Chipsmall Limited consists of a professional team with an average of over 10 year of expertise in the distribution of electronic components. Based in Hongkong, we have already established firm and mutual-benefit business relationships with customers from,Europe,America and south Asia,supplying obsolete and hard-to-find components to meet their specific needs.

With the principle of “Quality Parts,Customers Priority,Honest Operation,and Considerate Service”,our business mainly focus on the distribution of electronic components. Line cards we deal with include Microchip,ALPS,ROHM,Xilinx,Pulse,ON,Everlight and Freescale. Main products comprise IC,Modules,Potentiometer,IC Socket,Relay,Connector.Our parts cover such applications as commercial,industrial, and automotives areas.

We are looking forward to setting up business relationship with you and hope to provide you with the best service and solution. Let us make a better world for our industry!

Contact us

Tel: +86-755-8981 8866 Fax: +86-755-8427 6832
Email & Skype: info@chipsmall.com Web: www.chipsmall.com
Address: A1208, Overseas Decoration Building, #122 Zhenhua RD., Futian, Shenzhen, China

User's Manual

Micriμm

For the Way Engineers Work

Micriμm
1290 Weston Road, Suite 306
Weston, FL 33326
USA

www.micrium.com

600-uC-FS-001

Micriμm
For the Way Engineers Work

Table of Contents

Chapter

1

Introduction

Files and directories are common abstractions, which we encounter daily when sending an e-mail attachment, downloading a new application or archiving old information. Those same abstractions may be leveraged in an embedded system for similar tasks or for unique ones. A device may serve web pages, play or record media (images, video or music) or log data. The file system software which performs such actions must meet the general expectations of an embedded environment—a limited code footprint, for instance—while still delivering good performance.

1-1 µC/FS

µC/FS is a compact, reliable, high-performance file system. It offers full-featured file and directory access with flexible device and volume management including support for partitions.

Source Code: µC/FS is provided in ANSI-C source to licensees. The source code is written to an exacting coding standard that emphasizes cleanness and readability. Moreover, extensive comments pepper the code to elucidate its logic and describe global variables and functions. Where appropriate, the code directly references standards and supporting documents.

Device Drivers: Device drivers are available for most common media including SD/MMC cards, NAND flash, NOR flash. Each of these is written with a clear, layered structure so that it can easily be ported to your hardware. The device driver structure is simple—basically just initialization, read and write functions—so that µC/FS can easily be ported to a new medium.

Devices and Volumes: Multiple media can be accessed simultaneously, including multiple instances of the same type of medium (since all drivers are re-entrant). DOS partitions are supported, so more than one volume can be located on a device. In addition, the logical device driver allows a single volume to span several (typically identical) devices, such as a bank of flash chips.

FAT: All standard FAT variants and features are supported including FAT12/FAT16/FAT32 and long file names, which encompasses Unicode file names. Files can be up to 4-GB and volumes up to 8-TB (the standard maximum). An optional journaling module provides total power fail-safety to the FAT system driver.

Application Programming Interface (API): µC/FS provides two APIs for file and directory access. A proprietary API with parallel argument placement and meaningful return error codes is provided, with functions like `FSFile_Wr()`, `FSFile_Rd()` and `FSFile_PosSet()`. Alternatively, a standard POSIX-compatible API is provided, including functions like `fs_fwrite()`, `fs_fread()` and `fs_fsetpos()` that have the same arguments and return values as the POSIX functions `fwrite()`, `fread()` and `fsetpos()`.

Scalable: The memory footprint of µC/FS can be adjusted at compile-time based on the features you need and the desired level of run-time argument checking. For applications with limited RAM, features such as cache and read/write buffering can be disabled; for applications with sufficient RAM, these features can be enabled in order to gain better performance.

Portable: µC/FS was designed for resource-constrained embedded applications. Although µC/FS can work on 8- and 16-bit processors, it will work best with 32- or 64-bit CPUs.

RTOS: µC/FS does not assume the presence of a RTOS kernel. However, if you are using a RTOS, a simple port layer is required (consisting of a few semaphores), in order to prevent simultaneous access to core structures from different tasks. If you are not using a RTOS, this port layer may consist of empty functions.

1-2 TYPICAL USAGES

Applications have sundry reasons for non-volatile storage. A subset require (or benefit from) organizing data into named files within a directory hierarchy on a volume—basically, from having a file system. Perhaps the most obvious expose the structure of information to the user, like products that store images, video or music that are transferred to or from a PC. A web interface poses a similar opportunity, since the URLs of pages and images fetched by the remote browser would resolve neatly to locations on a volume.

Another typical use is data logging. A primary purpose of a device may be to collect data from its environment for later retrieval. If the information must persist across device reset events or will exceed the capacity of its RAM, some non-volatile memory is necessary. The benefit of a file system is the ability to organize that information logically, with a fitting directory structure, through a familiar API.

A file system can also store programs. In a simple embedded CPU, the program is stored at a fixed location in a non-volatile memory (usually flash). If an application must support firmware updates, a file system may be a more convenient place, since the software handles the details of storing the program. The boot-loader, of course, would need to be able to load the application, but since that requires only read-only access, no imposing program is required. The ROM boot-loaders in some CPUs can check the root directory of a SD card for a binary in addition to the more usual locations such as external NAND or NOR flash.

1-3 WHY FAT?

File Allocation Table (FAT) is a simple file system, widely supported across major OSs. While it has been supplanted as the format of hard drives in Windows PCs, removable media still use FAT because of its wide support. That is suitable for embedded systems, which would often be challenged to muster the resources for the modern file systems developed principally for large fixed disks.

µC/FS supports FAT because of the interoperability requirements of removable media, allowing that a storage medium be removed from an embedded device and connected to a PC. All variants and extensions are supported to specification.

A notorious weakness of FAT (exacerbated by early Windows system drivers) is its non-fail safe architecture. Certain operations leave the file system in an inconsistent state, albeit briefly, which may corrupt the disk or force a disk check upon unexpected power failure. µC/FS minimizes the problem by ordering modifications wisely. The problem is completely solved in an optional journaling module which logs information about pending changes so those can be resumed on start-up after a power failure.

1-4 CHAPTER CONTENTS

Figure 1-1 shows the layout and flow of the book. This diagram should be useful to understand the relationship between chapters. The first (leftmost) column lists chapters that should be read in order to understand µC/FS's structure. The chapters in the second column give greater detail about the application of µC/FS. Each of the chapters in the third column examines a storage technology and its device driver. Finally, the fourth column lists the appendices, the topmost being the µC/FS reference, configuration and porting manuals. Reference these sections regularly when designing a product using µC/FS.

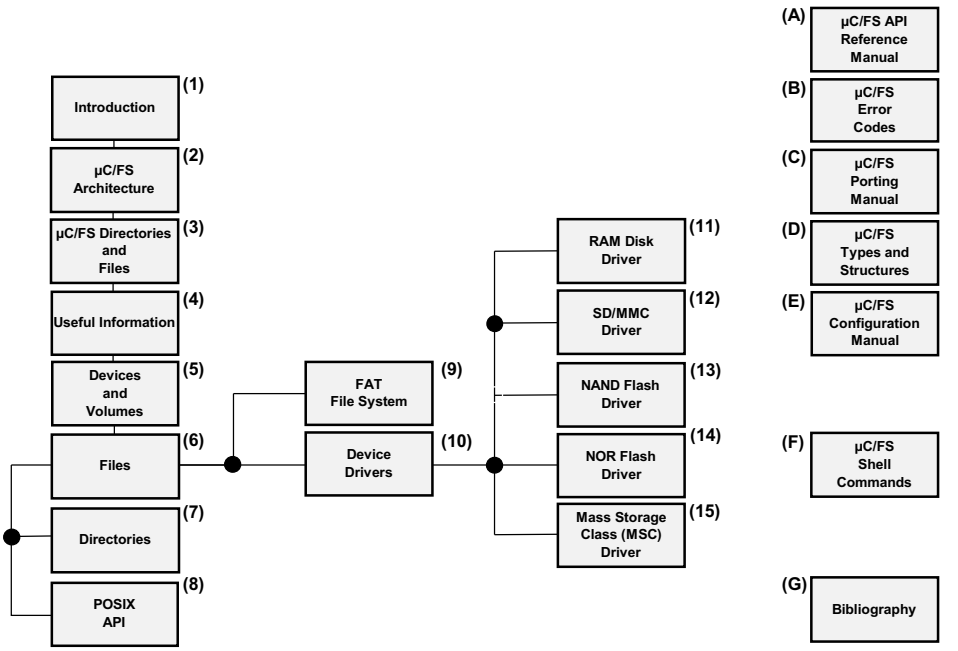

Figure 1-1 **µC/FS book layout**

Chapter 1, "Introduction". This chapter.

Chapter 2, "µC/FS Architecture". This chapter contains a simplified block diagram of the various different µC/FS modules and their relationships. The relationships are then explained.

Chapter 3, "µC/FS Directories and Files". This chapter explains the directory structure and files needed to build a µC/FS-based application. Learn about the files that are needed, where they should be placed, which module does what, and more.

Chapter 4, "Useful Information". In this chapter, you will learn the nomenclature used in µC/FS to access files and folders and the resources needed to use µC/FS in your application.

Chapter 5, "Devices and Volumes". Every file and directory accessed with µC/FS is a constituent of a volume (a collection of files and directories) on a device (a physical or logical sector-addressed entity). This chapter explains how devices and volumes are managed.

Chapter 6, "Files". µC/FS complements the POSIX API with its own file access API. This chapter explains this API.

Chapter 7, "Directories". µC/FS complements the POSIX API with its own directory access API. This chapter explains this API.

Chapter 8, "POSIX API". The best-known API for accessing and managing files and directories is specified within the POSIX standard (IEEE Std 1003.1), which is based in part in the ISO C standard (ISO/IEC 9899). This chapter explains how to use this API and examines some of its pitfalls and shortcomings.

Chapter 10, "FAT File System". This chapter details the low-level architecture of the FAT file system. Though the API of µC/FS is file system agnostic, the file system type does affect performance, reliability and security, as explained here as well.

Chapter 9, "Device Drivers". All hardware accesses are eventually performed by a device driver. This chapter describes the drivers available with µC/FS and broadly profiles supported media types in terms of cost, performance and complexity.

Chapter 11, "RAM Disk Driver". This chapter demonstrates the use of the simplest storage medium, the RAM disk.

Chapter 12, "SD/MMC Drivers". SD and MMC cards are flash-based removable storage devices commonly used in consumer electronics. For embedded CPUs, a SD/MMC card is an appealing medium because of its simple and widely-supported physical interfaces (one choice is SPI). This chapter describes the interface and function of these devices.

Chapter 13, "NAND Flash Driver". NAND flash is the first category of flash media. Write speeds are fast (compared to NOR flash), at the expense of slower read speeds and complexities such as bit-errors and page program limitations. This chapter describes the functions of these devices and the architecture of the supporting driver.

Chapter 14, "NOR Flash Driver". NOR flash is the second category of flash media. They suffer slow write speeds, balanced with blazingly-fast read speeds. Importantly, they are not plagued by the complications of NAND flash, which simplifies interfacing with them. This chapter describes the function of these devices and the architecture of the supporting driver.

Chapter 15, "MSC Driver". The now-common USB drive implements the Mass Storage Class (MSC) protocol, and a CPU with a USB host interface can access these devices with appropriate software. The MSC driver, discussed in this chapter, with µC/USB-Host is just such appropriate software.

Appendix A, "µC/FS API Reference". The reference manual describes every API function. The arguments and return value of each function are given, supplemented by notes about its use and an example code listing.

Appendix B, "µC/FS Error Codes". This appendix provides a brief explanation of µC/FS error codes defined in `fs_err.h`.

Appendix C, "µC/FS Porting Manual". The portability of µC/FS relies upon ports to interface between its modules and the platform or environment. Most of the ports constitute the board support package (BSP), which is interposed between the file system suite (or driver) and hardware. The OS port adapts the software to a particularly OS kernel. The porting manual describes each port function.

Appendix D, "µC/FS Types and Structures". This appendix provides a reference to the µC/FS types and structures.

Appendix E, "µC/FS Configuration". µC/FS is configured via defines in a single configuration file, `fs_cfg.h`. The configuration manual specifies each define and the meaning of possible values.

Appendix F, "Shell Commands". A familiar method of accessing a file system, at least to engineers and computer scientists, is the command line. In an embedded system, a UART is a port over which commands can be executed easily, even for debug purposes. A set of shell commands have been developed for µC/FS that mirror the syntax of UNIX utilities, as described in this chapter.

Appendix G, "Bibliography".

Appendix H, "µC/FS Licensing Policy".

Chapter

2

µC/FS Architecture

µC/FS was written from the ground up to be modular and easy to adapt to different CPUs (Central Processing Units), RTOSs (Real-Time Operating Systems), storage media and compilers. Figure 2-1 shows a simplified block diagram of the different µC/FS modules and their relationships.

Notice that all of the µC/FS files start with '`fs_`'. This convention allows you to quickly identify which files belong to µC/FS. Also note that all functions and global variables start with '`FS`', and all macros and `#defines` start with '`FS_`'.

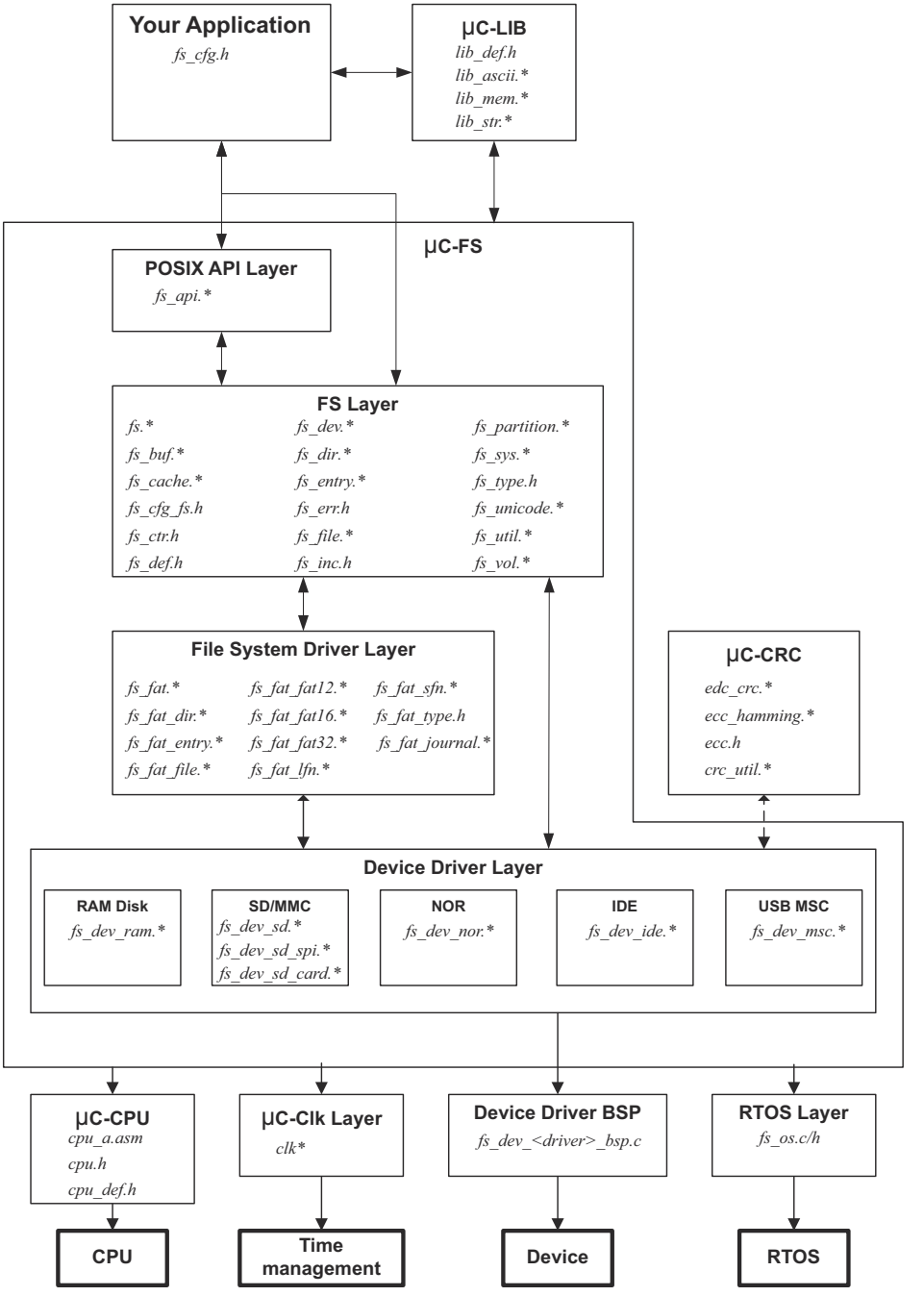

Figure 2-1 **µC/FS architecture**

2-1 ARCHITECTURE COMPONENTS

µC/FS consists of a set of modular software components. It also requires a few external components (provided with the release) be compiled into the application and a few configuration and BSP files be adapted to the application.

2-1-1 YOUR APPLICATION

Your application needs to provide configuration information to µC/FS in the form of one C header file named `fs_cfg.h`.

Some of the configuration data in `fs_cfg.h` consist of specifying whether certain features will be present. For example, LFN support, volume cache and file buffering are all enabled or disabled in this file. In all, there are about 30 `#define` to set. However, most of these can be set to their default values.

2-1-2 µC-LIB (LIBRARIES)

Because µC/FS is designed to be used in safety critical applications, all 'standard' library functions like `strcpy()`, `memset()`, etc., have been re-written to follow the same quality as the rest of the file system software.

2-1-3 POSIX API LAYER

Your application interfaces to µC/FS using the well-known `stdio.h` API (Application Programming Interface). Alternately, you can use µC/FS's own file and directory interface functions. Basically, POSIX API layer is a layer of software that converts POSIX file access calls to µC/FS file access calls.