Foenix/MCP

A Simple, Portable Operating System

for the Foenix Line of Computers

version 1.0-alpha.x

Peter Weingartner

November 9, 2021

Overview

The Foenix/MCP is the new kernel for the Foenix line of retro-style computers. Written in C, primarily for the A2560 series of M680x0 based computers, the kernel is meant to be simple and portable across the entire line of Foenix computers and able to run on any CPU the systems will provide.

The intention of Foenix/MCP is to provide very simple startup and access to the Foenix computers for their owners. It is not intended to be the operating system that solves all problems but is really little more than a glorified loader. Its purpose is to help the user get started with their Foenix and run programs, but as much as possible it should get out of the programmer's way rather than require them to program in a certain way. As such, there are a few key goals in the design, as well as some anti-goals:

Foenix/MCP Goals

- Allow the user to access and manage files on hard drive, SD card, or floppy disk (for machines that support floppy drives)
- Allow users to load and run programs
- Provide functions to support user programs in doing tedious or complex low level tasks that may be necessary but hardly interesting to write as part of a game.
- Provide initialization for the built-in devices at boot up

Foenix/MCP Anti-goals

- Enforce a certain way of programming a Foenix computer
- Lock any part of the machine down so it cannot be accessed by a user programmer
- Require user programs to incorporate or link multiple libraries of code to do anything

In keeping with these goals and anti-goals, Foenix/MCP is very simple. It is a single tasking kernel with no support for multi-tasking. As much as possible, however, the kernel routines are written in a re-entrant fashion to try to be thread safe if a user program wants to add multi-tasking. There is currently no memory manager included in the system calls, although one may need to be added in the near future. The concept is that once a user program is loaded, it may have access to everything. While drivers are included for many of the devices on the Foenix computers, programs are welcome to take over direct control of those devices or replace the included device drivers, as needed. Doing so may interfere with documented functionality of Foenix/MCP, but this to be encouraged rather than otherwise.

Your Foenix computer is yours, *absolutely*.[1]

Copyright Information

Devices

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

1 **Warning:** Please do not attempt to install an 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. The author of Foenix/MCP, ENCOM, and their associated subsidiaries and agents will not be liable for any unexpected behavior experienced by users. END-OF-LINE

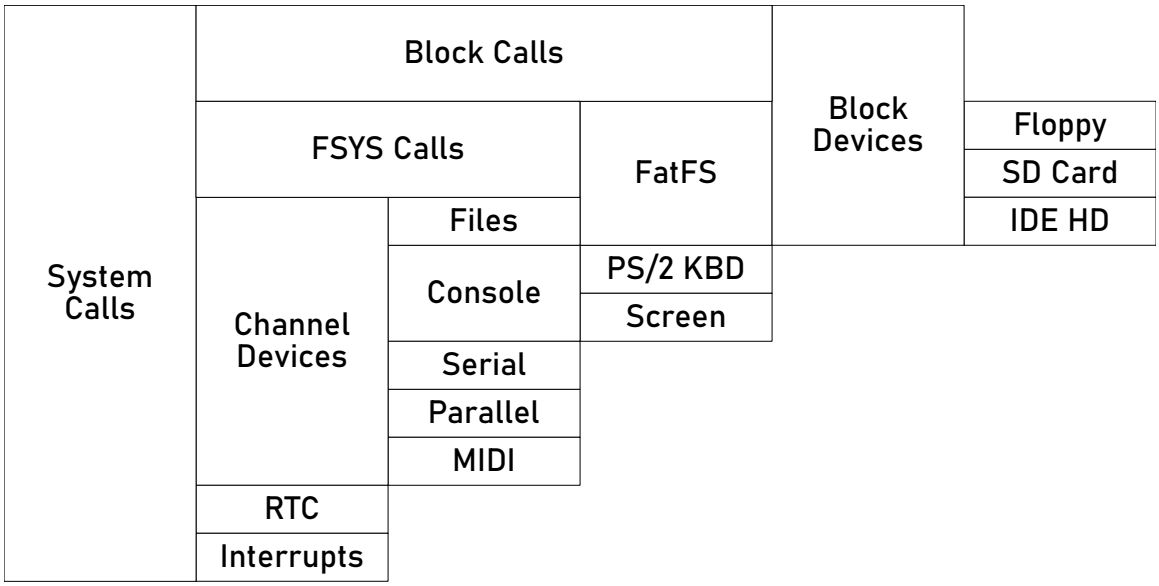

Figure 1: Organization of Foenix/MCP

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

The command line utility works much like the command line in CP/M or MS-DOS. The first "word" typed on a line is the "command" to be executed. There are several built-in commands, but if a command is not recognized as a built-in command, Foenix/MCP will try to find and run an executable file of that name. In the current version of Foenix/MCP, it will look for a PGX or PGZ file of that name in the current working directory. In future versions, a more sophisticated search mechanism will be in place, and other file formats may be supported.

The built in commands include:

`HELP / ?` – Print out a potentially useful help message, summarizing the commands that may be used.

`CD <path>` – Change the current working directory.

`CLS` – Clear the screen

`DEL <path>` – Delete a file or empty directory, given its path

`DISKFILL <drive #> <sector #> <byte>` – A diagnostic tool to fill a sector on a drive with a byte

`DISKREAD <drive #> <sector #>` – A diagnostic tool to read and display a sector on a drive

`DUMP <address> [<count>]` – Display <count> bytes of memory on the screen. If no count is provided, the command will display sixteen bytes.

`LABEL <drive #> <label>` – Set the volume label of a drive.

`LOAD <path>` – Read a file into memory

`MKDIR <path>` – Create a directory, given its path

`PEEK8 <address>` – Read and display a byte from an address in memory.

`PEEK16 <address>` – Read and display a 16-bit word from an address in memory.

`PEEK32 <address>` – Read and display a 32-bit word from an address in memory.

`POKE8 <address> <value>` – Store a byte at an address in memory

`POKE16 <address> <value>` – Store a 16-bit word at an address in memory

`POKE32 <address> <value>` – Store a 32-bit word at an address in memory

`PWD` – Display the current working directory

`REN <old path> <new path>` – Rename a file

`SET <name> <value>` – Assign a value to a setting (see below)

`GET <name>` – Display the current value of a setting

`SYSINFO` – Display potentially useful information about the Foenix computers

`TYPE <path>` – Read and print out a file

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*. Settings currently in the command line include:

`DATE yyyy-mm-dd` – The current date in the real time clock

`TIME hh:mm:ss` – The current time in the real time clock

`FONT <path>` – The current font used on the active text displayed

`KEYBOARD <path>` – The keyboard layout. The path provided must go to a binary file 1024 bytes long. These 1024 bytes provide all eight translation tables needed by the `sys_kbd_layout` system call (see below for details).

This set will expand over time. Settings to be implemented in the future might include:

`VOLUME <value>` – The volume level of audio

`GRAPHICS <value>` – The graphics setting for Vicky

`BORDERSIZE <value>` – The size of the border

`BORDERCOLOR <color>` – The color of the border

`TEXTCOLOR <color>` – The color of the text and the background

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_chan_open`	Open a channel device
`0x1B`	`sys_chan_close`	Close a channel
`0x1C`	`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 read • 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`	

Function	`0x1A`	`sys_chan_open`
Description	Open a channel device for reading or writing. Takes three arguments: • `dev`: the number of the device to open • `path`: a device-specific string describing any particular resource/or parameters for the connection. This might be ignored by the channel device, if it is irrelevant. • `mode`: a bit field specifying if the connection is for reading (0x01) or writing (0x02). A channel device might ignore this, if the direction is inherent. Returns the channel number (if positive) or an error code (if negative).	

Prototype	`short sys_chan_open(short dev,`
	`const char *path,`
	`short mode)`	

C Example	`// Open a connection to the serial port:` `// 9600 bps, 8-data bits, one stop bit, no parity`	

	`short chan = sys_chan_open(2, “9600,8,1,N”, 3);`

Assembly	`move.w #$1A,d0 ; Function: sys_chan_open`
	`move.w #2,d1 ; Device`
	`lea.l path,d2 ; Path`
	`move.w #3,d3 ; Mode`
	`trap #15`

Function	`0x1B`	`sys_chan_close`
Description	Close a channel that was previously open by `sys_chan_open`. Takes a channel number, but does not return anything useful.	
Prototype	`short sys_chan_close(short dev)`	

C Example	`short chan = sys_chan_open(2, “9600,8,1,N”, 3);`
	`…`
	`sys_chan_close(chan);`	

Assembly	`move.w #$1B,d0 ; Function: sys_chan_register`
	`move.w (chan),d1 ; Channel number`
	`trap #15`	

Function	`0x1C`	`sys_text_setsizes`
Description	Sets the text screen device driver to the current screen geometry, based on the display resolution and border size.	
Prototype	`void sys_text_setsizes()`	
C Example	`sys_text_setsizes()`	
Assembly	`move.w #$1C,d0 ; Function: sys_chan_register` `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_get_label`	Get the label of a volume
0x3C	`sys_fsys_set_label`	Set the label of a volume
0x3D	`sys_fsys_set_cwd`	Set the current working directory
0x3E	`sys_fsys_get_cwd`	Get the current working directory
`0x3F`	`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_get_label`
Description	Get the label of a volume. Takes two arguments: • `path`: the path to the drive to get the label from • `label`: a string large enough to take a label	
Prototype	`short sys_fsys_get_label(const char * path, char * label)`	
C Example	`char label[64];` `short result = sys_fsys_get_label("@hd0:", label);`	
Assembly		

Function	`0x3C`	`sys_fsys_set_label`

Description	Set the label of a volume. Takes two arguments: • `drive`: the number of the block device change • `label`: the new label for the volume
Prototype	`short sys_fsys_set_label(short drive, const char * label);`
C Example	`short result = sys_fsys_set_label(2, "FNXHD0");`
Assembly	

Function	`0x3D`	`sys_fsys_set_cwd`
Description	Set the current working directory. Takes a single string argument, the path to make the current working directory.	
Prototype	`short sys_fsys_set_cwd(const char * path)`	
C Example	`short result = sys_fsys_set_cwd("@hd0:Foo:Bar");`	
Assembly		

Function	`0x3E`	`sys_fsys_get_cwd`
Description	Get the current working directory. Takes two arguments: • `path`: string to fill with the current working directory • size: the size of the `path` string variable	
Prototype	`short sys_fsys_get_cwd(char * path, short size)`	
C Example	`char cwd[255];` `short result = sys_fsys_get_cwd(cwd, 255);`	
Assembly		

Function	`0x3F`	`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_jiffies	Get the number of “jiffies” 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_scancode	Return the next scan code from the keyboard
0x54	sys_kbd_setlayout	Set the keyboard layout translation tables
0x55	sys_err_message	Get the error message for a given error number

Function	0x50	sys_time_jiffies
Description	Returns the number of “jiffies” since system startup. A jiffy is 1/60 second. This clock counts the number of jiffies since the last system startup, but it is not terribly precise. This counter should be sufficient for providing timeouts and wait delays on a fairly course level, but it should not be used when precision is required. At the time of this writing, the jiffy counter is provided by the start-of-frame interrupt, and it can vary with different resolutions. In future, this timer should be provided by the real time clock and may be supplemented with a finer grain timer.	
Prototype	`long sys_time_jiffies()`	
C Example	`long ticks = sys_time_jiffies();`	
Assembly	`move.w #$50,d0  ; Function: sys_time_jiffies` `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`	

Function	`0x53`	`sys_kbd_scancode`
Description	Returns the next keyboard scan code (0 if none are available). Note that reading a scan code directly removes it from being used by the regular console code and may cause some surprising behavior if you combine the two. See below for details about Foenix scan codes.	
Prototype	`unsigned short sys_kbd_scancode()`	
C Example	`unsigned short code = sys_kbd_scancode();`	

Assembly	`move.w #$53,d0 ; Function: sys_kbd_scancode`
	`trap #15`
	`; D0 contains the scancode`	

Function	`0x54`	`sys_kbd_layout`

Description	Sets the keyboard translation tables converting from scan codes to 8-bit character codes. The table provided is copied by the kernel into its own area of memory, so the memory used in the calling program's memory space may be reused after this call. Takes a pointer to the new translation tables (see below for details). If this pointer is 0, Foenix/MCP will reset its translation tables to their defaults. Returns 0 on success, or a negative number on failure.
Prototype	`void sys_kbd_layout(const char *tables)`

C Example	`char * tables = ...;`
	`// …`
	`sys_kbd_layout(tables);`

Assembly	`move.w #$54,d0 ; Function: sys_kbd_layout`
	`lea.l tables,d1 ; Pointer to table structure`
	`trap #15`

Function	`0x55`	`sys_err_message`
Description	Given a Foenix/MCP error number, return a possibly helpful error message.	
Prototype	`const char * sys_err_message(short errno)`	

C Example	`short result = sys_chan_write(…);`
	`if (result != 0) {`
	`  char * message = sys_err_message(result);`
	`  …`
	`}`	

Assembly	`move.w #$55,d0 ; Function: sys_err_message`
	`move.w …,d1 ; Error number`
	`trap #15`
	`; D0 contains the pointer to the error message`	

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

By default, Foenix/MCP supports the US standard QWERTY style keyboard, but other keyboards can be used by providing custom translation tables to map from Foenix scan codes to 8-bit character codes. These tables can be activated in the kernel by calling the sys_kbd_layout system call, providing it with the appropriate translation tables. There are eight tables that are needed, each are 128 bytes long, and they are provided as consecutive tables in the following order:

1. `UNMODIFIED`: This table maps scan codes to characters when no modifier keys are pressed.
2. `SHIFT`: This table maps scan codes when either `SHIFT` key is pressed.
3. `CTRL`: This table maps scan codes when either `CTRL` key is pressed.
4. `CTRL_SHIFT`: This table maps scan codes when `SHIFT` and `CTRL` are both pressed.
5. `CAPS`: This table maps scan codes when `CAPSLOCK` is down but `SHIFT` is not pressed.
6. `CAPS_SHIFT`: This table maps scan codes when `CAPSLOCK` is down and `SHIFT` is pressed.
7. `ALT`: This table maps scan codes when either `ALT` is pressed.
8. `ALT_SHIFT`: This table maps scan codes when `ALT` is pressed and either `SHIFT` or `CAPSLOCK` are in effect (but not both).

For keys on the right side of the keyboard (cursor keys, number pad, `INSERT`, *etc.*), `NUMLOCK` being down causes the `CAPS` or `CAPS_SHIFT` tables to be used. For those keys, CTRL and ALT will have no effect when `NUMLOCK` is down.

In the current code, character codes 0x80 through 0x95 are reserved. These codes are used to designate special keys like function keys, cursor keys, *etc*. This means that Foenix/MCP cannot directly map characters using those code points to key presses, but in the various ISO-

8859 and related standards, those code points are reserved for control codes. Also, this design choice allows for maximum flexibility in keyboard layouts, since all these keys can be mapped to whatever scan codes the user desires.

Key	Code
Cursor UP	0x86
Cursor Down	0x87
Cursor Left	0x89
Cursor Right	0x88
HOME	0x80
INS	0x81
DELETE	0x82
END	0x83
PAGE UP	0x84
PAGE DOWN	0x85
F1	0x8A
F2	0x8B
F3	0x8C
F4	0x8D
F5	0x8E
F6	0x8F
F7	0x90
F8	0x91
F9	0x92
F10	0x93
F11	0x94
F12	0x95

File Loaders

Out of the box, Foenix/MCP supports only two simple file formats executables: PGX, and PGZ. Others (particularly ELF) may be supported in the future. Since this may not meet the needs of a user, the loading and execution of files may be extended using the `sys_fsys_register_loader` system call. This call takes an extension to map to a loader, and a pointer to a loader routine.

A loader routine can be very simple: it takes a channel to read from, an address to use as an optional destination, and a pointer to a `long` variable in which to store any starting address specified by an executable file.

To actually load the file, the loader just has to read the data it needs from the already open file channel provided. If a destination address was provided by the caller (any value other than 0), the loader should use that as the destination address, otherwise it should determine from the file or its own algorithm a reasonable starting address.

Once it has finished loading the file, if it had determined that the file is executable and knows the starting address, it should store that at the location provided by the start pointer.

Finally, if all was successful, it should return a 0 to indicate success. Otherwise, it should return an appropriate error number.

Example:

```
short fsys_pgz_loader(short chan, long destination, long * start) {
      …
      *start = start_address;
      return 0;
}
```

Appendix

ANSI Terminal Codes

Foenix/MCP supports a basic subset of the VT102 ANSI terminal codes. The following escape sequences are supported:

Sequence	Name	Function
ESC[# @	ICH	Insert characters
ESC [# A	CUU	Move the cursor up
ESC [# B	CUF	Move the cursor forward
ESC [# C	CUB	Move the cursor back
ESC [# D	CUD	Move the cursor down
ESC [# J	ED	Erase the screen
ESC [# K	EL	Erase the line
ESC [# P	DCH	Delete characters
ESC [# ; # H	CUP	Set the cursor position
ESC [# m	SGR	Set the graphics rendition

For the SGR sequence, a fairly limited set of codes are currently supported, mainly to do with the color and intensity of the text:

Code	Function
0	Reset to defaults
1	High intensity / Bold
2	Low intensity / Normal
22	Normal
30 – 37	Set foreground color
40 – 47	Set background color
90 – 97	Set bright foreground color

NOTE: If the program does not want the console to interpret ANSI codes, this feature can be turned off by calling `sys_chan_ioctrl` on the console channel to be changed. A command of 0x01 will turn ANSI interpretation on, while a command of 0x02 will turn it off. When ANSI interpretation is turned off, only the core ASCII control characters will still be recognized: 0x08 (backspace), 0x09 (TAB), 0x0A (linefeed), and 0x13 (carriage return).

For key presses, the following escape codes are sent to the calling program, when one of the `sys_chan_read` functions is used on either console channel. Note that this feature is always on in the current system. Also, in the following codes, there are no actual spaces.

Key	Code
ESC	`ESC ESC`
Cursor UP	`ESC [ # A`
Cursor Down	`ESC [ # B`
Cursor Left	`ESC [ # C`
Cursor Right	`ESC [ # D`
HOME	`ESC [ # 1 ~`
INS	`ESC [ # 2 ~`
DELETE	`ESC [ # 3 ~`
END	`ESC [ # 4 ~`
PAGE UP	`ESC [ # 5 ~`
PAGE DOWN	`ESC [ # 8 ~`
F1	`ESC [ # 11 ~`
F2	`ESC [ # 12 ~`
F3	`ESC [ # 13 ~`
F4	`ESC [ # 14 ~`
F5	`ESC [ # 15 ~`
F6	`ESC [ # 17 ~`
F7	`ESC [ # 18 ~`
F8	`ESC [ # 19 ~`
F9	`ESC [ # 20 ~`
F10	`ESC [ # 21 ~`
F11	`ESC [ # 23 ~`
F12	`ESC [ # 24 ~`

The "#" in the sequences above represent an optional modifier code. If SHIFT, CTRL, or ALT is pressed with the key, the number sign is replaced with a decimal number representing a bitfield of the modifier keys, followed by a semicolon. The bit values are: SHIFT = 2, CTRL = 4, and ALT = 8.

Keyboard Scan Codes

Foenix/MCP uses the same Foenix scan codes that the original 65816 Foenix kernel used. These scan codes are derived from the standard "set 1" scan codes with modifications to get the scan codes to fit within a single byte. The base scan codes for a US QWERTY keyboard are listed below.

When a key is pressed or released, bits 0 – 6 are the same, and follow the table below. A "make" scan code is sent when the key is pressed. For make scan codes, bit 7 is clear (0). A "break" scan code is sent when a key is released. For break scan codes, bit 7 is set (1).

Example—the user presses and releases the space bar: Two scan codes will be sent. First, the make code 0x39 will be sent. Second, the break scan code of 0xB9 will be sent when the key is released.

	0_	**1_**	**2_**	**3_**	**4_**	**5_**	**6_**	**7_**
_0		Q	D	B	F6	KP2	PRSCRN	
_1	ESC	W	F	N	F7	KP3	PAUSE	
_2	! 1	E	G	M	F8	KP0	INS	
_3	@ 2	R	H	< ,	F9	KP.	HOME	
_4	# 3	T	J	> .	F10	MONITOR	PGUP	
_5	$ 4	Y	K	? /	NUMLOCK	CTX	DEL	
_6	% 5	U	L	RSHIFT	SCRLOCK	HELP	END	
_7	^ 6	I	: ;	KP*	KP7	F11	PGDN	
_8	& 7	O	" '	LALT	KP8	F12	UP	
_9	* 8	P	~ `	SPACE	KP9	RBLANK	LEFT	
_A	(9	{ [	LSHIFT	CAPS	KP-	LBLANK	DOWN	
_B	) 0	}]	| \	F1	KP4	LFNX/OS	RIGHT	
_C	_ -	ENTER	Z	F2	KP5	RALT	KP/	
_D	+ =	LCTRL	X	F3	KP6	RFNX/MEN	KPENTER	
_E	BSPACE	A	C	F4	KP+	RCTRL		
_F	TAB	S	V	F5	KP1			

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

The PGX file format is the simplest executable format. It is similar in scale to MS-DOS's COM format, or the Commodore PRG format. It consists of a single segment of data to be loaded to a specific address, where that address is also the starting address.

PGX starts with a header to identify the file and the starting address:

- The first three bytes are the ASCII codes for "PGX".
- The fourth byte is the CPU and version identification byte. Bits 0 through 3 represent the CPU code, and bits 4 through 7 represent the version of PGX supported. At the moment, there is just version 0. The CPU code can be 1 for the WDC65816, or 2 for the M680x0.
- The next four bytes (that is, bytes 4 through 7) are the address of the destination, in big-endian format (most significant byte first). This address is both the address of the

location in which to load the first byte of the data and is also the starting address for the file.

All bytes after the header are the contents of the file to be loaded into memory.

PGZ File Format

The PGZ is a more complex format that supports multiple loadable segments, but is still to be loaded in set locations in memory.

The first byte of the file is a file signature and also a version tag. If the first byte is an upper case Z, the file is a 24-bit PGZ file (*i.e.* all addresses and sizes specified in the file are 24-bits). If the file is a lower case Z, the file is a 32-bit PGZ file (all address and sizes are 32-bits in length). Note that all addresses and sizes are in big endian format (that is, most significant byte first).

After the initial byte, the remainder of the PGZ file consists of segments, one after the other. Each segment consists of two or three fields:

Field	Size	Description
`address`	3 ("Z") or 4 ("z") bytes	The target address for this segment
`size`	3 ("Z") or 4 ("z") bytes	The number of bytes in the data field
`data`	`size` bytes	The data to be loaded [optional]

For a particular segment, if the size field is 0, there will be no bytes in the data field, and the segment specifies the starting address of the entire program. At least one such segment must be present in the PGZ file for it to be executable. If more than one is present, the last one will be the one used to specify the starting address.

What is Missing

Currently, Foenix/MCP is not complete. There are still some features to implement or finalize:

- Serial port drivers
- Parallel port drivers
- MIDI port drivers
- Floppy disk driver
- COPY command
- Disk partitioning and formatting commands

- Support for partitioned drives
- Boot from floppy, hard drive, or SD card
- Setting boot sequence and default display resolution from DIP switches
- ELF loader