Foenix/MCP

A Simple, Portable Operating System
for the Foenix Line of Computers
version 1.0-alpha.x

Overview

Warning

Please do not attempt to install a ENCOM SHV series digitizing laser on any Foenix computer running the Foenix/MCP operating system. Early versions of the MCP displayed erratic behavior when given access to a digitizing laser. Neither the author of Foenix/MCP nor ENCOM will be responsible for any unexpected behavior experienced by users.

Copyright Information

Foenix/MCP and all code except for the FatFS file system library are published under the BSD 3 Clause License. Please see the source code for the license terms.

The Foenix/MCP file system is provided by the FatFS file system, which is covered under its own license. For information about the author of FatFS and its license terms, please see the Foenix/MCP source code.

Devices

Devices on the Foenix computers fall into one of two main categories: channel devices, and block devices.

Channel Devices

Channel devices are predominantly sequential, byte oriented devices. They are essentially byte streams. A program can read or write a series of bytes from or to the device. A channel can have the notion of a "cursor" which represents the point where a read or write will happen. Examples of channel devices include the console, the serial ports, and files.

Currently, the only fully supported channel devices are open files, the keyboard, and the screen. In the future, there should be full support for the serial ports, the parallel port, and the MIDI ports. Channel devices are assigned as follows:

Number	Device
0	Main console (keyboard and screen A)
1	Secondary console (keyboard and screen B)
2	Serial Port #1
3	Serial Port #2
4	Parallel Port
5	MIDI Ports
6	Files

By default, channels 0 and 1 are open automatically to devices 0 and 1 respectively at boot time.

Block Devices

Block devices organize their data into blocks of bytes. A block may be read from or written to a block device, and blocks maybe accessed in any order desired. Examples of block devices include the IDE/PATA hard drive, the SD card, and the floppy drive.

Out of the box, there are three block devices supported by Foenix/MCP:

Number	Device
0	SD card
1	Floppy drive (if available)
2	IDE (PATA) hard drive

Files Channels

Files represent a special channel pseudo-device. Although files are stored on block devices, they may be open as file channels, which may be accessed like a channel device. There is a special file channel driver, which converts channel reads and writes on a file to the appropriate block calls. Access to these file channels is managed in part through the file system calls listed below.

Command Line Utility

Commands

Settings

In addition to commands, the command line utility provides a number of "settings." Settings may be changed with the `SET` command and viewed with the `GET` command (if they are readable). Settings will, in general, have some sort of side-effect by being set. They can turn on or off particular functions, change colors, etc.

System Calls

On the Motorola 68000 series computers, system calls are made through the `TRAP #15` instruction. The function number (which determines which call to make) is passed in the `D0` register. Parameters are passed in the data registers: `D1` for the first parameter, `D2` for the second, and so on. Results are returned in the `D0` register, and may be 8-bits, 16-bits, or 32-bits in length.

Note that even pointers are passed using the data registers, when address registers might make more sense. This was done to keep the bindings of the system calls more straightforward. If this turns out to be a problem, later versions of the FoenixMCP may use the stack instead.

For many system calls, the return value is simply a status indicator: 0 represents success, and a negative number indicates an error condition, with the value specifying what error. For those functions that return a value (*e.g.* `sys_chan_read`), the data returned will be 0 or positive for success, and a negative number for an error condition. An exception would be `sys_int_register`, which must return a pointer and has no error condition.

The system calls are broken out into five major blocks:

- Core: these are the most essential calls for the operating system. Mostly, this is where interrupts are managed.

- Channel: these are the calls for working with channels and channel devices.

- Block: these are the calls for accessing block devices at a low level. Most user programs will not need these calls

- File System: these are the calls for accessing files at a higher level

- Process and Memory: There's only one here at the moment, but this block will support running programs and managing memory.

- Miscellaneous: these are calls I could not figure out a better place for

Core Calls

Number	Name	Description
0x00	sys_exit	Exit the user program and return the command line
0x01	Reserved	
0x02	sys_int_register	Register a function as an interrupt handler
0x03	sys_int_enable	Enable a particular interrupt
0x04	sys_int_disable	Disable a particular interrupt
0x05	sys_int_enable_all	Enable all maskable interrupts
0x06	sys_int_disable_all	Disable all maskable interrupts
0x07	sys_int_clear	Clear an interrupt's pending flag
0x08	sys_int_pending	Return true if an interrupt's pending flag is set

Function	0x00	sys_exit
Description	This function ends the currently running program and returns control to the command line. It takes a single short argument, which is the result code that should be passed back to the kernel. This function does not return.	
Prototype	`void sys_exit(short result)`	
C Example	`sys_exit(0);   // Quit the program with result 0`	

Assembly	`clr.w d0   ; Function 0: sys_exit`
	`clr.w d1   ; Result code: 0`
	`trap #15`	

Function	0x02	sys_int_register
Description	Registers a function as an interrupt handler. An interrupt handler is a function which takes and returns no arguments and will be run in at an elevated privilege level during the interrupt handling cycle. The first argument is the number of the interrupt to handle, the second argument is a pointer to the interrupt handler to register. Registering a null pointer as an interrupt handler will "deregister" the old handler. The function returns the handler that was previously registered.	

Prototype	`p_int_handler sys_int_register(short int_num,` `p_int_handler handler)`

C Example	`void sof_handler() { … }`
	
	`sys_int_register(0, sof_handler);`

Assembly	`move.w #$02,d0        ; Function sys_int_register`
	`clr.w d1               ; 0 for Channel A SOF interrupt`
	`lea.l sof_handler,d2 ; Pointer to the handler`
	`trap #15`

Function	`0x03`	`sys_int_enable`
Description	This function enables a particular interrupt at the level of the interrupt controller. The argument passed is the number of the interrupt to enable. Note that interrupts that are enabled at this level will still be disabled, if interrupts are disabled globally by `sys_int_disable_all`.	
Prototype	`void sys_int_enable(short int_num)`	
C Example	`sys_int_enable(0);  // Enable the Channel A SOF interrupt`	

Assembly	`move.w #$03,d0   ; Function: sys_int_enable`
	`clr.w d1         ; 0 is Channel A SOF interrupt`
	`trap #15`	

Function	`0x04`	`sys_int_disable`
Description	This function disables a particular interrupt at the level of the interrupt controller. The argument passed is the number of the interrupt to disable.	
Prototype	`void sys_int_disable(short int_num)`	
C Example	`sys_int_disable(0);  // Disable the SOF interrupt`	

Assembly	`move.w #$04,d0   ; Function: sys_int_disable`
	`clr.w d1         ; 0 is Channel A SOF interrupt`
	`trap #15`	

Function	`0x05`	`sys_int_enable_all`
Description	This function enables all maskable interrupts at the CPU level. It returns a system-dependent code that represents the previous level of interrupt masking.	
Prototype	`short sys_int_enable_all()`	
C Example	`sys_int_enable_all();`	

Assembly	`move.w #$05,d0    ; Function: sys_int_enable_all` `trap #15`

Function	0x06	sys_int_disable_all
Description	This function disables all maskable interrupts at the CPU level. It returns a system-dependent code that represents the previous level of interrupt masking.	
Prototype	`short sys_int_disable_all()`	
C Example	`sys_int_disable_all();`	
Assembly	`move.w 0x06,d0    ; Function: sys_int_disable_all` `trap #15`	

Function	0x05	sys_int_enable_all
Description	This function enables all maskable interrupts at the CPU level. It returns a system-dependent code that represents the previous level of interrupt masking.	
Prototype	`short sys_int_enable_all()`	
C Example	`sys_int_enable_all();`	
Assembly	`move.w #$05,d0    ; Function: sys_int_enable_all` `trap #15`	

Function	0x06	sys_int_clear
Description	This function acknowledges the processing of an interrupt by clearing its pending flag in the interrupt controller.	
Prototype	`void sys_int_clear(short int_num)`	
C Example	`sys_int_clear(1); // Clear the Channel A SOL interrupt`	

Assembly	`move.w #$05,d0    ; Function: sys_int_clear`
	`move.w #1,d1      ; Channel A SOL interrupt`
	`trap #15`	

Function	0x06	sys_int_pending
Description	This function acknowledges the processing of an interrupt by clearing its pending flag in the interrupt controller.	
Prototype	`short sys_int_clear(short int_num)`	

C Example	`sys_int_clear(1); // Clear the Channel A SOL interrupt`

Assembly	`move.w #$05,d0   ; Function: sys_int_clear`
	`move.w #1,d1     ; Channel A SOL interrupt`
	`trap #15`

Channel Calls

Number	Name	Description
0x10	sys_chan_read	Read bytes from a channel
0x11	sys_chan_read_b	Read a byte from a channel
0x12	sys_chan_read_line	Read a line of text from a channel
0x13	sys_chan_write	Write bytes to a channel
0x14	sys_chan_write_b	Write a byte to a channel
0x15	sys_chan_flush	Ensure any pending writes are completed
0x16	sys_chan_seek	Set the position of the read/write cursor in the channel
0x17	sys_chan_status	Get the status of the channel
0x18	sys_chan_ioctrl	Send a command to the channel (channel dependent)
0x19	sys_chan_register	Register a channel device driver
0x1A	sys_text_setsizes	Configure the console for the display resolution

Function	0x10	sys_chan_read
Description		Read bytes from a channel and fill a buffer with them, given the number of the channel and the size of the buffer. Returns the number of bytes read.
Prototype		`short sys_chan_read(short channel,` `           unsigned char * buffer, short size)`

C Example		`short c = …; // The channel number`
		`unsigned char buffer[128];`
		`short n = sys_chan_read(c, buffer, 128);`

Assembly		`move.w #$10,d0   ; Function: sys_chan_read`
		`move.w chan,d1    ; Channel number`
		`lea.l buffer,d2   ; Address of buffer`
		`move.w #128,d3    ; Size of buffer`
		`trap #15`

Function	0x11	sys_chan_read_b
Description		Read a single byte from the channel. Returns the byte, or 0 if none are available.
Prototype		`unsigned char sys_chan_read_b(short channel)`

C Example	`short c = …; // The channel number`
	`unsigned char b = sys_chan_read_b(c);`
| Assembly | ```
move.w #$11,d0   ; Function: sys_chan_read
move.w chan,d1   ; Channel number
trap #15
; Byte in d0
``` |

Function	0x12	sys_chan_read_line
Description	Read a line of text from a channel (terminated by a newline character or by the end of the buffer). Returns the number of bytes read.	
| Prototype | ```
short sys_chan_read_line(short channel,
            unsigned char * buffer, short size)
``` | |
| C Example | ```
short c = …; // The channel number
unsigned char buffer[128];
short n = sys_chan_read_line(c, buffer, 128);
``` | |
| Assembly | ```
move.w #$12,d0   ; Function: sys_chan_read_line
move.w chan,d1   ; Channel number
lea.l buffer,d2  ; Address of buffer
move.w #128,d3   ; Size of buffer
trap #15
``` | |

Function	0x13	sys_chan_write
Description	Write bytes from a buffer to a channel, given the number of the channel and the size of the buffer. Returns the number of bytes written.	
| Prototype | ```
short sys_chan_write(short channel,
          unsigned char * buffer, short size)
``` | |
| C Example | ```
short c = …; // The channel number
unsigned char buffer[128];
// ...
short n = sys_chan_write(c, buffer, 128);
``` | |
| Assembly | ```
move.w #$13,d0   ; Function: sys_chan_write
move.w chan,d1   ; Channel number
lea.l buffer,d2  ; Address of buffer
move.w #128,d3   ; Size of buffer
trap #15
``` | |

Function	0x14	sys_chan_write_b
Description	Write a single byte to the channel.	

Prototype	`short sys_chan_write_b(short channel, unsigned char b)`

C Example	`short c = …; // The channel number`
	`sys_chan_read_b(c, 0x41);`

Assembly	`move.w #$14,d0    ; Function: sys_chan_write_b`
	`move.w chan,d1    ; Channel number`
	`move.b #$41,d2    ; The byte to write`
	`trap #15`

Function	0x15	sys_chan_flush
Description	Ensure any pending writes to a channel are completed.	

Prototype	`short sys_chan_flush(short channel)`

C Example	`short c = …; // The channel number`
	`sys_chan_flush(c);`

Assembly	`move.w #$15,d0    ; Function: sys_chan_flush`
	`move.w chan,d1    ; Channel number`
	`trap #15`

Function	0x16	sys_chan_seek
Description	Set the position of the input/output cursor. This function may not be honored by a given channel as not all channels are "seekable." In addition to the usual channel parameter, the function takes two other parameters: • `position` = the new position for the cursor • `base` = whether the position is absolute (0), or relative to the current position (1).	

Prototype	`short sys_chan_seek(short channel,`
	`                    long position,`
	`                    short base)`

C Example	`short c = …; // The channel number`
	`sys_chan_seek(c, -10, 1); // Move the point back 10 bytes`

Assembly	`move.w #$16,d0        ; Function: sys_chan_seek`
	`move.w chan,d1        ; Channel number`
	`move.l #$FFFFFFFF,d2 ; Position: -1`
	`move.w #1,d3          ; Base: relative`
	`trap #15`

Function	0x17	sys_chan_status
Description	Gets the status of the channel. The meaning of the status bits is channel-	

specific, but four bits are recommended as standard:

- 0x01: The channel has reached the end of its data

- 0x02: The channel has encountered an error

- 0x04: The channel has data that can be readable

- 0x08: The channel can accept data

Prototype	`short sys_chan_status(short channel)`

C Example	`short c = …; // The channel number`
	`sys_chan_status(c);`

Assembly	`move.w #$17,d0       ; Function: sys_chan_status`
	`move.w chan,d1       ; Channel number`
	`trap #15`

Function	0x18	sys_chan_ioctrl
Description		Send a command to a channel. The mapping of commands and their actions are channel-specific. The return value is also channel and command-specific. In addition to the channel number, the function takes three arguments:

- `command`: the number of the command to execute

- `buffer`: an array of bytes to serve as additional data for the command (may be null)

- `size`: the number of bytes in the buffer

Prototype	`short sys_chan_ioctrl(short channel,`
	`                      short command,`
	`             unsigned char * buffer,`
	`                      short size)`

C Example	`short c = …;                    // The channel number`
	`short cmd = …;                   // The command`
	`short r = sys_chan_status(c, cmd, 0, 0); // Send simple command`

Assembly	`move.w #$18,d0       ; Function: sys_chan_ioctrl`
	`move.w chan,d1       ; Channel number`
	`move.w #1,d2         ; Command 1`
	`move.l #0,d3         ; Null buffer`
	`move.w #0,d4         ; Buffer is empty`
	`trap #15`
	`; Result is in D0`

Function	0x19	sys_chan_register
Description		Register a device driver for a channel device. A device driver consists of a structure that specifies the name and number of the device as well as the various handler functions that implement the channel calls on a channel for that device. See the section "Extending the System" below for more information.
Prototype		`short sys_chan_register(struct s_dev_chan *device)`
C Example		`struct s_dev_chan dev;` `short r = sys_chan_register(&dev); // Register the driver`

Assembly		`move.w #$19,d0      ; Function: sys_chan_register`
		`lea.l dev,d1        ; Device descriptor`
		`trap #15`

Block Calls

Number	Name	Description
0x20	sys_bdev_getblock	Read a block from the block device
0x21	sys_bdev_writeblock	Write a block to a block device
0x22	sys_bdev_flush	Ensure any pending writes are completed
0x23	sys_bdev_status	Get the status of the block device
0x24	sys_bdev_ioctrl	Send a command to the block device (device dependent)
0x25	sys_bdev_register	Register a block device driver

Function	0x20	sys_bdev_getblock
Description		Read a block from a block device. Returns the number of bytes read. In addition the number of the block device, this function takes three arguments: • `lba`: the logical block address of the block to read • `buffer`: the byte array in which to store the data • `size`: the number of bytes in the byte array

Prototype		`short sys_bdev_getblock(short dev,`
		`                         long lba,`
		`             unsigned char * buffer,`
		`                         short size)`

C Example		`short bdev = BDEV_HDC; // The device number`

| | ```
unsigned char buffer[128];

// Read the MBR of the hard drive
short n = sys_bdev_getblock(bdev, 0, buffer, 128);
``` |
| Assembly | ```
move.w #$20,d0        ; Function: sys_bdev_getblock
move.w #BDEV_HDC,d1   ; Channel number
clr.l d2              ; LBA: 0 (MBR)
lea.l buffer,d3       ; Address of buffer
move.w #128,d4        ; Size of buffer
trap #15
``` |

Function	0x20	sys_bdev_putblock
Description		Write a block from a block device. Returns the number of bytes written. In addition the number of the block device, this function takes three arguments: • lba: the logical block address of the block to write • buffer: the byte array in which to store the data • size: the number of bytes in the byte array
| Prototype | | ```
short sys_bdev_putblock(short dev,
                            long lba,
                 unsigned char * buffer,
                            short size)
``` |
| C Example | | ```
short bdev = BDEV_HDC; // The device number
unsigned char buffer[128];

// Write the MBR of the hard drive
short n = sys_chan_putblock(bdev, 0, buffer, 128);
``` |
| Assembly | | ```
move.w #$21,d0        ; Function: sys_bdev_putblock
move.w #BDEV_HDC,d1   ; Channel number
clr.l d2              ; LBA: 0 (MBR)
lea.l buffer,d3       ; Address of buffer
move.w #128,d4        ; Size of buffer
trap #15
``` |

Function	0x22	sys_bdev_flush
Description		Ensure any pending writes to a block device are completed.
Prototype		`short sys_bdev_flush(short dev)`
C Example		`short bdev= …; // The device number`

	`sys_bdev_flush(bdev);`

Assembly	`move.w #$22,d0   ; Function: sys_bdev_flush`
	`move.w bdev,d1   ; Device number`
	`trap #15`

Function	0x23	sys_bdev_status
Description	Gets the status of a block device. The meaning of the status bits is device specific, but there are two bits that are required in order to support the file system: • 0x01: Device has not been initialized yet • 0x02: Device is present	
Prototype	`short sys_bdev_status(short dev)`	

C Example	`short bdev = …; // The channel number`
	`sys_chan_status(bdev);`	

Assembly	`move.w #$23,d0      ; Function: sys_bdev_status`
	`move.w bdev,d1      ; Device number`
	`trap #15`	

Function	0x24	sys_bdev_ioctrl
Description	Send a command to a block device. The mapping of commands and their actions are device-specific. The return value is also device and command-specific. In addition to the device number, the function takes three arguments: • command: the number of the command to execute • buffer: an array of bytes to serve as additional data for the command (may be null) • size: the number of bytes in the buffer Four commands should be supported by all devices: • GET_SECTOR_COUNT (1): Returns the number of physical sectors on the device • GET_SECTOR_SIZE (2): Returns the size of a physical sector in bytes • GET_BLOCK_SIZE (3): Returns the block size of the device. Really only relevant for flash devices and only needed by FatFS	

	• GET_DRIVE_INFO (4): Returns the identification of the drive
| Prototype | ```
short sys_bdev_ioctrl(short channel,
                      short command,
             unsigned char * buffer,
                      short size)
``` |
| C Example | ```
short dev = …;                  // The device number
short cmd = …;                  // The command
short r = sys_bdev_status(dev, cmd, 0, 0); // Send simple
command
``` |
| Assembly | ```
move.w #$24,d0        ; Function: sys_bdev_ioctrl
move.w bdev,d1        ; Channel number
move.w #1,d2          ; Command 1
move.l #0,d3          ; Null buffer
move.w #0,d4          ; Buffer is empty
trap #15
; Result is in D0
``` |

Function	0x25	sys_bdev_register
Description	Register a device driver for a block device. A device driver consists of a structure that specifies the name and number of the device as well as the various handler functions that implement the block device calls for that device. See the section "Extending the System" below for more information.	
Prototype	`short sys_bdev_register(struct s_dev_block *device)`	
| C Example | ```
struct s_dev_block dev;
short r = sys_bdev_register(&dev); // Register the driver
``` | |
| Assembly | ```
move.w #$25,d0        ; Function: sys_chan_register
lea.l dev,d1          ; Device descriptor
trap #15
``` | |

File System Calls

Number	Name	Description
0x30	sys_fsys_open	Open a file
0x31	sys_fsys_close	Close a file
0x32	sys_fsys_opendir	Open a directory
0x33	sys_fsys_closedir	Close a directory
0x34	sys_fsys_readdir	Read a directory entry
0x35	sys_fsys_findfirst	Find the first entry in a directory matching a pattern
0x36	sys_fsys_findnext	Find the next entry in a directory matching a pattern

0x37	sys_fsys_delete	Delete a file
0x38	sys_fsys_rename	Rename a file
0x39	sys_fsys_mkdir	Create a directory
0x3A	sys_fsys_load	Load a file into memory
0x3B	sys_fsys_save	Save a block of memory to a file
0x3C	sys_proc_run	Load and run an executable binary file
0x3D	sys_fsys_register_loader	Register a file loader

Function	0x30	sys_fsys_open
Description		Attempt to open a file in the file system for reading or writing. Two arguments are required:
		• path: the path to the file to open
		• mode: flags indicating how the file should be opened:
		○ 0x01: Read
		○ 0x02: Write
		○ 0x04: Create if new
		○ 0x08: Always create
		○ 0x10: Open file if existing, otherwise create
		○ 0x20: Open for append
		Returns a channel number associated with the file. If the returned number is negative, there was an error opening the file.
Prototype		`short sys_fsys_open(const char * path` `short mode)`
C Example		```short chan = sys_fsys_open("hello.txt", 0x01);` `if (chan > 0) {` ` // File is open for reading` `} else {` ` // File was not open… chan has the error number` `}```
Assembly		```move.w #30,d0 ; Function: sys_fsys_open` `lea path,d1 ; Path…` `move.w #$01,d2 ; Mode = 1 (read existing)` `trap #15` `; Channel number will be in d0```

Function	0x31	sys_fsys_close
Description	Close a file that was previously opened, given its channel number. If there were writes done on the channel, those writes will be committed to the block device holding the file.	
Prototype	`void sys_fsys_close(short chan);`	

C Example	```short chan = sys_fsys_open(…);```
	 ```// …```
	 ```sys_fsys_close(chan);```	

Assembly	```move.w #$31,d0 ; Function: sys_fsys_close```
	 ```move.w (chan),d1 ; Channel number for the file```
	 ```trap #15```	

Function	0x32	sys_fsys_opendir
Description	Open a directory on a volume for reading, given its path. Returns a directory handle number on success, or a negative number on failure.	
Prototype	`short sys_fsys_opendir(const char *path);`	

C Example	```short dir = sys_fsys_opendir("/hd0/System");```
	 ```if (dir > 0) {```
	 ``` // dir can be used for reading the directory entries```
	 ```} else {```
	 ``` // There was an error… error number in dir```
	 ```}```	

Assembly	```move.w #$32,d0 ; Function: sys_fsys_opendir```
	 ```lea.l path,d1 ; Path```
	 ```trap #15```
	 ```; D0 contains the directory number or an error```	

Function	0x33	sys_fsys_closedir
Description	Close a previously open directory, given its number.	
Prototype	`void sys_fsys_closedir(short dir);`	

C Example	```short dir = sys_fsys_opendir("/hd0/System");```
	 ```if (dir > 0) {```
	 ``` // dir can be used for reading the directory entries```
	 ```} else {```
	 ``` // There was an error… error number in dir```
	 ```}```	

| Assembly | ```
move.w #$33,d0       ; Function: sys_fsys_opendir
move.w (dir),d1      ; Directory number
trap #15
``` |
| --- | --- |

Function	0x34	sys_fsys_readdir
Description	Given the number of an open directory, and a buffer in which to place the data, fetch the file information of the next directory entry. (See below for details on the file_info structure.) Returns 0 on success, a negative number on failure.	
Prototype	short sys_fsys_readdir(short dir, struct s_file_info *file);	
| C Example | ```
short dir = sys_fsys_opendir("/hd0/System");
if (dir > 0) {
   // dir can be used for reading the directory entries
   struct s_file_info file;
   if (sys_fsys_readdir(dir, &file_info) == 0) {
      // file_info contains information...
   } else {
      // Could not read the file entry...
   }
} else {
   // There was an error… error number in dir
}
``` | |
| Assembly | ```
move.w #$34,d0       ; Function: sys_fsys_opendir
move.w (dir),d1      ; Directory number
lea.l file_info,d2   ; Pointer to the file info structure
trap #15
``` | |

Function	0x35	sys_fsys_findfirst
Description	Given the path to a directory to search, a search pattern, and a pointer to a file_info structure, return the first entry in the directory that matches the pattern. Returns a directory handle on success, a negative number if there is an error	
| Prototype | ```
short sys_fsys_findfirst(const char *path,
                         const char *pattern,
                    struct s_file_info *file);
``` | |
| C Example | ```
struct s_file_info file;
short dir = sys_fsys_findfirst("/hd0/System/",
                               "*.pgx",
                               &file_info);
``` | |

| | ```
if (dir == 0) {
   // file_info contains information...
} else {
   // Could not read the file entry...
}
``` |
| Assembly | ```
move.w #$34,d0        ; Function: sys_fsys_findfirst
lea.l path,d1         ; Pointer to path
lea.l pattern,d2      ; Pointer to pattern
lea.l file_info,d3    ; Pointer to the file info structure
trap #15
``` |

Function	0x36	sys_fsys_findnext
Description	Given the directory handle for a previously open search (from sys_fsys_findfirst), and a file_info structure, fill out the structure with the file information of the next file to match the original search pattern. Returns 0 on success, a negative number if there is an error	
| Prototype | ```
short sys_fsys_findfirst(const char *path,
                            const char *pattern,
                  struct s_file_info *file);
``` | |
| C Example | ```
struct s_file_info file;
short dir = sys_fsys_findfirst("/hd0/System/",
                                "*.pgx",
                                &file_info);
if (dir == 0) {
   // file_info contains information…

   // Look for the next…
   short result = sys_fsys_findnext(dir, &file_info);
} else {
   // Could not read the file entry...
}
``` | |
| Assembly | ```
move.w #$36,d0        ; Function: sys_fsys_findnext
move.w (dir),d1       ; Directory
lea.l file_info,d2    ; Pointer to the file info structure
trap #15
``` | |

Function	0x37	sys_fsys_delete
Description	Delete a file or directory, given its path. Returns 0 on success, a negative number if there is an error	
Prototype	short sys_fsys_delete(const char *path);	

C Example	short result = sys_fsys_delete("/hd0/test.txt");
| Assembly | ```
move.w #$37,d0        ; Function: sys_fsys_delete
lea.l path,d1         ; Path
trap #15
``` |


Function	0x38	sys_fsys_rename
Description	Rename a file or directory. Returns 0 on success, a negative number if there is an error	
| Prototype | ```
short sys_fsys_rename(const char *old_path,
                      const char *new_name);
``` | |
| C Example | short result = sys_fsys_rename("/hd0/test.txt", "doc.txt"); | |
| Assembly | ```
move.w #$38,d0        ; Function: sys_fsys_delete
lea.l path,d1         ; Old Path
lea.l new_name,d2     ; New Name
trap #15
``` | |


Function	0x39	sys_fsys_mkdir
Description	Create a directory. Returns 0 on success, a negative number if there is an error	
Prototype	short sys_fsys_mkdir(const char *path);	
C Example	short result = sys_fsys_mkdir("/hd0/Samples");	
| Assembly | ```
move.w #$39,d0        ; Function: sys_fsys_delete
lea.l path,d1         ; Path
trap #15
``` | |


Function	0x3A	sys_fsys_load

Description	Load a file into memory.
	
	Takes three arguments:
	
	    • path: the path to the file to load
	
	    • destination: the destination address in memory (0 to use the address in the file)
	
	    • start: a pointer to a long to receive the starting address, if the file is an executable binary.	

	Returns 0 on success, a negative number if there is an error

Prototype	```short sys_fsys_load(const char *path,
	                      long destination,
	                      long *start);```

C Example	```long start; short result = sys_fsys_load("hello.pgx", 0, &start);```
Assembly	```move.w #$3A,d0 ; Function: sys_fsys_load lea.l path,d1 ; Path clr.l d2 lea.l start,d3 trap #15```

Function	0x3B	sys_fsys_save
Description	Not implemented yet.	
Prototype		
C Example		
Assembly		

Function	0x3C	sys_fsys_register_loader
Description	Register a file loader for a binary file type. A file loader is a function that takes a channel number for a file to load, a long representing the destination address, and a pointer to a long for the start address of the program. These last two parameters are the same as are provided the sys_fsys_load. The registration function takes two arguments: • extension: a three character extension to map the file type to the loader • loader: a pointer to the loading routine. On success, returns 0. It there is an error in registering the loader, returns a negative number.	
Prototype	```short sys_fsys_register_loader(const char * extension, p_file_loader loader);```	

C Example	```short foo_loader(short chan, long destination, long * start) {
	    // Load file to destination (if provided)
	    // If executable, set start to address to run```	

| | ```
    return 0; // If successful
};
// ...
short result = sys_fsys_register_loader("FOO", foo_loader);
``` |
| Assembly | ```
move.w #$3C,d0        ; Function: sys_fsys_run
lea.l path,d1         ; Path
clr.w d2              ; argc is 0
clr.l d3              ; argv is null
trap #15
``` |

Process and Memory Calls

Number	Name	Description
0x40	sys_proc_run	Load and run an executable file

Function	0x40	sys_proc_run
Description	Load and run an executable binary file.	
	It takes three arguments:	
	• path: the path to the file to run	
	• argc: the number of parameters to give to the executable	
	• argv: an array of strings containing the parameters to give to the executable	
	This function will not return on success, since Foenix/MCP is single tasking. Any return value will be an error condition.	
| Prototype | ```
short sys_proc_run(const char * path,
                            int argc,
                            char * argv[]);
``` | |
| C Example | ```
int argc = 2;
char * argv[] = {
    "hello.pgx",
    "test"
};
short result = sys_proc_run("hello.pgx", argv, argc);
``` | |
| Assembly | ```
move.w #$40,d0        ; Function: sys_fsys_run
lea.l path,d1         ; Path
clr.w d2              ; argc is 0
clr.l d3              ; argv is null
trap #15
``` | |

Miscellaneous Calls

Number	Name	Description
0x50	sys_time_ticks	Get the number of "ticks" since system startup
0x51	sys_time_setrtc	Set the date and time in the real time clock
0x52	sys_time_getrtc	Get the date and time from the real time clock
0x53	sys_kbd_setlayout	Set the keyboard layout translation tables
0x54	sys_err_message	Get the error message for a given error number

Function	0x50	sys_time_ticks
Description	\multicolumn{2}{l}{Returns the number of "ticks" since system startup. Ticks are being timed by the real time clock chip, which allows for a periodic interrupt every 976 microseconds, or just a little short of once a millisecond. This tick counter will therefore serve as a reasonable approximation to a millisecond clock, but it should not be used for critical timing.}
Prototype	\multicolumn{2}{l}{`long sys_time_ticks()`}
C Example	\multicolumn{2}{l}{`long ticks = sys_time_ticks();`}
Assembly	\multicolumn{2}{l}{}

```
move.w #$50,d0        ; Function: sys_time_ticks
trap #15
; Tick count will be a 32 bit number in d0
```

Function	0x51	sys_time_setrtc
Description		Sets the date and time in the real time clock. The date and time information is provided in an s_time structure (see below).
Prototype		`void sys_time_setrtc(struct s_time *time)`
C Example		

```
struct s_time time;
// …
sys_time_setrtc(&time);
```

| Assembly | | |

```
move.w #$51,d0        ; Function: sys_time_setrtc
lea.l time,d1         ; Pointer to s_time structure
trap #15
```

Function	0x52	sys_time_getrtc
Description		Gets the date and time in the real time clock. The date and time information is provided in an s_time structure (see below).
Prototype		`void sys_time_getrtc(struct s_time *time)`

C Example	`struct s_time time;`
	`// …`
	`sys_time_getrtc(&time);`

Assembly	`move.w #$52,d0        ; Function: sys_time_getrtc`
	`lea.l time,d1         ; Pointer to s_time structure`
	`trap #15`

Extending the System

Foenix/MCP is designed to be somewhat extensible. Since it is meant to be small and stay as much out of the way of the user programs as possible, Foenix/MCP does not have all of the features that absolutely everyone will want. Therefore, there are four main ways that the user can extend the capabilities of Foenix/MCP: channel device drivers, block device drivers, keyboard translation tables, and file loaders.

Channel Device Drivers

Channel device drivers provide the functions needed by Foenix/MCP to support a channel opened on a device. With some exceptions, each channel system call is routed through the channel to the correct channel driver function. Channel drivers can be added to the system using the `sys_chan_register` call, specifying all of the relevant information about the driver using a structure:

```
struct s_dev_chan {
    short number;           // Number of the device
    char * name;            // Name of the device

    FUNC_V_2_S init;        // Initialize the device
    FUNC_CBS_2_S open;      // Open a new channel for the device
    FUNC_V_2_S close;       // Close a channel
    FUNC_CBS_2_S read;      // Read a sequence of bytes from the device
    FUNC_CBS_2_S readline;  // Read a line of text from the device
    FUNC_C_2_S read_b;      // Read a single byte from the device
    FUNC_CcBS_2_S write;    // Write a sequence of bytes to the device
    FUNC_CB_2_S write_b;    // Write a single byte to the device
    FUNC_C_2_S status;      // Return the status of the device
    FUNC_C_2_S flush;       // Commit any pending writes to the device
    FUNC_CLS_2_S seek;      // Set the in/out position of the device
    FUNC_CSBS_2_S ioctrl;   // Send commands to the device or driver
};
```

Most of the fields in the structure are function pointers, which have one of the following types:

```
typedef short (*FUNC_V_2_S)();
typedef short (*FUNC_CBS_2_S)(p_channel, unsigned char *, short);
typedef short (*FUNC_C_2_S)(p_channel);
typedef short (*FUNC_CcBS_2_S)(p_channel, const unsigned char *, short);
typedef short (*FUNC_CB_2_S)(p_channel, unsigned char);
```

```
typedef short (*FUNC_CLS_2_S)(p_channel, long, short);
typedef short (*FUNC_CSBS_2_S)(p_channel, short, unsigned char *, short);
```

Where `p_channel` is a pointer to a channel structure, which maps an open channel to its device and provides space for the channel driver to store data relevant to that particular channel. The channel device drivers are passed this structure directly by the channel system calls, rather than the channel number used by user programs.

```
struct s_channel {
    short number;           // The number of the channel
    short dev;              // The number of the channel's device
    unsigned char data[32]; // A block of channel specific data
};
```

To implement a driver for a new channel device, all of the functions should be implemented (if a function is not needed, it should still be implemented but return a 0). Then a s_chan_dev structure should be allocated and filled out, with the number being the number of the device to support, and name points to a suitable name for the device.

Most of the functions needed are directly mapped to to the channel system calls of the same name, and they simply perform the operations needed for those calls. Three functions should be called out for special consideration:

The `init` function performs initialization functions. It is called once per device. This can be a place for setting up the device itself or installing interrupt handlers for the device.

The `open` function is called when the user program opens a channel, after a channel structure has been allocated for the channel. This is the correct place for setting up a connection for a specific transaction on the device. This is another point where interrupt handlers might be installed or turned on, or when specific connection settings are made in the device (like serial baud rate).

The `close` function is called when the user program closes a previously opened channel. This function should perform any house keeping functions needed before the channel is returned to the kernel's pool. In particular, if the device buffers writes, any writes that are still pending should be written to the device.

Block Device Drivers

Block device drivers are used by Foenix/MCP to provide block level access to block devices like the SD card, floppy drive, and IDE/PATA hard drive. The main use of block device drivers is the FatFS file system, which is used to provide file channels. Block drivers can be added to the system in a similar way to channel device drivers by implementing the functions

needed by Foenix/MCP and registering them using the `sys_bdev_register` call. The information about the block device is provided through a `s_block_dev` structure:

```
struct s_dev_block {
    short number;              // The number of the device
    char * name;               // The name of the device
    FUNC_V_2_S init;           // Initialize the device
    FUNC_LBS_2_S read;         // Read a block from the device
    FUNC_LcBS_2_S write;       // Write a block to the device
    FUNC_V_2_S status;         // Get the status of the device
    FUNC_V_2_S flush;          // Ensure that any pending writes are completed
    FUNC_SBS_2_S ioctrl;       // Issue a control command to the device
};
```

The block device structure is similar to the channel device in that it mostly provides the functions needed to implement the block system calls, using the following function pointer types:

```
typedef short (*FUNC_LBS_2_S)(long, unsigned char *, short);
typedef short (*FUNC_LcBS_2_S)(long, const unsigned char *, short);
typedef short (*FUNC_SBS_2_S)(short, unsigned char *, short);
typedef short (*FUNC_LB_2_S)(long, short);
```

One difference with the channel drivers is that a block driver is tied to its specific device, therefore the handler functions do not take a device number or other structure.

As before, when registering a driver, the device number is provided in the `number` field, and a useful name is provided in `name`. The `init` function will be called once to allow the driver to initialize the device, install interrupt handlers, or perform other functions.

Otherwise, `read` and `write` perform the `getblock` and `putblock` functions, and take a block address, a buffer of bytes, and a buffer size as arguments. The `status` and `flush` functions map to the `sys_bdev_status` and `sys_bdev_flush` calls. And finally, `ioctrl` maps to the `sys_bdev_ioctrl` function, and takes a command number, a buffer of bytes, and a size of the buffer as arguments.

Keyboard Translation Tables

File Loaders

Appendix

ANSI Terminal Codes

Keyboard Scan Codes

Useful Data Structures

Time

```
// Structure used for real time clock functions
struct s_time {
    short year;        // Year (0 – 9999)
    short month;       // Month (1 = January through 12 = December)
    short day;         // Day of month (1 - 31)
    short hour;        // Hour (0 – 12 / 23)
    short minute;      // Minute (0 - 59)
    short second;      // Seconds (0 - 59)
    short is_pm;       // For 12-hour clock, 1 = PM
    short is_24hours;  // 1 = clock is 24-hours, 0 = clock is 12-hours
}
```

Directory Entries

```
// Structure used for directory entry information
struct s_file_info {
    long size;                 // Size of the file in bytes
    unsigned short date;       // Creation date
    unsigned short time;       // Creation time
    unsigned char attributes;  // Attribute bits
    char name[MAX_PATH_LEN];   // Name of the file (256 bytes)
}
```

File attribute bits:

0x01	Read only
0x02	Hidden file
0x04	System file
0x10	Directory
0x20	Archive

Error Codes

PGX File Format

PGZ File Format