

# Kinetis Bootloader to Update Multiple Devices in a Field Bus Network

## 1. Introduction

This application note describes how to perform an in-system reprogramming of Kinetis devices using standard communication media such as SCI. Most of the codes are written in C, so it is easy to migrate to other MCUs. The GUI is provided.

This bootloader is based on the FRDM-KL26 Freescale Freedom development board. However, in the software package, the bootloader and user application source codes for both the FRDM-KL26 Freescale Freedom development board (ARM® Cortex®-M0+ core) and the FRDM-K22F Freescale Freedom development board (ARM® Cortex®-M4 core) are provided. The customer can make their own bootloader applications based on the source code. The application can be used to upgrade a single target board and multi boards connected through networks such as RS485. The bootloader application checks the availability of the nodes between the input address range, and automatically upgrades the firmware nodes one by one.

The bootloader and application code are written in separate projects. For mass production, users can program the bootloader with tools such as CycloneMAX and J-Link, then boot the application code through UART. Another option is to combine the bootloader and application s19 file in one file, and

## Contents

1.	Introduction.....	1
2.	Bootloader Frame Protocol .....	3
2.1.	Bootloader Frame Protocol overview.....	3
2.2.	Command and Response Frame .....	4
3.	Memory relocation and code implementation .....	6
3.1.	Memory and vector relocation .....	6
3.2.	Bootloader flowchart.....	8
3.3.	Bootloader implementation .....	9
4.	Communication between GUI and Target.....	15
4.1.	Program execution – Program without verify .....	16
4.2.	Program execution - Program with verify .....	18
4.3.	Verify execution.....	19
4.4.	Log file generation .....	21
5.	Conclusion .....	21
6.	References .....	21
7.	Revision history .....	22



## Introduction

program it. This combination can be done by copying <user\_app>.s19 to the end of <boot\_loader>.s19.

### NOTE

Combination records started by s7, s8, or s9 in the boot.s19 section should be deleted, because these records are recognized as the end of the file.

When upgrading, the bootloader updates the user application code space starting from 0x1000. For verification, the user application space from 0x1000 is also verified. As all source codes are provided, users can define their own programming and verification memory range as needed.

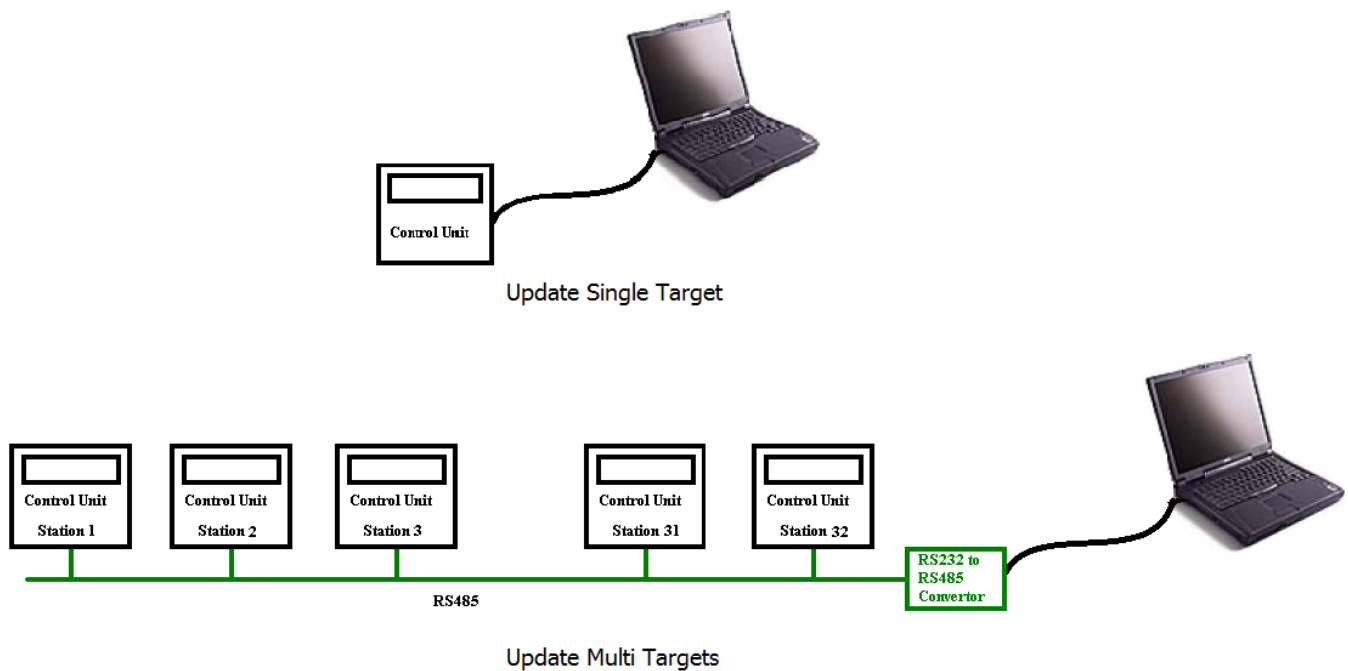


Figure 1. A typical use case

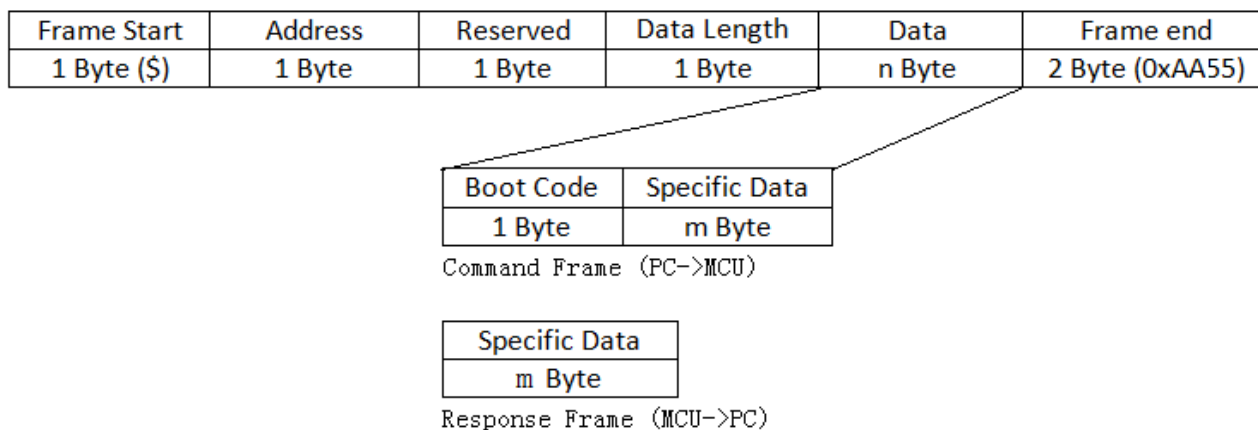
### Key features of the bootloader:

- Able to update (or verify) single or multiple devices in a network.
- Application code and bootloader code are in separate projects, convenient for mass production and firmware upgrading.
- Bootloader code size is small (around 2 K) which reduces the requirement of on-chip memory resources.
- GUI supports S19, HEX, and BIN format burning images.
- Source code is available for easy for reading and migration.

## 2. Bootloader Frame Protocol

### 2.1. Bootloader Frame Protocol overview

Bootloader frames are used for communication between a PC and target control units. The frame starts with a 1 byte frame start, followed by 1 byte address, 1 byte reserved, 1 bytes data length, 1 byte boot code (only for command frame),  $m$  byte-specific bytes, and 2 bytes of frame end. A node only processes frames using the same node address.



**Figure 2. Boot Frame**

- **Frame Start:** Frame header is a single byte, character '\$', which is a frame flag indicating the start of a frame.
- **Address:** The address could be from 1 to 255. The Master (GUI running on a PC) adds a target node's address in the 'Address' field to communicate with it. The node also adds its own node address in the 'Address' field of the response frame, so the Master knows which node is responding. Address 0 can be selected for broadcasting. In the bootloader code, the address is defined as a fixed const:

```
Station_number = 0x01;
```

Users can use different ways to implement the address of a node. I.e., reading a DIP Switch on GPIO pins, reading it from an external EEPROM, or keeping a sector in flash and programming the serial number using CycloneMAX (see the Cyclone\_MAX\_User\_Manual).

- **Reserved:** The user can customize it based on user-specific requirements.
- **Data Length:** For the command frame: Data Length = Boot Code length + Specific Data length. For the response frame: Data Length = Specific Data length.
- **Data:** The command frame is the data bytes from the bootloader frame, from the PC to the target MCU. The response frame are the data bytes from the bootloader frame, from the target MCU to PC. See Table 1 for more details.

- **Frame End:** This two-byte field is always filled with 0xAA55 as the end of frame. To improve robustness, users can add CRC data here and revise the bootloader code and GUI code accordingly.
- **Boot Code:** See Table 1 for more details.
- **Specific Data:** See Table 1 for more details.

## 2.2. Command and Response Frame

A Bootloader Frame command and response Frame as shown in below figure:

- Command Frame direction: PC GUI -> target MCU
- Response Frame direction: Target MCU -> PC GUI

**Table 1. Command and Response Frame**

NO.	Frame type	Data Length	Boot Code	Specific Data	Function
1	command	1	'B'	n/a	Forces target to enter bootloader mode.
	response	No response. The program jumps to bootloader immediately.			
2	command	1	'V'	n/a	Forces target to enter verification mode.
	response	No response. The program jumps to bootloader immediately.			
3	command	1	'I'	n/a	Identification of target information.
	response	Target responds with a string of MCU identification.			
4	command	1	'G'	n/a	Forces target to run.
	response	No response. The bootloader program forces a system reset and runs the application code.			
5	command	4	'E'	Addr2 Addr1 Addr0	Forces target Erase-specific flash block.
	response	If target executes the command successfully, it responds a 0 length frame. I.e., 0x24 0x01 0x00 0x00 0xAA 0x55. Otherwise, there is no response.			
6	command	5+LEN	'W'	Addr2 Addr1 Addr0 LEN Data	Forces target Program-specific flash block.
	response	If target executes the command successfully, it responds a 0 length frame. I.e., 0x24 0x01 0x00 0x00 0xAA 0x55. Otherwise, there is no response.			
7	command	5	'R'	Addr2 Addr1 Addr0 LEN	Forces target Read-specific flash block.
	response	LEN	n/a	LEN Data	Sends specific flash block content to PC.

Some of the boot codes are derived from FC protocol commands in *Developer's Serial Bootloader* (document [AN2295](#)) and *HCS08 Bootloader to Update Multiple Devices in a Field Bus Network* (document [AN4440](#)). For more detailed descriptions of the boot codes:

**‘B’** (0x42 in hex): This command forces the target to enter bootloader mode. It does not wait for a response. Follow with the ‘I’ command to check whether the target MCU is in bootloader mode. Use “APP\_OK” to erase the last 8 bytes of unprotected flash memory. If there is a power down event during the boot load procedure, the MCU is forced into bootloader mode in the next power up.

**‘V’** (0x56 in hex): This command forces the target to enter verification mode. It does not wait for a response. Follow with the ‘I’ command to check whether the target MCU is in verification mode. In verification mode, it only compares the contents in the target MCU with a specific S19 file. Nothing changes in the target MCU.

**‘I’** (0x49 in hex): After receiving the ‘I’ command, the MCU responds with a string of MCU identification from bootloader.

An example of a Response Frame to command ‘I’:

```
0x24 0x01 0x00 0x1F 0x4D 0x4B 0x4C 0x32 0x36 0x5A 0x31 0x32 0x38 0x23 0x31 0x2E 0x30 0x23
0x00 0x40 0x04 0x00 0x02 0x00 0x00 0x00 0x03 0xFC 0x00 0x03 0xFF 0x00 0x10 0x00 0x01 0xAA
0x55
```

In the above series, the “Specific Data” of the Response Frame is highlighted (See Figure 2 and Table 2). This means the target part number is “MKL26Z128”, the bootloader version is “1.0”, target write block size is 0x0040, target erase block size is 0x0400, target flash address end address is 0x00020000, The bootloader does not care memory start address is 0x0003FC, bootloader doesn’t care memory end address is 0x0003FF, user application start address is 0x001000, and target is ARM Cortex M0+ core.

“Specific data” of Response Frame to command ‘I’:

**Table 2: Specific data” of Response Frame to ‘I’ command**

NO.	Info length	Specific data	Description
1	1	length	The specific data length for responding to the ‘I’ command.
2	Not fixed	part number	Target part number, ending with ‘#’. I.e., "MKL26Z128#".
3	Not fixed	Boot version	Bootloader version information, ending with ‘#’. I.e., “1.0#”.
4	2	Size1 size0	Target-write-block size.
5	2	Size1 size0	Target-erase-block size.
6	3	Addr2 Addr1 Addr0	Target flash address end address. I.e., MKL26Z128 end address is 0x00020000.
7	3	Addr2 Addr1 Addr0	Bootloader doesn’t care memory start address. The GUI does not verify no-care memory when executing the verify command.
8	3	Addr2 Addr1 Addr0	Bootloader doesn’t care memory end address.
9	3	Addr2 Addr1	Application start address.

		Addr0	
10	1	Target core	1: Cortex M0+ 2:Cortex M4

**‘G’** (0x47 in hex): After receiving the ‘G’ command, the MCU exits bootloader and forces the user application code to run. It does not respond to a frame. The user can use application software to ensure that the target MCU is running the user code.

**‘E’** (0x45 in hex): This command is used to erase a block of the flash in the target MCU. Based on the Kinetis memory map, the address is 24 bits. Addr2 contains the highest bits, and Addr0 contains the lowest bits. After executing the command, it responds with a zero length frame. For example, 0x24 0x01 0x00 0x00 0xAA 0x55. Otherwise, there is no response from the target.

**‘W’** (0x57 in hex): the command is used to program a block of the flash in the target MCU. Based on the Kinetis memory map, the address is 24 bits. Addr2 contains the highest bits and Addr0 contains the lowest bits. LEN is the data length to be programmed and follows the data to be programmed. After executing the command successfully, the MCU responds a zero length frame. I.e., 0x24 0x01 0x00 0x00 0xAA 0x55. Otherwise, there is no response from the target.

**‘R’** (0x52 in hex): This command is used to read a block of flash from the target MCU. Based on the Kinetis memory map, the address is 24 bits. Addr2 contains the highest bits, and Addr0 contains the lowest bits. LEN is the data length to be read. After executing the command successfully, the MCU responds a Bootloader Response Frame beginning with LEN (the length read data) and following the read data.

**NOTE**

When using the 24-bit address, the supported address range is 0x000000 to 0xFFFFFFFF (16 M).

### 3. Memory relocation and code implementation

#### 3.1. Memory and vector relocation

The left side of the below figure is the default memory map of an MCU. The example is based on MKL26Z128. On the right side of the figure is relocated memory and vectors. In this example, the flash range from 0x00000000 to 0x00000FFF is protected. In this protected area, 0x00000000 to 0x000000C0 are the original vectors. The range from 0x00000410 to 0x00000FFF is for the bootloader code.

The unprotected flash from 0x00001000 to 0x00001FFFF is for the user code and can be updated in the system.

The reset vector is 0x00000000, and it is the entry of bootloader code. When exiting the bootloader code, it goes to the user application code by setting the Vector Table Offset Register (VTOR) and loading the new SP and PC values.

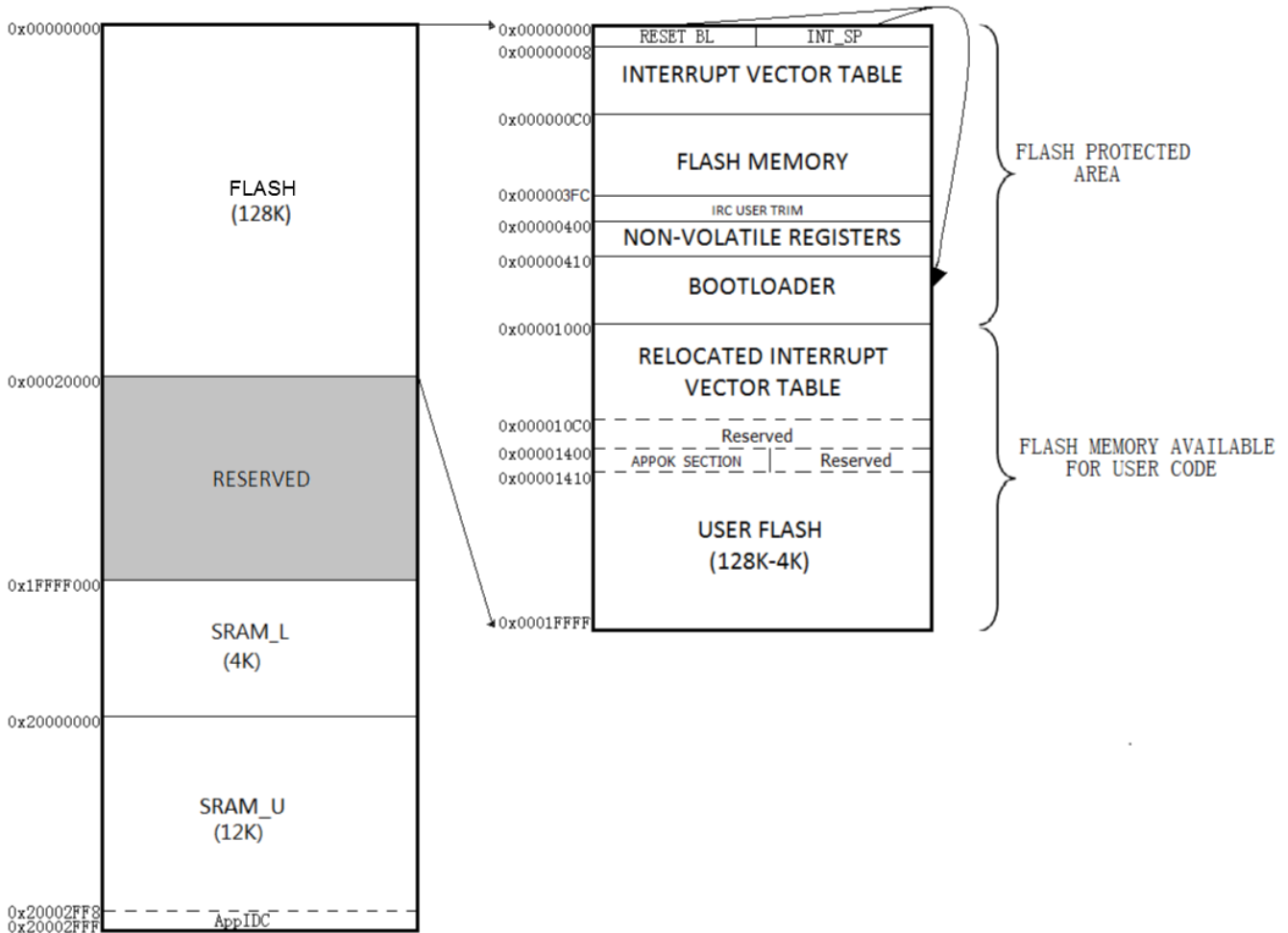


Figure 3. Memory allocation and vector relocation

When bootloader code runs, it uses the whole RAM space except the highest 8 bytes of RAM memory. For the KL26Z128 part, the address of the 8 bytes is 0x20002FF8-0x20002FFF. The stack pointer points to (RAM end - 8) but not the RAM end. The end 8 bytes of non-initialized RAM memory is reserved for application instruction to program or verify (see Section 3.3.3, “Bootloader Linker File Revision”). When the program switches from bootloader to application, all RAM memory space (except the end 8 byte address) is released to the user code. Both the user code and bootloader can only access 0x20002FF8-0x20002FFF by the pointer.

### 3.2. Bootloader flowchart

The on-chip flash programming routines simplify the bootloader and improve memory usage. The communication between the MCU and PC uses UART.

The following flowchart shows the basic principle of the bootloader algorithm:

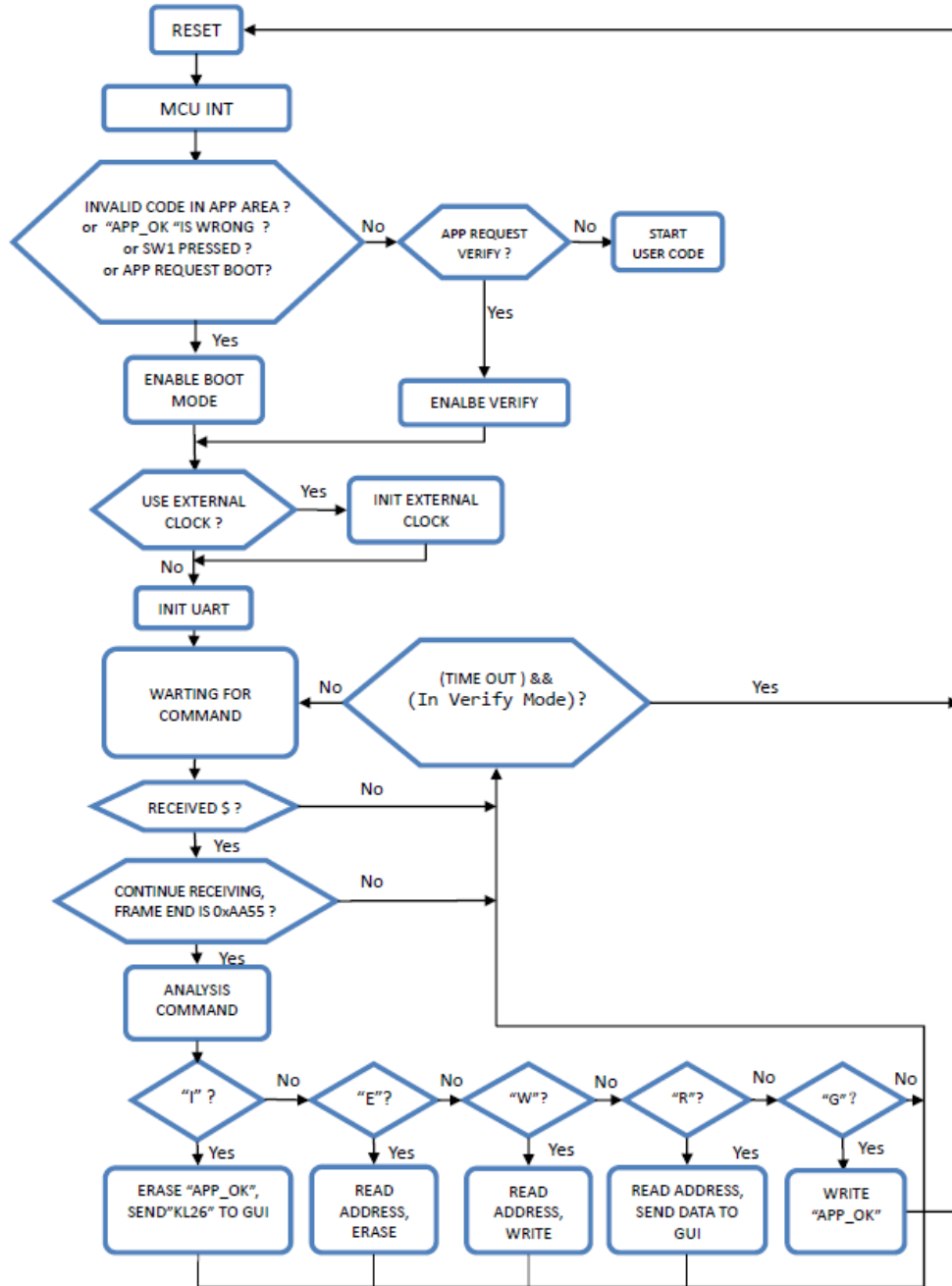


Figure 4. Bootloader flowchart



### 3.3. Bootloader implementation

The bootloader implementation is based on CodeWarrior v10.6. Additionally, this project is also ported to Kinetis Design Studio (KSD) IDE v3.0, Keil v5.15 and IAR v7.40. The user can locate these in the bootloader software package. The application demo projects are also provided.

#### 3.3.1. Bootloader clock initialization

The Kinetis L family has an Internal Reference Clock (IRC) Module which allows effective implementation of the bootloader without a crystal. In the bootloader, the user can choose either a default internal clock or external clock.

When using the default internal clock, the FEI mode is selected and the UART0 source is MCGFLLCLK. In the bootloader code, it is set to the same mode as the core clock.

```
//if use the default internal clock, FEI mode
#define BOOT_CORE_CLOCK      (32768*640)
#define BOOT_BUS_CLOCK      (32768*640)
#define BOOT_UART_BAUD_RATE  9600
```

Figure 5. Default internal clock

When using external crystal in bootloader demo, enabling macro USE\_EXTERNAL\_CLOCK can enable PEE with an 8 M crystal, 48 MHz core clock, and 24 MHz bus clock. The UART0 clock source is MCGPLLCLK/2. In the bootloader code, it is set to the same mode as the core clock.

```
//#define USE_EXTERNAL_CLOCK
#ifndef USE_EXTERNAL_CLOCK
// if use external clock
#define BOOT_CORE_CLOCK      (48000000)
#define BOOT_BUS_CLOCK      (24000000)
#define BOOT_UART_BAUD_RATE  9600
```

Figure 6. External clock

#### 3.3.2. Bootloader flash program routine

Using a Kinetis Cortex-M0+ core, the Platform Control Register (MCM\_PLACR) is added. The MCM\_PLACR register selects the arbitration policy for the crossbar masters and configures the flash memory controller. Enabling the ESFC bit can stall the flash controller when flash is busy. Setting the ESFC bit can well-balance the time sequence of flash reading and writing. When writing flash, reading the flash instructions can wait, and vice versa. Using the ESFC bit can make flash programming easier. Therefore, one flash can write itself, which is not possible for the other one flash MCU without ESFC bit control.

The following is the flash initialization example for ARM Cortex-M0+. The ESFC bit is easy to be set in C code.

```
#define MCM_PLACR_ESFC_MASK 0x10000u
*****/
void FLASH_Initialization(void)
{
    MCM_PLACR |= MCM_PLACR_ESFC_MASK; // 0xF000300C, PLACR,
|
}
```

Figure 7. ESFC bit in C code

For ARM Cortex-M0+ products, the user does not need to copy the flash operating routines to RAM. For the ARM Cortex-M4 core Kinetis, the CopytoRam code must be used for the flash programming routine. Therefore, the flash programming routine is executed from RAM but not flash.

The following is an example for flash initialization in ARM Cortex-M4.

```
Byte buffer[200]={0};
void FLASH_Initialization(void)
{
    LWord i;
    // copy flash code from flash to ram
    for(i=0;i<200;i++_
        buffer[i] = ((Byte*)FLASH_FlashCommandSequenceStart)[i-1];
}
```

### 3.3.3. Bootloader linker file revision

According to the memory allocation and vector relocation, the linker file should be revised. The 8 highest bytes should be allocated to RAM memory. For KL26Z128, the address 0x20002FF8-0x20002FFF is reserved without definition in the linker file. Therefore, it can be prevented from being initialized. The modification is highlighted in the figure below.

```
ENTRY(__thumb_startup)

/* Highest address of the user mode stack */
/* _estack = 0x20003000; */ /* end of SRAM */
_estack = 0x20003000 - 8; /* end of SRAM, revised for bootloader */
__SP_INIT = _estack;

/* Generate a link error if heap and stack don't fit into RAM */
__heap_size = 0x400; /* required amount of heap */
__stack_size = 0x400; /* required amount of stack */

/* Specify the memory areas */
MEMORY
{
    m_interrupts (rx) : ORIGIN = 0x00000000, LENGTH = 0xC0
    m_cfmprotrom (rx) : ORIGIN = 0x00000400, LENGTH = 0x10
    /*m_text (rx) : ORIGIN = 0x00000800, LENGTH = 128K - 0x800 */
    m_text (rx) : ORIGIN = 0x00000410, LENGTH = 128K - 0x410 /* revised for bootloader */
    /*m_data (rwx) : ORIGIN = 0x1FFFF000, LENGTH = 16K */ /* SRAM */
    m_data (rwx) : ORIGIN = 0x1FFFF000, LENGTH = 16K - 8 /* SRAM, revised for bootloader */
}
```

Figure 8. Bootloader linker file configuration

### 3.3.4. “Program” and “Verify” indicator

Although the linker file does not define 0x20002FF8-0x20002FFF, the code can still access it via pointer. In the bootloader demo code, the pointer to access it is defined as AppIDC.

```
// 8Byte RAM memory. 0x0000000B:app requests program. 0x0000000A:app requests verify.
#define AppIDC                *((LWord*)(0x20003000 - 8))
```

Figure 9. Program and verification indicator

The GUI is for sending either “program” or “verify” instructions to the target MCU. When the target receives a “program” instruction, it writes 0x0000000B to AppIDC, then forces a system reset and the bootloader goes into boot mode. When the target receives a “verify” instruction, it writes 0x0000000A to AppIDC, then forces a system reset and the bootloader goes into verify mode.

```
//condition for entering boot:
if(((unsigned long*)(RELOCATED_VECTORS + 8)) == 0xffffffff) //1. no valid code in APP vector section,
| | Boot_StrCompare(( Byte*)APPOK_START_ADDRESS, str_app_ok, APPOK_LENGTH) == CHECK_FAIL //2."APP_OK" is wrong in address APPOK_START_ADDRESS.
| | ((GPIO_PDIR_REG(BOOT_PIN_ENABLE_GPIO_BASE) & (1 << BOOT_PIN_ENABLE_NUM)) == 0) //3. SW1 is pressed
| | (AppIDC == 0x0000000B) //4. App request boot
{
    enableBootMode = 1; // enable boot
    BOOT_LED_ON;
    AppIDC = 0;
}
else if (AppIDC == 0x0000000A) // App request verify
{
    enableBootMode = 2; // enable verify mode
    BOOT_LED_ON;
    AppIDC = 0;
}
```

Figure 10. Bootloader verify mode

### 3.3.5. Application completeness symbol

Flash memory 0x00001400 - 0x00001407 for KL26Z128 is defined as the application code completeness symbol. The start address is defined as the following:

```
//store APP_OK: 0x1400 - 0x1407
#define APPOK_START_ADDRESS    0x1400
```

- After successfully updating an application in the bootloader, the address is filled as "APP\_OK". It erases when entering boot mode. If the updating process stops, or there is a power down event during the upload procedure, the new application is not successfully programmed, but the old application will be corrupted. When powering on the target, it goes directly to the bootloader because "APP\_OK" is not detected.
- In the user application, “APP\_OK” is initialized as a const string at 0x00001400:

```
//fill this field with "APP_OK", indicate app is successfully programmed
__attribute__((section(".cfmconfig")))const Byte str_app_ok[8] = "APP_OK";
```

In the case where the user is burning the combined bootloader and application together with the target, “APP\_OK” can also be programmed. The next time the user is powering on the target, the bootloader program detects that “APP\_OK” exists, and jumps to the application code.

In the GUI, the following are steps for handling the “APP\_OK” symbol:

1. The GUI reads out the application burning file into an array.
2. The GUI removes the content of “APP\_OK” at APPOK\_START\_ADDRESS from the array.
3. The GUI burns Step 2-generated array to the target.
4. The GUI burns “APP\_OK” to the target at APPOK\_START\_ADDRESS

The above steps guarantee that “APP\_OK” is only programmed after a full application code is programmed.

### 3.3.6. Flash protection

In bootloader, the flash range from 0x00000000 to 0x00000FFF is protected. The protection is defined in bootloader code:

```
// Protect flash memory 0-0x0FFF for bootloader area, NMI pin disabled, Flash unsecure
// User can change the configuration according to the application requirements
__attribute__((section(".cfmconfig"))) const FlashConfig_t Config __attribute__((used)) =
{
  {0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFBFE}
};
```

Here, bootloader regions (0x00000000 to 0x00000FFF) within the flash memory are protected from program and erase operations. Protection is controlled by NVM Flash Configuration Field (at address 0x00000408 - 0x0000040B).

Each bit of this field protects a 1/32 region of the program flash memory except for memory configurations with less than 32 KB of program flash, where each assigned bit protects 1 KB.

In the user application code, when powering on, the content of NVM Flash Configuration Field is loaded into the flash option register (FTFA\_FPROTn) at 0x40020010 to 0x40020013.

#### NOTE

The NVM Flash Configuration Field is configured only to protect from 0x00000000 to 0x00000FFF in the bootloader. If application needs all FLASH memory be protected, FTFA\_FPROTn need to be set as 0.

The following code can be used to protect all flash memory:

```
void flash_protect()
{
  FTFA_FPROT0 = 0x00;
  FTFA_FPROT1 = 0x00;
  FTFA_FPROT2 = 0x00;
  FTFA_FPROT3 = 0x00;
}
```

### 3.3.7. Watchdog

In the Kinetis device example, KL26Z128\_SIM\_COPC is a write-once register after reset.

Address: 4004\_7000h base + 1100h offset = 4004\_8100h

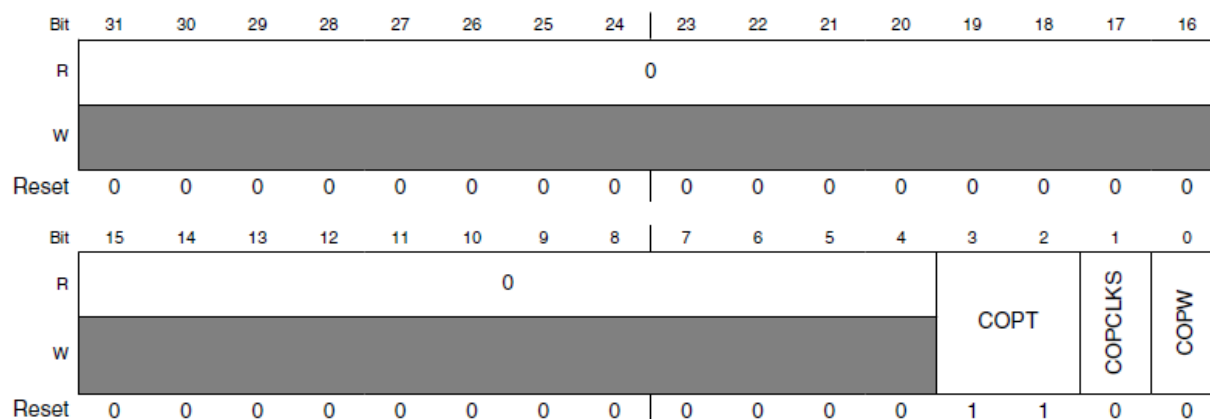


Figure 1. KL26Z128\_SIM\_COPC

The following lists the description of the COPT[3:2] bits:

- 00 – COP disabled.
- 01 – COP timeout after 25 LPO cycles or 213 bus clock cycles.
- 10 – COP timeout after 28 LPO cycles or 216 bus clock cycles.
- 11 – COP timeout after 210 LPO cycles or 218 bus clock cycles.

The COPT bits in SIM\_COPC are 1:1 after reset. If COPT bits are set with another value in the bootloader source code, the user cannot change COPT again in the user application code, because SIM\_COPC is a write-once register.

In some IDEs for example CodeWarrior v10.6, the default generated startup code disables the watchdog by manually setting the SIM\_COPC register. Therefore, any other change of SIM\_COPC in the user application code cannot be successfully performed again. The following is hardware initialization code in the CodeWarrior auto generated Kinetis\_sysinit.c file.

```
void __init_hardware()
{
    SCB_VTOR = (uint32_t)__vector_table; /* Set the interrupt vector table position */

    // Disable the Watchdog because it may reset the core before entering main().
    SIM_COPC = KINETIS_WDOG_DISABLED_CTRL;
}
```

In the bootloader implementation, because of the above reason, it is not suggested to change the default watchdog settings. If the default initialization code has completed it, it is suggested that the user comment it out.

```

void __init_hardware()
{
    SCB_VTOR = (uint32_t)__vector_table; /* Set the interrupt vector table position */

    // Disable the Watchdog because it may reset the core before entering main().
    //SIM_COPC = KINETIS_WDOG_DISABLED_CTRL;
}

```

Then, set `BOOTLOADER_INT_WATCHDOG` as 1 in `bootloader.h` to enable the watchdog feed.

```
#define BOOTLOADER_INT_WATCHDOG 1
```

### 3.3.8. User application code

The user can add bootloader communication code to a customized user application project with the following steps. The code has passed the test under CodeWarrior v10.6.

1. Revise the application linker file for vector relocation, and the last 8 bytes of RAM memory reservation:

```

_estack = 0x20003000 - 8; /* end of SRAM, revised for bootloader */
__SP_INIT = _estack;

/* Generate a link error if heap and stack don't fit into RAM */
__heap_size = 0x400; /* required amount of heap */
__stack_size = 0x400; /* required amount of stack */

/* Specify the memory areas */
MEMORY
{
    /* m_interrupts (rx) : ORIGIN = 0x00000000, LENGTH = 0xC0 */
    /* m_cfmprotrom (rx) : ORIGIN = 0x00000400, LENGTH = 0x10 */
    /* m_text (rx) : ORIGIN = 0x00000800, LENGTH = 128K - 0x800 */
    /* m_data (rwx) : ORIGIN = 0x1FFFF000, LENGTH = 16K */ /* SRAM */
    m_interrupts (rx) : ORIGIN = 0x00001000, LENGTH = 0xC0 /* revised for bootloader */
    m_cfmprotrom (rx) : ORIGIN = 0x00001400, LENGTH = 0x10
    m_text (rx) : ORIGIN = 0x00001410, LENGTH = 128K - 0x1410 /* revised for bootloader */
    m_data (rwx) : ORIGIN = 0x1FFFF000, LENGTH = 16K - 8 /* revised for bootloader */
}

```

2. Add `bl_communication.c/bl_communication.h` to the user application project.
3. Add the three functions to the user application file:

<pre> INIT_CLOCKS_TO_MODULES; // init clock module UART_Initialization(); // init UART module </pre>
<pre> while(1) {     UpdateAPP(); // update user's application } </pre>

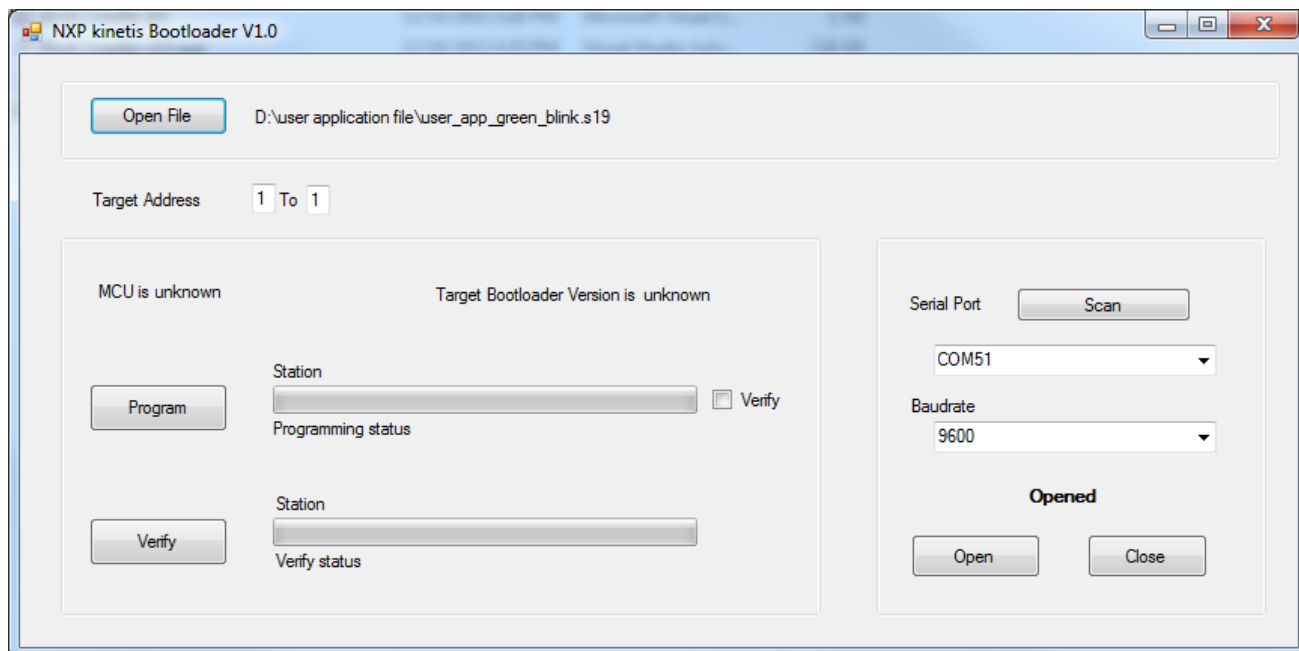
Now the user can use the bootloader to update the user application. Steps 2 and 3 enable the user application code to receive bootloader commands, and enter boot or verify mode. If this is not needed, then only Step 1 is required.

## 4. Communication between GUI and Target

The graphics user interface on the PC side is written using Microsoft Visual Express® 2013. It is a free edition of the program which can be downloaded from the Microsoft® website.

The GUI is compatible with 32/64-bit Windows® OS 7. For Windows® OS XP, Microsoft .NET Framework 4.0 needs to be installed. It can be downloaded from the Microsoft website, [www.microsoft.com/](http://www.microsoft.com/).

The below figure displays the GUI.



**Figure 11. Graphics User Interface (GUI)**

When clicking the “Program” button, the opened file will be programmed to the target nodes one-by-one, as defined in the Target Address. During implementation, the Target Address is limited to 1 to 32 (The user can revise the bootloader and GUI to expand the address range). After checking the Verify box and programming one block of flash, the contents are read back and compared with the S19 file. While in the programming process, the user should be able to terminate the process anytime.

The “Verify” button is used to compare a selected S19 file with target nodes one by one. It does not change the contents in the target MCUs.

Note that both program and verify commands will stop the target node from running. A ‘G’ command is issued automatically after the process successfully finishes, which forces the target node to run.

Remember to open the S19 file and open the COM port first. In the boot mode, the LED on the FRDM-KL26 Freescale Freedom development board is blue.

### 4.1. Program execution – Program without verify

The following flowchart shows what happens when clicking the “Program” button when “Verify” is not checked.

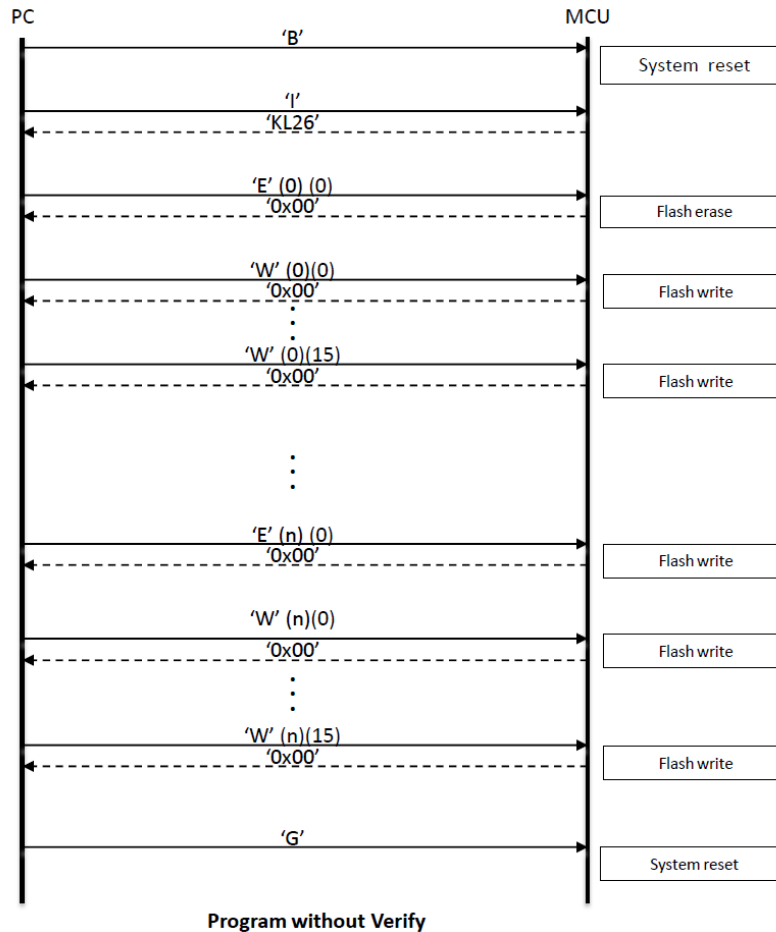


Figure 12. GUI communicates with Target – program without verify

1. The GUI sends a ‘B’ command to the target. If the target is in the running application, the target performs a system reset to the bootloader in program mode so it is ready for the next command.
2. The GUI sends an ‘I’ command, then the target erases APP\_OK and answers the GUI with the target identification.
3. The GUI sends an ‘E’ command, and the target erases one block flash, size 0x400. If successful, the answer is 0x00.
4. The GUI sends 16 ‘W’ commands successively. Each command requests that a target writes 0x40 data to flash.
5. The GUI continues to perform step 3 and step 4 until the user application s19 file finishes programming.



- The GUI sends a 'G' command, and the target writes APP\_OK in flash, then performs a system reset and runs the application code.

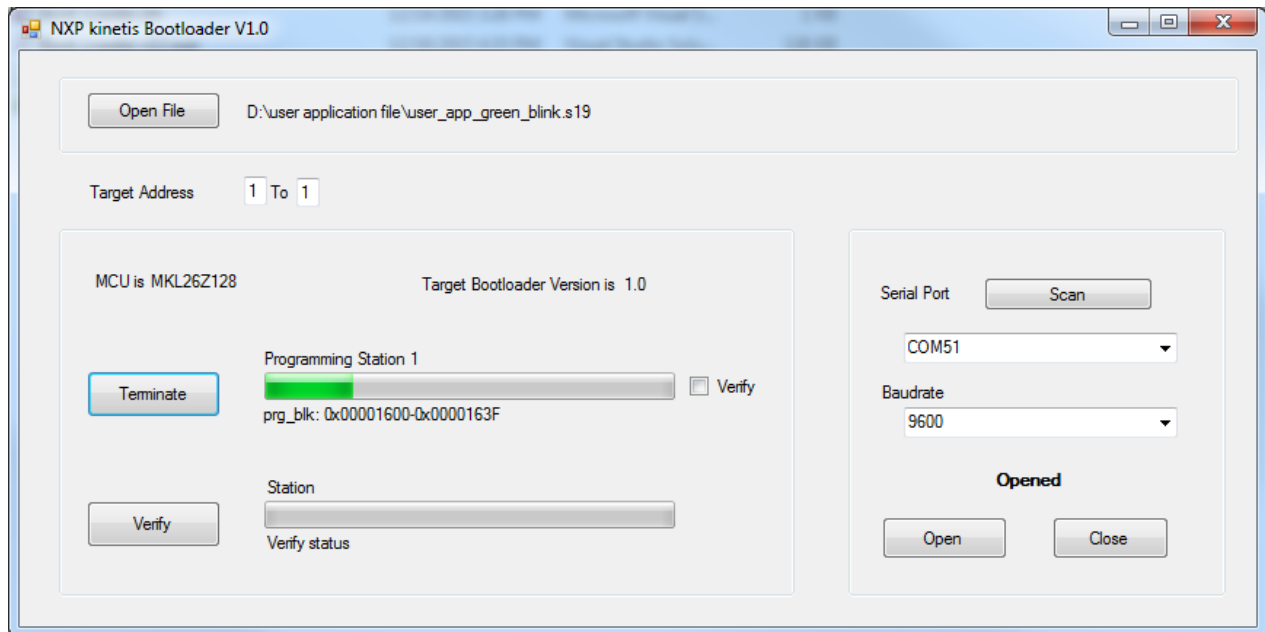


Figure 13. Running GUI – program without verify

## 4.2. Program execution - Program with verify

The flowchart below shows that when clicking the “Program” button, the “Verify” button is checked.

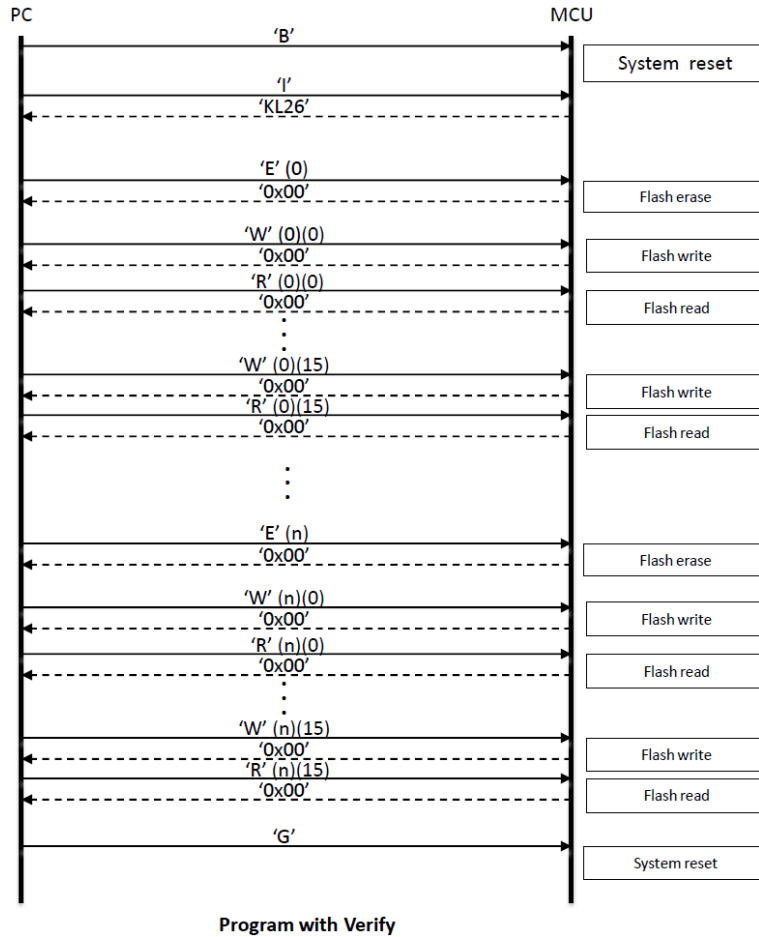


Figure 14. Running GUI – program with verify

1. The GUI sends a 'B' command to the target. If the target is in a running application, the target performs a system reset to the bootloader in program mode so it is ready for the next command.
2. The GUI sends an 'I' command, and the target erases APP\_OK and answers the GUI with the target identification.
3. The GUI sends an 'E' command, and the target erases one block of flash, size 0x400. If successful, it answers 0x00.
4. The GUI sends 16 'W' and 'R' commands successively. The 'W' command request target writes 0x40 data to flash, and the 'R' command reads data back 0x40 data from flash, then sends it to the GUI.
5. The GUI continues to perform step 3 and step 4 until the user application s19 file finishes programming.

- The GUI sends a ‘G’ command and the target writes APP\_OK in flash, then performs a system reset and runs the application code.

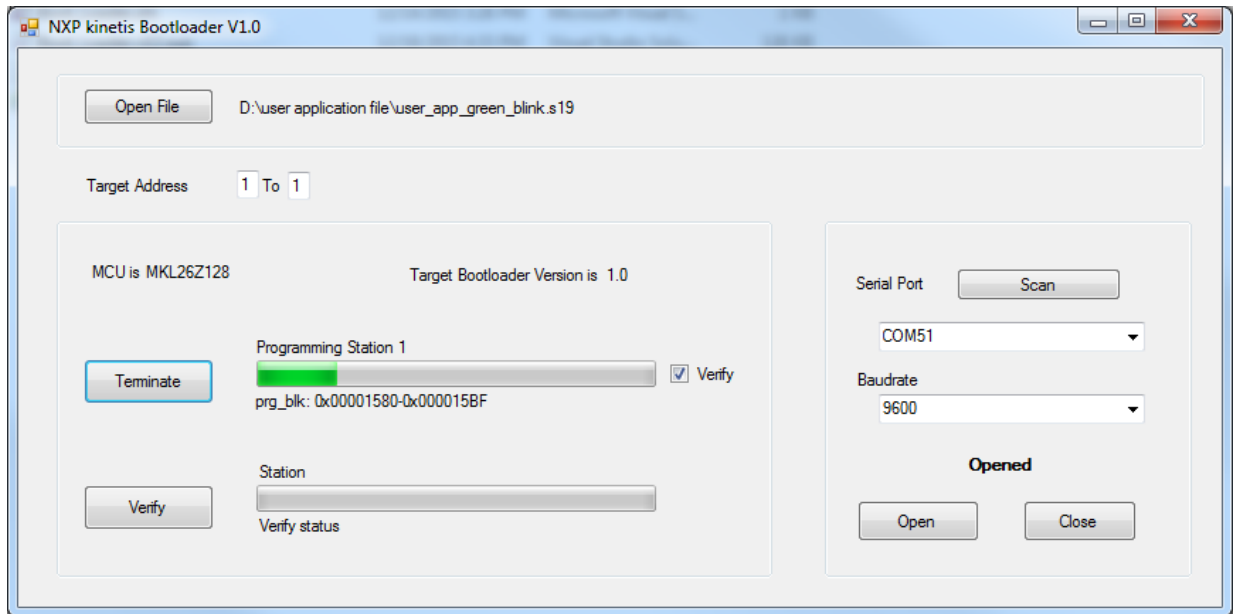


Figure 15. Running GUI – program with verification

### 4.3. Verify execution

The following flowchart determines when to click the “Verify” button.

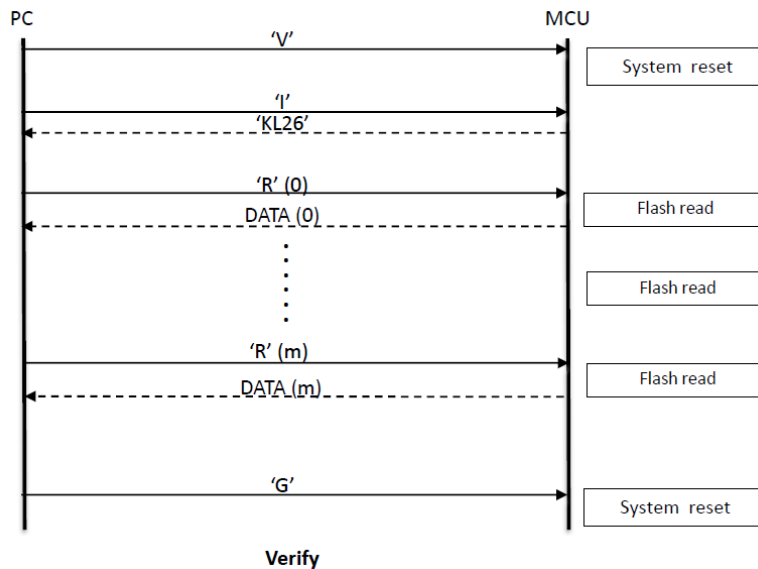


Figure 16. Running GUI – verify

- First, the GUI sends a ‘V’ command to the target. If the target is in the running application, the targets perform a system reset to the bootloader in verify mode so it is ready for the next command.

2. The GUI sends an 'I' command, and the target answers with the target identification.
3. The GUI sends an 'R' command and the target reads out a 0x40 byte, then answers it back to the GUI.
4. The GUI continues performing Step 3 until the user application s19 file finishes verification.
5. The GUI sends a 'G' command, then performs a system reset and runs the application code.

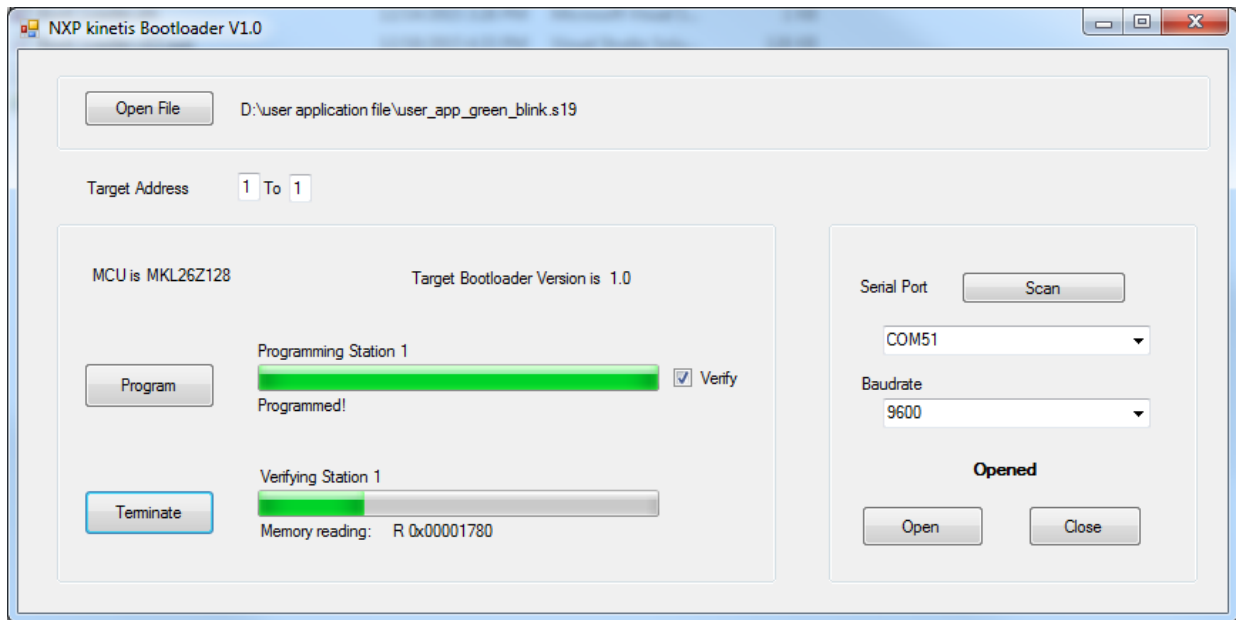
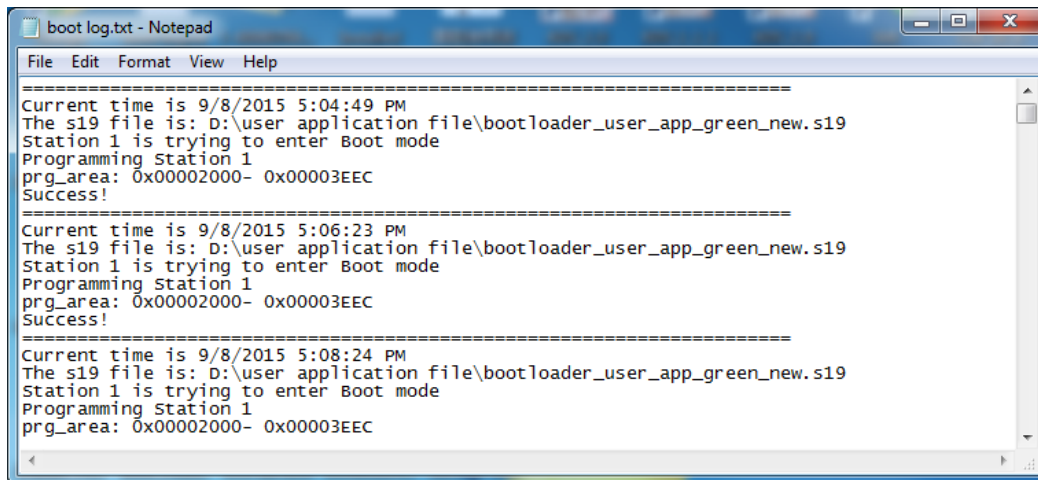


Figure 17. Running GUI – verification

## 4.4. Log file generation

A log file named boot log.txt is created in the same folder where the opened s19 file is stored. The operating history is recorded with time stamps.



```

boot log.txt - Notepad
File Edit Format View Help
=====
Current time is 9/8/2015 5:04:49 PM
The s19 file is: D:\user application file\bootloader_user_app_green_new.s19
Station 1 is trying to enter Boot mode
Programming Station 1
prg_area: 0x00002000- 0x00003EEC
Success!
=====
Current time is 9/8/2015 5:06:23 PM
The s19 file is: D:\user application file\bootloader_user_app_green_new.s19
Station 1 is trying to enter Boot mode
Programming Station 1
prg_area: 0x00002000- 0x00003EEC
Success!
=====
Current time is 9/8/2015 5:08:24 PM
The s19 file is: D:\user application file\bootloader_user_app_green_new.s19
Station 1 is trying to enter Boot mode
Programming Station 1
prg_area: 0x00002000- 0x00003EEC

```

Figure 18. Boot log file

## 5. Conclusion

This application note along with the MKL26Z128 device provides a detailed description regarding how to implement a bootloader. The bootloader and application code are separate projects. The user can use the user application S19 file for mass production and upgrading. A single user application S19 record file can be used for both mass production and in-system firmware upgrading. The application can be used for a single control unit, or to control units connected by networks. The firmware and GUI are the FRDM-KL26 Freescale Freedom development platform. Additionally, the codes can be easily migrated to other chips.

## 6. References

- *HCS08 Bootloader to Update Multiple Devices in a Field Bus Network* (document [AN4440](#))
- *Developer's Serial Bootloader MCUs* (document [AN2295](#))
- KL26 Sub-Family Reference Manual
- [www.visualstudio.com/en-US/products/visual-studio-express-vs](http://www.visualstudio.com/en-US/products/visual-studio-express-vs)

## 7. Revision history

This table summarizes revisions to this document.

<b>Table 2. Revision history</b>		
<b>Revision number</b>	<b>Date</b>	<b>Substantive changes</b>
0	01/2016	Initial release



---

**How to Reach Us:**

**Home Page:**  
[freescale.com](http://freescale.com)

**Web Support:**  
[freescale.com/support](http://freescale.com/support)

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: [freescale.com/SalesTermsandConditions](http://freescale.com/SalesTermsandConditions).

Freescale, the Freescale logo, CodeWarrior, and Kinetis are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off.

ARM and Cortex are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.

© 2016 Freescale Semiconductor, Inc.

Document Number: AN5204  
Rev. 0  
01/2016

