

# **Foenix Toolbox Programmer's Guide**

Firmware functions for the Foenix Retro System F256 computers

**Peter Weingartner**

September 28, 2024



# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Devices</b>	<b>7</b>
<b>3</b>	<b>Toolbox Functions</b>	<b>9</b>
3.1	Calling Convention . . . . .	9
3.2	General Functions . . . . .	10
3.3	Channel Functions . . . . .	14
3.4	Block Device Functions . . . . .	21
3.5	File System Functions . . . . .	24
3.6	Text System Functions . . . . .	26
3.7	Interrupt Functions . . . . .	28



# 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.

## Copyright Information

Foenix Toolbox and all code except for the FatFS file system library are published under the BSD 3 Clause License. Please see the source code for the license terms. The Foenix Toolbox file system is provided by the FatFS file system, which is covered under its own license. For information about the author of FatFS and its license terms, please see the Foenix Toolbox source code.

# 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).

### 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.

### 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).

### 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	Reserved
5	MIDI Ports
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. 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 so as 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 jumptable 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 command line. It takes a single short argument, which is the result code that should be passed back to the kernel. This function does not return.

<b>void sys_proc_exit(short result)</b>	
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 – 0xFFE0D8

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	<b>short sys_proc_run(const char * path, int argc, char * argv[])</b>
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\_get\_info – 0xFFE01C

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	<b>void</b> 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 – 0xFFE0B8

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 – 0xFFE0BC

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 except by restarting the system.

Prototype	<code>uint32_t sys_mem_reserve(uint32_t bytes)</code>
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` – `0xFFE0C0`

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 coarse level, but it should not be used when precision is required.

Prototype	<code>uint32_t sys_time_jiffies()</code>
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` – `0xFFE0C4`

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	<code>void sys_rtc_set_time(p_time time)</code>
time	pointer to a <code>t_time</code> 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 – 0xFFE0C8

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	<b>void</b> 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 – 0xFFE0CC

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 – 0xFFE0D4

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

## 3.3 Channel Functions

#### Example: C

#### Example: Assembler

### sys\_chan\_read\_b – 0xFFE024

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

Prototype	<b>short</b> sys_chan_read_b( <b>short</b> 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 – 0xFFE028

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	<b>short</b> sys_chan_read( <b>short</b> channel, <b>unsigned char</b> * buffer, <b>short</b> 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 – 0xFFE02C

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	<b>short</b> sys_chan_readline( <b>short</b> channel, <b>unsigned char</b> * buffer, <b>short</b> 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 – 0xFFE030

Write a single byte to the channel.

Prototype	<b>short</b> sys_chan_write_b( <b>short</b> channel, <b>uint8_t</b> 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 – 0xFFE034

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	<b>short</b> sys_chan_write( <b>short</b> channel, <b>const</b> uint8_t * buffer, <b>short</b> 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 – 0xFFE038

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	<b>short</b> sys_chan_status( <b>short</b> 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 – 0xFFE03C

Ensure any pending writes to a channel are completed.

Prototype	<b>short</b> sys_chan_flush( <b>short</b> 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 – 0xFFE040

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	<b>short</b> sys_chan_seek( <b>short</b> channel, <b>long</b> position, <b>short</b> 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\_ioctl – 0xFFE044

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	<b>short</b> sys_chan_ioctl( <b>short</b> channel, <b>short</b> command, uint8_t * buffer, <b>short</b> 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_ioctl(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_ioctl ; Issue the command

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

### sys\_chan\_open – 0xFFE048

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 bitfield:

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

- 0x03: Open for reading and writing

Prototype	<b>short</b> sys_chan_open( <b>short</b> dev, <b>const char</b> * path, <b>short</b> 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 – 0xFFE04C

Close a channel that was previously open by sys\_chan\_open.

Prototype	<b>short</b> sys_chan_close( <b>short</b> 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 – 0xFFE050

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	<b>short</b> sys_chan_swap( <b>short</b> channel1, <b>short</b> 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 – 0xFFE054

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

Prototype	<b>short</b> sys_chan_device( <b>short</b> 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

### sys\_bdev\_register – 0xFFE05C

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	<b>short</b> sys_bdev_register(p_dev_block device)
device	pointer to the description of the device to register
Returns	0 on succes, negative number on error

### sys\_bdev\_read – 0xFFE060

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

Prototype	<b>short</b> sys_bdev_read( <b>short</b> dev, <b>long</b> lba, uint8_t * buffer, <b>short</b> 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 intenal 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 – 0xFFE064

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

Prototype	<b>short</b> sys_bdev_write( <b>short</b> dev, <b>long</b> lba, <b>const</b> uint8_t * buffer, <b>short</b> 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 internal 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 – 0xFFE068

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	<b>short</b> sys_bdev_status( <b>short</b> 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 – 0xFFE06C

Ensure any pending writes to a block device are completed.

Prototype	<b>short</b> sys_bdev_flush( <b>short</b> 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\_ioctl – 0xFFE070

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	<b>short</b> sys_bdev_ioctl( <b>short</b> dev, <b>short</b> command, uint8_t * buffer, <b>short</b> 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_ioctl(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_ioctl    ; Send the command

ply
ply
ply

```

## 3.5 File System Functions

### sys\_fsopen – 0xFFE074

Prototype	<b>short</b> sys_fsopen( <b>const char</b> * path, <b>short</b> 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)

### sys\_fs\_close – 0xFFE078

Prototype	<b>short</b> sys_fs_close( <b>short</b> fd)
fd	the channel ID for the file
Returns	0 on success, negative number on failure

### sys\_fs\_opendir – 0xFFE07C

Prototype	<b>short</b> sys_fs_opendir( <b>const char</b> * path)
path	the path to the directory to open
Returns	the handle to the directory if i = 0. An error if i ≠ 0



### sys\_fsyz\_closedir – 0xFFE080

Prototype	<b>short</b> sys_fsyz_closedir( <b>short</b> dir)
dir	the directory handle to close
Returns	0 on success, negative number on error

### sys\_fsyz\_readdir – 0xFFE084

Prototype	<b>short</b> sys_fsyz_readdir( <b>short</b> 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

### sys\_fsyz\_findfirst – 0xFFE088

Prototype	<b>short</b> sys_fsyz_findfirst( <b>const char</b> * path, <b>const char</b> * 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	the directory handle to use for subsequent calls if i= 0, error if negative

### sys\_fsyz\_findnext – 0xFFE08C

Prototype	<b>short</b> sys_fsyz_findnext( <b>short</b> dir, p_file.info file)
dir	the handle to the directory (returned by fsyz_findfirst) to search
file	pointer to the t_file.info structure to fill out
Returns	0 on success, error if negative

### sys\_fsyz\_get\_label – 0xFFE090

Prototype	<b>short</b> sys_fsyz_get_label( <b>const char</b> * path, <b>char</b> * 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

### sys\_fsyz\_set\_label – 0xFFE094

Prototype	<b>short</b> sys_fsyz_set_label( <b>short</b> drive, <b>const char</b> * label)
drive	drive number
label	buffer that holds the label
Returns	0 on success, error if negative

### sys\_fsyz\_mkdir – 0xFFE098

Prototype	<b>short</b> sys_fsyz_mkdir( <b>const char</b> * path)
path	the path of the directory to create.
Returns	0 on success, negative number on failure.

### sys\_fsyz\_delete – 0xFFE09C

Prototype	<b>short</b> sys_fsyz_delete( <b>const char</b> * path)
path	the path of the file or directory to delete.
Returns	0 on success, negative number on failure.

### sys\_fsfs\_rename – 0xFFE0A0

Prototype	<b>short</b> sys_fsfs_rename( <b>const char</b> * old_path, <b>const char</b> * new_path)
old_path	the current path to the file
new_path	the new path for the file
Returns	0 on success, negative number on failure.

### sys\_fsfs\_set\_cwd – 0xFFE0A4

Prototype	<b>short</b> sys_fsfs_set_cwd( <b>const char</b> * path)
path	the path that should be the new current
Returns	0 on success, negative number on failure.

### sys\_fsfs\_get\_cwd – 0xFFE0A8

Prototype	<b>short</b> sys_fsfs_get_cwd( <b>char</b> * path, <b>short</b> size)
path	the buffer in which to store the directory
size	the size of the buffer in bytes
Returns	0 on success, negative number on failure.

### sys\_fsfs\_load – 0xFFE0AC

Prototype	<b>short</b> sys_fsfs_load( <b>const char</b> * 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

### sys\_fsfs\_register\_loader – 0xFFE0B0

Prototype	<b>short</b> sys_fsfs_register_loader( <b>const char</b> * 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

### sys\_fsfs\_stat – 0xFFE0B4

Prototype	<b>short</b> sys_fsfs_stat( <b>const char</b> * 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

## 3.6 Text System Functions

### sys\_txt\_set\_mode – 0xFFE0E0

Prototype	<b>short</b> sys_txt_set_mode( <b>short</b> screen, <b>short</b> mode)
screen	the number of the text device
mode	a bitfield of desired display mode options
Returns	0 on success, any other number means the mode is invalid for the screen

### sys\_txt\_set\_xy – 0xFFE0E8

Prototype	<b>void</b> sys_txt_set_xy( <b>short</b> screen, <b>short</b> x, <b>short</b> y)
screen	the number of the text device
x	the column for the cursor
y	the row for the cursor

### sys\_txt\_get\_xy – 0xFFE0EC

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

### sys\_txt\_get\_region – 0xFFE0F0

Prototype	<b>short</b> sys_txt_get_region( <b>short</b> 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

### sys\_txt\_set\_region – 0xFFE0F4

Prototype	<b>short</b> sys_txt_set_region( <b>short</b> 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

### sys\_txt\_set\_color – 0xFFE0F8

Prototype	<b>void</b> sys_txt_set_color( <b>short</b> screen, <b>unsigned char</b> foreground, <b>unsigned char</b> 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)

### sys\_txt\_get\_color – 0xFFE0FC

Prototype	<b>void</b> sys_txt_get_color( <b>short</b> screen, <b>unsigned char</b> * foreground, <b>unsigned char</b> * 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)

### sys\_txt\_set\_cursor\_visible – 0xFFE100

Prototype	<b>void</b> sys_txt_set_cursor_visible( <b>short</b> screen, <b>short</b> is_visible)
screen	the screen number 0 for channel A, 1 for channel B
is_visible	TRUE if the cursor should be visible, FALSE (0) otherwise

### sys\_txt\_set\_font – 0xFFE104

Prototype	<b>short</b> sys_txt_set_font( <b>short</b> screen, <b>short</b> width, <b>short</b> height, <b>unsigned char</b> * 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

### sys\_txt\_setsizes – 0xFFE0E4

Prototype	<b>void</b> sys_txt_setsizes( <b>short</b> chan)
chan	

### sys\_txt\_get\_sizes – 0xFFE108

Prototype	<b>void</b> sys_txt_get_sizes( <b>short</b> screen, p_extent text_size, p_extent pixel_size)
screen	the screen number 0 for channel A, 1 for channel B
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)

### sys\_txt\_set\_border – 0xFFE10C

Prototype	<b>void</b> sys_txt_set_border( <b>short</b> screen, <b>short</b> width, <b>short</b> 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)

### sys\_txt\_set\_border\_color – 0xFFE110

Prototype	<b>void</b> sys_txt_set_border_color( <b>short</b> screen, <b>unsigned char</b> red, <b>unsigned char</b> green, <b>unsigned char</b> 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)

### sys\_txt\_put – 0xFFE114

Prototype	<b>void</b> sys_txt_put( <b>short</b> screen, <b>char</b> c)
screen	the number of the text device
c	the character to print

### sys\_txt\_print – 0xFFE118

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

## 3.7 Interrupt Functions

### 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. 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	<code>void sys_int_enable_all()</code>
-----------	--

#### Example: C

```
// Enable processing of IRQs
sys_int_enable_all();
```

#### Example: Assembler

```
; Enable processing of IRQs
jrl 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. 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	<code>void sys_int_disable_all()</code>
-----------	---

#### Example: C

```
// Disable processing of IRQs
sys_int_disable_all();
```

#### Example: Assembler

```
; Disable processing of IRQs
jrl sys_int_disable_all
```

### `sys_int_disable` – 0xFFE00C

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	<code>void sys_int_disable(unsigned short n)</code>
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 – 0xFFE010

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	<b>void</b> sys_int_enable( <b>unsigned short</b> 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 – 0xFFE014

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( <b>unsigned short</b> 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 – 0xFFE018

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	<b>short</b> sys_int_pending( <b>unsigned short</b> 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 – 0xFFE020

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	<b>void</b> sys_int_clear( <b>unsigned short</b> 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
```