Application Note

Using an eZ80® Flash Memory File System

AN017303-0908

Abstract

This Application Note demonstrates a method for creating a file system that can be stored within the external Flash memory space connected to Zilog's eZ80® microprocessors. This file system is used with the Zilog TCP/IP (ZTP) Software Suite that runs on the XINU operating system. The Applications Programming Interface (API) functions developed for this file system can be used to develop applications with minimum effort. An eZ80 Development Module features 1 MB of Flash Memory, which is large enough to accommodate the application data and Flash file system storage.

The file system proposed in this Application Note cannot be used for HTTP implementation. HTTP uses sequentially stored files and does not support the file system API discussed in this Application Note.

Note: *The source code file associated with this application note,* `AN0173-SC01.zip` *is available for download at* www.zilog.com.

Product Overview

The Application Note supports the eZ80 family of microprocessing devices, which includes eZ80 microprocessors and eZ80Acclaim!® Flash microcontrollers unit (MCU). Both product lines are briefly described in this section.

eZ80Acclaim! Flash MCU

eZ80Acclaim! on-chip Flash MCU are an exceptional value for designing high performance, 8-bit MCU-based systems. With speeds up to 50 MHz and an on-chip Ethernet MAC (for eZ80F91 only), you have the performance necessary to execute complex applications quickly and efficiently. Combining Flash and SRAM, eZ80Acclaim! devices provide the memory required to implement communication protocol stacks and achieve flexibility when performing in-system updates of application firmware.

The eZ80Acclaim! Flash MCU can operate in full 24-bit linear mode addressing 16 MB without a Memory Management Unit (MMU). Additionally, support for the Z80®-compatible mode allows Z80/Z180 legacy code execution within multiple 64 KB memory blocks with minimum modification. With an external bus supporting eZ80, Z80, Intel, and Motorola bus modes and a rich set of serial communications peripherals, you have several options when interfacing to external devices.

Some of the applications suitable for eZ80Acclaim! devices include vending machines, point-of-sale (POS) terminals, security systems, automation, communications, industrial control and facility monitoring, and remote control.

eZ80® General-Purpose Microprocessors

The eZ80 has revolutionized the communication industry. It executes Zilog's Z80 code four times faster at the same clock speed of traditional Z80s and can operate at frequencies up to 50 MHz. Unlike most 8-bit microprocessors, which can only address 64 KB, the eZ80 can address 16 MB without a MMU.

Designed with over 25 years of experience with the Z80, this microprocessor is best suited for embedded internet appliances, industrial control, automation, web management, modem controller, electronic games, and personal digital assistant (PDA) applications.

Discussion

File systems offer a common way to store data of any type, often over an extended period of time. They are useful for accumulating and transferring data between different types of operating environments and devices. This section discusses the considerations for designing a file system for an embedded environment.

File System Overview

Widely-known examples of file systems include:

- Disk Operating System (DOS) featuring the FAT12 and FAT16 methods of file allocation
- 32-bit WinNT file system
- Compact Disk File System (CDFS)
- Compact Flash file system.

Each of these file systems is built for a particular hardware environment and is based on a particular operating system. Typically, data is stored within a long-term storage device and loaded into short-term operating storage for processing and manipulation. Whenever data is processed, it is stored within long-term storage.

The file system discussed in this Application Note is built for the embedded hardware environment based on Zilog's eZ80 development modules, which feature external Flash and RAM memory spaces. Flash Memory is used for long-term storage purposes, and RAM is used as operating storage for data processing.

There are strict standards that govern how data is stored. Though storage media are offered in different types, such as magnetic media, Flash Memory, and so on, storage organization must suit media requirements and must be optimized for the working environment. Despite these differing media types, there are certain specific terms used when describing nearly all file systems. First, the term *file* is used to define a file system unit, which is considered to be a sequence of records with the same structure or type. Second, the term *disk* is used to define a storage unit that contains a set of files.

In the evolution of file systems, APIs are gaining acceptance relatively recently as a standard for adding modular functionality to file systems. Such an API provides a set of C language functions to perform file create, open, read, write, and close operations for file manipulation.

The file system described in this Application Note is an embedded file system for the eZ80 family of microprocessors that includes standard set of C-language API functions, such as `fcreate()`, `fopen()`, `fread()`, `fwrite()`, `fclose()`, and others. For more details of the File System API functions, see Appendix B—File System APIs on page 13.

Using the Embedded Flash-Based File System

This section contains guidelines to be followed when using the embedded Flash-based file system to develop an application. To follow the guidelines, the section first discusses the software implementation for the embedded Flash-based file system.

Software Implementation

Figure 1 on page 3 displays the functional blocks of the embedded Flash-based file system. The functional blocks comprise a File System Configuration Component, File System Manipulation Functions, and the Flash Memory Interface.

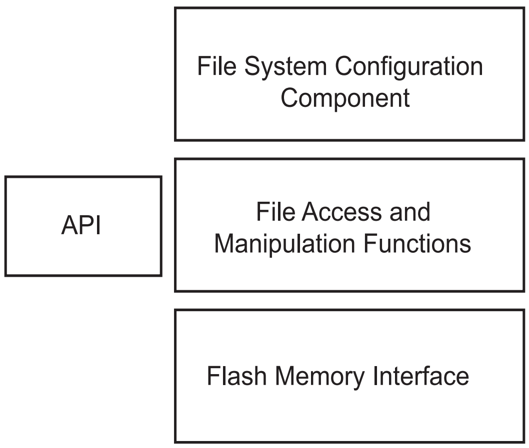

Figure 1. Functional Blocks of the Embedded Flash-Based File System

File System Configuration Component

In Figure 1, the File System Configuration Component block represents the configuration file, `fileconf.h`. It contains a set of configuration parameters that you can modify to suit the application. Table 1 lists the maximum allowable size of the file system and the capacity of its entities as defined by the configuration parameters.

Table 1. File System Configuration Parameters (fileconf.h)

Parameter	Description	Default Value
NFHANDLES	The number of opened files	10
FTABLESIZE	The number of file entries in the file system	32
NSECTORS	The number of sectors in the disk	1000
SECTORSIZE	The size of a sector in the disk	128
VHDRSIZE	Volume header size	128

The file system configuration parameters consist of several entities, few are discussed below:

File Table—The file table contains the names of the files represented in the current file system, along with their sizes and pointers to the data contained in each file. The maximum number of files in the file system is defined by the constant, `FTABLESIZE`.

Sector List—The sector list is the array of 16-bit values containing the sector addresses. The array maintains a size equal to the number of sectors on the disk, `NSECTORS`, to be able to address each of the physical sectors.

Sector Data Area—The sector data area is the space used by the file data. This space is logically divided to `NSECTORS` blocks, called sectors. Each of the sectors maintains a size of `SECTORSIZE` bytes.

The `init_file_system()` function creates or restores the file system according to the configuration parameters' values. An example of the `init_file_system()` function can be found in the `fileinit.c` file available in the `AN0173-SC01.zip` file associated with this Application Note.

File Access and Manipulation Functions

In Figure 1, the File Access and Manipulation Functions block represent the functions that are used to manage the files, after the file system is created and implemented. These functions are: `fcreate()`, `fopen()`, `fread()`, `fwrite()`, `fclose()`, `fexists()`, and `ferase()`, and together they form the file system API.

Flash Memory Interface

The Flash Memory Interface stores the volume of data (created as a result of operations in the file system and files contained in it) into Flash Memory. The Flash Memory Interface allows permanent data storage and enables you to continue working with the file system after cycling power to the storage device.

Any Flash Library[1] can be used as a Flash Memory Interface. The Flash Loader application available with the ZDS II–IDE is an example that uses a Flash Memory Interface. Yet another example is the eZ80® Remote Access Application Note (AN0134) available on the www.zilog.com.

File System Data Structure

This section discusses the `file_entry` data structure used for each structural units that compose the embedded Flash-based file system.

The `file_entry` structure used to maintain the file table entries has the following structure:

```
typedef struct
{
 char name[11]; // Name of the file
 unsigned short start;// Starting
                  //sector of the file
 unsigned int size;// Size of file in
                   //bytes
} file_entry;
```

The `start` variable contains the ordinal number of the first sector owned by the file. The value stored in the sector number cell is the number of the next sector of the file. Therefore, the sectors that pertain to the file form a chain, which is displayed in Figure 2. The final sector cell contains an end-of-file (EOF) condition.

[1] The Flash Library API Reference Manual (RM0013) describes APIs that can be used to program data into the Micron Flash device located on the target processor module.

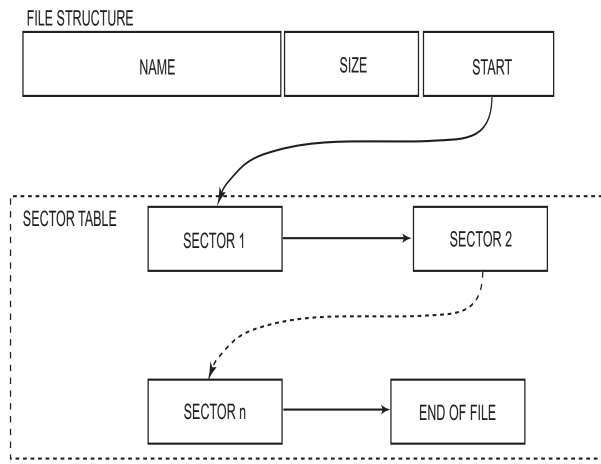

Figure 2. File Sector Chain

The size of the `file_entry` structure, represented by `sizeof(file_entry)`, is 16 bytes. The size of a sector number is 2 bytes (of type *unsigned short*). A volume header, which contains the volume parameters, must also be included in the structure.

Table 2 lists the embedded Flash-based file system representation with a 16-byte `file_entry` structure and a 2-byte sector number.

Table 2. Structural Unit Sizes

Structural Unit	Formula	Size (bytes)
File table	FTABLESIZE * sizeof(file_entry)	512
Sector table	NSECTORS * 2	2000
Sector data	NSECTORS * SECTORSIZE	128000
Total	The total size of the file system	130512
Volume header	VHDRSIZE	128

Table 2. Structural Unit Sizes (Continued)

Structural Unit	Formula	Size (bytes)
Flash block	Micron MT28F008B3 spec	131072
Note: Sizes are for a 16-byte file_entry structure and a 2-byte sector number.		

The `NSECTORS` value is set to 1000, which results in a disk size of 127.5 KB. This disk size allows accommodating the file system within one block of the Flash Memory integrated circuit (such as the Micron MT28F008B3 example in the table). The size of the disk header is added to the total volume listed in Table 2.

Working with Files

Before working with the embedded Flash-based file system, the file system must be placed into RAM. This is because all the file manipulation functions work with the file images in RAM. Two possible scenarios are described below.

1. Creating a New File System—In this case, a new file system is created using the `init_file_system()` function. The `init_file_system()` function allocates memory for the file system in RAM and initializes it for work—file tables are empty and all sectors are unused.
2. Working with a File System from an Existing Volume—In this scenario, it is assumed that a file system was previously created in RAM and stored in Flash. The file system must be *mounted*—its data must be copied into RAM and the parameters from the volume must be taken into account. The `mount()` API is used to mount a previously-created file system from Flash to RAM.

   The embedded Flash-based file system APIs are listed in Appendix B—File System APIs on page 13, and the XINU OS shell commands are listed in Appendix C—XINU OS Shell Commands on page 19.

Using an Application with the Embedded Flash-Based File System

The embedded Flash-based file system must be configured before the user application can use it. Upon setup, the user application can use the File System APIs directly or access them via an additional OS command shell interface (see Appendix C—XINU OS Shell Commands on page 19).

The File System APIs (includes the File Access and Manipulation Functions) are used to read, write, and control the data contained in the files. The File Access Library uses the Flash Memory Interface APIs to store file system data within the Flash Memory as *volume*. The File Access Library includes APIs and other functions to initialize, install, and configure the embedded Flash-based file system according to your specifications. Figure 3 on page 6 is a block diagram displaying how the user application utilizes the embedded Flash-based file system.

.

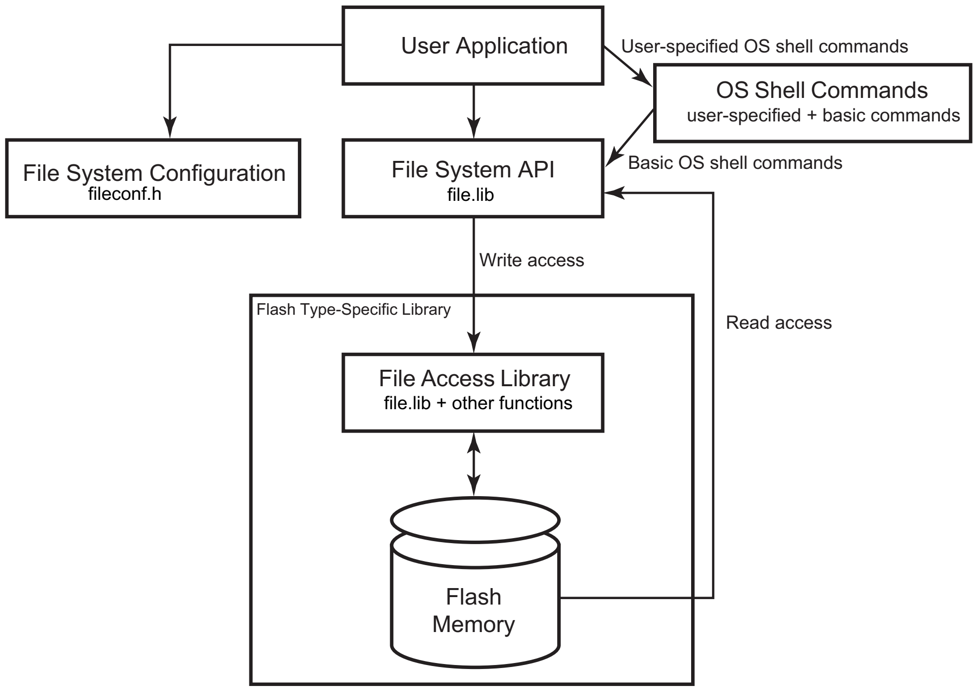

Figure 3. Structure of Software Usage

When developing an application to be used with the embedded Flash-based file system, you must include the `file.lib` file[2] in your project. The `file.lib` file contains basic XINU OS shell commands like `rename`, `chkdsk`, and `list`, in addition to the file system APIs. You can also develop XINU OS shell commands based on the APIs documented in this Application Note for specific file usage.

Notes: *1. The* `fread()` *function is independent of the Flash device that is read; it does not require any prescribed command sequence as the* `fwrite()` *function does.*

2. The `file.lib` *file is not capable of storing data into the EEPROM-based Flash Memory. To create an EEPROM-based Flash Memory, an application-specific access library must be developed.*

3. The user application can include demo-specific configuration information in the `fileconf.h` *file such that the configuration parameters are passed to the* `file.lib` *file.*

---

[2] The `file.lib` file is built using the `filelib.pro` project file available in `AN0173-SC01.zip` file, which is available for download at www.zilog.com.

Software Metrics

This section addresses the performance results related to the working of the embedded Flash-based file system.

To describe how fast the file system can operate, measurements were taken while testing this application. XINU shell commands were used to conduct the test, which was performed on an eZ80L92 Module containing an eZ80 CPU with an operating frequency of 50 MHz.

File system operating performance was measured in kilobytes per second (KBps) while executing open–write–close cycles for each operation. The benchmark was performed on a block size of 1 byte and on a block size of 128 bytes, where the latter corresponds to a sector size = 1.

The resulting performance yields the following system speeds:

- On a 1-byte block, open–read–write cycles occur at 0.3 KBps
- On a 128-byte block, open–read–write cycles occur in 21.6 KBps
- On a full disk, `chkdsk` command execution time is 8 seconds

Demonstrating the Embedded Flash-based File System

To demonstrate the embedded Flash-based file system, three projects—`filedemo_ez80.pro`, `filedemo_Acclaim.pro`, and `filemore_Acclaim.pro`—were developed. These demo projects were developed using the File System APIs.

Figure 4 displays the setup for the Flash-based File System Demo. This setup displays the connections between the PC, ZPAK II, LAN/WAN/Internet, and the eZ80L92 Development Kit.

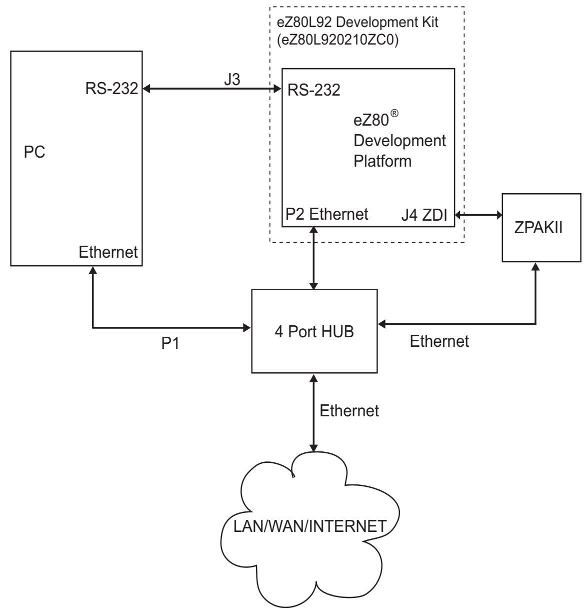

Figure 4. Setup for the Embedded Flash-based File System Demo

Equipment Required for the eZ80L92-Based Demo

The following equipment are required to execute the Flash-based File System demo on the eZ80L92 target platform:

- eZ80L92 Development Kit (eZ80L920210ZCO) that features the following:
  - eZ80 Development Platform
  - eZ80L92 Module
  - ZPAK II Debug Interface Module
  - ZDS II with C-Compiler Software and Documentation (CD-ROM)
  - Ethernet Hub
  - Power Supply (110/220 V)
  - Cables
- ZTP version 1.1
- `filedemo.pro` demo file, available in `AN0173-SC01.zip` on www.zilog.com.

The XINU OS libraries are a part of ZTP v1.1[3]. The project is set up to allow rebuilding when placed in a subdirectory of the ZTP installation directory. The settings and the steps to modify, build, and execute the demo project on the eZ80 Development Platform are provided in the following sections.

Settings

HyperTerminal Settings

Set HyperTerminal to 57.6 Kbps and 8–N–1, with no flow control

Jumper Settings

Following are the jumper settings for the eZ80 Development Platform:

- J11, J7, J2 are ON
- J3, J20, J21, J22 are OFF
- For J14, connect 2 and 3
- For J19, MEM_CEN1 is ON, and CS_EX_IN, MEM_CEN2, and MEM_CEN3 are OFF

For the eZ80L92 Module JP3 is ON

Modifying Demo-Specific Files in ZTP

To demonstrate the Flash File System described in this Application Note, the eZ80 Development Platform with the eZ80L92 Module and the ZTP stack are required along with the source code for the Flash File System. To execute this demo, the `\Filedemo` folder extracted from the `AN0173-SC01.zip` file is copied into the ZTP installation directory.

The ZTP stack is available on www.zilog.com and can be downloaded to a PC with a user registration key. ZTP can be installed in any location as specified by you; its default location is `C:\Program Files\ZiLOG`.

Note: *Before modifying the demo-specific files to ZTP, ensure that all the settings for the ZTP stack are at their default values.*

Perform the following steps to add and integrate the Demo files to the ZTP stack:

1. Download ZTP and browse to the location where ZTP is downloaded.
2. Download the `AN0178-SC01.zip` file and extract its contents to a folder on your PC. Notice there is a single `\Filedemo` folder in the extracted folder. The `\Filedemo` folder contains the following demo files[4]:

`filedemo_Acclaim.pro`	For the eZ80F91 MCU; only external Flash available for the File System.
`filedemo_ez80.pro`	For the eZ80L92 MPU; only external Flash available for the File System.
`filemore_Acclaim.pro`	For the eZ80F91 MCU; internal and external Flash available for the File System.

3. Copy the `\Filedemo` folder to the `<ZTP installed dir>\` directory.
4. Launch ZDS II and open the `filedemo.pro` file, which is available in the path: `..\ZTP\Filedemo`.
5. Open the `main0.c` file and observe the following `BootInfo` structure definition:

[3] At the time of publishing, ZTP 1.2.1 is available on Zilog.com. The XINU OS libraries of this release can also be used as they are unaltered from the earlier ZTP release.

[4] The fourth project file is `filelib.pro`, which when built generates the Flash File API library, `file.lib`, that can be used with other ZTP applications.

```
struct BootInfo Bootrecord = {
     "192.168.1.1",//Default
               //IP address//
     "192.168.1.4",//Default
                //Gateway//
     "192.168.1.5",//Default
              //Timer Server//
     "192.168.1.6",// Default
                //file Server//
     "",
     "192.168.1.7",// Default
                //Name Server//
     "",
     0xffffff00UL// Default
                //Subnet Mask//
     };
```

By default, the `Bootrecord` variable contains the network parameters and settings (in the four-octet dotted decimal format) that are specific to the LAN at Zilog®. Modify the above structure definition with appropriate IP addresses within your LAN.

6. Open the `eZ80_HW_Config.c` file and change the default MAC address (provided by ZTP) such that each eZ80 Development Platform on the LAN contains a unique MAC address. For example:

```
const BYTE f91_mac_addr
[EP_ALEN] = {0x00, 0x90, 0x23,
0x00, 0x0F, 0x91};
```

In the 6-byte MAC address described above, the first three bytes must not be modified; the last three bytes can be used to assign a unique MAC address to the eZ80 Development Platform.

7. Open the `ipw_ez80.c` file. For this application, Dynamic Host Configuration Protocol (DHCP) is disabled; therefore, ensure the following:

```
b_use_dhcp = FALSE
```

8. Save the files and close the `filedemo.pro` project.

Procedure to Build and Execute the Demo

The procedure to build and execute the `file-demo.pro` file, which demonstrates the use of the embedded Flash-based file system, is described in this section.

> **Note:** *Zilog recommends that you first build the project available in* `AN0173-SC01.zip` *file to get familiar with the file system's default configuration parameters and APIs before writing your own application and modifying the configuration parameters to suit it.*

1. Connect the ZPAK II debug interface tool and the PC to one network, and attach the ZPAK II unit to the debug port on the eZ80 Development Platform (see Figure 4 on page 7).
2. Power-up ZPAK II and the eZ80 Development Platform.
3. Connect the serial port of the eZ80 Development Platform to the serial communication port on the PC with a serial cable.
4. Launch a terminal emulation program such as Windows' HyperTerminal to examine the output of the board—this terminal serves as the XINU console.

> **Note:** *For details about obtaining an IP address for ZPAK II, refer to ZPAKII Product User Guide (PUG0015), available for download at* www.zilog.com.

5. Launch ZDS II and open the `file-demo.pro` project file located in the path: `..\ZTP\Filedemo.`
6. Navigate to **Build** → **Set Active Configuration** and ensure that the active project configuration selected in the **Select Configuration** dialog box is **eZ80L92–RAM.**

Note: *When using your own application, modify the parameters in the* `fileconf.h` *header file, if required, to set up the end-user file system.*

7. Navigate to **Build** → **Rebuild All** to rebuild the `filedemo.pro` project.
8. In the case of warnings or errors that may result, check the **Compiler** settings and ensure that the settings for the **include** files are properly indicated. Also check the **Linker** settings and ensure that the ZTP v.1.1 libraries are properly addressed. Repeat Step 6 until no compiler or linker warnings appear.
9. Navigate to **Build** → **Debug** → **Go** (alternatively, hit the **F5** key). The program downloads the project to the RAM on the eZ80L92 MCU.

Note: *When using your own application, the Flash file API library (*`file.lib` *file) must be rebuilt each time changes are made to it. This Application Note is based on the file system volume requires one entire Flash sector comprising 128 KB. If more file system storage is required, appropriate changes must be made.*

Executing the Demo

Upon executing a **Rebuild All,** the project files rebuild to form the `filedemo.lod` executable file for the RAM configuration, or the `filedemo.hex` executable file for the Flash configuration of the project.

When the resulting `.lod/.hex` file is obtained and successfully downloaded onto the eZ80L92 Module, perform the following steps to execute the demo:

1. In the HyperTerminal window, observe the XINU command prompt:

```
eth%
```

2. At the command prompt, type `test`, as follows:

```
eth% test
```

   and, observe the output:

```
Performing test...
Done.
```

3. Next, enter the following command at the command prompt:

```
eth% list
```

   The following directory listing appears:

```
Name          Size
test.log      004E19
file1.test    007CE0
file2.test    007C08
file3.test    007C98
01C399 bytes, 0019E7 free
```

The simple test procedure described above creates three files and writes several patterns to them, opening one file after another. During the execution of the procedure, a log file, `test.log`, is created and the process workflow is entered in the log file. The contents of each file can then be accessed using the `type <filename>` command.

To thoroughly test the file system, the *big test* procedure was performed. The *big test* performs continuous runs, writing pattern strings to the three files created previously, while opening and closing them one after another. When the three files occupy the entire disk space, one of them is deleted and recreated with 0 size, and the procedure continues. After a specified number of runs, this procedure ends.

Note: *To make the embedded Flash-based file system available for use as a module with other ZTP-based* *projects, another project file,* `filelib.pro` *is provided. This is part of the* `AN0173-SC01.zip` *located within the* `\Filedemo` *folder. This project file is used to build the* `file.lib` *file.*

Results

Upon executing the procedure, test files were generated. The `type` command was used on the test files to observe the test patterns written to the files using the file system API.

The `chkdsk` and `list` commands were used to view the results of the test. Erase operations did not fail and therefore no sectors were lost; all Write operations were successful and therefore no cross-linked files were created.

Summary

The absence of a file system is a limitation of the XINU OS available with eZ80 devices. With this Application Note, you can implement a file system, organize data in the form of files, and store the entire file system within the 1 MB Flash Memory space on eZ80 modules and the eZ80 Development Platform.

The eZ80 file system described in this Application Note can be used in a variety of ways. It can be used to store permanent information. Further, the file system APIs and commands can be used to create advanced applications, such as FTP applications or additional XINU command shell extensions.

Reference

The documents associated with eZ80®, eZ80Acclaim!®, eZ80F91, eZ80F92, ZDS II, and ZPAK II available on www.zilog.com are provided below:

- eZ80L92 External Flash Loader Product User Guide (PUG0013)
- Flash Library APIs for eZ80Acclaim!® MCUs (RM0013)
- eZ80® Remote Access Application Note (AN0134)
- eZ80® CPU User Manual (UM0077)
- eZ80L92 MCU Product Specification (PS0130)
- eZ80L92 Development Kit User Manual (UM0129)
- eZ80F91 Module Product Specification (PS0193)
- eZ80F91 Development Kit User Manual (UM0142)
- Zilog Developer Studio II–eZ80Acclaim!® User Manual (UM0144)
- ZPAK II Debug Interface Tool Product User Guide (PUG0015)

Appendix A—Glossary

Table 3 lists definitions for terms and abbreviations relevant to the Embedded Flash-based File System Application Note.

Table 3. Glossary

Term/Abbreviation	Definition
API	Application Programming Interface
CDFS	Compact Disk File System
CPU	Central Processing Unit
FAT	File Allocation Table
File	A sequence of components of one type – binary data or text records
FTP	File Transfer Protocol
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
MCU	MicroController Unit
MMU	Memory Management Unit
OS	Operating system
RTOS	Real Time Operating System
Sector	In a file system – the logical unit containing a portion of file data
TCP	Transaction Control Protocol
UDP	User Datagram Protocol
XINU	RTOS provided along with ZTP
ZDS	Zilog Developer Studio, an integrated development environment
ZTP	Zilog TCP/IP Protocol software suite

Appendix B—File System APIs

Table 4 lists Flash File System APIs for quick reference. Details are provided in this section.

Table 4. File Manipulation Routines

init_file_system( )	Creates a new file system or restores an existing file system.
fcreate ( )	Creates a file.
fexists( )	Checks if the file exists.
ferase( )	Deletes a file.
fopen( )	Opens a file.
fwrite( )	Writes data to a file.
fread( )	Reads data from a file.
fseek( )	Sets a file position indicator.
fclose( )	Closes a file.

init_file_system( )

Description

The `init_file_system()` function creates a new file system or restores an existing file system according to the configuration parameter values set in the `fileconf.h` file.

Argument(s)

None.

Return Value(s)

None.

Example

```
void init_file_system();
```

fcreate ( )

Description

This function creates a new file in the current file system.

Argument(s)

name	The name of the file

Return Value(s)

OK	On Success
SYSERR	On Failure (error)

Example

```
int fcreate(char* name);
```

fexists( )

Description

This function looks up the filename in the file table. It returns the table index for that filename, if the filename exists in the table.

Argument(s)

name	The name of the file

Return Value(s)

The index of the file entry in the file table	On Success (file name found)
SYSERR (–1)	On Failure (no such file name found)

Example

```
int fexists(char* name);
```

ferase( )

Description

This function deletes an existing file from the current file system.

Argument(s)

name	The name of the file

Return Value(s)

OK	On Success
SYSERR	On Failure (error)

Example

```
int ferase(char* name);
```

fopen( )

Description

This function opens an existing file for processing.

Argument(s)

name	The name of the file
mode	The mode of opening the file. The two modes are: FM_APPEND Open for write and append data at the end of the file FM_RDWR Read or write at the beginning of the file The FM_RDWR mode is the default

Return Value(s)

File handle pointer	On Success
NULL	On Failure (error)

Example

```
FILE* fopen(char* name, char mode);
```

fwrite( )

Description

This function writes data to an opened file.

Argument(s)

ptr	Pointer to the data records to be written to the file
size	Size of one data record in an array to be written to the file
n	The number of records of specified size to be written to the file
stream	The handle of the open file

Return Value(s)

OK	On Success
SYSERR	On Failure (error)

Example

```
int fwrite(const void *ptr, size_t size, int n, FILE *stream);
```

fread( )

Description

This function reads data from an opened file sector by sector, and copies the data to a memory buffer pointed to by the `ptr` parameter.

Argument(s)

ptr	Pointer to the memory destination where file data records are read into
size	Size of one data record in an array to be read from the file
n	The number of records of specified size to be read from the file
stream	The handle of the open file

Return Value(s)

OK	On Success
SYSERR	On Failure (error)

Example

```
int fread(const void *ptr, size_t size, int n, FILE *stream);
```

fseek( )

Description

The `fseek()` function sets the file position indicator for the stream pointed to by the *stream* parameter. The new position, measured in bytes, is obtained by adding offset bytes to the position specified by the *origin* parameter. If *origin* is set to `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, the offset is relative to the start of the file, the current position indicator, or the end-of-file, respectively. A successful call to the `fseek()` function clears the end-of-file indicator for the *stream* parameter.

Argument(s)

stream	The handle of the open file
offset	The offset to where the file pointer must be moved
origin	The origin from which the offset is counted

Return Value(s)

OK	On Success
SYSERR	On Failure (error)

Example

```
int fseek(FILE * stream, long offset, int origin);
```

fclose( )

Description

This function closes a previously-opened file.

Argument(s)

f	The handle of the open file

Return Value(s)

OK	On Success
SYSERR	On Failure (error)

Example

```
int fclose(FILE* f);
```

Appendix C—XINU OS Shell Commands

The ZTP suite runs on the XINU operating system. For easy file system evaluation and use, a set of operating system shell commands was developed. These commands allow performing certain user-level operations on existing files within the file system.

The XINU OS contains a shell module that must be initialized at system startup.The initialization must include the shell command extensions as presented in the code below.

```
struct cmdent file_cmds[] =
{
 { "mount", TRUE, (void*)sh_mount, NULL },
 { "store", TRUE, (void*)sh_store, NULL },
 { "list", TRUE, (void*)sh_list, NULL },
 { "type", TRUE, (void*)sh_type, NULL },
 { "copy", TRUE, (void*)sh_copy, NULL },
 { "rename",TRUE, (void*)sh_ren, NULL },
 { "chkdsk",TRUE, (void*)sh_chkdsk, NULL },
};

. . .

 init_file_system();

 // add shell extensions
 shell_add_commands(file_cmds, 7);

 // start the shell on serial interface
 open(SERIAL0, 0,0);
 if ((fd=open(TTY, (char *)SERIAL0,0)) == SYSERR)
 {
 kprintf("Can't open tty for SERIAL0\n");
 return SYSERR;
 }
 shell_init(fd);
```

Table 5 lists XINU file system extension commands for quick reference.

Table 5. XINU File System Extension Commands

mount	Install an existing Flash disk.
store	Save the contents of a file system.
list	List the contents of a file system.
type <filename>	Print file contents to a terminal.
chkdsk	Check the file system.
copy <srcfile> <dstfile>	Create a copy of a file.
rename <srcfile> <dstfile>	Rename a file.

mount

Description

The `mount` command installs an existing Flash disk; it copies its data to RAM and makes it available for the file system API.

Argument(s)

None.

store

Description

The `store` command permanently saves the contents of a complete file system from RAM into Flash Memory.

Argument(s)

None.

list

Description

The `list` command lists the contents of the file system on a terminal.

Argument(s)

None.

type <filename>

Description

The `type` command displays the contents of a selected file on the terminal.

Argument(s)

`filename`	The name of the file to be displayed

chkdsk

Description

The `chkdsk` command checks the file system for integrity. It reports the number of free sectors, lost sectors, and cross-linked sectors in the file system.

Argument(s)

None.

copy <srcfile> <dstfile>

Description

The `copy` command creates a copy of the file.

Argument(s)

`srcfile`	The name of the file to be copied
`dstfile`	The name of the file containing the copy

rename <srcfile> <dstfile>

Description

The `rename` command assigns a new name to an existing file.

Argument(s)

`srcfile`	The name of the file to be renamed
`dstfile`	The new name assigned to the file

Appendix D—API Usage

As an example of API usage, sample code is provided below. This code creates a file and writes some data patterns into it. The code also creates a log file into which the status text strings on the progress of the test are written.

```
FILE *hlog, *f;
char buffer[40];
int i;

    init_file_system( );

    fcreate("file1.test");
    fcreate("test.log");
    hlog = fopen("test.log", 0);

    f_log( "\nStarting file system test.\n", hlog );

    f_log( "\nOpening 1st test file.", hlog );
    f = fopen("file1.test", 0);

    strcpy(buffer, "\n<><><> test sequence #nnnnnn <><><>");
    f_log( "\nWriting 36*20 chars.", hlog );
    for( i=0; i<20; i++)
    {
        int2hex( i, buffer+23 );
        fwrite( buffer, 36, 1, f );
    }
    f_log( "\nClosing the test file.", hlog );
    fclose( f );

    f_log( "\n=== TEST FINISHED ===", hlog );

    fclose( hlog );
```

Warning: DO NOT USE IN LIFE SUPPORT

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer