# Si4010 NVM Burning Tools and Flows

## 1. Introduction

This document is a user's guide for the Si4010 NVM composer and burner related to the customer burn flow. It covers the details of the NVM organization, the actual burn algorithm, data composer tool, and recommended CRC flow. It also covers in detail how to program the Si4010 device with different configurations for each part. Finally, the last chapter describes how to use the NVM burning toolset to write the MTP memory of the Si4010.

## 2. Si4010 NVM Programming

This document describes the strategies for programming the NVM in customer production and focuses on command line versions of the tools. The boot process, NVM memory organization, and burn process are described in detail.

The Si4010 NVM Programming Utility, a graphical user interface (GUI), is also available for burning Si4010 devices. However, the GUI has some restrictions comparing the burn flows and tools described in this document. For more information on the GUI, refer to application note AN511. The burn tools can be downloaded from **http://www.silabs.com/Support%20Documents/Software/Si4010_Burn_Tools.exe**

### 2.1. System Requirements

The tools run on MS Windows XP and higher. The following must be installed on the customer machine:

1. The Microsoft Visual C++ 2010 redistributable DLL package must be installed on the user machine. If not, the tools will not work. For example, if the command line burner is run on the command line inside the **cmd.exe** window without any arguments and there is no screen output, then some required DLLs are missing. The MS Visual C++ 2010 redistributable package can be downloaded from Microsoft directly:

   Microsoft Visual C++ 2010 Redistributable Package (x86)

   **`http://www.microsoft.com/download/en/details.aspx?id=5555`**

Due to its size, it is not included in the tool distribution file.

2. The **hexext**, **hexdiff**, **nvmrev** and **nbfmod** scripts are Perl scripts. If the user desires to run these, there are two options:
   a. Install the free **Cygwin** library and toolset along with Perl and run those scripts from within **Cygwin**. The rest of the tools can be run from the **Cygwin** window as well.

      **`http://www.cygwin.com`**

   b. Install free Strawberry Perl and run the ***.bat** versions of the scripts mentioned above.

      **`http://strawberryperl.com`**

The scripts are guaranteed to run with version 5.12.1.0 of Strawberry Perl and 5.10.1 of the **Cygwin** version of Perl. In general, both 5.10 and 5.12 versions of Perl should work without problems. The tools were also tested with Strawberry Perl version 5.16.2.1

Apart from the supporting Perl scripts, there are three main executables related to Si4010 NVM programming. The terminology is as follows:

- CL .. command line
- GUI .. graphical GUI tool

The tools are:

1. GUI **`Si4010_NVM_Burner`** .. it performs two tasks:
   a. Graphical wrapper around the CL **`gui_composer`** to generate NBF file.

b. Interface to the Silicon Labs debug chain to use generated or existing NBF file, parse it, download the burn files and run them to achieve the NVM programming.

2. CL **`gui_composer`** .. it takes customer application and/or config IntelHEX data and generates an NVM burn file (NBF). The NBF file is a ASCII wrapper containing one or more IntelHEX files to be sent to the part and run to perform NVM burning.
3. CL **`Si4010_NVM_Burn_CL`**: Does the item b. of the GUI **`Si4010_NVM_Burner`**; it interfaces to the Silicon Labs debug chain, takes the existing NBF file, parses it, downloads the burn files, and runs them to achieve the NVM programming.

## 2.2. Delivered Tools

Table 1 describes all the files related to the NVM data composure and NVM burning.

**Table 1. NVM Data Composure and Burning Files**

| File Name | Description |
|---|---|
| `Si4010_NVM_Burner.exe` | Si4020 NVM Programming Utility. Main GUI wrapper, which invokes both the composing step by calling **gui_composer.exe** and the NVM burning step by loading the NBF file and burning its contents to the device. |
| `Si4010_NVM_Burn_CL.exe` | Command line burner. It reads the NBF file and burns its contents into the device. |
| `burn_cl.bat` | Convenience BAT wrapper around the **Si4010_NVM_Burn_CL.exe** interpreting and printing exit error values for user convenience. |
| `SiDebug.dll`<br>`USBHID.dll` | Silicon Labs libraries required for the tools to connect to the USB Debug Adapter and the device. They must reside in the same directory as the burner tools. |
| `gui_composer.exe`<br>`gui_burn.hex`<br>`mtp_burn.hex`<br>`library.zip`<br>`w9xpopen.exe`<br>`unicodedata.pyd`<br>`python25.dll`<br>`MSVCR71.dll`<br>`bz2.pyd` | Command line composer tool for converting the user application IntelHEX files into the burn NBF file. Invoked by the **Si4010_NVM_Burner.exe** GUI but can be invoked on a command line manually. The other files are required by the **gui_composer.exe** executable. |
| `hexext`<br>`hexext.bat` | Perl script for IntelHEX and Verilog MEM file format conversion, file concatenation, data extraction, and IntelHEX checksum fixing. |
| `hexdiff`<br>`hexdiff.bat` | Perl script for comparing IntelHEX and/or Verilog MEM files. It can load several files at the same time. |
| `nvmrev`<br>`nvmrev.bat` | Perl script for burned NVM image debugging. It converts the NVM image in HEX or MEM format generated by the `nbfmod --hex ...` or `nbfmod --mem ...` script back to the original user source HEX or MEM files, which were the inputs to the composer/burn process to generate NBF file(s). |

**Table 1. NVM Data Composure and Burning Files (Continued)**

| File Name | Description |
|---|---|
| ./nbf/nbfmod<br>./nbf/nbfmod.bat | NBF file concatenation and modification tool. Primary use is for sequential concatenation of data from several NBF files to create a single NBF burn file for single step burning. It can also generate the actual NVM content after the NBF file content is burned to the actual device. Necessary to use when implementing CRC burn flow. |
| **Special NBF Files** | |
| ./nbf/check_userempty.nbf | NBF file that checks whether the **User** part of NVM is empty (0x00 values). Include before first burn to make sure that NVM is empty. |
| ./nbf/burn_usercrc.nbf | NBF file that calculates CRC over the **User** part of NVM and burns it into the Silicon Labs private production test area of NVM for possible Silicon Labs retest and full failure analysis. |
| ./nbf/check_pt_usercrc.nbf | NBF file that calculates CRC over the **User** part of NVM and compares it with the value stored in the Silicon Labs private production test area of NVM. Return failure if they do not match. Good for final checking of the NVM integrity. |
| **Special NBF Files with CRC Checking** | |
| ./nbf/check_usercrc.nbf | Calculate the CRC over the current content of the **User** part of NVM and check it against the known, expected value. Return failure if they do not match. |
| ./nbf/check_burn_usercrc.nbf | Calculate the CRC over the current content of the **User** part of NVM and check it against the known, expected value. Return failure if they do not match. If they match, burn the CRC value into the Silicon Labs private production test area of NVM. Note that if the calculated and expected CRC do not match, there is no CRC burning. |
| ./nbf/burn_check_usercrc.nbf | Same as **.\nbf\check_burn_usercrc.nbf** above, but the burn and check tasks are swapped. Calculate the CRC over the current content of the **User** part of NVM and burn the CRC value into the Silicon Labs private production test area of NVM. After the burning, compare the calculated and burned CRC with the expected one. Return failure if there is no match. |
| ./nbf/check_pt3way_usercrc.nbf | Final CRC check. Requires one of the **.\nbf*burn_*usercrc.nbf** files to be used before this one. Calculate the CRC over the current content of the **User** part of NVM and check it against the known, expected value. Return failure if they do not match. If there is a match, read the burned **User** CRC value stored in the Silicon Labs private production test area of NVM and compare it to the calculated CRC. Return failure if there is no match. |

**Table 1. NVM Data Composure and Burning Files (Continued)**

| File Name | Description |
|---|---|
| **Convenience NBF Files for Setting Device to Run State** | |
| `./nbf/burn_run.nbf` | Convenience NBF file that puts the part to the **Run** state. No other NVM content is modified. |
| `./nbf/burn_nvmram.nbf` | Convenience NBF file that puts the part to the **Run** state and sets the NVM and RAM protections. Note that MTP protection is not set. No other NVM content is modified. |
| `./nbf/burn_nvmrammtp.nbf` | Convenience NBF file that puts the part to the **Run** state and sets the NVM, RAM, and MTP protections. No other NVM content is modified. |

# TABLE OF CONTENTS

**Section** **Page**

# 3. Si4010 Boot Process

The user should consult the Si4010 data sheet for details about boot. For convenience, a brief description of boot process and NVM organization is included in this document.

The device does not include a Flash memory for permanent code or data storage. Instead, the device contains 4.5 kB of RAM, which serves as a unified CODE and XDATA RAM memory. The device contains 8 kB of NVM (OTP) memory for user code and configuration storage. A small part of the NVM is reserved for Silicon Labs factory use and is not available to a user. In general, more than 7.5 kB of NVM is available for user application use.

## 3.1. Startup Overview

The user application code cannot be run directly from NVM since it is not mapped directly to the CPU address space. Instead, upon device reset, the device goes through a boot process during which the factory device configuration and the user application code and data are copied from NVM to the CODE/XDATA RAM. Only after the boot process finishes does the user code begin to be executed from CODE/XDATA RAM address 0x0000.

After reset, the device does not execute the user code immediately, but only after the boot process finishes. The time between the device wakeup (either caused by cycling the power or waking up from the shutdown mode by button press) and the start of the user application execution depends on the size of the user code load.

## 3.2. Reset

Reset circuitry allows the device to be placed in a predefined default condition. There is only one external reset source for the device, which is power on reset. It is invoked under the following conditions:

1. Power is supplied to the device. This means connecting the power supply to the disconnected device.
2. The device is waking up from a shutdown mode. The power supply was connected before, but the device was put into the shutdown mode. When it is awakened, the power is supplied internally to all the device systems.

## 3.3. Chip Program Levels

The boot process starts by reading the NVM configuration bytes in the **Factory** region of NVM. The information about the programmed level of the chip is read first, and the boot process acts accordingly.

There are three program levels of the chip:

1. **Factory** .. empty part leaving the factory. The factory chip calibration is written into NVM. ROM and NVM **Factory** region is not readable by the user. Part can be used with debugging chain for software development and **User** load can be programmed to the part. Boot process initializes the part based on the **Factory** settings.
2. **User** .. same as **Factory** part, but with the **User** region programmed with user code. The boot process will initialize the part according to the **Factory** settings and then copy the **User** load to the CODE/XDATA or IRAM based on the **User** load. The code is not automatically run. The part can be used with IDE for further software development. The part is still opened for further NVM programming, and the user can add additional data to the **User** region in the NVM. Debugging of the code loaded from NVM is possible.

   The user can modify the boot behavior of the **User** part by controlling two bits described in the Si4010 specification document such that the **User** load is automatically run or that the **User** part does not load the actual user code and behaves as the**Trim** part.
3. **Run** .. mission mode part, fully programmed for use in the field. No further NVM programming is possible; no C2 interface access is enabled, with the exception of special mode for retest. No possibility of IDE debug. The boot process is the same as in the case of **User** part, but after the user load is copied from NVM to RAMs, the boot loader executes a jump to RAM address 0x0000, and the user application is executed. The C2 is not enabled in this mode with the retest exception (briefly described in this document).

The IDE debugging environment can only be used with the **Factory** and **User** program chip levels, not with the **Run** part.

To protect the user intellectual property (IP) stored in NVM, there are protection flags the user can choose to enable. See the Si4010 data sheet for detailed descriptions of those. By default, the protections are disabled. It is up to the user to enable the IP protections during the NBF burn file composure process.

## 3.4. Boot Routine Destination Address Space

The boot process reads the formatted data from NVM and writes it to the desired destination. The format supports different address regions based on the destination RAM address. The destination RAM address is part of the NVM data frame format. The RAM destination address space as used by the internals of the NVM image depends on the program level of the chip and is shown in Figure 1.

*Boot routine view of the CPU memory space for writing*
*User data from the NVM to the RAM/register spaces*

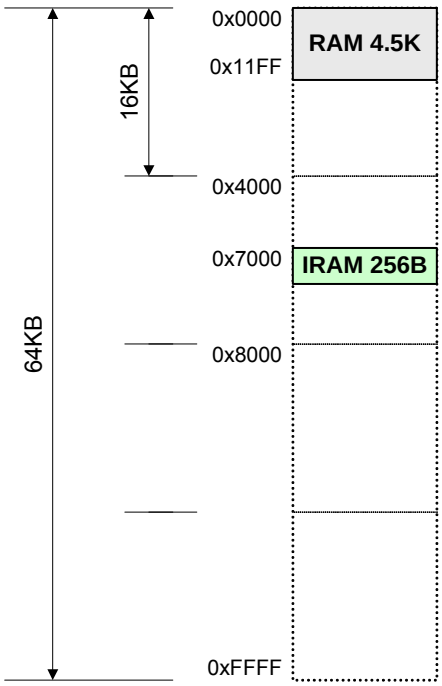


**Figure 1. Boot Routine Destination CPU Address Space for Copy from NVM**

- 0x0000 .. 0x11FF .. CODE/XDATA RAM. Note that the end portion of the RAM is reserved for the boot control data as described in the main data sheet and related application notes.
- 0x7000 .. 0x70FF .. virtually mapped 256 byte of IRAM for IDATA indirect access. Whenever the destination address in the NVM image is in this region, the data destination is going to be IDATA IRAM space. However, only region 0x7020 .. 0x70EF is writeable. That means that the first 32 and last 16 bytes of the IRAM are not writeable by a boot process nor by the **bNvm_CopyBlock()** or **bNvm_LoadBlock()** functions. The mapping is for indirect IDATA internal IRAM; so, SFR registers cannot be initialized by this process.

It is up to the user to generate IntelHEX or Verilog MEM files to be passed to the NVM programmer. The NVM programmer will ensure that the NVM gets programmed with the proper data structures such that the data values provided in the IntelHEX files appear at the RAM and IRAM addresses specified in the IntelHEX input file after the boot is done.

By using the unified CODE/XDATA memory and mapping the IRAM to the boot process address space, the user can initialize both XDATA and IRAM variables directly from the **User** NVM load without the need for running any startup code to perform variable initializations, resulting in the saving of code size.

One application of the data initialization by a boot process could be copying of keys from the NVM to fixed locations without any code intervention. The user can program all the chips with the same application in the factory and then add only a very small, chip-specific, **configuration** block with keys specifying where to the XDATA and/or IRAM memories the boot process should copy the values of the keys.

For example, to initialize IRAM location 0x56 to 0xA4 value, the user provides an IntelHEX file specifying that, at the RAM address 0x7056, the data value should be 0xA4.

## 4. NVM Organization

The 8 KB NVM (OTP) memory is virtually mapped to the device address space 0xE000 .. 0xFFFF. However, the CPU can access NVM only indirectly using the API function **bNvm_CopyBlock()** and a library function **bNvm_LoadBlock()**, which calls the former API function behind the scenes.

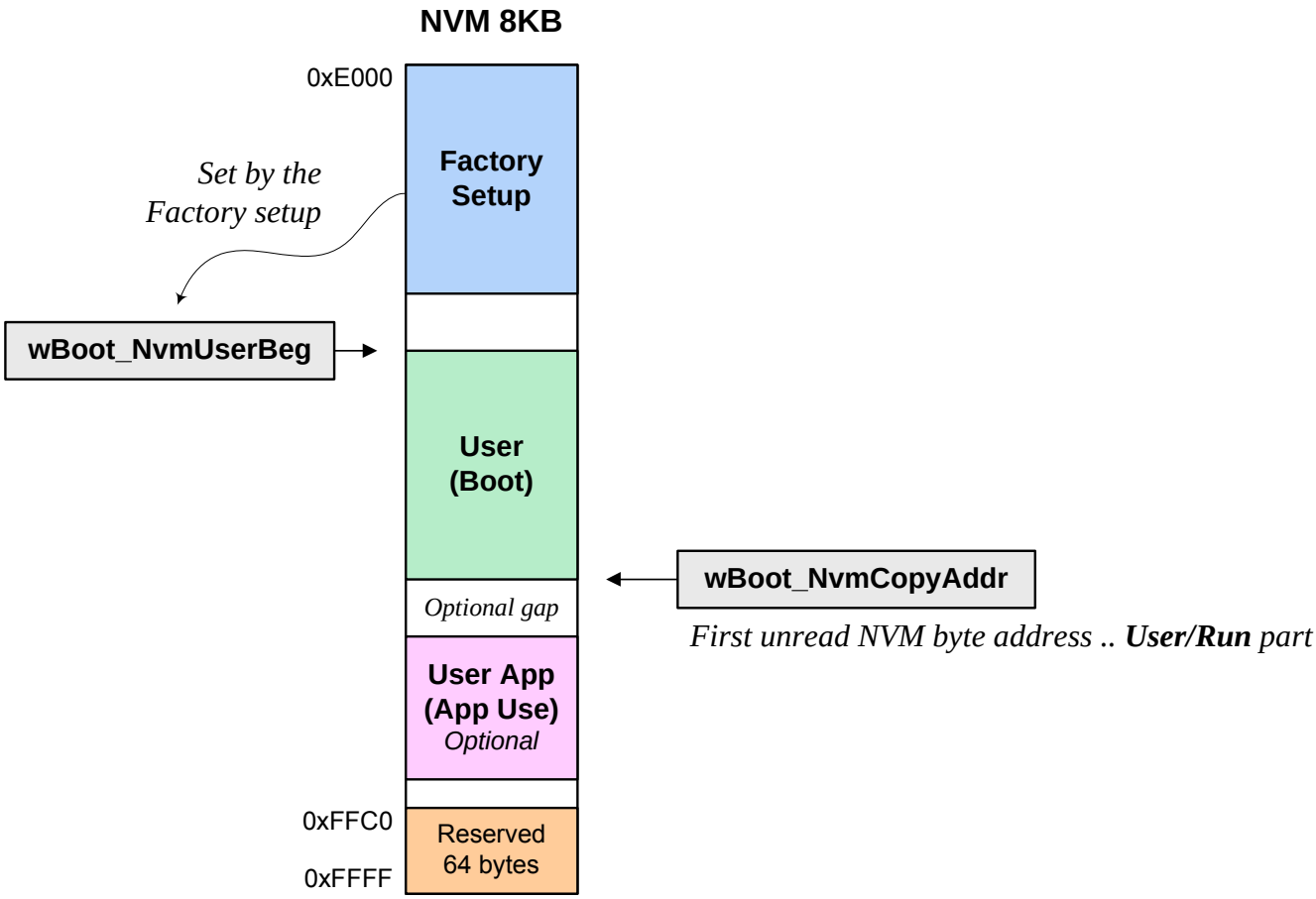


**Figure 2. NVM Address Map**

## 4.1. NVM Regions

The NVM address region is organized in the following fashion:

1. **Factory** region .. factory settings critical for chip functions. Variable size based on the device configuration.

2. **User** region .. region available for **User** application load at boot time. This region is sometimes referred to as **User Boot** for clarity. If the user application is not going to use overlays, then this will be the only user data region used. The region starts at **0xE180** NVM address.
3. **User App** optional region .. optional region not visible at boot time. If the user application is using overlays, then the overlay code will be stored in this region. It will be up to the user to load the application code from the NVM to CODE/XDATA RAM at runtime based on the user application request. An application note will be devoted to this technique.
4. **Reserved** region .. last 64 bytes of NVM are reserved for factory use and not available for user load. This is where the user CRC is stored if CRC burn flow is used.

The user content can occupy all NVM locations apart from **Factory** and **Reserved** regions.

Only the **User Boot** region is visible to the boot routine and will be loaded during boot. The user may decide that overlays will be used.

The **User App** region is the data region available to be loaded by the user application program at runtime. Boot routine does not load any data from that region. The user will have to call the API NVM copy routine **bNvm_CopyBlock()** or library function **bNvm_LoadBlock()** from the user application. If the user decides to use overlays, they have to be put in this space and loaded by the user application at runtime.

## 4.2. NVM Composed Data Organization

To be able to load user-specified application data in the IntelHEX format, the user data has to first be composed (converted) into the data structures that the boot process or data/overlay load functions **bNvm_CopyBlock()** and **bNvm_LoadBlock()** would understand. The structure of the data content in the NVM is shown in Figure 3:

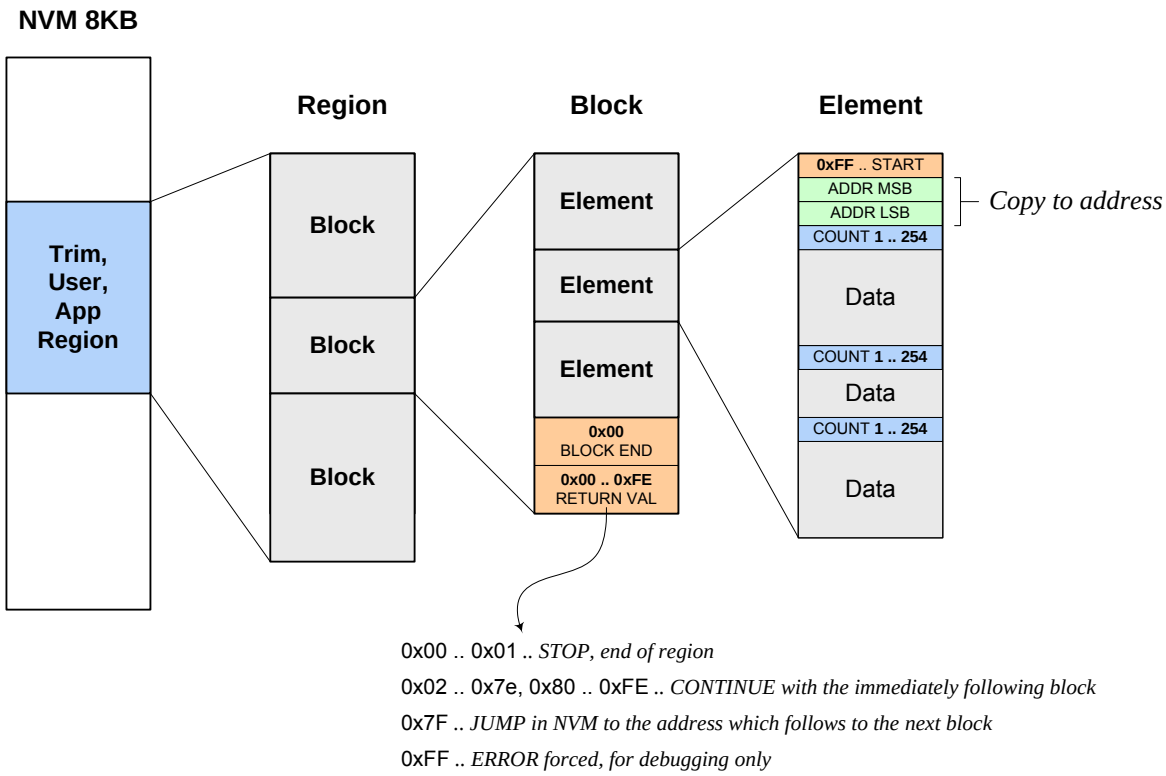


**Figure 3. Programmed NVM Region Frame Structure**

The NVM is organized in the following fashion:

- **Region** .. part of NVM that contains data to be copied to RAM. There is a valid **Factory** region only for devices in the **Factory** state and **Factory** and **User** regions for devices in the **User** and **Run** states.
- **Block** .. each region is a sequence of data blocks. Normally, the blocks must be organized back-to-back in the NVM. Block is the data structure copied by a single call to the **bNvm_CopyBlock()** or **bNvm_LoadBlock()** functions. If the region contains several blocks, each block is copied by a separate function call. The boot routine does this automatically during the boot process. At the end of the block, there is a return value byte. That byte is a return value of the function call when using overlays. The return value of **0xFF** is reserved for error status.
- **Element** .. each block is a sequence of elements. An element is a data structure for copying a continuous array of data bytes to the destination. The element starts with the 0xFF START element byte. This is followed by the destination address. After that, the data is split into arrays of lengths 1 to 254. The first byte in the first data field of the element gets copied to the copy to address, the second byte to the address+1, and so on. A new element is introduced if the destination address needs to change by more than +1. If the destination data regions to be copied are continuous, there is only a single element per up to 254 continuous data bytes.

The **bNvm_CopyBlock()** and **bNvm_LoadBlock()** functions copy one block. They return the **Block** return value specified at the end of the **Block** in NVM.

The NVM composer converts each **User App** user application IntelHEX or Verilog MEM file into its own **Block** with return value **0x01**. There is one **Block** per input IntelHEX or Verilog MEM file after conversion.

### 4.2.1. Gaps between NVM Blocks

Normally, the blocks should be organized to be placed in the NVM back-to-back with no gaps between them. However, there might be scenarios where gaps need to be introduced into NVM. The other scenario is that a user wants to program NVM with different **User** loads present in the NVM at the same time and then later decide to select the proper **User** load by programming only a few NVM bytes to select the proper one.

For those occasions, it is necessary to have an option not to have **Blocks** back-to-back in the NVM. To accommodate that, the block return value of **0x7F** is reserved and understood by the boot routine and **bNvm_CopyBlock()** and **bNvm_LoadBlock()** functions. When the **0x7F** return value is encountered at the end of the **Block**, the function will read two more bytes from NVM. Those two bytes represent the NVM address of the beginning of the new **Block**. It can be viewed as a “jump” to the beginning of the next **Block**.

The block structure for the **0x7F** return value case is shown in Figure 4.

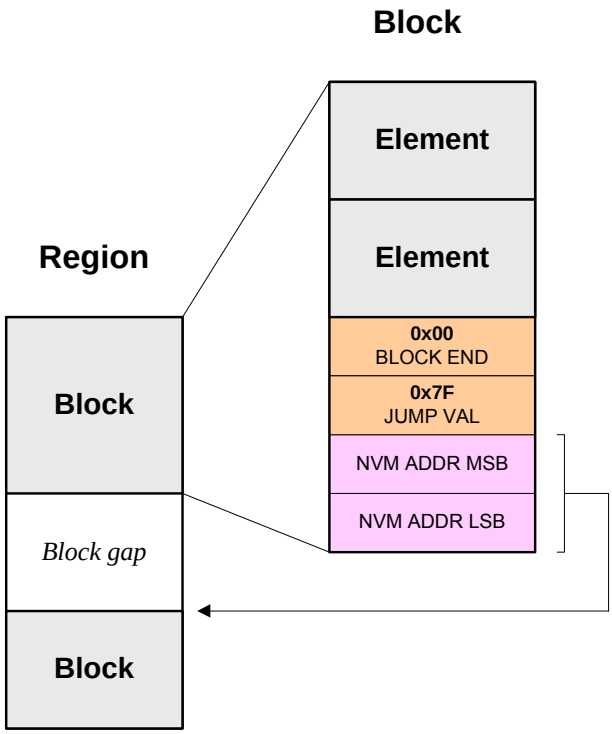


**Figure 4. Region Organization with Gaps between Blocks**

This jump is automatically generated by the **gui_composer** when the user has more than one IntelHEX file loaded by boot in the **User Boot** region and the user provides an NVM starting address for the second and subsequent files other than 0. The **gui_composer** generates one NVM **Block** per input file. It determines whether there is a gap in the NVM space between two composed **Blocks**. That can only happen if the user has supplied a specific NVM starting address for other than the first **Block**. If there is a gap, then the first NVM **Block** will end with the “jump” pointing to the NVM starting address of the next boot **Block**.

This can only happen for the **User Boot** region of the NVM. For the **User App** region, the composer composes one **Block** per input IntelHEX or Verilog MEM file with **Block** return value **0x01**.

### 4.2.2. Boot Routine NVM Copy Stop Condition

During boot, the boot routine copies the subsequent **User Boot** NVM **Blocks** as long as the **Block** return value is **0x02 .. 0xFE**. The boot will stop copying NVM **Blocks** once it encounters return values of **0x00** or **0x01**. If it encounters return value **0xFF**, it is interpreted as an error, and the boot process stops copying NVM and sets the boot fail flag (but it continues with the rest of the boot process). It is up to the user application to check the boot fail flag and determine whether to run the rest of the application. The boot fail flag is a bit 2 in BOOT_FLAGS register, mask 0x04.

# 5. NVM Composer (gui_composer.exe) and Burner

## 5.1. Overview

The peculiarity of the Si4010 chip is that the CPU does not run code directly from NVM. Upon boot, the boot routine copies the user code stored in the NVM to the RAM and runs it from there.

For the boot routine to understand the data, the user code in the form of IntelHEX or Verilog MEM input file has to be processed and “composed” to data structures the boot routine will understand, as shown in Figure 3. Therefore, the data has to be passed through NVM composer to prepare the boot data structures, which will be burned into the NVM.

The other task of the compose process is to create specialized code that will be downloaded to the device and burn the composed data into the NVM of the actual device. The output of the compose process is an NVM burn file (NBF) with ***.nbf** default extension, which contains all the information needed to program the device.

The NVM burn GUI is a graphic “shell” around the NVM composer command line executable.

The NVM composer command line executable name is **gui_composer.exe**; the names NVM composer and **gui_composer** refer to the same thing. It is important to know that running the command line composer does not require hardware connection of the PC to the Si4010 to be burnt.

## 5.2. Operation Modes

The NVM composer **gui_composer.exe** works in two mutually exclusive modes, which have corresponding tabs on the GUI:

1. **Main (Regular) Mode** .. this mode is a regular composer mode used by the user most of the time. It takes user input files in IntelHEX or Verilog MEM formats, **Block** NVM start addresses, and other flags and generates the proper NVM burn NBF file. The addresses in the input files are the addresses in the RAM where the user wants the data to be after the boot or user copy. In other words, the user supplies input files with RAM data locations where the data should be after boot or after data/overlay load. The composer will decide how that information maps into the NVM.
2. **Direct Burn Mode** .. advanced mode. It takes either a direct string of NVM address and data in the Verilog MEM file format, or the Verilog MEM file with direct NVM address and data specified. The input addresses and data are the actual direct NVM addresses and data. This mode is used only for advanced purposes, like burning a part-specific key during the two-step configuration programming process (described later in this document). IntelHEX data format cannot be used as an input for **Direct Burn** mode.

## 5.3. Burn Algorithms and Compose Mode

The unprogrammed (pristine) NVM bit has logic value 0. The programmed bit has logic value 1.

During the compose time, the user has to decide how the burn is going to be done. There are two burn algorithm modes that the composer can generate:

1. **Strict** .. during burn, the existing value of each NVM bit is checked immediately before it is going to be programmed. If 0 is to be programmed to the existing bit value of 1, the burner program returns error at runtime as “error on conflict” since it is not possible to program 0 into the existing bit of 1. The bit address of the conflict is also reported to the user. This is the bit address offset from the beginning of valid NVM addresses 0xE000. To get a byte address, divide the bit address by 8 and add 0xE000 to it.

   If the existing NVM bit value is 1 and the value to be programmed is also 1, then there is no action taken, and no error is reported. The burner will move to the next bit.
2. **Logic OR** .. there will be no bit conflict error during programming. The resulting new value of the NVM bit is the logic OR between the current value of the NVM bit and the new (to be programmed) value of the bit. If the bit is already programmed as 1 and the user desires to program it as 0, nothing is done to the bit, and there is no error. This mode allows programming of additional bits in already existing bytes without a need to know the current value of other bits in the byte. There is never a bit error conflict reported in this mode.

The user must decide at **compose** time what burner code, **Strict** or **Logic OR**, is going to be generated. For production code, it is highly recommended to use **Strict** mode.

## 5.4. Output File Format

The NVM composer, **gui_composer.exe**, generates a single output **NVM burn file** with a default ***.nbf** extension. The file consists of several sections and contains all the information needed to burn the data into NVM. The file can also be loaded to the NVM GUI or used by a command line burner at a later time and used for NVM burning without running the composer again. That is beneficial in situations where the user wants to burn the same NBF file data into many chips.

The NVM burn file from the composer is then loaded back into the NVM GUI or to a command line burner, parsed, and, when using the GUI, the **[Compose Map]** is displayed for user information. During the **Burn** process, the content of other sections of the file is used to actually program the NVM.

The structure of the NBF burn file is as follows:

```
>---- START OF FILE ----------------
; Comment starts with ';'. After ; everything till the end of the line
; is ignored. Empty lines are ignored.

; Command line of the gui_compose.exe which generated this file.
; This section is for information only and by cutting and pasting the line
; it is possible to recreated this file.
[Command]
gui_composer.exe --boot_hex=main_app.hex 0xe180 .....

; -- If there is an error in the gui_composer run the sections below
; will not be present in the output file.

; Composer output file map, 5 fields per line separated by spaces
;
;  File         | NVM range        | Length hex and dec | Conflict |
;  main_app.hex   0xE080    0xE08f   0x10   16            OK
;  data.hex       0xE090 .. 0xE0A0   0x11   17            OK

[Compose Map]
main_app.hex  0xE180  0xE18F  0x10  16  OK
data.hex      0xE190  0xE1A0  0x11  17  OK

; Start of the burn section. The Files=N specified the of the number
; of the embedded burn files to be sequentially downloaded and executed
; during the burn process itself.
; What follows are embedded IntelHEX burn files as section
; [File <file number>]
; starting with <file number> equal to 1.

[Burn]
Files=2

[File 1]
:038003000200195F
:10000000007581D0C281D281C282C28343800EE555
:1000100080E4C281C282028003C28343800E75A441
:100020000075A5007580FF7590FFC2855390FC7820
:0B0030000AD8FE43900343870180FEC6
:00000001FF

[File 2]
:0380000002AA00D1
:10900000E5D370FC53D4F8901100E0F5D2A3E0F55D
:10901000D175D302E5D370FCE5D6F460037FFF225F
:1090200043D40275D302E5D2B4FF16E5D1C394C090
:10903000400F74FF901100F0A3F0E5D370FC7FFFA8
:00000001FF

<---- END OF FILE ----------------
```

## 5.5. Input File Formats and Extensions

**Important: The composer requires that there be no space in the path or file name. There must not be any space in the file path or in the file name itself. Using quotes around the path name with spaces will not help.**

The composer understands two types of input files for both **User Boot** and **User App** regions. The file formats are distinguished by required file extensions:

1. IntelHEX .. required extension ***.hex**
   It is a standard, 16-bit address IntelHEX file.
2. Verilog MEM file .. required extension ***.mem**
   This file has a Verilog memory hexadecimal file format. The exception from the format is that the comment allowed in the file starts with "`//`", and whatever follows is a comment to the end of the line. Comments delimited by compound characters "`/* … */`" are not allowed.
   The file format is hexadecimal `@addr` destination address followed by one or more byte values `byte byte byte` on a single or multiple line with address being incremented with each byte until the next `@addr` is encountered. The hexadecimal values do not use the `0x` prefix or any other prefix. The address and byte data are separated by white spaces or end of lines. For example (the letter case does not matter, capital hex symbols are used in the example by choice):

```
@0003 15 A4 3E
7E 56 @0015 89 F5 CD
89
AB
@10F4
DF C7 A4
```

## 5.6. NVM Composer Process (gui_composer.exe)

The composer processes files according to the following general rules:

1. The composer processes each IntelHEX or Verilog MEM input file separately.
2. The composer process generates one **Block** per each input IntelHEX or Verilog MEM file specified.
3. The composer process loads each HEX/MEM file in its entirety into internal virtual memory and processes the memory separately for each file. It does not matter how fragmented the IntelHEX file is (there can be one byte per each IntelHEX record with addresses randomly spread over the whole file). The composer loads the IntelHEX into the virtual memory and processes the actual data at their "destination" addresses.
4. It will generate one **Element** for each continuous data block it can find in the input HEX/MEM file. By design, it generates as little NVM overhead as possible.

### 5.6.1. Processing of User Boot Input Files

The **User Boot** input files determine the user-specified load loaded to the chip at boot time if the chip state is **User** or **Run**. The **User Boot** load is the only user-specified NVM content of which the boot routine is aware.

The user can specify more than one input file to be loaded by boot. If there is more than one file, they are ordered in the GUI **User Boot** box. The file order matters, and they will be processed in the order listed. All the files listed in this box will be loaded by the boot routine at boot time.

If both IntelHEX (***.hex**) and Verilog MEM (***.mem**) file formats are mixed together, the composer processes all the IntelHEX files first in the order as they appear in the file box or on a command line, followed by all the Verilog MEM files in the order that they appear in the file input box or as they appear on the command line.

Only if both ***.hex** and ***.mem** files are present in the **User Boot** section, the first file in the file box or on the command line at address 0xE180 must be the ***.hex** file.

**NVM Addr**: Modify **NVM Address** for the selected file line. This is the address at which the file related converted **Block** will start in NVM. The first file address is not editable and is retrieved from the device and filled automatically. The value is **0xE180** and it is safe to say that it will never change. If the subsequent address value is set to 0 (default) then the converted **Blocks** will be stored in the NVM back to back without any gaps.

Each file in **User Boot** GUI box will be converted into a separate single **Block**. All **Blocks** are logically chained together so they are loaded by the boot routine in their entirety.

The user can choose to have gaps in between the converted boot **Blocks**. That is useful if some converted **Blocks** should be placed at specific NVM addresses to be possibly loaded by the application again at runtime. The composer will automatically introduce a gap in the NVM if the **Blocks** are not back to back. The boot routine will still boot all the **Blocks** in the **User Boot** GUI input box, even if there are gaps in between **Blocks**.

#### 5.6.2. Processing of User App Input Files

The **User App** input files must be loaded by the user application at runtime using **bNvm_CopyBlock()** or **bNvm_NvmLoad()** functions. Boot routine has no knowledge of those files in NVM. Those should be used for overlays, for example.

**NVM Addr**: Mandatory starting address of the file related **Block** in NVM. It must be supplied by user for all **User App** files.

Each file in the GUI **User App** box will be converted into a separate single **Block**. **Blocks** have no relation to each other.

Since each of the files require the **NVM Addr** to be specified, any mixture of IntelHEX and Verilog MEM files for **User App** region is allowed without any restriction.

## 5.7. Programming Algorithm

The burn algorithm is part of the NBF files. When the burn process is invoked from the NVM GUI by pressing **Burn** button or by using **Si4010_NVM_Burn_CL.exe** command line burner, the following steps are invoked:

1. The NBF file is parsed, and each **[File *N*]** section is processed sequentially and separately.
2. The **[File *N*]** is loaded into the device and executed. It contains the composed data and burn algorithm controls.
3. The programming algorithm burns the NVM bit by bit. For each bit, it first reads the current value of the bit from the NVM. Based on the current bit value, the desired programmed value, and the burner mode (**Strict** or **Logic OR**), it decides whether to burn the bit to 1, there is nothing to do (since the bit has the desired value already or will not be burned), or report an error-on-bit conflict.
   If the current bit value is 0 and the desired value is 1, then the bit programming sequence is invoked. At the end of the sequence, the programming algorithm makes one final check by reading the just-programmed bit from NVM and comparing it to the desired value of 1. If there is no match, it stops and reports an error. If there is a match, it moves to the next NVM bit address and the process repeats itself.
4. Once the **[File *N*]** is processed, the result is reported to the burner process on PC. If there is a success, then the next **[File *N+1*]** is loaded and processed.
5. The whole burn process stops once the first error is encountered. There might be some **[File *N*]** section unprocessed in that case.
6. Note that the **[File *N*]** section does not have to perform burning. It can perform some specialized checking tasks instead of burning. As long as it reports success or error status the same way as the programming algorithm, it will work correctly with both the GUI and command line burner application. This feature is used during CRC flow or MTP programming.

## 5.8. Programming Chip State as Run

To finalize the device programming, the user must set the part to the **Run** state by checking the **Run** box on the GUI or using `--state=run` on a **gui_composer** command line. Once the **Run** state is set and power is cycled to the device, the following is true:

1. The device boots, copies the user application, and runs it automatically.
2. All the user intellectual property protections (**NVM Disable**, **MTP Clear**, **RAM Disable**, and **C2 Disable**) are honored. Note that, if **C2 Disable** is set, the device is locked for good and Silicon Labs will not be able to do any failure analysis on a programmed chip at all. The other three flags provide complete protection of user intellectual property while allowing Silicon Labs to do retest and failure analysis on a device. It is recommended to never set the **C2 Disable** flag.
3. The NVM is write protected and cannot be written to anymore. The user should be aware that there is an advanced programming mode in which the user might decide to leave NVM programmable even in **Run** mode. This might be good for some specialized burn and debug scenarios. Contact Silicon Labs for details.

Even though the GUI does not allow setting of the user intellectual property flags while not setting the chip state to **Run**, the user can set those flags any time while using the **gui_composer.exe** directly on a command line by using `--bf_...` options. The flags will be ignored if the chip state is not yet in the **Run** state. Only when the chip is in the **Run** state are those protection flags honored.

**Note:** When changing the chip state to **Run**, the actual change will take effect only after the power is cycled to the device. As long as the power is not cycled, programming of the device can continue.

This fact has significant consequences for device programming flow using single or multiple NBF files. Note that the command line utility **Si4010_NVM_Burn_CL.exe** and associated **burn_cl.bat** files will burn only a single NBF file and will turn the power off immediately after burning unless the user provides some external power to the device.

**Note:** If the user desires to burn multiple NBF files into the same device, the **Run** state must be set in the very last NBF file.

To overcome this limitation, the user may choose to concatenate all the NBF files to be burned into a single NBF file by using the **nbfmod** utility described later. In that case, the concatenated NBF file is burned in a single session without power cycling, and, therefore, setting the **Run** state can happen anywhere in the file.

Setting the **Run** state early in the burn process has a limitation in that Silicon Labs will not be able to do a full failure analysis on a device that fails later during programming.

### 5.8.1. Programming Chip State as Run to Allow Full Retest

If the user concatenates several NBF files with the **nbfmod** utility and the **Run** state is set early in one of them, then Silicon Labs will not be able to do full NVM failure analysis on a device that failed during programming. This is because if the **Run** state is set early and the NVM programming fails later in the flow, the **Run** state and all the associated user intellectual property protection bits will take effect after the power cycle, disabling any NVM access other than the CRC calculation on which Silicon Labs performs a failure analysis.

**Note:** To enable full Silicon Labs retest of the part that fails during programming, setting the chip to the **Run** state must be done in the very last NBF file.

# 6. Si4010 NVM Programming Utility

## 6.1. Overview

This section provides a brief overview of the Si4010 Programming Utility. For more information, refer to AN511.

The NVM burn GUI **Si4010_NVM_Burner.exe** consists of two functional parts:

1. GUI shell around the NVM composer **gui_composer.exe**. It collects user input, forms a **gui_composer.exe** command line and executes the code when the user presses the **Compose** button to generate an NBF file.
2. It includes an NBF file reader and burner, which reads the NBF file, processes it and uses the information in the NBF file to communicate with the device to do the actual NVM burning. The standalone version of the same code is **Si4010_NVM_Burn_CL.exe** command line burner.

The user may decide not to use the NVM GUI and use the command line alternative instead. The user will use the following command line tools:

1. **gui_composer.exe** to compose the NBF burn file.
2. **Si4010_NVM_Burn_CL.exe** or its wrapper **burn_cl.bat** to achieve burning of the NBF from a command line or other user program.

## 6.2. Operation Flow Using GUI (Si4010_NVM_Burner.exe)

The NVM GUI **Si4010_NVM_Burner.exe** operation flow is as follows:

1. Select the **Main** tab on the GUI.
2. Select the **USB adapter**.
3. Hit the **Connect** button to connect to the part.

Either:

4. Select input IntelHEX or Verilog MEM files are inputs to the **User Boot** or **User App** (**Overlay)** section. The first file listed in **User Boot** section must be IntelHEX file. It cannot be Verilog MEM file.
5. Specify NVM start addresses in **User App** (**Overlay)** section, if there are any files. If trial run to determine the sizes of the overlays keep addresses as 0.
6. Specify NVM start **addresses** in **User Boot** section. That is rarely, if ever, needed. Keep all the address values as they are — first line has 0xE180 address *filled automatically*, subsequent addresses as 0x0.
7. Specify the new output **NVM Burn File**. Choose **Overwrite** if desired. GUI must check the file existence before running the composer since the composer always overwrites the existing output file.
8. Hit **Compose** and observe the results. The **gui_composer.exe** is invoked behind the scenes and the NBF file gets generated. If there are no errors or conflicts, one can proceed to burn at Step 10. If there are errors, then see the **Compose Log**, change the inputs, and hit **Compose** again until there are no errors. Then go to Step 10 if you want to burn or you are done and should stop here if you just want to compose NBF file for future use.

Or:

4. **Load** existing, previously generated, **NVM Burn File**. The **Compose Map** gets filled in from the file. Then go to Step 9.

In both cases:

9. Make sure the 6.5 V is connected to the GPIO[0] of the part. For example, slide the PROG switch on the MSC-BA4 board to the ON position.
10. Hit **Burn** to burn the device. The burning process loads the NBF file and uses the information in the NBF file to do the actual burning. Observe the results. The burning process stops at first error encountered.

# 7. Using Burn Flow with Checks and CRC (nbfmod)

To simplify the description, this section assumes that all the user parts have the same NVM content and the user desires to generate a burn flow with NVM empty checking before burn and burn data integrity checking using CRC.

## 7.1. Simple CRC Flow

Let's assume that the user compiled and linked the application into a single **app.hex** file. Then by using the NVM GUI or command line **gui_composer** the user generated the final NBF file for burning the application into the NVM, setting the device state as **Run**, and fully protecting the user intellectual property. Users should modify the IP setting used in the examples in this document to their particular needs:

```
gui_composer.exe \
   --boot_hex=app.hex 0xE180 \
   --nvm_burn_file=app.nbf \
   --state=run \
   --bf_nvm_dis \
   --bf_ram_clr \
   --bf_mtp_dis \
   --mode_strict
```

Then the user can generate many parts with the same content by using the command line burner, using, for example, the following command line:

```
burn_cl.bat app.nbf
```

However, the user may desire that the following checks are included during the programming flow to guarantee the NVM integrity of fully programmed device:

1. Before programming any bit check that the **User** part of the NVM is truly pristine (all bits at 0) and there are no accidentally programmed bits.
2. Program the user application using **app.nbf**
3. Check that everything was burned correctly by calculating CRC value over the whole **User** part of NVM and comparing that with the expected CRC value.
4. Burn the CRC into the Silicon Labs production test section of the NVM for possible future failure analysis.
5. Check the CRC again to make sure that noting happened in the NVM when burning burn of the CRC. This step is optional but encouraged.

To achieve this flow Silicon Labs provides specialized NBF files in the **.\nbf** directory along with the NBF concatenation and modifying script **nbfmod**.

To generate a single NBF file for burning which would implement the flow above, use the following command line, assuming running from Windows command prompt **cmd.exe**:

```
.\nbf\nbfmod.bat  \
   check_userempty \
   app.nbf \
   check_burn_usercrc \
   check_pt3way_usercrc \
   --output app_crcflow.nbf  \
   --autocrc \
   --verbose \
   --overwrite
```

The **nbfmod** script does the following:

1. It loads all the specified NBF files and concatenates their **[File *N*]** sections into its internal data structures. If only a name without a path and extension is specified then the ***.nbf** extension is added and the tool tries to locate the file in the same directory the **nbfmod** script resides in.
2. It simulates the programming (burning) process using the concatenated NBF data to create the same NVM image as the actual programming process is going to create on a real device.
3. It then calculates 32 bit CRC over the **User** part of the NVM. The calculated CRC gets reported to the screen if `--verbose` option is used. The user can output the CRC to a file by using `--crcout` option.
4. If `--autocrc` option was used then it replaces the CRC all zero value with the calculated one in all ***check_*usercrc** NBF files loaded. This will be the expected CRC value against which the actual chip calculated CRC will be compared during the programming process.
5. It generates a new NBF file with the **[File *N*]** sections concatenated in the order in which the original NBF files appear on the **nbfmod** command line.

Description of the **nbfmod** command line above and matching it to the desired flow:

1. Before programming anything, check that the **User** part of the NVM is truly pristine (all bits at 0) and there are no accidentally programmed bits.

Achieved by:

```
check_userempty
```

It loads **.\nbf\check_userempty.nbf** which runs the NVM empty check on the **User** section of the NVM.

2. Program the user application using **app.nbf**

Achieved by:

**app.nbf**

It loads the previously user generated **app.nbf** file which burns all the user application data and sets device to the **Run** state. See desired modification of application NBF file generation later.

3. Check that everything was burned correctly by calculating the CRC value over the whole **User** part of NVM and comparing it with the expected CRC value.
4. Burn the CRC into the Silicon Labs production test section of the NVM for possible future failure analysis. Both items 3. and 4. are achieved by :

```
check_burn_usercrc
```

It loads **.\nbf\check_burn_usercrc.nbf** which will calculate the CRC over the current content of the **User** section of the NVM during programming. It will then compare it with the expected CRC value (user must use `--autocrc` option). If there is no match, it will return immediately as failure during actual programming. It will then check whether the production test area reserved for **User** CRC (4 bytes) is pristine (0x00000000). If there is some previously burned **User** CRC there, it will return with failure. If not, then it will burn the **User** CRC into the Silicon Labs production test area of the NVM.

5. Check the CRC again to make sure that nothing happened in the NVM when burning burn of the CRC.

Achieved by :

```
check_pt3way_usercrc
```

It loads **.\nbf\check_pt3way_usercrc.nbf** which will not burn anything. This is a final check after all burning is done. It will calculate the CRC over the current content of the **User** section of the NVM. It will then compare it with the expected **User** CRC value. If there is no match, it will return as failure. If there is a match it will read the production test NVM area containing previously burned **User** CRC value. It will compare the just calculated CRC with the previously burned into the production test area of NVM. If there is not a match, return failure.

6. The remaining command line options specify the output concatenated NBF file:

```
--output app_crcflow.nbf
```

automatic CRC calculation and expected value replacement:

```
--autocrc
```

verbose mode with messages going to STDERR:

```
--verbose
```

and that the output file can be overwritten if it does exist:

```
--overwrite
```

Use the newly generated NBF file for programming of the devices:

```
burn_cl.bat app_crcflow.nbf
```

## 7.2. Recommended CRC Flow

The simple CRC flow sets the device **Run** state early in the **app.nbf** file. As described in "5.8.1. Programming Chip State as Run to Allow Full Retest" on page 18, this will not allow for full Silicon Labs retest of NVM when any of the subsequent NBF files in the concatenation fails, namely **check_burn_usercrc** and **check_pt3way_usercrc** ones.

To overcome that, two changes need to be made to the simple CRC flow above:

1. Generate a new **app_norun.nbf** file without changing the device state to **Run**.
2. Add an additional NBF file just to change the device state to **Run** at the end of the programming flow by adding **burn_run** in the NBF concatenation.

The new, retest-friendly, compose and concatenation CRC flow steps are:

Compose step does not have the `--state=run` option used, but sets the user intellectual property protections as part of the application NBF file. They will not take effect until the device is in **Run** state:

```
gui_composer.exe \
   --boot_hex=app.hex 0xE180 \
   --nvm_burn_file=app_norun.nbf \
   --bf_nvm_dis \
   --bf_ram_clr \
   --bf_mtp_dis \
   --mode_strict
```

Concatenation step uses the new **app_norun.nbf** file and adds **burn_run**. The output NBF name has changed as well:

```
.\nbf\nbfmod.bat \
   check_userempty \
   app_norun.nbf \
   check_burn_usercrc \
   burn_run \
   check_pt3way_usercrc \
   --output app_crcflow.nbf \
   --autocrc \
   --verbose \
   --overwrite
```

Burn the newly generated NBF file:

```
burn_cl.bat app_crcflow.nbf
```

Note that the **burn_run** was not added at the very end, but before the final check **check_pt3way_usercrc**. Setting the **Run** state should be the final *burn* NBF. The final check **check_pt3way_usercrc** does not burn anything, while the **burn_run** programs the **Run** state into the NVM. For the user to make sure that even the programming of the **Run** state did not accidentally alter any other part of the NVM, the final check should go after the last NBF which does actual burning, which is the **boot_run**.

**Note:** The retest-friendly CRC flow is the recommended programming flow with maximum NVM programming integrity checking, while keeping Silicon Labs retest and full failure analysis ability when NVM programming fails.

# 8. Viewing and Debugging NVM Content (nvmrev)

While the NVM organization is fully described, the user has no means to actually read the NVM directly. To be able to see how the NVM content is going to look like after the NBF file or files are programmed into the NVM, the **nbfmod** script simulates programming process to create the programmed NVM image and has means to output the actual NVM content after burning in either IntelHEX or Verilog MEM formats.

Then the **nvmrev** script can be used to simulate a boot process or functionality of the **bNvm_CopyBlock()** and **bNvm_LoadBlock()** functions when loading overlays to generate the device RAM content after the boot or copy function call.

After that the user can use the **hexdiff** content comparison script to compare the original IntelHEX files with the RAM content loaded by boot after the NVM is burned, completing the full circle.

## 8.1. Generating Programmed NVM Content for Debugging (nbfmod)

The **nbfmod** script simulates burning of all the **User** parts of NVM, chip state, and the user intellectual property control bits. While used in `--verbose` mode the actual state of the chip with all the flags is listed for the user to see and check whether those are expected. The inputs are the same NBF files as for the generation of the concatenated NBF file, or the final NBF file itself. The script will also report possible bit conflicts during the actual burn process.

Use `--hexout` or just the `--hex` option to output the NVM content in IntelHEX format and `--memout` or just `--mem` for Verilog MEM format.

Let's assume the recommended retest-friendly CRC flow was used, starting with

**app.hex**

file, composing application to

**app_norun.nbf**

file and then concatenating other files to make the final

**app_crcflow.nbf**

NBF file.

To generate the NVM image when the **app_crcflow.nbf** is burned to the device:

```
.\nbf\nbfmod.bat  \
   app_crcflow.nbf \
   --output app_crcflow.nvm.hex \
   --hexout \
   --verbose \
   --overwrite
```

To generate the Verilog MEM format, which is more human readable, use the following:

```
.\nbf\nbfmod.bat \
   app_crcflow.nbf \
   --output app_crcflow.nvm.mem \
   --memout \
   --verbose \
   --overwrite
```

The verbose output log on screen when running the latter would look like this:

```
INFO: nbfmod: 1.10 | November 24, 2011 | SiLabs.com
READ: Reading NBF (app_crcflow.nbf)
READ:   [File 1]
READ:   [File 2]
READ:   [File 3]
READ:   [File 4]
READ:   [File 5]

PROC: Processing NBF (app_crcflow.nbf)
PROC:   [File 1]
PROC:   [File 2] .. BURN (Strict)
PROC:     NVM: 0xE180
PROC:   [File 3]
PROC:   [File 4] .. BURN (Strict)
PROC:   [File 5]

CRC:  0xC33580A0

STAT: chip_state: Run
STAT: bf_xo_early_ena:  0 | bf_nvm_dis: 1
STAT: bf_exe_user_boot: 0 | bf_mtp_dis: 1
STAT:                     | bf_ram_clr: 1
STAT:                     | bf_c2_dis:  0

OUT:  Burnt NVM in MEM
OUT:  Writing (app_crcflow.nvm.mem)
INFO: Success
```

The user can visually check that the device state and all the protection flags were set correctly.

## 8.2. Simulating Programmed NVM Content (nvmrev)

It is possible to simulate the boot sequence or functionality of the **bNvm_CopyBlock()** and **bNvm_LoadBlock()** functions when loading overlay to generate a content of the device RAM after the boot or after the copy function call. That is achieved by using the **nvmrev** script.

The script takes an NVM burned image in the IntelHEX or Verilog MEM format along with either the `--boot` flag or `--addr <nvm_addr>` for loading overlays and generates RAM content after the boot or copy function call.

```
nvmrev.bat \
   app_crcflow.nvm.mem \
   --output app_crcflow.ram.hex \
   --boot \
   --verbose \
   --overwrite
```

The verbose mode output on the screen will show the boot process in detail and how the **Blocks** and **Elements** of NVM structures are processed:

```
INFO: nvmrev: 1.00 | November 07, 2011 | SiLabs.com
READ: Reading MEM (app_crcflow.nvm.mem)
INFO: Addr: <E000, FFFF>

PROC: Block   >            Start >  E180
PROC: Element | NVM:  RAM        |  E180:  0000
PROC:    Data | NVM:  RAM  Count |  E184:  0000  03 (3)
PROC: Element | NVM:  RAM        |  E187:  0400
PROC:    Data | NVM:  RAM  Count |  E18B:  0400  90 (144)
PROC: Block < | NVM:  NVM  Ret   |  E21B:  E21D  01 (1) <-- Ret

OUT:  Writing (app_crcflow.ram.hex)
INFO: Success
```

The Perl script tools automatically recognize whether the file inputs are in IntelHEX or Verilog MEM formats based on each file content.

The boot loaded RAM content should match the original **app.hex** exactly in this case. To make the comparison run the **hexdiff** script, which can take both IntelHEX and Verilog MEM file formats as inputs:

```
hexdiff.bat \
   app_crcflow.ram.hex \
   app.hex \
   --verbose
```

The result should be a complete match:

```
INFO: <-- Left
INFO: Reading HEX (app_crcflow.ram.hex)
INFO: Left  addr: <0000, 048F>
INFO: --> Right
INFO: Reading HEX (app.hex)
INFO: Right addr: <0000, 048F>
INFO: Include: 0000 .. FFFF
DIFF: OK: Files are identical
```

Note that this is not a textual diff, but a functional diff. Internally, files are loaded to virtual left and right memories and the content of those memories is compared byte by byte.

# 9. Programming Cases

There are two major cases to program:

1. All the parts are the same, no per-part configuration.
2. Each part requires the main **application** code, which is the same for all parts, with an additional block containing per-part **configuration**. This document uses the term **configuration**, while some users might use the term **serialization**.

## 9.1. All Parts Have the Same NVM Content

It is a straightforward case; using the provided tools the NVM burning can be tailored for either prototyping or for mass production.

Recommended flow of operations:

Do once:

1. Compile and link the application to get the hex file for the **application**.
2. Use the NVM burner GUI to create an NBF burn file.
3. If using the CRC flow process the composer-generated NBF file to generate final NBF file.

Do many times, one per part generated.

4. Burn the final NBF file using the NVM burner GUI or command line NVM burner utility.

The explanation of the CRC flow above assumed this scenario and therefore the case is described in detail in the previous sections.

## 9.2. Each Part Has Unique Configuration

In this scenario the part contains the main **application** code, which is the same for all parts, with an additional block containing per-part **configuration**.

The next section is dedicated to different configuration scenarios and solutions.

# 10. Configuration Loading

This section addresses the situation when each part contains the main **application** code, which is the same for all parts, with an additional block containing per-part **configuration**.

The term **configuration** means that each device will have unique, fixed data stored in NVM. That could be a serial number, AES keys, some other unique identifiers, etc.

For uniformity we will assume that the **configuration** is an array of bytes of customer selected size from 1 to 254 bytes, which gets copied in its entirety to a user-specified location, **configuration array RAM address**, into the XDATA/CODE RAM (4.5 KB) or the CPU internal IDATA RAM (256 bytes).

To copy data to IDATA CPU internal 256 byte RAM the user must specify addresses in IDATA RAM with added base offset of **0x7000** in the configuration IntelHEX file as a destination address. The IDATA RAM is virtually mapped to 0x7000 .. 0x70FF address space for boot and NVM copy purposes.

The user application has to declare a byte array of the appropriate size, which should be placed at a fixed address in XDATA or IDATA RAM. The user needs to select such **configuration array RAM address** and hardcode it into the **application**.

There are two main approaches to per-part NVM configuration:

1. The customer will load the **configuration** block from NVM by the application at runtime. At the beginning of the **application** there will be a call to the **bNvm_CopyBlock()** or **bNvm_LoadBlock()** function to read the configuration block from the NVM. It will cost about 20 bytes of customer's code space.
2. The customer wants the **configuration** to be loaded by boot at boot time automatically along with the main **application** so when the application starts, all the configuration data will be in RAM.

No matter what approach the customer chooses, there are two approaches for how to burn **configuration + application** sections into the part:

A. **Single NBF file**:
For each part (device) always start with part-specific **configuration** IntelHEX or Verilog MEM file and common **application** IntelHEX file. Generate a single new **configuration + application** NBF file for each part. Then use this file to burn the part. This is the easiest way to do it for any of the user approaches above. Only single burn is needed.

B. **Two NBF files**:
From the common **application** HEX file prepare common **application** NBF file. This will allow the burning of the preprogrammed "blank" parts. For each **configuration** IntelHEX/Verilog MEM file create a **configuration** specific NBF file and burn it later. Two separate burns are required, one for creating preprogrammed "blank", the other to burn the specific configuration to the preprogrammed "blank" part.

All 4 approaches can be easily scripted using command line versions of the **gui_composer** and **Si4010_Burn_CL** command line burn utility. The CRC flow can be used for all of the scenarios.

## 10.1. Notation

For the explanation let's assume the following:

1. The user **application** IntelHEX file name is **app.hex**. It is fixed, created once, and is the same for all parts.
2. The **configuration** file, unique per part, is named **config_part_1.hex** or **config_part_1.mem**, depending on the file format used. The letter "1" indicates that it is going to change from part to part and must be generated separately for each part.
3. Some flows require the **configuration** HEX file with all configuration data to be 0x00. The name of the file is **config_zero.hex** or **config_zero.mem**, depending on the file format used. It is generated only once.

## 10.2. Loading Configuration by User Application at Runtime

There are two cases: **1A** for single NBF file and **1B** for creating preprogrammed “blanks” with two NBF files.

### 10.2.1. Case 1A: Configuration Loaded by Application, Single NBF File

**Case 1A:** Boot loads the **application**, application loads the **configuration** at runtime, single NBF file.

#### 10.2.1.1. Regular Non-CRC Flow (1A)

Summary:

- Put **application** **app.hex** into the **User Boot** area in NVM
- Put configuration **config_part_1.hex** into the **User App** area in NVM at the address somewhere behind the main application. The NVM address needs to be hardcoded in the application for it to know where from NVM to load the configuration. It might be easier to generate the per-part configuration file in Verilog MEM format, **config_part_1.mem**, since it has a free format without strict IntelHEX format requirements.
- Run **gui_composer** to generate NBF burn file **app_config.nbf**.

For ***each part***, perform the following three steps:

1. Generate per-part specific configuration **config_part_1.mem** (or **config_part_1.hex**).
2. Run composer to generate NBF burn file:

```
gui_composer.exe \
   --boot_hex=app.hex 0xE180 \
   --app_mem=config_part_1.mem 0xF140 \
   --nvm_burn_file=app_config.nbf \
   --state=run \
   --bf_nvm_dis \
   --bf_ram_clr \
   --bf_mtp_dis \
   --mode_strict
```

The ***example*** shows the **configuration** data to be placed at NVM address **0xF140**. It is up to the user to choose the proper NVM address beyond the **app.hex** ending in NVM, which is also hardcoded in the application in the **bNvm_CopyBlock()** or **bNvm_LoadBlock()** function call.

3. Burn the part-specific **app_config.nbf** to the part:

```
burn_cl.bat app_config.nbf
```

#### 10.2.1.2. CRC Flow (1A)

Summary:

- Put **application** **app.hex** into the **User Boot** area in NVM
- Put configuration **config_part_1.hex** into the **User App** area in NVM at the address somewhere behind the main application. The NVM address needs to be hardcoded in the application for it to know where from NVM to load the configuration. It might be easier to generate the per part configuration file in Verilog MEM format, **config_part_1.mem**, since it has a free format without strict IntelHEX format requirements.
- Run **gui_composer** to generate NBF burn file **app_config_norun.nbf**.
- Run **nbfmod** to generate **app_config_crcflow.nbf**.

For ***each part***, perform the following steps:

1. Generate per-part specific configuration **config_part_1.mem** (or **config_part_1.hex**).
2. Run composer to generate NBF burn file:

```
gui_composer.exe \
   --boot_hex=app.hex 0xE180 \
   --app_mem=config_part_1.mem 0xF140 \
   --nvm_burn_file=app_config_norun.nbf \
   --bf_nvm_dis \
   --bf_ram_clr \
   --bf_mtp_dis \
   --mode_strict
```

The ***example*** shows the **configuration** data to be placed at NVM address **0xF140**. It is up to the user to choose the proper NVM address beyond the **app.hex** ending in NVM, which is also hardcoded in the application in the **bNvm_CopyBlock()** or **bNvm_LoadBlock()** function call.

3. Run **nbfmod** to generate **app_config_crcflow.nbf**:

```
.\nbf\nbfmod.bat \
   check_userempty \
   app_config_norun.nbf \
   check_burn_usercrc \
   burn_run \
   check_pt3way_usercrc \
   --output app_config_crcflow.nbf \
   --autocrc \
   --verbose \
   --overwrite
```

4. Burn the part-specific **app_config_crcflow.nbf** to the part:

```
burn_cl.bat app_config_crcflow.nbf
```

### 10.2.2. Case 1B: Configuration Loaded by Application, Two NBF Files

**Case 1B:** Boot loads the **application**, application loads the **configuration** at runtime, two NBF files. This flow is truly useful for creating preprogrammed “blanks” and adding the configuration by separate burn. If that is not the case, use the case **1A** flow instead.

#### 10.2.2.1. Regular Non-CRC Flow (1B)

Summary:

- Put **application** **app.hex** into the **User Boot** area in NVM.
- Run **gui_composer** to generate **app_norun.nbf** file for creating preprogrammed “blanks”.
- Put actual part configuration **config_part_1.hex** (or **config_part_1.mem**) into the **User App** area in NVM at the address somewhere behind the main application. The address needs to be hardcoded in the application for it to know where from NVM to load the configuration.
- Run **gui_composer** to generate configuration NBF configuration burn file **config_part.nbf**.

Flow steps:

Run just ***once*** to generate files for preprogrammed “blanks”:

1. Run composer to generate NBF burn file for preprogrammed “blanks”. There must not be any chip state set:

```
gui_composer.exe \
   --boot_hex=app.hex 0xE180 \
   --nvm_burn_file=app_norun.nbf \
   --mode_strict
```

For ***each part create a preprogrammed blank*** part by burning **app_norun.nbf** file:

```
burn_cl.bat app_norun.nbf
```

For ***each preprogrammed blank part*** do the following 3 steps:

1. Generate per-part specific configuration **config_part_1.mem** (or **config_part_1.hex**). The MEM file is used as input in this example.
2. Run composer to generate configuration NBF burn file with full **Run** state setting and user protection flags:

```
gui_composer.exe \
   --app_mem=config_part_1.mem 0xF140 \
   --nvm_burn_file=config_part.nbf \
   --state=run \
   --bf_nvm_dis \
   --bf_ram_clr \
   --bf_mtp_dis \
   --mode_strict
```

The ***example*** shows the **configuration** data to be placed at NVM address **0xF140**. It is up to the user to choose the proper NVM address beyond the app.hex ending in NVM, which is also hardcoded in the application in the **bNvm_CopyBlock()** or **bNvm_LoadBlock()** function call.

Note that setting the chip state to **Run** and the user protection flag setting is done during the configuration NBF file creation.

3. Burn the part-specific **config_part.nbf** to the part:

```
burn_cl.bat config_part.nbf
```

#### 10.2.2.2. CRC Flow (1B)

Summary:

- Put **application** **app.hex** into the **User Boot** area in NVM
- Run **gui_composer** to generate **app_norun.nbf** file for creating preprogrammed “blanks”.
- Run **nbfmod** to generate **app_norun_crcflow.nbf**
- Run **nbfmod** again to generate preprogrammed “blank” part NVM content **app.nvm.hex** file to be used later in the flow.
- Put actual part **configuration** **config_part_1.hex** (or **config_part_1.mem**) into the **User App** area in NVM at the address behind the main application. The address needs to be hardcoded in the application for it to know where from NVM to load the configuration.
- Run **gui_composer** to generate configuration NBF burn file **config_part.nbf**.
- Run **nbfmod** to generate **config_part_crcflow.nbf** file. This step uses the NVM content **app.nvm.hex** file generated previously.

CRC flow steps:

Run just ***once*** to generate files for preprogrammed “blanks”:

1. Run composer to generate NBF burn file for preprogrammed “blanks”. There must not be any chip state set:

```
gui_composer.exe \
   --boot_hex=app.hex 0xE180 \
   --nvm_burn_file=app_norun.nbf \
   --mode_strict
```

2. Run **nbfmod** to generate **app_norun_crcflow.nbf**:

```
.\nbf\nbfmod.bat  \
   check_userempty \
   app_norun.nbf \
   check_usercrc \
   --output app_norun_crcflow.nbf \
   --autocrc \
   --verbose \
   --overwrite
```

Note that only the NVM empty check and the user CRC check (not burn!) can be done, since the preprogrammed “blank” creating burn will not be the final burn.

3. Run **nbfmod** again to generate preprogrammed “blank” part NVM content **app.nvm.hex** file:

```
.\nbf\nbfmod.bat  \
   app_norun_crcflow.nbf \
   --output app.nvm.hex \
   --hexout \
   --verbose \
   --overwrite
```

The file **app.nvm.hex** file is needed for the final CRC calculation and checks during the configuration burn. Save the file for future use.

For ***each part create a preprogrammed blank*** part by burning **app_norun_crcflow.nbf** file:

```
burn_cl.bat app_norun_crcflow.nbf
```

For ***each preprogrammed blank part*** do the following steps to burn the configuration:

1. Generate per-part specific **configuration** **config_part_1.mem** (or **config_part_1.hex**).
2. Run **gui_composer** to generate configuration NBF burn file:

```
gui_composer.exe \
   --app_mem=config_part_1.mem 0xF140 \
   --nvm_burn_file=config_part_norun.nbf \
   --bf_nvm_dis \
   --bf_ram_clr \
   --bf_mtp_dis \
   --mode_strict
```

The ***example*** shows the **configuration** data to be placed at NVM address **0xF140**. It is up to the user to choose the proper NVM address beyond the **app.hex** ending in NVM, which is also hardcoded in the application in the **bNvm_CopyBlock()** or **bNvm_LoadBlock()** function call.

Note that setting the chip state is still unchanged but the user protection flag setting is done during the configuration NBF file creation.

3. Run **nbfmod** to generate **config_part_crcflow.nbf** file. This step uses the NVM content **app.nvm.hex** file generated previously.

```
.\nbf\nbfmod.bat \
   --nvmload app.nvm.hex \
   config_part_norun.nbf \
   check_burn_usercrc \
   burn_run \
   check_pt3way_usercrc \
   --output config_part_crcflow.nbf \
   --autocrc \
   --verbose \
   --overwrite
```

The file **app.nvm.hex** contains the NVM image of the preprogrammed “blank” part, which is needed for the script to see the correct NVM content for the final CRC calculation and burning.

4. Burn the part-specific configuration **config_part_crcflow.nbf** to the part:

```
burn_cl.bat config_part_crcflow.nbf
```

## 10.3. Loading Configuration by Boot

When the **configuration** is loaded by the boot along with the **application**, the configuration has to be an integral part of the NVM content in the **User Boot** NVM area and the configuration section has to be composed together with the main application to create the NBF file. In the case of two NBF files for creating the “blank” part, the empty (all 0x00) configuration section has to be incorporated into the first NBF file, which burns the user application code into the NVM to create the preprogrammed “blank” part.

There are several possible approaches and NVM arrangements how to achieve that. The focus will be on the two most useful ones. In both cases, after the device is fully finalized with chip state set to **Run** the NVM content would look is described below:

1. There will be two user application **Blocks** in NVM, one per each input IntelHEX file.
2. The composed **configuration** NVM **Block** starts at the beginning of the **User Boot** section of the NVM at address 0xE180. It consists of the following byte sequence:

| | | |
|---|---|---|
| 0xE180 | 0xFF | .. NVM block start flag |
| 0xE181 | ADDR_MSB | .. MSB of the **configuration** destination RAM address |
| 0xE182 | ADDR_LSB | .. LSB of the destination address |
| 0xE183 | COUNT N | .. value of 1 .. 254, number of data bytes which follow |
| 0xE180+4 | Data[0] | .. **configuration** data bytes |
| | ... | |
| 0xE180+N+4 | Data[N-1] | |
| 0xE180+N+5 | 0x00 | .. **Block** end |
| 0xE180+N+6 | 0x03 | .. return value of 0x03 indicates that the boot will continue loading next **Block** without stopping. |

3. The composed **application** NVM **Block** follows immediately after the Configuration block in NVM. It consists of the following byte sequence:

| | | |
|---|---|---|
| 0xE187+N | 0xFF | .. NVM block start flag |
| 0xE188+N | ADDR_MSB | .. MSB of the **application** destination RAM address |
| 0xE189+N | ADDR_LSB | .. LSB of the destination address |
| 0xE18A+N | COUNT M | .. value of 1 .. 254, number of data bytes which follow |
| 0xE18B+N | Data[0] | .. first section of first **Element** data byte |
| | ... | .. many **Elements** to transfer **application** to RAM |
| 0xEnnn | 0x00 | .. **Block** end |
| 0xEnnn+1 | 0x01 | .. return value of 0x01 indicates that the boot will stop loading data from NVM. End of data in **User Boot** section. |

There are two possible approaches for creating “blank” parts with “empty” configuration and then later adding the configuration into the part with the second burn, using the second NBF file. Both approaches are described in detail.

For example, the user desires 3 bytes of per-chip configuration and the application is written such that it expects those 3 bytes starting at the XDATA RAM address **0x0DFD**. The internals of the empty, all zero, configuration IntelHEX **config_zero.hex** file would therefore be:

```
:030DFD00000000F3
:00000001FF
```

If using the Verilog MEM format, the internals of the equivalent **config_zero.mem** file would be:

```
@0DFD
00 00 00
```

It is clear that the MEM format is easier to read and generate, since there is no need for checksum and end of file record.

The particular part configuration file **config_part_1.hex** or **config_part_1.mem** will have the 3 byte hex values replaced with the actual values and in the IntelHEX case **0xF3** checksum adjusted to the proper value based on the data content.

For example, specific part configuration file **config_part_1.hex**, which sets the configuration data bytes to **0x87**, **0xD5**, and **0x4A** values, would look like this:

```
:030DFD0087D54A4D
:00000001FF
```

The content of the corresponding **config_part_1.mem** would be:

```
@0DFD
87 D5 4A
```

There are two cases, **2A** for a single NBF file, **2B** for creating blanks with two NBF files. The case of **2B** has two possible variants when programming the actual configuration, using either a regular **configuration** file with RAM destination addresses (labeled **2B** in the text below), or using **Direct Burn** when the NVM bytes are written directly with the configuration values (labeled **2Bd** below).

### 10.3.1. Case 2A: Configuration Loaded by Boot, Single NBF File

**Case 2A:** Boot loads both the **configuration** and the **application**, single NBF file.

#### 10.3.1.1. Regular Non-CRC Flow (2A)

Summary:

- Put both **configuration** **config_part_1.hex** and **application** **app.hex** into the **User Boot** area in NVM. If the configuration is in Verilog MEM format **config_part_1.mem**, it must be converted to IntelHEX format using **hexext** script.
- Run **gui_composer** to generate NBF burn file **app_config.nbf**

For ***each part*** do the following steps:

1. Generate per part specific configuration **config_part_1.mem** (or **config_part_1.hex**).
2. Convert the Verilog MEM format to IntelHEX format. If the configuration file is already in the IntelHEX format skip this step:

```
hexext.bat \
   config_part_1.mem \
   --output config_part_1.hex \
   --verbose \
   --overwrite
```

3. Run the composer to generate NBF burn file:

```
gui_composer.exe \
   --boot_hex=config_part_1.hex 0xE180 \
   --boot_hex=app.hex 0 \
   --nvm_burn_file=app_config.nbf \
   --state=run \
   --bf_nvm_dis \
   --bf_ram_clr \
   --bf_mtp_dis \
   --mode_strict
```

4. Burn the part-specific **app_config.nbf** to the part:

```
burn_cl.bat app_config.nbf
```

#### 10.3.1.2. CRC Flow (2A)

Summary:

- Put both **configuration** **config_part_1.hex** and **application** **app.hex** into the **User Boot** area in NVM. If the configuration is in Verilog MEM format **config_part_1.mem** it must be converted to IntelHEX format using **hexext** script.
- Run **gui_composer** to generate NBF burn file **app_config_norun.nbf**
- Run **nbfmod** to generate **app_config_crcflow.nbf**

For ***each part*** do the following steps:

1. Generate per part specific configuration **config_part_1.mem** (or **config_part_1.hex**).
2. Convert the Verilog MEM format to IntelHEX format. If the configuration file is already in the IntelHEX format skip this step:

```
hexext.bat \
   config_part_1.mem \
   --output config_part_1.hex \
   --verbose \
   --overwrite
```

3. Run composer to generate NBF burn file:

```
gui_composer.exe \
   --boot_hex=config_part_1.hex 0xE180 \
   --boot_hex=app.hex 0 \
   --nvm_burn_file=app_config_norun.nbf \
   --bf_nvm_dis \
   --bf_ram_clr \
   --bf_mtp_dis \
   --mode_strict
```

4. Run **nbfmod** to generate **app_config_crcflow.nbf**:

```
.\nbf\nbfmod.bat  \
   check_userempty \
   app_config_norun.nbf \
   check_burn_usercrc \
   burn_run \
   check_pt3way_usercrc \
   --output app_config_crcflow.nbf \
   --autocrc \
   --verbose \
   --overwrite
```

5. Burn the part specific **app_config_crcflow.nbf** to the part:

```
burn_cl.bat app_config_crcflow.nbf
```

### 10.3.2. Case 2B: Configuration Loaded by Boot, Two NBF Files

**Case 2B:** Boot loads both the **configuration** and the **application**, two NBF files. This flow is truly useful for creating preprogrammed “blanks” and adding the configuration by separate burn later. If that is not the case, use the case **2A** instead.

The second NBF burning of the **configuration** uses regular configuration file with RAM destination addresses.

#### 10.3.2.1. Regular Non-CRC Flow (2B)

Summary:

- Put both all zero **configuration** **config_zero.hex** and **application** **app.hex** into the **User Boot** area in NVM. If the configuration is in Verilog MEM format **config_zero.mem** it must be converted to IntelHEX format using **hexext** script.
- Run **gui_composer** to generate NBF burn file **app_config_zero_norun.nbf** to be used to create preprogrammed “blanks”.
- Create actual per part configuration **config_part_1.hex** (or **config_part_1.mem**) and put it into the **User Boot** area on its own.
- Run **gui_composer** to generate configuration NBF configuration burn file **config_part.nbf** with **Block** return value set to **0x03**.

Flow steps:

Run just ***once*** to generate NBF file for blanks:

1. Convert the Verilog MEM format to IntelHEX format. If the zero **configuration** file is already in the IntelHEX format skip this step:

```
hexext.bat \
   config_zero.mem \
   --output config_zero.hex \
   --verbose \
   --overwrite
```

2. Run composer to generate NBF burn file for preprogrammed “blanks”. There must not be any chip state set and the **config_zero.hex** file must come first at 0xE180 address:

```
gui_composer.exe \
   --boot_hex=config_zero.hex 0xE180 \
   --boot_hex=app.hex 0 \
   --nvm_burn_file=app_config_zero_norun.nbf \
   --mode_strict
```

For ***each part create a preprogrammed blank*** part by burning the **app_config_zero_norun.nbf** file:

```
burn_cl.bat app_config_zero_norun.nbf
```

For ***each preprogrammed blank part***, perform the following steps:

1. Generate per part specific **configuration** **config_part_1.mem** (or **config_part_1.hex**). There is no need to convert the Verilog MEM file to an IntelHEX file. The Verilog MEM format will be used in this example.
2. Run the composer to generate NBF burn file:

```
gui_composer.exe \
   --boot_mem=config_part_1.mem 0xE180 \
   --nvm_burn_file=config_part.nbf \
   --boot_return_val=0x03 \
   --state=run \
   --bf_nvm_dis \
   --bf_ram_clr \
   --bf_mtp_dis \
   --mode_strict
```

Note that setting the chip state to **Run** and the user protection flag setting is done during the configuration NBF file creation. User `--boot_hex=config_part_1.hex` if using HEX format.

3. Burn the part specific **config_part.nbf** to the part:

```
burn_cl.bat config_part.nbf
```

#### 10.3.2.2. CRC Flow (2B)

Summary:

- Put both all zero **configuration** **config_zero.hex** and **application** **app.hex** into the **User Boot** area in NVM. If the configuration is in Verilog MEM format **config_zero.mem** it must be converted to IntelHEX format using **hexext** script.
- Run **gui_composer** to generate the NBF burn file **app_config_zero_norun.nbf**.
- Run **nbfmod** to generate **app_config_zero_norun_crcflow.nbf** to be used to create preprogrammed "blanks".
- Run **nbfmod** again to generate the preprogrammed "blank" part NVM content **app_config_zero.nvm.hex** file.
- Create actual part **configuration** **config_part_1.hex** (or **config_part_1.mem**) and put it into the **User Boot** area.
- Run **gui_composer** to generate the configuration NBF burn file **config_part.nbf** with **Block** return value set to **0x03**.
- Run **nbfmod** to generate **config_part_crcflow.nbf** file. This step uses the NVM content **app_config_zero.nvm.hex** file generated previously.

CRC flow steps:

Run just ***once*** to generate NBF file for preprogrammed "blanks":

1. Convert the MEM format to IntelHEX format. If the zero **configuration** file is already in the IntelHEX format skip this step:

```
hexext.bat \
   config_zero.mem \
   --output config_zero.hex \
   --verbose \
   --overwrite
```

2. Run the composer to generate NBF burn file for preprogrammed "blanks". There must not be any chip state set and that the **config_zero.hex** file must come first at 0xE180 address:

```
gui_composer.exe \
   --boot_hex=config_zero.hex 0xE180 \
   --boot_hex=app.hex 0 \
   --nvm_burn_file=app_config_zero_norun.nbf \
   --mode_strict
```

3. Run **nbfmod** to generate **app_config_zero_norun_crcflow.nbf**:

```
.\nbf\nbfmod.bat  \
   check_userempty \
   app_config_zero_norun.nbf \
   check_usercrc \
   --output app_config_zero_norun_crcflow.nbf \
   --autocrc \
   --verbose \
   --overwrite
```

Note that only the NVM empty check and the user CRC check (not burn!) can be done, since the first burn will not be the final burn.

4. Run **nbfmod** again to generate the preprogrammed “blank” part NVM content **app_config_zero.nvm.hex** file:

```
.\nbf\nbfmod.bat \
   app_config_zero_norun_crcflow.nbf \
   --output app_config_zero.nvm.hex \
   --hexout \
   --verbose \
   --overwrite
```

The file **app_config_zero.nvm.hex** file is needed for the final CRC calculation and checks during the configuration burn. Save the file for future use.

For ***each part create a blank*** part by burning **app_config_zero_norun_crcflow.nbf** file:

```
burn_cl.bat app_config_zero_norun_crcflow.nbf
```

For ***each preprogrammed blank part*** do the following steps to burn the configuration:

1. Generate per part specific **configuration** **config_part_1.mem** (or **config_part_1.hex**). There is no need to convert the MEM file to a HEX file. The MEM format will be used in this example.
2. Run the composer to generate the configuration NBF burn file:

```
gui_composer.exe \
   --boot_mem=config_part_1.mem 0xE180 \
   --nvm_burn_file=config_part_norun.nbf \
   --boot_return_val=0x03 \
   --bf_nvm_dis \
   --bf_ram_clr \
   --bf_mtp_dis \
   --mode_strict
```

Note that setting the chip state is still unchanged but the user protection flag setting is done during the configuration NBF file creation. Use `--boot_hex=config_part_1.hex` if using HEX format.

3. Run **nbfmod** to generate **config_part_crcflow.nbf** file. This step uses the NVM content **app_config_zero.nvm.hex** file generated previously.

```
.\nbf\nbfmod.bat \
   --nvmload app_config_zero.nvm.hex \
   config_part_norun.nbf \
   check_burn_usercrc \
   burn_run \
   check_pt3way_usercrc \
   --output config_part_crcflow.nbf \
   --autocrc \
   --verbose \
   --overwrite
```

The file **app_config_zero.nvm.hex** contains the NVM image of the “blank” part, which is needed for the script to see the correct NVM content for the final CRC calculation and burning.

4. Burn the part specific **config_part_crcflow.nbf** to the part:

```
burn_cl.bat config_part_crcflow.nbf
```

### 10.3.3. Case 2Bd: Configuration Loaded by Boot, Two NBF Files, Direct Burn

**Case 2Bd:** Boot loads both the **configuration** and the **application**, two NBF files. This flow is truly useful for creating preprogrammed “blanks” and adding the configuration by separate burn later. If that is not the case, use case **2A** instead.

The second NBF burning **configuration** uses **Direct Burn** of a specially designed **configuration** file with NVM destination addresses (instead of RAM destination addresses as in the 2B case above).

Since the preprogrammed “blank” part has all of the boot framing structures already present in the NVM, and we know the exact physical location of the configuration bytes array, 0xE184, we can use **Direct Burn** to directly burn NVM addresses to replace the 0x00 values with desired configuration values. This is because the preprogrammed “blank” part contains the **config_zero.hex** file content, which was placed at the 0xE180 NVM address by the “blank” programming flow, and the exact configuration data location in the NVM is known.

The N byte data array, matching the size of the **config_zero.hex** file, reside in NVM at NVM direct physical addresses

**0xE184 ... 0xE184+N-1**

We can therefore use NVM **Direct Burn** to burn the configuration data array directly into the NVM address 0xE184 and beyond.

The NVM configuration byte array can come in two forms:

1. Content of the ***.mem** file. Only Verilog MEM format is accepted for **Direct Burn**.
2. Direct data input to the GUI or on the **gui_composer.exe** command line.

The input string data format of both methods is the same – it follows the Verilog MEM file format.

Going back to our example, let’s assume we want to generate a part with **0x87**, **0xD5**, and **0x4A** as the configuration byte array and set the **Run** state at the same time.

Create the file, named **config_part_1_dir.mem**, with the following content:

```
@e184      // <-- Start of the config array in NVM
87 d5 4a   // Desired part configuration data
```

Then use the `--dir_burn_file` option of the **gui_composer.exe** command line:

```
--dir_burn_str=config_part_1_dir.mem
```

Alternatively, it is possible to supply the direct burn data as a string on the **gui_composer.exe** command line using `--dir_burn_str` option:

```
--dir_burn_str="@e184 87 d5 4a"
```

Using double quotes around the string on a command line is mandatory.

Using the direct string on a command line might be helpful for some scripting scenarios. The examples in the following section will only use the file method.

#### 10.3.3.1. Regular Non-CRC Flow (2Bd)

Summary:

- Put both the all zero **configuration** **config_zero.hex** and **application** **app.hex** file into the **User Boot** area in NVM. If the configuration is in Verilog MEM format **config_zero.mem**, it must be converted to IntelHEX format using **hexext** script.
- Run **gui_composer** to generate the NBF burn file **app_config_zero_norun.nbf** to be used to create preprogrammed “blanks”.
- Create the actual part **Direct Burn** configuration file **config_part_1_dir.mem**
- Run **gui_composer** to generate the **Direct Burn** configuration NBF burn file **config_part.nbf**

Flow steps:

Run just ***once*** to generate NBF file for preprogrammed “blanks”:

1. Convert the MEM format to IntelHEX format. If the zero **configuration** file is already in the IntelHEX format skip this step:

```
hexext.bat \
   config_zero.mem \
   --output config_zero.hex \
   --verbose \
   --overwrite
```

2. Run the composer to generate the NBF burn file for preprogrammed “blanks”. There must not be any chip state set and the **config_zero.hex** file must come first at 0xE180 address:

```
gui_composer.exe \
   --boot_hex=config_zero.hex 0xE180 \
   --boot_hex=app.hex 0 \
   --nvm_burn_file=app_config_zero_norun.nbf \
   --mode_strict
```

For ***each part create a preprogrammed blank*** part by burning the **app_config_zero_norun.nbf** file:

```
burn_cl.bat app_config_zero_norun.nbf
```

For ***each preprogrammed blank part*** do the following steps:

1. Generate per part specific **Direct Burn** configuration file **config_part_1_dir.mem**
2. Run the composer to generate NBF burn file:

```
gui_composer.exe \
   --dir_burn_file=config_part_1_dir.mem \
   --nvm_burn_file=config_part.nbf \
   --state=run \
   --bf_nvm_dis \
   --bf_ram_clr \
   --bf_mtp_dis \
   --mode_strict
```

Note that setting the chip state to **Run** and the user protection flag setting is done during the configuration NBF file creation.

3. Burn the part specific **config_part.nbf** to the part:

```
burn_cl.bat config_part.nbf
```

#### 10.3.3.2. CRC Flow (2Bd)

Summary:

- Put both all zero **configuration** **config_zero.hex** and **application** **app.hex** into the **User Boot** area in NVM. If the configuration is in Verilog MEM format **config_zero.mem** it must be converted to IntelHEX format using **hexext** script.
- Run **gui_composer** to generate the NBF burn file **app_config_zero_norun.nbf**
- Run **nbfmod** to generate **app_config_zero_norun_crcflow.nbf** to be used to create preprogrammed "blanks".
- Run **nbfmod** again to generate the preprogrammed "blank" part NVM content **app_config_zero.nvm.hex** file.
- Create the actual part **Direct Burn** configuration file **config_part_1_dir.mem**
- Run **gui_composer** to generate the configuration NBF burn file **config_part.nbf**
- Run **nbfmod** to generate **config_part_crcflow.nbf** file. This step uses the NVM content **app_config_zero.nvm.hex** file generated previously.

CRC flow steps:

Run just ***once*** to generate the NBF file for preprogrammed "blanks":

1. Convert the Verilog MEM format to the IntelHEX format. If the zero **configuration** file is already in the IntelHEX format skip this step:

```
hexext.bat \
   config_zero.mem \
   --output config_zero.hex \
   --verbose \
   --overwrite
```

2. Run the composer to generate the NBF burn file for "blanks". There must not be any chip state set and the **config_zero.hex** file must come first at 0xE180 address:

```
gui_composer.exe \
   --boot_hex=config_zero.hex 0xE180 \
   --boot_hex=app.hex 0 \
   --nvm_burn_file=app_config_zero_norun.nbf \
   --mode_strict
```

3. Run **nbfmod** to generate **app_config_zero_norun_crcflow.nbf**:

```
.\nbf\nbfmod.bat  \
   check_userempty \
   app_config_zero_norun.nbf \
   check_usercrc \
   --output app_config_zero_norun_crcflow.nbf  \
   --autocrc \
   --verbose \
   --overwrite
```

Note that only the NVM empty check and the user CRC check (not burn!) can be done, since the first burn will not be the final burn.

4. Run **nbfmod** again to generate the preprogrammed “blank” part NVM content **app_config_zero.nvm.hex** file:

```
.\nbf\nbfmod.bat \
   app_config_zero_norun_crcflow.nbf \
   --output app_config_zero.nvm.hex \
   --hexout \
   --verbose \
   --overwrite
```

The file **app_config_zero.nvm.hex** file is needed for the final CRC calculation and check during the configuration burn. Save the file for future use.

For ***each part create a blank*** part by burning **app_config_zero_norun_crcflow.nbf** file:

```
burn_cl.bat app_config_zero_norun_crcflow.nbf
```

For ***each preprogrammed blank part*** do the following steps to burn the configuration:

1. Create per part specific **Direct Burn** configuration file **config_part_1_dir.mem**
2. Run the composer to generate the configuration NBF burn file:

```
gui_composer.exe \
   --dir_burn_file=config_part_1_dir.mem \
   --nvm_burn_file=config_part_norun.nbf \
   --bf_nvm_dis \
   --bf_ram_clr \
   --bf_mtp_dis \
   --mode_strict
```

Note that setting the chip state is still unchanged but the user protection flag setting is done during the configuration NBF file creation.

3. Run **nbfmod** to generate the **config_part_crcflow.nbf** file. This step uses the NVM content **app_config_zero.nvm.hex** file generated previously.

```
.\nbf\nbfmod.bat \
   --nvmload app_config_zero.nvm.hex \
   config_part_norun.nbf \
   check_burn_usercrc \
   burn_run \
   check_pt3way_usercrc \
   --output config_part_crcflow.nbf \
   --autocrc \
   --verbose \
   --overwrite
```

The file **app_config_zero.nvm.hex** contains the NVM image of the “blank” part, which is needed for the tool to see the correct NVM content for the final CRC calculation and burning.

4. Burn the part specific **config_part_crcflow.nbf** to the part:

```
burn_cl.bat config_part_crcflow.nbf
```

# 11. Supply Voltage and Programming Voltage

The device requires the 3.3 V VDD and 6.5 V programming voltage switched on and off in the correct sequence to operate correctly. The 3.3 V VDD should be applied first, then the 6.5 V programming voltage.

The command line burner (and only the command line burner) performs the following sequence of events when executed:

1. It tries to connect to the USB Debug Adapter specified or to a single adapter if none is specified. It is important to note: It connects to the adapter, NOT to the part itself.
2. When connected to the adapter it turns on the USB VBUS voltage, which is the pin 10 on the USB Debug Adapter ribbon cable header. That is ~5V supplied from the USB.
3. After about 200 ms it tries to connect to the part.
4. When connected to the part it processes the NBF file sections, downloads the burn programs and burns the part.
5. After burning is done it disconnects from the part, removes the VBUS 5V voltage from the USB header, and disconnects from the USB adapter itself.

When the USB debug adapter is connected to the actual part, it samples the actual supply voltage of the target part about once a second (it looks at the C2CLK pin high level as a reference) and adjusts the output logic levels accordingly to match those of the target.

Therefore, connecting to the part and applying voltages are independent actions.

## 11.1. Supply Voltages are Generated by the User

If the user provides external power voltages to the device, bypassing the **MSC-BA4** board, the user has to generate the power supply sequencing. The VBUS pin (pin 10) on the USB debug header can be used as a control.

When the command line burner is used, then this is the USB ~5 V power supply which gets turned on before any connection to the part happens and turned off after the part is burned and the adapter disconnects from the part.

**Note:** If the user provides external power sequencing then the device 3.3 V VDD must come up before the 6.5 V VPP programming voltage. The 6.5 V VPP programming voltage has to be turned off before the 3.3 V VDD can be turned off.

There is no time limit for how long the 6.5 V VPP programming voltage could be applied to the device. The device has internal switches to internally apply the 6.5 V VPP programming voltage when it needs it.

## 11.2. Supply Voltages are Provided by the Silicon Labs Programming Adapter Board (MSC-BA4)

On that board the VBUS 5 V voltage is passed into two places: linear regulator generating main 3.3 V and a charge pump, which pumps it up to 6.5 V. Both have 10 µF capacitors at their outputs. If the VBUS 5 V is applied and the 6.5 V switch is on (required setting when using command line burner), then the 3.3 V output is assumed to be faster than the 6.5 V supply. On the way down when the 5 V input to both branches is turned off the 6.5 V is going to be discharged by device internal diodes as the 3.3 V voltage is going down so both voltages collapse quickly.

When using the Silicon Labs programming board the 6.5 V switch has to be turned on all the time, since applying the 5 V to the charge pump is controlled by the command line programmer as described above. This applies only for the command line programmer. This does not apply for the GUI version, where the sequencing is different.

**Notes:**

1. The 6.5 V is not applied all the time in this case, only the switch on the programming board is in the ON position all the time. The presence of the 6.5 V is controlled by the command line burner through the VBUS voltage and is indicated by a red LED on the **MSC-BA4** board.
2. The 6.5 V charge pump can supply only a few mA of current needed during burning, so it is not a hard power supply.

# 12. NVM Composer Details (gui_composer.exe)

## 12.1. NVM Burner GUI and NVM Composer Options Matching

The following tables describe the relationship between the NVM burner GUI and the corresponding command line options for the **gui_composer.exe** executable. When using the NVM burner GUI, based on the inputs from the GUI the **gui_composer.exe** command line is formed behind the scenes. The executed command line is also included at the beginning of each NBF file generated for reference.

**User** and **Run** check boxes. If the box is not set the command line argument is not used:

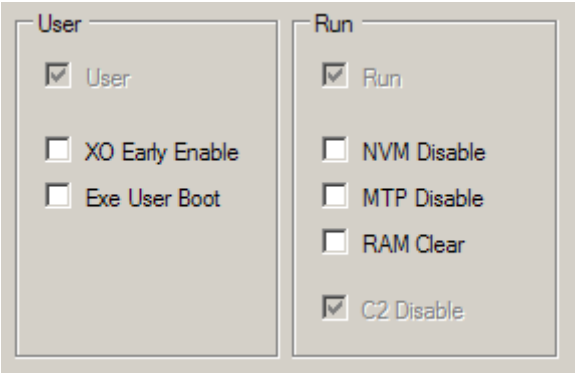


**Figure 5. User and Run Checkboxes**

**Table 2. Check Box Command Line Arguments**

| Check Box Set | Command Line Argument |
|---|---|
| User | `--state=user` |
| Run | `--state=run` |
| Both Run and User | `--state=run` |
| XO Early Enable | `--bf_xo_early_enable` |
| Exe User Boot | `--bf_exe_user_boot` |
| NVM Disable | `--bf_nvm_dis` |
| MTP Disable | `--bf_mtp_dis` |
| RAM Clear | `--bf_ram_clr` |
| C2 Disable | `--bf_c2_dis` |

**Compose Mode** radio button:

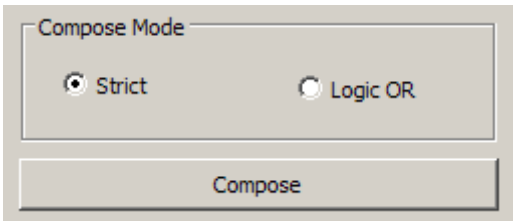


**Figure 6. Compose Mode Radio Button**

**Table 3. Compose Mode Command Line Arguments**

| Compose Mode | Command Line Argument |
|---|---|
| Strict | `--mode_strict` |
| Logic OR | None (default setting) |

**NVM Burn File** output file specification:

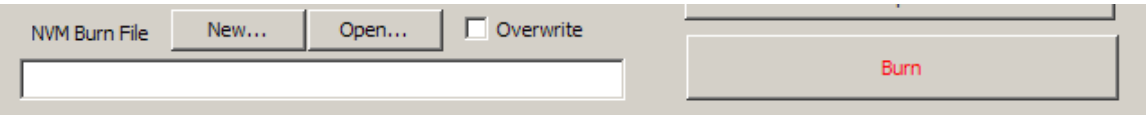


**Figure 7. NVM Burn File**

**Table 4. NVM Burn File Command Line Arguments**

| NVM Burn File | Command Line Argument |
|---|---|
| file.nbf | `--nvm_burn_file=file.nbf` |
| .\out\file.ext | `--nvm_burn_file=.\out\file.ext` |

The **User Boot Files** list has one-to-one correspondence with the command line arguments:

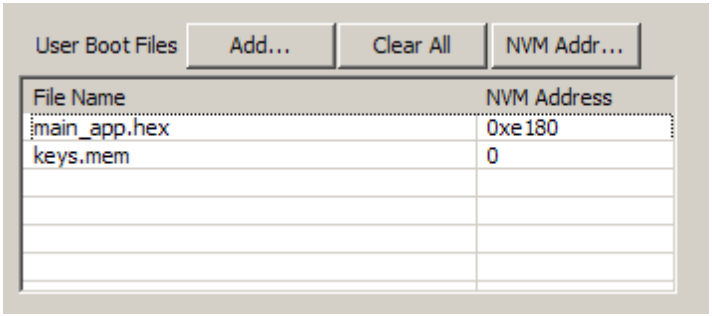


**Figure 8. User Boot Files List**

**Table 5. User Boot File NVM Addresses and Command Line Arguments**

| User Boot File | NVM Address | Command Line Argument |
|---|---|---|
| main_app.hex | `0xE180` | `--boot_hex=file_path\main_app.hex 0xE180` |
| keys.mem | `0` | `--boot_mem=file_path\keys.mem 0x0` |
| file.hex | `0xE230` | `--boot_hex=file_path\file.hex 0xE230` |
| file.mem | `0xE454` | `--boot_mem=file_path\file.mem 0xE454` |

The **User App Files** list has one-to-one correspondence with the command line arguments:

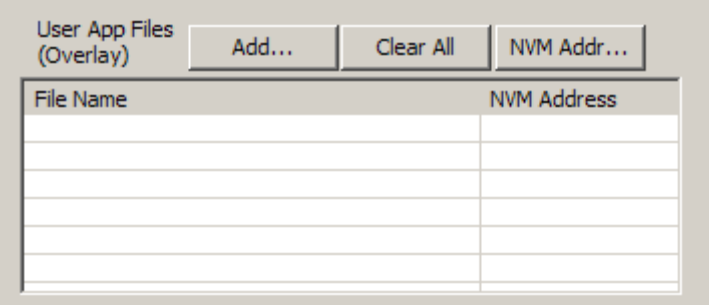


**Figure 9. User App Files List**

**Table 6. User App File NVM Addresses and Command Line Arguments**

| User App File | NVM Address | Command Line Argument |
|---|---|---|
| file.mem | `0xEF91` | `--app_mem=file_path\file.mem 0xEF91` |
| keys.hex | `0xF0AB` | `--app_hex=file_path\keys.hex 0xF0AB` |

**Direct Burn File** specification:

**Table 7. Direct Burn File Command Line Arguments**

| User App File | Command Line Argument |
|---|---|
| file.mem | `--dir_burn_file=file_path\file.mem` |

**Direct Burn Text** box content:

**Table 8. Direct Burn Text Command Line Arguments**

| Direct Burn Text | Command Line Argument |
|---|---|
| `@e184 87 d5 4a` | `--dir_burn_str="@e184 87 d5 4a"` |
| `@e1fa 34 @e205 c4 47` | `--dir_burn_str="@e1fa 34 @e205 c4 47"` |

There are **gui_composer.exe** command line options which have no GUI counterpart:

**Table 9. gui_composer.exe Command Line Arguments**

| Check Box Set | Command Line Argument |
|---|---|
| N/A | `--bf_run_nvm_wr` |
| N/A | `--boot_return_val=BOOT_RETURN_VAL` |

## 12.2. Composer Command Line Options (gui_composer.exe)

The following are command line options for the **gui_composer.exe** version 1.8

```
GUI Composer Version 1.8 | Region: 0x0500
```

**Table 10. gui_composer.exe Command Line Options**

| Option | Description |
|---|---|
| `--state=STATE` | Desired chip state (**keep**, **user**, **run**); **keep** leaves the chip state as it is. [default: **keep**] |
| `--nvm_burn_file=NVM_BURN_FILE` | NVM Burn File (NBF) file name. This file represents the output of composer process and input to burn process. |
| `--mode_strict` | If specified the generated NBF file burns the device in **Strict** mode. If not specified (default) then the **Logic OR** burn mode is generated. |
| `--bf_xo_early_enable` | Boot flag: Enable the crystal oscillator at the very beginning of the boot. |
| `--bf_exe_user_boot` | Boot flag: Sets the **ExeUserBoot** bit. If specified the part in **User** state executes the user application after the boot. If not specified, the part will boot the user application and it will stop. |
| `--bf_nvm_dis` | Boot flag: Disable NVM access while in **Run** state during retest. If specified then NVM will be disabled and unreadable when the **Run** device is opened for retest to protect user IP. |
| `--bf_mtp_dis` | Boot flag: Disable MTP access while in **Run** state during retest. If specified then MTP access will be disabled for both read and write after when the **Run** device is opened for retest to protect user IP. |
| `--bf_ram_clr` | Boot flag: Clear all RAM data while in **Run** state during retest. If specified both XDATA and DATA RAM's are cleared when the **Run** device is opened for retest to protect user IP even more. |
| `--bf_c2_dis` | Boot flag: Disable C2 interface in **Run** state. If specified the C2 interface is disabled once the device is programmed to **Run** state. The C2 interface is disabled and the device cannot be opened for retest and is locked. No further access is possible. Silicon Labs cannot retest the part and do failure analysis on a failed part. It is recommended not to use this option. Use with caution! |
| `--bf_run_nvm_wr` | If used then the NVM will be kept writable even after the device state changes to **Run**. If used, it must be specified at the same time when **`--state=run`** is specified. For experimental purposes only, do not use for production parts. If not used then the NVM is write-protected once the device is programmed to **Run** state. |

**Table 10. gui_composer.exe Command Line Options (Continued)**

| | |
|---|---|
| `--boot_hex=BOOT_HEX_FILE NVM_ADDR` | Input IntelHEX file to be composed into the **User Boot** region. The NVM_ADDR is the physical NVM address where the composed file begins in the NVM. The first file on the command line must have NVM_ADDR=0xE180. If the NVM_ADDR=0 then the file is composed in the NVM immediately after the previously processed file. |
| `--boot_mem=BOOT_MEM_FILE NVM_ADDR` | Input Verilog MEM file to be composed into the **User Boot** region. Same address comment as above. For further information please check section 12.4. |
| `--app_hex=APP_HEX_FILE NVM_ADDR` | Input IntelHEX file to be composed into **User App** region, which is not loaded at boot time and needs to be loaded by the application itself. The NVM_ADDR is required to be specified and must be non-zero. |
| `--app_mem=APP_MEM_FILE NVM_ADDR` | Input Verilog MEM file to be composed into User App region, which is not loaded at boot time and needs to be loaded by the application itself. |
| `--dir_burn_file=DIRECT_BURN_FILE` | Direct burn file in Verilog MEM format. Not to be used with the following options:<br>--boot_hex<br>--boot_mem<br>--app_hex<br>--app_mem |
| `--dir_burn_str="DIRECT_BURN_STR"` | Direct burn string in Verilog MEM format. Same comment as in `--dir_burn_file` above. Mutually exclusive with it. Note that the string must be in quotes.<br>Example: `"@e184 87 d5 4a"` |
| `--boot_return_val=BOOT_RETURN_VAL` | Hexadecimal value in 0xNN notation to be used as a return value of the last block. Normally it should not be used, but in some configuration burn flows it has to be specified for the boot to continue to load data. |
| `--verbose` | Display verbose messages of internal working of the NBF compose process. |

## 12.3. Composer Return Values (gui_composer.exe)

The **gui_composer.exe** returns a small positive value (decimal value less than 50) as a return value. Value of 0 means that everything was fine. Value greater than 0 means error:

**Table 11. Composer Return Values**

| Exit Value | Description |
|---|---|
| 0 | Success |
| 1 | **Command Line Error:** Some portion of the command line arguments was used incorrectly. |
| 2 | **First Boot File Address Error:** The first boot file, be it `--boot_hex` or `--boot_mem` has am invalid NVM address associated with it. |
| 3 | **Burner Creation Error:** Temporary burner files creation failure. The temporary burner files are created during the compose process and then deleted at the end. This error indicates that the tool was not able to create the required temporary files. Check the directory write permissions. |
| 4 | **Output Dir Error:** Output directory creation failed. |
| 5 | **Boot HEX Input Error:** Error while loading a file specified by the `--boot_hex` option. Error in the input IntelHEX file. |
| 6 | **Boot MEM Input Error:** Error while loading a file specified by the `--boot_mem` option. Error in the input Verilog MEM file. |
| 7 | **App HEX Input Error:** Problem loading a file specified by the `--app_hex` switch. Error in the input IntelHEX file. |
| 8 | **App MEM Input Error:** Problem loading a file specified by the `--app_mem` switch. Error in the input Verilog MEM file. |
| 9 | **Direct Burn Input Error:** Error in the direct burn file or direct burn string. |
| 10 | **Address Specification Error:** NVM address supplied outside the valid NVM address range. |
| 11 | **NVM Burn File Error:** NVM burn file (NBF) write error. |
| 12 | **Unhandled Exception:** There was a problem which isn't handled by the GUI Composer. |

## 12.4. Composer Limitations

As of **gui_composer.exe** version 1.8 there are the following limitations during the NBF creation process:

1. The first **User Boot** file placed at 0xE180 address must be a ***.hex** file, not the ***.mem** file if more than one boot file is specified. If the only **User Boot** file is a single ***.mem** file or all the boot files are ***.mem** files there is not a problem. Obviously, if all the boot files are ***.hex** files there is not a problem either.
2. It is recommended to use only either ***.hex** files or ***.mem** files during the single NBF file generation.
3. If the mixture of ***.hex** and ***.mem** files is used, then all ***.hex** files are processed first in the order in which they appear on the command line, followed by ***.mem** files in the order in which they appear on the command line.
4. It is not possible to change the state of the part when using the GUI while generating the NBF file for **Direct Burn**. It is possible to do that while using the **gui_composer.exe** command line.
5. **Important:** The composer requires that there is no space in the path or file name. There must not be any space in the file path or in the file name itself for any files supplied to the composer either in the GUI or on a command line.

# 13. Burn Process Return Values

During the burn process the burner sequentially processes the NBF file **[File *N*]** sections. It processes those sections sequentially **[File 1]**, **[File 2]**, etc. After each of the **[File *N*]** sections is processed, the burn numeric status is returned to the burner and it reports the status back to the user. The burner stops immediately when the first error is encountered. No further **[File *N*]** sections are processed from the NBF file after the first error was encountered.

The burn numeric status is an integer number. Value 0 means success and the burner will process subsequent **[File *N*]** sections. The value other than 0 means error and the burner will stop and report the value.

The user can burn the NBF file in two ways:

1. Using GUI **Si4010_NVM_Burner.exe**. The user will either use the NBF file just composed or load a previously composed NBF file. Then the **Burn** button will start the burn process. The burn status/error numeric value along with the text is described in a pop up window.
2. Using command line CL **Si4010_NVM_Burn_CL.exe** or the associated batch wrapper **burn_cl.bat**. The exit value of both is 0 when there are no errors, or a non-zero value if there was an error. Since there could be some other errors like failed connectivity, the burn error values from the burn process are offset by 30 by adding decimal number 30 to the actual burn process error status value.

Therefore, if the GUI burner burn error status value is 2, the CL burner will return 32 in that case.

## 13.1. GUI Burner Displayed Error Values (Si4010_NVM_Burner.exe)

The values returned from the actual burn process:

**Table 12. GUI Burner Error Codes**

| Burn Error Code | Description |
|---|---|
| 1 | **XDATA Address Overflow:** The burner program moved out of the valid XDATA address range. This should never happen. |
| 2 | **Bit Conflict:** The user was running the burner in strict mode and a 0 was scheduled to be burned into a location that already had a 1. |
| 3 | **Exceeded Burn Threshold:** This error occurs when a given bit in the NVM has not successfully burned after 10 attempts. |
| 4 | **Burn Outside Valid NVM Address:** The burner has been instructed to burn an NVM address which is outside the valid NVM address range. Last 64 bytes of NVM are reserved for Silicon Labs production test (PT). User code is not allowed to be burned there. |

If the user is using some of the provided **.\nbf*.nbf** files along with the **nbfmod.bat** script, the burn process does other things than actual burning and the user will encounter an extended set of burn process return values. Those are listed below along with the specialized NBF file names which return those values:

**Table 13. Special NBF File Burn Error Codes**

| Burn Error Code | Special NBF File | Description |
|---|---|---|
| 8 | `check_userempty` | **User NVM Not Empty:** Check whether the User portion of the NVM is completely blank failed. There was some bit encountered with value 1 (already burned). |
| 9 | `burn_usercrc`<br>`check_burn_usercrc`<br>`burn_check_usercrc` | **PT UserCRC Already Burned:** Production test (PT) area reserved for User CRC value (4 bytes) was already burned. It must have 0x00000000 value to be able to burn UserCRC there. |
| 10 | `check_burn_usercrc`<br>`burn_check_usercrc`<br>`check_usercrc`<br>`check_pt3way_usercrc` | **Calculated UserCRC Unexpected:** The calculated CRC over the User portion of the NVM does not match user-specified expected number. Only usable for fixed, non-serialized, NVM loads. |
| 11 | `check_pt3way_usercrc`<br>`check_pt_usercrc` | **Calculated UserCRC Does Not Match PT Stored:** The calculated CRC over the User portion of the NVM does not match the UserCRC value previously burned in the PT area of the NVM. |

## 13.2. Command Line Burner Exit Values (Si4010_NVM_Burn_CL.exe)

The values returned from the actual burn process when using the command line version of the tool are below. The convenience script **burn_cl.bat** will call the **Si4010_NVM_Burn_CL.exe** and exit with the same values, but will print the interpretation of the values before it exits. It is recommended to use **burn_cl.bat** rather than the **Si4010_NVM_Burn_CL.exe** directly when working on a command line.

**Table 14. Command Line Burner Exit Values**

| Exit Code | Description |
|---|---|
| –1 | **Wrong Response:** Unexpected device response. |
| 0 | **Success:** No errors, the burn process succeeded. |
| 1 | **File Error:** Input NBF file cannot be opened. |
| 2 | **Connection Error:** It is not possible to connect to the device. Maybe the device is not present, etc. |
| 3 | **Download Sequence Error:** Download of the burn code to the device failed. |
| 4 | **Reset Sequence Error:** The device cannot be reset. |
| 5 | **Disconnect Error:** The debugger encountered an error while disconnecting from the device. |
| 8 | **Invalid Option:** Invalid command line option. |
| 11 | **USB Debug Adapter DLL Error:** The USB Debug Adapter required DLL cannot be found. |
| 12 | **USB Debug Adapter Error:** USB Debug Adapter encountered an error. |
| 20 | **Burn Code Finish Error:** Burn code executed to the expected end but the status value is unexpected. |
| 21 | **Burn Code Stuck Error:** Burn code did not come to expected end and got lost/stuck during execution. |
| **Returned by Failed Burn Process (See Above):** | |
| 31 | **XDATA Address Overflow:** The burner program moved out of the valid XDATA address range. This should never happen. |
| 32 | **Bit Conflict:** The user was running the burner in **Strict** mode and a 0 was requested to be burned into a location that already had a 1. |
| 33 | **Exceeded Burn Threshold:** A bit in the NVM has not successfully burned after 10 attempts. |
| 34 | **Burn Outside Valid NVM Address:** The burner has been instructed to burn an NVM address which is outside the valid NVM address range. |
| **Returned by Failed Specialized NBF Files (See Above):** | |
| 38 | **User NVM Not Empty:** Check whether the user part of the NVM (**User Boot** and **User App**) is completely blank failed. There was some bit encountered with value 1 (already burned). |
| 39 | **PT User CRC Already Burned:** Production test (PT) area reserved for user CRC value (4 bytes) was already burned. It must have 0x00000000 value to be able to burn user CRC there. |

**Table 14. Command Line Burner Exit Values (Continued)**

| | |
|---|---|
| 40 | **Calculated User CRC Unexpected:** The calculated CRC over the user part of the NVM (**User Boot** and **User App**) does not match user-specified expected number. Only usable for fixed, non-serialized, NVM loads. |
| 41 | **Calculated User CRC Does Not Match PT Stored:** The calculated CRC over the User portion of the NVM (**User Boot** and **User App**) does not match the user CRC value previously burned in the PT area of the NVM. |

## 14. Si4010 MTP Programming

The Si4010 burn tools can also be used to program the 16 byte MTP (EEPROM) memory of the Si4010. To achieve this Silicon Labs provides a specialized mtp_burn.hex file along with the NBF concatenation and modifying script nbfmod. This example shows how to write the following three bytes to the MTP at MTP address 0x08 while keeping the rest of the MTP intact:

0xa1 0xc2 0xd3

First create a file, either IntelHEX or Verilog MEM, with the following content. This example uses Verilog MEM. The file name, for the sake of the example, would be mtp_data.mem. The address in the file is the MTP address in the range 0x00 .. 0x0F:

```
@08 // Hex MTP address
a1 c2 d3 // 3 bytes of MTP data to be written to 0x08, 0x09, and 0x0a
     //MTP address
```

Then create an mtp.nbf file:

```
gui_composer.exe \
--burner_hex mtp_burn.hex \
--boot_mem mtp_data.mem 0xF000 \
--nvm_burn_file mtp.nbf \
--mode_strict
```

The address 0xF000 is an arbitrary address in the range 0xE000 .. 0xFFBF. It is not used, but must be specified to satisfy the checkers in the gui_composer.exe code. It is a dummy value. Then to program the MTP just "burn" the mtp.nbf into the device:

```
burn_cl mtp.nbf
```

If a data byte at a particular MTP address is not specified in the mtp_data.mem file then it will be kept untouched.

The key line is the use of the special burn program mtp_burn.hex instead of the NVM burning default gui_burn.hex.

Note the --mode_strict switch. It has the following meaning:

--mode_strict used: The MTP data specified in the mtp_data.mem file will be written to the MTP directly, overwriting the current content.

--mode_strict NOT USED: The MTP data specified in the mtp_data.mem file will be first ORed with the current MTP content and the ORed value will then be written back to the MTP.

It is the same behavior in the case of NVM: Default is Logic OR if the --mode_strict is not used.

It is possible to concatenate the mtp.nbf file with the other NBF files when creating a final single NBF file.

For example, to clear the MTP during the programming of the NBF when using the burn flow, perform the following steps:.

1. Create the MTP file mtp_zeros.mem:
   @00
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 // 16 zeros
2. Create the mtp.nbf file:
   gui_composer.exe \
   --burner_hex mtp_burn.hex \
   --boot_mem mtp_zeros.mem 0xF000 \
   --nvm_burn_file mtp.nbf \
   --mode_strict
3. Add the generated mtp.nbf file into the final NBF creation. Run nbfmod to generate final NBF burn config_part_crcflow.nbf file concatenating all the intermediate NBF files.
   Add the MTP NBF there as well.

```
.\nbf\nbfmod.bat \
check_userempty \
app.nbf \
config_part.nbf \
mtp.nbf \
check_burn_usercrc \
check_pt3way_usercrc \
--output config_part_crcflow.nbf \
--autocrc \
--verbose \
--overwrite
```

4. Burn the application + configuration NBF + MTP clean burn file config_part_crcflow.nbf to the Si4010 part:

```
burn_cl.bat config_part_crcflow.nbf
```


## Simplicity Studio

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

*www.silabs.com/simplicity*

**MCU Portfolio**
*www.silabs.com/mcu*

**SW/HW**
*www.silabs.com/simplicity*

**Quality**
*www.silabs.com/quality*

**Support and Community**
*community.silabs.com*



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

**http://www.silabs.com**