fxReverse project documentation

For CASIO fx-9860G. Second revision.

Andreas Bertheussen
Simon Lothar

February, 2010

The authors of this document are in no way affiliated with or endorsed by CASIO. CASIO is a registered trademark of CASIO COMPUTER CO., LTD. SuperH is a trademark of Hitachi Ltd. All other trademarks are the property of their respective owners.

LaTeX (http://www.latex-project.org/) generated this file on February 9, 2010. Many thanks to the developers of LaTeX, and other open source tools like kile (http://kile.sourceforge.net/), pdftex (http://www.tug.org/applications/pdftex/), kdvi (http://developer.kde.org/~kdvi/) and evince (http://www.gnome.org/projects/evince/).

Contents

Chapter 1

Introduction

The intention of this document is to pass on the results of our research, and also tie together information that earlier has been lying spread. Simon Lothar and I started the work on this project late in February 2008, and a lot of interesting things have been found since then.

Ideas about reverse-engineering appeared at the Universal Casio Forum early 2006. An SDK for developing addins was released by CASIO early 2007, allowing users to write their applications in C, export addins, and also debug them on an included emulator. This more information available to the community about the calculators hardware, and also a supported[1] method of running binary code on its processor.

The SDK by CASIO included a library with stardard C functions and other specialized functions for printing text, drawing graphics etc. (see section 3.1). It was soon clear that the OS functions available in the official SDK were just a subset of a huge library. No real research had previously been done on these system calls, and so it became a central part of the project. In contrast to the Revolution-FX project (`http://revolution-fx.sf.net`), we do not intend to write new software but instead learn about the existing software in the OS.

Work on the protocol and development of a file transfer app was done by Manuel Naranjo and Andreas Bertheussen through analysis of logged communication by CASIO's FA-124.

[1] Not officially supported, but it did provide reliable information.

Chapter 2

Hardware

The fx-9860G calculator appears to be a typical microprocessor system, based on the SuperH architecture by Renesas (previously HITACHI). Most of the information on the hardware components was available before this document.

2.1 Components and features

The CPU is a SuperH 3 core, very similar to the SH7705 model. There are some deviations from the SH7705 such as ADC register and port addresses. We suspect that the die is customized by CASIO because of undocumented functionality. The CPU identification string changes also in the OS between hardware generation I and II, from "`SH7337`" to "`SH7355`". The CPU runs in big endian mode, at 14.74MHz and is possible to software overclock to 58.96MHz[1]. It works natively with the datatypes `long`, `word` and `byte` which are respectively 4, 2 and 1 bytes wide.

RAM: Amic LP62S16256F-T Series, 512k capacity.[2],[3]

Flash memory: Spansion S29JL032H, 4MB capacity.[4]

LCD driver: A Toshiba T6K11 variant.[5]

The system also includes a keyboard, the LCD, battery level sensors and an interface for SD cards (fx-9860G SD model). Serial and USB communication is provided by the CPU. In a hole next to the reset "button" are two contacts on the calculators circuit board. Upon shortening these together, a program starts for OS update.

RAM is mapped and accessed at address 0x88000000, the flash at 0x80000000 and 0xA0000000 (cached / uncached). The start of flash, 0xA0000000 is the CPUs reset vector, but the 0x80000000 range is the most commonly used because of speed advantages. Access to the display driver is done through 0xB4000000 for register selection and 0xB4010000 for data read/write.

2.2 Input hardware

The electrical configuration of they keyboard is a simple row-column setup. Default operation by the OS for reading keys, is to pull each of the rows to high, and then read the columns to identify key press shorts (0's) in that row.

[1] http://www.casiocalc.org?s=&showtopic=3782&view=findpost&p=43877
[2] http://www.amictechnology.com/pdf/LP62S16256F-T.pdf
[3] Not identified for GII due to glob-topped chip
[4] http://www.spansion.com/products/S29JL032H.html
[5] http://www.ortodoxism.ro/datasheets/toshiba/1952.pdf

In addition to the key PCB, there are several exposed pads on the main PCB that can be shorted, though these are not intended for users. If the battery lid is off, a hole in the shell exposes two of these contacts. When shorted, they will start an OS update program which ignores the version of the new firmware.

Chapter 3

Software

The ROM software/firmware is upgradeable through a USB connection to a computer. The latest OS version at time of writing for all fx-9860G variants, is 2.00.

3.1 System call table

To allow Addins to function properly between OS versions, the OS has a table of pointers to all of the API functions. This is common for all OS versions, and the code to access it lies at ROM address 0x80010070. A few entries in the table are pointers to strings, not functions, and could cause unexpected behavior if called. Version 1.04 increases the number of entries, adding syscall 0x1032 – named `Bkey_GetKeyTableInfoJumpFunc()` in the official SDK libraries.

The functions use the Renesas calling convention, first four arguments in registers r4 to r7, and additional arguments on the stack:

```
1          mov.l   syscall_number, r0
2          mov.l   syscall, r2
3          jmp     @r2
4          nop
5  syscall: .long 0x80010070
```

Several of the functions pointed to in this system call table table are documented in chapter **??** with their corresponding *syscall_number*.

3.2 Applications

The standard fx-9860G is equipped with several integrated applications that are accessible from the main menu. See table 3.1 for a listing. We do beleive that common functionality used in these applications, like expression parsers and input routines, are accessible through system calls - but these have so far been hard to understand or document properly.

Usage documentation for the applications can be found at CASIOs documentation site, http://world.casio.com/calc/download/en/manual/. Refer to section **??** for information on how to execute these applications from an addin application.

Table 3.1: List of applications

"Mode" name	Internal name	Description
RUN·MAT	`@RUNMAT`	Main mathematic expression evaluator interface
STAT	`@STAT`	Perform statistical calculations on input data
e·ACT	`@EACT`	Notebook and documentation tool
S·SHT	`@SSHEET`	Basic spreadsheet
GRAPH	`@GRAPH`	Plot and work with function graphs
DYNA	`@DYNA`	Graph with "animation"
TABLE	`@TABLE`	Generate function tables
RECUR	`@RECUR`	
CONICS	`@CONICS`	
EQUA	`@EQUA`	A solver for simultaneous, polynomial or general equations
PRGM	`@PRGM`	Create and run BASIC-like programs
TVM	`@FINANCE`	Perform common financial calculations
LINK	`@LINK`	Calculator⟶calculator or calculator⟶PC communication
MEMORY	`@MEMORY`	RAM, flash (and SD card) memory management
SYSTEM	`@SYSTEM`	General calculator settings and information

3.3 File formats

Data in the MCS (RAM filesystem) can both be stored to and loaded from "files", typically with the file name extension `.g1m`. Users also have the ability to store Addins (`.g1a` extension) in the storage memory. An icon will appear in the main menu if the Addin is in the correct format.

3.3.1 Header

All file formats that are read or written by the calculator share a common header format. This header is inverted before being written and read, so a description of the inverted file data make more sense than the normal file data:

Offset (b)	Size (b)	Description
	8	File identifier: "`USBPower`"
8	1	File type identifier
9	5	{0x00, 0x10, 0x00, 0x10, 0x00}
0xE	1	Control byte: (LSB of filesize (at offset 0x13)) + 0x41
0xF	1	0x01
0x10	4	Total file size as unsigned integer, big endian format
0x14	1	Control byte: (LSB of filesize (at offset 0x13)) + 0xB8
0x15	0x9	Unknown purpose, appears insignificant
0x1E	2	Number of objects contained, if file is of type G1M or G1R
0x20	...	Subheader depending on particular file type

Known file types are:

Value	Type
0xF3	Addin
0x31	General MCS memory file
0x49	E-Activity

3.3.2 Addin subheader

Refer to table 3.2 for the fields in the addin subheader. Upon execution, the addin file is loaded to 0x00300000, and the OS jumps to address 0x00300200 ("Addin code"). The addin code should use 0x08100000 for its `.bss` and `.data` sections. CASIOs SDK will automatically add initialization code that loads the `.data` section and performs various system calls. For a working reimplementation of this code for the GCC compiler, visit the fxSDK project (http://fxsdk.sourceforge.net/wikka.php?wakka=BinaryFormat).

Table 3.2: Addin subheader fields

Offset (b)	Size (b)	Description
0x20	8	"Internal" name, on the form "`@APPNAME`"
0x2B	1	Number of estrips
0x30	0xA	Version on the form "`01.23.4567`", first 5 bytes appear in SYSTEM app
0x3C	0xE	Creation date on the form "`YYYY.MMDD.HHMM`"
0x4C	0x44	A 30×18 pixel menu icon bitmap. Logical "1" is black, "0" is white"
0x1D4	8	Program title. Appears in SYSTEM app
0x1F0	4	Filesize as `unsigned long`, big endian.
0x200	...	Addin code

Some of the unlabeled memory above is related to e·Activity strips and their icons, but the function is not clear.

Chapter 4

Communication protocol

The CASIO fx-9860G can be connected with a computer or another compatible calculator. The calculator software allows file transmission, OS update, real-time screen picture transmission and many other functions. The PC software FA-124 by CASIO gives access to some of these.

4.1 Transport protocol

Two interfaces can be used to connect to the calculator:

- A 3-pin serial interface
- A USB 1.1 interface

Mentioned 3-pin serial interface is a EIA232/RS232-like interface, with the voltage level +4.2V used for logical 1, and 0V (GND) for logical 0. A voltage converter like the MAX232 chip is needed to use the interface with an EIA232-compatible device. The connector used is a 3-wire 2.5mm minijack, and Rx and Tx are the contacts near the tip. The cable is crossed so that Rx and Tx on one side is switched on the other side. Default setting for the interface is 8-bit bytes and a baud rate of 9600 bps.

The calculator can be located on the USB bus by the product ID 0x6101 and the provider ID 0x07CF. Some USB control messages must be sent to initiate the connection. Here is an example using libusb[1]:

```
int init_connection() {
        char *buffer = calloc(0x29, sizeof(char));
        usb_control_msg(usb_handle, 0x80, 0x6, 0x100, 0, buffer, 0x12, 200);
        usb_control_msg(usb_handle, 0x80, 0x6, 0x200, 0, buffer, 0x29, 250);
        usb_control_msg(usb_handle, 0x41, 0x1, 0x0, 0, buffer, 0x0, 250);
        free(buffer);
        return 0;
}
```

You should provide error checking of the `calloc()` and `usb_control_msg()` calls yourself.

4.2 Packets and packet flow

The protocol uses packets for communication. Packets are structured as in table 4.1, the top of the table indicates the beginning of the packet.

[1] http://libusb.sourceforge.net/

Table 4.1: Packet layout

Size	Field name	Description	Values
1 b	Type (*T*)	The basic purpose of the packet	0x00 to 0x1F
2 b	Subtype (*ST*)	The specific function of the packet	“00” to “57”
1 b	Extended (*EX*)	Decides if *DS* and *D* fields follow	’0’ - no, ’1’ - yes
(4 b)	Data size (*DS*)	Size of *D* field	“0000” to “FFFF”
(*n* b)	Data (*D*)	Additional data related to packet function	
2 b	Checksum (*CS*)	Checksum for integrity check	“00” to “FF”

The uppermost part of the table are the fields transferred first.

An extended packet has the *EX* field set to ’1’, and will also contain the *DS* and *D* fields. The *DS* field specifies the size of the *D* field, in ASCII-hex format (see below).

There are some restrictions on the content of a packet:

- Numeric values (except for *T* field) are encoded as ASCII characters, in hexidecimal base. Characters ’0’⟶’9’ and ’A’⟶’F’ are valid. Indicated by double quotes, like “BEEF” where numeric values are obvious, or by the term *ASCII-hex*.
- Bytes in the range 0x00⟶0x1F can not occur in the *D* field. Bytes with those values must be offset by 0x20 and prefixed with a 0x5C (’\’) byte. E.g. 0x0A (LF) becomes 0x5C2A. Any 0x5C bytes alone must be sent as 0x5C5C.

Checksums are computed by summing individual bytes in the *ST* and following fields, and then adding 1 to its bitwise complement. The following example in C shows how to compute the checksum, but the sum will have to be converted to ASCII-hex format before being used in a packet.

```
/* 'buffer' is a pointer to the packet structure with length 'length' */
int i;
char sum;
for (i = 1; i < length; i++) { /* skips T field */
        sum += buffer[i];
}
sum = (~sum)+1;
```

4.2.1 Packet types (*T*)

Seven packet types are known, listed in table 4.2.

Table 4.2: Various packet types

T	’Name’	Description
0x01	Command	Used to perform or request actions. *ST* field selects the action.
0x02	Data	Carries ’raw’ data in the context of a command. E.g. file data.
0x03	Roleswap	Causes devices to switch roles. Active becomes passive and vice versa.
0x05	Check	Used to verify if the devices are still connected.
0x06	Ack	Default response at successful packet reception.
0x15	Error	Default response if problems occur.
0x18	Terminate	Terminates communication

4.2.2 Flow patterns

A device can be either the primary (active) or secondary (passive) device on a link. (You can easily set the calculator in passive mode by selecting “RECV” in the LINK program.) This decides which device has the right to start sending commands. See table 4.3 for examples on flow patterns. Some of the protocol commands, like file transfers, exist in two forms; one that precedes the sending of data, the other type is a request for the passive side to perform that command. When the request command is sent, the active side sends the roleswap packet, and the previously passive side proceeds with the command that was requested by the now passive side.

Table 4.3: Usual packet flow patterns

Pattern	Primary		Secondary
Initialization	Check	$\Longrightarrow$	
		$\Longleftarrow$	*Ack*
(1) Single command	Command	$\Longrightarrow$	
		$\Longleftarrow$	*Ack*
(2) Command+data	Command	$\Longrightarrow$	
		$\Longleftarrow$	*Ack*
	Data packet 1	$\Longrightarrow$	
		$\Longleftarrow$	*Ack*
	...	...	...
	Data packet *n*	$\Longrightarrow$	
		$\Longleftarrow$	*Ack*
(3) Request 1	Request-command	$\Longrightarrow$	
		$\Longleftarrow$	*Ack*
	Roleswap	$\Longrightarrow$	
		$\Longleftarrow$	Requested command
	Ack	$\Longrightarrow$	
		$\Longleftarrow$	Roleswap
(4) Request 2	Request-command	$\Longrightarrow$	
		$\Longleftarrow$	*Ack*
	Roleswap	$\Longrightarrow$	
		$\Longleftarrow$	Requested command
	Ack	$\Longrightarrow$	
		$\Longleftarrow$	Data packet 1
	Ack	$\Longrightarrow$	
	...	...	...
		$\Longleftarrow$	Data packet *n*
	Ack	$\Longrightarrow$	
		$\Longleftarrow$	Roleswap
Termination	Terminate	$\Longrightarrow$	
		$\Longleftarrow$	*Ack*
	(End of communication)		

4.2.3 Timeouts and transmission problems

If no response is returned on the initial connection check, communication should stop. The passive side should close the interface if inactive for more than 6 minutes. This means that the active side must ’ping’

the passive with a check packet at least every 6 minutes during idle periods. If the time between bytes for a packet is more than 2 seconds, it is considered invalid, and must be requested again.

Possible problems that can occur during transmission are:

- Corrupted data.
- "Deaf" passive side.
- "Deaf" active side.

Data corruption is easily detected by a bad checksum. Upon discovering this, the appropriate Error packet is sent to request a retransmission. If no response packet to a sent command is received within 10 seconds, a check packet is sent. If the passive side receives the check packet, an error packet is returned as a retransmission request, the active side resends the packet and communication continues. If no response to the check packet is returned in 10 seconds, another one is sent. It must be answered within 10 seconds before communication is ended.

4.3 The command packet

There are three groups of commands:

- System commands
- MCS (RAM file system) commands
- Flash file system commands

4.3.1 Command packet data field layout (*D*)

The *D* subfields of a command packet are shown in table 4.4. You don't have to include a *D* field in commands that don't use any of its fields.

Table 4.4: Command packet subfields

Size	Field name	Description	Values
2 b	Overwrite (*OW*)	Mode of operation if a file exists.	"00" to "02"
2 b	Data type (*DT*)	The data type	Unknown, but likely ASCII-hex
8 b	Filesize (*FS*)	Size of file being transferred	ASCII-hex
2 b	*SD1*	Size of *D1*	ASCII-hex
2 b	*SD2*	...	...
2 b	*SD3*	...	...
2 b	*SD4*	...	...
2 b	*SD5*	...	...
2 b	*SD6*	Size of *D6*	ASCII-hex
(*SD1* b)	*D1*	First command argument field	
(*SD2* b)	*D2*	...	
(*SD3* b)	*D3*	...	
(*SD4* b)	*D4*	...	
(*SD5* b)	*D5*	...	
(*SD6* b)	*D6*	Last commandargument field	

Field *OW* specifies how to act in cases where a file exists:

- “00” Request user confirmation before overwriting.
- “01” Terminate if file already exists.
- “02” Force overwrite.

4.3.2 System command reference

The number in parentheses indicate the communication pattern it follows (see table 4.3).

“00”: Restart/reset (1)

Resets or restarts the equipment.

“01”: Get device info (1)

Returns device info in an extended Ack. See section **??**.

“02”: Set link settings (1)

D1	Baud rate as ASCII string: e.g. “9600” or “19200”.
D2	Parity: “ODD”, “EVEN” or “NONE”.
D3	Stop bits: “1” or “2”.

4.3.3 MCS command reference

Refer to table **??**, and remember that they operate on the RAM filesystem, and not the flash storage memory.

4.3.4 Flash command reference

Flash commands operate on the flash memory file system (device name “fls0”). Usage of these operations on the SD card (device name “crd0”) has not been tested. See table **??** for the listing.

4.4 The data packet

This packet type is used for transfer of “raw” data. The maximum safe amount of data transmitted with a data packet is 256 bytes. Because some bytes have to be escaped, 512 bytes is the maximum transmittable amount. Keep the data packets *ST* field identical to the *ST* field of the command that the data is transmitted in context with. The *D* field layout is shown in the following table, where the uppermost parts of the table are the fields that are transferred first:

Size	Field name	Description	Values
4 b	*TN*	Total number of data packets in transmission	“0001”-“FFFF”
4 b	*CN*	Current packet number	“0001”-“FFFF”
0-512 b	*DD*	The data	

4.5 The roleswap packet

Roleswapping can only be done in one way, so its subtype will always be “00”. Its usage as shown in 4.2.2, is to transfer control to the secondary side once a command has been requested with one of the request-commands.

	T	*ST*		*EX*	*CS*	
Hex	0x03	0x30	0x30	0x30	0x37	0x30
Char	.	'0'	'0'	'0'	'7'	'0'

4.6 The check packet

The two different types have the subtypes "00" and "01". The first is always used on start of communication (see section 4.2.2 for typical packet flow), the second type is the type used during communication, on timeouts and similar occasions where it is necessary to verify the connection.

4.7 The ack packet

The default ack response uses the subtype "00". The subtype "01" is used to confirm a request for file overwriting. The last ack type "02", is the *extended ack*, used to transmit varying information about the calculator as a response to command "01" - "Get device info". Only the extended ack packet uses the packets *D* field, its layout is seen in table **??**. Skipped fields contain 0xFF bytes.

4.8 The error packet

The error packet is used to indicate problems during communication.

ST	Description
0x00	Default
0x01	Resend request
0x02	Overwrite request if file exists
0x03	"No"-response to an overwrite request
0x04	Overwrite impossible
0x05	Memory is full

4.9 The terminate packet

This packet is used to reset any active commands or transfers, and must be respected by both the primary and secondary side.

ST	Description
0x00	Default
0x01	User requested termination
0x02	Termination caused by timeouts
0x03	Termination on overwrite request

4.10 Packet and flow examples

All examples will be given in hexadecimal base with spaces separating the fields. The left part is the part of the packet that is sent first. As a reminder, the common fields for packets are; *T*, *ST*, *EX* (*DS D*) and *CS*. We assume the PC is the primary side and the calculator is the secondary.

Packet data is either shown "raw", like `0x18`, or in ASCII form, like `Somedata`.

4.10.1 Initialization

Pri.					Sec.
⟹	0x05	00	0	70	⟹
⟸	0x06	00	0	70	⟸

This is a connection check. A check packet is sent with subtype "00", being typical for start of communication. It is answered with an ack packet. Note that the checksum for any packet with *ST* and *EX* fields '0'-ed, will alway be 0x3730, or the hex value 0x70 in ASCII form.

Pri.						Sec.
⟹	0x01	01	0	6F		⟹
⟸	0x06	02	1	00A4	*omitted D-field* E3	⟸

The first command is the system command for requesting device information. Our secondary side responds with an extended ack packet. This packet includes a *DS* and *D*-fields as indicated by the *EX* field being set to '1'. If we read the 4 bytes of the *DS* field as a string, it will read "00A4". 0x00A4 = 168, the size of the following *D*-field. The content of this field is described in table **??**.

4.10.2 Existing file situations

Command packets have an *OW*-field that dictates how the passive side should act if a file transmitted already exists. It is also possible for the primary side to be notified if a file exists before deciding to overwrite it, skip the file transfer or terminating communication. This mode of operation is the default and is used when *OW*-field is set to "00".

Pri.										Sec.
⟹	0x01	45	1	0024	00	00	00000008			⟹
	00	08	00	00	04	00	FILENAME	fls0	56	
⟸	0x15	02	0	6E						⟸

A flash file transfer is initiated with a file named "FILENAME", and the *OW*-field set to the default "00". Because this file already exists for this example, the secondary side responds with an error packet with a subtype indicating that it is an overwrite request. From this situation, the communication can continue in three different ways:

Overwrite accept

Pri.									Sec.
⟹	0x06	01	0	6F					⟹
⟸	0x06	00	0	70					⟸
⟹	0x02	45	1	0010	0001	0001	data1234	BF	⟹
⟸	0x06	00	0	70					⟸
⟹	Proceed with next command ...								⟹

In this example, we accept the overwrite request by sending an Ack with *ST* of "01". After this has been acknowledged by the secondary side, we can continue to transmit the data packet(s) related to the original command. The number of data packets depends on the size of the file - only one is needed for this example.

Overwrite decline

Pri.					Sec.
⟹	0x15	03	0	6D	⟹
⟸	0x06	00	0	70	⟸
⟹	Proceed with next command …				⟹

Terminate connection

Pri.					Sec.
⟹	0x18	03	0	6D	⟹
⟸	0x06	00	0	70	⟸
	Communication end				

4.11 Screen streaming

It is possible to set up the calculator to output the screen image continually. This is useful for demonstration purposes. The protocol for streaming does not require acknowledgement of received packets. Once the calculator is set in the streaming mode (OHP), it will transmit bitmap images in the following format:

Size	Contents
6	Type specifier and image format: 0x0B TYP01
1024	Picture data, one row at the time from the top, from left to right. (like VRAM)
2	Checksum of the picture data, calculation shown earlier.

The only purpose of the checksum is to detect and discard potentially bad frames. No retransmission functionality is available with this protocol. The images are not output at a onstant rate, but are instead triggered by display operations that programs perform. This means that a frame 'in transit' can be interrupted by a new transmission of a frame. See the source code of the program `screenstreamer`[2] for an example implementation.

[2] http://sourceforge.net/apps/trac/fxsdk/wiki/screenstreamer

Table 4.5: Overview of MCS commands

ST	***Name*** (*pattern*)
	Field *Description*
“20”	**Create directory** (1)
	D1 The name of the directory
“21”	**Delete directory** (1)
	D1 The name of the directory
“22”	**Rename directory** (1)
	D1 The name of the directory
	D2 New name for the directory
“23”	**Change working directory** (1)
	D1 The name of the directory
“24”	**File transfer request** (4)
	DT MCS data type
	D1 The name of the directory
	D2 The name of the file
	D3 Group name
“25”	**File transfer** (3)
	OW As described in section 4.3.1
	DT MCS data type
	FS Filesize
	D1 The name of the directory
	D2 The name of the file
	D3 Group name
“26”	**Delete file** (1)
	DT MCS data type
	FS Filesize
	D1 The name of the directory
	D2 The name of the file
	D3 Group name
“27”	**Rename file** (1)
	DT MCS data type
	D1 The name of the directory
	D2 The name of the file
	D2 New name for the file
“28”	**Copy file** (1)
	DT MCS data type
	D1 The name of the directory
	D2 The name of the file
	D3 Name for the new directory
	D4 Name for the new file
“29”	**File transfer all request** (4*)
“2A”	**Unknown - reset MCS?** (1)
“2B”	**Capacity transmit request** (3)
“2C”	**Capacity transmit** (1)
	FS The free capacity
“2D”	**File info transfer all request** (3*)
“2E”	**File info transfer** (1)
	DT MCS data type
	FS Filesize
	D1 The name of the directory
	D2 The name of the file
	D3 Group name
“2F”	**RAM image transfer request** (3)
“30”	**RAM image transfer** (2)
“31”	**Setup entry transfer request** (3)
	D1 Name of setup entry
“32”	**Setup entry transfer** (1)
	D1 Name of setup entry
	D2 Data (ASCII-hex)
“33”	**Setup entry transfer all request*** (3)

Patterns marked “*” mean that several commands or command+data can be expected before final roleswap.

Table 4.6: Overview of flash commands

ST	***Name*** (*pattern*)	
	Field	*Description*
"40"	**Created directory** (1)	
	D1	The name of the directory
	D5	The device name
"41"	**Delete directory** (1)	
	D1	The name of the directory
	D5	The device name
"42"	**Rename directory** (1)	
	D1	The name of the directory
	D2	New name for the directory
	D5	The device name
"43"	**Change working directory** (1)	
	D1	The name of the directory
	D5	The device name
"44"	**File transfer request** (4)	
	D1	The name of the directory
	D2	The name of the file
	D5	The device name
"45"	**File transfer** (2)	
	OW	As described in section 4.3.1
	FS	Filesize
	D1	The name of the directory
	D2	The name of the file
	D5	The device name
"46"	**Delete file** (1)	
	D1	The name of the directory
	D2	The name of the file
	D5	The device name
"47"	**Rename file** (1)	
	D1	The name of the directory
	D2	The name of the file
	D3	New name for the file
	D5	The device name
"48"	**Copy file** (1)	
	D1	The name of the directory
	D2	The name of the file
	D3	Name for the new directory
	D4	Name for the new file
	D5	The device name
"49"	**File transfer all request** (4*)	
	D5	The device name
"4A"	**Unknown - reset flash?** (1)	
"4B"	**Capacity transmit request** (3)	
	D5	The device name
"4C"	**Capacity transmit** (1)	
	FS	The free capacity
	D5	The device name
"4D"	**File info transfer all request** (3*)	
	D5	The device name
"4E"	**File info transfer** (1)	
	FS	Filesize
	D1	The name of the directory
	D2	The name of the file
	D5	The device name
"4F"	**Flash image transfer request** (4)	
	D5	The device name
"50"	**Flash image transfer** (2)	
	D5	The device name
"51"	**Optimize filesystem** (1)	
	D5	The device name
"52"	**OS update related** (?)	
"53"	**OS update related** (?)	
"54"	**OS update related** (?)	
"55"	**OS update related** (?)	
"56"	**OS update related** (?)	
"57"	**OS update related** (?)	

Patterns marked "*" mean that several commands or command+data can be expected before final roleswap.

Table 4.7: Layout of *D* field in extended ack packet

Size	Description	Format
8 b	Hardware identifier	ASCII string
16 b	Processor identifier	ASCII string
8 b	Preprogrammed ROM capacity	ASCII-hex (in k's)
8 b	Flash ROM capacity	ASCII-hex (in k's)
8 b	RAM capacity	ASCII-hex (in k's)
16 b	Preprogrammed ROM version	"xx.xx.xx" + type
16 b	Boot code version	"xx.xx.xx" + type
8 b	Boot code offset	ASCII-hex
8 b	Boot code size	ASCII-hex (in k's)
16 b	OS code version	"xx.xx.xx" + type
8 b	OS code offset	ASCII-hex
8 b	OS code size	ASCII-hex (in k's)
4 b	Protocol version	"xx.x"
16 b	Product ID	ASCII string
16 b	Name set by user in SYSTEM	ASCII string

The uppermost part of the table are the fields transferred first.

Chapter 5

Application programming interface

This chapter is meant to be a reference. See section 3.1 for information on calling convention etc. None of the function names presented are found anywhere in the software, but we consider them to be good and suggest that you use them.

5.1 Cursor control

Ability to move the cursor is available in the official SDK, by the `locate()` function. More advanced functionality exists however, like cursor flashing with the cursor styles shown in Figure **??**.

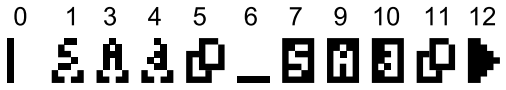

Figure 5.1: The different cursor flashing styles

Change note: As of the arrival of OS 2.0, the system calls 0x139 and 0x13A did not work as expected. They have therefore been replaced with calls in the 0x800-range, which also work in the old versions of the OS.

0x138: Set cursor position

Synopsis: `int Cursor_SetPosition(char column, char row);`

Description: Moves the cursor to the given position, where *column* is in the range [0,20] and *row* is in the range [0,7]. The official SDK function `locate()` is a wrapper for this function.
Returns: 1 on success, 0 if an argument is out of range

0x80E: Get cursor flash style

Synopsis: `int Cursor_GetFlashStyle();`

Description: Returns current flash style.
Returns: The current flash style.

0x811: Enable cursor flashing

Synopsis: `void Cursor_EnableFlash(char flash_style);`

Description: Enables cursor flashing, and sets the mode to the style given in *flash_style*. The different values are seen in figure **??**.
Returns: Flash mode.

0x812: Disable cursor flash

Synopsis: `void Cursor_DisableFlash();`

Description: Equivalent to calling `Cursor_SetFlashMode(0);`.
Returns: (void)

5.2 Text printing

A varied set of text printing functions are available in the official SDK. They are all different paths that end up with one or multiple calls to the core printing function named `PrintXY()`. The function dependency graph is shown below in figure **??**. Note that the `Print()` function for example will end up calling `PrintC()` repeatedly for each character in a string. However, the functions below such as `PrintAtCursor()` are well capable of printing strings and are not limited to just printing single characters. This is essentially like writing a custom printing function on a PC that splits a string in characters, and calls `printf()` for *each of the characters*, forgetting that `printf()` can handle strings as well. Considering this, you could make your Add-In more resource effective by calling `PrintAtCursor()` instead of `Print()`.

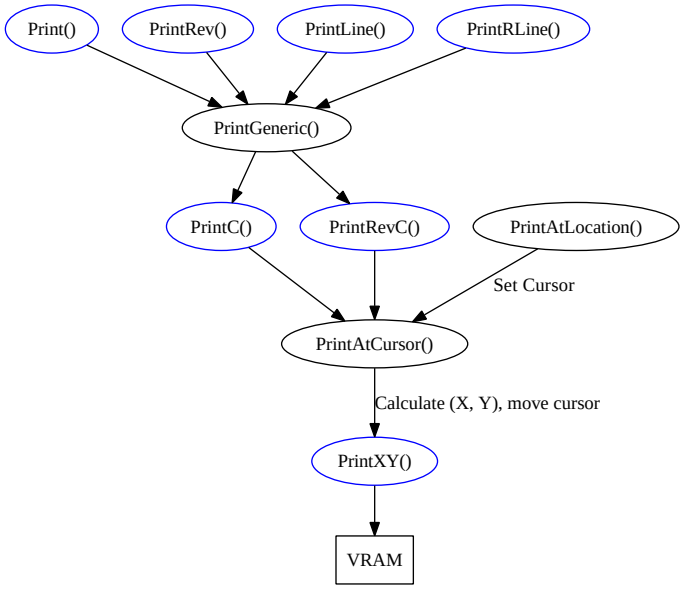

Figure 5.2: Print function dependency graph. Blue functions are in the SDK.

0x816: Print a string at the cursor

Synopsis: `void Print_Generic(int mode, char *string, int maxcol)`

Description: *string* is a pointer to the string. *mode* can be one of the following:

Value	Description
0	Ordinary `Print()` behavior. Value of *maxcol* is discarded (defaults to 21).
1	`PrintLine()` behavior. Prints until the specified column *maxcol* is reached.
0xA	`PrintRev()` behavior. Value of *maxcol* is discarded (defaults to 21).
0xB	`PrintRLine()` behavior. Prints until the specified column *maxcol* is reached.

Returns: (void)

0x13C: Print a string at the cursor

Synopsis: `void Print_AtCursor(char *string, int type)`

Description: Works like `Print()` if *type* is 0, but if *type* is 1 the text is reversed.
Returns: (void)

0x15D: Print a string at given location

Synopsis: `void Print_AtLocation(char *string, int column, int row)`

Description: Works like `Print()`, but the provided *column* and *row* in the ranges [0,20] and [0,7] are used to place the cursor before printing.
Returns: (void)

5.3 Keyboard interface

Read keys are initially represented by "matrix codes", indicating their physical connectivity. Key placement on the fx-9860G Slim is different than the other fx-9860G variants, which means that many of its keys have a different matrix code than earlier models. See tables **??** and **??**. When a key is pressed, an interrupt is triggered. Its interrupt handler then disables further keyboard interrupts if a key is detected. A matrix code of the detected key is stored in the matrix code buffer. The repeat timer (timer 2) is started and its handler checks if the key is being held down. If no keys are pressed after this, the repeat timer is stopped and interrupts are re-enabled.

Table 5.1: Matrix codes for fx-9860G variations

07__	06__	05__	04__	03__	02__	01__	Code
			DIAG		**OSUPD**		__0C
							__0B
F1	F2	F3	F4	F5	F6		__0A
SHIFT	OPTN	VARS	MENU	**LEFT**	**UP**		__09
ALPHA	x^2	x^n	EXIT	**DOWN**	**RIGHT**		__08
XTT	log	ln	sin	cos	tan		__07
ab/c	F<->D	(	)	,	−>		__06
7	8	9	DEL				__05
4	5	6	*	div			__04
1	2	3	+	-			__03
0	.	EXP	(-)	**EXE**			__02
						AC	__01

Table 5.2: Matrix codes for fx-9860G Slim

07__	06__	05__	04__	03__	02__	01__	Code
			DIAG		**OSUPD**		__0C
							__0B
MENU	XTT	SHIFT	ALPHA				__0A
F1	log	x^2	x^n	**LEFT**	**UP**		__09
F2	ln	,	OPTN	**DOWN**	**RIGHT**		__08
F3	sin	−>	VARS	HELP	LIGHT		__07
F4	cos	7	4	1	0		__06
F5	tan	8	5	2	.		__05
F6	ab/c	9	6	3	EXP		__04
EXIT	F<->D	DEL	*	+	(-)		__03
	(	)	div	**EXE**	-		__02
						AC	__01

The matrix code buffer is a FIFO that can hold 16 matrix codes. It is comprised of two arrays that hold the row and column value separately (two `char` arrays).

A separate key code buffer can hold 16 key codes (type `short`). These codes are independent of the keyboard layout.

The following functions are provided for experimental purposes.

0x247: Get a keypress from the keyboard

Synopsis: `int Kbd_GetKeyWait(int *col,int *row,int waittype,int timeout,int menu,ushort *keycode)`

Description: Reads a key from the keyboard. *col* and *row* will contain the column and row of the key that may be pressed. *waittype* specifies how to wait for a key according to the following table:

Macro	Value	Description
KEYWAIT_HALTON_TIMEROFF	0	Only return when a key has been pressed.
KEYWAIT_HALTOFF_TIMEROFF	1	Return immediately.
KEYWAIT_HALTON_TIMERON	2	Return when a key has been pressed or after a specified timeout.

A timeout is specified with *timeout*, in seconds. The maximum timeout period is one hour, or [0,3600] seconds. Whenever *waittype* is set to KEYWAIT_HALTOFF_TIMEROFF, the usual MENU key function is not provided. That is, the program won't suspend and return to the menu, but will instead return the key code for the menu key. In the other modes of waiting, this behavior can be forced by setting *menu* to a nonzero value. If you want the normal behavior of the MENU key, you must set *menu* to zero and not use the KEYWAIT_HALTOFF_TIMEROFF waiting mode.

Returns: An integer;

Macro	Value	Description
KEYREP_NOEVENT	0	No keypress was detected
KEYREP_KEYEVENT	1	Key was detected
KEYREP_TIMEREVENT	2	A timeout occured when waiting for key

5.4 Display buffer management

CASIOs SDK includes the functions `SaveDisp()` and `RestoreDisp()` to allow storage and recall of the screen buffer. Three buffers or "pages" of 1k were available for this. Some functions were found that provide more power than the original ones. Keep in mind that *pageID* is not the same number used in the SDK functions. Some pages appear to be used by functionality in other programs, but using them in you own code won't cause problems.

pageID	SDK `...Disp()`	SDK `disp.h` macro def
1	5	`SAVEDISP_PAGE2`
2	6	`SAVEDISP_PAGE3`
3	1	`SAVEDISP_PAGE1`
7	4	not defined
8	3	not defined

0x135: Get pointer to VRAM

Synopsis: `char *Disp_GetVRAMPtr(void);`

Description: Returns a pointer to the VRAM. This is important to use if your program writes directly to the VRAM, because the VRAM location changes between OS versions.
Returns: The pointer.

0x158: Core display buffer operations

Synopsis: `char *Disp_Manage(int pageID, int action);`

Description: If *action* is 0, restore VRAM to the data in the given *pageID*. If *action* is 1, save VRAM data to the given *pageID*. If *action* is 2, return a pointer to the page memory.
Returns: If *action* is 2, and a valid *pageID* is provided, a pointer is returned. Other cases return 0.

Three wrappers for `Disp_Manage()` exists, that could make your code more readable:

Syscall	Prototype	Equivalent call
0x153	`int Disp_Save(int pageID);`	`Disp_Manage(pageID, 1);`
0x154	`int Disp_Restore(int pageID);`	`Disp_Manage(pageID, 0);`
0x155	`char *Disp_GetPtr(int pageID);`	`Disp_Manage(pageID, 2);`

5.5 Integrated applications

It is possible to execute the intergrated applications through system calls. How to return from them to your addin code has not yet been figured out, if possible at all.

Syscall	Function	Application
0x985	`App_CONICS(...);`	CONICS
0x998	`App_DYNA(...);`	DYNA
0x9DF	`App_EACT(...);`	e·ACT
0x9E2	`App_EQUA(...);`	EQUA
0x9F5	`App_PRGM(...);`	PRGM
0xA00	`App_FINANCE(...);`	TVM
0xA48	`App_GRAPH(...);`	GRAPH
0xA4A	`App_LINK(...);`	LINK
0xA6B	`App_MEMORY(...);`	MEMORY
0xA75	`App_RECUR(...);`	RECUR
0xAAE	`App_RUNMAT(...);`	RUN·MAT
0xAC6	`App_STAT(...);`	STAT
0xAC8	`App_SYSTEM(...);`	SYSTEM

0x005: Refresh cached Add-In application table

Synopsis: `int App_RefreshAddinTable()`

Description: Scans the storage memory for Add-Ins, and updates the registry. This is required if you plan to run something with call 0x49A after you have copied an Add-In to the storage memory.
Returns: 0

0x46B: Get count of integrated applications.

Synopsis: `int App_GetIntegratedCount()`

Description: Returns the number of integrated applications.
Returns: The number.

0x49A: Execute any application

Synopsis: `int App_Run(int R4, int R4, int` *`index`*`, int` *`allow_recursion`*`)`

Description: Executes any application provided its *index*. The index range for the integrated applications, spans from 0 to (`App_GetintegratedCount() - 1`), while eventual add-in applacations follow. To allow an add-in application to run itself, make *allow_recursion* nonzero.
Returns: Depends on application.

Execute specific integrated application

Synopsis: `App_`*`APPNAME`*`(int` *`runMode`*`, unsigned short` *`stripNum`*`);`

Description: Refer to section 3.2 on how to use the applications. The parameters are assumed to work the same way as in the official SDK. In cases where *runMode* is 1, the application is executed from the main menu. When *runMode* is 0, the application was run as a strip in an e·Activity and *stripNum* indicates the strip number. How applications' e·Activity strips work is currently unknown, but running the application with *runMode* set to 1 works.

5.6 Serial interface access

The OS functions provide buffered access to the CPU's serial IO unit. The receive buffer holds 1kb, while the transmit buffer holds 256b.

0x40C: Read byte

Synopsis: `int Serial_ReadByte(unsigned char *dest);`

Description: Fetches one byte from the receive buffer to the character pointed to by *dest*
Returns: 0 on success, 1 if receive buffer is empty or 3 if device is not open.

0x40D: Read bytes

Synopsis: `int Serial_ReadBytes(unsigned char dest, int max, short *size);`

Description: Fetches up to *max* bytes from the character buffer into the buffer pointed to by *dest*. The number of bytes actually read from the buffer are stored at *size*.
Returns: 0 on success, 1 if receive buffer is empty or 3 if device is not open.

0x40E: Write byte

Synopsis: `int Serial_WriteByte(unsigned char byte);`

Description: Writes *byte* to the transmit buffer.
Returns: 0.

0x40F: Write bytes

Synopsis: `int Serial_WriteBytes(unsigned char src, int size);`

Description: Writes *size* bytes from the buffer at *src* to the transmit buffer, if there is room for it.
Returns: 0 on success, 2 if buffer is too full or 3 if the device is not open.

0x410: Transmit byte (unbuffered)

Synopsis: `int Serial_WriteByte(unsigned char byte);`

Description: Puts *byte* into the serial FIFO, if it is empty.
Returns: 0 on success, 1 if the FIFO is not empty or 3 if the device is not open.

0x411: Get size of receive buffer

Synopsis: `int Serial_GetRxBufferSize(void);`

Description: Returns the current size of the data in the receive buffer.
Returns: The current size.

0x412: Get free capacity of transmit buffer

Synopsis: `int Serial_GetTxBufferFreeCapacity`

Description: Returns the remaining capacity of the transmit buffer.

Returns: The capacity.

0x413: Clear receive buffer

Synopsis: `int Serial_ClearReceiveBuffer(void);`

Description: Clears the receive buffer of the opened device.
Returns: 0 on success or 3 if the device is not open.

0x414: Clear transmit buffer

Synopsis: `int Serial_ClearTransmitBuffer(void);`

Description: Clears the transmit buffer.
Returns: 0.

0x418: Open the serial link

Synopsis: `int Serial_Open(unsigned char *conf);`

Description: Opens and prepares the serial interface for communication. The provided *conf* is a pointer to an array, consisting of five bytes describing the interface settings:

Table 5.3: Parameters for `Serial_Open()`.

Setting:		`conf[0]`	`conf[1]`	`conf[2]`	`conf[3]`	`conf[4]`	`conf[5]`
Purpose:		?	Baud rate	Parity	Byte length	Stop bits	?
Setting value	0	(default)	300	none	8 bit	one	(default)
	1		600	odd	7 bit	two	
	2		1200	even			
	3		2400				
	4		4800				
	5		9600				
	6		19200				
	7		38400				
	8		57600				
	9		115200				

Returns: The function returns 0 on success, 3 if the interface is already opened, and 4 if *settings[0]* is nonzero.

0x419: Close the serial link

Synopsis: `int Serial_Close(int mode);`

Description: Clears transmission buffers, and disables the serial link function. If *mode* is 1, the interface is disabled regardless of waiting (not transmitted) data. If *mode* is not 1, this function will not close the interface if data is waiting, and will return 5.
Returns: 0 on success (link closed) or 5 if there is data waiting to be transmitted (link not closed).

0x422: Peek in receive buffer

Synopsis: `int Serial_Peek(int index, unsigned char *dest);`

Description: Copies the byte at *index* in the receive buffer to the character pointed to by *dest* if it is within the size of the receive buffer.
Returns: 0 on success, 1 if *index* does not point to an active byte in the buffer or 3 if the device is not open.

0x425: Get link status

Synopsis: `int Serial_IsOpen(void);`

Description: Gets the status of the serial link.
Returns: 1 if device is open or 3 if closed.

5.6.1 RTC operation

The SH7705 includes a real time clock which keeps time in hours, minutes, seconds and milliseconds (64Hz resolution). The OS provides a few functions for accessing the RTC module. Values in the RTC registers are BCD-coded, meaning each nibble (4 bits) has a range of 0-9 instead of the usual 0-15.

0x039: Reset RTC

Synopsis: `void RTC_Reset(uint mode)`

Description: If *mode* is zero, only the RTC "circuitry" is initialized (by setting RESET, START and RTCEN bits in RCR2). This is probably not what you want. Provide a nonzero *mode* to also reset the hours, minutes, seconds etc. registers.
Returns: (void)

0x03A: Get RTC time

Synopsis: `void RTC_GetTime(uint *hours, uint *mins, uint *secs, uint *msecs)`

Description: Fills the provided integers with the correct time. The RTC registers are in BCD format, and are *not* converted back to 'native' format. That means you will for example get 0x59 in *mins* if the clock is at 59 minutes.
Returns: (void)

0x03B: Get 64Hz tick

Synopsis: `int RTC_GetTicks()`

Description: Returns a counter that is incremented every 64Hz tick (every 15.625 ms).
Returns: The counter.

0x03C: Check if time has elapsed

Synopsis: `int RTC_HasElapsed_ms(int start_value, int duration)`

Description: *start_value* holds a timestamp of the starting point. Use `RTC_GetTicks()` to get this value. *duration* holds a time interval in milliseconds. This function returns immediately.
Returns: 0 if the *duration* has not elapsed after the time that *start_value* represents. 1 if the *duration* has elapsed.

0x23E: Set RTC date and time

Synopsis: `void RTC_SetDateTime(uchar **data)`

Description: Sets the RTC to the values given in *data*. *data* is an array of 7 pointers to the BCD encoded date and time details. The order in the array is: YYYY, unused, MM, DD, hh, mm, ss.

As an example, if we would want to set the RTC to 2009-07-04 12:00:00 we would do:

```
unsigned char *data[7] = {{0x20,0x09}, {0}, {0x07}, {0x04}, {0x12},{0},{0}};
```

Returns: (void)

5.7 Software timer functionality

The OS includes ten system-wide timer slots which can be used to execute code after a set interval. The following functions provide 'system-level' access to these timers. Each timer slot is associated with a *TimerID* with a value from 1 to 10. Slot 1 to 5 are used for system purposes, while slot 6 to 10 are allocated for application use through `SetTimer()` and `KillTimer()`. These user timers will be stopped on addin exit by cleanup code included by the official SDK.

0x118: Install timer

Synopsis: `int Timer_Install(int TimerID, void (*handler)(void), int delay);`

Description: Installs a timer to the slot specified by *TimerID*. If *TimerID* is zero, the first unused timer slot will be used. *handler* is the function that is set to be called when the timer expires. *delay* sets the duration of the delay, in multiples of 25ms.
Returns: Negative on failure or the newly installed *TimerID* on success.

0x119: Uninstall timer

Synopsis: `int Timer_Uninstall(int TimerID);`

Description: Frees the selected *TimerID*.

0x11A: Start timer

Synopsis: `int Timer_Start(int TimerID);`

Description: Resets and runs the timer installed at *TimerID*.

0x11A: Stop timer

Synopsis: `int Timer_Stop(int TimerID);`

Description: Resets and stops the timer installed at *TimerID*.

5.8 Setup variable access

Setup is the name of a simple, global array and the functions used to access it. It has room for 100 entries of size *char*. Variables in Setup are used by several of the preinstalled applications, known entries are listed in table **??**. Non-listed entries may be "occupied", but their purpose is unknown. The current list of variables is valid for OS version 1.02, but may be changed in later versions.

0x4DC: Read a Setup variable

Synopsis: `char Setup_GetEntry(unsigned int` *`index`*`);`

Description: `Setup_GetEntry()` is used to retreive a Setup variable with the given *index*.
Returns: The data at the given *index*.

0x4DD: Set a Setup variable

Synopsis: `char *Setup_SetEntry(unsigned int` *`index`*`, char` *`value`*`);`

Description: The function is used to set the specified Setup variable at *index*, to the given *value*
Returns: A pointer to the base of the Setup table

0x4DE: Get the address of a Setup variable

Synopsis: `char *Setup_GetEntryPtr(unsigned int` *`index`*`);`

Description: Returns a pointer to the Setup variable given by *index*.
Returns: The pointer to the Setup variable

5.9 Alpha variable access

The Alpha variables are commonly used in the built-in programs to store floating point values or fractions. Available Alpha variables are *A-Z*, θ, *r* and *Ans*. In functions where the variable has to be specified, variables *A-Z* are represented by their ASCII equivalent (0x41-0x5A). The variables θ, *r* and *Ans* have the character codes 0xCE, 0xCD and 0xC0.

0x4DF: Read an Alpha variable

Synopsis: `char *Alpha_GetData(char variable, char *dest);`

Description: Writes the raw data of the specified Alpha *variable* to the character array pointed to by *dest*. That character array must have room for 24 bytes.
Returns: The passed pointer, *dest*.

0x4E0: Set an Alpha variable

Synopsis: `char Alpha_SetData(char variable, char *src);`

Description: Writes the raw data from the array pointed to by *src* to the Alpha *variable*.
Returns: If *variable* is a valid Alpha variable, 0 is returned. If invalid, *variable* is returned.

0x4E1: Clear all alpha variables

Synopsis: `void Alpha_ClearAlpha(void);`

Description: Clears alpha variables (except *Ans*).
Returns: (void)

0x869: Clear all variables

Synopsis: `void Alpha_ClearAll(void);`

Description: Clears all alpha variables (including *Ans*).
Returns: (void)

5.10 Multibyte string manipulation

To support a large character set like the fx-9860G, multibyte characters are used. The bytes 0x7F, 0xF7, 0xF9, 0xE5, 0xE6 and 0xE7 are lead units, specifying which "character table" to use for the byte that follows it. Most plain ASCII strings won't need special care, but if you want to use the extended character sets, the multibyte-specific functions can be useful.

0x531: Check for lead character

Synopsis: `int MB_IsLead(char character);`

Description: Compares provided *character* with the predefined multibyte lead bytes 0xE5, 0xE6, 0xE7, 0xF7 or 0xF9.
Returns: 1 if character is a lead character. 0 otherwise.

0x533: Get string element count

Synopsis: `int MB_ElementCount(char *str);`

Description: Counts the number of elements in the string pointed to by *str*. A lead byte followed by character byte is considered one element. For example the string [0xF7 0x44][0x36][0x55][0xE8 0x32][0x00], will have 4 elements.
Returns: The number of elements, not counting terminating 0.

0x534: Get string byte count

Synopsis: `int MB_ByteCount(char *str);`

Description: Counts the number of bytes in the string pointed to by *str*. Lead bytes are counted.
Returns: The number of bytes, not counting terminating 0.

0x536: Concatenate strings

Synopsis: `char *MB_strcat(char *dest, char *src);`

Description: Operates like a normal `strcat()`.
Returns: *dest*

0x537: Concatenate strings, limited

Synopsis: `char *MB_strncat(char *dest, char *src, int bytes);`

Description: Operates like a normal `strncat()`. Note that the *bytes* limit is a hard limit, and multibyte elements can be "split".
Returns: *src*

0x538: Copy string

Synopsis: `char *MB_strcpy(char *dest, char *src);`

Description: Operates like a normal `strcpy()`.
Returns: *dest*

0x53C: Get pointer to second string element

Synopsis: `char *MB_GetSecondElemPtr(char *str);`

Description: Returns a pointer to the second element in the string at *str*.
Returns: The pointer.

0x53D: Get first string element data

Synopsis: `short int MB_GetElement(char *str);`

Description: Retreives the first element in the string at *str*.
Returns: The first element, e.g: 0x0041 or 0xF761.

5.11 `libc` standard functions

These functions correspond to their standard C functions with the same name, and are therefore not documented in depth. System calls marked "SDK" are included as system calls in the official SDK. If the functions not marked SDK are used, the SDK will link with object code from the file `fx9860G_library.lib`. This is often a waste of space because the same function could be accessible as a system call. In some cases is the system call code more optimized than the code linked in by the SDK.

Number	Name	SDK-syscall
0xACC	`free()`	✓
0xACD	`malloc()`	✓
0xACE	`memcmp()`	
0xACF	`memcpy()`	
0xAD0	`memset()`	
0xAD4	`strcat()`	
0xAD5	`strcmp()`	
0xAD6	`strlen()`	
0xAD7	`strncat()`	
0xAD8	`strncmp()`	
0xE6B	`calloc()`	✓
0xE6C	`memmove()`	
0xE6D	`realloc()`	✓

Entry.	Name (*title*)	Default	Other values
0x13	`Angle`	1: Radians	0: Degrees, 2: gradians
0x14*	Shift/alpha status	0: Both off	1: Shift on, 4: alpha on, 0x84: both on
0x15*	INS/overwrite status	2: Insert	1: Overwrite
0x17	`Mode`	0: Comp	9: Dec, 0xF: Hex, 1: Bin, 7: Oct
0x18	`Functype`	0: $Y=$	1: $r=$, 2: Param, 3: $X=c$, 4: $Y>$
			5: $Y<$, 6: $Y\geq$, 7: $Y\leq$
0x19	`Draw Type`	0: Con	1: Plot
0x1A	`Derivative`	1: Off	0: On
0x1B	`Coord`	0: On	1: Off
0x1C	`Grid`	1: Off	0: On
0x1D	`Axes`	0: On	1: Off
0x1E	`Label`	1: Off	0: On
0x20	`Stat Wind`	0: Auto	1: Manual
0x21	`Graph Func`	0: On	1: Off
0x22	`Dual Screen`	3: Off	?
0x23	`Simul Graph`	1: Off	0: On
0x24	`Dynamic Type`	1: Stop	0: Cont
0x25	`SigmaDisplay` (Σ *Display*)	1: Off	0: On
0x26	`Slope`	1: Off	0: On
0x27	`Payment`	1: End	0: Bgn
0x28	`Date Mode`	0: 365	1: 360
0x29	`Answer Type`	0: Real	1: Complex
0x2A	`Complex Mode`	0: Real	1: a+bi, 2: $r<T$
0x2B	`Display`	0: Norm1	0x10: Norm2
0x2B	`Display`	0: Norm1	0x30..0x39: Sci0..Sci9
			0x20..0x29: Fix0..Fix9
			Bit 0x80: “/E” mode
0x2C	`Background`	0: None	1..20: Pict1..Pict20
0x2D	`Resid List`	0: None	1..26: List1..List26
0x2E	`List File`	1: File1	02..06: File2..File6
0x2F	`Variable`	0: Range	01..26: List1..List26
0x30*	`Recur Type`	?	?
0x31	`Recur Dual` (*Dual Screen*)	1: Off	2: T+G
0x4E	`Auto Calc`	0: On	1: Off
0x4F	`Show Sell` (*Show Cell*)	0: Formula	1: Value
0x50	`Move`	0: Lower	1: Right
0x51	`Sub Name`	0: On	1: Off
0x53	`Input Mode`	1: Linear	0: Math
0x54	`Locus`	1: Off	0: On
0x55	`Y=Draw Speed`	0: Norm	1: High
0x56	`Sketch Line`	0: Normal	1: Thick, 2: Broken, 3: Dot
0x57	`Frac Result`	0: d/c	1: a + b/c

Entries marked with a * are not found in the global Setup editor.

Changelog

Revision 1⟶2:

Additions

This chapter.
Missing table (**??**) of existing setup entries.
API call 0x005, `App_RefreshAddinTable()`.
API call 0x46B, `App_GetIntegratedCount()`.
API call 0x49A, `App_Run()`.
API call 0x869, `Alpha_ClearAll()`.
API section **??**, on key and keyboard handling.
API section **??**, on multibyte string manipulation.
API section **??**, on display buffer management.
API section **??**, for text printing functions.
API section **??**, for manipulating system timers.
Protocol section **??**, on streaming screen images from the calculator.

Other modifications

API call 0x4E1, renamed: `Alpha_ClearAll()`⟶`Alpha_ClearAlpha()`
API call 0x812, renamed: `Cursor_SetFlashOff()`↛`Cursor_DisableFlash()`
API calls 0x137 & 0x139 replaced with 0x80E & 0x811 (`Cursor_GetFlashStyle()`, `Cursor_EnableFlash()`.
Included keyboard info in the hardware chapter (2).