtual namespace, the file status will become just as if it were created from this namespace, giving you full POSIX capabilities with no compression or accompanying restrictions. The opposite is also true: if a non-compressed file is opened with truncation via `.cmp`, then compression rules apply. `truncate()` functionality isn't provided with compressed files, but is supported with regular files.

Finally, while fault tolerance is a feature of the QNX flash file system, caution should be taken to ensure that the system is not rebooted while data is being written to flash. Well-designed flash will support failsafe features that prevent power-off during a write ■

By Paul Leroux, Senior Technology Analyst at QNX Software Systems.



Find all the information you need on our website

The following topics are available:

- About Dedicated Systems Magazine
- About Dedicated Systems Gazette
- Editorial content of Dedicated Systems Magazine
- Editorial schedule 2001
- Index of all articles since 1Q93
- Subscribing to Dedicated Systems Magazine?
- Contributing to Dedicated Systems Magazine?
- Call for papers
- Advertising information

<http://www.dedicated-systems.com>