# Foenix Toolbox Programmer's Guide

Firmware functions for the Foenix Retro System F256 computers

**Peter Weingartner**

September 30, 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.

## 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).

## 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.

| **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 – 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 | **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_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 | **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 – 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 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 – 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 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 – 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 | **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 – 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 | **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 – 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 | **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

**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 | **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 – 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 | **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 – 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 | **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 – 0xFFE030

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 – 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 | **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 – 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 | **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 – 0xFFE03C

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 – 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 | **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 – 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 | **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 – 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 | **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 − 0xFFE04C

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 – 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 | **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 – 0xFFE054

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

### 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 | **short** 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 | **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 – 0xFFE064

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 – 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 | **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 – 0xFFE06C

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 – 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 | **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 – 0xFFE074

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 bitfield, 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 – 0xFFE078

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 – 0xFFE07C

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 – 0xFFE080

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 – 0xFFE084

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 – 0xFFE088

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 – 0xFFE08C

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 – 0xFFE090

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 – 0xFFE094

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 – 0xFFE098

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 – 0xFFE09C

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 – 0xFFE0A0

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 – 0xFFE0AC

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 – 0xFFE0B0

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. It there is an error in registering the loader, returns a negative number.

| Prototype | **short** sys_fsys_register_loader(**const char** * extension, p_file_loader loader) |
|---|---|
| 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 – 0xFFE0B4

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

### sys_txt_set_mode – 0xFFE0E0

| Prototype | **short** sys_txt_set_mode(**short** screen, **short** 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 | **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 |

### sys_txt_get_xy – 0xFFE0EC

| 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 |

### sys_txt_get_region – 0xFFE0F0

| 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 |

### sys_txt_set_region – 0xFFE0F4

| 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 |

### sys_txt_set_color – 0xFFE0F8

| 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) |

### sys_txt_get_color – 0xFFE0FC

| 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) |

### sys_txt_set_cursor_visible – 0xFFE100

| Prototype | **void** sys_txt_set_cursor_visible(**short** screen, **short** 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 | **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 |

### sys_txt_setsizes – 0xFFE0E4

| Prototype | **void** sys_txt_setsizes(**short** chan) |
|---|---|
| chan | |

### sys_txt_get_sizes – 0xFFE108

| Prototype | **void** sys_txt_get_sizes(**short** 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 | **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) |

### sys_txt_set_border_color – 0xFFE110

| 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) |

### sys_txt_put – 0xFFE114

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

### sys_txt_print – 0xFFE118

| Prototype | **void** sys_txt_print(**short** screen, **const char** * 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 | **void** sys_int_enable_all() |
|---|---|

**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 | **void** sys_int_disable_all() |
|---|---|

**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 | **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 – 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 | **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 – 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(**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: Assember**

```
; 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 | **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 – 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 | **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
```

# 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, an the expansion cartridge, the Toolbox is looking for a special header that contains a signature and specifies the starting address to run. For 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 describe 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.