Foenix Toolbox Programmer's Guide

Firmware functions for the Foenix Retro System F256 computers

Toolbox Version 1.01

Peter Weingartner

November 19, 2024

Contents

Chapter 1

Introduction

The Foenix Toolbox is simple firmware package for the Foenix Retro Systems F256 computers, when they are fitted with the 65816 CPU and the chipset for the flat memory model. It can be thought of being similar to a BIOS or like the Macintosh Toolbox from the original 68000 based Macintoshes. It has three main purposes:

- Boot up the computer from a cold boot, initializing all devices
- Look for, load, and start whatever program the user wants to run. Such a program may be on the internal SD card, the external SD card, the flash memory of the F256, or a flash cartridge plugged into the expansion port. For the purposes of code development and testing, the program can be loaded into RAM under certain conditions.
- Provide a standard collection of functions to make it easier for users to write programs to run on the F256s. The functions mainly cover those areas of programming for the machine that would otherwise require a lot of uninteresting re-work or are particularly complicated.

What the Foenix Toolbox is not is a complete operating system. This is on purpose. The Toolbox is meant to help the user get a programming running, but it tries to stay out of the user's way as much as possible. What this means is:

- The Toolbox uses an absolute minimum of interrupts and hardware timers
- Although the Toolbox provides an interrupt dispatch system for the user program, user programs may take complete control over for the interrupt system and just call into the Toolbox if it needs those services affected
- There is no memory protection or really any memory management set by the Toolbox
- The Toolbox does not provide a command line interface (CLI). Although a separate project may provide a simple one, if the user wants one.
- The Toolbox does not provide a graphical user interface (GUI). If a user wants to create their own, of course they are welcome to it.

The philosophy of the Toolbox is that the owner of a Foenix computer has bought the machine to tinker with and make it do what they want it to do. The Toolbox should be there to help the user but not hinder them or restrict their freedom to do what they want with the machine.

How to Read This Manual

Well, with your eyes, naturally.

I have tried to follow some conventions in this manual.

- Each function description starts with the name of the function, followed by its address in the Toolbox jumptable.
- Function descriptions that include a version number at the end of the heading (*e.g.* "v1.01") indicate that the function was added in that version of the Toolbox and is not present in earlier builds.
- Function descriptions include a small paragraph describing the purpose and over-all usage of the function and is followed by a prototype of the function, describing the parameters and return results.
- Most functions will include simple usage examples in both C and assembly.
- C functions will usually be displayed like so: **extern int** sample_function(**char** c)
- Assembly code will usually be displayed in a typewriter style font: `label: jsl sample_function`

Copyright Information

Chapter 2

Devices

Devices on the Foenix computers fall into one of three main categories: channel devices, block devices, and files (which are really special purpose channel devices).

2.1 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 shown in table 2.1:

By default, channel 0 is open automatically to device 0 (the console) at boot time.

2.2 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. The F256K2e comes with two block devices: the internal and external SD cards (see table 2.2).

2.3 File 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

Number	Device
0	Main console (keyboard and main screen)
1	Reserved
2	Serial Port 1
3 – 5	Reserved
6	Files

Table 2.1: Channel devices

Number	Device
0	sd0—External SD card
1	sd1—Internal SD card

Table 2.2: Channel devices

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.

Paths

File and directory names follow the Unix style path conventions. That is, the forward slash (/) is used as a separator, and drives are treated as directories ("/sd", "/hd", *etc.*). FAT32 long file names are supported, but not Unicode characters. The special path names "." and ".." are supported to specify a path relative to the current path. Example paths are:

```
/sd0/hello.txt
/sd1/system/format.elf
../games/HauntedCastle/start
```

Chapter 3

Toolbox Functions

3.1 Calling Convention

All Toolbox functions are long call functions (*i.e.* using the `JSL` and `RTL` instructions) using the Calypsi "simple call" calling convention:

- left-most parameter is placed in the accumulator for 8 and 16-bit types, and the X register and the accumulator for 24 and 32 bit types (X taking the most significant bits).
- remaining parameters are pushed on to the stack in right to left order (that is, the second parameter in a call is at the top of the stack just before the `JSL`).
- 8-bit types are pushed as 16-bit values to avoid switching register sizes mid-call
- 24-bit types are pushed as 32-bit values for the same reason
- the return value is placed in the accumulator for 8 and 16-bit types, or in the X register and accumulator for 24 and 32 bit types (most significant bits in the X register).
- The caller is responsible for removing the parameters from the stack (if any) after the call returns.

Furthermore, Toolbox functions are written to save the direct page and data bank registers of the caller and to restore them before returning to the caller. This means that a user program can do whatever it likes with the direct page and data bank registers, and the Toolbox will not interfere with those settings. The Toolbox does use those registers itself, but so long as the user program does not alter the Toolbox's RAM blocks (see the memory maps), there should be no interference between the two.

The Toolbox functions are accessed through a jump table located in the F256's flash memory, starting at 0xFFE000. Each entry is four bytes long, and the address of each function is called out in their detailed descriptions below.

NOTE: Calypsi's "simple call" convention is not the fastest way to pass parameters to functions, and it is not Calypsi's only calling convention. There is also a calling convention that uses pseudo-registers in the direct page to pass parameters. Unfortunately, the rules for which parameter goes where in direct page are rather involved. While that convention is preferable when Calypsi is the only compiler involved, the Toolbox needs to allow for other development tools to be used. The stack based convention is more likely to be supported by other compilers. So speed was traded for broader compatibility.

3.2 General Functions

sys_proc_exit – 0xFFE000

This function ends the currently running program and returns control to the kernel. It takes a single short argument, which is the result code that should be passed back to the kernel. This function does not return.

void sys_proc_exit(**short** result)	
result	the code to return to the kernel

Example: C

sys_proc_exit(0); *// Quit the program with a result code of 0*

Example: Assembler

```
lda #0                ; Return code of 0
jsl sys_proc_exit     ; Quit the program
```

sys_proc_run – 0xFFE0DC

Load and run an executable binary file. This function will not return on success, since Foenix Toolbox is single tasking. Any return value will be an error condition.

Prototype	**short** sys_proc_run(**const char** * path, **int** argc, **char** * argv[])
path	the path to the executable file
argc	the number of arguments passed
argv	the array of string arguments
Returns	the return result of the program

Example: C

// Attempt to load and run /sd0/hello.pgx
// Pass the command name and "test" as the arguments

int argc = 2;
char * argv[] = {
  "hello.pgx",
  "test"
};
short result = sys_proc_run("/sd0/hello.pgx", argc, argv);

Example: Assembler

```
pei #‘argv          ; Push pointer to the arguments
pei #<>argv
pei #2              ; Push the argument count
ldx #‘path          ; Point to the path to load
lda #<>path
jsl sys_proc_run    ; Try to load and run the file

ply                 ; Clean up the stack
```

```
    ply
    ply

    ; If we get here, there was an error loading or running
    ; the file. Error number is in the accumulator

    ...

path:
    .null "/sd0/hello.pgx"

argv:
    .null "hello.pgx"
    .null "test"
```

sys_proc_set_shell – 0xFFE128 – v1.01

Set the address of a handler to be called in the event that a program calls sys_proc_exit. If the address is 0 (the default), the Toolbox will restart the machine when sys_proc_exit is called. If a non-zero address is provided, then the code at that address will be called in the same manner that program code is started. This function is provided to allow for the creation of shell programs and is not expected to be called by normal programs.

Prototype	**void** sys_proc_set_shell(uint32_t address)
Purpose	Set the address of the code that should handle a process exiting
address	the address of the handler code for proc_exit

Example: C

```
uint32_t shell_entry = ...;
sys_proc_set_shell(shell_entry);
```

Example: Assembler

```
ldx ##.word2 shell_entry
lda ##.word0 shell_entry
jsl sys_proc_set_shell
```

sys_proc_get_result – 0xFFE12C – v1.01

If a program called sys_proc_exit, this function returns the result code passed in that call.

Prototype	**int** sys_proc_get_result()
Purpose	Set the address of the code that should handle a process exiting

Example: C

```
// Get the result of the last program
int result = sys_proc_get_result();
```

Example: Assembler

```
; Get the result of the last program
jsl sys_proc_get_result
; Result code in the 16-bit accumulator
```

sys_reboot – 0xFFE124 – v1.01

Force the system to reboot.

Prototype	**void** sys_reboot()
Purpose	Force the system to reboot

Example: C

```
    // Reboot the F256
sys_reboot();
```

Example: Assembler

```
; Reboot the F256
    jsl sys_reboot
```

sys_get_info – 0xFFE020

Fill out a structure with information about the computer. This information includes the model, the CPU, the amount of memory, versions of the board and FPGAs, and what optional equipment is installed.

There is no return value.

Prototype	**void** sys_get_info(p_sys_info info)
info	pointer to a s_sys_info structure to fill out

Example: C

```
struct s_sys_info info;
sys_get_info(&info);
printf("Machine: %s\n", info.model_name);
```

Example: Assembler

```
ldx #‘info ; Point to the info structure
lda #<>info
jsl sys_get_info

; The structure at info now has data in it
```

sys_mem_get_ramtop – 0xFFE0BC

Return the limit of accessible system RAM. The address returned is the first byte of memory that user programs may not access. User programs may use any byte from the bottom of system RAM to RAMTOP - 1.

Prototype	uint32_t sys_mem_get_ramtop()
Returns	the address of the first byte of reserved system RAM

sys_mem_reserve – 0xFFE0C0

Reserve a block of memory from the top of system RAM. This call will reduce the value returned by sys_get_ramtop and will create a block of memory that user programs and the kernel will not change. The current user program can load into that memory any code or data it needs to protect after it has quit (for instance, a terminate-stay-resident code block). sys_mem_reserve returns the address of the first byte of the block reserved.

NOTE: a reserved block cannot be returned to general use accept by restarting the system.

Prototype	uint32_t sys_mem_reserve(uint32_t bytes)
bytes	the number of bytes to reserve
Returns	address of the first byte of the reserved block

Example: C

```
// Reserve a block of 256 bytes...
uint32_t my_block = sys_mem_reserve(256);
```

Example: Assembler

```
ldx #0                  ; Push the amount requested (256 bytes)
lda #256
jsl sys_mem_reserve     ; Attempt to reserve the block
stx my_block+2          ; Save the address of the block reserved
sta my_block
```

sys_time_jiffies – 0xFFE0C4

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.

Prototype	uint32_t sys_time_jiffies()
Returns	the number of jiffies since the last reset

Example: C

```
long jiffies = sys_time_jiffies();
```

Example: Assembler

```
jsl sys_time_jiffies    ; Get the time

; Jiffy count is now in X:A
```

sys_rtc_set_time – 0xFFE0C8

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_rtc_set_time(p_time time)
time	pointer to a t_time record containing the correct time

Example: C

```
struct s_time time;

// time structure is filled in with the current time

// Set the time in the RTC
sys_rtc_set_time(&time);
```

Example: Assembler

```
; time structure is filled in with the current time

...

ldx #‘time              ; Point to the time structure
lda #<>time
jsl sys_rtc_set_time    ; Set the time in the RTC
```

sys_rtc_get_time – 0xFFE0CC

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_rtc_get_time(p_time time)
time	pointer to a t_time record in which to put the current time

Example: C

```
struct s_time time;
// ...
sys_rtc_get_time(&time);
```

Example: Assembler

```
ldx #‘time              ; Point to the time structure
lda #<>time
jsl sys_rtc_get_time    ; Get the time from the RTC
```

sys_kbd_scancode – 0xFFE0D0

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	uint16_t sys_kbd_scancode()
Returns	the next scan code from the keyboard... 0 if nothing pending

Example: C

```
// Wait for a keypress
uint16_t scan_code = 0;
do {
    // Get the Foenix scan code from the keyboard
    scan_code = sys_kbd_scancode();
} while (scan_code == 0);
```

Example: Assembler

```
wait:
    jsl sys_kbd_scancode    ; Get the scan code from the keyboard
    cmp #0                  ; Keep checking until we get a keypress
    beq wait
```

sys_kbd_layout – 0xFFE0D8

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 Toolbox will reset its translation tables to their defaults.

Returns 0 on success, or a negative number on failure.

Prototype	**short** sys_kbd_layout(**const char** * tables)
tables	pointer to the keyboard translation tables
Returns	0 on success, negative number on error

3.3 Channel Functions

The channel functions provide support for channel or stream based I/O devices. Any time a device or program wants to work with data as a sequential stream of bytes or characters, that device should be a channel device. Examples of channel devices that the Toolbox supports are the console (screen and keyboard), the serial port, MIDI devices, and files open on SD cards. In the future, there may be other channel devices as well (*e.g.* network streams). Some channel devices can provide support for higher level functionality. As an example, the console channel device provides support for printing ANSI terminal codes to the screen and for reading certain key presses back as ANSI escape sequences (for instance, function key presses).

sys_chan_read_b – 0xFFE028

Read a single character from the channel. Returns the character, or 0 if none are available.

Prototype	**short** sys_chan_read_b(**short** channel)
channel	the number of the channel
Returns	the value read (if negative, error)

Example: C

```
// Read a byte from channel #0 (keyboard)
short b = sys_chan_read_b(0);
if (b >= 0) {
    // We have valid data from 0–255 in b
}
```

Example: Assembler

```
lda #0                  ; Select channel #0
jsl sys_chan_read_b     ; Read from the channel
bit #$ffff              ; If negative...
bmi error               ; Process an error

; We have valid data in A
```

sys_chan_read – 0xFFE02C

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)
channel	the number of the channel
buffer	the buffer into which to copy the channel data
size	the size of the buffer.
Returns	number of bytes read, any negative number is an error code

Example: C

```
char buffer[80];
short n = sys_chan_read(0, buffer, 80);
if (n >= 0) {
    // We correctly read n bytes into the buffer
    } else {
    // We have an error
}
```

Example: Assembler

```
pei #80             ; Push the size of the buffer

pei #‘buffer        ; Push the address of the buffer
pei #<>buffer

lda #0              ; Select channel #0

jsl sys_chan_read   ; Try to read the bytes from the channel

ply                 ; Clean up the stack
ply
ply
```

```
bit #$ffff          ; If result is negative...
bmi error           ; Go to process the error

sta n               ; Otherwise: save the number of bytes read
```

sys_chan_readline – 0xFFE030

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_readline(**short** channel, **unsigned char** * buffer, **short** size)
channel	the number of the channel
buffer	the buffer into which to copy the channel data
size	the size of the buffer
Returns	number of bytes read, any negative number is an error code

Example: C

short c = ...; *// The channel number*
unsigned char buffer[128];
short n = sys_chan_read_line(c, buffer, 128);

Example: Assembler

```
pei #128                ; Push the size of the buffer
pei #‘buffer            ; Push the pointer to the buffer
pei #<>buffer
lda c                   ; Set the channel number to read from
jsl sys_chan_read_line  ; Attempt to read a line from the console

ply                     ; Clean up the stack
ply
ply

sta n                   ; Save the number of bytes read
```

sys_chan_write_b – 0xFFE034

Write a single byte to the channel.

Prototype	**short** sys_chan_write_b(**short** channel, uint8_t b)
channel	the number of the channel
b	the byte to write
Returns	0 on success, a negative value on error

Example: C

// Write ‘a’ to the console
short result = sys_chan_write_b(0, 'a');

Example: Assembler

```
    pei #'a'                ; Push 'a' as the b parameter
    lda #0                  ; Select the console (channel #0)
    jsl sys_chan_write_b    ; Write the character to the console
    ply                     ; Clean up the stack
```

sys_chan_write – 0xFFE038

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, **const** uint8_t * buffer, **short** size)
channel	the number of the channel
buffer	
size	
Returns	number of bytes written, any negative number is an error code

Example: C

```
char * message = 'Hello, world!\n';
short n = sys_chan_write(0, message, strlen(message));
```

Example: Assembler

```
    pei #15               ; Push the size of the buffer
    pei #‘message         ; Push the pointer to the message
    pei #<>message
    lda #0                ; Select the console (channel #0)
    jsl sys_chan_write    ; Write the buffer to the console

    ply                   ; Clean up the stack
    ply
    ply

    ; ...
message:
    .null "Hello, world!", 13, 10
```

sys_chan_status – 0xFFE03C

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)
channel	the number of the channel
Returns	the status of the device

Example: C

```
// Check the status of the file_in channel
short status = sys_chan_status(file_in);
if (status & 0x01) {
    // We have reached end of file
}
```

Example: Assembler

```
lda file_in
jsl sys_chan_status
and #$01
beq have_data
; We have reached end of file
```

sys_chan_flush – 0xFFE040

Ensure any pending writes to a channel are completed.

Prototype	**short** sys_chan_flush(**short** channel)
channel	the number of the channel
Returns	0 on success, any negative number is an error code

Example: C

```
short file_out = ...; // Channel number
sys_chan_flush(file_out); // Flush the channel
```

Example: Assembler

```
lda file_out          ; Channel number
jsl sys_chan_flush    ; Flush the channel
```

sys_chan_seek – 0xFFE044

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)
channel	the number of the channel
position	the position of the cursor
base	whether the position is absolute or relative to the current position
Returns	0 = success, a negative number is an error.

Example: C

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

Example: Assembler

```
pei #1              ; Move is relative
pei #-10            ; Move back by 10 bytes
lda c               ; Select the channel
jsl sys_chan_seek   ; Move the channel cursor

ply                 ; Clean up the stack
ply
```

sys_chan_ioctrl – 0xFFE048

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. The `buffer` and `size` parameters provide additional data to the commands, what exactly needs to go in them (if anything) is command-specific. Some commands require data in the buffer, and others do not.

Prototype	**short** sys_chan_ioctrl(**short** channel, **short** command, uint8_t * buffer, **short** size)
channel	the number of the channel
command	the number of the command to send
buffer	pointer to bytes of additional data for the command
size	the size of the buffer
Returns	0 on success, any negative number is an error code

Example: C

short c = ...; *// The channel number*
short cmd = ...; *// The command*
short r = sys_chan_ioctrl(c, cmd, 0, 0); *// Send simple command*

Example: Assembler

```
pei #0                  ; Push 0 for the size
pei #0                  ; Push the null pointer for the buffer
pei #0
lda cmd                 ; Push the command
pha
lda c
jsl sys_chan_ioctrl     ; Issue the command

ply                     ; Clean up the stack
ply
ply
ply
```

sys_chan_open – 0xFFE04C

Open a channel device for reading or writing given: the number of the device, the path to the resource on the device (if any), and the access mode. The access mode is a bit field:

- 0x01: Open for reading
- 0x02: Open for writing

- 0x03: Open for reading and writing

Prototype	**short** sys_chan_open(**short** dev, **const char** ∗ path, **short** mode)
dev	the device number to have a channel opened
path	a ”path” describing how the device is to be open
mode	s the device to be read, written, both
Returns	the number of the channel opened, negative number on error

Example: C

// Serial port: 9600bps, 8−data bits, 1 stop bit, no parity
short chan = sys_chan_open(2, `"9600,8,1,N"`, 3);

Example: Assembler

```
    pei #3                ; Mode: Read & Write
    pei #‘path            ; Pointer to the path
    pei #<>path
    lda #2                ; Device #2 (UART)
    jsl sys_chan_open     ; Open the channel to the UART

    ply                   ; Clean up the stack
    ply
    ply

    ; ...

path:
    .null "9600,8,1,N"
```

sys_chan_close − 0xFFE050

Close a channel that was previously open by sys_chan_open.

Prototype	**short** sys_chan_close(**short** chan)
chan	the number of the channel to close
Returns	nothing useful

Example: C

short c = ...; *// The channel number*
sys_chan_close(c); *// Close the channel*

Example: Assembler

```
    lda c               ; Get the channel number
    jsl sys_chan_close  ; Close the channel
```

sys_chan_swap – 0xFFE054

Swaps two channels, given their IDs. If before the call, channel ID `channel1` refers to the file "hello.txt", and channel ID `channel2` is the console, then after the call, `channel1` is the console, and `channel2` is the open file "hello.txt". Any context for the channels is preserved (for instance, the position of the file cursor in an open file).

Prototype	**short** sys_chan_swap(**short** channel1, **short** channel2)
channel1	the ID of one of the channels
channel2	the ID of the other channel
Returns	0 on success, any other number is an error

sys_chan_device – 0xFFE058

Given a channel ID (the only parameter), return the ID of the device associated with the channel. The channel must be open.

Prototype	**short** sys_chan_device(**short** channel)
channel	the ID of the channel to query
Returns	the ID of the device associated with the channel, negative number for error

3.4 Block Device Functions

The block device functions provide low-level support for access to block-based storage devices. The main operations on block devices are reading a block of data from a device (given the device number and the address of the block to read), and writing a block of data to the device. These functions are used by the driver to the FatFS library to provide FAT32 file based access to those block devices. Currently, for the F256, this support is limited to SD cards. The F256jr and F256K have just the one SD card, but the F256K2e has an internal and an external SD card. Future machines might provide additional block devices (*e.g.* floppy drives or hard drives), and if someone wanted to build some sort of block device for the F256, a driver to support it could be added to the Toolbox to add those devices to the FAT32 file support.

sys_bdev_register – 0xFFE060

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(p_dev_block device)
device	pointer to the description of the device to register
Returns	0 on success, negative number on error

sys_bdev_read – 0xFFE064

Read a block from a block device. Returns the number of bytes read.

Prototype	**short** sys_bdev_read(**short** dev, **long** lba, uint8_t * buffer, **short** size)
dev	the number of the device
lba	the logical block address of the block to read
buffer	the buffer into which to copy the block data
size	the size of the buffer.
Returns	number of bytes read, any negative number is an error code

Example: C

```
unsigned char buffer[512];

// Read the MBR of the interal SD card
short n = sys_bdev_read(BDEV_SD1, 0, buffer, 512);
```

Example: Assembler

```
pei #512 ; Push the size of the buffer
pei #‘buffer ; Push the pointer to the read buffer
pei #<>buffer
pei #0 ; Push LBA = 0
pei #0
lda #1 ; The device number for the internal SD

jsl sys_bdev_read ; Read sector 0

ply ; Clean up the stack
ply
ply
ply
ply
```

sys_bdev_write – 0xFFE068

Write a block from a block device. Returns the number of bytes written.

Prototype	**short** sys_bdev_write(**short** dev, **long** lba, **const** uint8_t * buffer, **short** size)
dev	the number of the device
lba	the logical block address of the block to write
buffer	the buffer containing the data to write
size	the size of the buffer.
Returns	number of bytes written, any negative number is an error code

Example: C

```
unsigned char buffer[512];

// Fill in the buffer with data...

// Write the MBR of the interal SD card
short n = sys_bdev_write(BDEV_SD1, 0, buffer, 512);
```

Example: Assembler

```
pei #512 ; Push the size of the buffer
pei #‘buffer ; Push the pointer to the read buffer
pei #<>buffer
pei #0 ; Push LBA = 0
pei #0
lda #1 ; The device number for the internal SD
```

```
jsl sys_bdev_write ; Write sector 0

ply ; Clean up the stack
ply
ply
ply
ply
```

sys_bdev_status – 0xFFE06C

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)
dev	the number of the device
Returns	the status of the device

Example: C

short bdev = ...; *// The device number*
short status = sys_bdev_status(bdev);

Example: Assembler

```
lda bdev            ; Load the device number
jsl sys_bdev_flush  ; Attempt to flush pending writes

; Status is in the accumulator
```

sys_bdev_flush – 0xFFE070

Ensure any pending writes to a block device are completed.

Prototype	**short** sys_bdev_flush(**short** dev)
dev	the number of the device
Returns	0 on success, any negative number is an error code

Example: C

short bdev = ...; *// The device number*
sys_bdev_flush(bdev);

Example: Assembler

```
lda bdev            ; Load the device number
jsl sys_bdev_flush  ; Attempt to flush pending writes
```

sys_bdev_ioctrl – 0xFFE074

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.

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** dev, **short** command, uint8_t * buffer, **short** size)
dev	the number of the device
command	the number of the command to send
buffer	pointer to bytes of additional data for the command
size	the size of the buffer
Returns	0 on success, any negative number is an error code

Example: C

short dev = ...; *// The device number*
short cmd = ...; *// The command*
short r = sys_bdev_ioctrl(dev, cmd, 0, 0); *// Send simple command*

Example: Assembler

```
pei #0                  ; Push buffer size of 0
pei #0                  ; Push null pointer for buffer
pei #0
lda cmd                 ; Push the command number
pha
lda bdev                ; Select the block device

jsl sys_bdev_ioctrl     ; Send the command

ply                     ; Clean up the stack
ply
ply
```

3.5 File System Functions

sys_fsys_open – 0xFFE078

Attempt to open a file in the file system for reading or writing. Returns a channel number associated with the file. If the returned number is negative, there was an error opening the file.

The mode parameter indicates how the file should be open and is a bit field, where each bit has a separate meaning:

- 0x01: Read

- 0x02: Write
- 0x04: Create if new
- 0x08: Always create
- 0x10: Open file if it exists, otherwise create
- 0x20: Open file for appending

Prototype	**short** sys_fsys_open(**const char** * path, **short** mode)
path	the ASCIIZ string containing the path to the file.
mode	the mode (e.g. r/w/create)
Returns	the channel ID for the open file (negative if error)

Example: C

```
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
}
```

Example: Assembler

```
    pei #1              ; Push

    ldx #‘path          ; Point to the path
    lda #<>path

    jsl sys_fsys_open   ; Try to open the file

    ply                 ; Clean up the stack

    bit #$ffff          ; Check to see if we opened the file
    bmi error

    ; File is open for reading

error:

    ; There was an error
    ; The error number is in the accumulator

path:
    .null "hello.txt"
```

sys_fsys_close – 0xFFE07C

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	**short** sys_fsys_close(**short** fd)
fd	the channel ID for the file
Returns	0 on success, negative number on failure

Example: C

```
short chan = sys_fsys_open(...);
// ...
sys_fsys_close(chan);
```

Example: Assembler

```
lda chan
jsl sys_fsys_close
```

sys_fsys_opendir – 0xFFE080

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)
path	the path to the directory to open
Returns	the handle to the directory if ¿= 0. An error if ¡ 0

Example: C

```
short dir = sys_fsys_opendir("/sd0/System");
if (dir > 0) {
  // dir can be used for reading the directory entries
} else {
  // There was an error... error number in dir
}
```

Example: Assembler

```
    ldx #‘path              ; Point to the path
    lda #<>path

    jsl sys_fsys_opendir    ; Try to open the directory

    bit #$ffff              ; Check to see if we opened the directory
    bmi error

    ; Directory is open for reading

error:

    ; There was an error
    ; The error number is in the accumulator

path:
    .null "/sd0/System"
```

sys_fsys_closedir – 0xFFE084

Close a previously open directory, given its number.

Prototype	**short** sys_fsys_closedir(**short** dir)
dir	the directory handle to close
Returns	0 on success, negative number on error

Example: C

```
short dir = ... // Number of the directory to close
sys_fsys_closedir(dir);
```

Example: Assembler

```
lda dir                 ; Get the number of the directory to close
jsl sys_fsys_closedir   ; Close the directory
```

sys_fsys_readdir – 0xFFE088

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, p_file_info file)
dir	the handle of the open directory
file	pointer to the t_file_info structure to fill out.
Returns	0 on success, negative number on failure

Example: C

```
short dir = sys_fsys_opendir("/sd0/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
}
```

Example: Assembler

```
ldx #‘path              ; Point to the path
lda #<>path

jsl sys_fsys_opendir    ; Try to open the directory

bit #$ffff              ; Check to see if we opened the directory
bmi error
```

```
    ; Directory is open for reading

    sta dir                 ; Save the directory number

    pei #‘file_info         ; Set the pointer to the file info
    pei #<>file_info

    ; Directory number is already in A

    jsl sys_fsys_readdir    ; Try to read from the directory

    ply                     ; Clean up the stack
    ply

    bit #$ffff              ; If result is <0, there is an error
    bmi error

    ; Entry is loaded into structure at file_info

error:

    ; There was an error
    ; The error number is in the accumulator

path:
    .null "/sd0/System"
file_info:
    .dstruct s_file_info
```

sys_fsys_findfirst – 0xFFE08C

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, p_file_info file)
path	the path to the directory to search
pattern	the file name pattern to search for
file	pointer to the t_file_info structure to fill out
Returns	error if negative, otherwise the directory handle to use for subsequent calls

Example: C

```
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...
}
```

Example: Assembler

```
    pei #‘file_info         ; Point to the file_info
    pei #<>file_info

    pei #‘pattern           ; Point to the search pattern
    pei #<>pattern

    ldx #‘path              ; Point to the directory to search
    lda #<>path

    jsl sys_fsys_findfirst  ; Try to find the first match

    ply                     ; Clean up the stack
    ply
    ply
    ply

    bit #$ffff              ; Check to see if error (negative)
    bmi error

    ; File info should contain the first match

error:

    ; There was an error

file_info:
    .dstruct s_file_info
pattern:
    .null "*.pgx"
path:
    .null "/sd0/System"
```

sys_fsys_findnext – 0xFFE090

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_findnext(**short** dir, p_file_info file)
dir	the handle to the directory (returned by fsys_findfirst) to search
file	pointer to the t_file_info structure to fill out
Returns	0 on success, error if negative

Example: C

```
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...
  }
```

Example: Assembler

```
    pei #‘file_info         ; Point to the file_info
    pei #<>file_info

    pei #‘pattern           ; Point to the search pattern
    pei #<>pattern

    ldx #‘path              ; Point to the directory to search
    lda #<>path

    jsl sys_fsys_findfirst  ; Try to find the first match

    ply                     ; Clean up the stack
    ply
    ply
    ply

    bit #$ffff              ; Check to see if error (negative)
    bmi error

    sta dir                 ; Save the open directory number

    ; File info should contain the first match

    ; ...

    ; Find the next

    pei #‘file_info         ; Point to the file_info
    pei #<>file_info

    lda dir                 ; Get the directory number

    jsl sys_fsys_findnext   ; Try to find the next match

    ply                     ; Clean up the stack
    ply

    bit #$ffff              ; Check to see if error
    bmi error

    ; File info should contain next match

error:
```

```
    ; There was an error

file_info:
    .dstruct s_file_info
pattern:
    .null "*.pgx"
path:
    .null "/sd0/System"
```

sys_fsys_get_label – 0xFFE094

Get the label of a volume.

Prototype	**short** sys_fsys_get_label(**const char** * path, **char** * label)
path	path to the drive
label	buffer that will hold the label... should be at least 35 bytes
Returns	0 on success, error if negative

Example: C

```
char label[64];
short result = sys_fsys_get_label("/sd0", label);
```

Example: Assembler

```
    pei #‘label             ; Point to the label buffer
    pei #<>label

    ldx #‘path              ; Point to the path of the drive
    lda #<>path

    jsl sys_fsys_get_label  ; Attempt to get the label

    ply                     ; Clean the stack
    ply

    bit #$ffff              ; Check for an error
    bmi error

    ; We should have the label filled

error:

    ; There was an error

path:
    .null "/sd0"
label:
    .fill 64
```

sys_fsys_set_label – 0xFFE098

Set the label of a volume.

Prototype	**short** sys_fsys_set_label(**short** drive, **const char** * label)
drive	drive number
label	buffer that holds the label
Returns	0 on success, error if negative

Example: C

```
short result = sys_fsys_set_label(0, "FNXSD0");
```

Example: Assembler

```
    pei #‘label             ; Point to the label
    pei #<>label

    lda #0                  ; Set the volume number

    jsl sys_fsys_set_label  ; Attempt to set the label

    ply                     ; Clean the stack
    ply

    bit #$ffff              ; Check for an error
    bmi error

    ; We should have the label updated

error:

    ; There was an error

label:
    .null "FNXSD0"
```

sys_fsys_mkdir – 0xFFE09C

Create a directory.

Prototype	**short** sys_fsys_mkdir(**const char** * path)
path	the path of the directory to create.
Returns	0 on success, negative number on failure.

Example: C

```
short result = sys_fsys_mkdir("/sd0/Samples");
```

Example: Assembler

```
    ldx #‘path
    lda #<>path

    jsl sys_fsys_mkdir  ; Attempt to create the directory

    bit #$ffff          ; Check for an error
    bmi error

    ; Directory should be created

error:

    ; There was an error

path:
    .null "/sd0/Samples"
```

sys_fsys_delete – 0xFFE0A0

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)
path	the path of the file or directory to delete.
Returns	0 on success, negative number on failure.

Example: C

```
short result = sys_fsys_delete("/sd0/test.txt");
```

Example: Assembler

```
    ldx #‘path          ; Point to the path to delete
    lda #<>path

    jsl sys_fsys_delete ; Try to delete the file
    bit #$ffff
    bmi error

    ; File was deleted...

error:

    ; There was an error

path:
    .null "/sd0/test.txt"
```

sys_fsys_rename – 0xFFE0A4

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_path)
old_path	he current path to the file
new_path	the new path for the file
Returns	0 on success, negative number on failure.

Example: C

```
short result = sys_fsys_rename("/sd0/test.txt", "doc.txt");
```

Example: Assembler

```
    pei #‘new_path      ; Push the pointer to the new name
    pei #<>new_path

    ldx #‘old_path      ; Point to the original file name
    lda #<>old_path

    jsl sys_fsys_rename ; Try to rename the file the file

    ply                 ; Clean up the stack
    ply

    bit #$ffff          ; Check for an error
    bmi error

    ; File was named...

error:

    ; There was an error

old_path:
    .null "/sd0/test.txt"
new_path:
    .null "doc.txt"
```

sys_fsys_load – 0xFFE0B0

Load a file into memory. This function can either load a file into a specific address provided by the caller, or to the loading address specified in the file (for executable files). For executable files, the function will also return the starting address specified in the file.

Prototype	**short** sys_fsys_load(**const char** * path, uint32_t destination, uint32_t * start)
path	the path to the file to load
destination	the destination address (0 for use file's address)
start	pointer to the long variable to fill with the starting address
Returns	0 on success, negative number on error

Example: C

```
uint32_t start;
short result = sys_fsys_load("hello.pgx", 0, &start);
```

Example: Assembler

```
    pei #‘start         ; Push the pointer to the start variable
    pei #<>start

    pei #0              ; Push 0 to leave a load address unspecified
    pei #0

    ldx #‘path          ; Point to the file name
    lda #<>path

    jsl sys_fsys_load   ; Try to rename the file the file

    ply                 ; Clean up the stack
    ply
    ply
    ply

    bit #$ffff          ; Check for an error
    bmi error

    ; File was loaded

error:

    ; There was an error

path:
    .null "hello.pgx"
start:
    .dword ?
```

sys_fsys_register_loader – 0xFFE0B4

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`.

On success, returns 0. If 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)
extension	the file extension to map to
loader	pointer to the file load routine to add
Returns	0 on success, negative number on error

Example: C

```
short foo_loader(short chan, uint32_t destination, uint32_t * 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);
```

sys_fsys_stat – 0xFFE0B8

Check to see if a file is present. The `s_file_info` structure will be populated if the file is found. Returns 0 on success or a negative number on an error.

Prototype	**short** sys_fsys_stat(**const char** * path, p_file_info file)
path	the path to the file to check
file	pointer to a file info record to fill in, if the file is found.
Returns	0 on success, negative number on error

Example: C

```
s_file_info file_info;
short result = sys_fsys_stat("/sd0/fnxboot.pgx", &file_info);
```

3.6 Text System Functions

Many programs will likely use the console channel device and the `sys_chan_write` call to print most things to the screen, but there are certain operations that a program might need to carry out that do not fit well with the channel device. Also, programs may want lower level control over the text screen. These functions are part of the text block of functions.

Functions in this block allow a program to find out what kinds of text modes the screen is capable of, change the size of the display text, manipulate the cursor and the border of the screen, and even change the font and display colors. Additionally, the text functions also provide for "regions" which may be used to create simple text windows—smaller rectangles on the screen where printing will go, leaving other portions of the text screen unchanged.

The F256 supports only the one screen, but the text system functions were written with support for multiple screens in mind. All text functions take a screen number. For the F256 as of the time of this writing, that number will always be 0. If at some point, an F256 with multiscreen support is created or a graphics expansion card is produced, that additional screen could be supported by the Toolbox with the addition of a text mode driver.

sys_txt_get_capabilities – 0xFFE0E4

Gets the description of a screen's capabilities. The capabilities are returned as a pointer to a structure that provides a bit field of the various modes supported, a listing of different font sizes that are supported (the F256 currently supports only 8 by 8 fonts), and a listing of different screen resolutions supported by the screen.

```
struct s_txt_capabilities {
        short number; /* The unique ID of the screen */
        short supported_modes; /* The display modes supported on this screen */
        short font_size_count; /* The number of supported font sizes */
        p_extent font_sizes; /* Pointer to a list of t_extent listing all supported font sizes */
        short resolution_count; /* The number of supported display resolutions */
        p_extent resolutions; /* Pointer to a list of t_extent for supported resolutions (in pixels) */
}
```

Prototype	**const** p_txt_capabilities sys_txt_get_capabilities(**short** screen)
screen	the number of the text device
Returns	a pointer to the read-only description (0 on error)

sys_txt_set_mode – 0xFFE0E4

Set the display mode of the screen. There are five basic modes supported which are indicated by the five flags:

- `TXT_MODE_TEXT`—Render base text
- `TXT_MODE_BITMAP`—Render bitmap graphics
- `TXT_MODE_TILE`—Render tilesets
- `TXT_MODE_SPRITE`—Render sprites
- `TXT_MODE_SLEEP`—Puts the monitor in power-saving mode by turning off the sync signals

These flags are returned in the `supported_modes` field of the `t_txt_capabilities` structure returned by `sys_txt_get_caps`, and they may be combined to mix the different rendering engines if supported by the hardware (for instance, `TXT_MODE_TEXT | TXT_MODE_SPRITE` would combine text and sprites). `TXT_MODE_SLEEP` will over-ride all the other modes.

The result of turning off all the mode flags is system dependent, but should result in a blank screen without putting the monitor into sleep mode.

Returns 0 on success, any other number means the mode was invalid for the screen or the screen was invalid.

Prototype	**short** sys_txt_set_mode(**short** screen, **short** mode)
screen	the number of the text device
mode	a bit field of desired display mode options
Returns	0 on success, any other number means the mode is invalid for the screen

Example: C

```
// Set screen 0 to text and tiles
short result = sys_txt_set_mode(0, TXT_MODE_TEXT | TXT_MODE_TILE);
if (result) {
  // Handle the error
}
```

Example: Assembler

```
pei #(TXT_MODE_TEXT | TXT_MODE_TILE)  ; Turn on text and tile modes
lda #0                                ; On screen 0

jsl sys_txt_set_mode

ply                                   ; Clean up the stack
```

sys_txt_set_resolution – 0xFFE0E8

Set the display resolution of the screen. The width and height must match one of the resolutions listed in the screen's capabilities.

Prototype	**short** sys_txt_set_resolution(**short** screen, **short** width, **short** height)
screen	the number of the text device
width	the desired horizontal resolution in pixels
height	the desired vertical resolution in pixels
Returns	0 on success, any other number means the mode is invalid for the screen

Example: C

```
// Set screen 0 resolution to (320, 240)
short result = sys_txt_set_resolution(0, 320, 240);
```

Example: Assembler

```
    pei #240 ; Resolution: 320 by 240
pei #320
    lda #0                          ; On screen 0

    jsl sys_txt_set_mode

    ply                               ; Clean up the stack
ply
```

sys_txt_set_xy – 0xFFE0F0

Sets the position of the cursor on the screen.

The call takes the number of the screen and the character row (y) and column (x) of the cursor. The cursor positions are specified relative to the origin of the current region set on the screen, so (0, 0) will be the origin of the region, (0, 1) will be the character position right below the origin, and so on.

Prototype	**void** sys_txt_set_xy(**short** screen, **short** x, **short** y)
screen	the number of the text device
x	the column for the cursor
y	the row for the cursor

Example: C

```
// Move the cursor to the home position in the current region
sys_txt_set_xy(0, 0, 0);
```

Example: Assembler

```
    pei #0              ; Set y = 0
    pei #0              ; Set x = 0
    lda #0              ; Screen 0

    jsl sys_txt_set_xy  ; Set the position

    ply                 ; Clean up the stack
    ply
```

sys_txt_get_xy – 0xFFE0F4

Gets the position of the text cursor, given two parameters: the screen number, and the pointer to a `t_point`. The cursor position will be copied into the `t_point` object.

Prototype	**void** sys_txt_get_xy(**short** screen, p_point position)
screen	the number of the text device
position	pointer to a t_point record to fill out

Example: C

```
// Get the cursor position
t_point position;
sys_txt_get_xy(0, &position);
```

Example: Assembler

```
    pei #‘position      ; Pointer to the position object
    pei #<>position

    lda #0              ; Screen 0

    jsl sys_txt_get_xy  ; Get the cursor position

    ply                 ; Clean up the stack
    ply

position:
    .dstruct s_position
```

sys_txt_get_region – 0xFFE0F8

Gets the origin and size of the rectangle describing the current region.

The call takes a screen number and a pointer to a `t_rect` structure to fill out with the current information. Returns 0 on success, any other number is an error.

Prototype	**short** sys_txt_get_region(**short** screen, p_rect region)
screen	the number of the text device
region	pointer to a t_rect describing the rectangular region (using character cells for size and size)
Returns	0 on success, any other number means the region was invalid

Example: C

```
// Get the current region
t_rect region;
sys_txt_get_region(0, &region);
```

Example: Assembler

```
    pei #‘region            ; Pointer to the position object
    pei #<>region

    lda #0                  ; Screen 0

    jsl sys_txt_get_region  ; Get the current region

    ply                     ; Clean up the stack
    ply

region:
    .dstruct s_rect
```

sys_txt_set_region – 0xFFE0FC

Sets the rectangular region of the screen that will be used for all subsequent printing, scrolling, and filling. This call takes the screen number and a pointer to a `t_rect` structure containing the origin (upper-left corner) and the size (width and height) of the region. These values are specified in character cells, with (0, 0) being the upper-left corner of the screen. If the size of the rectangle is 0 (width = height = 0), then the region will be the full screen.

Returns 0 on success, any other number is an error.

Prototype	**short** sys_txt_set_region(**short** screen, p_rect region)
screen	the number of the text device
region	pointer to a t_rect describing the rectangular region (using character cells for size and size)
Returns	0 on success, any other number means the region was invalid

Example: C

```
// Set the region to a 5x5 panel in the upper left
t_rect region;
region.origin.x = 0;
region.origin.y = 0;
region.size.width = 5;
region.size.height = 5;
short result = sys_txt_set_region(0, &region);
if (result) {
  // Handle the error
}
```

Example: Assembler

```
pei #‘region            ; Pointer to the position object
pei #<>region

lda #0                  ; Screen 0

jsl sys_txt_set_region  ; Set the new region

ply                     ; Clean up the stack
ply

region:
    .word 0, 0, 5, 5
```

sys_txt_set_color – 0xFFE100

Set the foreground and background color to use for subsequent prints to the screen. Takes the screen number and the color indexes for foreground and background colors (0 – 15). Returns 0 on success, any other number is an error.

Prototype	**void** sys_txt_set_color(**short** screen, **unsigned char** foreground, **unsigned char** background)
screen	the number of the text device
foreground	the Text LUT index of the new current foreground color (0 – 15)
background	the Text LUT index of the new current background color (0 – 15)

Example: C

```
// Set the text color to cyan on black (in standard colors)
sys_txt_set_color(0, 6, 0);
```

Example: Assembler

```
pei #0                  ; Background to black
pei #6                  ; Foreground to cyan

lda #0                  ; Screen 0

jsl sys_txt_set_color   ; Set the text color

ply                     ; Clean up the stack
ply
```

sys_txt_get_color – 0xFFE104

Gets the current foreground and background color settings. Takes the screen number and two pointers: one for the foreground color value, and one for the background color value. Returns 0 on success, any other number is an error.

Prototype	**void** sys_txt_get_color(**short** screen, **unsigned char** * foreground, **unsigned char** * background)
screen	the number of the text device
foreground	the Text LUT index of the new current foreground color (0 - 15)
background	the Text LUT index of the new current background color (0 - 15)

Example: C

```
// Gets the text color for the screen
short foreground = 0;
short background = 0;
if (sys_txt_get_color(0, &foreground, &background)) {
  // Handle error
}
```

Example: Assembler

```
pei #‘background        ; Push address of background variable
pei #<>background

pei #‘foreground        ; Push address of foreground variable
pei #<>foreground

lda #0                  ; Screen 0

jsl sys_txt_get_color   ; Get the color

ply                     ; Clean up the stack
ply
ply
```

```
    ply

    ; ...

foreground:
    .word ?
background:
    .word ?
```

sys_txt_set_cursor – 0xFFE108

Set the appearance of the text mode cursor.

Prototype	**void** sys_txt_set_cursor(**short** screen, **short** enable, **short** rate, **char** c)
screen	the screen number
enable	0 to hide, any other number to make visible
rate	the blink rate for the cursor (0=1s, 1=0.5s, 2=0.25s, 3=1/5s)
char	the character in the current font to use as a cursor

Example: C

```
// Set the cursor on screen 0
    // Visible, blink period 0.25s, character @
sys_txt_set_cursor(0, 1, 2, '@');
```

Example: Assembler

```
pei #'@'; Cursor character @
pei #2 ; Blink period 0.25s
    pei #1      ; Show the cursor
    lda #0      ; Screen 0

    jsl sys_txt_set_cursor

    ply         ; Clean the stack
ply
ply
```

sys_txt_set_cursor_visible – 0xFFE10C

Sets the visibility of the text cursor.

Prototype	**void** sys_txt_set_cursor_visible(**short** screen, **short** is_visible)
screen	the screen number
is_visible	TRUE if the cursor should be visible, FALSE (0) otherwise

Example: C

```
// Hide the cursor on screen 0
sys_txt_set_cursor_visible(0, 0);
```

Example: Assembler

```
pei #0      ; Hide the cursor
lda #0      ; Screen 0

jsl sys_txt_set_cursor_visible

ply         ; Clean the stack
```

sys_txt_set_font – 0xFFE110

Set the font to be used in text mode on the screen. Takes the screen number, the width and height of the characters (in pixels), and a pointer to the actual font data. Returns 0 on success, any other number means the screen is invalid, or the font size is invalid.

NOTE: the font size must be listed in the `font_sizes field` of the `t_txt_capabilities` structure returned by `sys_txt_get_caps`.

Prototype	**short** sys_txt_set_font(**short** screen, **short** width, **short** height, **unsigned char** ∗ data)
screen	the number of the text device
width	width of a character in pixels
height	of a character in pixels
data	pointer to the raw font data to be loaded

Example: C

```
// Set the font of screen 0 to an 8x8 font
unsigned char * font_data;
font_data = ...;
short result = sys_txt_set_font(0, 8, 8, font_data);
if (result) {
  // Handle error
}
```

Example: Assembler

```
pei #‘font_data         ; Push pointer to the font data
pei #<>font_data

pei #8                  ; Push size of 8x8
pei #8

lda #0                  ; Screen 0

jsl sys_txt_set_font    ; Set the font

ply                     ; Clean up the stack
ply
ply
ply
```

sys_txt_setsizes – 0xFFE0EC

Sets the text screen device driver to the current screen geometry, based on the display resolution and border size. If a program changes the border or display resolution on its own but still needs to use the Toolbox console or text routines to display text, it should call this function to have the Toolbox recalculate the screen geometry.

Prototype	**void** sys_txt_setsizes(**short** screen)
screen	the number of the text device

Example: C

```
// Recalculate geometry of screen 0
sys_txt_setsizes(0);
```

Example: Assembler

```
lda #0
jsl sys_txt_setsizes
```

sys_txt_get_sizes – 0xFFE114

Gets the size of the screen in total pixels (not taking the border into consideration) and visible characters (taking the border into account).

NOTE: `text_size` and `pixel_size` can be null (0), in which case that structure will not be filled out, so you do not have to provide a `t_extent` for a measurement you do not need.

Prototype	**void** sys_txt_get_sizes(**short** screen, p_extent text_size, p_extent pixel_size)
screen	the screen number
text_size	the size of the screen in visible characters (may be null)
pixel_size	the size of the screen in pixels (may be null)

Example: C

```
// Hide the cursor on screen 0
t_rect text_matrix;
t_rect pixel_matrix;
sys_txt_get_sizes(0, &text_matrix, &pixel_matrix);
```

Example: Assembler

```
pei #‘pixel_matrix      ; Push pointer to pixel extent
pei #<>pixel_matrix

pei #‘text_matrix       ; Push pointer to text extent
pei #<>text_matrix

lda #0

jsl sys_txt_get_sizes   ; Get the sizes

ply                     ; Clean up the stack
ply
```

```
    ply
    ply

    ; ...

pixel_matrix:               ; Holds size of screen in pixels
    .dstruct s_extent
text_matrix:                ; Holds size of screen in characters
    .dstruct s_extent
```

sys_txt_set_border – 0xFFE118

Sets the size of the border around the screen. Takes the number of the screen and the size of the border width and height. In this context, width is the width of the left and right borders taken separately, and height is the height of the top and bottom borders. So if width is 8 and height is 16, 32 lines will be taken up by the top and bottom borders together, and 16 columns will be taken up by the left and right borders.

NOTE: if the width and height of the borders are 0, the border will be disabled.

Prototype	**void** sys_txt_set_border(**short** screen, **short** width, **short** height)
screen	the number of the text device
width	the horizontal size of one side of the border (0 – 32 pixels)
height	the vertical size of one side of the border (0 – 32 pixels)

Example: C

```
// Set the border on screen 0: width of 16, height of 8
sys_txt_set_border(0, 16, 8);
```

Example: Assembler

```
    pei #8                  ; 8 pixels vertically
    pei #16                 ; 16 pixels horizontally

    lda #0                  ; Screen 0

    jsl sys_txt_set_border  ; Set the border size

    ply                     ; Clean up the stack
    ply
```

sys_txt_set_border_color – 0xFFE11C

Set the color of the border, using red, green, and blue components (which may go from 0 to 255).

Prototype	**void** sys_txt_set_border_color(**short** screen, **unsigned char** red, **unsigned char** green, **unsigned char** blue)
screen	the number of the text device
red	the red component of the color (0 - 255)
green	the green component of the color (0 - 255)
blue	the blue component of the color (0 - 255)

Example: C

```
// Set the border of screen 0 to dark blue
sys_txt_set_border_color(0, 0, 0, 128);
```

Example: Assembler

```
pei #128                ; Push blue
pei #0                  ; Push green
pei #0                  ; Push red

lda #0                  ; Screen 0

jsl sys_txt_set_border  ; Set the border color

ply                     ; Clean up the stack
ply
ply
```

sys_txt_put – 0xFFE120

Print a character to the screen.

NOTE: No this function does not interpret ANSI terminal codes and will display the characters corresponding to those bytes on the screen. To print with ANSI terminal code support, use the console channel device.

Prototype	**void** sys_txt_put(**short** screen, **char** c)
screen	the number of the text device
c	the character to print

Example: C

```
// Print 'A' to the screen
sys_txt_put(0, 'A');
```

Example: Assembler

```
pei #'A'        ; Push the character
lda #0          ; Screen 0

jsl sys_txt_put ; Print the character

ply             ; Clean up the stack
```

sys_txt_print – 0xFFE124

Print a null-terminated ASCII string to the screen.

NOTE: No this function does not interpret ANSI terminal codes and will display the characters corresponding to those bytes on the screen. To print with ANSI terminal code support, use the console channel device.

Prototype	**void** sys_txt_print(**short** screen, **const char** * message)
screen	the number of the text device
message	the ASCII Z string to print

Example: C

```
// Print a message to the screen
sys_txt_print(0, "Hello, Foenix!\n");
```

Example: Assembler

```
    pei #‘message       ; Push pointer to message
    pei #<>message

    lda #0              ; Screen 0

    jsl sys_txt_print   ; Print the message

    ply                 ; Clean up the stack
    ply

    ; ...

message:
    .null "Hello, Foenix!\n"
```

3.7 Interrupt Functions

Interrupts in the Toolbox are managed at a device level. The F256 includes an interrupt controller which assigns a different interrupt to each device that can raise an interrupt. The interrupt controller provides for separate masking and interrupt flags for each device interrupt. The Toolbox allows programs to register an interrupt handler for the specific device-level interrupt the program needs to handle. That handler is just a regular subroutine (it should not be coded to return with an `RTI` instruction). The Toolbox will take care of checking the various interrupt controller registers to determine which interrupts are currently pending and will call the associated interrupt handler automatically. The Toolbox will also take care to save register states to avoid interfering with the currently running program.

On the F256, interrupts are arranged into three groups, with each interrupt getting a bit within one of the groups in each of the control registers for that group. For instance, the start of frame interrupt (vertical blank interrupt) is the least significant bit of group 0. The Foenix Toolbox, on the other hand, assigns a single number to each interrupt and internally maps that number to the appropriate group and bit. When enabling or disabling an interrupt or when registering an interrupt handler, it is this internal interrupt number that is used.[1]

In addition to controlling interrupts at a device level (*e.g.* the serial port interrupt), the Toolbox also has routines to allow a program to enable or disable IRQ processing at the CPU level. The int_enable_all, int_disable_all, and int_restore_all, functions work at the CPU level and do not affect the mask bits in the F256's interrupt controller. The int_enable, and int_disable functions work at the level of the individual device interrupt in the interrupt controller.

Overriding Toolbox Interrupt Handling

The Toolbox takes care of the details of the F256's interrupt controller for user programs, but that comes at the cost of overhead. Many programs will need more efficient control over interrupts and will prefer to manage interrupts for themselves, without the Toolbox intervening.

[1]The interrupt number assignments may seem arbitrary, but they are actually just inherited from Foenix MCP and the A2560s, where the interrupt number is essentially just the group and bit numbers packed into an 8-bit value.

Hardware Group	Hardware Bit	Toolbox Number	Purpose
0	0x01	0x00	Start of frame
	0x02	0x01	Start of line
	0x04	0x10	PS/2 Keyboard
	0x08	0x12	PS/2 Mouse
	0x10	0x18	Timer 0
	0x20	0x19	Timer 1
	0x40	—	Reserved
	0x80	0x06	External Expansion
1	0x01	0x13	Serial Port
	0x02	—	Reserved
	0x04	—	Reserved
	0x08	—	Reserved
	0x10	0x1F	Real Time Clock
	0x20	0x1D	VIA 0
	0x40	0x1E	VIA 1 (F256K)
	0x80	0x22	SD Card
2	0x01	—	Reserved
	0x02	—	Reserved
	0x04	—	Reserved
	0x08	—	Reserved
	0x10	—	Reserved
	0x20	—	Reserved
	0x40	—	Reserved
	0x80	0x21	SD Card Inserted

Table 3.1: F256 Interrupt Assignments

This is perfectly fine.

The 65816's interrupt vectors are stored in RAM (the F256 populates them based on data in flash at boot time). Programs can write their own interupt handler addresses to those vectors to take over the handling of interrupts. The only problem with this is that the Toolbox depends upon interrupts to handle keystrokes. By default, the Toolbox uses the start-of-frame (SOF) interrupt on the F256k and F256k2 to periodically trigger a scan of the keyboard matrix and the PS/2 interrupt on the F256jr. If a program takes over interrupts but still needs to use the Toolbox's keyboard routines, it will need to call sys_kbd_handle_irq function periodically (see below) to trigger the matrix scan or check the PS/2 port for keystrokes.

sys_int_enable_all – 0xFFE004

This function enables all maskable interrupts at the CPU level. It returns a system-dependent code that represents the previous level of interrupt masking. Returns a machine dependent representation of the CPU interrupt mask state that can be used to restore the state later. Note: this does not change the mask status of interrupts in the machine's interrupt controller, it just changes if the CPU ignores IRQs or not.

Prototype	**short** sys_int_enable_all()

Example: C

```
// Enable processing of IRQs
short state = sys_int_enable_all();
```

Example: Assembler

```
; Enable processing of IRQs
jsl sys_int_enable_all
```

sys_int_disable_all – 0xFFE008

This function disables all maskable interrupts at the CPU level. It returns a system-dependent code that represents the previous level of interrupt masking. Returns a machine dependent representation of the CPU interrupt mask state that can be used to restore the state later. Note: this does not change the mask status of interrupts in the machine's interrupt controller, it just changes if the CPU ignores IRQs or not.

Prototype	**short** sys_int_disable_all()

Example: C

// Disable processing of IRQs
short state = sys_int_disable_all();

Example: Assembler

```
; Disable processing of IRQs
jsl sys_int_disable_all
```

sys_int_restore_all – 0xFFE00C

Restores

Prototype	**void** sys_int_restore_all(**short** state)
state	the machine dependent CPU interrupt state to restore

Example: C

// Restore state of IRQ processing after enabling/disabling all
short state = ...;

sys_int_restore_all(state);

Example: Assembler

```
; Restore state of IRQ processing after enabling/disabling all
lda state
jsl sys_int_restore_all
```

sys_int_disable – 0xFFE010

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(**unsigned short** n)
n	the number of the interrupt: n[7..4] = group number, n[3..0] = individual number.

Example: C

```
// Disable the start–of–frame interrupt
sys_int_disable(INT_SOF_A);
```

Example: Assembler

```
    lda #INT_SOF_A        ; Enable the start-of-frame interrupt
    jsl sys_int_disable
```

sys_int_enable – 0xFFE014

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(**unsigned short** n)
n	the number of the interrupt

Example: C

```
// Enable the start–of–frame interrupt
sys_int_enable(INT_SOF_A);
```

Example: Assembler

```
    lda #INT_SOF_A        ; Enable the start-of-frame interrupt
    jsl sys_int_enable
```

sys_int_register – 0xFFE018

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(**unsigned short** n, p_int_handler handler)
n	the number of the interrupt
handler	pointer to the interrupt handler to register
Returns	the pointer to the previous interrupt handler

Example: C

```
// Handler for the start–of–frame interrupt
// Must be a far sub–routine (returns through RTL)
__attribute__((far)) void sof_handler() {
        // Interrupt handler code here...
}

// Register a handler for the start–of–frame interrupt
p_int_handler old = sys_int_register(INT_SOF_A, sof_handler);
```

Example: Assembler

```
; Handler for the start-of-frame interrupt
; Must be a far sub-routine (returns through RTL)
sof_handler:
    ; Handler code here...
    rtl

    ; Code to register the handler...
    pei #‘sof_handler       ; push pointer to sof_handler
    pei #<>sof_handler

    lda #INT_SOF_A          ; A = the number for the SOF_A interrupt

    jsl sys_int_register

    ply                     ; Clean up the stack
    ply

    sta old                 ; Save the pointer to the old handler
    stx old+2
```

sys_int_pending – 0xFFE01C

Query an interrupt to see if it is pending in the interrupt controller. NOTE: User programs will probably never need to use this call, since it is handled by the Toolbox itself.

Prototype	**short** sys_int_pending(**unsigned short** n)
n	the number of the interrupt: n[7..4] = group number, n[3..0] = individual number.
Returns	non-zero if interrupt n is pending, 0 if not

Example: C

```
// Check to see if start–of–frame interrupt is pending
short is_pending = sys_int_pending(INT_SOF_A);
if (is_pending) {
        // The interrupt has not yet been acknowledged
}
```

Example: Assembler

```
    ; Check to see if the start-of-frame interrupt is pending
    lda #INT_SOF_A
    jsl sys_int_pending
    cmp #0
    beq sof_not_pending

    ; Code for when start-of-frame is pending

sof_not_pending:
```

sys_int_clear – 0xFFE024

This function acknowledges the processing of an interrupt by clearing its pending flag in the interrupt controller. NOTE: User programs will probably never need to use this call, since it is handled by the Toolbox itself.

Prototype	**void** sys_int_clear(**unsigned short** n)
n	the number of the interrupt: n[7..4] = group number, n[3..0] = individual number.

Example: C

```
// Acknowledge the processing of the start-of-frame interrupt
sys_int_clear(INT_SOF_A);
```

Example: Assembler

```
; Acknowledge the processing of the start-of-frame interrupt
lda #INT_SOF_A
jsl sys_int_clear
```

sys_kbd_handle_irq – 0xFFE120 – v1.01

This function causes the keyboard processing code to try to process keystrokes. In the case of the mechanical keyboard on the F256k, it will scan the keyboard matrix and process any changes to the key positions. In the case of the optical keyboard on the F256k2, it will check the optical keyboard queue for any pending keystrokes. In the case of the PS/2 keyboard on the F256jr, it will check the PS/2 device for pending bytes.

Internally, this function is called during the SOF interrupt on the F256k and F256k2 and is called in response to a PS/2 interrupt on the F256jr. The routine is exposed through the jumptable in case a program wants to take over the interrupt processing but still wants the Toolbox to interpret keystrokes. To get characters from the console device or to get keyboard scancodes, this function must be called periodically, as it is this function that interprets keypresses and queues up scancodes and console bytes.

Prototype	**void** sys_kbd_handle_irq()
Purpose	Handle an IRQ to query the keyboard

Example: C

```
// Look for and process pending keystrokes
sys_kbd_handle_irq();
```

Example: Assembler

```
; Look for and process pending keystrokes
jsl sys_kbd_handle_irq
```

3.8 IEC Bus Functions

This collection of functions expose low-level access to the IEC bus (aka Commodore serial bus). The functions allow a caller to issue TALK, UNTALK, LISTEN, UNLISTEN commands to devices on the bus. The functions also allow a caller to send and receive data bytes and to check for the end-of-interaction byte ("EOI") when listening to a talking device.

Currently, these functions do not provide higher level access to devices, like managing files and directories on disk drives. That level of access may be added in the future.

As of version 1.01, these functions should be considered experimental and quite probably buggy.

sys_iecll_ioinit – 0xFFE130 – v1.01

Prototype	**short** sys_iecll_ioinit()
Purpose	Initialize the IEC interface
Returns	short 0 on success, -1 if no devices found

sys_iecll_talk – 0xFFE140 – v1.01

Prototype	**short** sys_iecll_talk(uint8_t device)
Purpose	Send a TALK command to a device
device	the number of the device to become the talker
Returns	short

sys_iecll_talk_sa – 0xFFE144 – v1.01

Prototype	**short** sys_iecll_talk_sa(uint8_t secondary_address)
Purpose	Send the secondary address to the TALK command, release ATN, and turn around control of the bus
secondary_address	the secondary address to send
Returns	short

sys_iecll_listen – 0xFFE14C – v1.01

Prototype	**short** sys_iecll_listen(uint8_t device)
Purpose	Send a LISTEN command to a device
device	
Returns	short the number of the device to become the listener

sys_iecll_listen_sa – 0xFFE150 – v1.01

Prototype	**short** sys_iecll_listen_sa(uint8_t secondary_address)
Purpose	Send the secondary address to the LISTEN command and release ATN
secondary_address	the secondary address to send
Returns	short

sys_iecll_untalk – 0xFFE148 – v1.01

Prototype	**void** sys_iecll_untalk()
Purpose	Send the UNTALK command to all devices and drop ATN

sys_iecll_unlisten – 0xFFE154 – v1.01

Prototype	**void** sys_iecll_unlisten()
Purpose	Send the UNLISTEN command to all devices

sys_iecll_in – 0xFFE134 – v1.01

Prototype	uint8_t sys_iecll_in()
Purpose	Try to get a byte from the IEC bus
Returns	uint8_t the byte read

sys_iecll_eoi – 0xFFE138 – v1.01

Prototype	**short** sys_iecll_eoi()
Purpose	Check to see if the last byte read was an EOI byte
Returns	short 0 if not EOI, any other number if EOI

sys_iecll_out – 0xFFE13C – v1.01

Prototype	**void** sys_iecll_out(uint8_t byte)
Purpose	Send a byte to the IEC bus. Actually sends the previous byte and queues the current byte.
byte	the byte to send

sys_iecll_reset – 0xFFE158 – v1.01

Prototype	**void** sys_iecll_reset()
Purpose	Assert and release the reset line on the IEC bus

Chapter 4

F256 Toolbox Boot Process

The Toolbox does not really "do" anything once it has finished initializing the hardware. There is no CLI to use to enter commands, no GUI to use, not even a machine language monitor to fall into. To actually do something, the user needs to have executable code somewhere on one of the SD cards or in the cartridge. When the Toolbox finishes initializing the system, it will look in memory and in the SD cards for executable code. If it finds it, it will load and run the code. The exact process is fairly involved.

To begin with, the Toolbox has the notion of a "boot source," which is really just a storage device that can hold the executable code. There are several boot sources: the internal SD card, the external SD card, a flash cartridge inserted into the expansion port, the parts of flash memory the Toolbox does not occupy, and finally (under certain conditions) the system RAM. To include the system RAM as a boot source, DIP switch 1 must be in the "ON" position. The Toolbox will scan each of the boot sources in a priority order.

1. If DIP switch 1 is ON, the system RAM is checked first.
2. If a flash cartridge is present, it is checked next.
3. If an SD card is present in the external slot, it is checked next.
4. The internal SD card is checked next.
5. Finally, the flash memory is checked.

For the two SD cards, the Toolbox is looking for an executable file in the root directory of the card—either `fnxboot.pgx` or `fnxboot.pgz`. For RAM, flash memory, and the expansion cartridge, the Toolbox is looking for a special header that contains a signature and specifies the starting address to run. In RAM, the entire memory from 0x00:0000 to the top of system RAM will be checked on 8KB alignment boundaries (that is, the header should be at 0x00:0000, or 0x00:2000, or 0x00:4000, *etc.*). For the cartridge, it should be at the start of the cartridge's memory (0xF4:0000). For the flash memory, it should be at the start of the flash memory (0xF8:0000)[1].

The header for the executable code can be described with this C structure:

```
struct boot_record_s {
        char signature1;    // Needs to be $f8
        char signature2;    // Needs to be $16
        uint8_t version;    // Currently $00
        uint32_t start_address;   // Address to start executing (in little-endian format of the 65816)
        uint32_t icon_address;    // Address of an icon to show (32x32 sprite data, use 0 for no icon)
        uint32_t clut_address;    // Address of the palette for the icon in Vicky format (0 to use the default)
        const char * name;    // A display name/command word for the program (not currently used)
}
```

[1] This may change in future.

- The header starts with the hexadecimal value 0xF816 in big-endian format (for Foenix 65816).
- The third byte is the version number, which is currently 0.
- The next four bytes are the address of the code to start executing (really a 24-bit pointer packed into 32-bits for convenience).
- The next four bytes are a pointer to the raw Vicky sprite bitmap data for an icon to show in the boot screen. If no icon is needed, this should be 0.
- The next four bytes are a pointer to the Vicky graphics CLUT data for the icon, to be copied into graphics CLUT 2. Again, if no CLUT needs to be provided, this should be 0. An icon may be provided without a CLUT, in which case the default graphics CLUT will be used.[2]
- The last four bytes are a pointer to a null-terminated ASCII string providing the name of this code. Currently, this is not being used, but it is intended to be the equivalent of a file name in terms of readability and possibly being used to select from multiple options, in later versions of the Toolbox.

[2]The default CLUT is the "Google" color palette from the Aseprite package.

Chapter 5

Extending the System

Foenix Toolbox 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 Toolbox does not have all the features that absolutely everyone will want. Therefore, there are four main ways that the user can extend the capabilities of Foenix Toolbox: channel device drivers, block device drivers, keyboard translation tables, and file loaders.

5.1 Channel Device Drivers

Channel device drivers provide the functions needed by Foenix Toolbox 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:

```
typedef struct s_dev_chan {
    short number; // The number of the device (assigned by registration)
    char * name; // The name of the device
    short (*init)(); // Initialize the device
    short (*open)(p_channel chan, const uint8_t * path, short mode); // -- open a channel for the device
    short (*close)(p_channel chan); // Called when a channel is closed
    short (*read)(p_channel chan, uint8_t * buf, short size); // Read a a buf from the device
    short (*readline)(p_channel chan, uint8_t * buf, short size); // Read a line of text from the device
    short (*read_b)(p_channel chan); // -- read a single uint8_t from the device
    short (*write)(p_channel chan, const uint8_t * buf, short size); // Write a buf to the device
    short (*write_b)(p_channel chan, const uint8_t b) ; // Write a single uint8_t to the device
    short (*status)(p_channel chan); // Get the status of the device
    short (*flush)(p_channel chan); // Ensure that any pending writes to teh device have been completed
    short (*seek)(p_channel chan, long position, short base); // Attempt to move the "cursor" in the channel
    short (*ioctrl)(p_channel chan, short cmd, uint8_t * buf, short size); // Issue a command to the device
} t_dev_chan, *p_dev_chan;
```

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
    uint8_t data[32]; // A block of state data that the channel code can use for its own purposes
};
```

To implement a driver for a new channel device, all 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 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 housekeeping functions needed before the channel is returned to the kernel's pool. In particular, if the device buffers write operations, any writes that are still pending should be written to the device.

5.2 Block Device Drivers

Block device drivers are used by Foenix Toolbox 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 similarly to channel device drivers by implementing the functions needed by Foenix Toolbox 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 (assigned by registration)
    char * name;               // The name of the device
    void * data;               // Device-specific data block
    short (*init)(struct s_dev_block *);          // Initialize the device
    short (*read)(struct s_dev_block *, long lba, uint8_t * buf, short size); // Read a block from the device
    short (*write)(struct s_dev_block *, long lba, const uint8_t * buf, short size); // Write a block to the device
    short (*status)(struct s_dev_block *);        // Get the status of the device
    short (*flush)(struct s_dev_block *);         // Completes any pending writes to the device
    // Issue a command to the device
    short (*ioctrl)(struct s_dev_block *, short cmd, unsigned char * buf, short size);
}
```

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.

5.3 Keyboard Translation Tables

By default, Foenix Toolbox 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 Toolbox 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. See table 5.1 for the detailed mapping.

5.4 File Loaders

Out of the box, Foenix Toolbox supports only two simple file formats executables: PGX, PGZ, and ELF. Others 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.

Key	Code
Cursor UP	0x86
Cursor Down	0x89
Cursor Left	0x87
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

Table 5.1: Special Key Codes

Example File Loader

```
short fsys_pgz_loader(short chan, long destination, long * start) {
    // Use the channel calls to read the input file and load into memory
    // ...

    // If sucessful and found a start address:
    *start = start_address;
    return 0;
}
```

Chapter 6

Appendix

6.1 Console IOCTRL Commands

The console channel driver supports the following commands for `sys_chan_ioctrl`. None of these IOCTRL commands require a buffer, so passing NULL for the buffer and 0 for the size is recommended.

1. `CON_IOCTRL_ANSI_ON`: Turn on ANSI escape sequence processing.[1]
2. `CON_IOCTRL_ANSI_OFF`: Turn off ANSI escape sequence processing.
3. `CON_IOCTRL_ECHO_ON`: Turn on character echoing for `sys_chan_read_b`.[2]
4. `CON_IOCTRL_ECHO_OFF`: Turn off character echoing for `sys_chan_read_b`.
5. `CON_IOCTRL_BREAK`: Check to see if the user has pressed the BREAK key sequence.[3]
6. `CON_IOCTRL_CURS_ON`: The text mode cursor should be visible.[4]
7. `CON_IOCTRL_CURS_OFF`: The text mode cursor should be hidden

6.2 ANSI Terminal Codes

Foenix Toolbox supports a basic subset of the VT102 ANSI terminal codes. The following escape sequences are shown in table 6.1.

For the SGR sequence, a fairly limited set of codes are currently supported, mainly to do with the color and intensity of the text (see table: 6.2).

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, escape codes (see table 6.3) are sent to the calling program, when one of the `sys_chan_read` functions is used on the channel. Note that this feature is always on in the current system. Also, in the following codes, there are no actual spaces.

[1] ANSI processing is on by default.

[2] Echo is on be default.

[3] `sys_chan_ioctrl` will return a non-zero value if the BREAK key was pressed, and 0 if not. On all Foenix machines, CTRL-C (code point 0x03) will be treated as the BREAK key. On the A2560K, the combination Foenix-ESC will also work as the BREAK key. On the F256K, the RUN/STOP key will be treated as the BREAK key.

[4] Cursor is on by default.

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

Table 6.1: ANSI Terminal Codes

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
100 – 107	Set bright background color

Table 6.2: ANSI SGR Codes

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 ; # ~`

Table 6.3: ANSI Key Codes

6.3 Keyboard Scan codes

Foenix Toolbox uses the same Foenix scan codes that the original 65816 Foenix kernel and Foenix/MCP 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.

	0x00	0x10	0x20	0x30	0x40	0x50	0x60
0x00		Q	D	B	F6	KP2	PRSN
0x01	ESC	W	F	N	F7	KP3	PAUSE
0x02	1	E	G	M	F8	KP0	INS
0x03	2	R	H	<	F9	KP.	HOME
0x04	3	T	J	>	F10		PGUP
0x05	4	Y	K	/	NUMLCK		DEL
0x06	5	U	L	R SHFT	SCRLCK		END
0x07	6	I	;	KP*	KP7	F11	PGDN
0x08	7	O	"	L ALT	KP8	F12	UP
0x09	8	P	~	SPACE	KP9		LEFT
0x0A	9	[	L SHFT	CAPS	KP-		DOWN
0x0B	0	]	\	F1	KP4	L FNX	RIGHT
0x0C	-	ENTER	Z	F2	KP5	R ALT	KP/
0x0D	=	L CTRL	X	F3	KP6	R FNX	KPENTER
0x0E	BKSP	A	C	F4	KP+	R CTRL	
0x0F	TAB	S	V	F5	KP1		

Table 6.4: Keyboard Scan codes

6.4 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

System Information

```
/*
 * Structure to describe the hardware
 */
struct s_sys_info {
    uint16_t mcp_version;   /* Current version of the MCP kernel */
    uint16_t mcp_rev;   /* Current revision, or sub-version of the MCP kernel */
    uint16_t mcp_build;   /* Current vuild # of the MCP kernel */
    uint16_t model;   /* Code to say what model of machine this is */
    uint16_t sub_model;  /* 0x00 = PB, 0x01 = LB, 0x02 = CUBE */
    const char * model_name; /* Human readable name of the model of the computer */
    uint16_t cpu;   /* Code to say which CPU is running */
    const char * cpu_name; /* Human readable name for the CPU */
    uint32_t cpu_clock_khz;  /* Speed of the CPU clock in KHz */
    unsigned long fpga_date; /* YYYYMMDD */
    uint16_t fpga_model;   /* FPGA model number */
    uint16_t fpga_version;   /* FPGA version */
    uint16_t fpga_subver;   /* FPGA sub-version */
    uint32_t system_ram_size;  /* The number of bytes of system RAM on the board */
    bool has_floppy; /* TRUE if the board has a floppy drive installed */
    bool has_hard_drive; /* TRUE if the board has a PATA device installed */
    bool has_expansion_card; /* TRUE if an expansion card is installed on the device */
    bool has_ethernet; /* TRUE if an ethernet port is present */
    uint16_t screens;   /* How many screens are on this computer */
};
```

Model and CPU IDs

The numbers listed in table 6.5 are used to distinguish between the different models of Foenix computers. The numbers listed in table 6.6 are used to distinguish between the different CPUs. Both the machine and CPU IDs are also used by the Toolbox's make file.

Screen Information

There are several structures defined to provide information about the text screen and to be used in controlling various aspects of the text screen.

Model	Number
C256 FMX	0
C256 U	1
C256 GenX	4
C256 U+	5
A2560 U+	6
A2560 X	7
A2560 U	9
A2560 K	11

Table 6.5: Foenix Model IDs

CPU	Number
M68SEC000	0
M68020	1
M68EC020	2
M68030	3
M68EC030	4
M68040	5
M68040V	6
ME68EC040	7
i486DX 50	8
i468DX 60	9
i468DX4	10

Table 6.6: Foenix CPU IDs

Mode	Number
TXT_MODE_TEXT	0x0001
TXT_MODE_BITMAP	0x0002
TXT_MODE_SPRITE	0x0004
TXT_MODE_TILE	0x0008
TXT_MODE_SLEEP	0x0010

Table 6.7: Toolbox Screen Mode Flags

```
/*
 * Structure to specify the size of a rectangle
 */
typedef struct s_extent {
    short width; /**< The width of the region */
    short height; /**< The height of the region */
} t_extent, *p_extent;

/*
 * Structure to specify the location of a point on the screen
 */
typedef struct s_point {
    short x; /**< The column of the point */
    short y; /**< The row of the point */
} t_point, *p_point;

/*
 * Structure to specify a rectangular area on the screen
 */
typedef struct s_rect {
    t_point origin; /**< The upper-left corner of the rectangle */
    t_extent size; /**< The size of the rectangle */
} t_rect, *p_rect;
```

The capabilities of the screen are listed in the text capabilities structure. These capabilities include the supported display modes on the screen (as a bit field, the values of which are listed in table 6.7), the number and sizes of the fonts supported, and the display resolutions supported.

```
/*
* Structure to specify the capabilities of a screen's text driver
*/
typedef struct s_txt_capabilities {
    short number; /**< The unique ID of the screen */
    short supported_modes; /**< The display modes supported on this screen */
    short font_size_count; /**< The number of supported font sizes */
    p_extent font_sizes; /**< Pointer to a list of t_extent listing all supported font sizes (in pixels) */
    short resolution_count; /**< The number of supported display resolutions */
    p_extent resolutions; /**< Pointer to a list of t_extent listing all supported display resolutions (in pixels) */
} t_txt_capabilities, *p_txt_capabilities;
```

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]

Table 6.8: PGZ File Segments

6.5 Foenix Executable File Formats

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 little-endian format (that is, least-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, shown in table 6.8.

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.

6.6 Memory Map

The Foenix Toolbox uses different sections of both flash and RAM memory to provide its functions. The memory map in table 6.9 marks the major areas and pseudo-registers for the Toolbox. The table also marks out sections of memory that are reserved for future use by the Toolbox (that is, sections not currently used but which may be used in the future), and areas specifically reserved for user programs (called out in boldface).

Roughly speaking, all bank 0 RAM below 0x00:D000 and all RAM from bank 1 through bank 6 are free for user programs to do with as they wish. The stack is allocated and shared between the user programs and

Address	Purpose
0x00:0000 - 0x00:CFFF	**User Memory**
0x00:D000 - 0x00:DFFF	Toolbox Low Memory
0x00:E000 - 0x00:EDEA	Reserved
0x00:EDEB - 0x00:FDEB	Stack
0x00:FDEC - 0x00:FDEF	User IRQ Vector
0x00:FDF4 - 0x00:FDF7	User NMI Vector
0x00:FDF8 - 0x00:FDFF	Reserved
0x00:FE00 - 0x00:FEFF	Toolbox Direct Page
0x00:FF00 - 0x00:FFFF	Toolbox Bootstrap Shadow
0x01:0000 - 0x06:FFFF	**User Memory**
0x07:0000 - 0x07:FFFF	Toolbox Memory
0xFC:0000 - 0xFF:DFFF	Toolbox Firmware
0xFF:E000 - 0xFF:FEFF	Toolbox Jump Table
0xFF:FF00 - 0xFF:FFFF	Toolbox Bootstrap and Vectors

Table 6.9: Toolbox Memory Usage

the toolbox functions. If a program wishes to move the stack for some reason, this should be safe enough to do, although some Toolbox functions might fail, if they use too much of the stack the user program reserved.

Of course, if a program takes complete control over the F256 and does not require any toolbox functions, the program is free to do whatever it requires with memory. This memory map is only of importance for programs that need the Toolbox functions to work correctly.