Programming OTP Bits and Encrypting Firmware in the STMP37xx

Version 1.0
April 9, 2008

Table of Contents

1. Revision History

REVISION	DATE	DESCRIPTION
1.0	04/09/08	Initial Release

2. Scope

This document describes the OTP bits in the STMP37xx devices and the process for creating a downloadable plugin to the Freescale Manufacturing Tool that will burn these OTP bits. It also covers the input file formats, the script used to create and encrypt the OTP burner .sb file, other tools used in the OTP bit burning process and in the encryption process for the standard firmware files, and the procedure used for each of the encryption options.

3. Overview

The STMP37xx family of chips has 1kb of on-chip OTP (One Time Programmable) memory. This memory is used for a number of purposes, from setting boot ROM options to holding a custom crypto key. There are also a number of bits reserved solely for customer use.

An important part of the customer end-product manufacturing process is the burning of these OTP bits. This is especially true for the crypto key, as it is recommended that all customer firmware is encrypted with an encryption key.

Chapter 8 of the STMP37xx data sheet (reference 1) covers the on-chip OTP memory. It describes the customer-accessible registers and pre-defined fields, and what their purposes are. Please read this chapter thoroughly before continuing.

The basic process for OTP manufacturing is as follows:

1. Create a customer AES-128 crypto key file, if a custom key is to be used. This step is not needed if the default key is to be used.
2. Create the OTP bit settings file that specifies the bits to be burned.
3. Run the otp_burner.py Python script and give it the two files mentioned above. This script will produce an OtpInit.sb file (boot image) that is used to burn all of the specified bits in a safe manner.
4. Place the OtpInit.sb file in the Update operation folder in the currently selected profile of the Manufacturing Tool application.

4. Customer Settable OTP Bits

It is recommended that a couple things be taken into consideration when programming the OTP bits available to customers. The crypto bits can be programmed by defining them in the crypto key file described in Section 5, and all other settings bits can be defined in the bit setting file (for example, bit_settings.txt) described in Section 6. Refer to the OCOTP controller section in the STMP37xx data sheet for more information on the procedure for programming these bits.

The customer is free to program and use the following groups of OTP bits:

- A total of 128 bits in four Customer Registers
- A 128-bit encryption value in four Crypto Registers
- Two bit fields in the Customer Capability Register
- Several boot setting fields in three ROM Registers
- Related lock bits in the Lock Register

4.1. HW_OCOTP_CUSTx registers

The four 32-bit HW_OCOTP_CUSTx registers can be used for customer-specified purposes. These are the HW_OCOTP_CUST0, HW_OCOTP_CUST1, HW_OCOTP_CUST2, and HW_OCOTP_CUST3 registers. There are no restrictions on which bits, if any, get programmed or how they are used. The standard SDK for the 37xx does not use or check any of these bits in any of its operations.

4.2. HW_OCOTP_CRYPTOx registers

The four 32-bit HW_OCOTP_CRYPTOx registers can be used to store a custom crypto key. These are the HW_OCOTP_CRYPTO0, HW_OCOTP_CRYPTO1, HW_OCOTP_CRYPTO2, and HW_OCOTP_CRYPTO3. These registers can be left unprogrammed to use the default key, or they can be programmed with a custom key, generated either with the keygen utility (described in next section) or derived by the customer.

4.3. HW_OCOTP_CUSTCAP register

There are two OTP bits in HW_OCOTP_CUSTCAP register that can be set by the customer. Most of the bits in this register are reserved and should not be programmed. In general, capability bits are used to restrict or select the use of specific functional modules within the chip to create a subset of chip definitions using the same chip implementation. Unlike the Customer, Crypto and ROM OTP bits, the customer capability bits are shadowed within memory mapped registers that can be read directly without having to perform a bank open operation to access the OTP array. Following reset, customer capability bits are loaded serially into their shadow register from the OTP array. These shadow bits can be written at any time until the CUSTCAP_SHADOW lock bit has been blown in the HW_OCOTP_LOCK register.

This register has the following bit definitions.

Table 1. HW_OCOTP_CUSTCAP Register Customer Settings

Bits	Label	Definition
2	RTC_XTAL_32768_PRESENT	Set to indicate the presence of an optional 32.768KHz off-chip oscillator.
1	RTC_XTAL_32000_PRESENT	Set to indicate the presence of an optional 32.000KHz off-chip oscillator.

4.4. HW_OCOTP_ROMx registers

The ROM Use bits are described in detail in the STMP37xx Data Sheet, and they are summarized below. Most of the fields in these register pertain to boot options, and extreme care should be taken when programming and locking these registers. Note that fields in these three registers not listed below are reserved bits and they should not be programmed.

Note: In the tables below, some STMP37xx derivatives are singled out as having additional bits or a different meaning for the setting of certain bits. Unless otherwise indicated, the existence of the bits in these registers, and the meaning of these bits, are universal for all STMP37xx derivatives. As new derivatives are added to the family, this document will be updated to reflect any new bits or any unique meaning of existing bits.

Table 2. HW_OCOTP_ROM0 Register Customer Settings

Bits	Label	Definition
31:24	BOOT_MODE	Encoded boot mode.
21:20	SD_POWER_GATE_GPIO	SD Card Power Gate GPIO Pin Select. *For all derivatives except STMP378x:* 00 = PWM3 01 = PWM4 10 = ROTARYA 11 = NO_GATE *For STMP378x:* 00 = PWM0 01 = LCD_DOTCLK 10 = PWM3 11 = NO_GATE
19:14	SD_POWER_UP_DELAY	SD Card Power-Up Delay Required after Enabling GPIO Power Gate: 000000 = 0 ms 000001 = 10 ms 000010 = 20 ms … 111111 = 630 ms
13:12	SD_BUS_WIDTH	SD Card Bus Width. 00 = 4-bit 01 = 1-bit 10 = 8-bit 11 = Reserved
11:8	SSP_SCK_INDEX	Index to SSP clock speed. 0000 = 240kHz 0001 = Slow 0010 = 1MHz 0011 = 2MHz 0100 = 4 MHz 0101 = 6MHz 0110 = 8MHz 0111 = 10MHz 1000 = 12MHz 1001 = 16MHz 1010 = 20MHz 1011 = 24MHz 1100 = 40MHz 1101 = 48MHz 1110 = 240kHz 1111 = Custom
6	DISABLE_SPI_NOR_FAST_READ	Set to disable SPI NOR fast reads which are used by default.
5	ENABLE_USB_BOOT_SERIAL_NUM	Set to enable USB boot serial number.
4	ENABLE_UNENCRYPTED_BOOT	Set to enable unencrypted boot modes.
3	SD_MBR_BOOT	Set to enable SD card master boot record boot mode.
1	USE_ALT_DEBUG_UART_PINS *[For all derivatives except STMP378x; Reserved for STMP378x]*	Use alternate ROTARYA/B Debug UART RX/TX pins.

The ENABLE_UNENCRYPTED_BOOT bit must be set in order to boot unencrypted images. Customers who want to use encrypted images, either with the default key or with a custom key, should not burn this bit and should lock ROM0 so it cannot be burned in the field.

Table 3. HW_OCOTP_ROM1 Register Customer Settings

Bits	Label	Definition
29:28	USE_ALT_GPMI_RDY3 *[For STMP378x only; Reserved for all derivatives except STMP378x]*	Set to cause the ROM NAND driver to enable one of 3 alternate pins for GPMI_RDY3. 00 = GPMI_RDY3 01 = PWM2 10 = LCD_DOTCK
27:26	USE_ALT_GPMI_CE3 *[For STMP378x only; Reserved for all derivatives except STMP378x]*	Set to cause the ROM NAND driver to enable one of 4 alternate pins for GPMI_CE3. 00 = GPMI_D15 01 = LCD_RESET 10 = SSP_DETECT 11 = ROTARYB
25	USE_ALT_GPMI_RDY2 *[For STMP378x only; Reserved for all derivatives except STMP378x]*	Set to cause the ROM NAND driver to enable alternate pins for GPMI_RDY2.
24	USE_ALT_GPMI_CE2 *[For STMP378x only; Reserved for all derivatives except STMP378x]*	Set to cause the ROM NAND driver to enable alternate pins for GPMI_CE2.
23	ENABLE_NAND3_CE_RDY _PULLUP *[For STMP377x/378x only – Reserved for all other derivatives]*	Set to cause the ROM NAND driver to enable internal pull ups for pins GPMI_CE3 and GPMI_RDY3.
22	ENABLE_NAND2_CE_RDY _PULLUP *[For STMP377x/378x only – Reserved for all other derivatives]*	Set to cause the ROM NAND driver to enable internal pull ups for pins GPMI_CE2 and GPMI_RDY2.
21	ENABLE_NAND1_CE_RDY _PULLUP *[For STMP377x/378x only – Reserved for all other derivatives]*	Set to cause the ROM NAND driver to enable internal pull ups for pins GPMI_CE1 and GPMI_RDY1.
20	ENABLE_NAND0_CE_RDY _PULLUP *[For STMP377x/378x only – Reserved for all other derivatives]*	Set to cause the ROM NAND driver to enable internal pull ups for pins GPMI_CE0 and GPMI_RDY0.
19	UNTOUCH_INTERNAL_SSP_PULLUP *[For STMP377x/378x only – Reserved for all other derivatives]*	*For STMP377x:* This bit can be set to cause the ROM to ignore bits 17-18 in this register which are used to disable internal pull ups. *For STMP378x:* When this bit is set then internal SSP pull-ups are neither enabled or disabled. This bit is used only if external pull-ups are implemented and either bits 17 or 18 in this register is set.
18	SSP2_EXT_PULLUP	Set to indicate external pullups are implemented for SSP2.
17	SSP1_EXT_PULLUP	Set to indicate external pullups are implemented for SSP1.

Bits	Label	Definition
16	SD_INCREASE_INIT_SEQ_TIME	Set to increase the SD card initialization sequence time from 1 ms (default) to 4 ms.
15	SD_INIT_SEQ_2_ENABLE	Set to enable the second initialization sequence for SD boot.
14	SD_CMD0_DISABLE	Set to disable Command0 in the SD card.
13	SD_INIT_SEQ_1_DISABLE	Set to disable the first initialization sequence in the SD card.
12	USE_ALTERNATE_CE *[For all derivatives except STMP378x; Reserved for STMP378x]*	Set to direct boot loader to use the alternate chip enables.
11:8	BOOT_SEARCH_COUNT	Set to specify the number of 64-page blocks that should be read by the boot loader.
2:0	NUMBER_OF_NANDS	Encoded value indicates number of external NAND devices (0 to 7). 0 = Indicates ROM will probe for the number of NAND devices connected in the system.

Important! For the STMP377x/378x, in order to take advantage of the internal pull-up resistors on the GPMI_CEx & GPMI_RDYx pins by the setting of bits 23:20, the NUMBER_OF_NANDS OTP field, 2:0, must be set to match the number of NANDs in the system. Also, note that there is a bug in the 3770 TA1 ROM code that prevents NUMBER_OF_NANDS from exceeding 2, which means the NAND2 and NAND3 internal CE & RDY pull-ups cannot be used! So external pull-ups will be required for CE2, RDY2, CE3 and RDY3.

Care should be taken when programming the BOOT_SEARCH_COUNT field. The default value for this OTP field is interpreted in the SDK as a 1, but this value causes the window size to be two 64-page strides, and if any block in the BCB is bad, then there will not be any other place to put the boot block and the allocation will fail. To boot from the NAND, ROM needs to find one set of BCB's (NCB, LDLB and DBBT) intact. Because certain NANDs (for examle, MLC) are unreliable, multiple copies of BCBs should be maintained on the NAND and each BCB should appear in its own search area. This search area/window is configured by setting this boot search OTP field.

One of the main reasons for a bigger search window is to work around bad blocks at manufacturing time, such that if a bad block is encountered at the beginning of the NAND (where BCB's reside), then a wider search window will allow the placing of the BCB in other blocks within that window size. Therefore, customers should set this field to something other than the default value. The recommended setting for this BOOT_SEARCH_COUNT field is 3, which means that there will be 1<<3 = 8 strides of 64 pages that will be searched for the boot control block. For a 4k per page NAND, this results in 8 * 64 * 4K = 2048KB, which is a 4-block search for this type of NAND. See section 34.9 – NAND Boot Mode in the STMP37xx Data Sheet for background information about NAND booting.

Table 4. HW_OCOTP_ROM2 Register Customer Settings

Bits	Label	Definition
31:16	USB_VID	USB Vendor ID used only in recovery mode. If field is 0, then Freescale vendor ID is used.
15:0	USB_PID	USB Product ID used only in recovery mode.

Important! If these fields are changed, then a customized updater needs to be used. Keep the default values for compatibility with standard updater and Manufacturing Tool.

4.5. HW_OCOTP_LOCK register

The bits in the HW_OCOTP_LOCK register allow and reflect the locking of the customer-accessible OTP bits/registers described above. Fields in this register not listed below are reserved bits, and they should not be programmed. Once a register is locked, that register cannot be changed ever.

Table 5. HW_OCOTP_LOCK Register Customer Settings

Bits	Label	Definition
26	ROM2	Status of ROM2 register's write lock bit. When set, register is locked.
25	ROM1	Status of ROM1 register's write lock bit. When set, register is locked.
24	ROM0	Status of ROM0 register's write lock bit. When set, register is locked.
22	CRYPTODCP_ALT	Status of alternate bit for CRYPTODCP lock.
21	CRYPTOKEY_ALT	Status of alternate bit for CRYPTOKEY lock.
9	CUSTCAP	Status of Customer Capability lock.
7	CUSTCAP_SHADOW	Status of Customer Capability shadow register lock. When set, override of customer capability shadow bits is blocked.
5	CRYPTODCP	Status of read lock bit for DCP APB crypto access. When set, the DCP disallows reads of its crypto keys via its APB interface.
4	CRYPTOKEY	Status of crypto registers' read/write lock bit. When set, registers are locked.
3	CUST3	Status of CUST3 register's write lock bit. When set, register is locked.
2	CUST2	Status of CUST2 register's write lock bit. When set, register is locked.
1	CUST1	Status of CUST1 register's write lock bit. When set, register is locked.
0	CUST0	Status of CUST0 register's write lock bit. When set, register is locked.

5. Crypto Key File

A custom AES-128 crypto key is stored in a file with a very simple format. A key, which is entered on a single line in a key file, is an uninterrupted string of 32 hexadecimal characters, for a total of 128 bits of key data. Any number of keys may appear in a key file, each on a separate line. The line ending format does not matter.

```
3F3CFBC001F399991035C3C6C7065924
```

Example 1. Example Key

Normally, customers will never need to manually create a key file, although this is certainly possible. Instead, the keygen tool should be used to generate a key that is sufficiently random for cryptographic use. To create a key file, run:

```
keygen customkey.txt
```

on the command line. A new “customkey.txt” file will be created that contains a single randomly generated key.

Important! The contents of a key file are in plain text, and as such, Freescale recommends that customer key files are stored on a single workstation and never moved to other computers. Additionally, it is recommended that key files are stored in an encrypted zip file or similar encrypted storage and only extracted from this storage in order to produce a manufacturing build of the firmware. Developers should never need the key file because they can use hardware with the default crypto key for all development.

Important! There are potential customer return complications for application and operations resulting from use a of a custom key. Specifically, customers that intend to use a custom key should be aware that if they expect Freescale to analyze their returned devices, they will have to do the following:

- Provide the bit_settings.txt file used in generating the OtpInit.sb file for programming the OTP bits in the returned devices.
- Provide the OtpInit.sb file that was generated with the customer’s custom key.
- Follow the encryption and .sb file generation instructions of section 8 in this document to create encrypted images, based on the customer’s custom key, of an appropriate example player build.
- Provide the encrypted images (firmware.sb and Updater.sb) of the example player and actual customer production images to Freescale for analysis.

6. OTP Bit Settings Files

An important input to the OTP manufacturing script is the bit settings file, a text file that lists all of the bit values that the customer wishes to program. Any OTP bit can be assigned a value. It is even possible to set a custom crypto key with this file; however, that is not recommended. But remember, when it comes to actually burning the bits in a device, some OTP registers may already be locked by lock bits, so only unlocked bits will be burned. This is only likely to be an issue if the manufacturing process is executed multiple times.

An example OTP bit settings file, called bit_settings.txt, has been provided as an a accompaniment to this document. It demonstrates the format and syntax described herein, includes commands to set each register or bit field that can be set by customers, and lists the available constant (enumerated) types that are supported for each bit field. The file, as provided,

sets the bit fields to the default or recommended value, but can be edited as needed by the customer.

6.1. Format

The format of the bit settings file is very straightforward. Each line simply contains an assignment statement that applies a value to a bit field or a subrange of a bit field.

```
*chip-family-3700*

# Set whole customer registers to some values.
hw_ocotp_cust0 = 0x10000
hw_ocotp_cust1 = 0x20000

# Setting part of a register.
hw_ocotp_cust2[31:15] = 99999

# You can also set just one bit.
hw_ocotp_cust2[0] = 1

# Set bit fields by name.
use_alt_debug_uart_pins = yes
```

Example 2. OTP Bit Settings File Syntax

The first line in Example 2 above is a pragma statement that sets the chip family for which the file is setting bits. The chip family selects the set of available bit field names, as some chips have different or additional bits or fields defined. Every bit settings file should include this pragma so that they can remain unchanged when future chip families are supported.

6.2. Bit Fields

Refer to the STMP37xx data sheet, or the tables of section 4 above, for the list of bit field names that may be used on the left hand side of the assignment statements. Bit field names match the data sheet exactly, except that they are not case sensitive. Both whole register names and fields within a register are available, and in fact work exactly the same way. If a bit is assigned a value more than once within a file, the last assignment takes precedence.

The only bit field names that do not match those in the data sheet are the ones for the lock bits. This is because the lock bit names are somewhat ambiguous, and are not unique. To solve this, each lock bit simply has the prefix “lock_” added to it to form the bit field name.

Table 6. HW_OCOTP_LOCK Register Bit Names

Bit Field Name	Lock Register Field	Bit Number
lock_rom2	ROM2	26
lock_rom1	ROM1	25
lock_rom0	ROM0	24
lock_cryptodcp_alt	CRYPTODCP_ALT	22
lock_cryptokey_alt	CRYPTOKEY_ALT	21
lock_custcap	CUSTCAP	9
lock_custcap_shadow	CUSTCAP_SHADOW	7
lock_cryptodcp	CRYPTODCP	5
lock_cryptokey	CRYPTOKEY	4
lock_cust3	CUST3	3
lock_cust2	CUST2	2
lock_cust1	CUST1	1
lock_cust0	CUST0	0

6.3. Constants

Some of the fields in the ROM OTP registers have enumerated values and consist of more than one bit. In order to make the bit settings file easier to read and understand, constant names can be used as the values of these fields. In addition, there are several general purpose constants also intended to make the file easier to read.

The three bit fields that have enumerated values are:

- boot_mode
- sd_power_gate_gpio
- sd_bus_width

All three fields belong to register HW_OCOTP_ROM0. The constants for the values of these bit fields all have the field name as a prefix. The tables below list all of the constant names for the enumerated values of these bit fields.

Table 7. Constants for Boot_mode Bits

Value	Constant Name
0x11	boot_mode_i2c_1v8
0x01	boot_mode_i2c_3v3
0x06	boot_mode_jtag_wait
0x14	boot_mode_nand_ecc4_1v8
0x04	boot_mode_nand_ecc4_3v3
0x1c	boot_mode_nand_ecc8_1v8
0x0c	boot_mode_nand_ecc8_3v3
0x15	boot_mode_nor_16bit_1v8
0x05	boot_mode_nor_16bit_3v3
0x1d	boot_mode_nor_8bit_1v8
0x0d	boot_mode_nor_8bit_3v3
0x19	boot_mode_sdmmc_ssp1_1v8
0x09	boot_mode_sdmmc_ssp1_3v3
0x1a	boot_mode_sdmmc_ssp2_1v8
0x0a	boot_mode_sdmmc_ssp2_3v3
0x18	boot_mode_spi_eeprom_ssp2_1v8
0x08	boot_mode_spi_eeprom_ssp2_3v3
0x12	boot_mode_spi_flash_ssp1_1v8
0x02	boot_mode_spi_flash_ssp1_3v3
0x13	boot_mode_spi_flash_ssp2_1v8
0x03	boot_mode_spi_flash_ssp2_3v3
0x00	boot_mode_usb

Table 8. Constants for sd_power_gate_gpio Bits

Value	Constant Name	
	For All Derivatives Except STMP3780	For STMP3780
0x0	sd_power_gate_gpio_pwm3	sd_power_gate_gpio_pwm0
0x1	sd_power_gate_gpio_pwm4	sd_power_gate_gpio_lcd_dotclk
0x2	sd_power_gate_gpio_rotarya	sd_power_gate_gpio_pwm3
0x3	sd_power_gate_gpio_no_gate	sd_power_gate_gpio_no_gate

Table 9. Constants for sd_bus_width Bits

Value	Constant Name
0x0	sd_bus_width_4bit
0x1	sd_bus_width_1bit
0x2	sd_bus_width_8bit

In addition to the above constants, there are several general ones to represent the Boolean values 0 and 1. These are listed in the table below.

Table 10. Generic Constants

Positive	Negative
yes	no
true	false
on	off

7. Encryption Tools

7.1. keygen.exe

The keygen tool is used to create a file to hold an encryption key. The command syntax is as follows:

```
keygen [-? | --help] [-v | --version] [-q | --quiet]
       [-V | --verbose] [-n | --number <int>] key-filename
```

where the command line options are defined in the following table.

Table 11. keygen.exe Command Line Options

Long Form	Short Form	Action
--help	-?	Show help
--version	-v	Display tool version
--quiet	-q	Output only warnings and errors
--verbose	-V	Output extra detailed log information
--number <int>	-n <int>	Number of keys to generate per file (default=1)

For our use, when creating a key file, the form of the command will be:

```
keygen customkey.txt
```

This will generate a key file called customkey.txt, and it will contain a pseudo-randomly generated custom key.

7.2. Otp_burner.py Script

This Python script is the central piece in preparing to burn OTP bits during customer manufacturing. It accepts a bit settings file and, optionally, a crypto key file and produces an .sb file. The command syntax is as follows:

```
otp_burner.py –o  [-h | --help] [-v | --version]
                  [-i <PATH> | --input=PATH]
                  [-o <PATH> | --output=PATH]
                  [-k <PATH> | --key=PATH]
                  [-n <NUM> | --key-number=NUM]
                  [-p or --print-otp] [-e | --elftosb]
                  [-E | --no-elftosb]
```

where the command line options are defined in the following table.

Table 12. OTP_burner.py Command Line Options

Long Form	Short Form	Action
--help	-h	Show help message and exit
--input=PATH	-i PATH	Specify the input OTP bit settings file
--output=PATH	-o PATH	Write output to this file. Optional; if not provided then the output file name is OtpInit.sb.
--key=PATH	-k PATH	Specify the input crypto key file
--key-number=NUM	-n NUM	Use key number N (base 0) from the crypto key file (default 0 – the first key); the last key in the file is referenced by (number of total keys – 1)
--print-otp	-p	Print the resulting OTP registers
--elftosb	-e	Run elftosb to generate the .sb file (the default)
--no-elftosb	-E	Do not run elftosb

The basic command line is:

```
otp_burner.py -i BIT_SETTINGS_FILE -o [OUTPUT_FILE]
[-k KEY_FILE]
```

Many customers will have a custom crypto key, so they should additionally add the “-k KEY_FILE” option to the command line. This will fill in the HW_OCOTP_CRYPTO[0-3] registers with the correct values to be able to boot firmware encrypted with the provided key. In cases where the key file has more than one key, the number of the key to use can be selected with the “-n KEY_NUMBER” command line option.

For our use, the specific command line used when using a custom key is:

```
otp_burner.py -i bit_settings.txt -k customkey.txt
```

This will cause the otp_burner.py script to use bit_settings.txt file as the input and to use the key listed in the customkey.txt file as the encryption key. The OtpInit.sb file generated by the execution of this script is used by the Manufacturing Tool in its OTP burning step. The Manufacturing Tool will not work if this file is not named “OtpInit.sb”. If the –o option is omitted, the default output file will be OtpInit.sb. In place of bit_settings.txt, refer to the actual path and name of the OTP bit settings file in the format described above.

7.3. elftosb.exe

The elftosb tool is used to create the .sb file. It can be called automatically by the OTP_burner.py script as described in the previous section. Note that elftosb generates a new random session key each time it is invoked, so running elftosb twice with the same inputs generates two unique boot images. However, both images are signed with the same keys, and they will boot on the same devices. The signing process is cryptographically secure and cannot be subverted by simple attacks on the image file.

The command syntax is as follows:

```
elftosb     [-? | --help] [-v | --version]
            [-f | --chip-family <family>]
            [-c | --command <file>] [-o | --output <sfile>]
            [-P | --product <version>]
            [-C | --component <version>]
            [-k | --key <file>] [-z | --zero-key]
            [-s | --key-set <value>] [-D | --define <const>]
            [-O | --option <option>]
            [-d | --debug] [-q | --quiet] [-V | --verbose]
            [-p | --search-path <path>] files...
```

where the command line options are defined in the following table.

Table 13. elftosb.exe Command Line Options

Long Form	Short Form	Action
--help	-?	Show help
--version	-v	Display tool version
--key <file>	-k <file>	Add OTP key used for decryption
--chip-family 36xx | 37xx	-f 36xx | 37xx	Select the chip family (default is 37xx)
--command <file>	-c <file>	Use this command file
--output <file>	-o <file>	Write output to this file
--search-path <path>	-p <path>	Add a search path used to find input files
--product <version>	-P <version>	Set product version
--component <version>	-C <version>	Set component version
--key <file>	-k <file>	Add OTP key, enable encryption (37xx)
--zero-key	-z	Add default key of all zeroes (37xx)
--key-set <file>	-s <file>	Specify key set (36xx)
--define <const>=<int>	-D <const>=<int>	Define or override a constant value
--option <name>=<value>	-O <name>=<value>	Set or override a processing option
--debug	-d	Enable debug output
--quiet	-q	Output only warnings and errors
--verbose	-V	Output extra detailed log information

7.4. sbtool.exe

The sbtool is used to verify an encrypted image. The command syntax is as follows:

```
sbtool [-? | --help] [-v | --version] [-k | --key <file>]
       [-z | --zero-key] [-x | --extract <value>]
       [-b | --binary] [-d | --debug] [-q | --quiet]
       [-V | --verbose] sb-file
```

where the command line options are defined in the following table.

Table 14. sbtool.exe Command Line Options

Long Form	Short Form	Action
--help	-?	Show help
--version	-v	Display tool version
--key <file>	-k <file>	Add OTP key used for decryption
--zero-key	-z	Add default key of all zeroes
--extract <index>	-x <index>	Extract section number <index>
--binary	-b	Extract section data as binary
--debug	-d	Enable debug output
--quiet	-q	Output only warnings and errors
--verbose	-V	Output extra detailed log information

8. Encryption

There are three options for a customer in choosing his type of encryption: no encryption, encrypting with the default key, and encrypting with a custom key. This section describes the procedure that a customer should follow when encrypting an image; however, using an unencrypted image is not recommended and is therefore not described in this document. For those customers who absolutely must use unencrypted images, unencrypted images can be generated and are enabled by setting the ENABLE_UNENCRYPTED_BOOT bit in the HW_OCOTP_ROM0 register.

If the customer wants to use his own custom key, he can enter his key in a key file. If he wants to randomly generate a key, the keygen tool should be used. Encrypted images are encrypted using a randomly generated session key and are digitally signed using one or more OTP keys. The 37xx ROM will only boot encrypted images that have been signed with the key burned into the device's OTP bits.

The OTP_burner.py script uses the bit settings file, here called bit_settings.txt and optionally uses the encryption key file, called customkey.txt, to create OtpInit.sb that is used by the Manufacturing Tool to burn the OTP bits. This script calls the elftosb tool to create the binary with the selected encryption. After an .sb file is created, the sbtool tool can be used to verify that the image has been signed with the desired key.

When using any of the encryption options described in the sections below, creating or editing the OTP bit settings file is a prerequisite. After creating this OTP bit setting file, move or copy it to the SOCFirmware\bin folder. To program only customer OTP bits, execute this command:

```
otp_burner.py -i bit_settings.txt
```

This python script creates the OtpInit.sb file.

Note that the type of encryption used will determine which command line flags, or options, must be used when running elftosb, or verifying with sbtool, for each binary. The following table specifies the flag(s) used for each encryption type and binary generation, where:

-k = Add OTP key, enable encryption (37xx)

–z = Add default key of all zeroes (37xx)

-k –z = Add both default and custom key

Table 15. elftosb Command Line Options by Encryption Type

Binary File	Default Key Encrypted	Custom Key Encrypted	Unencrypted (Not Recommended)	Notes
otpinit.sb	-z	-z –k	-z	Booted before and after OTP bits are programmed
updater.sb	-z	-z –k	-z	Booted before and after OTP bits are programmed. For an unencrypted updater, there will need to be another updater binary created without –z option for updating after the intial manufacturing process.
firmware.sb	-z	-k	None	Booted only after OTP bits are programmed.

8.1. Using the Default Key to Generate an Encrypted Image

When a customer wishes to use the default key for his encryption, he doesn't need to supply a crypto key to program the Crypto registers. The default key is used for all devices that do not have a customized key stored in the OTP. When a custom crypto key file is not present and the OTP bit ROM0.ENABLE_UNENCRYPTED_BOOT bit is not set, then encryption with the default key is done, and the following procedure should be followed.

8.1.1. Edit the bit settings file

Create or edit the OTP bit settings file. After editing this OTP bit setting file, move or copy it to the SOCFirmware\bin folder.

Customers who use encrypted images, either with the default key or with a custom key, should lock ROM0 so it cannot be burned in the field. The customer can lock ROM0 by adding the following to the bit settings file:

```
lock_rom0 = true
```

When a customer runs otp_burner.py using the default crypto key (that is, no -k key_file), the script does not automatically set the lock bits for the OTP crypto key. So the customer should add the following two lines in his settings file:

```
lock_cryptokey = true
lock_cryptokey_alt = true
```

Locking the crypto key bits prevents someone from overwriting the key with, for example, all FFs.

8.1.2. Create the OtpInit.sb file

At the SOCFirmware\bin DOS prompt, check to see if the following files exist. If they are missing, unzip the accompanying otp_burner.zip file into this bin folder.

- otp_burner.py
- otp_burner.bd
- pitc_otp_mfg
- rom.key

To create the OtpInit.sb with only the default key, execute this command:

```
otp_burner.py -i bit_settings.txt
```

8.1.3. Verify the OtpInit.sb file

To verify that the correct (default) key has been used to encrypt the file, execute this command:

```
sbtool -z OtpInit.sb
```

8.1.4. Burn the OTP bits

Use the Freescale Manufacturing Tool to burn OTP bits before updating the device's firmware. The Manufacturing Tool looks for OtpInit.sb in its firmware profile, so copy or move the OtpInit.sb file to the firmware folder under the Manufacturing Tool's profile. To verify the file, execute this command:

```
sbtool -z OtpInit.sb
```

8.1.5. Create encrypted device firmware

The SDK project encrypts the .sb files with Freescale's default key (elftosb using –z option) and there is no need to modify the project files.

1. Copy the firmware files (firmware.sb and updater.sb) to the Manufacturing Tool's profile.
2. Check updater.sb by running this command:

   ```
   sbtool -z updater.sb
   ```

3. Check firmware.sb by running this command:

   ```
   sbtool –z firmware.sb
   ```

4. Copy firmware to the Manufacturing Tool's profile so that it has the otpinit.sb files:
5. Configure and run the Manufacturing Tool. If the correct firmware is used, the tool should complete without error. For an encrypted device, the ROM output to the debug port can be used to check if there is any ROM error during the manufacturing process.

8.2. Using a Custom Key to Generate an Encrypted Image

The process for generating a custom crypto key and creating the encrypted firmware is described in this section. Customers can create their own key to encrypt their firmware. The custom key, along with other customer-set OTP bits, are programmed into the OTP ROM by the Freescale Manufacturing Tool when the OtpInit.sb file is present. If there is no key represented by programmed OTP bits in OtpInit.sb, then the device will use the default key, while other OTP bits may be burned by the Manufacturing Tool.

On a customer's manufacturing line, a 37xx will start with the default OTP key and finish with the customer's OTP key. Thus the manufacturing images (that is, OtpInit.sb and updater.sb) should be signed with both keys (-z –k), so they are guaranteed to boot before and after burning the OTP bits. The player firmware, firmware.sb, will be encrypted with only the custom key.

8.2.1. Generating a custom key

To generate a randomly generated custom key, first open a DOS window and navigate to the SOCFirmware\bin directory. At the DOS prompt, enter:

```
keygen customkey.txt
```

8.2.2. Edit the bit settings file

Edit the OTP bit settings file. After editing this OTP bit setting file, move or copy it to the SOCFirmware\bin folder.

Customers who use encrypted images, either with the default key or with a custom key, should lock ROM0 so it cannot be burned in the field. The customer can lock ROM0 by adding the following to the bit settings file:

```
lock_rom0 = true
```

When a customer runs otp_burner.py and supplies a custom crypto key (i.e., -k key_file), the script automatically sets the lock bits for the OTP crypto key. So it behaves as if the customer had the following two lines in his settings file:

```
lock_cryptokey = true
lock_cryptokey_alt = true
```

Locking the crypto key bits prevents someone from overwriting the key with, for example, all FFs. To make the OTP crypto key completely secure, however, the customer must burn two additional bits. For debug purposes, the DCP block provides a backdoor that allows the ARM core to read the OTP crypto key. This backdoor is disabled by setting two bits in the OTP LOCK register. Therefore, customers that burn a private OTP crypto key should add the following two lines to their settings file:

```
lock_cryptodcp = true
lock_cryptodcp_alt = true
```

8.2.3. Create the OtpInit.sb file

At the SOCFirmware\bin DOS prompt, check to see if the following files exist. If they are missing, unzip the accompanying otp_burner.zip file into this bin folder.

- otp_burner.py
- otp_burner.bd
- pitc_otp_mfg
- rom.key

To customize the .sb file by programming OTP bits with a customized key, assuming the key is stored in customkey.txt, execute this command:

```
otp_burner.py -i bit_settings.txt -n 0 -k
customkey.txt
```

to create the OtpInit.sb with the default key and the key listed in the customkey.txt file.

8.2.4. Verify the OtpInit.sb file

Execute these commands:

```
sbtool -z OtpInit.sb
```

and

```
sbtool -k customkey.txt OtpInit.sb
```

to verify that the correct keys have been used to encrypt the file.

8.2.5. Burn the OTP bits

Use the Freescale Manufacturing Tool to burn OTP bits before updating the device's firmware. The Manufacturing Tool looks for OtpInit.sb in its firmware profile, so copy or move the OtpInit.sb file to the firmware folder under the Manufacturing Tool's profile.

8.2.6. Create encrypted device firmware

The SDK project encrypts the .sb files with Freescale's default key (elftosb using –z option). The project must be modified to use the custom OTP key.

1. The project files with the final encrypted firmware must include the custom key. Since the project_config.py tool overrides the top project files and the original project files should be preserved, create a new project folder and copy all project files to it.
2. Copy the key file to the working project file. In this example, we'll copy customkey.txt to the SOCFirmware\applications\examples\players\cinema folder.

3. Create the encrypted Updater.sb by first modifying the parameters in the elftosb post-execution line to include the -k customkey.txt option in the updater top-level project as follows. The -z option in the elftosb post-execution shell of the SDK project instructs the elftosb to use the Freescale default key for encrypting the .sb files.

```
:postexecShell="$ROOT
bin
elftosb -V -d -z -k customkey.txt -f
$CHIP_FAMILY -P 5.000.581 -C 5.000.581 -c
$OUTDIR
sdk_os_media_updater.bd -o $OUTDIR
$THIS_PROFILE_NAME.sb -p $OUTDIR
$THIS_PROFILE_NAME
$THIS_PROFILE_NAME"
```

4. Build the project.
5. Check updater.sb by running these commands:

```
sbtool -z updater.sb
```

and

```
sbtool -k customkey.txt updater.sb
```

6. Build the hostlink.
7. Create the encrypted firmware.sb file by removing the -z and add the -k customkey.txt option to the elftosb shell command line in the player top level project as follows:

```
:postexecShell="$ROOT
bin
elftosb -V -d -k customkey.txt -f $CHIP_FAMILY -
c $OUTDIR
sdk_os_media_player.bd -o $OUTDIR
firmware.sb -p $OUTDIR -P 5.000.581 -C
5.000.581"
```

8. Build the project.
9. Check firmware.sb by running this command:

```
sbtool –z firmware.sb
```

It should fail with the default key, but pass, using this command

```
sbtool -k customkey.txt firmware.sb
```

10. Copy firmware to the Manufacturing Tool's profile so that it has the following .sb files:
    - otpinit.sb
    - firmware.sb
    - updater.sb
11. Configure and run the Manufacturing Tool. If the correct firmware is used, the tool should complete without error. For an encrypted device, the ROM output to the debug port can be used to check if there is any ROM error during the manufacturing process.

9. References

1. *STMP37xx Product Data Sheet*, Version 1.00, Document #5-37xx-DS1-1.00-090707, Freescale, Inc.

10. Appendix: OTP Bit Settings File Grammar

The grammar for the OTP bit settings file format is presented below in Extended Backus-Naur Format (EBNF) for reference.

```
input               ::=  statement*
                    ;

statement           ::   pragma
                    |    assignment
                    ;

pragma              ::=  '*' IDENT '*'
                    ;

statement           ::=  IDENT bit-slice? '=' value
                    ;

bit-slice           ::=  '[' INTEGER ( ':' INTEGER )? ']'
                    ;

value               ::=  INTEGER
                    |    IDENT
                    ;
```