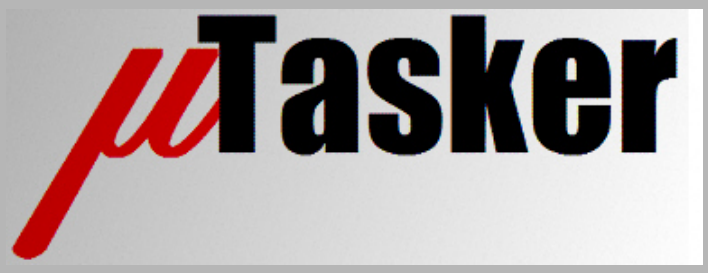
Embedding it better...



µTasker Document

µTasker – Secure SREC Bootloader

Table of Contents

1. Introduction

This document gives an example of a project using µTasker "Bare-Minimum" boot loader as primary loader for secure, encrypted uploading with SPI FLASH as intermediate storage space, plus SREC secondary loader. The SREC loader can be used as stand-alone boot loader for local UART based, alternative loading, whereby the SREC content is also encrypted.

The examples are based on a Luminary Micro (LM3Sxxxx) project but are equally relevant for any other target with the same hardware resources.

2. Bare-Minimum Boot Loader

The Bare-Minimum boot loader is the primary boot loader. It occupies a minimum of space at the start of the processor's FLASH and is responsible for handling the reset of the processor. It checks whether there is new software to be loaded from external SPI FLASH and performs the steps required to decrypt this and commit it to internal application space when necessary. If there is no new code to be copied it starts the application. In this case the application is the secondary boot loader, positioned after the Bare-Minimum loader in FLASH.

For details of the Bare-Minimum boot loader see the document
http://www.utasker.com/docs/uTasker/uTaskerBoot_003.PDF

The difference to the standard case is that the application space is defined to be after the secondary loader space. The secondary loader is however started by the bare-minimum loader instead of the application in application space.

The bare-minimum and secondary boot loaders are loaded to the board in a single step, either with or without an application. *The secondary loader cannot be updated by the bare-minimum boot loader.*

3. SREC Loader

The secondary loader is an SREC loader, which uses the serial interface to load an application in the SREC format. The application program is an SREC representation of a binary file in the format required by the bare-minimum boot loader, which is encrypted in this example.

When a valid application is loaded, the SREC loader will start it unless this is overridden by an input being held in a defined state on start-up.

The SREC loader is configured to load the application to the SPI FLASH rather than directly to its application space. On completion of the loading process the board is reset so that the bare-minimum boot loader can complete the decryption and programming process.

For details of the SREC serial loader see the document
http://www.utasker.com/docs/uTasker/uTaskerSerialLoader.PDF

The difference to the standard case is that the SREC loader deletes SPI FLASH space and loads the application to there instead of working with the final application space.

4. Application

The application software is linked to a location after the primary and secondary boot loaders. It can be loaded either together with an image containing both boot loaders (usually via JTAG) or else it can be loaded by an existing application as a binary file (encrypted in a format compatible with the bare-minimum boot loader), usually via Ethernet or USB.

Alternatively, the binary file can be converted to an SREC representation and loaded via UART using a standard terminal emulator program.

5. Configuring the Bare-Minimum Boot Loader

For this example the LM3S6965 is used and so this device is first selected in the `config.h` file in the bare-minimum boot loader project (µTaskerBoot).

Since it is to use external SPI FLASH as intermediate storage space this is enabled by activating `SPI_SW_UPLOAD` in `config.h`.

An AT45DB081D device (1MByte) is to be used and so this is configured with `SPI_FLASH_AT45DB081`. Note that this has either 256 or 264 byte pages (delivered in 264 byte mode but one-time configurable to 256 bytes). The setting `SPI_FLASH_PAGE_LENGTH` is set to match the value to be used and the driver also configures the device to 256 byte mode if this is set. After this configuration, which only happens once, it is however necessary to power cycle the board so that it is actually used for the first time.

Since the SPI Flash is a D-type the define `SPI_FLASH_AT45XXXXD_TYPE` is also set, which is however only used by the simulator for added accuracy.

In `uTaskerBoot.c` the encryption is activated in the LM3Sxxxx setup using _ENCRYPTED. The encryption settings are defined for the project according to the Bare-minimum Boot-loader guide and the application area configured as follows:

```
#define UPLOAD_FILE_LOCATION   (unsigned char *)(SIZE_OF_FLASH + UPLOAD_OFFSET) // start of firmware in SPI FLASH

#define UTASKER_CODE_START     0xc00 // external SPI FLASH solution requires two FLASH blocks for the boot code because it needs both FLASH and SPI drivers

#define UTASKER_APP_START      0x3000

#define UTASK_APP_LENGTH       (MAX_FILE_LENGTH)(119 * 1024)// 119k
```

The location of the uploaded code with reference to the start of the SPI FLASH is defined in `config.h` by:

```
#define UPLOAD_OFFSET    (1952 * SPI_FLASH_PAGE_LENGTH) // firmware offset from start of SPI FLASH
```

Finally the SPI interface and chip select used by the SPI Flash is defined in the same header file using:

```
#define CS0_LINE  PORTA_BIT3 // CS0 line used when SPI FLASH is enabled
#define SPI_CS0_PORT   GPIODATA_A
```

```
#define CONFIGURE_CS_LINES() _CONFIG_DRIVE_PORT_OUTPUT_VALUE(A,
CS0_LINE, CS0_LINE);
```

The boot loader project can now be built in the µTasker simulator environment (for subsequent complete project testing) and using a cross-compiler for the HW target. In this example a stand-along GCC build is called as a post-build step to building in the simulator environment. The GCC stand-alone build generates a binary file called `uTasker_boot.bin` which is situated in the directory `\Applications\uTaskerBoot\GNU_LM3SXXXX\`

This binary file will later be linked together with other files to generate the complete downloadable. In this specific case the size of the binary file is 2264 bytes (it is larger than a simple boot loader configuration due to the fact that it contains internal and SPI FLASH drivers plus decryption. This will occupy the first 3 FLASH sectors (3k) in the chip and so also defines the starting location for the serial loader to be at 0xc00 (3k).

6. Configuring the SREC-Loader

The serial loader project is configured to match the boot loader by setting the same processor in `config.h`. As in the boot loader project the define `SPI_SW_UPLOAD` is also set so that the SPI FLASH is used for intermediate storage. To ensure that encrypted content is correctly handled the define `SPI_SW_UPLOAD_ENCRYPTED` is also set.

The serial loader is linked to start at `0x00000c00` rather than `0x00000000` so that it works together with the boot loader.

In `Loader.h` the serial interface and its speed is set as required (usually 115'200 Baud). Details for the application software are configured in the same header file with:

```
#define UTASKER_APP_START (12*1024) // application starts at this
address
#define UTASKER_APP_END (unsigned char *)(UTASKER_APP_START +
(116*1024))// end of application space - after maximum application
size
#define UPLOAD_OFFSET  (1952 * SPI_FLASH_PAGE_LENGTH)     // offset
to image area in SPI Flash (when used)
```

The serial SREC loader is now build in the µTasker environment and for the target using a cross-compiler. In the case of the GCC stand-alone cross-compiler build the binary output is called `uTaskerSerialBoot.bin` and is situated in the directory `\Applications\uTaskerSerialBoot\GNU_LM3SXXXX\`

Since the serial loader saves encrypted binary to the intermediate area with a compatible µFileSystem header it is important that the header matches the application/boot loader settings. For example usually these are valid:

```
typedef unsigned long   MAX_FILE_LENGTH;
#define SUPPORT_MIME_IDENTIFIER
```

This gives a file length saved as 4 bytes plus a file type saved as a fifth byte

This binary file will later be linked together with other files to generate the complete downloadable. In this specific case the size of the binary file is 8230 bytes. This will occupy the 9 FLASH sectors following the boot loader (9k) in the chip and so also defines the starting location for the application software to be at 0x3000 (12k).

7. Memory Layout and Application Details

The size of the boot loaders essentially dictates the remaining space for the application. This is shown in the following table:

0x00000000 .. 0x00000bff	Bare-Minimum boot loader	Internal FLASH - 3k
0x00000c00 .. 0x00002fff	SREC loader	Internal FLASH – 9k
0x00003000 .. 0x0003ffff	Application space (possibly including internal file system and parameters)	Internal FLASH – 244k
0x00040000 .. 0x0013ffff	External File System space	External SPI FLASH – 1MByte*

The application space may however not be utilised purely for application code but may also be used for saving parameters or contain an internal file system. As seen from the memory layout, the application has a start address of 0x3000 so that it occupies the FLASH area immediately following the serial loader. The application code is linked to start at the corresponding address but is otherwise unchanged from application code that runs autonomously (without any boot loaders installed).

An example of the use of the application FLASH space could be as follows, whereby a code area of 119k is reserved, followed by 2k of parameters and then 126k internal file system. Note that the application code area is not necessarily filled with code but it represents the maximum size that the application can grow to during development and still fit within its area. When uploads are performed the application space is fully deleted – areas reserved for other uses are not deleted. In this case the maximum code size corresponds to the values used to configure both the boot loader and the serial loader:

0x00003000 .. 0x0001ffff	Application code space	Internal FLASH – 116k
0x00020000 .. 0x000207ff	Internal Parameter System space	Internal FLASH – 2k
0x00020800 .. 0x0003ffff	Internal File System space	Internal FLASH – 126k

Also the use of the external SPI FLASH by the application is could be divided into different areas, whereby the upload image area corresponds to the settings above.

0x00040000 .. 0x0013ffff	File system area	External FLASH – 488k*
0x000ba000 .. 0x000d6fff	Upload image area	External FLASH – 116k*
0x000d7000 .. 0x0013ffff	Data logging area	Internal FLASH – 420k*

Note that the SPI FLASH addresses are virtual addresses since SPI FLASH is serial in nature and so is not truly memory mapped in the system. The SPI FLASH drivers allow the user to address locations as if they were a part of real address space. Normally the SPI FLASH virtual address space is located immediately after the end of internal FLASH address space, which also allows a file system (or other use) to extend from within internal SPI FLASH to external SPI FLASH without the application needing to be aware of the details.

The application binary image can be combined with the binary images of the boot loader and serial loader to give a single binary image for downloading to the target using standard programming techniques (JTAG, etc.). The application image can also be loaded in encrypted SREC form if only the boot loader and serial loader are available.

**For simplicity the sizes are accurate for the FLASH in 256 byte page mode, otherwise the area is correspondingly larger (264/256).*

8. Encrypted SREC Loading Procedure

Base on the example configuration, starting with all three elements in place (boot loader, serial loader and application) the procedure for uploading a new application can be presented:

1) The boot loader checks for a new image but doesn't find one so starts the serial loader. The serial loader detects a valid application and starts it. The application runs.

0x00000000 .. 0x00000bff	Bare-Minimum boot loader	Starts serial loader
0x00000c00 .. 0x00002fff	SREC loader	Starts application
0x00003000 .. 0x0003ffff	Application space	**Application running**
0x00040000 .. 0x0013ffff	External File System space	No image available

2) The boot loader checks for a new image but doesn't find one so starts the serial loader. The serial loader either detects an input commanding it to enter serial loading mode or doesn't detect a valid application. The serial loader runs.

0x00000000 .. 0x00000bff	Bare-Minimum boot loader	Started serial loader
0x00000c00 .. 0x00002fff	SREC loader	**Serial loader running**
0x00003000 .. 0x0003ffff	Application space	Application not started
0x00040000 .. 0x0013ffff	External File System space	No image available

3) The image area is deleted (if not blank) and an encrypted SREC file is loaded to the image area.

0x00000000 .. 0x00000bff	Bare-Minimum boot loader	Started serial loader
0x00000c00 .. 0x00002fff	SREC loader	**Serial loader running (deletes image area and loads new encrypted application)**
0x00003000 .. 0x0003ffff	Application space	Application not started
0x00040000 .. 0x0013ffff	External File System space	**Encrypted application stored**

4) A reset is performed and the boot loader detects the new, valid encrypted image. It first deletes the application area.

0x00000000 .. 0x00000bff	Bare-Minimum boot loader	**Boot loader running**
0x00000c00 .. 0x00002fff	SREC loader	Serial loader not running
0x00003000 .. 0x0003ffff	Application space	**Application Area deleted**
0x00040000 .. 0x0013ffff	External File System space	**Encrypted application stored**

5) The boot loader now decrypts the image in external SPI FLASH and copies its decrypted content to the application space.

0x00000000 .. 0x00000bff	Bare-Minimum boot loader	**Boot loader running**
0x00000c00 .. 0x00002fff	SREC loader	Serial loader not running
0x00003000 .. 0x0003ffff	Application space	**New decrypted code in the application space**
0x00040000 .. 0x0013ffff	External File System space	**Encrypted application stored**

6) The boot loader checks that the decrypted code is valid (using check sums) and deletes the intermediate image storage space

0x00000000 .. 0x00000bff	Bare-Minimum boot loader	**Boot loader running**
0x00000c00 .. 0x00002fff	SREC loader	Serial loader not running
0x00003000 .. 0x0003ffff	Application space	**New decrypted code in the application space**
0x00040000 .. 0x0013ffff	External File System space	No image available

7) The boot loader resets the board and returns to step 1 since it no longer finds an image waiting to be updated, thus starting the serial loader. The serial loader identifies a valid application and starts this.

It is to be noted that the boot loader can recover from a reset or power cycle at any stage of the process. Since the encrypted code contains secret keys which need to match before it is recognized as valid it is not possible that invalid code is committed that will then leave the device in an un-operational state. Furthermore, since the application is started only after the serial loader has allowed it to start, the serial loaded can also be forced into active mode based on an input state or other logic.

9. Combining Binary Files and Generating an Encrypted SREC

The previous sections have detailed generating boot loader and serial loader binaries. In addition it has been noted that the only difference between standard application software and software that can operate with the loaders is its start address. Therefore, assuming compiling with GCC and using the µTasker demonstration application linked to the correct start address and called `uTaskerBM.bin` the files can be converted for loading to the target.

Input files:

`\Applications\uTaskerBoot\GNU_LM3SXXXX\uTasker_boot.bin`

`\Applications\uTaskerSerialBoot\GNU_LM3SXXXX\uTaskerSerialBootBM.bin`

`\Applications\uTaskerV1.4\GNU_LM3SXXXX\ uTaskerV1.4_BM.bin`

Utilities can be used to combine the binary files. To combine the boot loader with the serial loader (respecting their locations) the following command can be used (assuming called from the serial loader directory):

```
uTaskerCombine "..\..\uTaskerBoot\GNU_LM3SXXXX\uTasker_boot.bin"
uTaskerSerialBootBM.bin 0xc00 uTaskerSB.bin
```

This generates the file `uTaskerSB.bin` containing the boot loader, followed by the serial loader (starting at 0x0c00 binary offset).

To add the application the following command can then be executed (assuming called from the application directory):

```
uTaskerCombine "..\..\uTaskerSerialBoot\GNU_LM3SXXXX\uTaskerSB.bin "
uTaskerV1.4_BM 0x3000 uTaskerCombined.bin
```

This file can then be loaded using JTAG or other programming tools and includes all three software elements. The application will be immediately invoked.

To generate an encrypted version of the application to be downloaded via the serial loader the following command can be used.

```
uTaskerConvert.exe uTaskerV1.4_BM encrypted.bin -0x1235 -
a748b6531124 -ff25a788f2e681338777 -afe1 -226a
```

The use of the encryption values is descried in the boot loader documentation and the values will be specific for a project and user and generally kept secret. The output from the conversion step is a file called `encrypted.bin`. This is not yet in SREC format and so needs to be further converted as follows:

```
arm-none-eabi-objcopy --input-target=binary encrypted.bin --output-
target=srec encrypted.srec
```

This uses the GCC `objcopy` utility to generate an SREC from the binary file. The SREC contains address information starting at `0x00000000` but the serial loader will nevertheless save it to the defined location in external SPI FLASH so that the subsequence procedure (see previous section) operates correctly after the target has been reset.

Generally the following three output files are the most interesting:

- `uTaskerCombined.bin` This can be loaded to a board and contains all three programs. The application will run.

- encrypted.bin This is the encrypted application suitable for being uploaded by some applications to the intermediate storage space (eg. via Ethernet, which is not described further in this document but is discussed in the boot loader documentation).
- encrypted.srec This can be loaded via the serial interface when the serial loader is active.

10. Checking the SREC Download

When the serial loader starts (when it doesn't start a valid application) it displays a start menu including information about its configuration:

```
uTasker Serial Loader
=====================
[0x00003000/0x0001ffff]
[0x000ba000/0x000d6fff]
bc = blank check
du = delete upload area
ld = start load
go = start application
>
```

The first address range corresponds to the application code area and the second the intermediate storage area. This corresponds to the configuration as presented in this document. The information is only displayed when SHOW_APP_DETAILS is active in Loader.h. The commands are also valid for the configuration with SPI_SW_UPLOAD set in config.h, whereby the commands are named slightly differently to the case where only internal memory is used.

- bc verifies that the upload area (area in the second address area in external SPI FLASH) has been deleted ready to receive new intermediate code
- du deletes the upload area and should be used before an SREC load if not blank
- ld is used to start the SREC loading
- go causes the serial boot loader to exit by starting the application. It doesn't check whether the application is valid and so the command forces it to run either a good or bad application

The SREC download procedure is described in full details in the serial loader documentation. The file that would be used in the discussion case is encrypted.srec.

11. Conclusion

This document has described the configuration and operation of a project combining the capabilities of the µTasker boot loader and serial loader to implement a serial SREC loader with encrypted SREC content. Use is made of an external SPI FLASH device as intermediate storage to maximise the possible application size and ensure reliable operation even in cases of power cycles during the update process.

The reference configuration has been discussed in detail and simple configuration modifications can be performed to achieve the same results in a wide range of memory layouts and for all processors supported by the µTasker project.

Although the document has described the SREC method of loading a new application via serial interface, the installed application can also load the new software to the intermediate location and reset to achieve the same result. This is typically performed by Ethernet (FTP, HTTP POST), UART, USB, etc. This may be considered a primary update method (when a valid application is installed and can achieve this) whereas the SREC loader can then be considered a secondary loader which comes in to play when either the primary loader is not available or has operation difficulties.

Modifications:

V0.00 30.10.2010: Initial version.

V0.01 31.10.2010: Corrections and more information about terminal output and loading application using alternative methods.

V0.02 05.11.2010: Add `UTASKER_CODE_START` define to boot loader configuration.