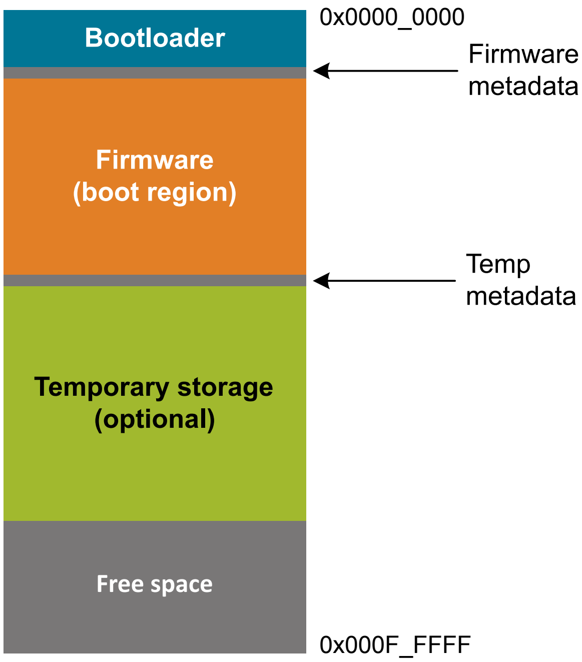
AN0060: Bootloader with AES Encryption

This application note describes the implementation of a bootloader capable of accepting AES-encrypted firmware updates. The full source code for a bootloader as well as a tool for encrypting firmware images on a PC are included.

This application note discusses implementations for EFM32GG and EFM32HG Series 0 Gecko MCU devices. For more information on the CRYPTO module included on Series 1 EFM32 and EFR32 devices, see *AN0955: CRYPTO*.

KEY POINTS

- This application note includes:
  - This PDF document
  - Bootloader C-code
  - 256-bit and 128-bit Encryption command line tool (in binary and C-code)



1. Introduction

Releasing new firmware updates can be a good way to extend the lifetime of a product. At the same time, whenever a new firmware is distributed there is a risk that people tamper with or copy the program. In some applications sensitive information has to be embedded in the firmware update itself. In these cases the firmware should be distributed in an encrypted form and only be decrypted after it is uploaded to the device itself.

This application note describes a bootloader implementation for the EFM32 Series 0 devices that can receive encrypted application upgrades. The private key is stored in the bootloader and is only otherwise known in the development environment. After a new firmware has been created it is immediately encrypted and can then be freely distributed. Only when the firmware is uploaded and stored in the internal flash on the EFM32 is the firmware decrypted.

Source code for both the bootloader and a PC encryption tool is provided with this application note. These can be used unmodified or extended to fit more specific requirements.

2. Security Considerations

The bootloader relies on private key cryptography. The key is only known in the development environment and inside the MCU. The encryption key has to be kept secret for all third parties. If a third party manages to get knowledge of the encryption key he can both steal and modify every new firmware.

When programming the EFM32 with the bootloader, the encryption key is stored in the internal flash. It is important that an intruder is not able to read this key.

One way to read out memory contents is over the debug interface. By default, the debug interface on the EFM32 is enabled. By connecting to this interface, it is possible to both read and write to the internal memory. This is necessary in order for developers to program and debug applications, but it also means that an attacker can use this interface to gain access to the encryption key. Therefore debug access should be locked before the product is shipped. Debug access is locked by clearing the Debug Lock Word and then resetting the EFM32. When debug access is locked, the core and debug registers are unavailable through the debug interface. The only way to unlock debug access again is through a Device Erase which will erase all Flash and SRAM (except the User Data Page, so the key must not be placed here!).

The bootloader needs a method to verify that the uploaded application is correct. Without this, an attacker could upload arbitrary data to the bootloader which would 'decrypt' the data and attempt to run it. This could lead to information about, or even recovery of, the encryption key. A hash of the entire application is therefore appended to the encrypted firmware. The bootloader will verify this hash before booting the application. The hash itself is also encrypted along with the firmware.

It is important to note that if the application itself has a vulnerability, an attacker may be able to exploit this to read contents of the flash including the encryption key. It is therefore important to consider security aspects also when developing the firmware.

A bootloader with encryption also does not protect against intrusive attacks. An attacker with physical access and enough resources and equipment could still open the chip and read out the flash contents.

An attacker with physical access to the debug pins can also reprogram the entire MCU with his own code. In this case however, the encryption key is not revealed and the source code for the original firmware is still safe.

3. Bootloader Implementation

This chapter will explain the bootloader implementation and functionality. The bootloader will either attempt to boot the application in flash or wait for a new firmware update depending on the state of a preconfigured pin when coming out of RESET. When uploading new firmware, the bootloader will decrypt, verify and store the new application in flash. It can optionally be configured to make use of a temporary storage area, to make sure that the MCU always contains a valid program.

3.1 Bootloader State Machine

The figure below shows the state diagram for the bootloader. Out of reset the bootloader will first check the state of the PF0 pin (this can be configured to use any GPIO pin). If this pin is pulled HIGH, the MCU will enter [**bootloader mode**] and wait for commands over UART. The default pins for TX/RX are PB9/PB10 on the EFM32GG-DK3750 and PE10/PE11 for EFM32HG-SLSTK3400A. These pins are connected to the RS232 connector. If it receives the upload command (lower case 'u'), it will enter [**upload mode**], where it will accept an encrypted firmware image over the XMODEM-CRC protocol.

In [**upload mode**], the bootloader will receive XMODEM-CRC packets, decrypt them using the private key, and write them to internal flash. The firmware should be prepared by using the encrypt tool provided with this application note. The EFM32GG supports 256-AES encryption, and EFM32HG supports 128-AES encryption, so the appropriate version of the tool must be used with each supported device.

If the PF0 pin is left LOW when the MCU comes out of reset, the bootloader will attempt to boot the application in flash. The bootloader will first verify the application by checking the verified flag in the firmware header. If temporary storage is enabled and the application in the boot area fails verification the bootloader will check the temporary storage. If it finds a valid application here it will start copying it to the boot area and then boot it. When there are no valid applications in memory the bootloader will enter a 'low power wait mode' and wait for the bootloader pin to be pulled HIGH.

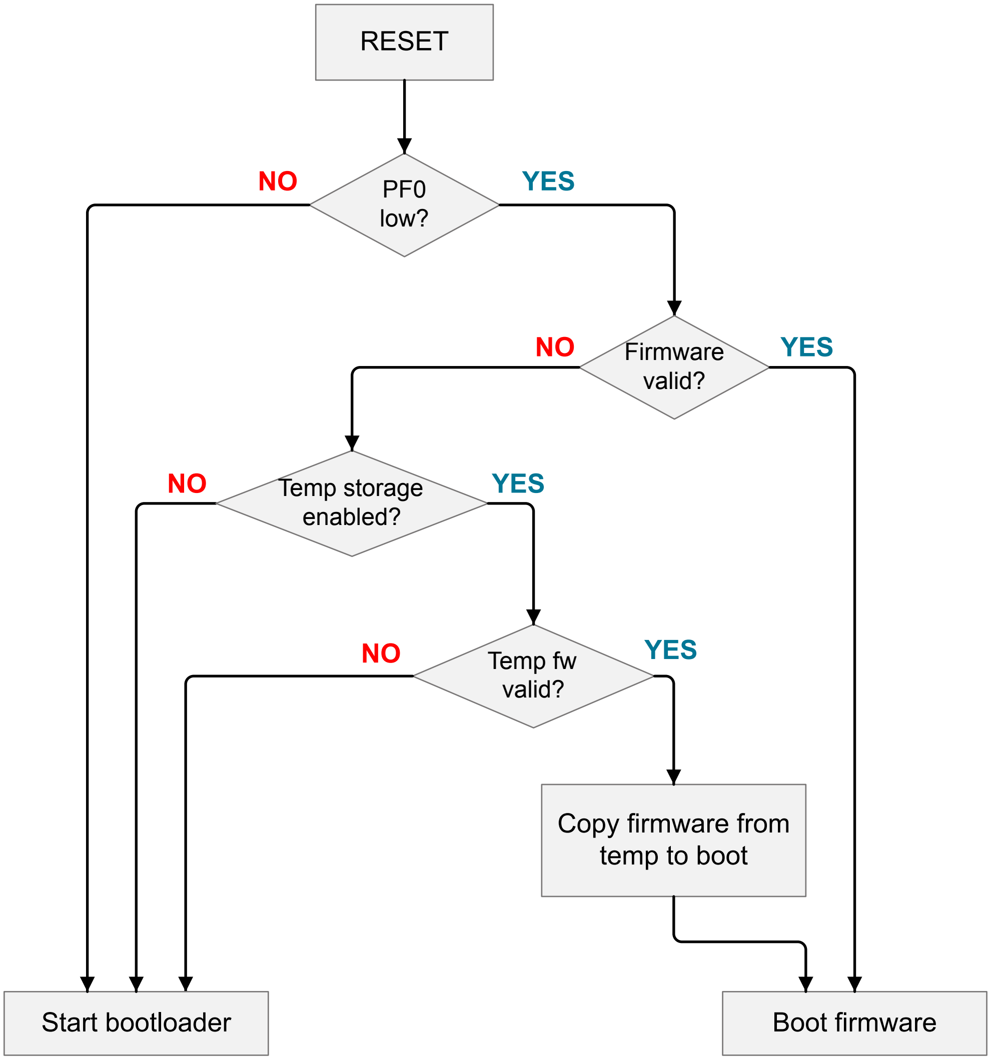


Figure 3.1. Bootloader State Machine

3.2 Decryption

When a new firmware is uploaded, it is decrypted on the fly before writing it to flash. The encryption algorithm is AES in CBC (Cipher Block Chaining) mode. Decryption is performed using the internal AES peripheral in the EFM32. See *AN0033: AES Cipher Modes with EFM32* for more information about the decryption algorithm and AES peripheral.

The encryption key is stored in the bootloader at compile time by including the file `aes_keys.c`, which is generated from the encryption tool. See 4.2 Preparing Encryption Keys for information about how to generate this file.

3.3 Memory Layout

The following figure describes the memory layout of the boot loader. The bootloader is placed at the start of flash. This ensures that the bootloader will always be started first when the MCU is reset. The firmware region (boot region) is placed after the bootloader. This region includes both meta information about the firmware (application size, hash and a verified flag) and the actual application code. If temporary storage is enabled it is placed after the boot region. The temporary storage needs to be the same size as the boot region, since the bootloader will store a copy of the firmware image here. The rest of flash is free to be used by the application itself to store non-volatile data.

The size of each region can be configured when compiling the bootloader. The configuration is in `bootloader_config.h`.

Note: The size of each region should be aligned to page size borders.

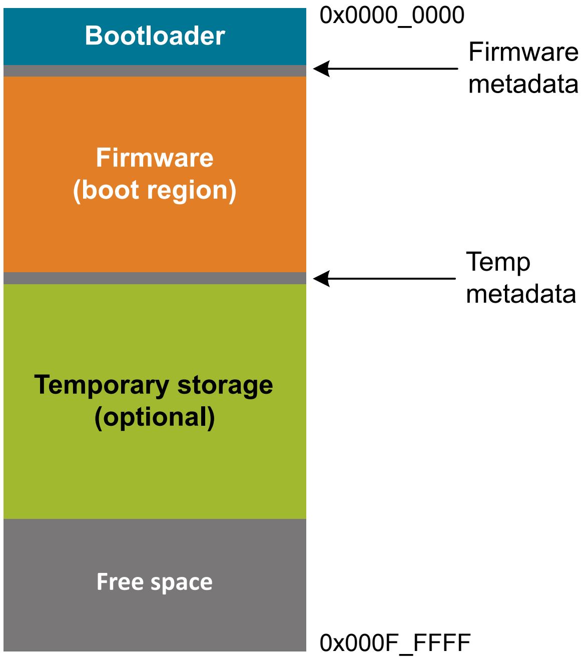


Figure 3.2. Memory Layout

3.4 Firmware Verification

After the firmware has been decrypted and written to flash, a verification algorithm computes a hash of the entire image to check that it is in fact a valid application. The computed hash is compared against the expected hash stored in the firmware header. If it matches, the verified field in the header is updated to indicate that the firmware is valid. The bootloader later only has to check this field to verify that the current firmware is valid.

The verification algorithm is a 128-bit CBC AES with its own key and initial vector. All the decrypted blocks are discarded except the last block. The last cipher block is dependent on the entire image and is used as the hash.

3.5 Temporary Storage

If it is important that the MCU always contain a valid application, even in the case of a critical failure during update, the bootloader can be configured to make use of a temporary storage when uploading a new firmware. When this option is enabled, the bootloader will not immediately overwrite the existing application, but instead decrypt and verify the new firmware in using a separate region of flash. Only when the uploaded application has passed the verification will it be copied to the boot area. Even if a power loss occurs during copying the image, the temporary storage still contains a valid application. The bootloader will sense this at boot and restart the copy operation. The drawback of this method is that the MCU needs twice as much flash as required to hold the firmware image.

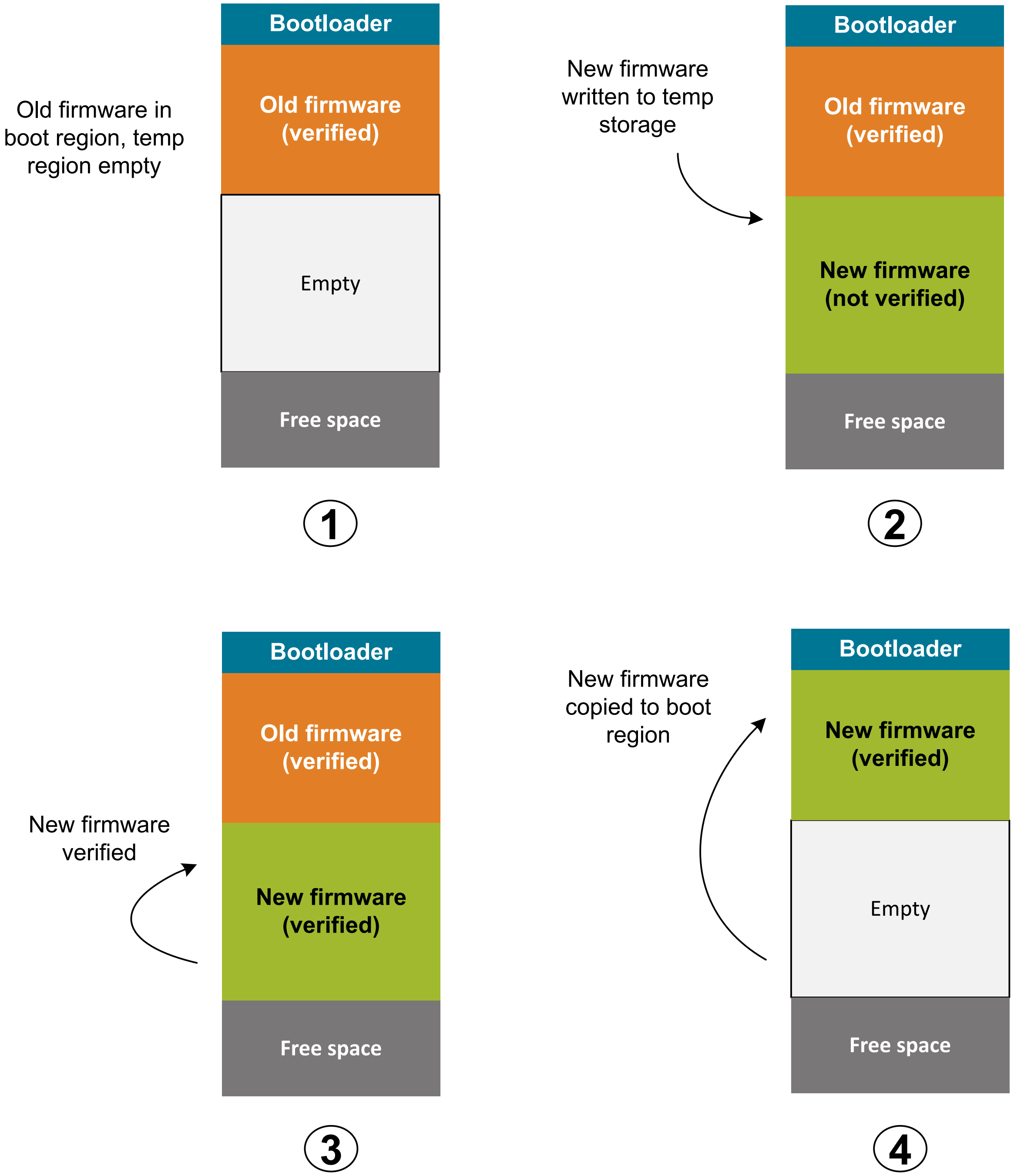


Figure 3.3. Using Temporary Storage

3.6 Uploading New Firmware

To upload new firmware, the bootloader should first be started by pulling the bootloader pin HIGH and then reset the MCU. When the bootloader has started, it will accept commands over the UART interface. To upload a new firmware the remote side should first send a u (lower case 'u') and then start sending the encrypted firmware with the XMODEM-CRC protocol. If using a PC, any program capable of sending files with the XMODEM-CRC protocol can be used. The bootloader has been tested on Windows 7 with Tera Term. After the firmware has been uploaded, release the bootloader pin and reset the MCU. The new firmware will then be booted.

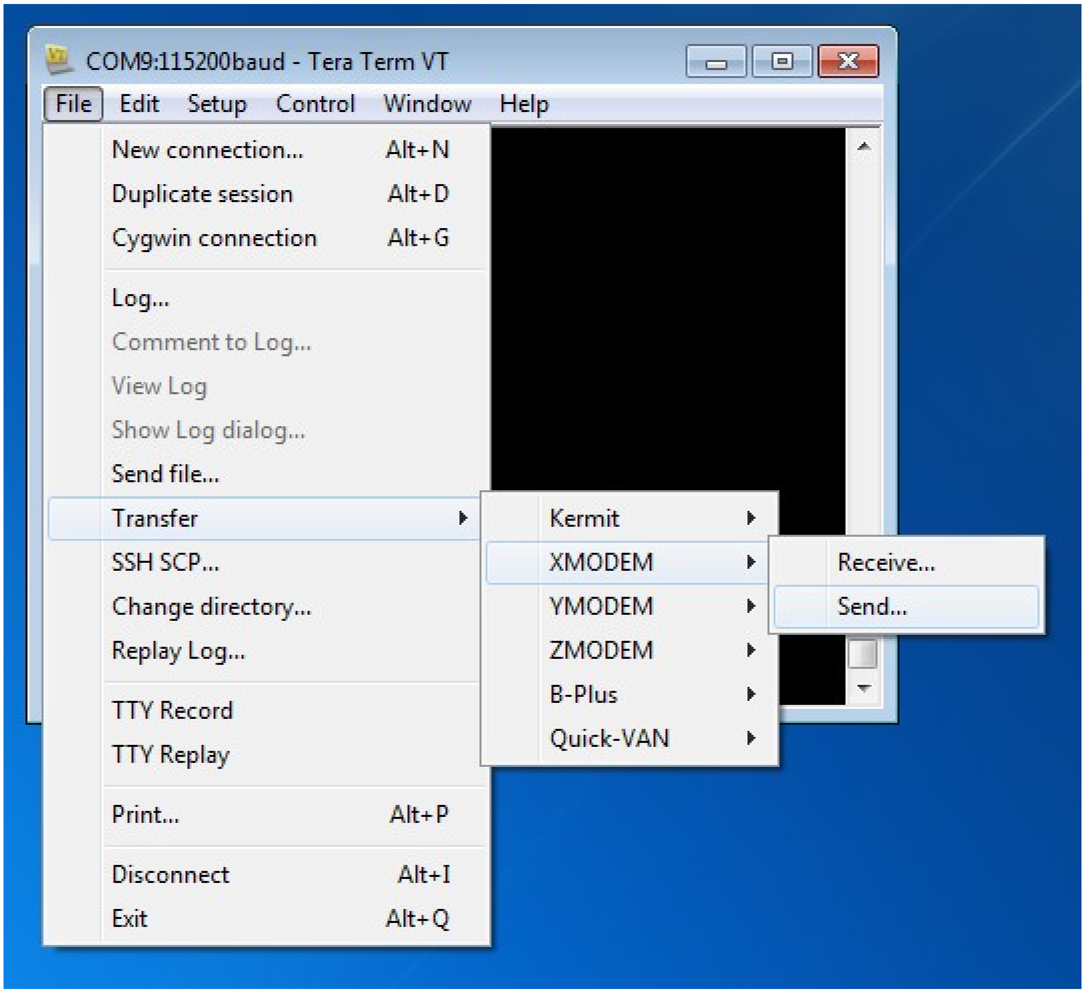


Figure 3.4. Sending XMODEM File with Tera Term

The following commands are accepted by the bootloader in [**bootloader mode**]:

u

upload encrypted firmware

r

reset the MCU

h

displays help

v

verifies the boot area

l

lock debug access

3.7 Locking Debug Access

As mentioned in 1. Introduction, Debug Lock should be enabled to protect the private key. The bootloader can lock debug access by using the **[l]** command and then performing a hard reset (pin reset). The **[l]** command will clear the Debug Lock Word. After the next reset debug access will be locked. The only way to regain debug access is to perform a Device Erase via the AAP registers as described in the Debug Interface chapter in the reference manual. A Device Erase will erase the entire Flash and SRAM contents (except the User Page).

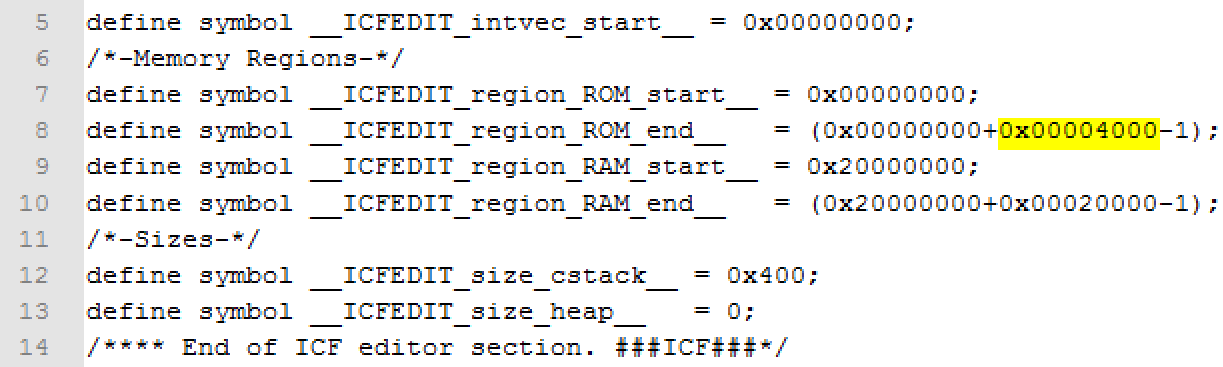
Note: On EFM32G Gecko devices, the Debug Lock Word can only be cleared when in debug mode. On these devices, the debug interfaces is first bit-banged to enter debug mode before clearing the word. For the bit-bang sequence to work, the SWDIO pin must be left unconnected.

3.8 Compiling the Bootloader

Before compiling the bootloader, the file `aes_keys.c` must be generated using the encrypt tool. The application note contains an example `aes_keys.c` which should be replaced with the generated file. See 4.2 Preparing Encryption Keys on how to generate this file.

The configuration file, `bootloader_config.h`, should be modified to select the pins used and the size of the firmware region(s). If the temporary storage is used, it must be the same size as the boot region. Both firmware regions should be aligned to page boundaries. Note that different EFM32 devices have different page sizes. See the reference manual for each device for information about the page size.

Make sure the bootloader and firmware regions do not overlap. Modifying the bootloader or using different compilers can produce a bootloader which is too large, causing it to overwrite parts of itself when uploading new firmware. It is therefore advisable to limit the size of the bootloader at compile time. This is done by restricting the available space for the bootloader in the linker file. The figure below shows how the linker file in IAR is modified to limit the size of the flash when compiling the bootloader to 0x4000 bytes. With this limitation in place the compiler will never create a bootloader which is larger than this and therefore it is safe to place the boot region at 0x4000.



```
define symbol __ICFEDIT_intvec_start__ = 0x00000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__  = 0x00000000;
define symbol __ICFEDIT_region_ROM_end__    = (0x00000000+0x00004000-1);
define symbol __ICFEDIT_region_RAM_start__  = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__    = (0x20000000+0x00020000-1);
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x400;
define symbol __ICFEDIT_size_heap__   = 0;
/**** End of ICF editor section. ###ICF###*/
```

Figure 3.5. Configuring the Max Size of the Bootloader

3.8.1 Bootloader Source

Below is a brief overview of the source files for the AES Bootloader and what they do.

- `aes.c` — Encryption and decryption functions using the integrated AES peripheral.
- `boot.c` — Boots the firmware.
- `bootloader.c` — Main file for the bootloader. Contains the UART command loop.
- `crc.c` — CRC calculation algorithm for the XMODEM protocol.
- `flash.c` — Functions for writing and erasing pages of flash.
- `verify.c` — Calculates hash of the firmware.
- `uart.c` — UART communication.
- `xmodem.c` — Implementation of the XMODEM-CRC protocol.
- `bootloader_config.h` — Compile time configuration of the bootloader.

4. Encrypting Firmware

This chapter explains how to encrypt firmware for use with the AES Bootloader. To encrypt the firmware, a command line tool, encrypt.exe, is used.

4.1 Compile and Link Application

When compiling the firmware, the linker file must be updated to place the application at the correct location in flash. This must match the configuration of the bootloader. The boot address must be offset 0x100 bytes from the start of the boot region defined in the bootloader because this space is reserved for metadata about the firmware.

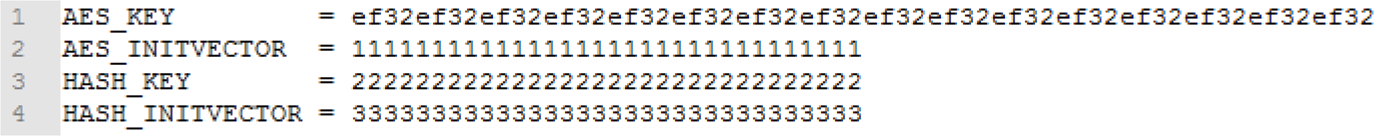
As an example, let the boot region as defined in `bootloader_config.h` be from 0x00004000 and of size 0x60000 (end address is 0x00063FFF). The linker file when compiling the firmware should then define the start of ROM to be 0x00004100 and the size to be 0x5ff00 (0x60000 - 0x100). This will ensure that the firmware is placed in the correct location and that it will not be too large to fit in the boot region.

For how to configure the linker file to ensure the application at the correct location in flash, please reference 6. Appendix B — Editing Firmware Linker Files.

Many IDEs will output an object file which includes debug information and other meta data. To encrypt the firmware, it is necessary to produce a raw binary image of the application. The raw binary image will be the input to the encrypt.exe encryption tool. See 5. Appendix A — Creating Binary Output for instructions on how to do this in different IDEs.

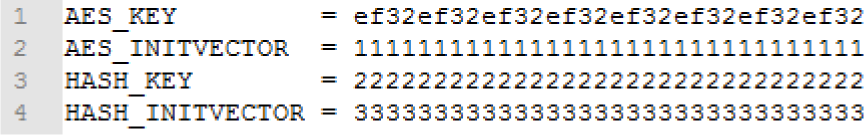
4.2 Preparing Encryption Keys

The `encrypt.exe` tool expects a config file called `keys.txt` to be present in the same folder. The config file specifies the 256-bit or 128-bit encryption key and initial vector for the CBC AES algorithm. It also provides the key and initial vector for generation of the verification hash. The template config files are included.



```
AES_KEY         = ef32ef32ef32ef32ef32ef32ef32ef32ef32ef32ef32ef32ef32ef32ef32ef32
AES_INITVECTOR  = 11111111111111111111111111111111
HASH_KEY        = 22222222222222222222222222222222
HASH_INITVECTOR = 33333333333333333333333333333333
```

Figure 4.1. 256-Bit Encryption Config File, `keys.txt`



```
AES_KEY         = ef32ef32ef32ef32ef32ef32ef32ef32
AES_INITVECTOR  = 11111111111111111111111111111111
HASH_KEY        = 22222222222222222222222222222222
HASH_INITVECTOR = 33333333333333333333333333333333
```

Figure 4.2. 128-Bit Encryption Config File, `keys.txt`

The config file should be edited to include your own secret keys and initial vectors. Once the `keys.txt` file has been updated, the `encrypt.exe` tool can automatically generate the `aes_keys.c` file that is needed when compiling the bootloader, by running the following command:

```
encrypt.exe --create-bootloader-keys
```

4.3 Encrypt Binary Image

To encrypt the firmware image, run `encrypt.exe` with two parameters: the raw binary image and the path to write the encrypted image. To encrypt the `blink.bin` file run the following (assuming that blink.bin is in the same folder as `encrypt.exe`):

```
encrypt.exe blink.bin encrypted_blink.bin
```

This creates a new file, `encrypted_blink.bin`, which contains the encrypted version of the firmware. The encrypted file also includes the generated hash which is used to verify the application once it has been decrypted on the MCU. The entire file is padded to fit within an integer number of XMODEM packets.

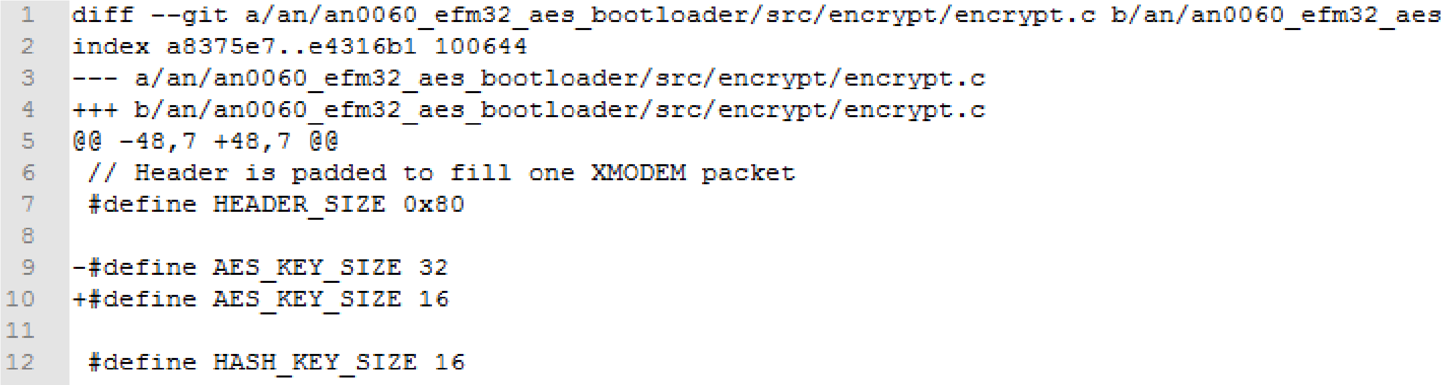
The encrypted file can be uploaded directly to the MCU via a terminal program, as explained in 3.6 Uploading New Firmware.

4.4 Compiling the Encrypt Tool

The full source code for the encrypt tool is provided with this application note. The tool uses the `libtomcrypt` library to perform the actual AES encryption. `Libtomcrypt` can be downloaded from http://www.libtom.org/. The tool can be compiled with the following command (assuming both gcc and `libtomcrypt` have been installed):

```
gcc -o encrypt encrypt.c --ltomcrypt
```

By default, the source code will generate the 256-bit AES encryption tool. To generate the 128-bit AES encryption tool, modify `AES_KEY_SIZE` from 32 to 16 within `encrypt.c`:



```
diff --git a/an/an0060_efm32_aes_bootloader/src/encrypt/encrypt.c b/an/an0060_efm32_aes
index a8375e7..e4316b1 100644
--- a/an/an0060_efm32_aes_bootloader/src/encrypt/encrypt.c
+++ b/an/an0060_efm32_aes_bootloader/src/encrypt/encrypt.c
@@ -48,7 +48,7 @@
 // Header is padded to fill one XMODEM packet
 #define HEADER_SIZE 0x80

-#define AES_KEY_SIZE 32
+#define AES_KEY_SIZE 16

 #define HASH_KEY_SIZE 16
```

Figure 4.3. Generating the 128-Bit AES Encryption Tool

5. Appendix A — Creating Binary Output

This appendix shows how to configure IDEs to generate binary output.

5.1 Simplicity Studio

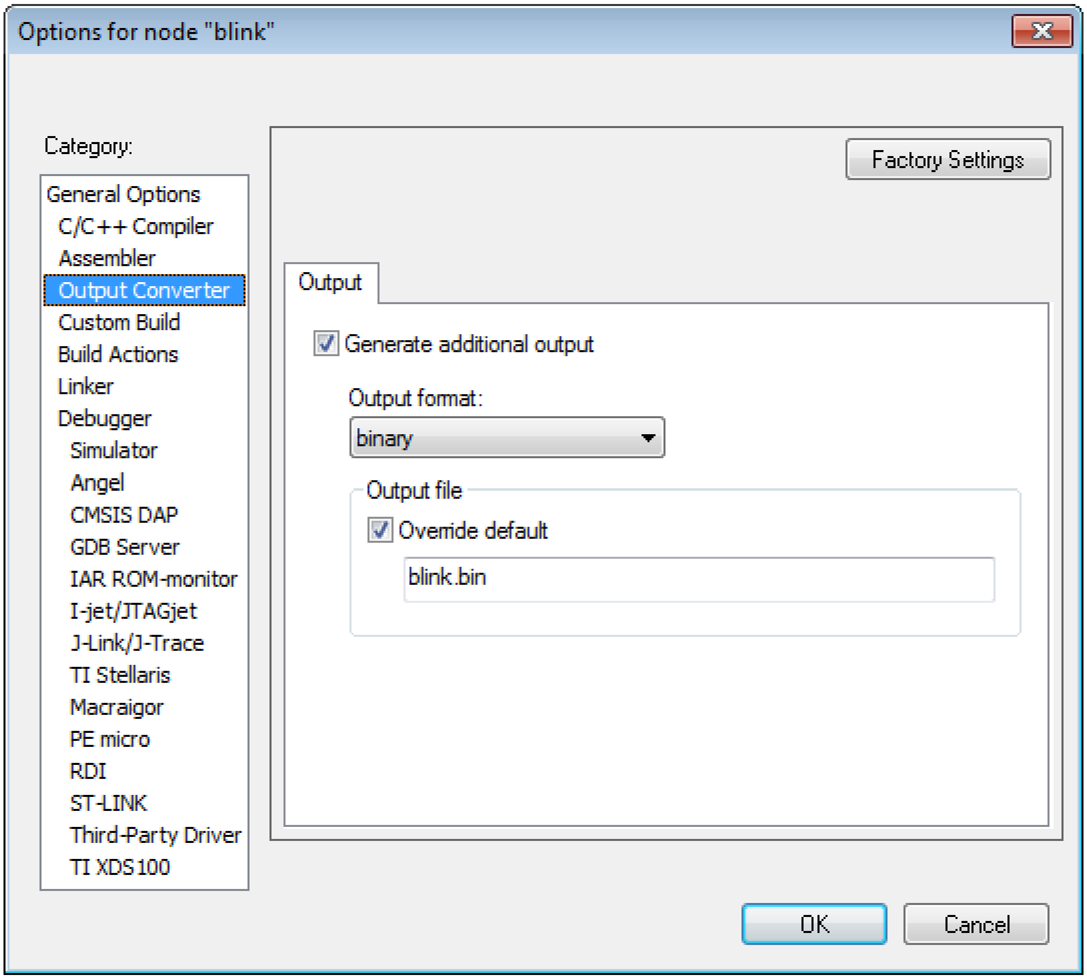
Simplicity Studio will generate the binary file output automatically by default. No additional steps are needed when using Simplicity Studio to create this file.

5.2 IAR Embedded Workbench

For IAR Embedded Workbench:

1. Open [**Project**]>[**Options**]>[**Output Converter**].
2. Select binary output as shown in the figure below.
3. Build the project normally.

The binary file can be found under `\Debug\Exe`.



☑ Generate additional output

Output format: binary

Output file

☑ Override default

blink.bin

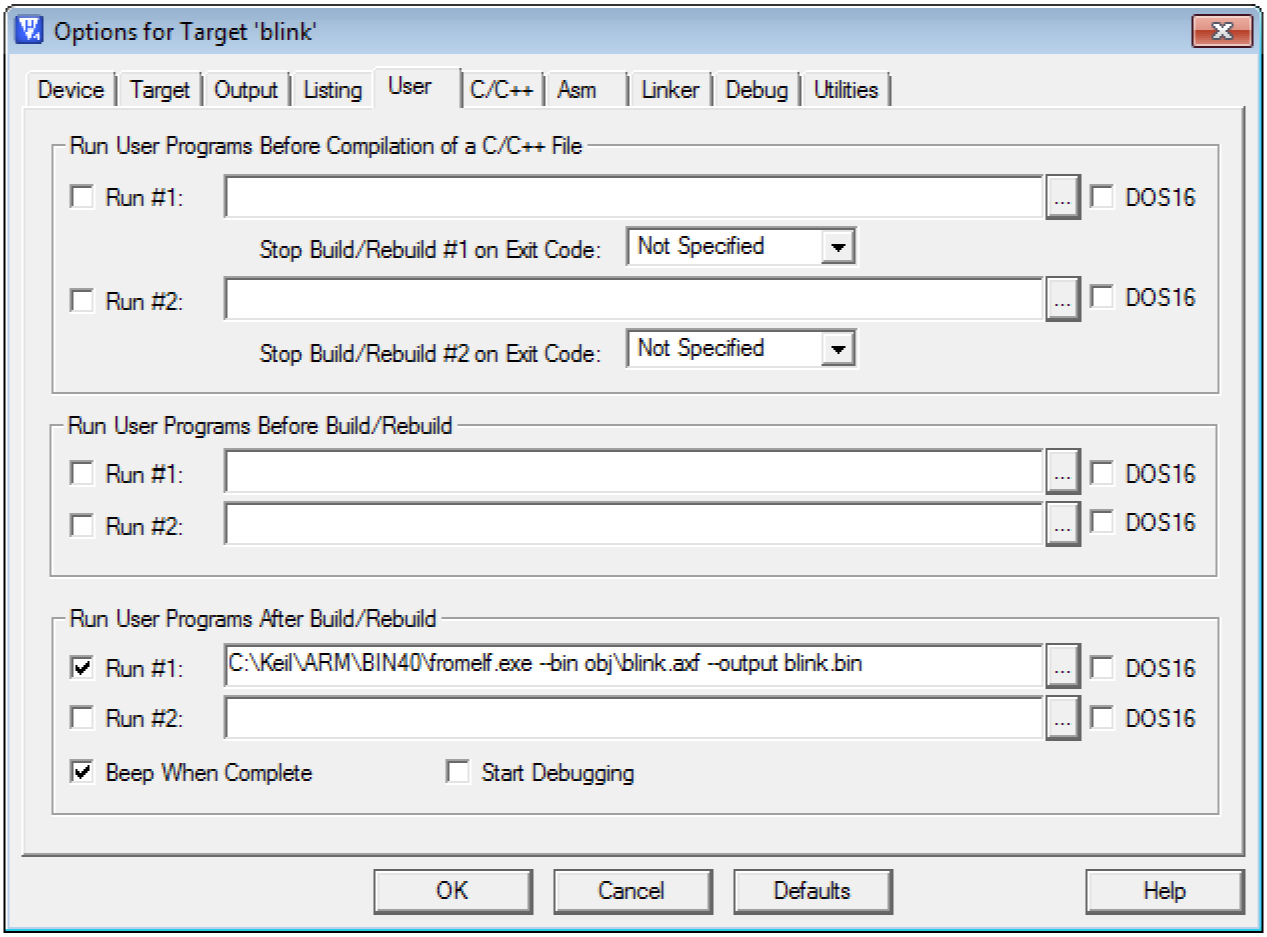
Figure 5.1. Binary Output in IAR

5.3 Keil MDK-ARM µVision

For Keil µVision:

1. Open [**Project**]>[**Options**]>[**User**].
2. Enter the following command in the [**Run User Programs After Build/Rebuild**] area (change the path if you have installed Keil in a different directory. <filename> refers to the output filename of your project):

```
C:\Keil\ARM\BIN40\bin\fromelf.exe --bin obj\<filename>.axf --output<filename>.bin
```



Run User Programs Before Compilation of a C/C++ File

☐ Run #1: ☐ DOS16

Stop Build/Rebuild #1 on Exit Code: Not Specified

☐ Run #2: ☐ DOS16

Stop Build/Rebuild #2 on Exit Code: Not Specified

Run User Programs Before Build/Rebuild

☐ Run #1: ☐ DOS16

☐ Run #2: ☐ DOS16

Run User Programs After Build/Rebuild

☑ Run #1: C:\Keil\ARM\BIN40\fromelf.exe –bin obj\blink.axf –output blink.bin ☐ DOS16

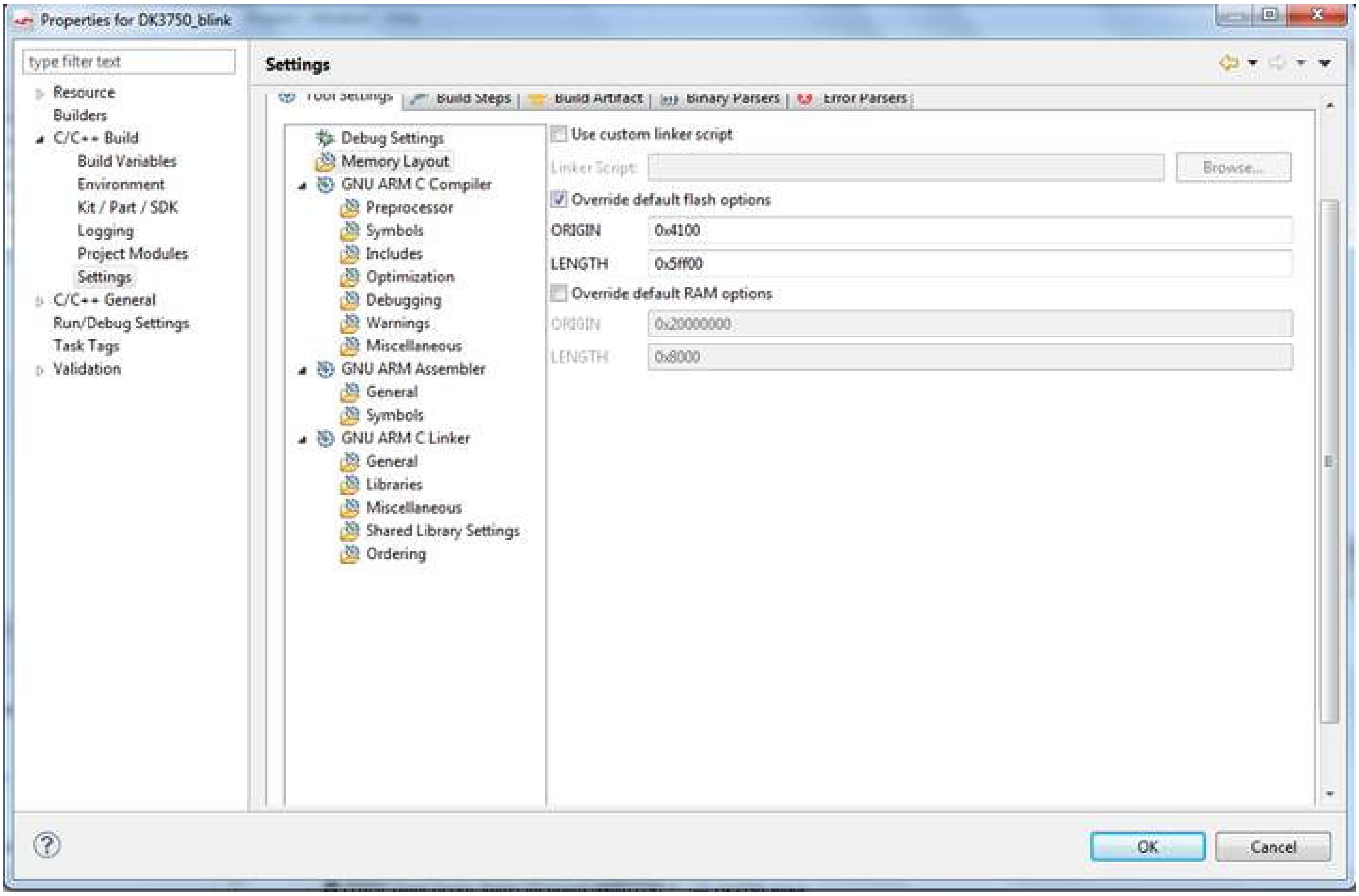
☐ Run #2: ☐ DOS16

☑ Beep When Complete ☐ Start Debugging

Figure 5.2. Binary Output in µVision

6. Appendix B — Editing Firmware Linker Files

This section illustrates which parameters must be edited in the linker files when compiling firmware for the EFM32GG-DK3750 AES bootloader. For the EFM32HG-SLSTK3400A bootloader, reference the `bootloader_config.h` file for the correct location.



□ Use custom linker script

Linker Script:

☑ Override default flash options

ORIGIN 0x4100

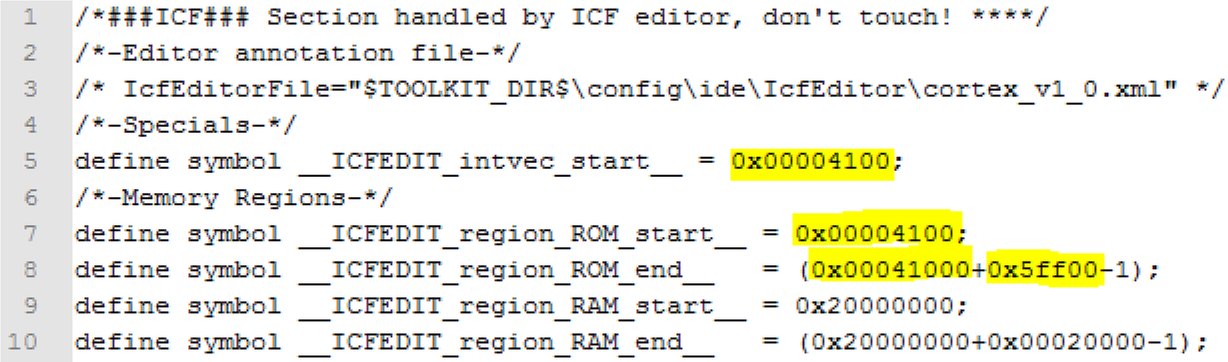
LENGTH 0x5ff00

□ Override default RAM options

ORIGIN 0x20000000

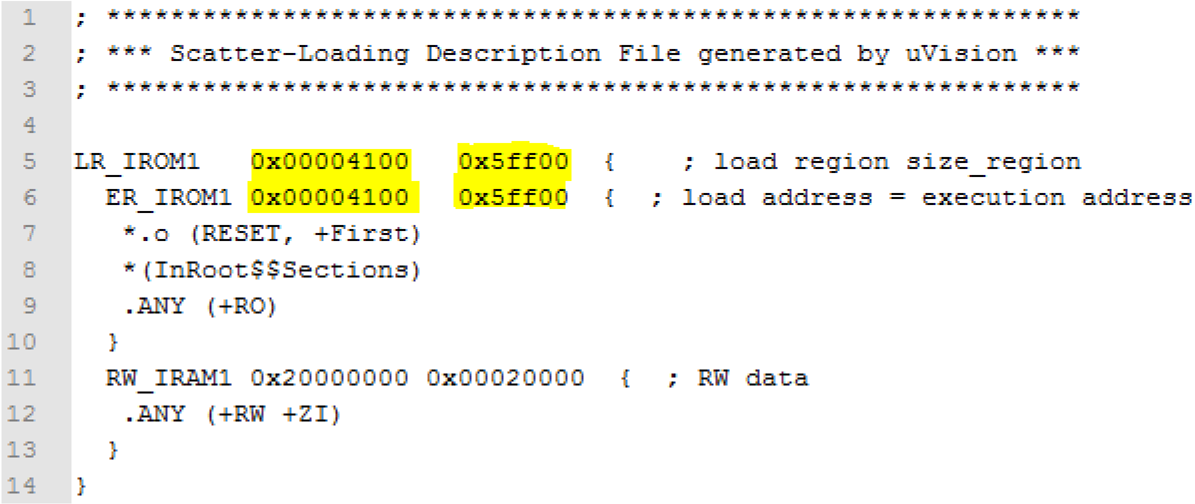
LENGTH 0x8000

Figure 6.1. Simplicity Studio Firmware Linker File



```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x00004100;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__  = 0x00004100;
define symbol __ICFEDIT_region_ROM_end__    = (0x00041000+0x5ff00-1);
define symbol __ICFEDIT_region_RAM_start__  = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__    = (0x20000000+0x00020000-1);
```

Figure 6.2. IAR EWARM Firmware Linker File



```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

LR_IROM1    0x00004100    0x5ff00  {    ; load region size_region
  ER_IROM1 0x00004100    0x5ff00  {  ; load address = execution address
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
  RW_IRAM1 0x20000000 0x00020000  {  ; RW data
   .ANY (+RW +ZI)
  }
}
```

Figure 6.3. Keil MDK-ARM μVision Firmware Linker File

7. Revision History

7.1 Revision 1.05

2016-11-02

Add EFM32HG 128-AES encryption support.

Removed Rowley and ARM GCC support, including the figures in 6. Appendix B — Editing Firmware Linker Files.

Added Simplicity Studio support.

7.2 Revision 1.04

2013-09-23

Interrupts are now disabled during flash write

7.3 Revision 1.03

2014-05-07

Updated example code to CMSIS 3.20.5

Changed source code license to Silicon Labs license

Added project files for Simplicity IDE

Removed makefiles for Sourcery CodeBench Lite

Added linkerfiles for ARM GCC

7.4 Revision 1.02

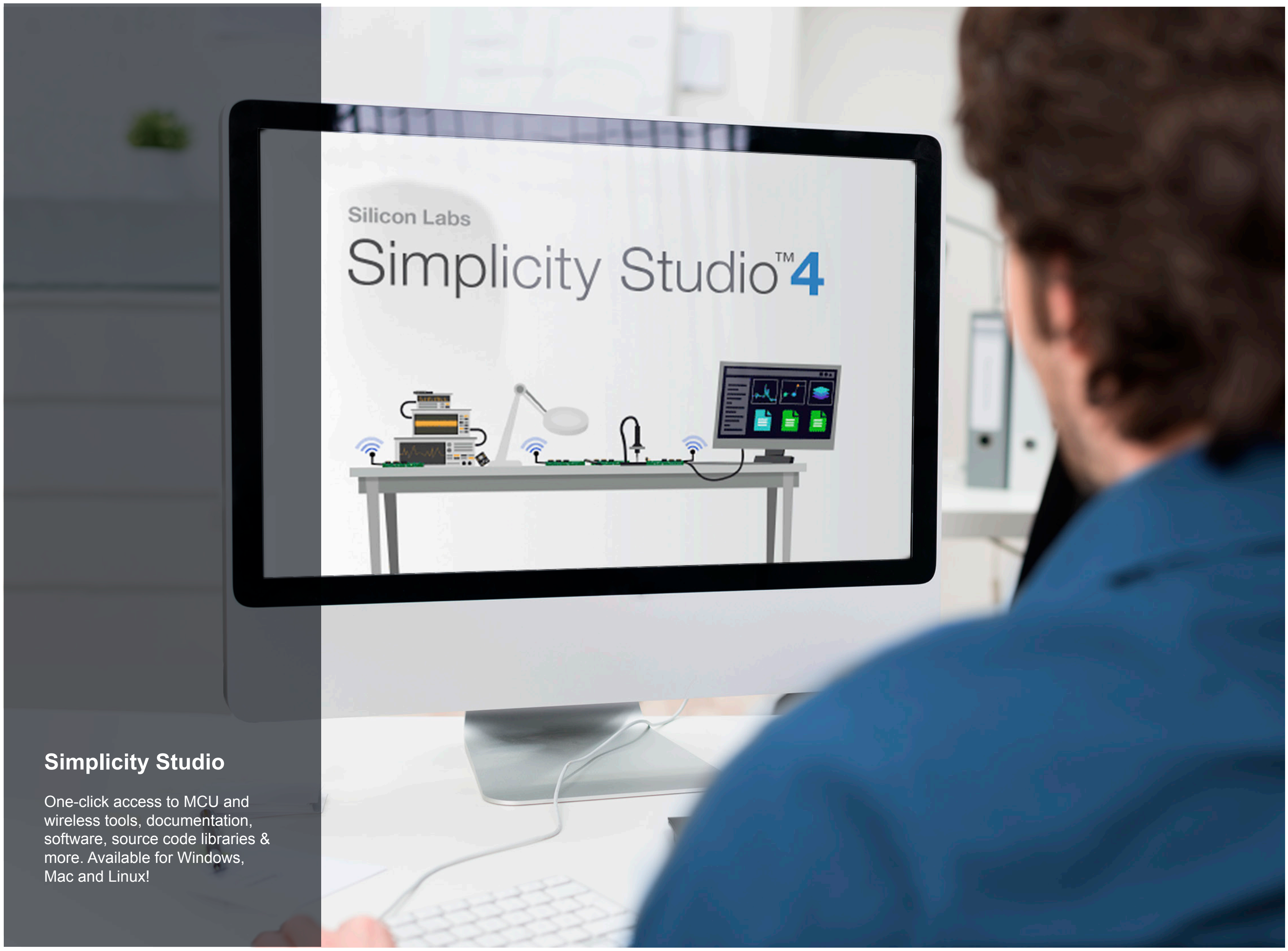
2013-10-14

New cover layout

7.5 Revision 1.00

2013-05-08

Initial revision.



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

Disclaimer

Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Labs shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any Life Support System without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

Trademark Information

Silicon Laboratories Inc.® , Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, Clockbuilder®, CMEMS®, DSPLL®, EFM®, EFM32®, EFR, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, ISOmodem®, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress® and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. All other products or brand names mentioned herein are trademarks of their respective holders.

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com