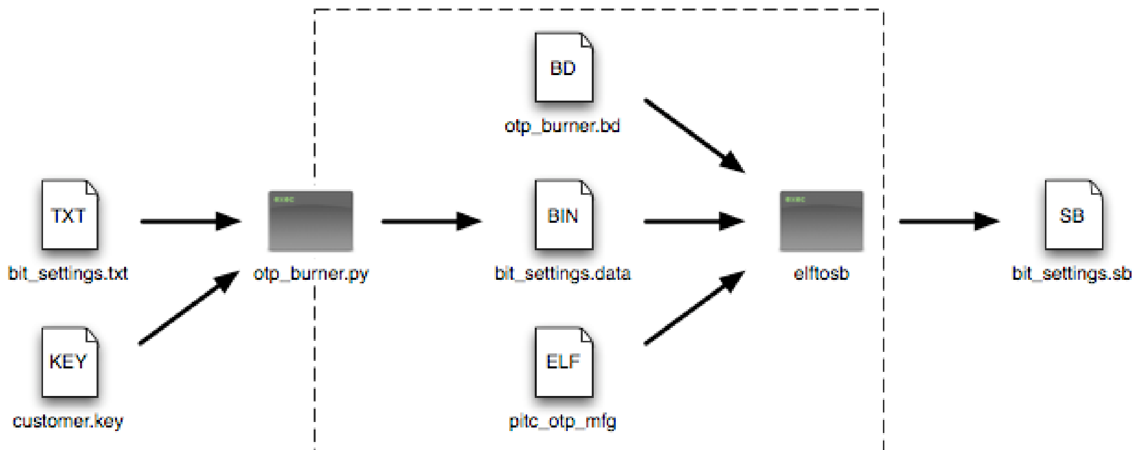
otp_burner.py documentation

Introduction

The `otp_burner.py` script is used to generate an .sb file that burns OTP bits specified by the customer. This output .sb file is then used by the manufacturing tool or with `BitInit.exe`.

The data flow through the `otp_burner.py` script looks like this:



The dashed rectangle surrounds the operations that are under control of the Python script. The `bit_settings.data` file is a temporary intermediate file that is produced by the script as an input to elftosb. It holds a binary representation of all of the OTP register values. The `otp_burner.bd` and `pitc_otp_mfg` files are both parts of the script distribution and do not change.

Usage

Here is the command line help output from version 1.3.

```
Usage
=====
  otp_burner.py [options] [-k FILE] [-r FILE] -i FILE [-o FILE]

Options
=======
--help, -h              show this help message and exit
--version, -V           Show version information.
--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 generated from the input
                        file name.
--key=PATH, -k PATH     Specify the input crypto key file.
--key-number=NUM, -n NUM
                        Use key number N from the crypto key file (default 0).
--srk=PATH, -r PATH     Specify the binary super root key hash file.
--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.
--hab, -a               generate a HAB compatible .sb (MX28 only; default)
--no-hab, -A            generate a non-HAB .sb (MX28 only)
```

The `--input` argument specifies the bit settings file, the format of which is described in the section below. This argument is always required. You can change the name of the output file with the `--output` option. By default, the output file will be named `OtpInit.sb`, as this is what the MfgTool expects.

Normally, otp_burner.py will run automatically elftosb itself to produce the output .sb file. If you do not need this feature, for instance if you are using custom bit burning firmware, then you can disable it with the `--no-elftosb` option.

An intermediate data file containing the binary representation of the OTP registers is always created. The default name of this file is the input file name with the extension changed to .dat. If you disable automatic running of elftosb, the `--output` option is used to set the name of this file instead of the .sb file (since no .sb file is produced automatically).

Another important option is `--key`. It tells the script to put an AES-128 key into the OTP key registers. The key file format is the same as that used by elftosb and generated by keygen. If you have multiple keys in the provided key file, you can select which one to place in OTP with the `--key-number` option.

The i.MX28 has a couple special command line options that apply only to it. These options are ignored unless the MX28 family is specified by the chip family pragma in the bit settings file (see below for more about these pragmas). Similar to `--key`, the `--srk` option lets you select a binary file containing the SHA-256 hash of the Super Root Key for placement in the SRK OTP registers. Unless you are using code signed HAB4 images, you can ignore this option.

The other MX28 only options form a pair, to enable and disable HAB support in the output .sb file. They are `--hab` and `--no-hab`. By default when the bit settings file specifies the MX28 family, the output .sb will be HAB enabled. If you turn off automatic elftosb execution, these options have no effect.

Bit settings file

The bit settings file is the main input file to the Python script. It contains a list of all the OTP bits that are to be burned by the output .sb file. The format is very simple and easy to read.

Chip family

Every bit settings file should begin with a "chip family pragma statement". This statement declares the chip family on which the OTP bits will be burned. The main effect of this is to change the set of recognized register and field names to match the OTP registers for the given chip family. Setting the chip family to MX28 will also cause the output .sb file to be HAB enabled by default. The `--no-hab` command line option can be used to disable this.

Supported chip families and the associated pragma statement are shown in this table:

Chip family	Pragma statement
STMP37xx	`*chip-family-3700*`
STMP377x	`*chip-family-3770*`
STMP378x	`*chip-family-3780*`
i.MX23	`*chip-family-mx23*`
i.MX28	`*chip-family-mx28*`

Table 1. Chip family pragmas

Bit field assignment

Each line in the bit settings file sets a range of bits in an OTP register to a value. The basic assignment statement structure looks like this:

```
hw_ocotp_cust0 = 1234
```

This statement sets all bits of the named register to the given value. The left hand side of the assignment specifies the register to modify, either by naming the register itself or one of the fields of a register. Further examples will be given below. The right hand side of the assignment statement is either a decimal or hexidecimal integer value, or one of a number of constant identifiers. These constants are listed in detail below. Notice that there is no semicolon at the end of the statement.

You can also set the value of one bit or a range of bits of the named register using an array syntax. This looks like:

```
# One bit
hw_ocotp_cust0[12] = 1

# A bit range
hw_ocotp_cust1[31:15] = 0xff
```

Blank lines and comment lines starting with '#' are also accepted, as shown above. Comments may also start on a statement line, and run until the end of the line. Extra whitespace throughout the file is ignored and has no effect.

Register names

The left hand side of any assignment statement can contain the name of any available register or field for the selected chip family. Register and field names are not case sensitive. As described above, you can set individual bit values within the register or field by using array or slice syntax.

These are the register names for all chip families:

Bank 0	Bank 1	Bank 2	Bank 3	Bank 4 (i.MX28 only)
hw_ocotp_cust0	hw_ocotp_hwcap0	hw_ocotp_lock	hw_ocotp_rom0	hw_ocotp_srk0
hw_ocotp_cust1	hw_ocotp_hwcap1	hw_ocotp_ops0	hw_ocotp_rom1	hw_ocotp_srk1
hw_ocotp_cust2	hw_ocotp_hwcap2	hw_ocotp_ops1	hw_ocotp_rom2	hw_ocotp_srk2
hw_ocotp_cust3	hw_ocotp_hwcap3	hw_ocotp_ops2	hw_ocotp_rom3	hw_ocotp_srk3
hw_ocotp_crypto0	hw_ocotp_hwcap4	hw_ocotp_ops3	hw_ocotp_rom4	hw_ocotp_srk4
hw_ocotp_crypto1	hw_ocotp_hwcap5	hw_ocotp_un0	hw_ocotp_rom5	hw_ocotp_srk5
hw_ocotp_crypto2	hw_ocotp_swcap	hw_ocotp_un1	hw_ocotp_rom6	hw_ocotp_srk6
hw_ocotp_crypto3	hw_ocotp_custcap	hw_ocotp_un2	hw_ocotp_rom7	hw_ocotp_srk7

Table 2. OTP register names

As shown in the table, bank 4 is only present on the i.MX28 family.

Field names

In addition to the register names above, there is also a field name identifier for each of the bit fields listed in those registers in the datasheet. The actual set of field names varies for each chip family, sometimes considerably. The field name identifiers match the field names in the datasheet exactly, except that they are not case sensitive. The only field name identifiers that differ from the datasheet are those for the HW_OCOTP_LOCK register, in which case the corresponding identifiers are prefixed with "`lock_`".

For instance, on the i.MX28 the register `hw_ocotp_rom4` contains the following fields as listed in the database.

Field name	Equivalent
nand_badblock_marker_reserve	hw_ocotp_rom4[31]
nand_read_cmd_code2	hw_ocotp_rom4[23:16]
nand_read_cmd_code1	hw_ocotp_rom4[15:8]
nand_column_address_bytes	hw_ocotp_rom4[7:4]
nand_row_address_bytes	hw_ocotp_rom4[3:0]

Table 3. Fields of the `hw_ocotp_rom4` register on the i.MX28

You can use these field names exactly like you would use the equivalent. If you try to use a field name that does not exist on the chip family that was specified with the pragma statement, the script will print an error message.

Constants

There are a number of constants that are applicable to all registers and represent boolean values.

Name	Value	Name	Value
yes	1	no	0
true	1	false	0
on	1	off	0
blown	1	unblown	0

Table 4. Common constants

These constants can be used in place of any right hand side value in an assignment statement, as shown here:

```
cust_jtag_lockout = on
use_parallel_jtag = yes
sd_mbr_boot = false
```

Certain bit fields also have special constants defined for their values in order to make the bit settings file easier to read. The names of all of these special constants are prefixed with the name of the bit field to which they apply.

Constant name	Value
sd_bus_width_4bit	0
sd_bus_width_1bit	1
sd_bus_width_8bit	2

Table 5. `sb_bus_width` constants for all chips

Constant name	Value
sd_power_gate_gpio_pwm3	0
sd_power_gate_gpio_pwm4	1
sd_power_gate_gpio_rotarya	2
sd_power_gate_gpio_no_gate	3

Table 6. STMP3700 and STMP3770 `sd_power_gate_gpio` constants

Constant name	Value
boot_mode_usb	00h
boot_mode_i2c_3v3	01h
boot_mode_i2c_1v8	11h
boot_mode_spi_flash_ssp1_3v3	02h
boot_mode_spi_flash_ssp1_1v8	12h
boot_mode_spi_flash_ssp2_3v3	03h
boot_mode_spi_flash_ssp2_1v8	13h
boot_mode_nand_ecc4_3v3	04h
boot_mode_nand_ecc4_1v8	14h
boot_mode_nor_16bit_3v3	05h
boot_mode_nor_16bit_1v8	15h
boot_mode_jtag_wait	06h
boot_mode_spi_eeprom_ssp2_3v3	08h
boot_mode_spi_eeprom_ssp2_1v8	18h
boot_mode_sdmmc_ssp1_3v3	09h
boot_mode_sdmmc_ssp1_1v8	19h
boot_mode_sdmmc_ssp2_3v3	0ah
boot_mode_sdmmc_ssp2_1v8	1ah
boot_mode_nand_ecc8_3v3	0ch
boot_mode_nand_ecc8_1v8	1ch

boot_mode_nor_8bit_3v3	0dh
boot_mode_nor_8bit_1v8	1dh

Table 7. STMP3700 and STMP3770 `boot_mode` constants

Constant name	Value
sd_power_gate_gpio_pwm0	0
sd_power_gate_gpio_lcd_dotclk	1
sd_power_gate_gpio_pwm3	2
sd_power_gate_gpio_no_gate	3

Table 8. STMP3780/i.MX23 `sd_power_gate_gpio` constants

Constant name	Value
use_alt_gpmi_rdy3_gpmi_rdy3	0
use_alt_gpmi_rdy3_pwm2	1
use_alt_gpmi_rdy3_lcd_dotclk	2

Table 9. STMP3780/i.MX23 `use_alt_gpmi_rdy3` constants

Constant name	Value
use_alt_gpmi_ce3_gpmi_d15	0
use_alt_gpmi_ce3_lcd_reset	1
use_alt_gpmi_ce3_ssp_detect	2
use_alt_gpmi_ce3_rotaryb	3

Table 10. STMP3780/i.MX23 `use_alt_gpmi_ce3` constants

Constant name	Value
boot_mode_usb	00h
boot_mode_i2c	01h
boot_mode_spi_flash_ssp1	02h
boot_mode_spi_flash_ssp2	03h
boot_mode_nand	04h
boot_mode_jtag_wait	06h
boot_mode_spi_eeprom_ssp2	08h
boot_mode_sdmmc_ssp1	09h
boot_mode_sdmmc_ssp2	0ah

Table 11. STMP3780/i.MX23 `boot_mode` constants

Constant name	Value
sd_power_gate_gpio_pwm3	0
sd_power_gate_gpio_pwm4	1
sd_power_gate_gpio_lcd_dotclk	2
sd_power_gate_gpio_no_gate	3

Table 12. i.MX28 `sd_power_gate_gpio` constants

Constant name	Value
sd_mmc_mode_mbr	0
sd_mmc_mode_bcb	1
sd_mmc_mode_emmc_fast_boot	2
sd_mmc_mode_esd_fast_boot	3

Table 13. i.MX28 `sd_mmc_mode` constants

Constant name	Value
hab_config_hab_fab	0
hab_config_hab_open	1
hab_config_hab_close	2

Table 14. i.MX28 `hab_config` constants

Constant name	Value
boot_mode_usb	00h
boot_mode_i2c_3v3	01h
boot_mode_i2c_1v8	11h
boot_mode_spi_flash_ssp2_3v3	02h
boot_mode_spi_flash_ssp2_1v8	12h
boot_mode_spi_flash_ssp3_3v3	03h
boot_mode_spi_flash_ssp3_1v8	13h
boot_mode_nand_3v3	04h
boot_mode_nand_1v8	14h
boot_mode_jtag_wait	06h
boot_mode_spi_eeprom_ssp3_3v3	08h
boot_mode_spi_eeprom_ssp3_1v8	18h
boot_mode_sdmmc_ssp0_3v3	09h
boot_mode_sdmmc_ssp0_1v8	19h
boot_mode_sdmmc_ssp1_3v3	0ah
boot_mode_sdmmc_ssp1_1v8	1ah

Table 15. i.MX28 `boot_mode` constants