



Serial Boot Loader For CC2538 SoC

Document Number: SWRA431

Version 1.1

TABLE OF CONTENTS

1. PURPOSE	3
2. FUNCTIONAL OVERVIEW.....	3
3. ASSUMPTIONS	3
4. DEFINITIONS, ABBREVIATIONS, ACRONYMS.....	3
5. REFERENCES	4
6. REVISION HISTORY	4
7. DESIGN CONSTRAINTS AND REQUIREMENTS	4
8. DESIGN.....	5
8.1 SBL CONTEXT.....	5
8.2 FUNCTIONAL DESCRIPTION	5
8.2.1 <i>Boot Code</i>	5
8.2.2 <i>SBL transport protocol</i>	6
8.2.2.1 <i>Specific command structure</i>	7
8.2.2.2 <i>Specific response structure</i>	8
8.2.3 <i>SBL-compatible Z-Stack</i>	10
9. PRODUCING SBL BOOT CODE TO BE PROGRAMMED.	10
9.1 SEPARATE BUILD & DEBUG OF BOOT CODE.....	10
10. PRODUCING SBL-COMPATIBLE APPLICATION CODE	11
10.1 CONFIGURE LINKER OPTIONS FOR THE SBL FUNCTIONALITY	11
10.1.1 <i>checksum symbol</i>	11
10.1.2 <i>checksum calculation</i>	12
10.1.3 <i>Extra symbols</i>	13
10.1.4 <i>Select output file format</i>	14
10.1.5 <i>Change the linker configuration file</i>	15
10.2 BUILDING THE APPLICATION CODE FOR SBL.	16
10.3 DEBUGGING THE APPLICATION CODE WITH SBL.	16
10.3.1 <i>Preserve the SBL.</i>	16
10.4 FORCING BOOT-MODE OR EARLY JUMP TO APPLICATION CODE.....	16
11. DOWNLOADING THE SBL IMAGE USING THE SBDemo PC TOOL	16
12. COMBINING THE BOOT LOADER + APPLICATION IMAGE VIA HEX FILES	17
13. COMBINING THE BOOTLOADER + APPLICATION IMAGE VIA BIN FILES.....	18
14. APPENDIX: MEMORY MAP FOR SBL ON CC2538	21

1. Purpose

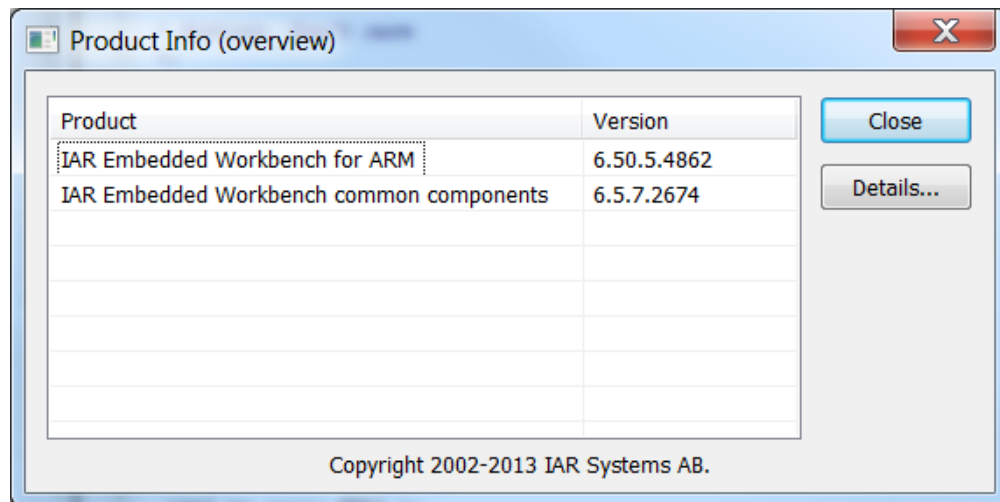
This document provides a developer's guide to implement serial boot loader (SBL) compatibility in any sample or proprietary Z-Stack Application for the CC2538 SoC.

2. Functional Overview

SBL is provided as a value-enhancing sample solution that enables the updating of code in devices without the cost of maintaining any download-related code in the user application other than ensuring a compatible flash memory mapping of the final output. SBL is implemented as a managed client-server mechanism which requires a serial master to drive the process (i.e. a PC hosted GUI application with access to the serial connection to the CC2538.)

3. Assumptions

1. SBL is a generic feature that should deviate as little as possible from one implementation to another by only supporting the idiosyncrasies of the specific medium (e.g. UART, SPI) crucial to the level of service necessary to complete a code image download in a reasonable amount of time. For the sake of example only, the UART SBL specific references will be made in this document, although merely changing the medium-specific verbiage and paths should allow this document to sufficiently describe any Z-Stack SBL. (When using CC2538 EM with SmartRF06 EB, the UART is actually emulated over USB by an external USB-UART converter on the SmartRF06 platform).
2. All examples below were tested with the following version of IAR environment:



4. Definitions, Abbreviations, Acronyms

Term	Definition
PC	Personal Computer
NV	Non-volatile (e.g. memory that persists through power cycles.)
SBL	Serial Boot Load(er)

5. References

[1] Z-Stack Developer's Guide (SWRA176)

6. Revision History

Date	Document Version	Description of changes
06/12/13	1.0	Initial Release with Z-stack Home 1.1.0
11/20/13	1.1	Added third page of flash for boot loader. Re-worked process to combine bootloader and application to use .hex files

7. Design Constraints and Requirements

1. A serial bus master must drive the download across the serial bus to the CC2538 - the means by which to design or implement such an application is beyond the scope of this document.
2. The Boot Code occupies the three flash pages (addresses 0x0027B000 – 0x0027C7FF) immediately preceding the NV area. These flash pages cannot be used by the user application.
3. The image to be loaded via SBL (or be pre-programmed alongside the SBL) must conform to a flash memory mapping compatible with the serial boot loader, and include a specific control structure as follows:
 - a. The image must be built to start at address 0x00200000 (FLASH start address). This is a restriction of the Serial Boot Tool, since the input image file that it expects is stripped of all address information, so the image is assumed to start at the flash start. If the user implements a bootloading server of his own, than this restriction does not apply.
 - b. The image size cannot exceed 505856 bytes (total flash size, minus the size reserved for the bootloader, NV Memory and the lock bits page).
 - c. The SBL Header structure must be located at address 0x0020011C. This address is hardcoded in the bootloader firmware. It was chosen so there is exactly enough space before it to have the vector table at the start of the flash. This structure is formatted as follows:

Address offset from 0x0020011C	Element type and name	Value / Details
0	uint32 checksum	Lower 16 bits: ielftool-generated CRC-16, polynomial 0x18005, (MSB first), calculated over the whole image except the two bytes at address 0x0020011C (which stores the result checksum) Higher 16 bits: 0x0000
4	uint32 compatibility_flags	0xFFFFFFFF (for future use)

8	uint32 img_status	0xA5A5A5A5 Indicates to the bootloader that there is an image in flash, which have not been verified yet. After the image is programmed into flash, and the bootloader code verified it, it will update the value of this field to 0x05A0A005 to signal that the image passed integrity check, so the next time the device is power-cycled, there is no need to recalculate the checksum.
12	uint32 checksum_begin	The address where the checksum calculation starts. Typically, this is the first address of the image.
16	uint32 checksum_end	The address where the checksum calculation ends (inclusive). Typically, this is the last address of the image.
20	uint32 vector_table_address	The start address of the vector table.

8. Design

8.1 SBL Context

An SBL-enabled system is comprised of two images: the 'boot loader code' and the Z-Stack with its Application(s) built compatibly – the 'Application Code' / 'Active Code'. The placement of each of the two images into the internal flash is handled by the unique IAR linker command file used by each.

8.2 Functional Description

8.2.1 Boot Code

The SBL solution requires the use of boot code to handle receiving a new application code and write it to flash memory, as well as check the integrity of the active code image before jumping to it. This check guards against an incomplete or incorrect programming of the active image area. The SBL boot code provides the following functionality:

1. At power-up, the vector table pointer points to the vector table of the boot code (this is set by the internal ROM bootloader, which is out of scope for this document).
2. The boot code waits for the SBL master (SBL server) to communicate with it. If there is no communication for about 30 seconds after power-up, and there is a valid application image loaded into flash, the boot code will change the vector table pointer to point to the vector table of the active application, and will then execute this application. If there is no valid image loaded, the bootloader continues to wait for a master communication.

3. The SBL master can send to the bootloader any of the following commands:

a. SB_HANDSHAKE_CMD

Must be sent as the first command. Used to tell the boot code to prepare to download an image, reset internal counters, etc.

b. SB_WRITE_CMD

Send a block of data to be programmed at a given flash address.

c. SB_READ_CMD

Request to read a block of data from a given flash address.

This command is optional. The SBL Master can use this command in order to verify that the downloaded image is stored correctly in flash.

d. SB_ENABLE_CMD

Asks the boot code to mark the image as 'verified', and execute it.

This command is optional. If it is not sent, then the image will be verified at next power-up (after the initial 30 seconds wait): The boot code will calculate the image's checksum, and if it matches the checksum stored in the SBL Header structure, it will mark the image as valid and execute it.

At the following power-up, when the image is already marked as 'valid', no CRC calculation will be done, but instead, after the initial 30 second wait, the boot code will immediately execute the image.

4. Note that unless the image is marked as 'valid' by the SBL Master (using the SB_ENABLE_CMD), the boot code will guard against interrupted, incomplete or incorrect programming of the active image area by checking the validity of the active application code image via CRC. If the image is not valid then the boot code will not allow it to run.

8.2.2 SBL transport protocol

The packets sent over the serial transport are divided into two types:

Commands – from the SBL Master to the Bootloader on CC2538

Responses – from the Bootloader on CC2538 to the SBL Master

Both commands and responses have the same format:

The following fields are sent in the given order (top to bottom in the table), LSB first.

Field name	Size [Bytes]	Details
Start Of Frame (SOF)	1	0xFE (constant value)
Length8	1	0x00 – 0xFE : Payload length if smaller than 255

		0xFF: Payload length is specified in an additional field, Length32, sent later. It is done this way to preserve compatibility with older protocol, where the size was restricted to 255.
Frame ID	1	0x4D (constant value. Specifies a bootloader packet).
Command	1	0x01 - SB_WRITE_CMD 0x02 - SB_READ_CMD 0x03 - SB_ENABLE_CMD 0x04 - SB_HANDSHAKE_CMD The above values are OR'd with 0x80 when the packet is a response.
Length32	4	Payload length. This field exists if and only if Length8 == 0xFF.
Payload	Payload length	Payload specific to the current command
Frame Check Sequence	1	Xor of all message bytes following the SOF byte (not including the SOF byte)

8.2.2.1 Specific command structure

The payload of each packet is according to the specific command being sent, and is described in the following paragraphs.

8.2.2.1.1 **SB_HANDSHAKE_CMD**

No payload required.

8.2.2.1.2 **SB_WRITE_CMD**

Send a block of data to be written to the flash memory at a given address.

For reducing the time it takes to download an image, when the data block ends with a series of all-one bytes (0xFF), it is possible to include in the write command only the data bytes up to the last non-0xFF byte in the block. The remaining 0xFF bytes will be generated internally by the bootloader on the CC2538 SOC. With this optimization, erasing a section of memory of any size, can be done easily by sending a write command with 0 data. The parameters “First Address” and “Operation Size” (see below) defines the actual memory section that needs to be erased / written.

Payload field name	Size [Bytes]	Details
First Address	4	The first address to write the given data block to.

		The address must be 32bit-aligned.
Operation Length	4	The total number of bytes that this write operation covers. Note that this number may be larger than the actual number of data bytes sent in this command (see explanation above) . The length must be 32bit-aligned.
Data Bytes	(Payload length) – 8	Payload length is as defined in the generic structure table above.

8.2.2.1.1 **SB_READ_CMD**

Read a block of data from the CC2538 SOC flash, specifying the start address and the number of bytes to read.

For reducing the time it takes to read an image, when the data block ends with a series of all-one bytes (0xFF), the bootloader will actually send the data only up to (and including) the last non-0xFF byte in the block. The remaining bytes can be internally generated by the SBL Master according to the Operation Length field.

Payload field name	Size [Bytes]	Details
First Address	4	The first address to read from.
Operation Length	4	The number of bytes to read.

8.2.2.1.1 **SB_ENABLE_CMD**

No payload required.

8.2.2.2 Specific response structure

The payload of each packet is according to the specific response being sent and is described in the following paragraphs.

8.2.2.2.1 **SB_HANDSHAKE_CMD**

Payload field name	Size [Bytes]	Details
Response Status	1	0x00 - SUCCESS
Bootloader Revision	4	0x01 (currently)

Device Type	1	0x01 – CC2538
Buffer Length	4	The maximum data size to use with Read / Write command
Page Size	4	0x800 – CC2538 flash page size

8.2.2.2.1 **SB_WRITE_CMD**

Payload field name	Size [Bytes]	Details
Response Status	1	0x00 – SUCCESS 0x01 – FAILURE – if the arguments are wrong.

8.2.2.2.1 **SB_READ_CMD**

See the remark at the SB_READ_CMD command above, regarding the optimization of 0xFF trailing bytes.

Payload field name	Size [Bytes]	Details
First Address	4	Should be the same as the respective parameter of the Read command.
Operation Length	4	Should be the same as the respective parameter of the Read command.
Data Bytes	(Payload length) – 8	Payload length is as defined in the generic structure table above.

8.2.2.2.1 **SB_ENABLE_CMD**

Payload field name	Size [Bytes]	Details
Response Status	1	0x00 – SUCCESS 0x07 – Failed to mark image as ‘valid’

8.2.3 SBL-compatible Z-Stack

An SBL-compatible Z-Stack is implemented as a standard ZigBee Application, with minimum changes in the linker file and settings, which are all covered in the next sections.

9. Producing SBL Boot Code to be programmed.

9.1 Separate Build & Debug of Boot Code

The Boot Code is separately built and debugged or programmed via the IAR IDE by opening the SBL Boot Project here:

```
$INSTALL_DIR$\Projects\zstack\Utilities\BootLoad\CC2538\Boot.eww
```

Before debugging or physically programming the SBL-compatible Application code produced in the next section, this SBL Boot code must first be programmed into the flash (but only this once, since, as the following section mentions, the default option for application code is to preserve this SBL Boot code on successive debugging or programming.)

10. Producing SBL-compatible Application Code

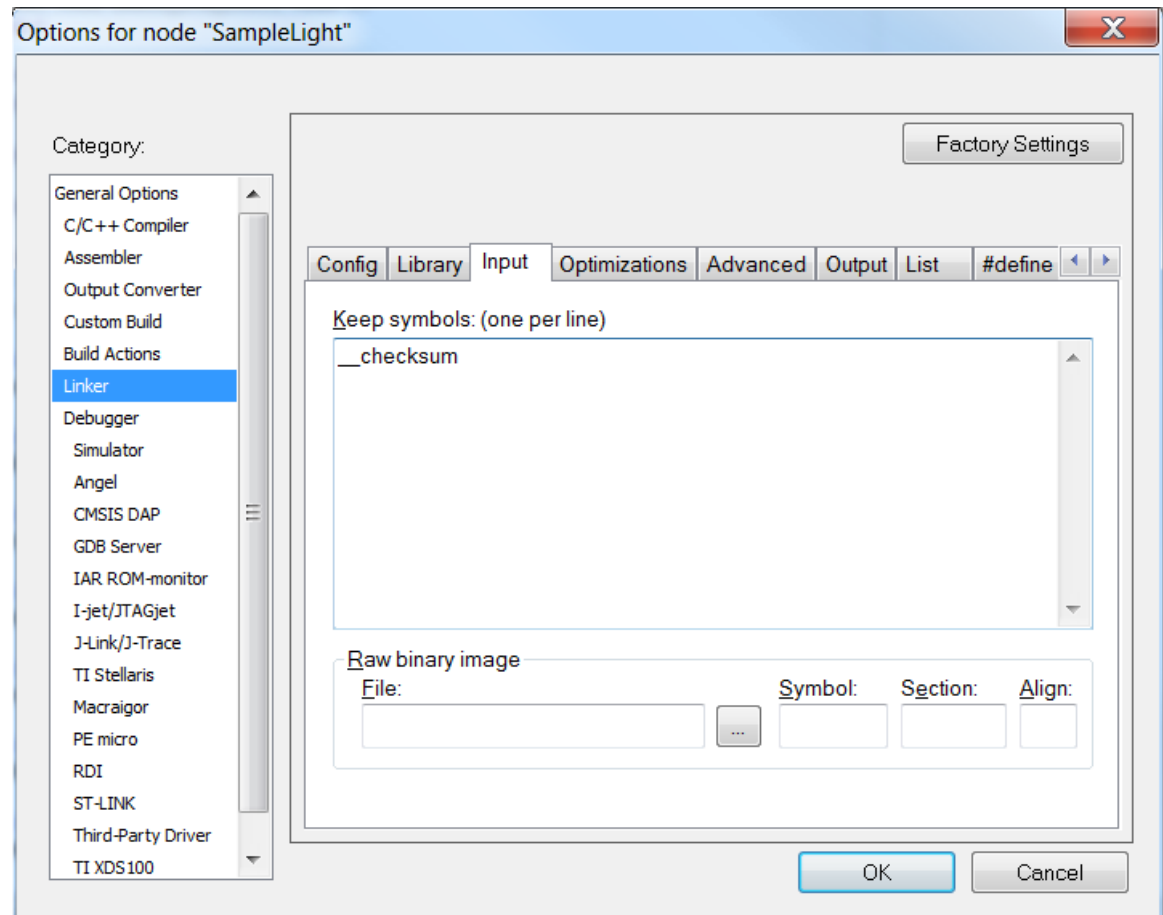
The “SampleLight - Coordinator” build of the Z-Stack sample application known as SampleLight is used below for demonstration purposes only - the customer would apply the following steps in their own, proprietary Z-Stack application and make the corresponding changes to all of the paths below that are specific to SampleLight.

10.1 Configure linker options for the SBL functionality.

10.1.1 checksum symbol

Define the symbol used by the linker to store the calculated checksum:

Linker→Input→Keep symbols: add “__checksum”

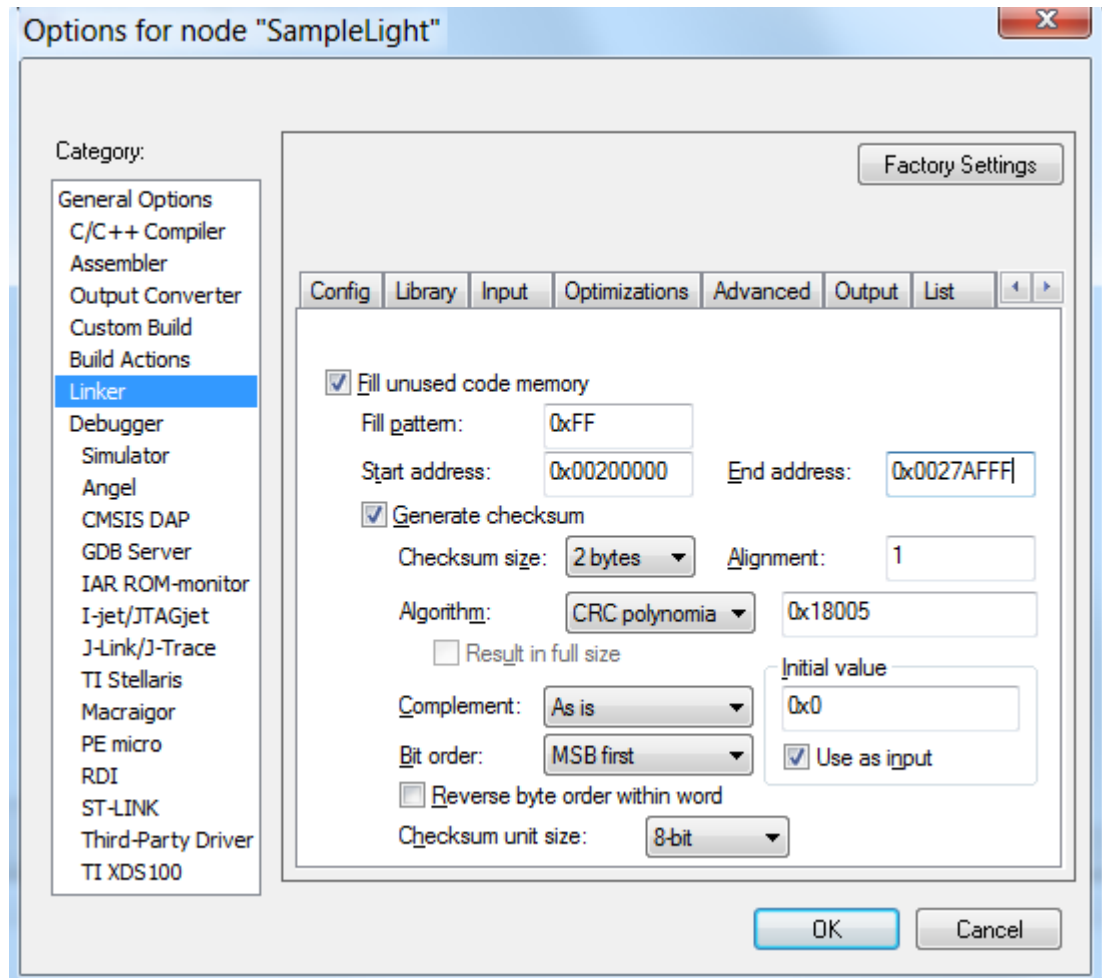


10.1.2 checksum calculation

Tell the linker the type of checksum to calculate, and the address range to include in this calculation. (Note that if the symbol `__checksum` is within the given range, it is skipped when calculating the checksum).

Linker→Checksum:

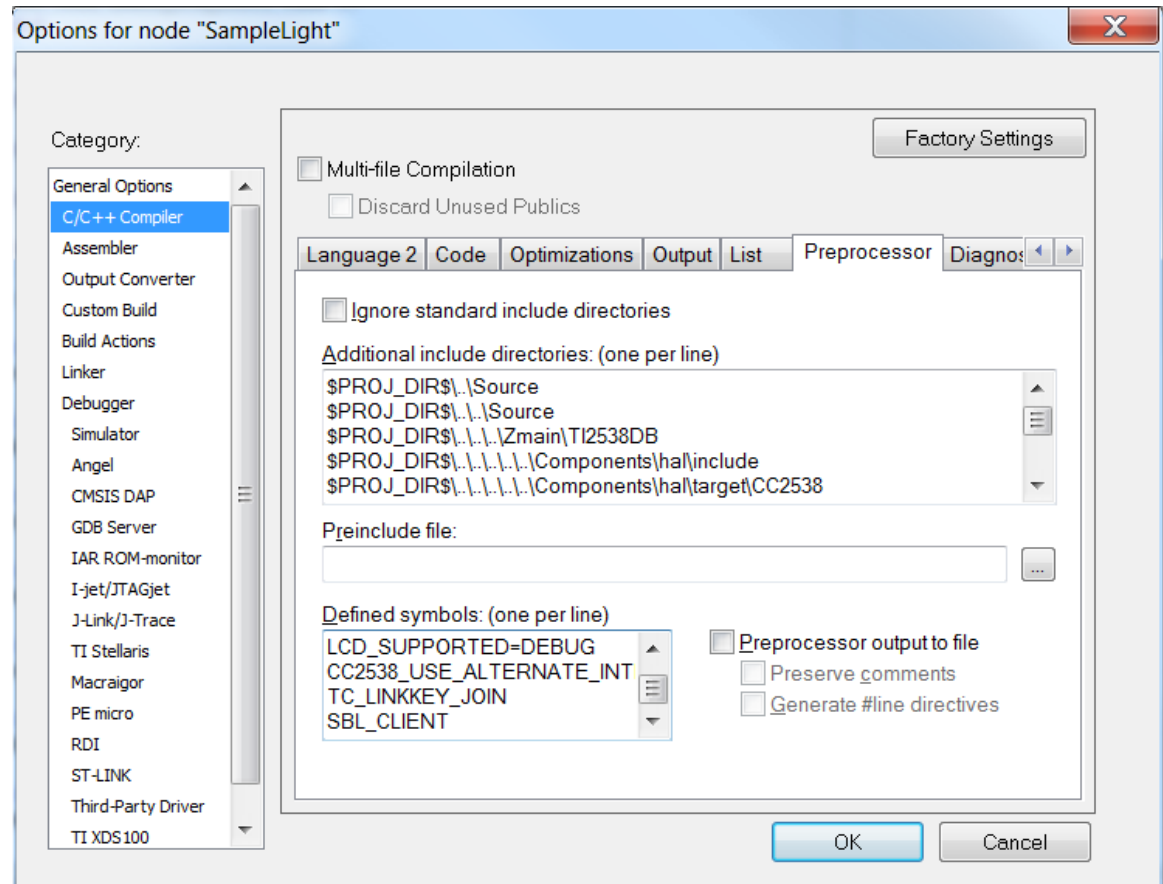
Copy the settings from the image below:



10.1.3 Extra symbols

Enable SBL specific code changes, by defining the appropriate symbol:

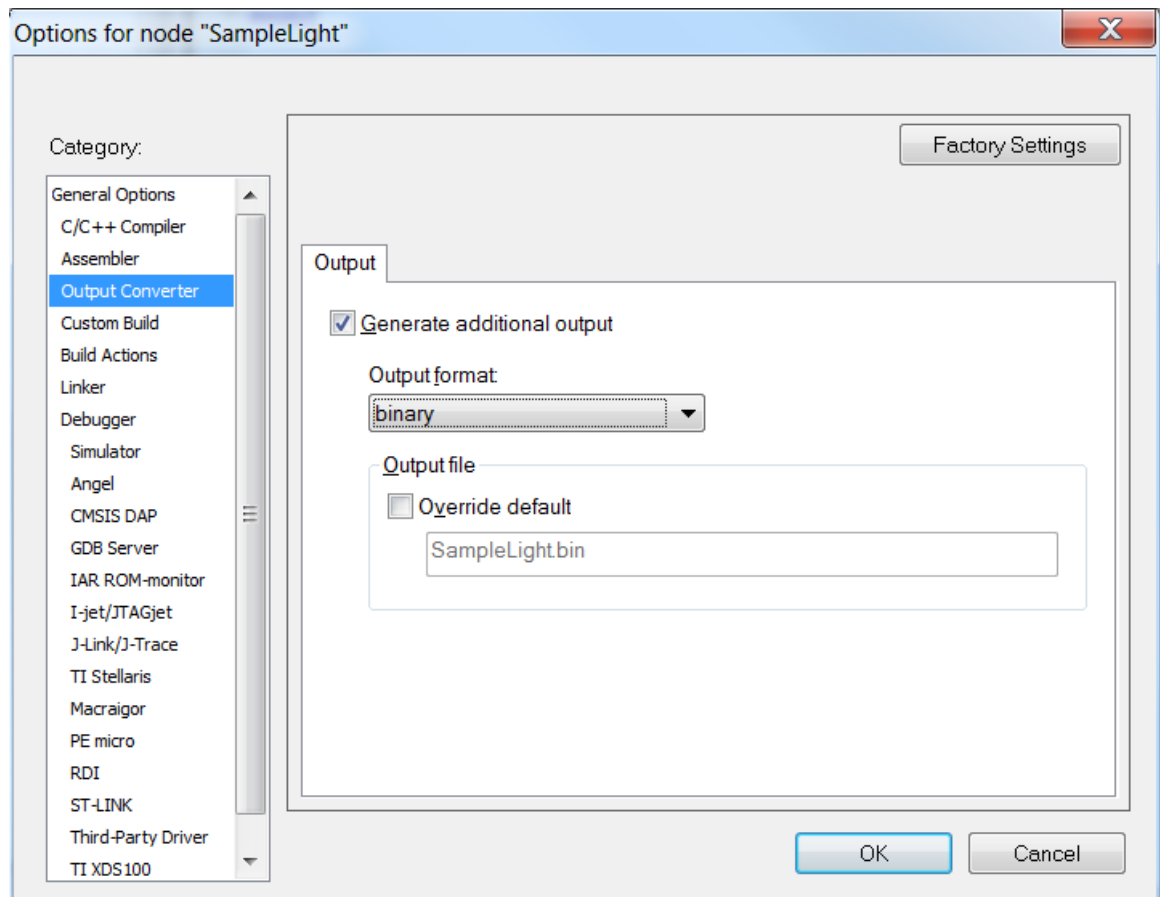
C/C++ Compiler→Preprocessor→Defined symbols: add SBL_CLIENT



10.1.4 Select output file format

For downloading an image using the SerialBootTool SBL Master tool, the linker should output the build result in a binary format.:

Output Converter→Output format: set to 'binary':

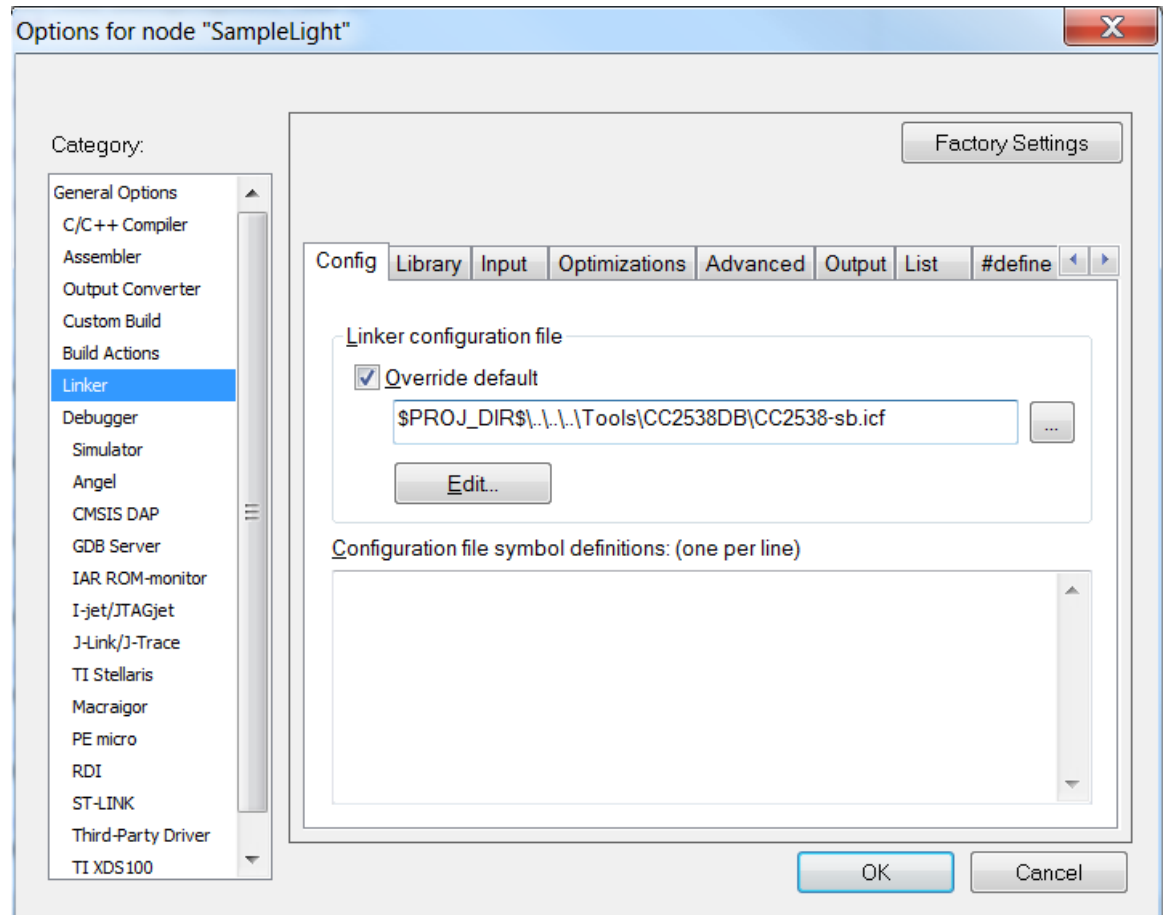


10.1.5 Change the linker configuration file

This is a slightly modified linker configuration file, which tells the linker where to store the calculated checksum, and also defines the correct address for the rest of the SBL Header struct.

Linker→Config→Linker configuration file:

Change CC2538.icf to CC2538-sb.icf



10.2 Building the Application Code for SBL.

After applying the configuration settings as specified in the previous paragraphs, simply build the project from the IAR IDE as you normally would. The binary file produced, which is to be loaded by SBL, is found here:

`$PROJ_DIR$\Coordinator\Exe\SampleLight.bin`

10.3 Debugging the Application Code with SBL.

10.3.1 Preserve the SBL.

In order to run or debug the Application Code, a Boot Code image must have already been downloaded to the CC2538 SoC (see the previous section.) Now, the application code can be programmed normally from IAR – by default, it will not overwrite the bootloader pages.

10.4 Forcing boot-mode or early jump to Application code.

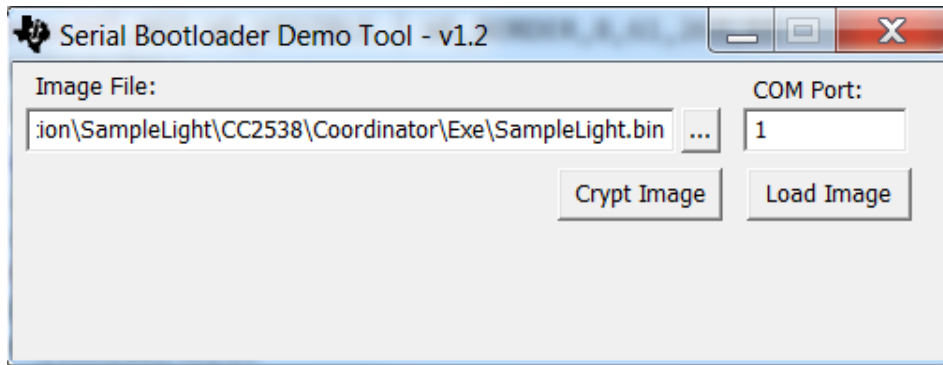
The SBL boot code receives control at power-up and verifies whether valid Application code is present. If so, then the SBL gives the bus master (the SBL master) a window in which to force bootloading mode or an immediate jump to Application code. If the bus master does not send any command within this timeframe, then the boot code will pass control to the application code. If there is no valid application code, then the boot-mode is immediately executed.

- For initiating boot-mode, the bus master shall send the SB_HANDSHAKE_CMD;
- For immediate jump to the existing application, the bus master shall send the SB_ENABLE_CMD

The timeframe during which the bus master can initiate communication is hardcoded in the boot code. It can be changed in the bootloader project, by changing the startup delay value in `bootloader_communication_requested()` in `sb_main.c`

11. Downloading the SBL Image using the SBDemo PC Tool

As mentioned above, upon reset, the SBL will give the bus master a window of opportunity to perform an image download to the device. At this time, the SerialBootTool.exe PC Tool can be used to download a .bin file by selecting the desired file and setting the COM port as appropriate (as shown in the figure below). The PC Tool can be found in the Z-Stack installation folder, under “Tools\SBL Tool\SBDemo.exe”

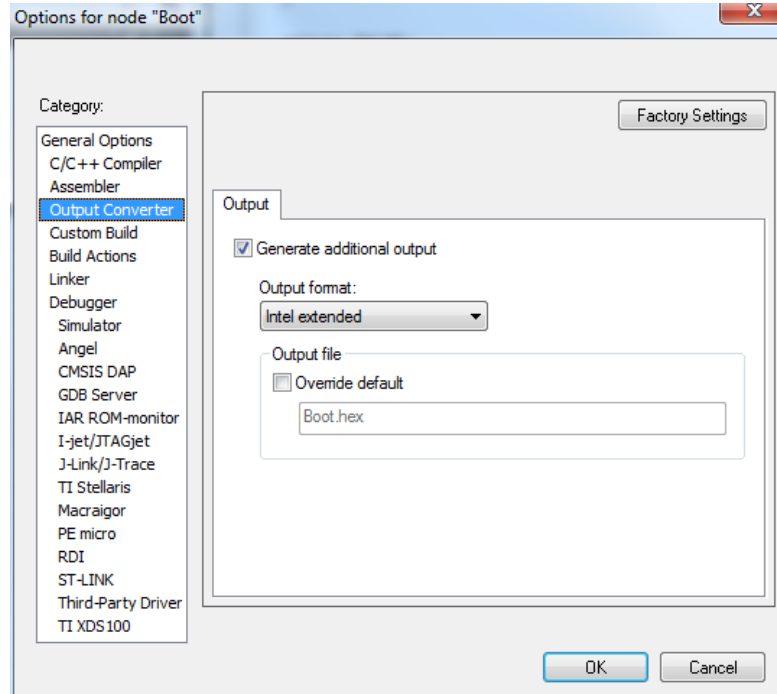


Note the “Crypt Image” button is relevant only for BLE applications and is not applicable for ZigBee and the CC2538.

12. Combining the Boot Loader + Application Image via Hex Files

For mass-production programming, it's important to have a single image containing both the Serial Boot loader and Application code so that the part must only be programmed once. The following example assumes that the SmartRF Flash Programmer 2 tool will be used for programming a hex formatted file into the CC2538 SoC.

In the steps outlined below, you'll need to rebuild the application and the boot loader so that the generated output file is in hex format. This can be accomplished by modifying the IAR project options. Specifically, you'll need to change the Output format to “Intel extended” via the Output Converter project options show in the image below:



1. Modify the Output Converter settings, as shown above, for both the boot loader and application projects and rebuild both.
2. Using a text editor, “stitch” (combine) the two .hex files created in step 1. For the CC2538, the boot loader actually resides near the end of flash (higher address) as

opposed to the start of flash (lower address) so we'll be prepending the Application image to the boot loader image.

- a. Open the boot loader .hex file in a text editor such as Notepad++.

Note: you'll find the .hex files in the "Exe" folder of your project.

e.g. Projects\zstack\Utilities\BootLoad\CC2538\Basic\Exe\boot.hex

- b. Completely remove the first line of the file. It should look something like this:

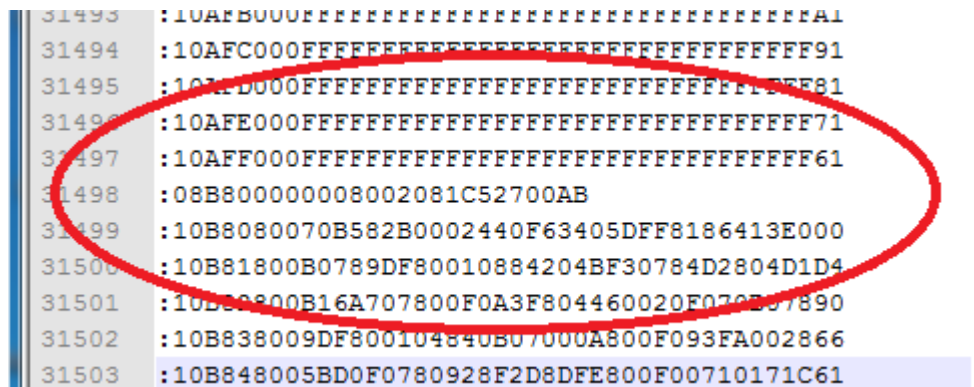
:020000040027D3

- c. Open the application .hex file separately and remove the last two trailing lines. They should look something like this:

:04000005002204F7DA

:00000001FF

- d. Now select all of the remaining lines of the application .hex file and copy and paste them into the editor with the modified boot loader .hex file, starting at the first line. You're basically prepending the application .hex file to the modified boot loader .hex file. The line where the two files now meet, application + boot loader, should look something like this:



```
31493 :10AFB000FFFFFFFFFFFFFFFFFFFFFFFFA1
31494 :10AFC000FFFFFFFFFFFFFFFFFFFFFFFF91
31495 :10AFD000FFFFFFFFFFFFFFFFFFFFFFFF81
31496 :10AFE000FFFFFFFFFFFFFFFFFFFFFFFF71
31497 :10AFF000FFFFFFFFFFFFFFFFFFFFFFFF61
31498 :08B80000008002081C52700AB
31499 :10B8080070B582B0002440F63405DFF8186413E000
31500 :10B81800B0789DF80010884204BF30784D2804D1D4
31501 :10B82800B16A707800F0A3F804460020F070B7890
31502 :10B838009DF800104840B07000A800F093FA002866
31503 :10B848005BD0F0780928F2D8DFE800F00710171C61
```

- e. Save this file as something else such as:

ApplicationWithBootloader.hex

You can then program the CC2538 device using this new combined image hex file and the SmartRF Flash Programmer 2 tool. The tool is available at <http://www.ti.com/tool/flash-programmer>.

13. Combining the Bootloader + Application Image via Bin Files

For mass-production programming, it will be important to have a single image containing both the SBL Boot and Application code so that the part must only be programmed once. The following example assumes that the SmartRF Flash Programmer 2 tool will be used for programming a binary formatted file into the CC2538 SoC.

First, you will need to build the application and the bootloader as explained in the previous chapters.

Then, combine the two resulting .bin file into a single file, where the bootloader part is right after the application part. You may do so by running the following command line (assuming boot.bin is the bootloader image, and SampleLight.bin is the application image, and that they both exist in the same folder):

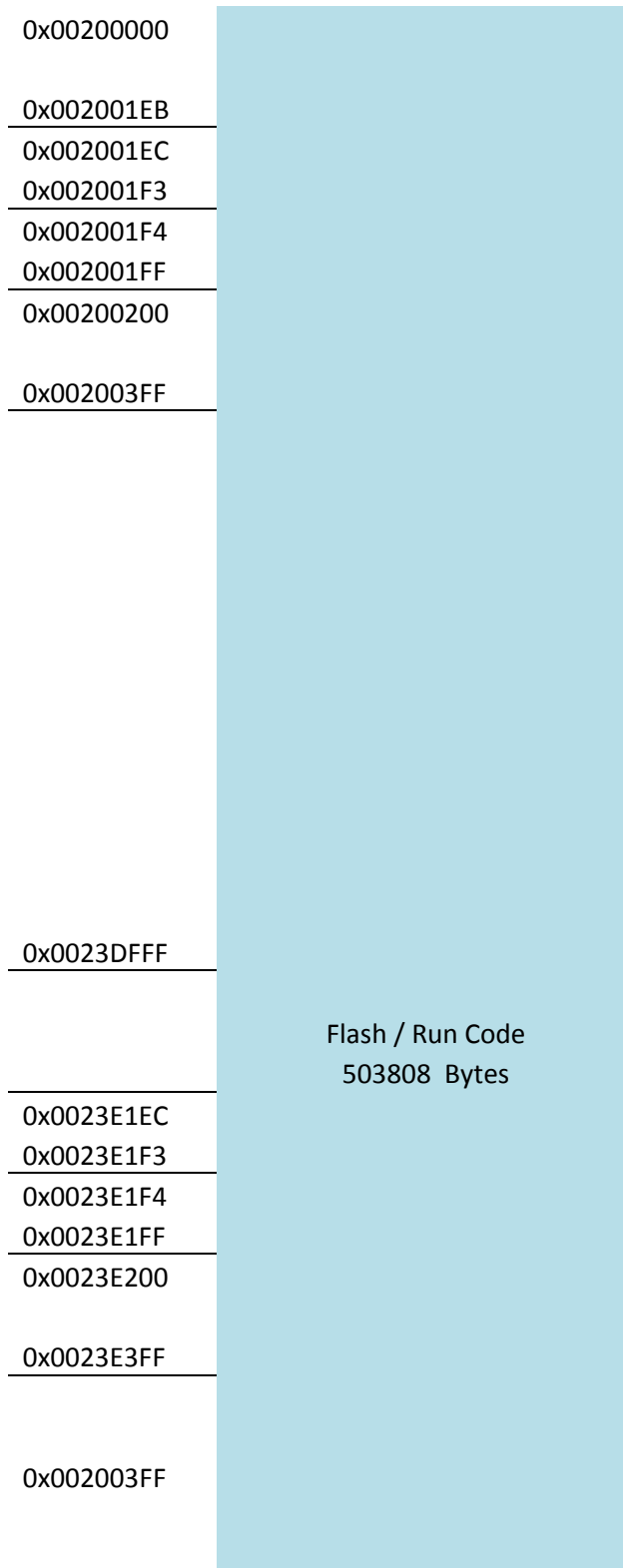
Copy /b SampleLight.bin + boot.bin SampleLight_with_bootloader.bin

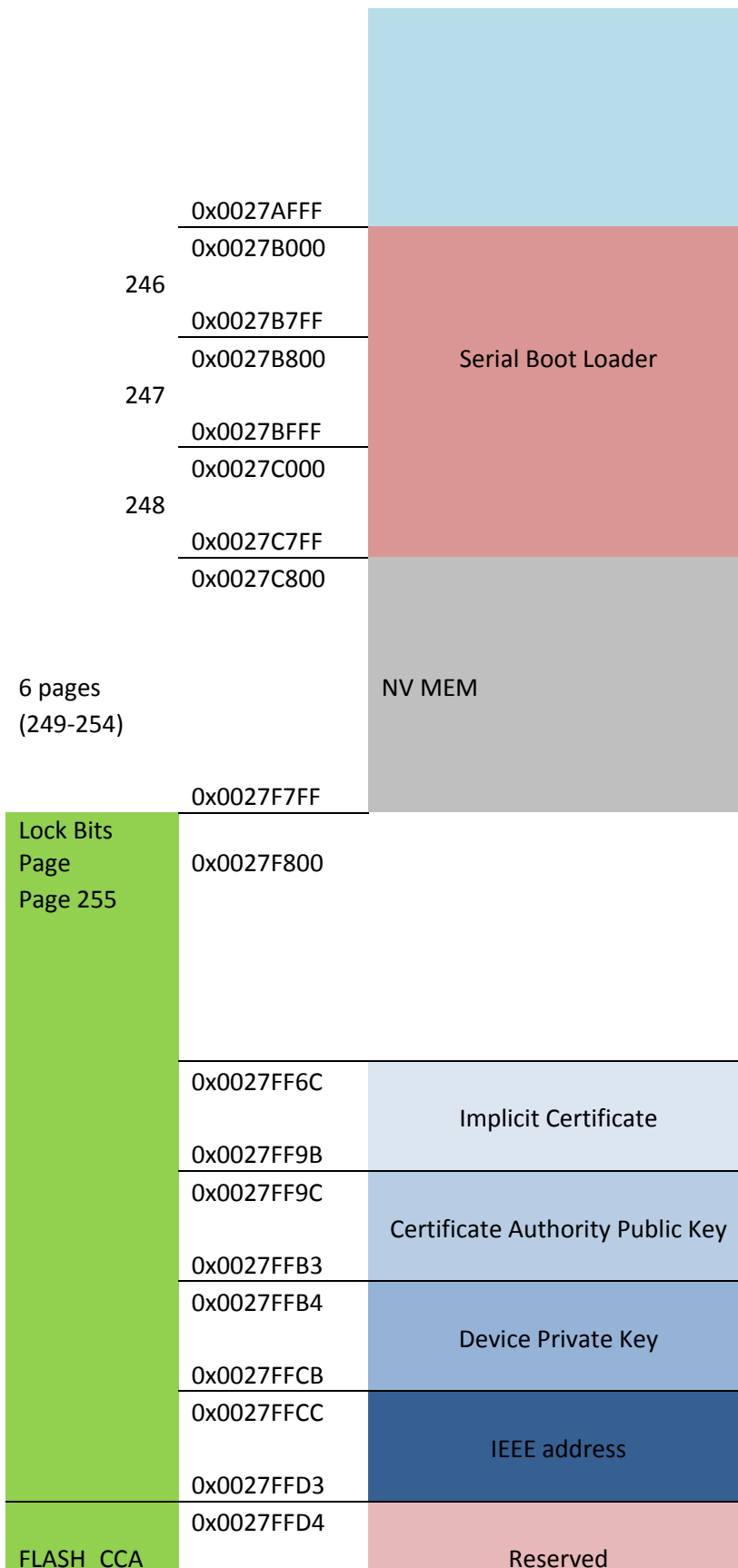
This will create a single image named SampleLight_with_bootloader.bin, containing the application image immediately followed by the boot image.

You can then program the CC2538 device using the resulting binary image and the SmartRF Flash Programmer 2 tool. The tool is available at <http://www.ti.com/tool/flash-programmer>. The image shall be loaded to the normal flash start address of CC2538, i.e. 0x00200000

14. Appendix: Memory Map for SBL on CC2538

SBL





	0x0027FFD6	
	0x0027FFD7	Bootloader Backdoor
	0x0027FFD8	Image Valid
	0x0027FFDB	
	0x0027FFDC	Application Entry Point
	0x0027FFDF	
	0x0027FFE0	Lock Bits
	0x0027FFFF	