Kinetis Bootloader to Update Multiple Devices in a Field Bus Network

by Jennie Zhang, Alice Yang

Contents

1. Introduction

This application note describes how to do in-system reprogramming of Kinetis devices using standard communication media such as SCI. Most of the codes are written in C so that make it easy to migrate to other MCUs. GUI is also provided. The solution has been adopted by customers.

This bootloader is based on FRDM-KL26 demo board and Codewarrior 10.6. The bootloader and user application source codes are provided. Customer can make their own bootloader applications based on them. The application can be used to upgrade 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 upgrades firmware nodes one by one automatically.

The bootloader and application code are written in separated projects. For mass production, users can program the bootloader by tools such as CycloneMAX. Then boot the application code

through UART. Another choice is to combine the bootloader and application s19 file in one file and program it. Combination can be done by copying <user_app>.s19 to the end of <boot_loader>.s19.
Please note after combination records started by s7 s8 or s9 in boot.s19 section should be deleted. Because these records will be recognized as the end of the file.

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

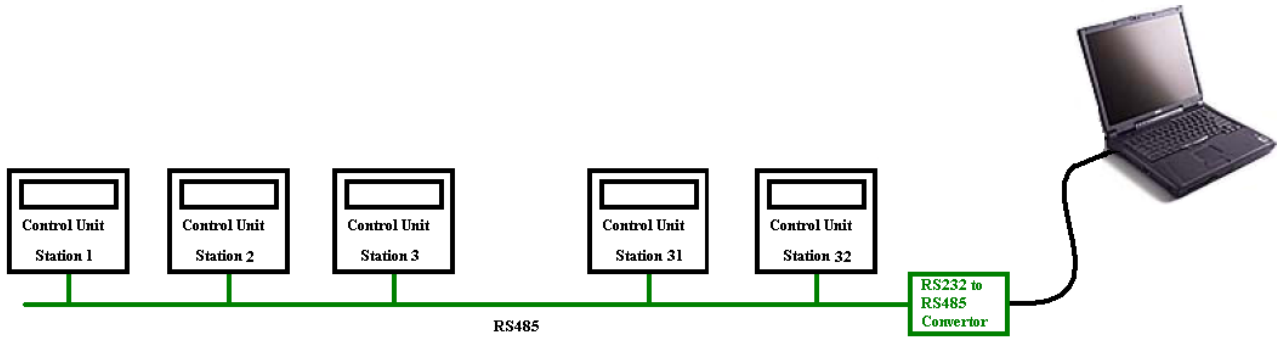

Figure 1. A typical Use Case

Key features of the bootloader:

- Able to update (or just verify) multiple devices in a network.
- Application code and bootloader code are in separated projects, convenient for mass production and firmware upgrading.
- Bootloader code size is small, only around 2K, which reduces on chip memory resources.
- Source code available, easy for reading and migrating.

2. Bootloader Frame Protocol

2.1. Bootloader Frame Protocal overview

Bootloader Frames are used for communication between PC and target control units.
The frame starts with 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 the frames use the same node address.

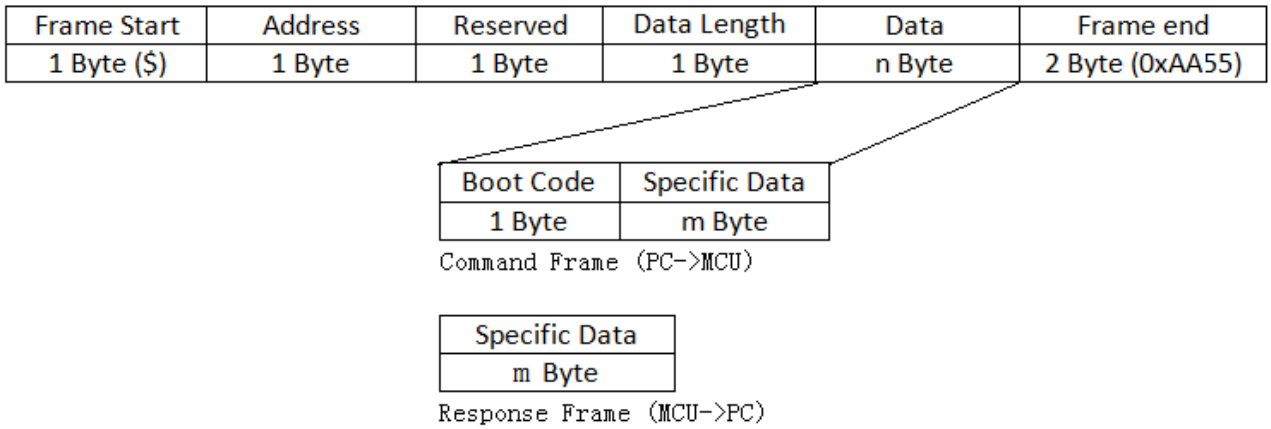

Figure 2. Boot Frame

- **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 PC) put a target node's address in the Address field as to communicate with it. The node also put 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.
- **Reserved:** User can customize it based on user's specific requirement.
- **Data Length:** for command frame: Data Length = Boot Code length + Specific Data length. For response frame: Data Length = Specific Data length.
- **Data:** when the bootloader frame is from PC to target MCU, these data bytes are command Frame. When the bootloader frame is from target MCU to PC, these data bytes are response Frame. See Table 1. Command and Response Frame.
- **Frame End:** This two bytes field is always filled with 0xAA55 as the end of frame.
- **Boot Code:** see Table 1. Command and Response Frame
- **Specific Data:** see Table 1. Command and Response Frame

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

NO.	Frame type	Data Length	Boot Code	Specific Data	Function
1	command	1	**'B'**	n/a	Force target to enter Bootloader mode
	response	No response. program jumps to bootloader immediately			
2	command	1	**'V'**	n/a	Force target to enter Verification mode
	response	No response. program jumps to bootloader immediately			
3	command	1	**'I'**	n/a	Identification of target MCU
	response	target responds a string of MCU identification.			
4	command	1	**'G'**	n/a	Force target to run
	response	No response. program jumps to user application code immediately			
5	command	4	**'E'**	Addr2 Addr1 Addr0	Force target erase specific FLASH block
	response	If target executes command successfully, it responds a 0 length frame. For example: 0x24 0x01 0x00 **0x00** 0xAA 0x55. Otherwise, no response			
6	command	5+LEN	**'W'**	Addr2 Addr1 Addr0 LEN Data	Force target Program specific FLASH block
	response	If target executes command successfully, it responds a 0 length frame. For example: 0x24 0x01 0x00 **0x00** 0xAA 0x55. Otherwise, no response			
7	command	5	**'R'**	Addr2 Addr1 Addr0 LEN	Force target Read specific FLASH block
	response	LEN	n/a	LEN Data	Send specific FLASH block content to PC

Table 1. Command and Response Frame

Some of the Boot Codes are derived from FC protocol commands in AN2295 and AN4440. More detailed descriptions of the Boot Codes:

'B' (0x42 in hex): the command forces the target to enter Bootloader mode. It does not wait for a response. A command 'I' is followed to check whether the target MCU is in Bootloader mode. "APP_OK" at the last 8 bytes unprotected FLASH memory will be erased. So if there is a power down event during the bootload procedure, the MCU will be forced into Bootloader mode in the next power up.

'V' (0x56 in hex): the command forces the target to enter Verification mode. It does not wait for a response. A command 'I' is followed to check whether the target MCU is in Verification mode. In Verification mode, it will only compare the contents in the target MCU with specific S19 file. Nothing will be changed in the target MCU.

'I' (0x49 in hex): after receiving 'I' command the MCU will respond a string of MCU identification from bootloader.
An example of Response Frame to command 'I':

Data Length	Specific Data
0x09	0x4D 0x4B 0x4C 0x32 0x36 0x5A 0x31 0x32 0x38

The identification string is "MKL26Z128".

'G' (0x47 in hex): after receiving the 'G' command the MCU will exit bootloader and force the user application code to run. It does not respond a frame. The user can use application software to make sure that the target MCU is running the user code.

'E' (0x45 in hex): the command is used to erase a block of the FLASH in the target MCU. Based on 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 a zero length frame. For example: 0x24 0x01 0x00 0x00 0xAA 0x55. Otherwise, no response from target.

'W' (0x57 in hex): the command is used to program a block of the FLASH in the target MCU. Based on 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. For example: 0x24 0x01 0x00 0x00 0xAA 0x55. Otherwise, no response from target.

'R' (0x52 in hex): the command is used to read a block of FLASH from the target MCU. Based on 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 begins with LEN (the length read data) and follows the read data.

Note: as we use 24 bit address, the supported address range is 0x000000 to 0xFFFFFF(16M).

3. Memory Relocation and Code Implementation

3.1. Memory and Vectors Relocation

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

The unprotected FLASH from 0x00001000 to 0x0001FFFF is for user code and can be updated in system.

The reset vector is 0x00000000. It is the entry of bootloader code. When exit the bootloader code, It goes to user application code by setting Vector Table Offset Register (VTOR) and then load the new SP and PC vlues.

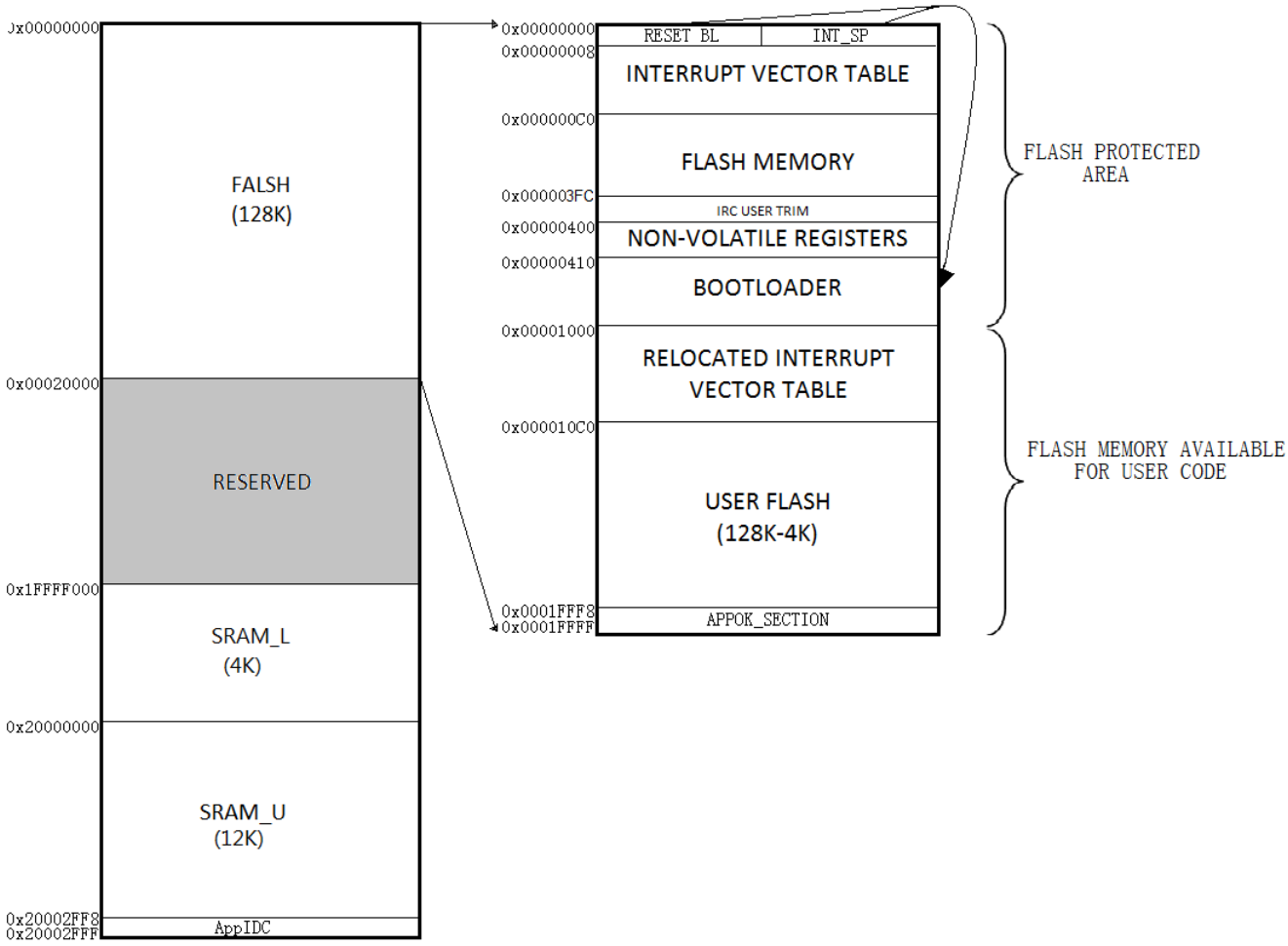

Figure 3. Memory Allocation and Vector Relocation

When bootloader code runs, it uses the whole RAM space except the highest 8 bytes RAM memory, for KL26Z128 the address is 0x20002FF8-0x20003000. Stack pointer points to (RAM end - 8) but not RAM end. The end 8 bytes non-initialized RAM memory is reserved for application instruction – program or verify (for detail, see section 3.3.3 Bootloader Linker File Revision). Local variables in bootloader code are allocated on the stack. The stack will be reinitialized by the startup code when user code runs. Then all RAM memory space (except end 8 byte address) will be released to the user code. Both user code and bootloader can only access 0x20002FF8-0x20003000 by 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 usesUART.
The following flowchart shows the basic principle of the bootloader algorithm:

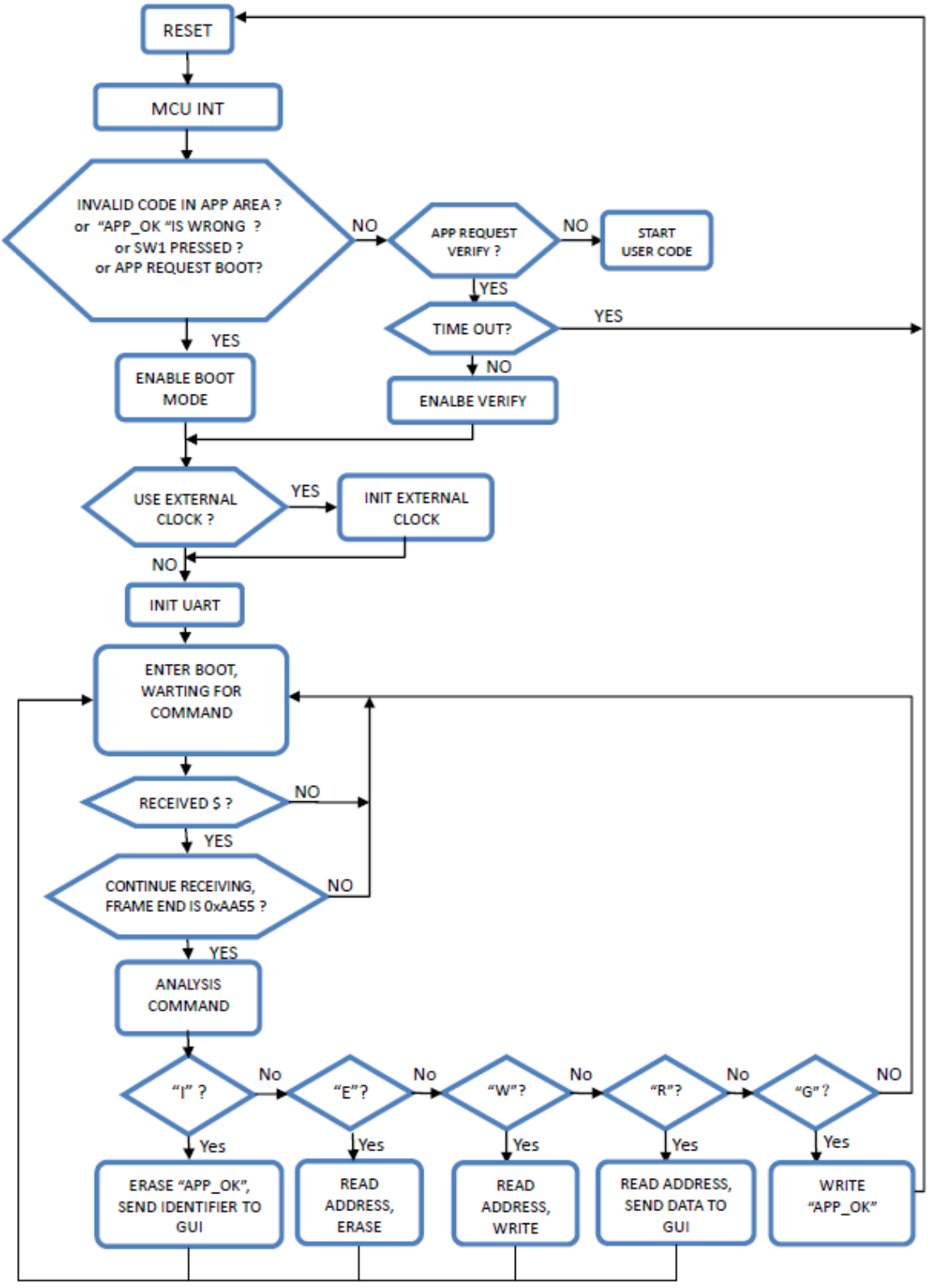

Figure 3. Bootloader Flowchart

3.3. Bootloader Implementation

3.3.1. Bootloader Clock initialization

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

When use default internal clock, FEI mode is selected. UART0 source is MCGFLLCLK. In the bootloader code it is set to the same 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
```

When use external crystal in bootloader demo, enabling macro USE_EXTERNAL_CLOCK can enable PEE with 8M crystal, with 48MHz Core clock and 24MHz Bus clock . UART0 clock source is MCGPLLCLK/2. In the bootloader code it is set to the same as the core clock.

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

3.3.2. Bootloader Flash Program Routine

In Cotex M0+ core kinetis, 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 ESFC bit can stall flash controller when flash is busy. Setting ESFC bit can well-balance time sequence of Flash reading and writing – when writing Flash, reading Flash instruction can wait, and vice versa. Using ESFC bit can make our flash programming easier. Thus one Flash can write itself, which is not possible for other one Flash MCU without ESFC bit control.

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,

}
```

For M0+ products, we do not need to copy the Flash operating routines to RAM. While for Contex M4 core Kinetis, CopytoRam code must be used for Flash Programming routine.

3.3.3. Bootloader Linker File Revision

According to the memory allocation and vector relocation, linker file should be revised. 8 higest bytes in RAM memory, for KL26Z128 address 0x20002FF8-0x20003000 is reserved without definition in linker file, thus it can be prevented from being initialized. The modification is highlighted as 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 */
}
```

Bootloader Linker File Configuration

3.3.4. “Program” and “Verify” indicator

Although linker file doesn’t define 0x20002FF8-0x20003000, code can still access it via pointer. In bootloader demo code, we define AppIDC as a pointer to access it.

```
// 8Byte RAM memory.  0x00000000B:app requests program. 0x00000000A:app requests verify.
#define AppIDC                      *((LWord*)(0x20003000 - 8))
```

GUI is for sending instruction to target MCU – “program” or “verify”. When target receives “program” instruction, it writes 0x0000000B to AppIDC and then force a system reset, bootloader goes into program mode. When target receives “verify” instruction, it writes 0x0000000A to AppIDC and then force a system reset, 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;
}
```

3.3.5. Application completeness Symbol

The highest 8 bytes Flash memory, for KL26Z128 0x1FFF8 - 0x1FFFF is defined as application code completeness Symbol. The start address is defined:

```
//store APP OK: 0x1FFF8 - 0x1FFFF
#define APPOK_START_ADDRESS        0x1FFF8
```

In bootloader, after a successful updating application, this address will be filled with "APP_OK". It will be erased when entering boot mode. If the updating process is stopped by some reason or there is a power down event during the uploading procedure, new application is not successfully programmed but old application was corrupt. In this case when power on the target, it will go to bootloader directly because "APP_OK" is not detected.

3.3.6. Flash Protection

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

```
// Protect flash memory 0-0x0FFF for bootloader erea, 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, 0xFFFFFFFE, 0xFFFFFBFE}
};
```

Here, bootloader regions (0x00000000 to 0x00000FFF) within the flash memory is 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 32Kbytes of program flash where each assigned bit protects 1 Kbyte.

In user application code, when power on, the content of NVM Flash Configuration Field are loaded into flash option register (FTFA_FPROTn) at 0x40020010 to 0x40020013.

Note NVM Flash Configuration Field is configured only protecting from 0x00000000 to 0x00000FFF in bootloader. If application needs all FLASH memory be protected, FTFA_FPROTn need to be set as 0.

Below code can be used to protect entire FLASH memory:

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

3.3.7. User application Linker File Revision

This is the sample of implementing KL26Z128 vector redirection under CW10.6 to allocate vector table from 0x00000000 to 0x00001000

```
_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_text        (rx) : ORIGIN = 0x000010C0, LENGTH = 128K - 0x10C0          /* revised for bootloader */
  m_data       (rwx) : ORIGIN = 0x1FFFF000, LENGTH = 16K - 8                /* revised for bootloader */
}
```

User Application Linker File Configuration

In CW10.6, If the project is created with project wizard, setting SCB_VTOR register is done in default generated kinetis_sysinit.c, __init_hardware() function.

```
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;
}
```

Here __vector_table is interrupt vector address which is defined in linker file *.ld memory allocation section.

```
SECTIONS
{
  /* The startup code goes first into Flash */
  .interrupts :
  {
    __vector_table = .;
    . = ALIGN(4);
    KEEP(*(.vectortable)) /* Startup code */
    . = ALIGN(4);
  } > m_interrupts
```

Below is the testing result in debugger: vector table is relocated to 0x1000. Interrupt can work well after vector relocation.

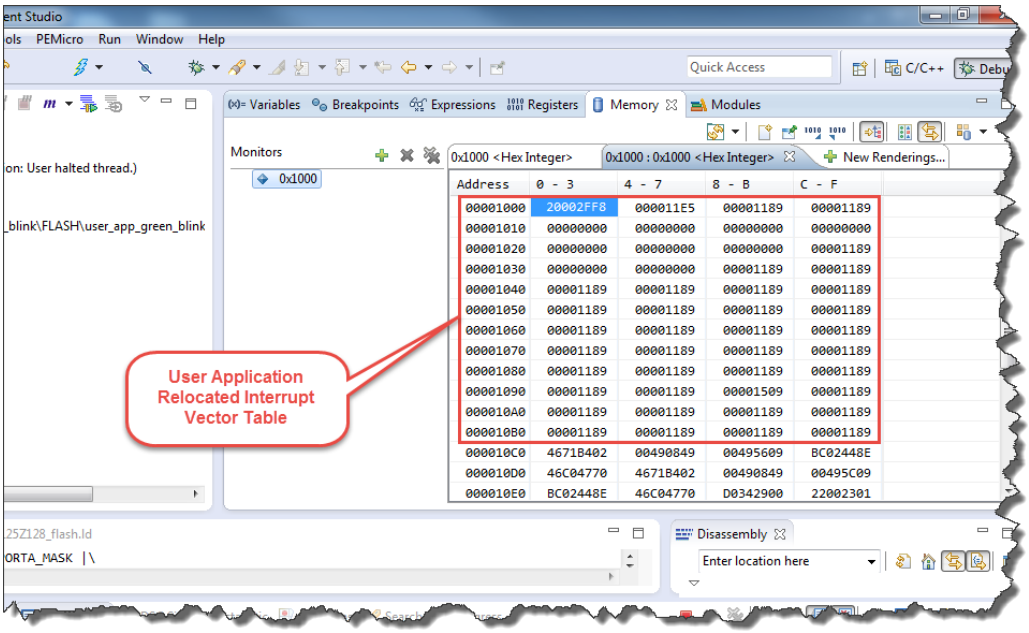

Figure 4. Relocated Vector Table in Debugger

3.3.8. Steps of Adding Bootloader Communication Code to User code

User can add bootloader communication code to customized user application project with below steps (code is passed test under CodeWarrior 10.6):

Step 1: Revise application linker file for vector relocation and last 8 bytes 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_text        (rx) : ORIGIN = 0x000010C0, LENGTH = 128K - 0x10C0        /* revised for bootloader */
  m_data       (rwx) : ORIGIN = 0x1FFFF000, LENGTH = 16K - 8              /* revised for bootloader */
}
```

Step 2: Add bl_communication.c/bl_communication.h to user application project.

Step 3: Add the three functions to user application file:

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

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

4. Communication between GUI and Target

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

GUI is compatible with 32/64bit windows 7. For windows XP, Microsoft .NET Framework 4.0 needs to be installed. It can be can downloaded from Microsoft website: http://www.microsoft.com/ .

This is what GUI looks like:

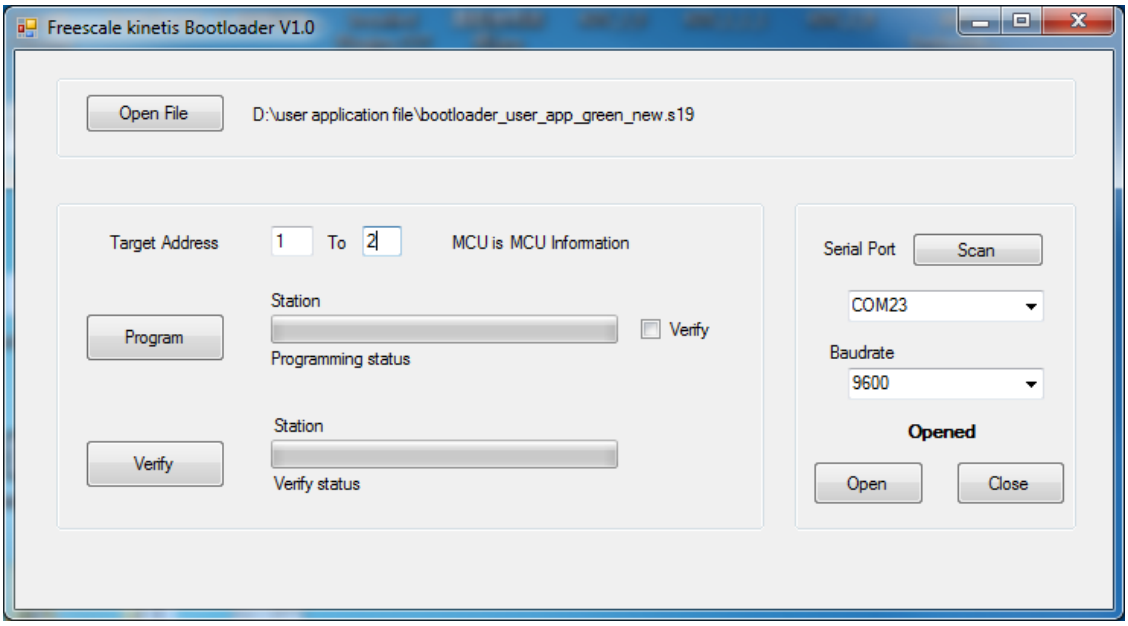

Figure 5. Graphics User Interface

When click Program button, the opened file will be programmed to the target nodes one by one as defined in the Target Address. In implementation, Target Address is limited to 1 to 32. If we check the check box Verify, after programming one block of FLASH, the contents will be read back and compared with the S19 file. While in programming process, we can terminate the process anytime.

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

Please note that both Program and Verify commands will stop the target node from running. A 'G' command will be issued automatically after the process finished successfully which force the target node to run.

Remember to open the S19 file and open the COM port first.

In boot mode, the LED on FRDM-KL26 board is in blue.

4.1. Program execution – Program without Verify

Here is the flowchart when click on “Program” button, “Verify” is not checked.

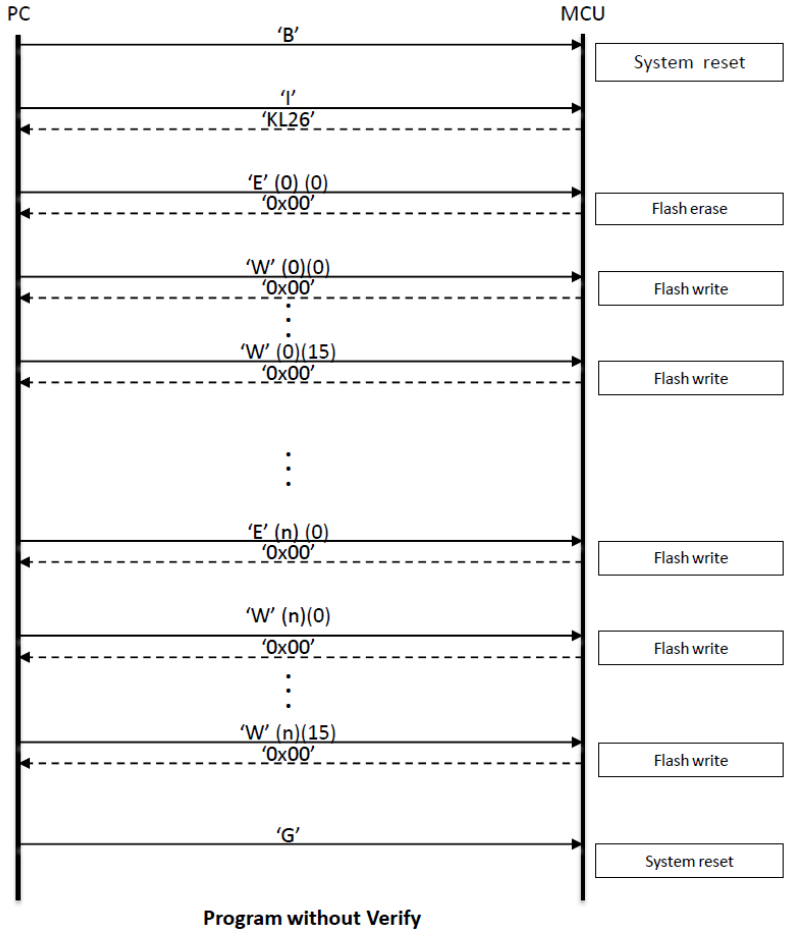

Figure 6 GUI communicates with Target – Program without Verify

Action 1: First, GUI sends ‘B’ command to target. If target is in running application, targets perform system reset to bootloader in program mode, ready for the next command.

Action 2: Next, GUI sends ‘I’ command, target erases APP_OK and answers GUI with the target identification.

Action 3: Next, GUI sends ‘E’ command, target erases one block Flash, size 0x400. If success, answer 0x00.

Action 4: Next, GUI sends 16 ‘W’ command successively. Each command request target writes 0x40 data to Flash.

Action 5: Next, GUI keeps on performing action3 and action4 until user application s19 file finished programming.

Action 6: Last, GUI sends ‘G’ command, target writes APP_OK in Flash. System reset, run application code.

Running GUI:

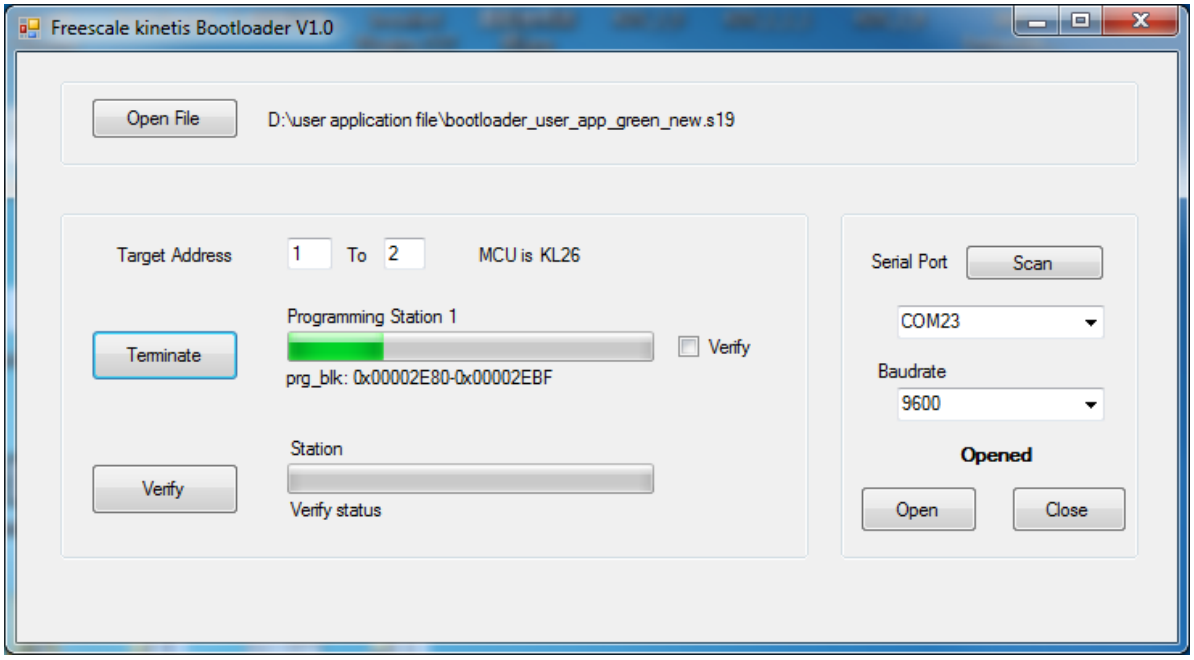

Figure 7. Running GUI – Program without Verify

4.2. Program execution - Program with Verify

Here is the flowchart when click on “Program” button, “Verify” is checked.

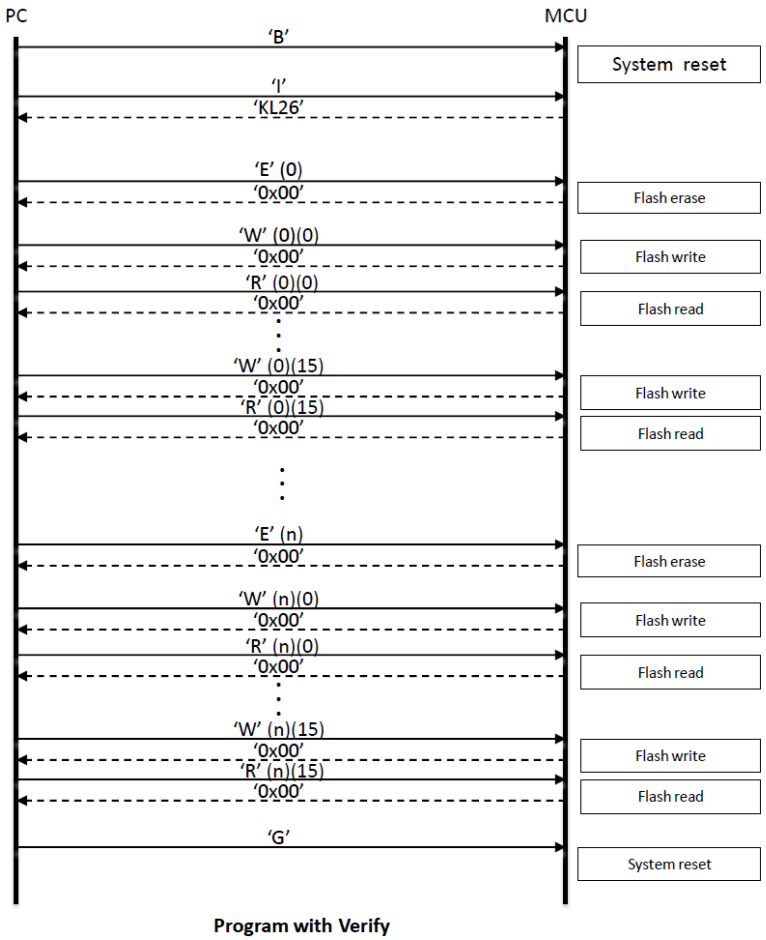

Figure 8. Running GUI – Program with Verify

Action 1: First, GUI sends 'B' command to target. If target is in running application, targets perform system reset to bootloader in program mode, ready for the next command.

Action 2: Next, GUI sends 'I' command, target erases APP_OK and answers GUI with the target identification.

Action 3: Next, GUI sends 'E' command, target erases one block Flash, size 0x400. If success, answer 0x00.

Action 4: Next, GUI sends 16 'W' and 'R' command successively. 'W' command request target writes 0x40 data to Flash and 'R' command read data back 0x40 data from Flash then send them to GUI.

Action 5: Next, GUI keeps on performing action3 and action4 until user application s19 file finished programming.

Action 6: Last, GUI sends 'G' command, target writes APP_OK in Flash. System reset, run application code.

Running GUI:

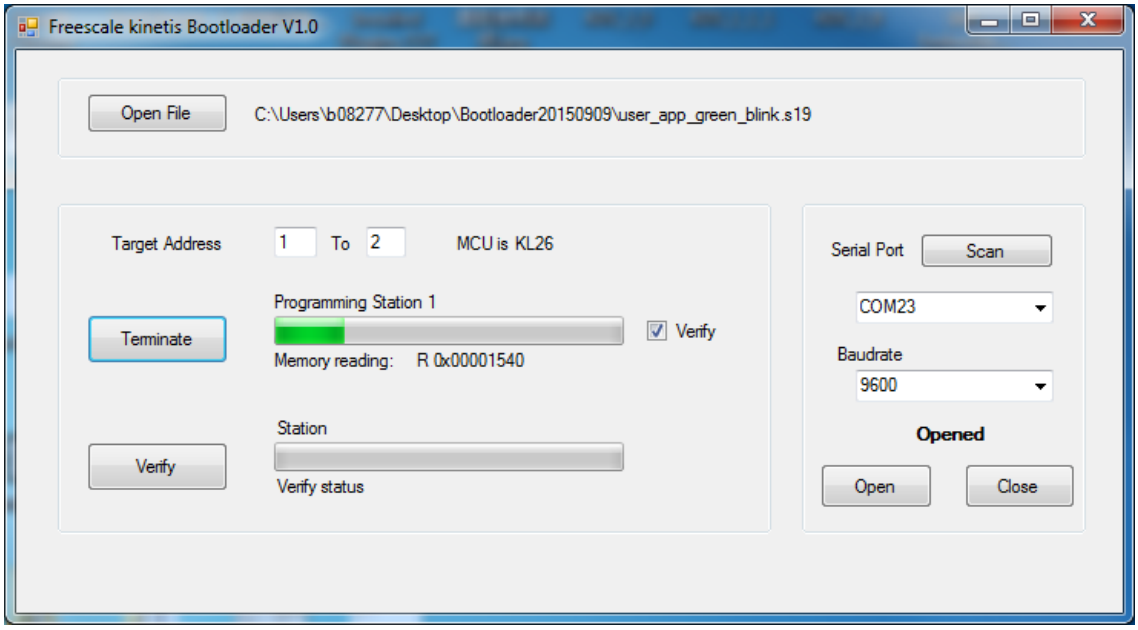

Figure 9. Running GUI – Program with Verify

4.3. Verify execution

Here is the flowchart when click on "Verify" button.

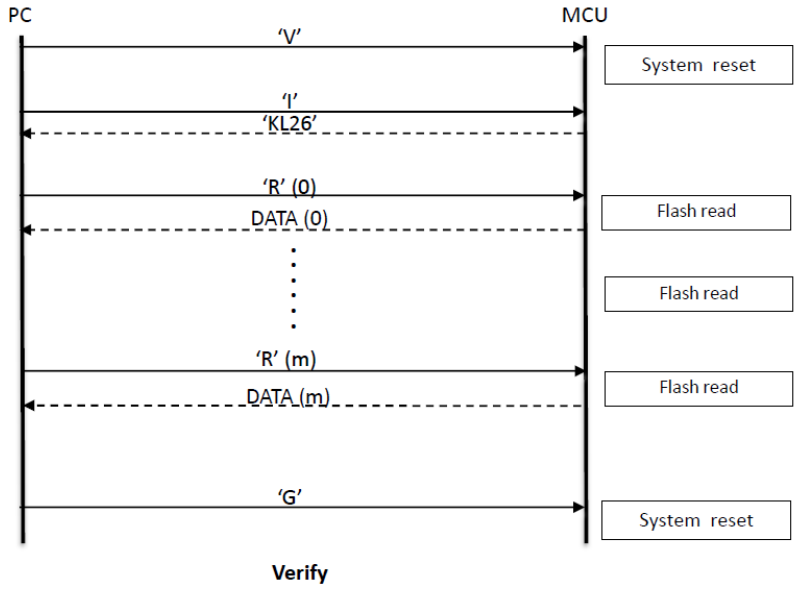

Figure 10. Running GUI – Verify

Action 1: First, GUI sends 'V' command to target. If target is in running application, targets perform system reset to bootloader in verify mode, ready for the next command.

Action 2: Next, GUI sends 'I' command, target answers with the target identification.

Action 3: Next, GUI sends ‘R’ command, target reads out 0x40 byte then answers it back to GUI.

Action 4: Next, GUI keeps on performing action3 until user application s19 file finished verification.

Action 6: Last, GUI sends ‘G’ command. System reset, run application code.

Running GUI:

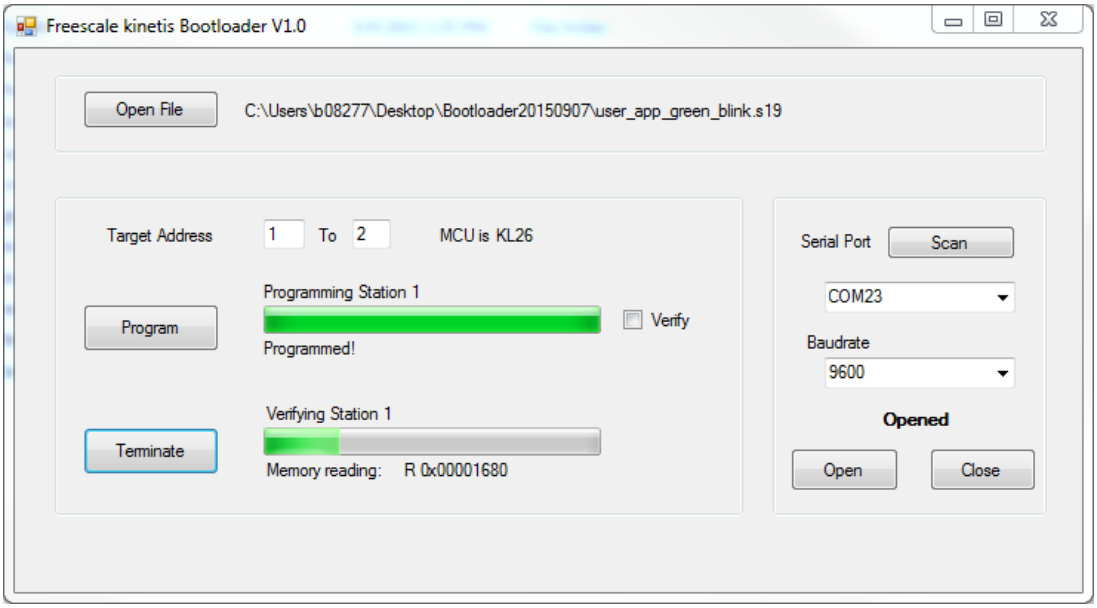

Figure 11. Running GUI – Verify

4.4. Log file generation

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

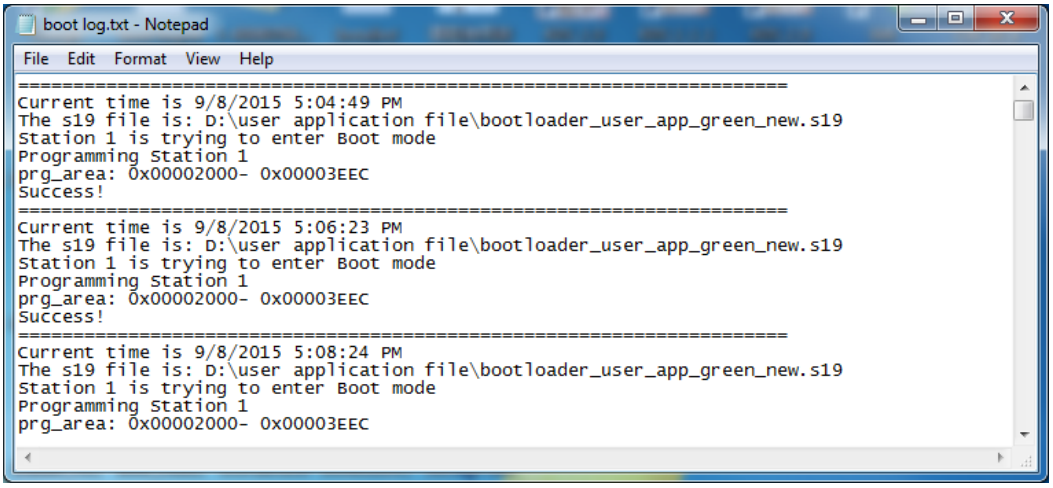

Figure 12. Boot Log File

5. Conclusion:

This application note, with MKL26Z128, makes a detailed description on how to implement a bootloader. The bootloader and application code are separated projects. User can use 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 single control unit or control units connected by networks. The firmware and GUI are FRDM-KL26 demo board. And the codes can be migrated to other chips easily.

6. References

1 AN4440 HCS08 Bootloader to Update Multiple Devices in a Field Bus Network
2 AN2295 Developer's Serial Bootloader MCUs.
3 KL26 Sub-Family Reference Manual
4 https://www.visualstudio.com/en-US/products/visual-studio-express-vs