The Compact-Flash Wildcard

and

CF Card Software Package User Guide

Version 1.1

October, 2002

The Compact Flash Card Module and CF Card Software Package User Guide

Table of Contents

Overview 1
- Specifications 1
- Built-in Software 1
- File Management Functions 2
- Automated File Loading and Execution 2

Hardware 2
- Overview 2
- Flash Card 2
- Connecting To the Wildcard Carrier Board 2
- Selecting the Module Address 3
- Installing the CF Card 3

CF Card Software Package User Guide and Glossary 4
Introduction 4
How To Install the CF Software 4
- Using the Driver Code with C 5
- Using the Driver Code with Forth 6

CF Card Software 7
Categorized List of Functions and Constants 8
- ATA Failure Codes 8
- ATA Primitives 8
- CF Card Information 8
- Directory 8
- File Access and Position Modes 8
- File I/O 8
- File Processing 9
- File System Error Handling 9
- Initialization 9

Overview of Glossary Notation 10
Using the File System Functions 11
- Commonly Used Terms 11
- String Parameters and the Use of THIS_PAGE 11
- Access Privileges 11
- Root Directory Only and File Name Restrictions 11
- File Position Indicator 12
- Error Handling 12
- Real-Time Clock on the Host Computer Enables Time/Date Marking of Files 12
- Initialization 12
- Automatic Initialization and File Processing 13
- Upgrade note for former Memory Interface Board (MIB) Users 14

How To Create and Use an AUTOEXEC.QED File 14
- Compile a Program into Flash Memory 14
- Create a Set of Image Files 15
- Transfer the Image File Contents into Memory 16
- Restrictions on Software Upgrades 17
- Recovering from Buggy AUTOSTART.QED Files 17

Compact Flash Card Software Package Main Glossary 18
Upgrade Notice for Prior Users of the Memory Interface Board 45
Sample FILEDEMO.C File 46
Sample FILEDEMO.4th File 49
CF Module Hardware Schematic 53

The Compact Flash Wildcard and CF Card Software Package User Guide

Overview

The Compact Flash Wildcard expands the memory capabilities of the QED board by providing a plug-in interface to large-capacity removable flash data storage. Flash is solid-state nonvolatile read/write memory that maintains its contents even in the absence of applied power. The Compact Flash Wildcard is ideal for applications that require large amounts of memory, the convenience of removable storage, and file-based data exchange with a PC.

This Wildcard allows you to plug in widely available Compact Flash memory cards that measure only 1.5" by 1.7" and hold many megabytes of nonvolatile data. We recommend using SanDisk Compact Flash cards which are available from Mosaic Industries and from computer retailers.

The Compact Flash card is a physically smaller implementation of the popular PCMCIA card (also known as a PC Card). Most laptop computers have a socket that hosts PCMCIA memory cards. A low-cost adapter converts the Compact Flash card to the PCMCIA form factor for easy use with a laptop computer. These cards use the IBM-compatible ATA disk drive interface standard, and can be read and written from most laptop computers.

This tiny 2" by 2.5" board is a member of the Wildcard series that connects to the QED Board™ or PanelTouch Controller™ host via the Wildcard Carrier Board.

This document reviews the CF Module hardware jumper settings, and describes the ATA Flash Card Software Package which is provided as a pre-packaged "kernel extension" that is easily loaded onto the host QED4 Board. This software is referred to as the "CF Card Software".

For those customers interested in using CF cards with an embedded controller that features 1/4 VGA graphics display, analog touchscreen, and ethernet connectivity, please note that the EtherSmart Controller™ from Mosaic Industries also hosts an on-board CF card socket that uses similar software drivers.

Specifications

Memory Sizes Available	Compact FLASH cards of 16 or 64 Mbyte
File Format	DOS/Windows compatible FAT-12 and FAT-16
Compatability	File exchange with any Windows 95, 98, NT, 2000, or XP machine using standard PMCIA socket.
Application Interface	ANSI C file manipulation functions including create, open, close, read, write, rename, copy and delete. Directory listing, file_type and file_capture commands
File Transfer	Files can be created, read, and modified on either the Mosaic or Windows platforms.
Automated File Processing	Files may be automatically executed on startup, input/output piped from/to any file, and controller programs automatically upgraded.

Built-in Software

Built-in software running on the companion QED Board implements C file manipulation functions and supports a standard DOS- and Windows-style "FAT" (File Allocation Table) file system, allowing files to

be created on a PC and read via the CF Wildcard, or visa versa. An automated file processing capability facilitates fool-proof software upgrades and data exchanges.

File Management Functions

A comprehensive set of file management functions modeled on the ANSI C file library lets you create, open, close, read, write, rename, copy and delete files. Other software features include directory listing commands that behave like the DOS DIR command, printing and file capture commands, and a powerful redirection capability that allows any function to take its input from a specified file and send its output to a specified file. This set of functions allows files to be managed from within the QED and/or the PC environment.

Automated File Loading and Execution

Fool-proof software upgrades are facilitated by the automated file processing feature. You can specify one or more files to be automatically loaded to or from the QED Board's memory at startup. This powerful capability enables field software upgrades that are accomplished by simply inserting a pre-configured CF Card into the CF Wildcard socket.

Hardware

Overview

The CF Module comprises a Wildcard bus header, digital interface chips, module address selection jumpers, and a CF Card Socket. The CF Card Software transparently controls the hardware interface to the CF Card.

Flash Card

The CF Module supports standard ATA-compatible Compact Flash cards. These are available in sizes up to hundreds of Megabytes. Sandisk is reportedly the largest manufacturer of these cards, and the hardware and software have been fully tested using their SDCFB Compact Flash cards. We also recommend the Compact Flash PC Card Adapter which makes it easy to exchange data files from a laptop PC to and from the QED Board via the CF Module. CF cards and adapters can be purchased from Mosaic Industries or computer store retailers.

Connecting To the Wildcard Carrier Board

To connect the CF Module to the Wildcard Carrier Board (also called the "Module Carrier Board"), follow these simple steps:

1. Connect the Wildcard Carrier Board to the QED Board as outlined in the "Wildcard Carrier Board User Guide".
2. With the power off, connect the 24-pin Module Bus on the CF Module to Module Port 0 or Module Port 1 on the Wildcard Carrier Board. If you are using the Panel-Touch Controller, Module Port 0 is recommended, as this allows the CF Card to be accessed from the side of the assembly. The corner mounting holes on the module should line up with the standoffs on the Wildcard Carrier Board. The module ports are illustrated in the "Wildcard Carrier Board User Guide". Note that the CF Module headers are configured to allow direct stacking onto the Wildcard Carrier Board, even if other modules are also installed. Moreover, the latest version of the Wildcard Carrier Board is designed to directly stack onto the QED Board. Do not use ribbon cables to connect to the CF Module. The CF Module is not specified for use with ribbon cables between the CF Module and the Wildcard Carrier Board, or between the QED Board and the Wildcard Carrier Board. CAUTION: The Wildcard Carrier Board does not have keyed connectors. Be sure to insert the module so that all pins are connected. The Wildcard Carrier Board and the CF Module can be permanently damaged if the connection is done incorrectly.

Selecting the Module Address

Once you have connected the CF Module to the Wildcard Carrier Board, you must set the address of the module using jumper shunts across J1 and J2.

The Module Select Jumpers, labeled J1 and J2, select a 2-bit code that sets a unique address on the module port of the Wildcard Carrier Board. Each module port on the Wildcard Carrier Board accommodates up to 4 modules. Module Port 0 on the Wildcard Carrier Board provides access to modules 0-3 while Module Port 1 provides access to modules 4-7. Two modules on the same port cannot have the same address (jumper settings). Table 1 shows the possible jumper settings and the corresponding addresses.

Module Port	Module Address	Installed Jumper Shunts
0	0	None
	1	J1
	2	J2
	3	J1 and J2
1	4	None
	5	J1
	6	J2
	7	J1 and J2

Table 1: Jumper Settings and Associated Addresses

Installing the CF Card

Install a Compact Flash Card into the socket on the CF Module. These cards are designed to fit into the sockets only one way. Damage may occur to the CF card or the socket if the card is forced into the socket in an incorrect orientation.

Once the hardware and software are properly configured, files on the CF Card can be created, read, written, and manipulated by the pre-coded high-level driver routines. You can run the file system demonstration program as described in the readme.txt file on the File System distribution diskette, or read the comments of the FILEDEMO code listing in this document.

CF Card Software Package User Guide and Glossary

Introduction

The rest of this document describes the Compact Flash (CF) Card Software Package. This package is shipped as a kernel extension that is easily loaded onto the host controller (QED Board, Panel-Touch Controller, or EtherSmart Controller). The software enables a product containing a host controller, a Wildcard Carrier Board and a CF Module to read from and write to vast amounts of flash memory on convenient removable media. The software supports the standard DOS/Windows "FAT" file system, allowing files to be created on a PC and read by the QED Board, or visa versa. An automated file processing facility facilitates software upgrades or data exchanges.

Flash is a state of the art memory technology that is programmable yet nonvolatile: it maintains its contents even in the absence of power. Widely available Compact Flash Cards can be read or written from most laptop computers with a low-cost PCMCIA form-factor adapter. They adhere to the ATA standard for disk drives, a widespread software and hardware interface. "ATA" stands for "AT Attachment", an interface developed for the IBM PC AT. ATA Flash cards can be operated in a "true IDE" mode that makes them behave as a standard IDE hard drive such as those found in every PC.

How To Install the CF Software

The CR Card device driver software is provided as a pre-coded modular runtime library, known as a “kernel extension” because it enhances the on-board kernel's capabilities. The library functions are accessible from C and Forth.

Mosaic Industries can provide you with a web site link that will enable you to create a packaged kernel extension that has drivers for all of the hardware that you have on your system. In this way the software drivers are customized to your needs, and you can generate whatever combination of drivers you need. Make sure to specify the CF Module Driver in the list of kernel extensions you want to generate, and download the resulting “packages.zip” file to your hard drive.

For convenience, a separate pre-generated kernel extension for the CF Module is available from Mosaic Industries on the Demo and Drivers media (diskette or CD). Look in the Drivers directory, in the subdirectory corresponding to your hardware (QED, PanelTouch, or EtherSmart), in the CF_Module folder.

The kernel extension is shipped as a “zipped” file named “packages.zip”. Unzipping it (using, for example, winzip or pkzip) extracts the following files:

readme.txt - Provides summary documentation about the library.
install.txt - The installation file, to be loaded to COLD-started QED Board.
library.4th - Forth name headers and utilities; prepend to Forth programs.
library.c - C callers for all functions in library; #include in C code.
library.h - C prototypes for all functions; #include in extra C files.

Library.c and library.h are only needed if you are programming in C. Library.4th is only needed if you are programming in Forth. The uses of all of these files are explained below.

We recommend that you move the relevant files to the same directory that contains your application source code.

To use the kernel extension, the runtime kernel extension code contained in the install.txt file must first be loaded into the flash memory of the QED Board. Start the QED Terminal software with the QED board connected to the serial port and turned on. If you have not yet tested your QED board and terminal

software, please refer to the documentation provided with the QED Terminal software. Once you can hit enter and see the 'ok' prompt returned in the terminal window, type
COLD
to ensure that the board is ready to accept the kernel extension install file. Use the "Send File" menu item of the terminal to download the install.txt to the QED Board or Panel-Touch Controller.
Now, type
COLD
again and the kernel has been extended! Once install.txt has been loaded, it need not be reloaded each time that you revise your source code.

Using the Driver Code with C

Move the library.c and library.h files into the same directory as your other C source code files. After loading the install.txt file as described above, use the following directive in your source code file:
`#include "library.c"`
This file contains calling primitives that implement the functions in the kernel extension package. The library.c file automatically includes the library.h header file. If you have a project with multiple source code files, you should only include library.c once, but use the directive
`#include "library.h"`
in every additional source file that references the CF Card software functions.

To load the optional demonstration program described above, use the "make" icon of the C compiler to compile the file named

FileDemo.c

that is provided on the Demos and Drivers media. Use the terminal to send the resulting FileDemo.txt file to the QED Board, and type `main` to run the program. See the demo source code listing below for more details.

Note that all of the functions in the kernel extension are of the _forth type. While they are fully callable from C, there are two important restrictions. First, _forth functions may not be called as part of a parameter list of another _forth function. Second, _forth functions may not be called from within an interrupt service routine unless the instructions found in the file named

\fabius\qedcode\forthirq.c

are followed. Also, in most cases file manipulation functions should not be called from within interrupt service routines. Interrupt service routines should be kept short so that the processor can efficiently respond to events, and file operations take a relatively long time. More importantly, in multitasking environments any function that calls the Get or Release functions from within an interrupt service routine can halt the multitasker. All of the file I/O functions that access the CF Card Get and Release the CF resource variable, so it is not wise to call them from within an interrupt service routine.

NOTE: If your compiler was purchased before June 2002, you must update the files named qlink.bat and qmlink.bat in your /fabius/bin directory on your installation before using the kernel extension. You can download a zip file of new versions at

http://www.mosaic-industries.com/Download/new_qlink.zip

The two new files should be placed in c:\Fabius\bin. This upgrade only has to be done once for a given installation of the C compiler.

Using the Driver Code with Forth

After loading the install.txt file and typing `COLD`, use the terminal to send the "library.4th" file to the QED Board. Library.4th sets up a reasonable memory map and then defines the constants, structures, and name headers used by the CF Module kernel extension. Library.4th leaves the memory map in the download map.

After library.4th has been loaded, the board is ready to receive your high level source code files. At this point you may want to load the FileDemo.4th file that is provided on the demo media.

Be sure that your software doesn't initialize the memory management variables DP, VP, or NP, as this could cause memory conflicts. If you wish to change the memory map, edit the memory map commands at the top of the library.4th file itself. The definitions in library.4th share memory with your Forth code, and are therefore vulnerable to corruption due to a crash while testing. If you have problems after reloading your code, try typing `COLD`, and reload everything starting with library.4th. It is very unlikely that the kernel extension runtime code itself (install.txt) can become corrupted since it is stored in flash on a page that is not typically accessed by code downloads.

We recommend that your source code file begin with the sequence:

```
WHICH.MAP 0=
IFTRUE 4 PAGE.TO.RAM  \ if in standard.map...
      5 PAGE.TO.RAM
      6 PAGE.TO.RAM
     DOWNLOAD.MAP
ENDIFTRUE
```

This moves all pre-loaded flash contents to RAM if the QED Board is in the standard (flash-based) memory map, and then establishes the download (RAM-based) memory map. At the end of this sequence the QED Board is in the download map, ready to receive additional code.
We recommend that your source code file end with the sequence:

```
4 PAGE.TO.FLASH
5 PAGE.TO.FLASH
6 PAGE.TO.FLASH
STANDARD.MAP
SAVE
```

This copies all loaded code from RAM to flash, and sets up the standard (flash-based) memory map with code located in pages 4, 5 and 6. The `SAVE` command means that you can often recover from a crash and continue working by typing `RESTORE` as long as flash pages 4, 5 and 6 haven't been rewritten with any bad data.

CF Card Software

The CF Software Package comprises four layers. The first two are essentially invisible to the end user, and the latter two provide high level capabilities:

- The ATA/IDE software driver layer enables drive identification and read/write operations using standard ATA commands.
- The FAT layer implements a DOS-compatible FAT (File Allocation Table) file interface to track the placement of files on a pre-formatted ATA flash card in a PC-compatible format. Both FAT-12 and the newer FAT-16 formats are supported, and the FAT format is supported by all Microsoft operating systems including Windows NT, Windows 2000 and Windows XP. The key restrictions are that files must be in the root directory, and filenames are limited to the "8+3" scheme: up to 8 characters in the filename, and up to 3 characters in the file extension.
- The file management layer includes a set of user-callable commands modeled on the ANSI C file manipulation functions that enable the programmer to create, open, close, read, write, rename, copy and delete files in the root directory. Other software features include directory listing commands that behave like the DOS DIR command, a File_Type command that prints the contents of a specified file to the active serial port, and a File_Capture function to capture text sent from a terminal into a specified file. This set of functions allows files to be managed from within the QED environment and/or the PC environment. Files can be created on one platform and read and modified on the other.
- The automated file processing and redirection layer enables the programmer to specify a file to be automatically processed by the QED-Forth interpreter/compiler at startup. The software searches the root directory of a detected CF card for a file named "AUTOEXEC.QED" and, if it is found, interprets and executes the commands in the file. Two easy-to-use functions named Page_To_File and File_To_Page can be invoked in the autoexec file to store memory contents in a file, or transfer file contents to memory. This enables software upgrades to be performed in the field by simply plugging in a CF card that contains the proper update files. Finally, a powerful Redirect function allows any function to take its input from a specified file, and/or send its output to a specified file.

Categorized List of Functions and Constants

This section lists all of the keywords (functions, constants and variables) in the ATA Flash Card Software Package. Keywords are in **bold** typeface, and functions include C-style parameter lists: the returned data type is shown before the function name, and a comma-delimited list between parentheses shows the type and a descriptive name for each input parameter.

ATA Failure Codes

FAIL_ADDR	**FAIL_ARGS**	**FAIL_BUSY**
FAIL_CMD	**FAIL_DATAREQ**	**FAIL_EXTERR**
FAIL_ID_DRIVE	**FAIL_NO_CARD**	**FAIL_READY**

ATA Primitives

int **ATA_Command**(xaddr xbuf, ulong startsector, uint numsectors, int cmd, int xfer_type, int features)
int **ATA_Fail**(void)
int **ATA_ID_Drive**(xaddr xbuffer)
int **ATA_Read**(xaddr xbuf, ulong startsector, uint numsectors)
int **ATA_Set_Features**(int config_contents, int feature_contents)
int **ATA_Write**(xaddr xbuf, ulong startsector, uint numsectors)
int **PCC_Present**(void)

CF Card Information

ulong **Card_Size**(void) Int **Hidden_Sectors**(void)
int **Numsectors_Transferred**(void) **SECTOR_SIZE**
uint **Volume_Label**(char* string_addr, uint string_page)

Directory

void **Dir**(void) void **Dir_Names**(void) void **Dir_Open**(void)
xaddr **File_Time**(int file_id)
int **Set_Filesize**(ulong needed_filesize, int file_id)

File Access and Position Modes

A_MODE	**APLUS_MODE**	**FROM_CURRENT**
FROM_END	**FROM_START**	**R_MODE**
RPLUS_MODE	**W_MODE**	**WPLUS_MODE**

File I/O

END_CAPTURE (Forth only)
int **File_Capture**(int file_id)
int **File_Close**(int file_id)
xaddr **File_Contents**(uint file_position, int file_id)
int **File_Copy**(char* prior_name_addr, uint prior_name_page, int prior_count, char* new_name_addr, uint new_name_page, int new_count)
int **File_Flush**(int file_id)
int **File_Getc**(int file_id)
int **File_Gets**(char* dest_addr, uint dest_page, uint bufsize, int file_id)
int **File_Open**(char* name_addr, uint name_page, int name_count, int access_mode)
int **File_Putc**(char c, int file_id)

int **File_Put_CRLF**(int file_id)
int **File_Puts**(char* source_addr, uint source_page, uint maxchars, int file_id)
long **File_Read**(void* dest_addr, uint dest_page, ulong numbytes, int file_id)
int **File_Remove**(char* name_addr, uint name_page, int name_count)
int **File_Rename**(char* prior_name_addr, uint prior_name_page, int prior_count,
char* new_name_addr, uint new_name_page, int new_count)
int **File_Rewind**(int file_id)
int **File_Seek**(int file_id, long offset, int mode)
int **File_Set_Pos**(int file_id, long offset)
ulong **File_Size**(int file_id)
long **File_Tell_Pos**(int file_id)
long **File_To_Memory**(void* dest_addr, uint dest_page, ulong numbytes, int file_id)
void **File_Type**(char* name_addr, uint name_page, int name_count)
int **File_Ungetc**(char c, int file_id)
long **File_Write**(void* source_addr, uint source_page, ulong numbytes, int file_id)
ulong **Max_File_Size**(void)

File Processing

void **Do_Autoexec**(void)
int **File_Ask_Key**(void)
void **File_Interpreter**(void)
void **File_To_Page**(int page)
void **Page_To_File**(int page)
void **File_Abort_Action**(void)
void **File_Emit**(char c)
uchar **File_Key**(void)
NO_ECHO_FILEID
int **Process_Autoexec**(int autoexec_echo)
void **Process_File**(int input_file_id, int output_file_id)
void **Put_Default_Serial**(void)
void **Redirect**(int input_file_id, int output_file_id, void(*fn)(), uint fn_page)

File System Error Handling

void **Clear_File_Error**(int file_id)
ERR_ATA_READ
ERR_ATA_WRITE
ERR_BAD_OR_UNOPEN_FILEID
ERR_CANNOT_ADD_CLUSTER
ERR_CANNOT_TRUNCATE_FILE
ERR_DISK_IS_FULL
ERR_EOF
ERR_FAIL_ID_DRIVE
ERR_FILE_DOES_NOT_EXIST
ERR_INVALID_SECTOR_NUMBER
ERR_NEGATIVE_FILE_POSN
ERR_NOT_DOSFAT_DRIVE
ERR_READ_ACCESS_VIOLATION
ERR_ROOT_DIR_FULL
ERR_TOO_MANY_FILES_OPEN
ERR_WRITE_ACCESS_VIOLATION
int **File_EOF**(int file_id)
int **File_Error**(int file_id)
uint **Report_File_Errors**(int file_id, char* string_addr, uint string_page, uint maxbytes)
uint **Report_File_Open_Errors**(char* string_addr, uint string_page, uint maxbytes)

Initialization

xaddr **FI**(void)
int **Init_File_Heap**(uint maxopen_files, uint file_bufsize, xaddr xheap_start, xaddr xheap_end)
int **Init_File_IO**(uint maxopen_files, uint file_bufsize, xaddr xheap_start, xaddr xheap_end,
xaddr xfat_info)
int **Init_File_System**(void)
void **Link_File_IO**(void)
int **Read_CF_Module**(void)
Set_CF_Module(int module_num)

Overview of Glossary Notation

The main glossary entries presented in this document are listed in case-insensitive alphabetical order (the underscore character comes at the end of the alphabet). The keyword name of each entry is in **bold** typeface. Each function is listed with both a C-style declaration and a Forth-style stack comment declaration as described below. The "C:" and "4th:" tags at the start of the glossary entry distinguish the two declaration styles.

The Forth language is case-insensitive, so Forth programmers are free to use capital or lower case letters when typing keyword names in their program. Because C is case sensitive, C programmers must type the keywords exactly as shown in the glossary. The case conventions are as follows:

- Function names begin with a capital letter, and every letter after an underscore is also capitalized. Other letters are lower case, except for capitalized acronyms such as "ATA".
- Constant names and C macros use capital letters.
- Variable names use lower case letters.

It is best to discuss glossary notation in terms of an example. Here is a partial glossary entry:

C: int **File_Open**(char* name_addr, uint name_page, int name_count, int access_mode)
4th: **File_Open** (name_xaddr\name_count\access_mode -- file_id)

Opens in the root directory the file having the name and extension contained in the name_addr string, and assigns access privileges as specified by the access_mode. Returns a non-negative file_id if the open was successful. ... Valid access modes are R_MODE, RPLUS_MODE, W_MODE, WPLUS_MODE, A_MODE, and APLUS_MODE. ... The filename is passed to the function as a character string whose first byte is at the specified name_addr on the specified page. Forth programmers should pass a string that has been "unpacked" using the COUNT function. C programmers must pass name_count = -1 to inform the function that this is a null-terminated string. C programmers will typically pass the THIS_PAGE macro (defined in \include\mosaic\types.h) to specify the name_page parameter. ... "Long file names" (that is, more than 8 characters in the name) are not supported. ...

The C declaration is first: the returned data type is shown before the function name, and the comma-delimited input parameters are listed between parentheses showing the type and a descriptive name for each. In this example, the returned file_id (file identifier) parameter is an integer (int), and the input parameters are a pointer to a character string (char*) and the memory page upon which the string resides, the integer count of the string, and an integer access mode.

The Forth declaration contains a "stack picture" between parentheses; this is recognized as a comment in a Forth program. The items to the left of the double-dash (--) are input parameters, and the item to the right of the double-dash is the output parameter. Forth is stack-based, and the first item shown is lowest on the stack. The backslash (\ ) character is read as "under" to indicate the relative positions of the input parameters on the stack. In the Forth declaration the parameter names and their data types are combined. All unspecified parameters are 16-bit integers. Forth promotes all characters to integer type. An "xaddr" is an "extended address", meaning a 32-bit address that contains page information. The presence of "xaddr" or "xbuf" in a parameter name means that it is a 32-bit address parameter. Forth refers to 32-bit integers as "double" numbers indicated by "d." names. For example, Card_Size returns parameter "d.size".

The presence of both C and Forth declarations is helpful: the C syntax explicitly shows the types of the parameters, and the Forth declaration provides a descriptive name of the output parameter.

Using the File System Functions

This section discusses some important concepts that will help you use the file system software. The next section provides additional information about how to perform software upgrades using the automated file processing capability. The programming examples at the end of this document also provide lots of information that is useful when crafting your application.

Commonly Used Terms

The sample entry for File_Open brings up some commonly used terms which we now describe. The "file_id" parameter returned by File_Open is a file identifier that is passed to many of the file manipulation functions to specify which open file should be operated upon. Valid file_id's are small non-negative integers assigned by the File_Open function. The first file to be opened is assigned file_id = 0, the next is assigned 1, etc. The maximum number of files that may be open at one time is set by Init_File_IO; the default set by Init_File_System is 5. A file's file_id is recycled (made available) when the file is closed. The file_id returned by File_Open should be saved by the user's program to enable accessing and closing of the file.

String Parameters and the Use of THIS_PAGE

The character string containing the name is handled slightly differently in each of the two programming languages. In C, the starting address and page of the string are passed in as two separate parameters. Typically, C programmers will pass the address of a character string or buffer as the name_addr, and will pass the macro THIS_PAGE as the name_page. THIS_PAGE resolves at runtime to the current page, and works well for strings that are compiled in the .strings area in ROM (this is the default place that strings are located by the C compiler and linker), or for string buffers in common RAM. To signal to the function that this is a null-terminated string, C programmers must pass -1 as the name_count.

In Forth, the starting extended address (that is, the 32-bit address of the first character of the string) is passed as name_xaddr, and the count is passed as name_count. Typically, these parameters are obtained by defining the string using double quotes, then using COUNT to unpack the counted string; see the definition of COUNT in the main QED-Forth glossary.

Access Privileges

All files are opened with a specified access_mode parameter. The access mode determines whether the file has read-only privileges, write-only privileges, is append-only, or some combination of these. Consult the complete glossary entry for File_Open for a description of the meaning of these modes. The access modes are modeled on ANSI-C: C's "r" is our R_MODE (read-only mode), C's "a+" is our APLUS_MODE (append mode with read privileges), etc.

Root Directory Only and File Name Restrictions

This file system software does not support directories. That is, only files in the "root directory" of the ATA flash card are recognized. This limits the number of files on the flash card; for example, the root directory of a 10 Megabyte card can contain a maximum of 512 files. Only DOS- and Windows 3.xx style names are supported: filenames must have 8 or fewer characters in the main part of the name, with an optional period and filename extension of up to 3 characters. All characters in the filenames are converted to upper case. "Long filenames" are not supported. See the glossary entry for Max_File_Size for a description of the filesize limitation. Please keep these restrictions in mind if you are using a PC to create files that are to be accessed via the CF Module.

File Position Indicator

Random access within an open file can be accomplished using any of the read or write functions, including File_Getc, File_Gets, File_Putc, File_Put_CRLF, File_Puts, File_Read, File_To_Memory, and File_Write. Each of these accesses an open file (specified by a file_id) starting at the current file position. When a file is first opened, its file position equals zero. The first character that is read from the file is the one located at the file position (in this case, at the start of the file), and the file position indicator is automatically incremented with each character read. If we open a file and immediately start writing to it, the characters are written starting at the current file position (in this case, at the start of the file) unless the append mode (A_MODE or APLUS_MODE) is active in which case the file position indicator is automatically moved to the end of the file. Again, the file position indicator is automatically incremented with each character written.

A program can modify the file position indicator using the functions File_Seek, File_Set_Pos, or File_Rewind. To find out the current file position, call the function File_Tell_Pos. Read the glossary entry of File_Seek for a brief but comprehensive overview of how the file position indicator works.

Error Handling

Many of the file system functions return an error parameter. This integer equals zero if there were no errors, and equals a bit-mapped error code if some failure occurred. The error code returned by the latest file function is also bit-wise OR'd with the prior error parameter, and saved in a system variable associated with the specified file that can be accessed using the File_Error function. Each error condition is represented by a named constant that starts with "ERR_"; see their glossary entries for details. If your program needs to test for a particular type of error, perform at bit-wise AND operation between the relevant ERR_ code and the error parameter; if the result is nonzero, the specified error occurred. To clear the error flag associated with a specified open file, call Clear_Error or File_Rewind.

Often a program must test for the end of file (EOF) condition. While the procedure described above can be used (via the ERR_EOF constant), it may be easier to call the pre-coded File_EOF function which returns a true (-1) flag if an end of file has been encountered.

To print an error message summarizing the error conditions that have occurred, use Report_File_Errors. To print a message summarizing errors that occurred while trying to open a file, use Report_File_Open_Errors. See their glossary entries for details.

A set of failure codes is associated with the low-level ATA driver routines. While these are typically not used, they are available for special applications. See the glossary entry for ATA_Fail for more details.

Real-Time Clock on the Host Computer Enables Time/Date Marking of Files

If a battery-backed real-time clock is available on the host controller board, then the directory entries of all writeable files that are created or modified by the file system will be properly marked with DOS-compatible time and date information. This can make it easier to manage files. The time and date information is reported by the Dir and Dir_Open functions.

Initialization

A CF Card must be plugged into its socket to perform an initialization of the CF Software. The most commonly used initialization function is

    Init_File_System

This convenient function does not require or return any parameters. Before calling it, the function

    Set_CF_Module

must be called with a parameter equal to the module number as specified by the jumper settings and module port (see the section titled “Selecting the Module Address” and Table 1, above). For example, if both jumper shunts are installed and the CF Module is plugged into module port 1, then Set_CF_Module must be called with a parameter of 7 before calling Init_File_System.

Init_File_System sets up a default set of RAM-based data structures in the top 16 Kbytes of page 3 RAM (this implies that you must operate in the STANDARD.MAP to run this code). The function locates the fat_info structure pointed to by FI at 0x4000 to 0x417F on page 3, and declares a heap for use by the CF Software at 0x4180 to 0x7FFF on page 3. (Note that "0x" in front of a number means that the number is shown in hexadecimal base.) It specifies a maximum of max 5 open files with a 1 Kbyte user buffer per file. Up to 5 open files at a time are supported, and a 1 Kilobyte File_Contents buffer is dimensioned in the page 3 heap for each open file. This File_Contents buffer is available to the programmer; see the File_Contents glossary entry for details.

Init_File_System calls a subsidiary function named
    Init_File_IO

that accepts a set of parameters that allow the programmer to customize the memory map. If your application uses the top 16 Kbytes of page 3 for other purposes, use Init_File_System to set up the CF Software with an appropriate memory map. It lets you specify the location of the fat_info structure, the location and size of the heap that contains all of the file system data structures, the maximum number of open files, and the size of the File_Contents buffers. The designated heap can span multiple pages of RAM, so large numbers of open files and/or large buffers can be accommodated.

Automatic Initialization and File Processing

File processing is a powerful feature that lets you reconfigure, upgrade or exchange data with a QED-based instrument by inserting a CF Card and invoking the file processing function. This section provides an overview of these features; for more details consult the following section and see the glossary entries for Process_File, Process_Autoexec, and Do_Autoexec.

The most comprehensive initialization and file processing function is
    Do_Autoexec

This function does not require or return any parameters. Before calling it, the function
    Set_CF_Module

must be called with a parameter equal to the module number as specified by the jumper settings and module port (see the section titled “Selecting the Module Address” and Table 1, above. If a CF Card is installed, the Do_Autoexec function calls Init_File_System (described above) and then calls Process_Autoexec which looks for a file named AUTOEXEC.QED and executes it if it is found.

To configure an individual QED-Flash Board for automatic initialization and file processing, craft a Priority Autostart routine that calls Set_CF_Module and then calls Do_Autoexec. Even if you do not plan to use the AUTOEXEC.QED file feature, this approach yields automatic initialization of the file system if a CF Card is detected at powerup or restart.

Simple files can perform memory imaging and automated software upgrades. Sample files are included at the end of this document, and the next section describes how to set up these files.

For those familiar with Forth, additional flexibility is available. Any executable Forth code can be handled during file processing. Files can cause other files to be processed. That is, recursive nesting is allowed (although End_Update halts recursion; see its glossary entry). The key restriction is that the Forth headers of any function that is referenced must be loaded. Forth programmers will typically load the library.4th file that declares the CF Module Forth headers, so that the file operations can be referenced during file processing.

Upgrade note for former Memory Interface Board (MIB) Users

The Link_File_IO function is no longer needed and should never be called. If called, it would load the headers for the obsolete MIB software in the QED V4.xx kernel. Note that the Arm_Autoexec and Disarm_Autoexec functions are no longer implemented. Moreover, because the CF Card Software is not a built-in part of the QED-Forth kernel, the allowed contents of the AUTOSTART.QED file may be restricted, depending on how the CF Card Software was loaded into memory. The recommended approach to creating a file for run-time processing is described in the following section. A more detailed upgrade notice is presented after the glossary.

How To Create and Use an AUTOEXEC.QED File

The CF Card Software facilitates in-the-field software upgrades. A small AUTOEXEC.QED file can copy the image of upgraded software page-by-page into the memory of the QED Board, Panel-Touch Controller, or EtherSmart Controller. The operating system can be configured to look for and automatically execute such a file named AUTOEXEC.QED each time the system starts up. This means that an upgrade can be performed by simply plugging a pre-configured CF card into its socket.

The most common uses for the automated processing capability are to image the QED memory into files, and transfer the contents of files to memory. Two powerful functions named Page_To_File and File_To_Page have been designed to make things easy. Each of these functions accepts a single parameter that specifies a 32 Kbyte page of memory, and accesses a binary file. The filename is fixed as PAGEpp.BIN, where pp is the 2-digit hexadecimal representation of the specified page. Page_To_File creates a 32 Kbyte file whose contents equal the contents of the specified page. File_To_Page reads a 32 Kbyte file and stores the contents to the specified page in memory.

To perform a software upgrade, we will see how to:

1. compile a program into flash memory;
2. create a set of files that contain the desired images of the flash pages; and
2. transfer the contents of the image files to a target system's memory in the field.

Compile a Program into Flash Memory

Pages 4, 5, and 6 of memory typically contain the program code of an application. These pages are flash memory in the Standard.Map as described in the QED Board documentation. The top-level function of a C application is called "main", and this is located on page 4 in QED memory. The top-level function of a QED-Forth program can be located on page 4, 5 or 6.

To take advantage of automated file processing, include two statements in your top-level function: the first is a call to Set_CF_Module, and the second invokes the Do_Autoexec function. Note that parameter passed to the Set_CF_Module funtion must match the hardware jumper settings described in Table 2 above. Let's assume that the CF Module is configured as module number 0. In C, the following two statements should be in the top-level function:

```
Set_CF_Module( 0 );
Do_Autoexec( );
```

In Forth, the required statements are:

```
0  Set_CF_Module
Do_Autoexec
```

In a typical application, the top-level function is configured as a "Priority Autostart" routine. The Priority Autostart vector is stored near the top of page 4 flash memory. For example, the function named "main" is made to automatically execute upon each startup or restart by the declaration:

```
CFA.FOR  main  PRIORITY.AUTOSTART
```

which is typically typed at the terminal. (The EtherSmart Controller uses the syntax: Autostart: main). This statement causes an autostart vector to be written to the top of page 4, causing the specified function to be automatically executed at startup.

Create a Set of Image Files

After compiling the application and declaring the flash-based Autostart vector, we are ready to make a set of image files named PAGE04.BIN, PAGE05.BIN, and PAGE06.BIN. All we need to do is use a PC to store a proper version of the AUTOEXEC.QED file on a CF Card, install the CF Card into its socket on the CF Module, and restart the board. After the restart (which can be invoked from the terminal by typing WARM or ABORT), there will be a delay, then you will see the echoed report of the file processing.

Listing 1 shows the contents of the required file, which is provided in the PAGE2FILE directory of the CF Module code distribution.

Listing 1. Page-to-File Version of AUTOEXEC.QED

```
\ This AUTOEXEC.QED file is executed at startup
\ if the autostart routine calls Set_CF_Module and Do_Autoexec.
\ It creates a set of 32 Kbyte files, one per page for the specified pages.

\ Comment out the following line to suppress echoing to the serial port:
cfa.for emit1 uemit x!  \ Vector output to the primary serial port

standard.map  \ in the standard map, pages 4,5,6 are flash
hex  \ interpret numbers in hexadecimal base

\ comment out the lines for any pages you do not want to image into a file:
4 Page_To_File
5 Page_To_File
6 Page_To_File
```

The file listing starts with some comments delimited by the \ (backslash plus space) comment character. The first executable line vectors the "emit" serial output primitive to the default primary serial port so that the progress of the upgrade can be monitored at the serial terminal. If this line is commented out by placing a backslash and a space at the start of the line, the serial output is suppressed during processing of the file. It is recommended that serial output be enabled, as the Page_To_File routine prints useful diagnostic information while it runs.

The `standard.map` directive ensures that we are in the standard memory map (pages 4, 5, and 6 are flash). The `hex` directive means that all numbers (in this case, the page numbers) are interpreted in hexadecimal base. This directive can be changed to `decimal` if you prefer. Do not use the prefix "0x" to denote hex numbers in the AUTOEXEC.QED file.

The next three executable lines specify that pages 4, 5, and 6 are to be imaged into files named PAGE04.BIN, PAGE05.BIN, and PAGE06.BIN, respectively. Note that there must be at least one space between the page number and the function name. You can comment out any lines that refer to pages you do not want to image, and you can add additional lines if necessary. If these Page_To_File commands are executed after software has been loaded into flash memory, then the files contain the information required to perform a software upgrade, including setting up a revised Priority Autostart vector.

Any page can be imaged, including RAM pages 1, 2, and 3, and writeable flash pages 7 and 0x0D (decimal thirteen). If you are using a Wildcard Carrier Board with installed memory, then RAM is present starting at page 0x40, and flash is present starting at page 0x50 (check the board specifications to determine how much memory is available).

You can also create the image files by executing a program that calls Set_CF_Module, calls Init_File_System, and invokes the Page_To_File function for each page to be imaged.

Note to experts: You can also use the File_Open and File_Write functions called from a runtime C or Forth program to save memory contents to files. This gives you the option of specifying custom file names and the address ranges imaged into the files. In this case, however, you cannot use File_To_Page to store the file contents to memory; rather, you would use customized functions that you write.

Transfer the Image File Contents into Memory

Now let's assume that you want to create a CF card that will automatically install into a field instrument the software images contained in PAGE04.BIN, PAGE05.BIN, and PAGE06.BIN. We also assume that the Autostart routine running the the controller calls Set_CF_Module and Do_Autoexec as described above. All we need to do is use the PC to replace the AUTOEXEC.QED file on the CF card with a file that contains the proper File_To_Page commands, install the CF Card into its socket on the CF Module, and restart the board. Listing 2 shows the contents of the required file, which is provided in the FILE2PAGE directory of the CF Module code distribution.

Listing 2. File-to-Page Version of AUTOEXEC.QED

```
\ This AUTOEXEC.QED file is executed at startup
\ if the autostart routine calls Set_CF_Module and Do_Autoexec.
\ It copies the contents of a set of 32 Kbyte files into memory (RAM or Flash).
\ There must be one file named PAGEpp.BIN per page, where pp is the (hex) page.

\ Comment out the following line to suppress echoing to the serial port:
cfa.for emit1 uemit x!  \ Vector output to the primary serial port

standard.map   \ in the standard map, pages 4,5,6 are flash
hex  \ interpret numbers in hexadecimal base

\ comment out the lines for any pages you do not want to update:
4 File_To_Page
5 File_To_Page
6 File_To_Page
\ you may remove the following line if you are not updating the autostart routine:
End_Update
```

The file listing starts with some comments delimited by the \ (backslash plus space) comment character. The first executable line vectors the "emit" serial output primitive to the default primary serial port so that the progress of the upgrade can be monitored at the serial terminal. If this line is commented out by placing a backslash and a space at the start of the line, the serial output is suppressed during processing of the file. It is recommended that serial output be enabled, as the File_To_Page routine prints useful diagnostic information while it runs.

The `standard.map` directive ensures that we are in the standard memory map (pages 4, 5, and 6 are flash). The `hex` directive means that all numbers (in this case, the page numbers) are interpreted in hexadecimal base. This directive can be changed to `decimal` if you prefer. Do not use the prefix "0x" to denote hex numbers in the AUTOEXEC.QED file.

The next three executable lines specify that the contents of files PAGE04.BIN, PAGE05.BIN, and PAGE06.BIN are to be transferred to memory pages 4, 5, and 6, respectively. You can comment out any lines that refer to pages you do not want to store, and you can add additional lines to update other pages that were imaged into properly named binary files. These operations perform the software upgrade.

The last line of the file handles the tricky situation that develops when one attempts to upgrade software that is running while the upgrade is in progress. In this case, the top-level Priority Autostart routine (for example, "main") invoked the Do_Autoexec function that started the upgrade process. If we're not very careful, the processor could try to continue executing a function that has been changed by the upgrade process! This could cause a software crash. To avoid this problem, End_Update prints the message:

Update complete, please remove CF Card...

and waits until the CF card is removed before allowing execution to proceed beyond the end of the End_Update function. When the CF card is removed, End_Update executes ABORT to restart the controller. If you are sure that no executing functions are affected by the File_To_Page directives, the

End_Update function may be commented out. If present, End_Update must be the last function in the file.

At the completion of processing this AUTOEXEC.QED file, the software upgrade is done. As far as the field user was concerned, all that was required was to insert a CF Card, reboot the board (via reset or power cycle), wait for the upgrade to finish, and remove the CF card.

Note to experts: The contents of the AUTOEXEC.QED file must be capable of being processed (interpreted and executed) at runtime by the QED-Forth interpreter. The Page_To_File, File_To_Page, and End_Update functions are always recognized by the file interpreter, even if the programmer is using C and has not loaded the Forth-style CF Card Software headers to the QED Board.

Restrictions on Software Upgrades

Overwriting the page that contains the CF Module driver code is not allowed during a software upgrade. This is because the CF Module code is executing during the upgrade, and flash-resident code cannot in general modify itself while it is running. The CF Module driver code is supplied as a "kernel extension" and generally resides in page 0x0D (decimal page thirteen), but it may reside in page 7 in some systems. If you have any questions, please contact Mosaic Industries for technical support.

Likewise, the data structures used by the CF Module driver code cannot be modified during an upgrade. The top 16 Kbytes of page 3 are the default location for these data structures; see the glossary entry for Init_File_System. Thus, unless you relocate these data structures, File_To_Page should not be used to write to page 3.

Recovering from Buggy AUTOSTART.QED Files

If an error occurs during the automated processing of the AUTOEXEC.QED file at startup, the operating system will continue trying to process the file contents until the card is removed or until a "factory cleanup" operation is performed. This implies two things. First, all AUTOEXEC.QED files should be carefully debugged before attempting to automatically load them into a QED Board. Second, if your AUTOEXEC.QED file does have a bug in it, remove the CF Card from the socket (or perform a factory cleanup using the DIP switches) to regain control of the computer, and then repair the file and re-post the Autostart or Priority Autostart function.

Compact Flash Card Software Package Main Glossary

C: **APLUS_MODE**
4th: **APLUS_MODE** (-- n)
A constant that is passed as a file-access privilege parameter to the File_Open function to indicate that the file may be read or written. If APLUS_MODE is specified, random-access reads are allowed, but all data written to the file will automatically be appended (and the file pointer will be moved) to the end of the file regardless of the original file position. See also R_MODE, W_MODE, A_MODE, RPLUS_MODE, and WPLUS_MODE.

C: int **ATA_Command**(xaddr xbuf, ulong startsector, uint numsectors, int cmd, int xfer_type, int features)
4th: **ATA_Command** (xbuf\startsector\numsectors\cmd\xfer_type\features -- failure_code)
This primitive function is rarely used by the programmer; it is called by higher level functions that perform Read, Write, ID Drive, and Set Features operations on the ATA device. This routine performs the ATA command specified by parameter cmd, transferring data to/from the card as indicated by the xfer_type using the specified host data buffer starting at xaddress xbuf (the "host" is the QED Board). Command codes are specified by the ATA standard. Valid transfer types are 0 (no transfer), 1 (card-to-host transfer), or 2 (host-to-card transfer). The features parameter is used by the Set Features command (see ATA_Set_Features). The returned failure parameter equals 0 if no error occurred; otherwise a FAIL_ code is returned to describe the error (see the glossary entries for the named constants that begin with "FAIL", and ATA_Fail). If numsectors = 0, no sectors are transferred. If numsectors > 255, the specified command is sent to the ATA device multiple times, each time with sector_count constrained to the range 0 to 255. The maximum value of numsectors is 65,535, leading to a maximum data transfer size of just under 32 Megabytes. The actual number of sectors transferred can be determined by calling the Numsectors_Transferred function. Make sure that ATA_ID_Drive has been executed since the last restart before calling this function; otherwise the specified command will not be executed, thereby protecting against reads and writes to an unknown device. This function Gets and Releases the CF resource variable (located at the 4 bytes starting at FI) to manage multitask access. This command works on hard drives up to 128 Gigabytes. NOTE that accesses to the CF Card should NOT be executed from within an interrupt service routine, as the PAUSE operation called by the Get function may cause an infinite delay if invoked from within an interrupt routine.

C: int **ATA_Fail**(void)
4th: **ATA_Fail** (-- fail_code)
Returns the failure code from the most recently executed ATA command. The returned failure parameter equals 0 if no error occurred; otherwise a nonzero code is returned to describe the error. If several errors occurred, the relevant bit-mask FAIL_ codes are OR'd together. See the glossary entries for the named constants that begin with "FAIL".

C: int **ATA_ID_Drive**(xaddr xbuffer)
4th: **ATA_ID_Drive** (xbuffer -- failure_code)

This function transfers 512 bytes of identification information from the ATA card to QED memory starting at the 32-bit QED address xbuffer. This information is used by the higher level initialization functions to store relevant information about the card in the file system data structures. The user typically does not invoke this command directly; rather, it is called by the high level functions Init_File_IO or Init_File_System which first call Init_File_Heap (which sets up the data structures used by this function) and then call this ATA_ID_Drive function. This command must be called (via Init_File_IO or Init_File_System) after every powerup or restart, before other accesses to the ATA card; otherwise the specified ATA access command will not be executed and higher level commands will return the ERR_FAIL_ID_DRIVE error code. This scheme protects against reads and writes to an unknown device. This command must be called (via Init_File_IO or Init_File_System) after a new ATA card is inserted into the CF Card socket. The returned failure parameter equals 0 if no error occurred; otherwise a FAIL_ code is returned to describe the error (see the glossary entries for the named constants that begin with "FAIL"). If no error occurs during command, this routine initializes the parameter used by the Card_Size function to report the number of bytes in the card. This function Gets and Releases the CF resource variable (located at the 4 bytes starting at FI) to manage multitask access. NOTE that accesses to the CF Card should NOT be executed from within an interrupt service routine, as the PAUSE operation called by the Get function may cause an infinite delay if invoked from within an interrupt routine. For implementation details, see ATA_Command.

C: int **ATA_Read**(xaddr xbuf, ulong startsector, uint numsectors)
4th: **ATA_Read** (xbuf\d.startsector\numsectors -- failure_code)

Transfers data from the ATA flash card to QED memory (the "host"). The source sector in the ATA card is specified by the 32-bit startsector parameter. The destination data is written starting at extended address xbuf in the QED memory. The number of bytes transferred equals numsectors times 512 (see SECTOR_SIZE). The returned failure parameter equals 0 if no error occurred; otherwise a FAIL_ code is returned to describe the error (see the glossary entries for the named constants that begin with "FAIL"). If numsectors = 0, no sectors are transferred. The maximum value of numsectors is 65,535, leading to a maximum data transfer size of just under 32 Megabytes. The actual number of sectors transferred can be determined by calling the Numsectors_Transferred function. Make sure that ATA_ID_Drive has been executed since the last restart before calling this function; otherwise the specified command will not be executed, thereby protecting against reads and writes to an unknown device. This function Gets and Releases the CF resource variable (located at the 4 bytes starting at FI) to manage multitask access. This command works on hard drives up to 128 Gigabytes. NOTE that accesses to the CF Card should NOT be executed from within an interrupt service routine, as the PAUSE operation called by the Get function may cause an infinite delay if invoked from within an interrupt routine. For implementation details, see ATA_Command. If for some reason you need to read the master boot record in the hidden sector region at the physical start of the drive, specify a startsector of -{Hidden_Sectors}; see the Hidden_Sectors glossary entry.

C: int **ATA_Set_Features**(int config_contents, int feature_contents)
4th: **ATA_Set_Features** (config_contents\feature_contents -- failure_code)

Writes to the ATA card to select features. This command is typically not required and is not called by any of the high level file functions; it is included for completeness. For example, the default features for Sandisk SDCFB series cards are correct without modification. For most features, the config_contents parameter is irrelevant; the exception is the Sandisk set-current-limit feature; see the Sandisk product manual for details if you want to use this feature. This function must be called AFTER invoking Init_File_IO or Init_File_System to ensure that FI points to a block of 300 bytes of available RAM, and a 32-bit zero must be stored into the extended address returned by FI to initialize the CF resource variable. The returned failure parameter equals 0 if no error occurred; otherwise a FAIL_ code is returned to describe the error (see the glossary entries for the named constants that begin with "FAIL"). This function gets and releases the CF resource variable to manage multitask access. For implementation details, see ATA_Command.

C: int **ATA_Write**(xaddr xbuf, ulong startsector, uint numsectors)
4th: **ATA_Write** (xbuf\d.startsector\numsectors -- failure_code)

Transfers data from QED memory (the "host") to the ATA flash card. The source data starts at extended address xbuf in the QED memory. The number of bytes transferred equals numsectors times 512 (see SECTOR_SIZE). The destination sector in the ATA card is specified by the 32-bit startsector parameter. The returned failure parameter equals 0 if no error occurred; otherwise a FAIL_ code is returned to describe the error (see the glossary entries for the named constants that begin with "FAIL"). If numsectors = 0, no sectors are transferred. The maximum value of numsectors is 65,535, leading to a maximum data transfer size of just under 32 Megabytes. The actual number of sectors transferred can be determined by calling the Numsectors_Transferred function. Make sure that ATA_ID_Drive has been executed since the last restart before calling this function; otherwise the specified command will not be executed, thereby protecting against reads and writes to an unknown device. This function Gets and Releases the CF resource variable (located at the 4 bytes starting at FI) to manage multitask access. This command works on hard drives up to 128 Gigabytes. NOTE that accesses to the CF Card should NOT be executed from within an interrupt service routine, as the PAUSE operation called by the Get function may cause an infinite delay if invoked from within an interrupt routine. For implementation details, see ATA_Command.

C: **A_MODE**
4th: **A_MODE** (-- n)

A constant that is passed as a file-access privilege parameter to the File_Open function to indicate that the file is append-only. If A_MODE is specified, reads are not allowed, and all data written to the file will automatically case the file position to be moved to the end of the file regardless of the originally specified file position. See also R_MODE, W_MODE, RPLUS_MODE, WPLUS_MODE, and APLUS_MODE.

C: ulong **Card_Size**(void)
4th: **Card_Size** (-- d.size)

Returns the card size as a 32-bit number. It is calculated as:

SECTOR_SIZE * (1 + LAST_SECTOR)

where LAST_SECTOR is a value obtained by ATA_ID_Drive. The returned card size does not include the "hidden" sectors at the physical start of the drive where the master boot record is kept (see Hidden_Sectors). This software package supports ATA flash cards or hard drives up to 128 Gigabytes. Note that the file system must be initialized by Init_File_System or Init_File_IO before using this function; otherwise the returned value is meaningless.

C: void **Clear_File_Error**(int file_id)
4th: **Clear_File_Error** (file_id --)

Clears the file error flag so that an immediately following call to the File_Error function would return zero, and an immediate call to Report_File_Errors would report no errors. Also zeros the file-open error flag so that an immediate call to Report_File_Open_Errors would report no errors. This function is called by File_Rewind. See the glossary entries that begin with "Err_" for a list of bit-mapped error codes.

C: void **Dir**(void)
4th: **Dir** (--)

Prints to the active serial port a directory listing for the root directory that is similar in format to the a DOS DIR command. Does not list hidden, system, volume-label, directory, or erased files. Lists the filename and extension, filesize, time and date of last modification (using 24 hour time reporting), and summarizes the number of files listed and the total size in bytes. Typically typed interactively from the terminal to the QED-Forth monitor as:

Dir

Implementation Details: The format of each line is as follows:

filename ext filesize mm-dd-yyyy hh:mm

where the filename is 8 characters and ext is 3 characters (both are padded with spaces on the right if necessary), mm=month, dd=date, yyyy=4digit year, hh=hour (24-hour military style), and mm=minutes. The format of the final summary line is:

nn file(s) ddddddd bytes

The function prints "Initialization Error!" if the file system has not been initialized (see Init_File_System), and prints "Error: Can't read disk!" if a read error occurs.

C: void **Dir_Names**(void)
4th: **Dir_Names** (--)

Prints to the active serial port a directory listing for the files in the root directory, listing only the name and extension of each file. Does not list hidden, system, volume-label, directory, or erased files. See the glossary entry for Dir for more details.

Implementation Details: The format of each line is as follows:

filename ext

where the filename is 8 characters and ext is 3 characters (both are padded with spaces on the right if necessary).

C: void **Dir_Open**(void)
4th: **Dir_Open** (--)

Prints to the active serial port a directory listing for the open files in the root directory. Uses the same format as the DIR command, except that the file_id is also listed for each open file. See DIR. Typically typed interactively from the terminal to the QED-Forth monitor as:

Dir_Open

Implementation Details: Unlike Dir, the Dir_Open function extracts file information from RAM-based structures, and so may report information (especially file size and date/time) that has not yet been flushed to the disk. See File_Flush.

C: int **Do_Autoexec** (void)
4th: **Do_Autoexec** (-- error)

If a CF Card is present, this routine calls Init_File_System and Process_Autoexec to search for a file named AUTOEXEC.QED and process it if it is found. See the entries for Init_File_IO and Process_Autoexec for detailed descriptions. Note that the Set_CF_Module function must be executed before calling this function. The module number must correspond to the module address set by the module address selection jumpers and the module port (see Table 1 above). This function is typically called from within the Priority Autostart routine of an application, so that the CF Card (if present) is initialized and automated processing of AUTOEXEC.QED can be used for automated software upgrades or data exchanges.

Note: this function uses the top 16 Kbytes of the page 3 RAM. User-available RAM on page 3 in the STANDARD.MAP thus ends at 0x4000. If you need this page 3 RAM for other purposes, do not use this function; rather, set call Set_CF_Module, call Init_File_IO with parameters of your choosing to set up an appropriate memory map, and then explicitly call Process_Autoexec.

4th: **END_CAPTURE** (--)

A do-nothing function that serves as a marker used to terminate File_Capture. To be recognized, END_CAPTURE must be the first non-white-space item on a line. If executed by Forth (for example, during file processing), END_CAPTURE has no effect. See the glossary entry for File_Capture.

C: int **End_Update** (void)
4th: **End_Update** (-- error)

This routine must be the last function called in an AUTOEXEC.QED file (or other processed file) that is used to perform a software update of the page that contains the code that invoked the update. For example, if a priority autostart routine located on page 4 calls Do_Autoexec, and if the AUTOEXEC.QED file contains 4 File_To_Page, then that file must end with End_Update. End_Update prints a message asking the user to remove the CF Card, stops task switching, closes input and output files, restores prior serial vectors and abort parameters, waits for the CF Card to be removed, then aborts to reboot the processor. This approach prevents a crash which would occur if the autostart routine was replaced while it was executing! This routine is always callable by Process_File, even if the CF Card Forth headers have not been loaded onto the QED Board. See the "How to Create an Autoexec.QED File" section for an example of use.

C: **ERR_ATA_READ**
4th: **ERR_ATA_READ** (-- n)

A constant (equal to 0x0001) that is returned as an error code when an error occurs during a read of the ATA flash card. A high level file system routine that calls ATA_Read converts any nonzero failure code returned by ATA_Read into the ERR_ATA_READ error code. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function or by File_Error. See File_Error.

C: **ERR_ATA_WRITE**
4th: **ERR_ATA_WRITE** (-- n)

A constant (equal to 0x0002) that is returned as an error code when an error occurs during a write to the ATA flash card. A high level file system routine that calls ATA_Write converts any nonzero failure code returned by ATA_Write into the ERR_ATA_WRITE error code. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function or by File_Error. See File_Error.

C: **ERR_BAD_OR_UNOPEN_FILEID**
4th: **ERR_BAD_OR_UNOPEN_FILEID** (-- n)

A constant (equal to 0x4000) that is returned as an error code when an attempt to access a file fails because the specified file_id is invalid, or is not associated with an open file. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function. Because this error is associated with operations that are performed on a file that does not have a valid file_id, it cannot be accessed by File_Error; use Report_File_Open_Errors for diagnostics.

C: **ERR_CANNOT_ADD_CLUSTER**
4th: **ERR_CANNOT_ADD_CLUSTER** (-- n)

A constant (equal to 0x0010) that is returned as an error code when an attempt to allocate a disk cluster fails; this may be because the disk is full. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function or by File_Error. See File_Error.

C: **ERR_CANNOT_TRUNCATE_FILE**
4th: **ERR_CANNOT_TRUNCATE_FILE** (-- n)

A constant (equal to 0x0200) that is returned as an error code when an attempted truncation of a file fails. Truncation is required when opening a file with W_MODE or WPLUS_MODE access privileges. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function. Because this error is associated with operations that are performed on a file that does not have a valid file_id, it cannot be accessed by File_Error; use Report_File_Open_Errors for diagnostics.

C: **ERR_DISK_IS_FULL**
4th: **ERR_DISK_IS_FULL** (-- n)

A constant (equal to 0x1000) that is returned as an error code when an attempt to write to the disk fails because the disk is full. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function. Because this error is associated with operations that are performed on a file that does not have a valid file_id, it cannot be accessed by File_Error; use Report_File_Open_Errors for diagnostics.

C: **ERR_EOF**
4th: **ERR_EOF** (-- n)

A constant (equal to 0x8000) that is returned as an error code when an end of file condition is encountered during a file access. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function or by File_Error. See File_Error and File_EOF.

C: **ERR_FAIL_ID_DRIVE**
4th: **ERR_FAIL_ID_DRIVE** (-- n)

A constant (equal to 0x0100) that is returned as an error code when a the ATA_ID_Drive command fails. This error message is also returned if Init_File_System or Init_File_IO is not called before attempting to call a file manipulation function. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function or by File_Error (see its glossary entry). This error code is relevant for both the low level ATA drivers and the high level file system functions; thus it also has the name FAIL_ID_DRIVE to be consistent with the ATA failure codes.

C: **ERR_FILE_DOES_NOT_EXIST**
4th: **ERR_FILE_DOES_NOT_EXIST** (-- n)

A constant (equal to 0x0400) that is returned as an error code when a file is required to exist but does not. Pre-existence is required when opening a file with R_MODE or RPLUS_MODE access privileges. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function. Because this error is associated with operations that are performed on a file that does not have a valid file_id, it cannot be accessed by File_Error; use Report_File_Open_Errors for diagnostics.

C: **ERR_INVALID_SECTOR_NUMBER**
4th: **ERR_INVALID_SECTOR_NUMBER** (-- n)

A constant (equal to 0x0008) that is returned as an error code when a disk access to an out-of-range sector number is attempted. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function or by File_Error. See File_Error.

C: **ERR_NEGATIVE_FILE_POSN**
4th: **ERR_NEGATIVE_FILE_POSN** (-- n)

A constant (equal to 0x0020) that is returned as an error code when the file position pointer is set to a negative value. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function or by File_Error. See File_Error.

C: **ERR_NOT_DOSFAT_DRIVE**
4th: **ERR_NOT_DOSFAT_DRIVE** (-- n)

A constant (equal to 0x0004) that is returned as an error code when the boot sector of an ATA Flash drive is not properly formatted with a DOS-compatible FAT file system. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function or by File_Error. See File_Error.

C: **ERR_READ_ACCESS_VIOLATION**
4th: **ERR_READ_ACCESS_VIOLATION** (-- n)

A constant (equal to 0x0080) that is returned as an error code when a read is attempted on a write-only file. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function or by File_Error. See File_Error.

C: **ERR_ROOT_DIR_FULL**
4th: **ERR_ROOT_DIR_FULL** (-- n)

A constant (equal to 0x0800) that is returned as an error code when an attempt to add a file to the root directory fails. The maximum number of files in the root directory is set when the flash card is formatted. For example, the root directory of a 10 Megabyte card can contain a maximum of 512 files. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function. Because this error is associated with operations that are performed on a file that does not have a valid file_id, it cannot be accessed by File_Error; use Report_File_Open_Errors for diagnostics.

C: **ERR_TOO_MANY_FILES_OPEN**
4th: **ERR_TOO_MANY_FILES_OPEN** (-- n)

A constant (equal to 0x2000) that is returned as an error code when an attempt to open a file fails because the maximum number of files as set by Init_File_Heap (called by Init_File_IO and Init_File_System) has been exceeded. Note that some file operations such as File_Type or File_Copy need to open and then close one or two files. To proceed, use File_Close to close any unneeded files. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function. Because this error is associated with operations that are performed on a file that does not have a valid file_id, it cannot be accessed by File_Error; use Report_File_Open_Errors for diagnostics.

C: **ERR_WRITE_ACCESS_VIOLATION**
4th: **ERR_WRITE_ACCESS_VIOLATION** (-- n)

A constant (equal to 0x0040) that is returned as an error code when a write is attempted to a read-only file. Note that File_Open will set the access mode of a file to R_MODE (read only) if the directory entry for the file (initialized when the file was originally created) specifies read-only access. All error codes returned by the file system functions have names starting with "ERR", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by a calling file system function or by File_Error. See File_Error.

C: **FAIL_ADDR**
4th: **FAIL_ADDR** (-- n)

A constant (equal to 0x0004) that is returned as a failure code when the requested sector number is out of range. All failure codes returned by the low level ATA access routines (such as ATA_Read and ATA_Write) have names starting with "FAIL", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by an ATA function.

C: **FAIL_ARGS**
4th: **FAIL_ARGS** (-- n)

A constant (equal to 0x0001) that is returned as a failure code when one of the arguments that is passed to the low level ATA function is invalid. All failure codes returned by the low level ATA access routines (such as ATA_Read and ATA_Write) have names starting with "FAIL", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by an ATA function.

C: **FAIL_BUSY**
4th: **FAIL_BUSY** (-- n)
A constant (equal to 0x0080) that is returned as a failure code when a timeout error is encountered during an access to the ATA flash card. All failure codes returned by the low level ATA access routines (such as ATA_Read and ATA_Write) have names starting with "FAIL", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by an ATA function.

C: **FAIL_CMD**
4th: **FAIL_CMD** (-- n)
A constant (equal to 0x0008) that is returned as a failure code when an ATA command fails to complete. All failure codes returned by the low level ATA access routines (such as ATA_Read and ATA_Write) have names starting with "FAIL", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by an ATA function.

C: **FAIL_DATAREQ**
4th: **FAIL_DATAREQ** (-- n)
A constant (equal to 0x0020) that is returned as a failure code when the "DRQ" bit in the ATA hardware status register indicates that the drive is not synchronized with the data requests to/from the host as it should be during a data transfer. All failure codes returned by the low level ATA access routines (such as ATA_Read and ATA_Write) have names starting with "FAIL", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by an ATA function.

C: **FAIL_EXTERR**
4th: **FAIL_EXTERR** (-- n)
A constant (equal to 0x0010) that is returned as a failure code when an attempt to get an "extended error" description fails after a failure occurs while accessing the ATA device; this extended error feature is supported only by Sandisk ATA flash cards. All failure codes returned by the low level ATA access routines (such as ATA_Read and ATA_Write) have names starting with "FAIL", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by an ATA function.

C: **FAIL_ID_DRIVE**
4th: **FAIL_ID_DRIVE** (-- n)
A constant (equal to 0x0100) that is returned as a failure code when an error is encountered by the ATA_ID_Drive command. All failure codes returned by the low level ATA access routines (such as ATA_Read and ATA_Write) have names starting with "FAIL", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by an ATA function. See also ERR_FAIL_ID_DRIVE.

C: **FAIL_NO_CARD**
4th: **FAIL_NO_CARD** (-- n)
A constant (equal to 0x0002) that is returned as a failure code when there is no card present. All failure codes returned by the low level ATA access routines (such as ATA_Read and ATA_Write) have names starting with "FAIL", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by an ATA function.

C: **FAIL_READY**
4th: **FAIL_READY** (-- n)

A constant (equal to 0x0040) that is returned as a failure code when the "Drive Ready" bit in the ATA hardware status register indicates that the drive is not ready when it should be. All failure codes returned by the low level ATA access routines (such as ATA_Read and ATA_Write) have names starting with "FAIL", have a single-bit set, and may be OR'd together to represent multiple errors. This constant may be used as a bitmask to decode the value returned by an ATA function.

C: xaddr **FI**(void)
4th: **FI** (-- fat_info_xaddr)

This function returns the 32-bit extended base address (xaddr) of the fat_info structure that holds all of the key variables and array parameter fields used by the file system software. This address is set by Init_File_IO or Init_File_System. The returned fat_info_xaddr parameter is the base address of a block of 300 bytes of RAM in either common or paged memory. The first entry at offset 0 in this structure is the resource variable that mediates access to the CF Card access functions so that this software package is re-entrant with respect to multitasking. (Note that the functions that access the CF Card cannot be called from inside interrupt service routines; see the glossary entries for ATA_Read and ATA_Write). The fat_info structure stores low level parameters, buffers, and array parameter fields that manage ATA access, FAT file manipulation, and file access. The arrays associated with the array parameter fields are maintained in the heap that is initialized by Init_File_Heap. With the possible exception of getting or releasing the resource variable at the base of this structure, the programmer does not need to directly access the contents of the structure.

C: void **File_Abort_Action**(void)
4th: **File_Abort_Action** (--)

This abort handler is installed by Process_File. If an abort occurs during file processing, this handler attempts to "clean up" by calling Put_Default_Serial and by attempting to close the files indicated by the input_file_id and output_file_id parameters passed to Process_File. Then it executes the standard abort behavior. This function is typically not directly called by the programmer.

C: int **File_Ask_Key**(void)
4th: **File_Ask_Key** (-- flag)

Installed by Process_File as a revector of the AskKey (?KEY in Forth) serial input primitive function that is called by File_Interpreter. Based on the key_fileid initialized by Process_File, this function returns a -1 flag if the specified file is open and if the file_position is non-negative and is less than the file's size. If the file is not open or end of file is encountered, a false flag is returned. When coding a function that takes its output from a file via Redirect (see its glossary entry), File_Ask_Key may be called repeatedly to gracefully terminate when end of file is encountered.

C: int **File_Capture**(int file_id)
4th: **File_Capture** (fileid -- error)

Stores a stream of text that is being sent to the QED Board via the active serial port into the specified open file. If used interactively, any commands following File_Capture on the same line will be ignored, but text starting with the next line will be captured. This function puts text into the specified file starting at the current file position, ending each line with 'CR' 'LF' (hex 0D 0A), DOS-style. Terminates and closes the specified file when a line containing END_CAPTURE as its first blank-delimited word is encountered; END_CAPTURE is not written to the file. Also terminates and closes the specified file if the called function File_Write returns a nonzero error code.

CAUTION: If this function or other functions that accept name strings as parameters are typed interactively at the QED-Forth monitor using the Forth syntax, the dictionary pointer must point to modifiable RAM, not flash memory. This is because Forth emplaces strings typed interactively at the current dictionary pointer, and attempts to emplace data in flash without using the specially coded flash operations will cause a software crash. To avoid this problem, you can usually type the command

```
1  USE.PAGE
```

before an interactive session; this puts the dictionary pointer on page 1 RAM on a QED-Flash board that has 128K RAM and is operating in the STANDARD.MAP mode. If only 32K of RAM is present in the center socket, you could type

```
HEX BO00 0 DP X!
```

to locate the dictionary pointer in the processor's on-chip RAM.

EXAMPLE OF INTERACTIVE USE: This interactive Forth code opens a file, captures text into it, and closes the file (note that there is 1 blank space after the leading quote in the MYFILE.TXT string):

```
INTEGER: myfileID
" MYFILE.TXT" COUNT W_MODE  File_Open  TO myfileID
myfileID  File_Capture
< text to be captured goes here; may be 1 or many lines>
END_CAPTURE
```

C: int **File_Close**(int file_id)
4th: **File_Close** (file_id -- error)

Marks the specified file as closed and frees its file_id. If the specified file is writeable, calls File_Flush to update the disk version of the file. If the flush operation fails, returns ERR_ATA_WRITE. If the specified file_id is invalid, returns ERR_BAD_OR_UNOPEN_FILEID.

C: xaddr **File_Contents**(uint file_position, int file_id)
4th: **File_Contents** (file_position\file_id -- xaddr)

Returns the extended address in the file_contents array of the specified position (byte offset) for the file specified by file_id. This buffer is dimensioned by Init_File_Heap (typically called via Init_File_IO or Init_File_System) and is available to the programmer as a scratchpad area for file operations. The File_Contents buffer of a file that is explicitly opened by the user is modified only by the File_To_Memory function, and not by any of the other pre-coded file system functions. File_Copy and File_Type transiently open a file and use its File_Contents buffer as a scratchpad. C programmers should note that this buffer is in paged memory, so the paged memory access functions such as StoreChar(), FetchChar(), and CmoveMany() must be used to access the buffer.

C: int **File_Copy**(char* prior_name_addr, uint prior_name_page, int prior_count,
char* new_name_addr, uint new_name_page, int new_count)

4th: **File_Copy** (prior_name_xaddr\prior_count\new_name_xaddr\new_count--error)

Duplicates the specified prior file's contents into a file having a specified new name in the root directory, leaving the prior file unchanged. Returns a nonzero error code if the operation failed. The filenames are passed to the function as a pair of character strings, each of whose first byte is at the specified address on the specified page. Forth programmers should pass strings that have been "unpacked" using the COUNT function. C programmers must pass name_count = -1 for each string to inform the function that these are null-terminated strings. C programmers will typically pass the THIS_PAGE macro (defined in \include\mosaic\types.h) to specify the prior_name_page and new_name_page parameters. See the glossary entry for File_Open to review the restrictions on the names. This File_Copy function opens the two specified files; this implies that there must be at least two available file_id's before this function is called; otherwise the ERR_TOO_MANY_FILES_OPEN error code is returned. Returns the ERR_FILE_DOES_NOT_EXIST error code if the specified prior filename does not exist. Returns -1 if a file with the specified new filename already exists, or if the file system was not properly initialized. See the CAUTION notice in the File_Open glossary entry if you intend to call this function interactively.

C: void **File_Emit**(char c)

4th: **File_Emit** (char --)

Installed by Process_File as a replacement for (revector of) the Emit serial output primitive function that is called by File_Interpreter. This function is typically not used directly by the programmer. Based on the emit_fileid initialized by Process_File, this function calls File_Putc to write the next character to the specified file if emit_fileid is non-negative. If the emit_fileid equals -1, then a standard Emit to the active serial port is performed, and if the emit_fileid equals NO_EMIT_FILEID, then no echo is performed by Process_File; these options offer much faster execution than echoing into a file.

C: int **File_EOF**(int file_id)

4th: **File_EOF** (file_id -- flag)

Examines the error code for the specified file and returns true (-1) if the End Of File (EOF) was encountered during a file operation; otherwise returns false (0). See also ERR_EOF, File_Error, Clear_File_Error, Report_File_Errors, and Report_File_Open_Errors.

C: int **File_Error**(int file_id)

4th: **File_Error** (file_id -- error_code)

Returns the error code for the specified file. See the glossary entries that begin with "Err_" for a list of bit-mapped error codes. See also Clear_File_Error, Report_File_Errors, and Report_File_Open_Errors.

C: int **File_Flush**(int file_id)

4th: **File_Flush** (file_id -- error)

If the specified file is writeable, flushes to the flash card the ATA buffers, FAT buffers, and directory entry information whose corresponding update bits were set, and then clears the update bits. The buffers are not erased. Upon failure, returns a nonzero error code; see the glossary entries that start with "ERR_" for a complete list of error codes. Unlike some file system implementations, this flush routine is called automatically as needed by the File_Read and File_Write functions, so reads and writes to a file may be freely mixed without explicit calls to File_Flush. This function is also called by File_Seek, File_Set_Pos, File_Rewind, and File_Close.

C: int **File_Getc**(int file_id)
4th: **File_Getc** (file_id -- [char] or [err_eof])

Reads and returns 1 character at the current file position from the specified open file in the root directory, and then increments the file position pointer by 1. If there was a prior call to the File_Ungetc function, the "ungot" character is returned in place of the first file character. Returns ERR_EOF if the end of file was reached before the character was read, or if a read error occurred. If ERR_EOF is returned, further information about the error may be obtained by executing File_Error; see its glossary entry. See also File_Read and File_Putc.

C: int **File_Gets**(char* dest_addr, uint dest_page, uint bufsize, int file_id)
4th: **File_Gets** (dest_xaddr\bufsize\file_id -- numchars_read)

Reads a string from a text file and returns the number of characters read as a 16-bit number. Reads the contents of the specified file starting at the current file position, into the destination buffer starting at dest_addr on the specified page. C programmers will typically specify a dest_addr in common RAM, with dest_page = 0. Terminates when bufsize - 1 chars have been read, or when EOF (end of file) is reached, or when a linefeed (ASCII 0x0A) is encountered, whichever occurs first. The terminating linefeed (if present) is included at the end of the stored string. If at least 1 character was read, this function stores a null (0) byte in the destination buffer after the last char read; the null character is not included in the returned numchars_read parameter. Returns 0 if the end of file is encountered before any chars are read; in this case, nothing is stored in the destination buffer. If there was a prior call to the File_Ungetc function, the "ungot" character is moved to the destination in place of the first file character. Note that, unlike fgets() in C, this File_Gets function returns numchars_read instead of returning a pointer to the destination string. Also note that the LF (linefeed) end-of-line character scheme works for DOS text files which use CR/LF (carriage return followed by linefeed) to end each line. It also works for UNIX, which ends text lines with a LF character. Use cautiously with Apple Macintosh text files, which end lines with only the CR character. Use File_Error or File_EOF to test for errors after this function executes. See also File_Puts.

Benchmark: This function typically executes in under 14 milliseconds per character.

C: void **File_Interpreter**(void)
4th: **File_Interpreter** (--)

A replacement for QUIT, the standard QED-Forth interpreter. This function is typically not called directly; rather, it is invoked by Process_File. It is an infinite loop that typically takes its input from a specified file. It is capable of loading hex records (Intel or Motorola S records) into memory, calling File_To_Memory to perform a fast binary transfer of code, or executing or compiling any other valid Forth commands including commands to program flash, remove or set priority autostart vectors, or even recursively interpret other files. It is terminated by error-induced aborts, or when an end of file is encountered.

Implementation Details: This modified Forth interpreter assumes that the serial input primitives Key and AskKey have been revectored to point to File_Key and File_Ask_Key (see their glossary entries) so that input comes from a file, and that SERIAL_ACCESS has been properly initialized. These details are handled automatically by Process_File (which passes the xcfa of File_Interpreter to the Redirect function); see its glossary entry.

C: uchar **File_Key**(void)
4th: **File_Key** (-- char)

Installed by Process_File as a revector of the Key serial input primitive function that is called by File_Interpreter. This function is typically not used directly by the programmer. Based on the key_fileid initialized by Process_File, this function calls File_Getc to get the next character from the file. If the end of file is encountered, this function returns an ASCII carriage return. No error reporting is performed via the return value; the calling program must use File_Error to trap errors such as end of file.

C: int **File_Open**(char* name_addr, uint name_page, int name_count, int access_mode)
4th: **File_Open** (name_xaddr\name_count\access_mode -- file_id)

Opens in the root directory the file having the name and extension contained in the name_addr string, and assigns access privileges as specified by the access_mode. Returns a non-negative file_id if the open was successful. Returns -2 if the file system was not initialized (see Init_File_System); returns -1 if the open failed for any other reason. Valid access modes are R_MODE, RPLUS_MODE, W_MODE, WPLUS_MODE, A_MODE, and APLUS_MODE. The access modes behave just like the ANSI C standard access modes; see their glossary entries for details. Briefly, the modes that start with "R" (read) require that the file pre-exists. The modes that start with "W" (write) cause a like-named pre-existing file (if present) to be truncated to zero size. The modes that start with "A" (append) force any write to occur at the end of the file. The modes that include "PLUS" allow both reads and writes; otherwise, only the operation suggested by the leading letter of the mode is allowed ("R" = read, "W" or "A" = write). All files are treated as binary files (that is, text files are not stored in a different format than other files). The filename is passed to the function as a character string whose first byte is at the specified name_addr on the specified page. Forth programmers should pass a string that has been "unpacked" using the COUNT function. C programmers must pass name_count = -1 to inform the function that this is a null-terminated string. C programmers will typically pass the THIS_PAGE macro (defined in \include\mosaic\types.h) to specify the name_page parameter. The name string may have up to 8 characters followed by an optional period and up to 3 extension characters. No leading or trailing spaces are allowed in the name string. All names are automatically converted to upper case by this function. "Long file names" (that is, more than 8 characters in the name) are not supported. The following are examples of valid names:

myname.4th
MYNAME.C
MYname
SOMEBODY.TXT

The following are invalid for the reason shown in parentheses:

MYNAME.	(no extension is present after the .)
..	(directories are not supported)
TOOMANYCHARS.txt	(too many characters in name)
MYNAME.MANY	(too many characters in extension)

NOTE: It is the user's responsibility to pass a correctly formed name string to the function. Not all malformed strings will produce an explicit error condition.

The function will return an error (that is, file_id = -1) under the following conditions:

truncate failed in W_MODE or WPLUS_MODE;
file did not pre-exist in R_MODE or RPLUS_MODE;
too many files are open (i.e., no file_id's are available);
disk is full;
root directory is full (see ERR_ROOT_DIR_FULL for details).

See the glossary entries that start with "ERR_" for a complete list of error codes.

This function sets the file position pointer to zero, but note that File_Write may automatically relocate the file position to the end of the file if the access mode is A_MODE or APLUS_MODE. If a pre-existing file is opened and its directory entry specifies read-only access, the file's access mode is set to R_MODE regardless of the access_mode parameter that is passed to this function.

CAUTION: If this function or other functions that accept name strings as parameters are typed interactively at the QED-Forth monitor using the Forth syntax, the dictionary pointer must point to modifiable RAM, not flash memory. This is because Forth emplaces strings typed interactively at the current dictionary pointer, and attempts to emplace data in flash without using the specially coded flash operations will cause a software crash. To avoid this problem, you can usually type the command

```
1 USE.PAGE
```

before an interactive session; this puts the dictionary pointer on page 1 RAM on a QED-Flash board that has 128K RAM and is operating in the STANDARD.MAP mode. If only 32K of RAM is present in the center socket, you could type

HEX BO00 0 DP X!

to locate the dictionary pointer in the processor's on-chip RAM.

EXAMPLE OF INTERACTIVE USE: Typical interactive use from the terminal is as follows (note that there is 1 blank space after the leading quote in the MYFILE.TXT string):

INTEGER: myfileID

" MYFILE.TXT" COUNT WPLUS_MODE FILE_OPEN TO myfileID

Then, to put text into the file, see the glossary entry for File_Capture. To close the file, execute:

myfileID FILE_CLOSE

C: int **File_Putc**(char c, int file_id)

4th: **File_Putc** (char\file_id -- [char] or [err_eof])

Writes the specified character to the specified open file in the root directory at the current file position, and increments the file position pointer by 1. Returns ERR_EOF if a write error occurred. If ERR_EOF is returned, further information about the error may be obtained by executing File_Error; see its glossary entry. See also File_Write and File_Getc.

C: int **File_Puts**(char* source_addr, uint source_page, uint maxchars, int file_id)

4th: **File_Puts** (source_xaddr\maxchars\file_id -- numchars_written)

Writes a string to a text file and returns the number of characters written as a 16-bit number. Writes the contents from the source buffer starting at source_addr on the specified page to the specified file starting at the current file position. C programmers will typically specify a source_addr in common RAM with source_page = 0. Terminates when maxchars bytes have been written, or when a null character in the source buffer is encountered, whichever occurs first. There is no special treatment of any characters other than the null. Note that, unlike fputs() in C, this File_Puts function returns numchars_written instead of returning a pointer to the source string. Use File_Error or File_EOF to test for errors after this function executes. See also File_Gets.

Benchmark: This function typically executes in under 21 milliseconds per character.

C: int **File_Put_CRLF**(int file_id)

4th: **File_Put_CRLF** (file_id--error)

A handy utility for emplacing a carriage return (ASCII 0x0D) followed by a linefeed (ASCII 0x0A) sequence into the specified file at the current file position. Increments the file position by 2, and returns a nonzero error flag if a write error occurred.

C: long **File_Read**(void* dest_addr, uint dest_page, ulong numbytes, int file_id)

4th: **File_Read** (dest_xaddr\d.numbytes\file_id -- d.numbtyes_read)

Reads from the specified open file to the specified destination in RAM memory, and returns the number of bytes read as a 32-bit number. (See File_To_Memory if you need to transfer data from a file to flash memory). This function reads numbytes characters starting at the current file position. The bytes are placed in memory starting at the specified dest_addr on the specified page. Programs that read small amounts of data at a time may specify a dest_addr in common RAM with dest_page = 0. A standard procedure is for the calling program to test whether the returned number of bytes read equals the specified input parameter numbytes, and to infer that an error condition occurred if they are unequal. If an error occurs during the read operation, the error code is OR'd with any prior error condition, and the composite bitmapped error code can be retrieved by calling File_Error. An end of file condition can be separately tested for by calling the File_EOF function. See the glossary entries that start with "ERR_" for a complete list of error codes.

Implementation details: If there was a prior call to the File_Ungetc function, the "ungot" character is moved to the destination in place of the first file character. This function uses a modified buffered I/O scheme. Two 1-sector (512-byte) buffers are maintained. Each read operation is broken into 3 segments that are aligned to the sectors on the flash disk device: a starting partial sector, a number of whole sectors, and an ending partial sector. The starting partial sector (if present) is handled by pre-reading the full starting sector's contents from the disk into the starting sector buffer, and then moving the partial sector contents from the buffer

to the destination in memory. The whole sectors (if present) are transferred directly from the file on disk to the destination memory without the use of buffers; this speeds the process. The ending partial sector (if present) is handled in the same manner as the starting partial sector. Before reading data into either partial sector buffer, this File_Read function automatically flushes any buffers that require flushing; this allows the user to freely mix File_Read, File_Write, File_Seek, and File_Set_Pos operations without explicit calls to File_Flush.
Benchmark: Transfers from the ATA flash card to RAM take approximately 125 msec per Kbyte.

C: int **File_Remove**(char* name_addr, uint name_page, int name_count)
4th: **File_Remove** (name_xaddr\name_count -- error)
Removes the pre-existing file from the root directory with the name and extension contained in the string at name_addr on the specified page, and returns a nonzero error code if the operation failed. The filename is passed to the function as a character string whose first byte is at the specified name_addr on the specified page. Forth programmers should pass a string that has been "unpacked" using the COUNT function. C programmers must pass name_count = -1 to inform the function that this is a null-terminated string. C programmers will typically pass the THIS_PAGE macro (defined in \include\mosaic\types.h) to specify the name_page parameter. See the glossary entry for File_Open to review the restrictions on the name. This File_Remove function opens the specified file before removing it; this implies that there must be at least one available file_id before this function is called; otherwise the ERR_TOO_MANY_FILES_OPEN error code is returned. Returns the ERR_FILE_DOES_NOT_EXIST error code if the specified file does not exist. Returns -1 if the specified file was already open. See the CAUTION notice in the File_Open glossary entry if you intend to call this function interactively.
Implementation details: This function opens the specified file in the root directory, writes 0 to the file's clusters in the FAT so no clusters are assigned, then writes the DOS-specified 0xE5 (erase character) to the first character of the name in the specified file's directory entry to mark it as erased. Finally, closes the file which flushes the modified FAT and directory entry contents to disk.

C: int **File_Rename**(char* prior_name_addr, uint prior_name_page, int prior_count,
char* new_name_addr, uint new_name_page, int new_count)
4th: **File_Rename** (prior_name_xaddr\prior_count\new_name_xaddr\new_count--error)
Renames the pre-existing file in the root directory with the prior name specified by the string at prior_name_addr on the specified page. Assigns the new name specified by the string at new_name_addr on the specified page. Returns a nonzero error code if the operation failed. The filenames are passed to the function as a pair of character strings, each of whose first byte is at the specified address on the specified page. Forth programmers should pass strings that have been "unpacked" using the COUNT function. C programmers must pass name_count = -1 for each string to inform the function that these are null-terminated strings. C programmers will typically pass the THIS_PAGE macro (defined in \include\mosaic\types.h) to specify the prior_name_page and new_name_page parameters. See the glossary entry for File_Open to review the restrictions on the names. This File_Rename function opens the specified file before removing it; this implies that there must be at least one available file_id before this function is called; otherwise the ERR_TOO_MANY_FILES_OPEN error code is returned. Returns the ERR_FILE_DOES_NOT_EXIST error code if the specified prior filename does not exist. Returns -1 if the specified file was already open, or if a file with the specified new filename already exists, or if the file system was not properly initialized. See the CAUTION notice in the File_Open glossary entry if you intend to call this function interactively.
Implementation details: This function opens the specified file in the root directory, rewrites the name in the specified file's directory entry, and closes the file, which flushes the modified directory entry contents to disk.

C: int **File_Rewind**(int file_id)
4th: **File_Rewind** (file_id --)

Sets the file position indicator of the specified file to point to the first byte of the file at offset 0, and clears the error flag associated with the file. Equivalent to calling File_Seek with offset = 0 and mode = FROM_START, except that no error value is returned. See File_Seek.

C: int **File_Seek**(int file_id, long offset, int mode)
4th: **File_Seek** (file_id\d.offset\mode -- error)

Moves the file position indicator according to the specified signed 32-bit offset and mode; subsequent reads and writes of the specified file will start at the indicated file position. The allowed mode parameters are FROM_START, FROM_CURRENT, and FROM_END; see their glossary entries. Note that specifying offset = 0 with mode = FROM_END sets the file position to (filesize-1) which points to the last character in the file. To append a character at the end of a file, specify offset = 1 with mode = FROM_END (that is, point 1 byte beyond the end of the file). Specifying offset = 0 with mode = FROM_START points to the first byte in the file. If the seek operation is successful, the function sets the file position, calls File_Flush, undoes the effect of any prior call to Ungetc, clears the EOF (end of file) error flag, and returns zero to indicate success. If the requested position is less than 0, returns the ERR_NEGATIVE_FILE_POSN error code and leaves the file position unchanged. If the requested position is greater than the filesize, returns ERR_EOF and leaves the file position unchanged. If the specified file_id is invalid, returns ERR_BAD_OR_UNOPEN_FILEID. See also File_Set_Pos, File_Rewind, and File_Tell_Pos.

C: int **File_Set_Pos**(int file_id, long offset)
4th: **File_Set_Pos** (file_id\d.offset -- error)

Moves the file position indicator of the specified file to the specified positive 32-bit offset from the start of the file by calling File_Seek using the FROM_START mode. See File_Seek, File_Rewind, and File_Tell_Pos.

C: ulong **File_Size**(int file_id)
4th: **File_Size** (file_id -- d.file_size)

Returns the size of the specified file as a 32-bit number.

C: long **File_Tell_Pos**(int file_id)
4th: **File_Tell_Pos** (file_id -- d.offset)

Returns the current file position of the specified file as a 32-bit offset from the start of the file. The first byte in the file is at offset = 0, and the last byte in the file is at offset (filesize - 1). Returns -1 if the file is not open. See File_Seek and File_Set_Pos.

C: xaddr **File_Time**(int file_id)
4th: **File_Time** (file_id -- xaddr)

Based on the information stored in the FAT directory entry of the specified file, decodes time and date and writes it to a structure associated with the file located in the file system heap. Returns the 32-bit extended base address of the structure. This low-level function is provided as a convenience; it is not used in most applications. The data is stored as follows:

Data	Offset	Type	Min.	Max.
Seconds	0	byte	0	59
Minutes	1	byte	0	59
Hours	2	byte	0	23
Date	3	byte	1	31
Month	4	byte	1	12
Year	5,6	int	2000	2099

The "offset" specifies where the parameter is stored, relative to the xaddr returned by this function. The "Min." and "Max." columns specify the allowed ranges for each data type. C programmers should note that the time information is stored in paged memory, so standard C structure notation does not work, and the paged memory access functions such as StoreChar(), FetchChar(), and CmoveMany() must be used to access the data.

C: long **File_To_Memory**(void* dest_addr, uint dest_page, ulong numbytes, int file_id)
4th: **File_To_Memory** (dest_xaddr\d.numbytes\file_id -- d.numbtyes_read)

Reads from the specified open file to the specified destination in RAM or QED flash memory, and returns the number of bytes read as a 32-bit number. This is the fastest way to load the saved binary image of a compiled program from a CF card into memory (see the AUTOEXEC.QED example at the end of this document for an example of use). File_To_Memory reads numbytes characters starting at the current file position. If the specified destination region is not flash memory, this function simply calls File_Read to perform the data transfer. If the specified destination region is flash memory as described in the implementation details section below, then this function transfers the data into flash memory via the specified file's File_Contents RAM buffer (see the File_Contents glossary entry). The bytes are placed in memory starting at the specified dest_addr on the specified page. A standard procedure is for the calling program to test whether the returned number of bytes read equals the specified input parameter numbytes, and to infer that an error condition occurred if they are unequal. If an error occurs during the read operation, the error code is OR'd with any prior error condition, and the composite bitmapped error code can be retrieved by calling File_Error. An end of file condition can be separately tested for by calling the File_EOF function. See the glossary entries that start with "ERR_" for a complete list of error codes. Implementation details: If there was a prior call to the File_Ungetc function, the "ungot" character is moved to the destination in place of the first file character. To transfer an entire file, use the File_Size function to specify d.numbytes (but recall that C programmers can't call the File_Size function from within the parameter list of File_To_Memory; see the "Tips for C Programmers" section above). This function tests the specified dest_addr as well as the last address to be written; if either of these memory locations is in a known flash area, the flash transfer is used (note that RAM is still properly programmed by the flash transfer, but it takes longer). Otherwise, a RAM transfer is performed by File_Read. Known flash areas are on the QED Board page 7; page 0x0D; and pages 1 to 3 in the DOWNLOAD.MAP or pages 4 to 6 in the STANDARD.MAP.

Benchmark: Transfers from the ATA flash card to RAM take approximately 125 msec per Kbyte, and transfers from the ATA flash card to flash memory take approximately 420 msec per Kbyte. Transfers to flash disable interrupts for up to 25 msec per flash sector.

C: void **File_To_Page** (uint page)
4th: **File_To_Page** (page --)

Looks for a file named PAGEpp.BIN, where pp is a hexadecimal representation of the specified page. If the file is found, opens it in read mode, invokes File_To_Memory to copy the file contents to the specified page, and closes the file. For example, if the specified page is 4, then PAGE04.BIN will be opened and its contents copied to page 4. See the glossary entries for File_To_Memory and Page_To_File. Before calling this function, the file system must have been initialized. The file must be a binary file containing exactly 32 kbytes; otherwise an error will be reported and no information will be copied. This function prints a string to the serial output reporting the result. Output messages are:

File_To_Page completed copy of PAGEpp.BIN
File_To_Page couldn't open PAGEpp.BIN
File_To_Page error: wrong file size at PAGEpp.BIN
File_To_Page could not complete copy of PAGEpp.BIN

This routine is always callable by Process_File, even if the CF Card Forth headers have not been loaded onto the QED Board.

C: void **File_Type**(char* name_addr, uint name_page, int name_count)
4th: **File_Type** (name_xaddr\name_count --)

Types to the active serial port the contents of specified text file in the root directory, terminating when EOF (end of file) is reached or when a carriage return or period is typed from the terminal. Implements PauseOnKey (PAUSE.ON.KEY in Forth) functionality after each character is sent; see the glossary entry in the standard C or Forth glossary. The "pause on key" feature enables XON/XOFF flow control, and also allows the user to pause and resume the output stream by typing characters (such as the space character) from the terminal. For example, typing one space pauses the listing, and typing a second space resumes the listing. Typing a carriage return (or period) aborts the typing process and closes the file. This function uses the File_Contents buffer; see its glossary entry. Because it is typically used interactively, this function does not return an error flag; error conditions may be determined by calling File_Error after this function terminates. An error occurs if the specified filename does not exist, or if the maximum number of files are already open before this function is called. The filename is passed to the function as a character string whose first byte is at the specified name_addr on the specified page. Forth programmers should pass a string that has been "unpacked" using the COUNT function. C programmers must pass name_count = -1 to inform the function that this is a null-terminated string. C programmers calling this function from within a C program will typically pass the THIS_PAGE macro (defined in \include\mosaic\types.h) to specify the name_page parameter. See the glossary entry for File_Open to review the restrictions on the name.

CAUTION: If this function or other functions that accept name strings as parameters are typed interactively at the QED-Forth monitor using the Forth syntax, the dictionary pointer must point to modifiable RAM, not flash memory. This is because Forth emplaces strings typed interactively at the current dictionary pointer, and attempts to emplace data in flash without using the specially coded flash operations will cause a software crash. To avoid this problem, you can usually type the command

```
1 USE.PAGE
```

before an interactive session; this puts the dictionary pointer on page 1 RAM on a QED-Flash board that has 128K RAM and is operating in the STANDARD.MAP mode. If only 32K of RAM is present in the center socket, you could type

```
HEX BO00 0 DP X!
```

to locate the dictionary pointer in the processor's on-chip RAM.

EXAMPLE OF INTERACTIVE USE: Typical interactive use from the terminal is as follows (note that there is 1 blank space after the leading quote in the MYFILE.TXT string):

```
" MYFILE.TXT" COUNT File_Type
```

To pause and resume, use the space bar. To abort the listing, type a carriage return.

C: int **File_Ungetc**(char c, int file_id)
4th: **File_Ungetc** (char\file_id -- [char] or [err_eof])

If the specified file is open, and its file position is non-zero, and there was no previous pending "ungot" character, this function "pushes" the specified character onto the specified file stream so that it will be the pending next character accessed by any of the file read functions (File_Read, File_Gets, and File_Getc). If there is no error, this function returns the specified character; otherwise, ERR_EOF is returned.

Implementation Details: This function sets a char_pending flag (which is cleared by File_Seek, File_Set_Pos, and File_Read), saves the specified character in an internal structure, and decrements the file position pointer.

C: long **File_Write**(void* source_addr, uint source_page, ulong numbytes, int file_id)
4th: **File_Write** (source_xaddr\d.numbytes\file_id -- d.numbytes_written)

Writes from memory to the specified open file, and returns the number of bytes written as a 32-bit number. This function writes numbytes characters starting at the specified source_addr on the specified page. C programmers will typically specify a source_addr in common RAM with source_page = 0. The bytes are placed in the file starting at the current file position, except that the file position is forced to be at the end of the file (that is, file position = filesize) if the file was opened in A_MODE or APLUS_MODE. A standard procedure is for the calling program to test whether the returned number of bytes written equals the specified input parameter numbytes, and to infer that an error condition occurred if they are unequal. If an error occurs during the write operation, the error code is OR'd with any prior error condition, and the composite bitmapped error code can be retrieved by calling File_Error. An end of file condition can be separately tested for by calling the File_EOF function. See the glossary entries that start with "ERR_" for a complete list of error codes. Note that this function can be used to save a binary image of a compiled program into a file on the ATA flash card; to perform a software upgrade on another QED Board, the file can then be loaded into a board under the control of an AUTOEXEC.QED program that calls File_To_Memory.
Implementation Details: This function uses a modified buffered I/O scheme. Two 1-sector (512-byte) buffers are maintained. Each write operation is broken into 3 segments that are aligned to the sectors on the flash disk device: a starting partial sector, a number of whole sectors, and an ending partial sector. The starting partial sector (if present) is handled by pre-reading the full starting sector's contents from the disk into the starting sector buffer, and then overwriting the buffer with the partial sector contents from the source memory. The whole sectors (if present) are transferred directly from the source memory to the file on disk without the use of buffers; this speeds the process. The ending partial sector (if present) is handled in the same manner as the starting partial sector. Before writing data to either partial sector buffer, this File_Write function automatically flushes any buffers that require flushing; this allows the user to freely mix File_Read, File_Write, File_Seek, and File_Set_Pos operations without explicit calls to File_Flush.
Benchmark: Transfers to the ATA flash card take approximately 125 msec per Kbyte.

C: **FROM_CURRENT**
4th: **FROM_CURRENT** (-- n)

A constant that is passed as a parameter to the File_Seek function to indicate that the position is to be set with respect to the current file position. See File_Seek, FROM_START and FROM_END.

C: **FROM_END**
4th: **FROM_END** (-- n)

A constant that is passed as a parameter to the File_Seek function to indicate that the position is to be set with respect to the end of the file. See File_Seek, FROM_START and FROM_CURRENT.

C: **FROM_START**
4th: **FROM_START** (-- n)

A constant that is passed as a parameter to the File_Seek function to indicate that the position is to be set with respect to the start of the file. See File_Seek, FROM_END and FROM_CURRENT.

C: int **Hidden_Sectors**(void)
4th: **Hidden_Sectors** (-- n)

Returns the number of “hidden” sectors on the ATA drive. This parameter is stored in the FI structure, and is initialized by ATA_ID_Drive which is called by Init_File_IO and Init_File_System. If for some reason you need to read the master boot record in the hidden sector region at the physical start of the drive, specify a startsector of -{Hidden_Sectors} and call ATA_Read.

C: int **Init_File_Heap**(uint maxopen_files, uint file_bufsize, xaddr xheap_start, xaddr xheap_end)
4th: **Init_File_Heap** (maxopen_files\file_bufsize\xheap_start\xheap_end -- error)

Initializes a heap in the memory region defined by heap_start to heap_end to contain the arrays and buffers used by the file system. All arrays in the specified file heap are dimensioned, and all except the File_Contents buffer are zeroed. This function is typically not called directly by the programmer; rather, it is called via Init_File_IO or Init_File_System functions. See the glossary entry of Init_File_System for the default parameter values passed to this function. FI must point to a 300 byte block of available RAM before this function is called; this initialization is handled by Init_File_IO or Init_File_System. The parameter maxopen_files specifies the maximum number of files that can be open at one time, and file_bufsize indicates the size of a file contents buffer that is available to the user and is also used by the File_Copy and File_Type functions (see File_Contents). A minimum size of 512 bytes (i.e., 1 sector) is enforced on file_bufsize to ensure that File_Copy, File_To_Memory, and File_Type will work properly. The returned error parameter is -1 if the specified heap is too small to accommodate the required arrays and buffers. The heap size must be at least:

2.5K + [(1.6K + file_bufsize) * maxopen_files]

The heap may span multiple contiguous pages.

C: int **Init_File_IO**(uint maxopen_files, uint file_bufsize, xaddr xheap_start, xaddr xheap_end, xaddr xfat_info)
4th: **Init_File_IO** (maxopen_files\file_bufsize\xheap_start\xheap_end\xfat_info -- error)

Initializes all of the data structures needed by the ATA/FAT file system as described below. Note that the Set_CF_Module must be executed before calling this function. The module number must correspond to the module address set by the module address selection jumpers and the module port (see Table 1 above). See the glossary entry of Init_File_System for the default parameter values passed to this function. The xfat_info parameter must point to a 300 byte block of available RAM in either common memory or paged memory. The xheap_start and xheap_end parameters are the extended addresses that define the boundaries of the heap that contains the arrays and buffers used by the file system. This function calls Init_File_Heap to set up the heap, dimension the required arrays in the heap, and zero all arrays except the File_Contents array. The parameter maxopen_files specifies the maximum number of files that can be open at one time, and file_bufsize indicates the size of a file contents buffer that is available to the user (see File_Contents). A minimum size of 512 bytes (i.e., 1 sector) is enforced on file_bufsize to ensure that File_Copy, File_To_Memory, and File_Type (which use File_Contents) will work properly. The returned error parameter is nonzero if the specified heap is too small to accommodate the required arrays and buffers. The heap size must be at least:

2.5K + [(1.6K + file_bufsize) * maxopen_files]

The heap may span multiple contiguous pages. The returned error parameter is 0 if there are no errors. The error parameter is -1 if the specified heap is too small to accommodate the required arrays and buffers; the error parameter may also equal ERR_NON_DOS_DRIVE or ERR_ATA_READ or ERR_FAIL_ID_DRIVE (see their glossary entries).

Initialization Procedure: This function stores the specified xfat_info as the base address of the master parameter (see the FI glossary entry), sets the hardware control signals to their inactive states, calls Init_File_Heap (see its glossary entry) to configure the heap and dimension the arrays and buffers used by the file system, zeros the CF resource variable at the base of FI and, if there are no heap errors, calls ATA_ID_DRIVE to obtain necessary information about the ATA flash card. Then this function reads the boot sector on the flash disk. The relevant information obtained by these queries is stored in the appropriate data structures for use by the file system routines. This function or Init_File_System should be called any time (that is, after) a new ATA Flash card is placed into the CF Card socket.

C: int **Init_File_System**(void)
4th: **Init_File_System** (-- error)

This is the highest level initialization routine for the ATA/FAT file system. This function calls Init_File_IO with the following parameters:

max_open_files:	5;
file_bufsize:	1 Kbyte per file;
xheap_start:	0x4180 on page 0x03;
xheap_end:	0x7FFF on page 0x03;
xfat_info:	0x4000 on page 0x03;
module_num:	specified by Set_CF_Module

See the entry for Init_File_IO for a detailed description of these parameters and the initialization. Note that the Set_CF_Module must be executed before calling this function. The module number must correspond to the module address set by the module address selection jumpers and the module port (see Table 1 above). See the glossary entry for FI for a description of the fat_info structure. Note that this function uses the top 16 Kbytes of the page 3 RAM. User-available RAM on page 3 in the STANDARD.MAP thus ends at 0x4000. If you need this page 3 RAM for other purposes, call Init_File_IO with parameters of your choosing to set up an appropriate memory map.

Implementation Notes: If a CF Card is installed, this routine is automatically called by Do_Autoexec; see its glossary entry for details. Thus, one way to ensure that the CF Card is automatically initialized at startup is to include in your Priority Autostart routine an initialization statement that calls Set_CF_Module and then calls Do_Autoexec. Another way is to invoke Set_CF_Module, and then call Init_File_System or Init_File_IO directly from the Priority Autostart routine. This function or Init_File_IO should be called any time (that is, after) a new CF Card is placed into the socket.

C: void **Link_File_IO**(void)
4th: **Link_File_IO** (--)

A do-nothing function included for backward compatibility with prior kernel-resident versions of this software. The legacy kernel-resident version supports only the Memory Interface Board (MIB) hardware.

C: ulong **Max_File_Size**(void)
4th: **Max_File_Size** (-- d.numbytes)

Returns the maximum number of bytes per file for the flash card that was present when Init_File_System or Init_File_IO was performed. The returned value is the product of the implementation-defined maximum of 128 allowed clusters per file times the number of bytes per cluster for the installed flash disk. A "cluster" is a group of sectors (see SECTOR_SIZE) that is allocated at one time by the FAT file system; cluster size is determined when a disk is originally formatted. This function may be called anytime after the file system is initialized to determine the maximum file size that can be managed for a given ATA flash card. Zero is returned if the file system is not initialized. The implementation-defined maximum of 128 clusters per file supports a minimum file size of 64 Kbytes at 1 sector per cluster, which is typical of a 1 MByte card. The maximum file size increases to 512 Kbytes for a typical 10 Mbyte flash disk with 8 sectors/cluster.

C: **NO_ECHO_FILEID**
4th: **NO_ECHO_FILEID** (-- n)

A constant equal to -2 that is passed as an output_fileid (emit_fileid) to the Process_File function to suppress echoing of the input file's contents to the serial port. See Process_File.

C: int **Numsectors_Transferred**(void)
4th: **Numsectors_Transferred** (-- n)

Returns the number of sectors transferred by the most recent ATA command. This low-level function is typically not called by the programmer. Use care when interpreting the returned value, and note that the file system operations such as File_Read and File_Write may execute multiple ATA commands.

C: void **Page_To_File** (uint page)
4th: **Page_To_File** (page --)

Creates a file named PAGEpp.BIN, where pp is a hexadecimal representation of the specified page, invokes File_Write to copy the contents of the specified page to the file, and closes the file. If the file already exists, it is overwritten. For example, if the specified page is 4, then PAGE04.BIN will be created and the 32 kbyte contents of page 4 will be written to the file. its contents copied to page 4. See the glossary entries for File_Write and File_To_Page. The file system must have been initialized prior to calling this function. This function prints a string to the serial output reporting the result. Output messages are:

Page_To_File completed copy of PAGEpp.BIN
Page_To_File couldn't open PAGEpp.BIN
Page_To_File could not complete copy of PAGEpp.BIN

This routine is always callable by Process_File, even if the CF Card Forth headers have not been loaded onto the QED Board.

C: int **PCC_Present**(void)
4th: **PCC_Present** (-- flag)

Returns a true (-1) flag if a CF Card (formerly called a PC Card, or PCC) is installed; otherwise returns a false (0) flag.

Implementation details: Reads the card status bits. A true flag is returned if the active-low card detection bits 0 and 1 are low, the READY signal is active high, and the /WAIT signal is inactive high.

C: int **Process_Autoexec**(int autoexec_echo)
4th: **Process_Autoexec** (autoexec_echo -- error)

This function checks for a file named AUTOEXEC.QED in the root directory of the flash card and, if the file is present, calls Process_File to interpret and execute its contents. The autoexec_echo parameter is a boolean flag that specifies whether the contents of AUTOEXEC.QED are echoed to the active serial port by Process_File. Typically, autoexec_echo is set true (nonzero) only during debugging or diagnostic sessions so that echoed source code is visible via the serial port, and is false (zero) in an actual application. If the AUTOEXEC.QED file is not found, the returned error flag is ERR_FILE_DOES_NOT_EXIST. This function is called (with autoexec_echo = FALSE) by Do_Autoexec if a CF card is present. This automated file processing capability facilitates in-field software upgrades. See the AUTOEXEC.QED examples at the end of this document.

Implementation Notes: If the AUTOEXEC.QED file is present, it is opened as a read-only file with file_id = 0, Process_File is called to interpret its contents, and then the file is closed. As mentioned above, if this function is called by Do_Autoexec, the echo is suppressed. If you need to monitor the echoed output from the processing of the AUTOEXEC.QED file via the serial port, you can explicitly accomplish this by placing the following Forth command at the top of the AUTOEXEC.QED file:

```
CFA.FOR  EMIT1  UEMIT  X!
```

To direct the output to the secondary serial port, simply substitute EMIT2 for EMIT1 (assuming that the secondary serial port has been initialized). Note that the programmer can explicitly call this function from an autostart or priority autostart routine after calling Init_File_IO or Init_File_System. If the contents of AUTOEXEC.QED cause a crash or abort, then upon the next startup during which the CF Card is not present, Put_Default_Serial will be called to vector the serial input primitives Key and AskKey (KEY and ?KEY in Forth) back to their standard serial port versions if they previously pointed to File_Key/File_Ask_Key. In other words, to recover from a buggy AUTOEXEC.QED file, simply remove the CF Card from its socket.

C: void **Process_File**(int input_file_id, int output_file_id)
4th: **Process_File** (input_fileid\output_fileid --)

This powerful function processes (that is, interprets and executes) the contents of the ASCII text file specified by input_fileid, and optionally directs the serial output (including echoing of the input contents) to the file indicated by output_file_id. This function is called by Process_Autoexec (see its glossary entry) to automatically check for and process a file named AUTOEXEC.QED. This function can be explicitly called by the programmer from within a function or from within a file that is itself being processed by Process_File. In other words, file processing can be nested. The specified input and output files must be in the root directory. This function assumes that Init_File_IO or Init_File_System has already executed to initialize all the required data structures. If the specified input_fileid parameter is non-negative, this function revectors the serial input primitives to point to File_Key and File_Ask_Key; if the input_fileid is negative, the serial input primitives are not changed. If the specified output_fileid parameter is non-negative or equals NO_ECHO_FILEID, this function revectors the serial output primitive to point to File_Emit. If the output_fileid is a negative value other than NO_ECHO_FILEID, the serial output primitive is not changed. (File_Emit suppresses serial output if the output_fileid equals NO_ECHO_FILEID). Note that execution speed is greatly enhanced if the output is not directed to a file. After revectoring the serial primitives, this function sets the serial access mode to RELEASE_ALWAYS, installs File_Abort_Action as the abort handler, and calls File_Interpreter to process the specified input file. Note that the specified input file and output file must be open before this function is called; see File_Open. The File_Interpreter function processes all of the contents of the file starting at the current file position, returning when the end of file is encountered. Then Process_File restores all of the serial primitives, the serial access mode, and the abort handler to their prior behaviors. Note that this function does not close the input or output files; the calling program should call File_Close to take care of this. This function gets and releases the CF resource variable to ensure uncontested access to the CF Card in multitasking applications. The file contents must be executable QED-Forth code (note that the *.txt output file created by the C compiler contains executable QED-Forth code, so a C program can be installed into QED flash memory using this method). The file will typically contain a File_To_Memory command to rapidly load pre-compiled binary code and data into modifiable RAM or flash. Another common entry in the file may be an AUTOSTART or PRIORITY.AUTOSTART statement. See the glossary entries for AUTOSTART and PRIORITY.AUTOSTART in the QED Board Glossary documentation.

To Use: If you want to interpret the executable (Forth) source code in a file, or if you want to compile the results of a *.txt file created by the C compiler, use File_Open to open the file and obtain a valid fileid, then pass it as input_fileid to process_file. If you want standard echoing of the source code to the active serial port, specify output_fileid = -1. If you want no echo, specify output_fileid = NO_ECHO_FILEID (= -2). If you want the echoed source to be stored into a file (it cannot be the input_fileid), use File_Open to open the file, and pass its output_fileid to Process_File (this will significantly slow the processing). When Process_File completes (returns), you should call File_Close to close any files that you opened.

Example of Use: See the AUTOEXEC.QED example section of this document.

Implementation Notes: This function simply passes the extended code field address (xcfa) of File_Interpreter to the Redirect function. Process_File sets SERIAL_ACCESS equal to RELEASE_ALWAYS during file processing, then restores SERIAL_ACCESS to its prior value at file end. If desired, this serial access mode may be overridden by inserting the command:

```
SERIAL_ACCESS = RELEASE_NEVER;  // if programming in C
```

or:

```
RELEASE.NEVER  SERIAL.ACCESS  !        \ if programming in Forth.
```

at the top of the input file to be processed.

Caution: The RELEASE_AFTER_LINE access mode is not supported.

C: void **Put_Default_Serial**(void)
4th: **Put_Default_Serial** (--)

If any of the key or emit serial primitives have been revectored to the file versions (File_Key, File_Ask_Key, or File_Emit), this function restores the vectors in the current user area to point to the default serial channel that is enabled at startup. This function is serial-aware: it knows whether serial1 or serial2 is the default serial port on the QED Board at startup. This low level utility function is called by Init_File_IO and File_Abort_Action; it is typically not called directly by the programmer.

C: int **Read_CF_Module**
4th: **Read_CF_Module** (-- module_num)

Returns the value that was set using the most recent call to Set_CF_Module; see its glossary entry.

C: void **Redirect**(int input_file_id, int output_file_id, void(*fn)(), uint fn_page)
4th: void **Redirect** (input_file_id\output_file_id\fn_xcfa --)

This powerful function is capable of revectoring the serial input and serial output primitives and then calling the function specified by the fn pointer. This enables the specified called function to accept input from a file specified by input_file_id and/or place output into a file specified by output_file_id. The specified input and output files must be in the root directory. This function assumes that Init_File_IO or Init_File_System has already executed to initialize all the required data structures. If the specified input_fileid parameter is non-negative, this function revectors the serial input primitives to point to File_Key and File_Ask_Key; if the input_fileid is negative, the serial input primitives are not changed. If the specified output_fileid parameter is non-negative or equals NO_ECHO_FILEID, this function revectors the serial output primitive to point to File_Emit. If the output_fileid is a negative value other than NO_ECHO_FILEID, the serial output primitive is not changed. (File_Emit suppresses serial output if the output_fileid equals NO_ECHO_FILEID). After revectoring the serial primitives, this function sets the serial access mode to RELEASE_ALWAYS, installs File_Abort_Action as the abort handler, and calls the function whose extended code field address is fn_xcfa (in Forth), or whose function pointer is specified by void(*fn)() on fn_page (in C). Recall that in C, function pointers are passed by referencing the function name without its parentheses, and the page is typically referenced by the macro THIS_PAGE (defined in \include\mosaic\types.h) as long as both the calling and callee functions are in the same C source code file . Note that the specified input file and output file must be open before this function is called; see File_Open. The file input and output occurs at the current file positions of the input and output files, respectively. When coding an action function that accepts input from a file, it is important that the function terminate gracefully when AskKey (?KEY in Forth) or File_Ask_Key returns 0; this indicates that the end of file has been reached. After the called action function terminates, Redirect restores all of the serial primitives, the serial access mode, and the abort handler to their prior behaviors. Note that this function does not close the input or output files; the calling program should call File_Close to take care of this. This function gets and releases the CF resource variable to ensure uncontested access to the card in multitasking applications. This function is called by Process_File which passes the xcfa of File_Interpreter as the called action function. Consult the glossary entry for Process_File to see how Redirect can be used to implement powerful capabilities.

C: uint **Report_File_Errors**(int file_id, char* string_addr, uint string_page, uint maxbytes)
4th: **Report_File_Errors** (file_id\string_xaddr\maxbytes -- count)

Stores a string starting at the specified string_addr that describes the file errors for the open file with the specified valid file_id. Returns the number of characters placed in the string (not including the terminating null); if no errors occurred, the returned count equals zero. Limits the number of bytes stored in the string to maxbytes; however, maxbytes+1 bytes must be available in memory to accommodate the string plus the terminating null byte. If multiple errors are present, the messages are concatenated, separated by 1 space per message. Messages are 13 to 26 bytes each. After this function executes, the contents of the string may be printed to display user-friendly error diagnostics. C programmers will typically specify a string_addr in common RAM with string_page = 0. See the glossary entries that begin with "Err_" for a list of bit-mapped error codes. Note that this function does not clear the file_errors flag; use Clear_File_Error or File_Rewind to do that. A related function is Report_File_Open_Errors, which is useful to diagnose errors that occur while a file is being opened, or that occur when an invalid file_id is supplied to a function. See also File_Error.
CAUTION: C programmers should note that printf() on the QED Board limits the size of the printed string to 80 characters, so maxbytes should be no greater than 79.

C: uint **Report_File_Open_Errors**(char* string_addr, uint string_page, uint maxbytes)
4th: **Report_File_Open_Errors** (string_xaddr\maxbytes -- count)

Stores a string starting at the specified string_addr that describes the file errors that occurred during the most recent File_Open operation. Returns the number of characters placed in the string (not including the terminating null); if no errors occurred, the returned count equals zero. Limits the number of bytes stored in the string to maxbytes; however, maxbytes+1 bytes must be available in memory to accommodate the string plus the terminating null byte. If multiple errors are present, the messages are concatenated, separated by 1 space per message. Messages are 13 to 26 bytes each. After this function executes, the contents of the string may be printed to display user-friendly error diagnostics. C programmers will typically specify a string_addr in common RAM with string_page = 0. See the glossary entries that begin with "Err_" for a list of bit-mapped error codes. Note that this function does not clear the file_errors flag; use Clear_File_Error or File_Rewind to do that. Report_File_Open_Errors is useful to diagnose errors that occur while a file is being opened, or that occur when an invalid file_id is supplied to a function. A related function is Report_File_Errors, which reports errors related to open files that have a valid file_id. See also File_Error.
CAUTION: C programmers should note that the printf() implementation on the QED Board limits the size of the printed string to decimal 80 characters, so maxbytes should be no greater than 79.

C: **RPLUS_MODE**
4th: **RPLUS_MODE** (-- n)

A constant that is passed as a file-access privilege parameter to the File_Open function to indicate that the file may be read or written. If RPLUS_MODE is specified, an error will be returned by File_Open if the file does not already exist. See also R_MODE, W_MODE, A_MODE, WPLUS_MODE, and APLUS_MODE.

C: **R_MODE**
4th: **R_MODE** (-- n)

A constant that is passed as a file-access privilege parameter to the File_Open function to indicate that the file is read-only. If R_MODE is specified, an error will be returned by File_Open if the file does not already exist. Once the file is open, writes are not allowed to the file, and the directory entry will not be updated when the file is closed. Note that File_Open will automatically set the access mode of a file to R_MODE (read only) if the directory entry for the file (initialized when the file was originally created) specifies read-only access. See also W_MODE, A_MODE, RPLUS_MODE, WPLUS_MODE, and APLUS_MODE.

C: **SECTOR_SIZE**
4th: **SECTOR_SIZE** (-- n)
A constant that returns 512, which is the sector size (also called the block size) of the ATA drive.

C: int **Set_Filesize**(ulong needed_filesize, int file_id)
4th: **Set_Filesize** (d.needed_filesize\file_id -- error)
A low-level utility that changes the size of the specified file to the indicated size, allocating or de-allocating clusters from the File Allocation Table (FAT) as needed. Returns a nonzero error if a cluster cannot be added; otherwise, writes the new filesize to the mirror copy of the directory entry in RAM, and marks it for update upon the next File_Flush operation. The maximum filesize is limited to Max_File_Size; see its glossary entry for details. The minimum number of clusters per file is 1, even if the filesize=0. This low-level utility should not be needed in most applications, as the File_Open and File_Write functions truncate and add to the file size as needed.

C: void **Set_CF_Module**(int module_num)
4th: **Set_CF_Module** (module_num --)
Sets the module number specified by the user. This function must be executed before calling any of the initialization functions (Init_File_IO, Init_File_System, or Do_Autoexec). The module number parameter must correspond to the module address set by the module address selection jumpers and the module port (see Table 1 above).

C: uint **Volume_Label**(char* string_addr, uint string_page)
4th: **Volume_Label** (string_xaddr -- count)
Moves the volume label (that is, the assigned disk name) that was read from the boot sector during initialization, as a null-terminated string to the specified string_addr, and returns the count of the string. The count does not include the terminating null. The string is left-justified and space-padded to decimal 11 bytes, and the buffer at string_addr must be at least 12 bytes long. This function returns 0 if the file system is not initialized (see Init_File_System). C programmers will typically specify a string_addr in common RAM with string_page = 0.

C: **WPLUS_MODE**
4th: **WPLUS_MODE** (-- n)
A constant that is passed as a file-access privilege parameter to the File_Open function to indicate that the file may be read or written. If WPLUS_MODE is specified, File_Open will truncate the specified file to zero size if it already exists. See also R_MODE, W_MODE, A_MODE, RPLUS_MODE, and APLUS_MODE.

C: **W_MODE**
4th: **W_MODE** (-- n)
A constant that is passed as a file-access privilege parameter to the File_Open function to indicate that the file is write-only. If W_MODE is specified, File_Open will truncate the specified file to zero size if it already exists, and reads of the file will not be allowed. See also R_MODE, A_MODE, RPLUS_MODE, WPLUS_MODE, and APLUS_MODE.

Upgrade Notice for Prior Users of the Memory Interface Board

The predecessor to the CF Module was the Memory Interface Board (MIB) which hosts a PCMCIA PC Card using software that is built into the QED Kernel Flash. This software was linked (made accessible to interactive calls) by calling Link_File_IO, and set up its data structures on dedicated RAM resident on the MIB itself. The "Do_Autoexec" function was embedded as part of the kernel's startup procedure, and so did not have to be explicitly included in a Priority Autostart function. Also, there was no "module number" to keep track of. The C library calls were declared and defined in the \fabius\include\mosaic\ataflash directory, in files ataflash.c and ataflash.h. These files cannot be used with the CF Module.

Programmers should follow the instructions presented in the section titled "How to Install the CF Software" presented above.

While most of the MIB/ATA software is unchanged from the programmer's point of view, the following changes have been made:

These functions are no longer needed and have been deleted:

- Arm_Autoexec
- Disarm_Autoexec
- Link_File_IO
- PCC_Offset_To_Xaddr
- PCC_bank
- Slow_Fetch
- Slow_Store
- Sectors_Per_Track

The variable

- xfat_info_save

has been deleted. Its value can be accessed using the FI function.

These functions have been added:

- Hidden_Sectors
- Do_Autoexec
- Read_CF_Module
- Set_CF_Module

The Link_File_IO function is no longer needed and should never be called. If called, it would load the headers for the obsolete MIB software in the QED V4.xx kernel. Note that the Arm_Autoexec and Disarm_Autoexec functions are no longer implemented. Moreover, because the CF Card Software is not a built-in part of the QED-Forth kernel, the allowed contents of the AUTOSTART.QED file may be restricted, depending on whether the CF Card Software Forth headers in library.4th are loaded into memory.

Init_File_System now locates the required data structures in the top 16 Kbytes of page 3, instead of in the RAM located on the MIB itself.

Sample FILEDEMO.C File

This demonstration program is included in the CF Card software distribution. For instructions on how to use the demo, see the comments at the end of this code listing. The distribution also contains a parallel file named FILEDEMO.4th for Forth programmers.

```
// FILEDEMO.C

// See the "HOW TO RUN THE DEMO" section at the bottom of this file for instructions.
// Be sure to define CF_MODULE_NUM so that it matches the hardware settings!

// This file demonstrates how to use some of the common file manipulation functions
// in the CF Card Software Package.
// Both Forth (*.4th) and C (*.C) language versions of this FILEDEMO file exist.

// The Demo function creates a file named "TESTER.QED" and writes to it a
// sequence of bytes. It uses WPLUS_MODE to open the file, which means that if
// the file already exists it is truncated to zero size, and that the file is
// readable and writeable.  Using the pre-dimensioned File_Contents buffer,
// we write increasing values from 0 to 255 in each of the first 256 bytes.
// Then we store a message string in the file; it says:
//  " Alphabet in forward order: "
// We then selectively read parts of the file using a buffer in
// common RAM, printing the message followed by an upper case alphabetic listing.
// Finally, we close the file.
// This File I/O code demonstrates the following:
// How to open and close a file;
// How to use File_Set_Pos and File_Tell_Pos to control random file access;
// How to use the pre-dimensioned File_Contents buffer in heap as a scratchpad;
// How to use another buffer (we call it show_buffer) as a scratchpad;
// How to use File_Write, File_Puts and File_Put_CRLF to put content into a file;
// How to use File_Gets to fetch contents out of a file.

// For clarity, only simple error handling is performed in this code.
// As an exercise for yourself, you can add more robust error handling.
// In most cases, you can either report the returned error codes,
// or simply check that the number of characters read or written
// (as returned by the function) matches what you expect.
// To report one overall code, you can logically OR the error codes of the
// individual file operations.

// Copyright 2002 Mosaic Industries, Inc.  All Rights Reserved.
//  Disclaimer: THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, WITHOUT ANY
//  WARRANTIES OR REPRESENTATIONS EXPRESS OR IMPLIED, INCLUDING, BUT NOT
//  LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS
//  FOR A PARTICULAR PURPOSE.

// ****

// We recommend that you type COLD before loading the compiled *.txt file.

#include <\mosaic\allqed.h>     // include all of the qed and C utilities
#include "library.c"    // this is a kernel extension file;
// assume it's in the source code directory; if not, edit the file specification


// ****** Constants, Variables and Buffers ************

#define CF_MODULE_NUM  0    // MUST match hardware jumper settings! Inits cf_module
#define NUM_ASCENDING_BYTES  256
#define SHOW_BUF_SIZE       60          // bigger than we need
#define LETTERS_PER_ALPHABET ('Z' + 1 - 'A')

#define BYTES_PER_PAGE      (1024 * 32)    // 1 page= 32 Kb; see Make_Image_File

static int demo_file_id;    // used to save the file_id
static long string1_start;  // holds 32bit offset of the message string in file
```

```
static long numbytes_accessed; // value returned by file_read or file_write; not checked
static int f_error;         // holds logical or of error codes; not checked

static char show_buffer[SHOW_BUF_SIZE];  // allocate buffer for Show_File below

// ********* Demo code showing how to use the file I/O functions ***********

_Q void Init_Contents_Buffer( int file_id )
// writes an ascending pattern starting at zero into the File_Contents buffer
// note that we can't embed the call to File_Contents in the parameter list of
//    StoreChar, as nested calls of _forth functions are not permitted.
{   int i;
    xaddr buf_xaddress;
    for( i=0; i < NUM_ASCENDING_BYTES; i++ )
    {   buf_xaddress = File_Contents( i, file_id);
        StoreChar( (char) i, buf_xaddress);
    }
}

_Q void Make_File( void )
// Opens a file named TESTER.QED, transfers an ascending sequence of 256
// bytes to the file, followed by a message string.
// Initializes the global variables demo_file_id and string1_start
{   xaddr source_xaddr;
    char* string_ptr = "Alphabet in forward order: ";
    int string_size = strlen(string_ptr);
    int demo_file_id = File_Open("TESTER.QED",THIS_PAGE,-1,WPLUS_MODE);
    if(demo_file_id < 0)        // negative file_id means file_open failed
        printf("\nFile open failed!\n");
    else                        // if no error, continue...
    {   Init_Contents_Buffer(demo_file_id);     // put ascending byte pattern
        source_xaddr = File_Contents(0, demo_file_id);
        numbytes_accessed =     // write buffer to file; we discard numbytes written
            File_Write( XADDR_TO_ADDR(source_xaddr),XADDR_TO_PAGE(source_xaddr),
                    (long) NUM_ASCENDING_BYTES, demo_file_id);

        string1_start = File_Tell_Pos(demo_file_id);  // save string offset
        numbytes_accessed =         // write string -> file; ignore numchars written
            File_Puts( string_ptr, THIS_PAGE, string_size, demo_file_id);

        f_error |= File_Put_CRLF(demo_file_id);    // mark line end; or error code
    }
}

_Q void Show_File( void )
// this function prints some of the contents of the file whose id
// is in the demo_file_id variable.
// First we print the message that starts at d.offset = string1_start,
// and then we print the upper case alphabet which starts at
// an offset equal to ASCII A (promoted to 32 bits).
{   int numchars_read;
    f_error |= File_Set_Pos( demo_file_id, string1_start);  // get "ascending" label
    numchars_read = File_Gets(show_buffer,THIS_PAGE,SHOW_BUF_SIZE,demo_file_id);
    show_buffer[ numchars_read ] = 0;   // put terminating null
    printf("\n%s",show_buffer);             // type string1 (ends in CRLF)
    f_error |= File_Set_Pos(demo_file_id,(long) 'A');
    numchars_read =                     // ends when 26 letters are read
          File_Gets(show_buffer,THIS_PAGE,LETTERS_PER_ALPHABET,demo_file_id);
    show_buffer[ numchars_read ] = 0;   // put terminating null
    printf("%s\n",show_buffer);         // type ascending alphabet, add newline
       // as an excercise: add a line showing the lower case alphabet!
}
```

```
_Q void Demo( void )    // this is the demonstration function
// Initializes file system, Opens and initializes TESTER.QED file,
// reports selected contents, then closes the file.
{   f_error = 0;
    printf("\nInitializing File System...");
    Set_CF_Module( CF_MODULE_NUM );  // must init before calling Init_File_System
    if(Init_File_System())      // if error...
        printf("\nCouldn't initialize file system!\n");
    else                        // if no error...
    {   printf("\nCreating and initializing the TESTER.QED file...\n");
        Make_File();
        Show_File();          // print announcement and alphabet
        File_Close(demo_file_id);   // ignore error flag
    }
}


void main( void )   // sets up automated file system init and calls the Demo function
// this top-level function can be declared as a priority.autostart routine.
{  Set_CF_Module( CF_MODULE_NUM );   // must init before calling Init_File_System
//   Do_Autoexec();  // un-comment this if you want to use the autoexec.qed capability
   Demo();
}

/* ************* HOW TO RUN THE DEMO USING C ********************

Make sure that your QED Board or Panel-Touch Controller is communicating
with your PC, that the Wildcard Carrier Board is mounted, and that the
CF Module is installed.  Check that the module port and jumper settings
match the CF_MODULE_NUM defined in this program.  Plug a formatted
CF Card into the socket on the CF Module.

Follow these steps:

1. Drag the LIBRARY.C and LIBRARY.H files that contain the CF Card software drivers
   to the same directory as your source code (or, if you choose to put these files
   in another location, edit the #include directives at the top of this file
   accordingly).
2. Use QEDTERM to send the INSTALL.TXT file from the CF Card software driver
   to the QED Board.  This loads the drivers onto the board.  You need not
   repeat this step when recompiling your C application program.
3. Compile this demo program by opening the file from WinEdit
   and pressing the hammer (make) icon.
4. Use QEDTERM to send the resulting FILEDEMO.TXT file to the QED Board.
5. Type
        MAIN
   from your terminal to run the demo program from this file.
   You should see the following:
        Initializing File System...
        Creating and initializing the TESTER.QED file...

        Alphabet in forward order:
        ABCDEFGHIJKLMNOPQRSTUVWXYZ

6. You can optionally type
        CFA.FOR  MAIN  PRIORITY.AUTOSTART
   from your terminal to automatically run the program after every startup.

   To remove the autostart, type NO.AUTOSTART from your terminal.

*/
```

Sample FILEDEMO.4th File

This demonstration program is included in the CF Card Software distribution. For instructions on how to use the demo, see the comments at the end of this code listing. The distribution also contains a parallel file named FILEDEMO.C for C programmers.

```
\ FILEDEMO.4TH

\ See the "HOW TO RUN THE DEMO" section at the bottom of this file for instructions.
\ Be sure to define CF_MODULE_NUM so that it matches the hardware settings!

\ This file demonstrates how to use some of the common file manipulation functions
\ in the CF Card Software Package.
\ Both Forth (*.4th) and C (*.C) language versions of this FILEDEMO file exist.

\ The Demo function creates a file named "TESTER.QED" and writes to it a
\ sequence of bytes. It uses WPLUS_MODE to open the file, which means that if
\ the file already exists it is truncated to zero size, and that the file is
\ readable and writeable.  Using the pre-dimensioned File_Contents buffer,
\ we write increasing values from 0 to 255 in each of the first 256 bytes.
\ Then we store a message string in the file; it says:
\   " Alphabet in forward order: "
\ We then selectively read parts of the file using a buffer in
\ common RAM, printing the message followed by an upper case alphabetic listing.
\ Finally, we close the file.
\ This File I/O code demonstrates the following:
\ How to open and close a file;
\ How to use File_Set_Pos and File_Tell_Pos to control random file access;
\ How to use the pre-dimensioned File_Contents buffer in heap as a scratchpad;
\ How to use another buffer (we call it show_buffer) as a scratchpad;
\ How to use File_Write, File_Puts and File_Put_CRLF to put content into a file;
\ How to use File_Gets to fetch contents out of a file.

\ For clarity, very little error handling is performed in this code.
\ As an exercise for yourself, you can add more robust error handling
\ to the code.  In most cases, you can either report the returned error codes,
\ or simply check that the number of characters read or written
\ (as returned by the function) matches what you expect.
\ To report one overall code, you can logically OR the error codes of the
\ individual file operations.

\ Copyright 2002 Mosaic Industries, Inc.  All Rights Reserved.
\  Disclaimer: THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, WITHOUT ANY
\  WARRANTIES OR REPRESENTATIONS EXPRESS OR IMPLIED, INCLUDING, BUT NOT
\  LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS
\  FOR A PARTICULAR PURPOSE.

\ ****

\ You must load install.txt and LIBRARY.4TH containing the CF Card Software Drivers
\ before loading this file. Review the memory map statements at the top of the files
\ to ensure that there are no memory conflicts.


DECIMAL         \ interpret all numbers using decimal base
9 WIDTH !       \ avoid non-unique names

WHICH.MAP  0=
      IFTRUE  4 PAGE.TO.RAM      \ if in standard.map...
                                 5 PAGE.TO.RAM
                                 6 PAGE.TO.RAM
           DOWNLOAD.MAP
      ENDIFTRUE


ANEW DEMO_CODE  \ a marker to simplify reloading of this code
```

```
\ ****** Constants, Variables and Buffers ************

0   CONSTANT CF_MODULE_NUM      \ MUST match hardware jumper settings! Inits cf_module
256 CONSTANT NUM_ASCENDING_BYTES
60  CONSTANT SHOW_BUF_SIZE          \ bigger than we need
ASCII Z 1+ ASCII A - CONSTANT LETTERS_PER_ALPHABET

1024 32 * CONSTANT BYTES_PER_PAGE   \ each page is 32 Kbytes; see Make_Image_File

INTEGER: demo_file_id       \ used to save the file_id
DOUBLE:  string1_start      \ holds 32bit offset of the message string in file

SHOW_BUF_SIZE V.INSTANCE: show_buffer   \ allocate RAM buffer for Show_File below

\ ********* Demo code showing how to use the file I/O functions ***********

: Init_Contents_Buffer  ( file_id -- )
\ writes an ascending pattern starting at zero into the File_Contents buffer
   LOCALS{ &file_id }
   NUM_ASCENDING_BYTES 0
   DO  I
      I &file_id File_Contents C!
   LOOP
;

: Make_File ( -- )
\ Opens a file named TESTER.QED, transfers an ascending sequence of 256
\ bytes to the file, followed by a message string.
\ Initializes the global self-fetching variables demo_file_id and string1_start
   " TESTER.QED" COUNT WPLUS_MODE File_Open        ( -- file_id)
   DUP TO demo_file_id         \ save in global variable
   0<                          \ negative file_id means file_open failed
   IF CR ." File open failed!" CR
   ELSE                        \ if no error, continue...
      demo_file_id Init_Contents_Buffer   \ put ascending byte pattern
      0 demo_file_id File_Contents
      NUM_ASCENDING_BYTES U>D
      demo_file_id           ( xsrc_addr\d.numbytes\file_id -- )
      File_Write 2DROP        \ write buffer to file; ignore d.numbytes_written

      demo_file_id File_Tell_Pos TO string1_start     \ save string offset
      " Alphabet in forward order: " COUNT
      demo_file_id               ( xsource\max_chars\fileid -- )
      File_Puts DROP              \ write string -> file; ignore numchars_written
      demo_file_id File_Put_CRLF DROP     \ mark line end; drop error code
   ENDIF
;

: Show_File ( -- )
\ this function prints some of the contents of the file whose id
\ is in the demo_file_id variable.
\ First we print the message that starts at d.offset = string1_start,
\ and then we print the upper case alphabet which starts at
\ an offset equal to ASCII A (promoted to 32 bits).
   demo_file_id string1_start  File_Set_Pos    \ get "ascending" label
   DROP                                \ drop error flag
   show_buffer SHOW_BUF_SIZE demo_file_id  ( xdest\bufsize\fileid -- )
   File_Gets          ( -- numchars_read) \ terminates at linefeed character
   show_buffer ROT CR TYPE                \ type string1 (ends in CRLF)
   demo_file_id ASCII A U>D File_Set_Pos
   DROP                                \ drop error flag
   show_buffer LETTERS_PER_ALPHABET demo_file_id   ( xdest\bufsize\fileid -- )
   File_Gets             ( -- numchars_read ) \ ends when 26 letters are read
   show_buffer ROT TYPE    \ type ascending alphabet
   CR                    \ we need to explicitly add carriage return
      \ as an excercise: add a line showing the lower case alphabet!
   ;
```

```
: Demo  ( -- )  \ this is the demonstration function
\ Opens TESTER.QED, initializes it, and reports selected contents, then closes the file.
\ The optional first line relinks the names up to this point in this file.
\   [ LATEST ] 2LITERAL VFORTH X!  \ this line is optional; it makes names accessible
    CR ." Initializing File System..."
    CF_MODULE_NUM Set_CF_Module        \ must init before calling Init_File_System
    Init_File_System        ( error -- )
    IF CR ." Couldn't initialize file system!" CR
    ELSE
       CR ." Creating and initializing the TESTER.QED file..." CR
       Make_File
       Show_File
       demo_file_id File_Close DROP    \ drop error flag
    ENDIF
    ;

: Startup   ( -- )     \ calls the Demo function
\ this top-level function can be declared as a priority.autostart routine.
    CF_MODULE_NUM Set_CF_Module        \ must init before calling Init_File_System
\           Do_Autoexec    \ un-comment this if you want to use the autoexec.qed capability
    Demo
    ;

\ We can also set up a PRIORITY.AUTOSTART routine.
\ We'll make the Demo Program run on each subsequent startup:
\ CFA.FOR Startup PRIORITY.AUTOSTART

\ To remove, type NO.AUTOSTART from your terminal.

4 PAGE.TO.FLASH
5 PAGE.TO.FLASH
6 PAGE.TO.FLASH
STANDARD.MAP
SAVE            \ even after a cold restart, you can still call the demo function by typing  RESTORE

0 1 DP X!       \ this puts dictionary pointer in RAM so that interactive use
              \ of strings (as in file_open, file_type, etc.) works!

\  ************* HOW TO RUN THE DEMO USING FORTH ********************
\ Make sure that your QED Board or Panel-Touch Controller is communicating
\ with your PC, that the Wildcard Carrier Board is mounted, and that the
\ CF Module is installed.  Check that the module port and jumper settings
\ match the CF_MODULE_NUM defined in this program.  Plug a formatted
\ CF Card into the socket on the CF Module.

\ Follow these steps:
\ 1. Use QEDTERM to send to the QED Board
\    the INSTALL.TXT file and the LIBRARY.4TH file that contain the CF Card Software
\    drivers.  The former file need not be re-sent each time the application program
\    is downloaded.  The LIBRAY.4TH file must be sent with each download.
\ 2. Use QEDTERM to send this file to the QED Board.
\ 3. Type
\         Startup
\    from your terminal to run the demo program from this file.
\    You should see the following:
\         Initializing File System...
\         Creating and initializing the TESTER.QED file...
\
\         Alphabet in forward order:
\         ABCDEFGHIJKLMNOPQRSTUVWXYZ
\
\ 4. To automatically run the program upon each powerup and restart, type from the  terminal:
\         CFA.FOR STARTUP PRIORITY.AUTOSTART
\    This places the autostart pattern near the top of page 4 flash.
\    To remove the autostart, simply type
\         NO.AUTOSTART
\     from your terminal.  If you remove the autostart, and
\     if power has cycled and you want to re-run the demo, you will have to type
\         RESTORE
\    before typing Startup.  This relinks the names so that the operating
```

\ system can interactively recognize interactively typed names.

CF Wildcard Hardware Schematic