5G

K.NarendraKumar,
Associate professor cse
Chalapthi Institute of Engineering and Technology, lam,Guntu,india

Abstract
5G stands for the fifth generation and refers to the next and newest mobile wireless standard based on the IEEE 802.11ac standard of broadband technology. 5G technology is expected to be faster, have fewer dead zones and end data caps on cellular contracts.

Introduction

G stands for a generation of mobile communication technology which is used in the mobile phones for communication. Different generations have different advances in technology. Basically, 1G was/is a voice-only phone, commensurate with those brick-like devices of the 80's. You'd be dealing with poor voice quality and that and poor battery life.

2G (aka Global system for mobile communication) denotes the transition from analog to digital in 1991, and the introduction of call and text encryption, plus data services like SMS, picture messages, and MMS. Then there's a bit of a nexus between 2G and 3G, with the interim 2.5G and 2.75G, making it possible to access the web pages via a mobile phone. 3G was developed in 1998 and upgraded audio and video meant better voice calling quality. 3G also brought faster data-transmission speeds making video calling and mobile internet more viable. 3.5G and 3.75G bought faster data processing and reduced latency.

Now, we're at 4G. 4G is up to 10 times faster than 3G services. Sprint was the first carrier to offer 4G speeds in the U.S. beginning in 2009. While all 4G service is called 4G or 4G LTE, the underlying technology is not the same with every carrier. Some use WiMax technology for their 4G network, while Verizon Wireless uses a technology called Long Term Evolution, or LTE.

So, What's So Good About 5G?

5G technology is expected to be faster, have fewer dead zones and end data caps on cellular contracts. The GMSA (The body behind MWC) specifies that to qualify for a 5G a connection should meet most of these eight criteria:

- One to 10Gbps connections to endpoints in the field
- One millisecond end-to-end round-trip delay
- 1000x bandwidth per unit area
- 10 to 100x number of connected devices
- (Perception of) 99.999 percent availability
- (Perception of) 100 percent coverage
- 90 percent reduction in network energy usage
- Up to ten-year battery life for low power, machine-type devices

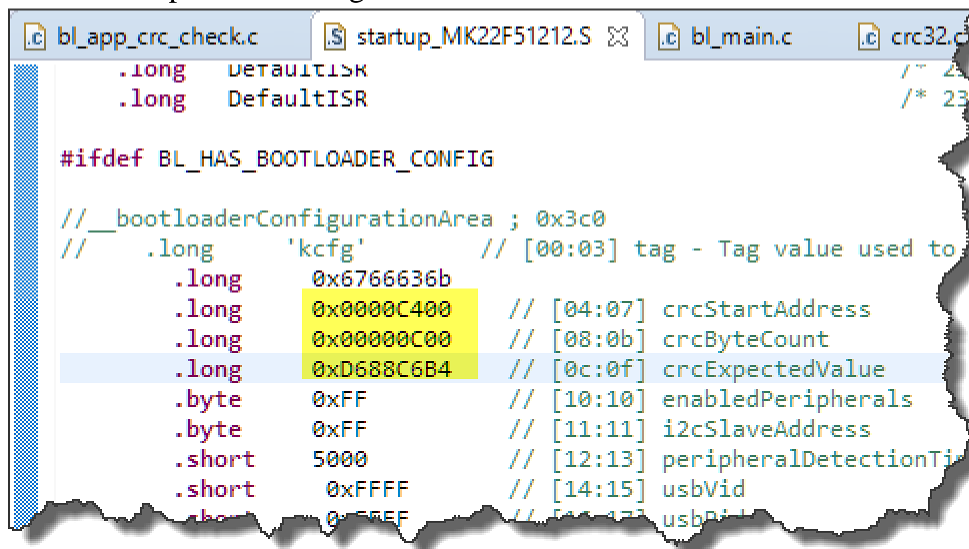
When you consider the sheer number of devices connected to the web including mobiles, wearable tech, AR and VR devices needs to accommodate increased traffic at greater speed but also be able to provide broader coverage for IoT devices (and we're not even at autonomous car stage yet.) If you want to stream video seamlessly, play a VR game or receive real-time insights, these are the kind of capabilities that 5G promises to achieve.

The official 5G standard has not yet been established. As noted by Engadget, The International Telecommunication Union has published draft 5G specs that set performance expectations. Users should get 100Mbps download speeds and 50Mbps for uploads -- unlike with LTE, though, that's more of a consistent baseline than a theoretical maximum. Consumers should also see an extremely low lag of no more than 4ms (versus 20ms for LTE), and service should work on trains traveling as quickly as 500km/h (311MPH). In short, this should be as fast as a *good* home internet connection.

So far, no smartphones support 5G because there aren't any mainstream 5G networks to which they can connect. Once these networks begin rolling out, we'll begin to see smartphones with 5G support. When things hit the mainstream in kind of widespread capacity, across regions and countries, is anyone's guess. Let's hope it's worth the wait.

CRC32 Checksum With the KBOOT Bootloader

In Flash-Resident USB-HID Bootloader with the NXP Kinetis K22 Microcontroller, I presented how I'm using the tinyK22 (or FRDM-K22F) with a flash resident USB HID bootloader. To make sure that the loaded application is not corrupted somehow, it is important to verify it with a Cyclic Redundancy Checksum (CRC). The NXP KBOOT Bootloader can verify such a CRC, but how to generate and use one is not really obvious (at least to me), so this article explains how to generate that CRC.



CRC Values for KBOOT

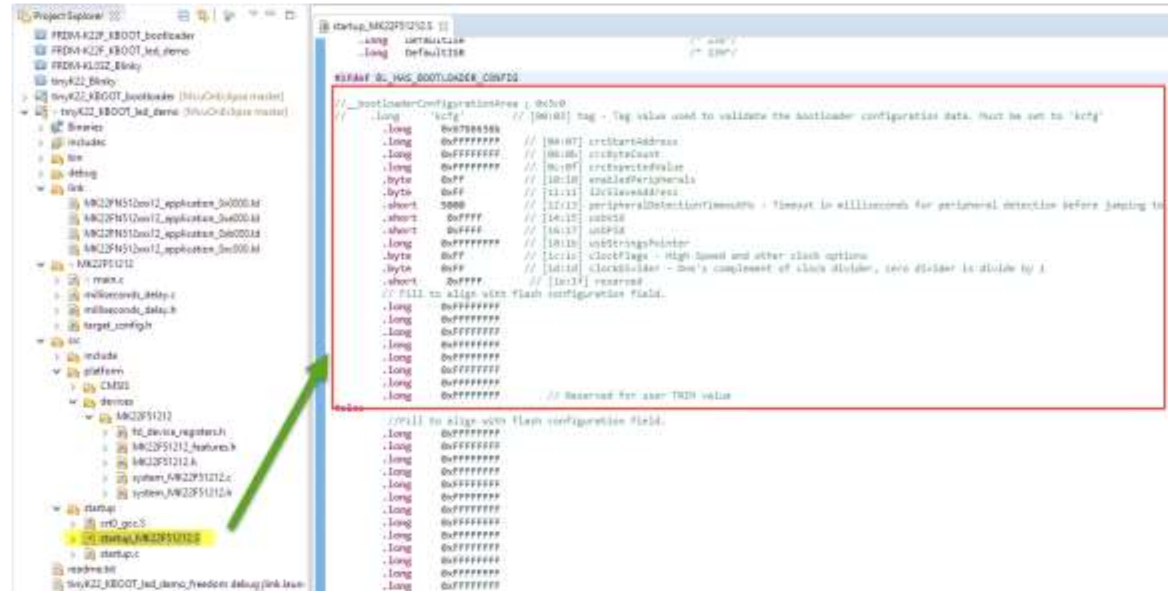
This article explains how to calculate the CRC32 for KBOOT, both from binary and S-Record files, and how to insert the values into the BCA (Bootloader Configuration Area). Additionally, it gives tips for debugging the bootloaded application.

Bootloader Configuration Area (BCA)

The bootloader is configured with a BCA (Bootloader Configuration Area). As explained in Getting Started: ROM Bootloader on the NXP FRDM-KL03Z Board, it configures the ROM bootloader. That ROM bootloader for the KL03Z does **not** implement the checksum feature, so I would have to build a flash-flash resident bootloader as explained in Flash-Resident USB-HID Bootloader with the NXP Kinetis K22 Microcontroller.

For the flash-resident bootloader, the BCA has part of the application to be loaded as well and located at offset 0x3C0 — right after the vector table located at offset 0x0000.

So if the application gets loaded at 0xC000 (as used in this tutorial), the BCA is located at 0xC3C0. That BCA can be implemented as a struct in C as in Getting Started: ROM Bootloader on the NXP FRDM-KL03Z Board or it could be part of the assembly code e.g. in the startup file as in this tutorial:



Bootloader Configuration Area

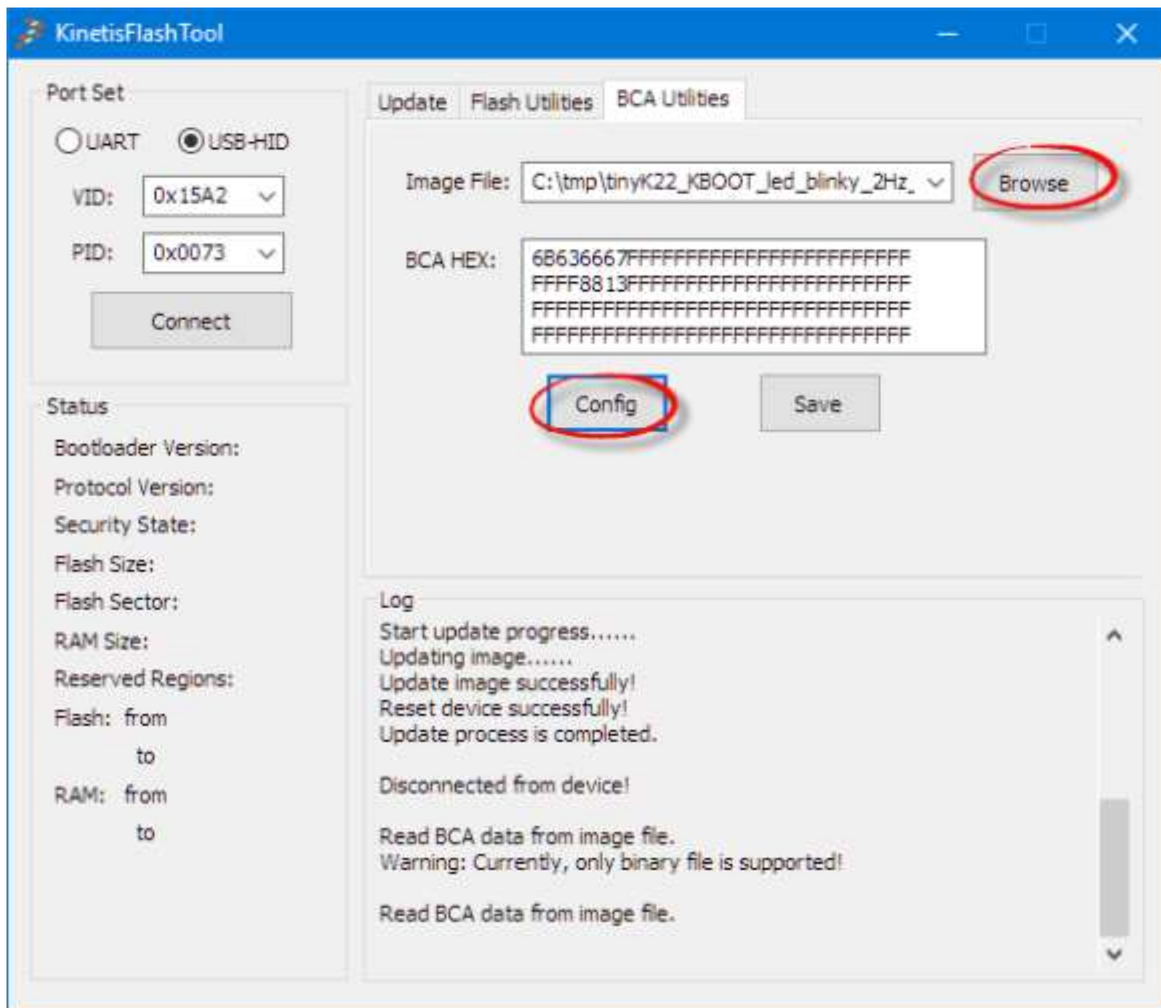
The CRC start address, size in bytes and the expected CRC values are just behind the tag bytes:

- 0x3c0 + 0x4: [04:07] crcStartAddress
- 0x3c0 + 0x8: [08:0b] crcByteCount
- 0x3c0 + 0xC: [0c:0f] crcExpectedValue

The question is: How do we calculate that expected CRC value?

CRC Value With KinetisFlashTool

One possibility to calculate the expected CRC values is to use the Kinetis Flash Tool. In the tool, browse for the binary (.bin) file (which is the only supported format) and press the Config button:



Config in KinetisFlashTool

In the dialog, enable the CRC Check with the image address:

CRC value

But this is a very painful process, and only works with .bin (Binary) files.

CRC With S-Record Files

A better approach is using the srec_cat utility (CRC Checksum Generation with 'SRecord' Tools for GNU and Eclipse):

```
srec_cat tinyK22_KBOOT_led_demo.srec -fill 0xff 0xc000 0xd000 -crop 0xc400 0xD000 -Bit_Reverse -CRC32LE 0x1000 -Bit_Reverse -XOR 0xff -crop 0x1000 0x1004 -Output - -hex_dump
```

Kudos go to Robert Poor (see https://community.nxp.com/thread/462397) **who has found out the correct command line to generate the CRC32 needed by KBOOT.**

- **-fill 0xFF 0xC000 0xD000**: fill memory from 0xC400..0xD000 with 0xff.

- **-crop 0xc400 0xD000**: just keep the area for the CRC calculation. This does **not** include the vector table and BCA itself.
- **-Bit_Reverse -CRC32LE 0x1000 -Bit_Reverse -XOR 0xff**: used to generate the correct CRC32 as expected by KBOOT and store it the given address.
- **-crop 0x1000 0x1004 -Output -HEX_DUMP**: Crop everything around the generated CRC32 and dump the output to the console.

Ideally, the vector table at 0xC000 and the BCA at 0xC3C0 would be included into the CRC, but KBOOT does not support this, so for simplicity, I keep it excluded from the CRC calculation.

To find out the size, use the linker map file or use srec_info:

```
srec_info inputfile.srec
```

Which gives something like:

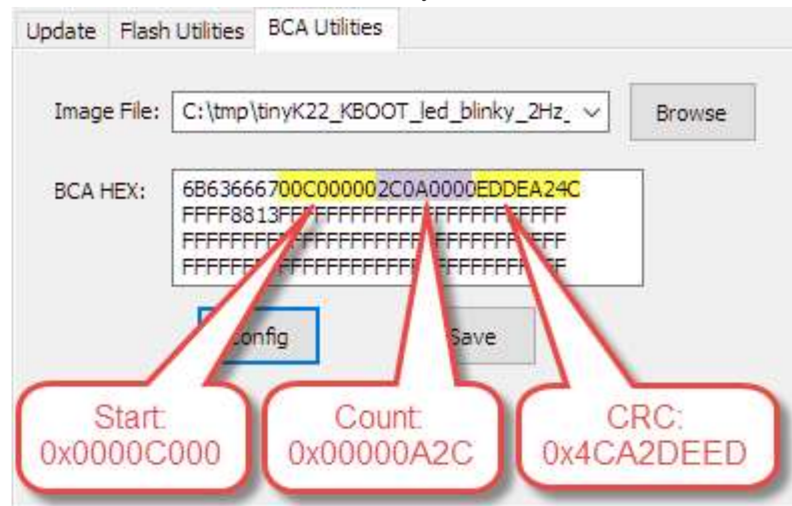
```
Format: Motorola S-Record
Header: "tinyK22_KBOOT_led_demo.srec"
Execution Start Address: 0000C4D9
Data: C000 - CA2B
```

This then produces something like this:

```
srec_cat tinyK22_KBOOT_led_demo.srec -fill 0xff 0xc000 0xd000 -crop 0xc400 0xd000 -Bit_Reverse -CRC32LE 0x1000 -Bit_Reverse -XOR 0xff -crop 0x1000 0x1004 -Output - -hex_dump
00001000: D6 88 C6 B4            #V.F4
```

The CRC32 is **0xD688C6B4**.

That value with the number of bytes and start address then can be entered into the sources like this:



Application CRC Values

However, this would require a recompilation of the application. So an easier way is to directly add the CRC32 to the .srec file:

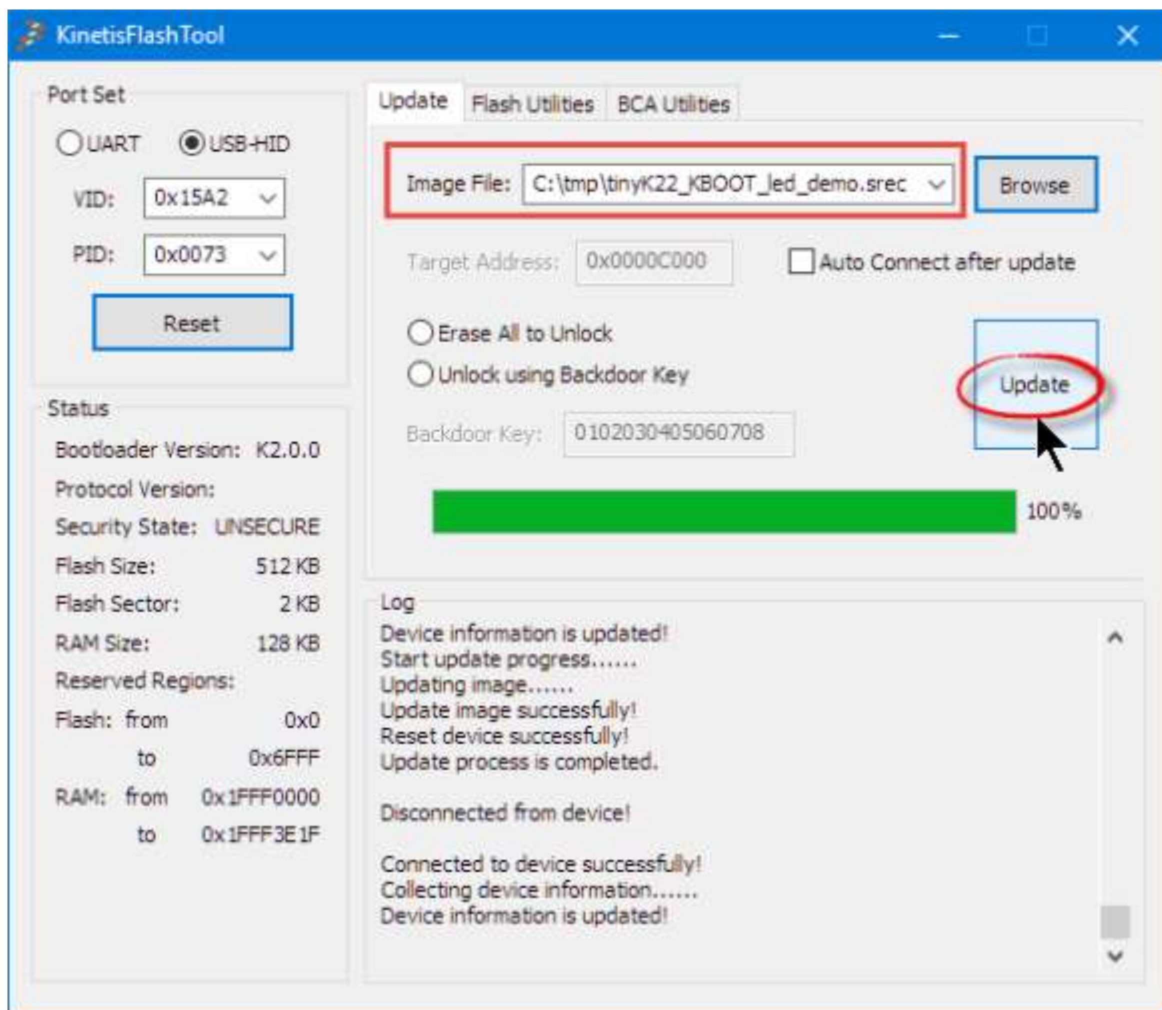
```
srec_cat -generate 0xc3cc 0xc3d0 -constant-l-e 0xD688C6B4 4 tinyK22_KBOOT_led_demo.srec -exclude 0xc3cc 0xc3d0 -Output_Block_Size 16 -output newWithCRC32.srec
```

Then load the new file with the KinetisFlashTool:

Kinetis Bootloader Configuration
Tag: Tag
Boot: Enable Direct Boot
Timeout: Timeout 5000 ms
Crc Check: Enable Image Address 0x0000c000
Peripheral: UART I2C SPI CAN USB
USB: VID 0x15A2 PID 0x0073 USB String Pointer 0x00000000
CAN: TXID 0x123 RXID 0x321
Disable Detection 125KHz 250KHz 500KHz 1MHz Specify
PREDIV 0xFF PSEG1 0x7 PSEG2 0x7
RJW 0x3 PROPSEG 0x7
Clock: Enable High Speed Clock Divider 0
I2C: I2C Slave Address 0x10
QSPI: QSPI Config Pointer 0x00000000
OTFAD: Key Blob Pointer 0x00000000
MMCAU: MMCAU Pointer 0x00000000
Reload OK Cancel
Generate C Code

Updating with SRecord Files

The CRC check can be debugged in the Bootloader inside the function is_application_crc_check_pass() inside bl_app_crc_check.c:



Code in Bootloader to Check CRC

CRC32 With Binary Files

Here is how to show the CRC fields of the BCA inside a binary file:

```
srec_cat tinyK22_KBOOT_led_demo.bin -binary -crop 0x3c0 0x3d0 -output - -hex_dump
```

This gives:

```
000003C0: 6B 63 66 67 00 C4 00 00 00 0C 00 00 B4 C6 88 D6 #kcfg.D......4F.V
```

```
bl_app_crc_check.c   startup_MK22F51212.S   bl_main.c   crc32.c   bl_app_crc_check.h

    property_store_t *propertyStore = g_bootloaderContext.propertyInterface->store;
    if (kStatus_AppCrcCheckInvalid != propertyStore->crcCheckStatus)
    {
        isCrcCheckPassed = false;

        // Check if crc check addresses reside in the range of valid memory space (Pflash or QSPIFlash)
        if (is_crc_check_address_valid(&propertyStore->configurationData))
        {
            uint32_t calculated_crc = calculate_application_crc32(
                (crc_checksum_header_t *)&propertyStore->configurationData, kBootloaderConfigAreaAddress);

            if (calculated_crc != propertyStore->configurationData.crcExpectedValue)
            {
                propertyStore
            }
            else
            {
                isCrcCheckPas
                propertyStore
            }
        }
        else
        {
            propertyStore->cr
        }
#if BL_FEATURE_CRC_ASSERT
        if (!isCrcCheckPassed
        {
            assert_pin_to_ind
        }
#endif
    }

    return isCrcCheckPassed;
}

Expression                              Type        Value
propertyStore->configurationD           uint32_t    0xd688c6b4

Name : propertyStore->configurationData.crcExpectedValue
    Details:3599287988
    Default:3599287988
    Decimal:3599287988
    Hex:0xd688c6b4
    Binary:11010110100010001100011010110100
    Octal:032642143264
```

Inspecting values in a binary file

To calculate the CRC32 value from a Binary, I use the following command line:

```
srec_cat tinyK22_KBOOT_led_demo.bin -binary -fill 0xff 0x0000 0x1000 -crop 0x0400 0x1000 -Bit_Reverse -CRC32LE 0x1000 -Bit_Reverse -XOR 0xff -crop 0x1000 0x1004 -Output - -hex_dump
00001000: 52 04 06 B8 #.c^B
```

It first fills the memory from 0x0000 to 0x1000 with the 0xff filler, then cuts out the area between 0x400 to 0x1000, calculates the checksum and issues it on the console.

Then add it to a binary. Below is the command line to insert that CRC32 value into the binary file at offset 0x3C4:

```
srec_cat -generate 0x3cc 0x3d0 -constant-b-e 0x52E406B8 4 tinyK22_KBOOT_led_demo.bin -binary -exclude 0x3cc 0x3d0 -output newWithCRC32.bin -Binary
```

CRC Checks and Debugging With 'Software' Breakpoints

There is one potential problem with the CRC calculation done by the bootloader: If your debugger probe modifies the flash memory for setting breakpoints (e.g. Segger J-Link can do this to get 'unlimited' breakpoints (see Software and Hardware Breakpoints, then this is changing the code, and as such invalidates the CRC checksum/check. So if loading and CRC-checking application code, make sure you don't have any breakpoints set in that area.

Automating

This article describes the manual steps to determine the CRC value and then add it to the application. For automating things with a Python script, see the work of Robert Poor at https://github.com/rdpoor/srec-crc.

Conclusion

Adding a CRC32 check in the bootloader makes the process more reliable, as it can detect bits in the loaded image. Knowing the correct polynomial and CRC calculation way is not always straightforward. Kudos to Robert Poor who has found out how to create the CRC32 for KBOOT. I'm now able to generate the checksum both from S19 and binary files. I'm doing things semi-automated (calculate the CRC and insert it into the file), and if I find time, I plan to automate it further. Until then, have a look how Robert is automating it (see the previous section).

The projects used in this tutorial are available on GitHub (see the links at the end of this article).

Happy checking!

Reference:

- Projects used in this Tutorial on GitHub: https://github.com/ErichStyger/mcuoneclipse/tree/master/Examples/KDS/tinyK22
- Flash-Resident USB-HID Bootloader with the NXP Kinetis K22 Microcontroller
- Generating CRC-32 for KBOOT (by Robert Poor): https://community.nxp.com/thread/462397
- How to automate CRC generation (by Robert Poor): https://github.com/rdpoor/srec-crc
- CRC Checksum Generation with 'SRecord' Tools for GNU and Eclipse
- Software and Hardware Breakpoints