05/05/2020 Samuele Stefanoni

DC11150

AN00184: Why using FRB file format

FlashRunner 2.0 is a Universal In-System Programmer, which feature several options to integrate flashing into your test system. This Application Note describes what is our proprietary FRB file format used to manage data to flash into a target device.

1. Introduction

FlashRunner 2.0 is able to program data content into a huge set of different devices. Each silicon manufacturer provides also an IDE. An IDE is a software dedicated to write firmware for a device set which includes code editor, standard libraries, compiler, linker and a set of useful utilities for software developers.

When software development ends, compiler and linked provide an output file which needs to be flashed into device embedded memory.

Every device has a so called, memory map. This means that embedded memory starts from a specific start address and ends on a specific end address. This range is described into device datasheet. A single device can have embedded inside multiple embedded memories dedicated to different purposes and of course each one of them will be memory mapped into a specific address range.

2. Output file types

Although there is no standard officially adopted by silicon manufacturers, there are at least three output file types which are commonly adopted. They are:

- Binary (.bin)

- Intel Hex (.hex)

- Motorola SREC (.s19, .srec, .s37)

Each format has different features, and they are suitable for different purposes. They act like a “container” which stores the firmware to flash and, in some cases, much more. There are also other output file formats, usually designed from silicon manufacturer itself which are designed to fit some particular device features. By creating a dedicated file format, silicon producers aim to store inside the output file format specific device information which can be then automatically parsed by other tools and be translated in actions on target device to be flashed.

3. A bunch of math

A numeral system is a writing system for expressing numbers; that is, a mathematical notation for representing numbers of a given set, using digits or other symbols in a consistent manner.

The same sequence of symbols may represent different numbers in different numeral systems. For example, "11" represents the number eleven in the decimal numeral system (used in common life) and the number three in the binary numeral system (used in computers)

The number the numeral represents is called its value.

$11_2 = 3_{16} = 3_{10}$

$11111110_2 = FE_{16} = 254_{10}$

What I just wrote here is value on the left is the same as value on the right, but its representation is different: on the left I have number represented in base 2 (binary), in the middle I represented numbers in base 16 and on the left I represented number in base 10, which is the base we learned at first in our first grade school and the one we use every day for our calculations.

If want to understand more about base conversion you can check out this Wikipedia page https://en.wikipedia.org/wiki/Positional_notation

In section “00…0F”, by default, you have numeral representation in hexadecimal systems or, equivalently, numbers in base 16. You can change view into whatever base but hexadecimal system is by far the most common base in computer science.

The reason for which hexadecimal is so common is simple: how many digits are there in hexadecimal notation? There are 16 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. How many digits are there in binary notation? There are 2 digits: 0, 1.

Now, how can I write the greatest digit, i.e. F_{16} in hexadecimal system in binary? $F_{16} = 1111_2$. By definition, 1 bit is a value in base 2 and 8 bits by definition forms 1 byte.

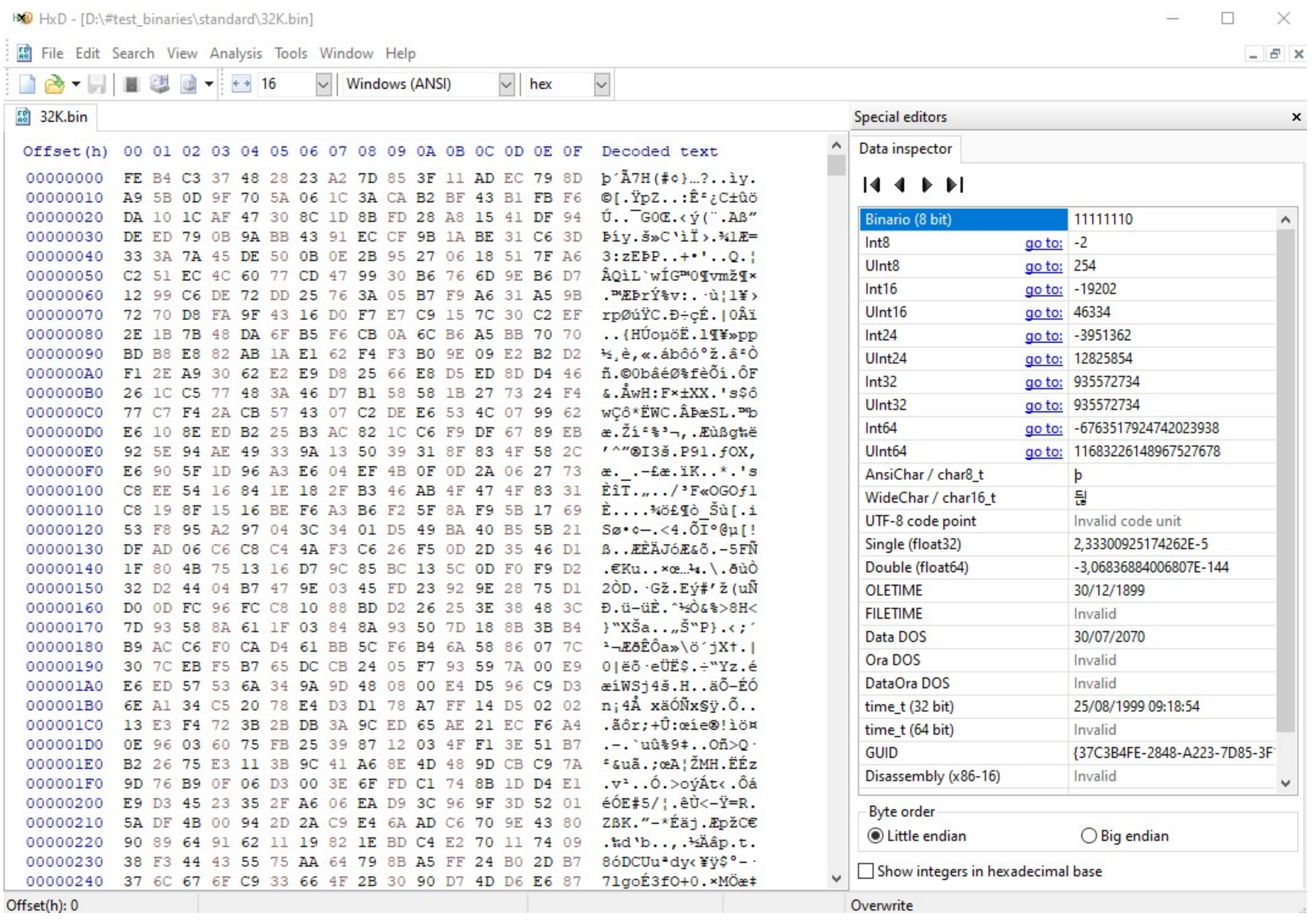
Here’s the conclusion: in order to represent 1 hexadecimal digits you’ll need 4 bits, and in order to represent 2 hexadecimal digits you’ll need 8 bits, i.e. one byte. This means that if you work with computers you’ll get familiar with bits and bytes quite soon, and hexadecimal system results to be the simplest and the clearest way for humans to understand what’s going on inside our computers.

For simplicity, when we use hexadecimal notation we place a 0x prefix to the value, for example: FE_{16} = 0xFE.

4. Binary

Binaries are the simplest, and immediate form of data container. It simply contains data, with no other additional information. In order to see what there's inside a binary, you need to download and install dedicate software utilities, like for example HxD, downloadable at https://mh-nexus.de/.

If you open a .bin file with HxD you'll find inside something like this. There are three sections: "Offset", "00….0F" and "Decoded text". What stays in the middle, i.e. section "00…0F" is actually data content, it's your firmware, in hexadecimal notation. On the "Offset" column you have addresses on which data will be written, also in hexadecimal notation.


HxD - [D:\#test_binaries\standard\32K.bin]
File Edit Search View Analysis Tools Window Help
16
Windows (ANSI)
hex
32K.bin
Special editors
Data inspector

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

00000000  FE B4 C3 37 48 28 23 A2 7D 85 3F 11 AD EC 79 8D  þ´Ã7H(#¢}…?..ìy.
00000010  A9 5B 0D 9F 70 5A 06 1C 3A CA B2 BF 43 B1 FB F6  ©[.ŸpZ..:Ê²¿C±ûö
00000020  DA 10 1C AF 47 30 8C 1D 8B FD 28 A8 15 41 DF 94  Ú..¯G0Œ.‹ý(¨.Aß”
00000030  DE ED 79 0B 9A BB 43 91 EC CF 9B 1A BE 31 C6 3D  Þíy.š»C‘ìÏ›.¾1Æ=
00000040  33 3A 7A 45 DE 50 0B 0E 2B 95 27 06 18 51 7F A6  3:zEÞP..+•'..Q.¦
00000050  C2 51 EC 4C 60 77 CD 47 99 30 B6 76 6D 9E B6 D7  ÂQìL`wÍG™0¶vmž¶×
00000060  12 99 C6 DE 72 DD 25 76 3A 05 B7 F9 A6 31 A5 9B  .™ÆÞrÝ%v:.·ù¦1¥›
00000070  72 70 D8 FA 9F 43 16 D0 F7 E7 C9 15 7C 30 C2 EF  rpØúŸC.Ð÷çÉ.|0Âï
00000080  2E 1B 7B 48 DA 6F B5 F6 CB 0A 6C B6 A5 BB 70 70  ..{HÚoµöË.l¶¥»pp
00000090  BD B8 E8 82 AB 1A E1 62 F4 F3 B0 9E 09 E2 B2 D2  ½¸è‚«.ábôó°ž.â²Ò
000000A0  F1 2E A9 30 62 E2 E9 D8 25 66 E8 D5 ED 8D D4 46  ñ.©0bâéØ%fèÕí.ÔF
000000B0  26 1C C5 77 48 3A 46 D7 B1 58 58 1B 27 73 24 F4  &.ÅwH:F×±XX.'s$ô
000000C0  77 C7 F4 2A CB 57 43 07 C2 DE E6 53 4C 07 99 62  wÇô*ËWC.ÂÞæSL.™b
000000D0  E6 10 8E ED B2 25 B3 AC 82 1C C6 F9 DF 67 89 EB  æ.Ží²%³¬‚.Æùßg‰ë
000000E0  92 5E 94 AE 49 33 9A 13 50 39 31 8F 83 4F 58 2C  ’^”®I3š.P91.ƒOX,
000000F0  E6 90 5F 1D 96 A3 E6 04 EF 4B 0F 0D 2A 06 27 73  æ._.–£æ.ïK..*.'s
00000100  C8 EE 54 16 84 1E 18 2F B3 46 AB 4F 47 4F 83 31  ÈîT.„../³F«OGOƒ1
00000110  C8 19 8F 15 16 BE F6 A3 B6 F2 5F 8A F9 5B 17 69  È....¾ö£¶ò_Šù[.i
00000120  53 F8 95 A2 97 04 3C 34 01 D5 49 BA 40 B5 5B 21  Sø•¢—.<4.ÕIº@µ[!
00000130  DF AD 06 C6 C8 C4 4A F3 C6 26 F5 0D 2D 35 46 D1  ß..ÆÈÄJóÆ&õ.-5FÑ
00000140  1F 80 4B 75 13 16 D7 9C 85 BC 13 5C 0D F0 F9 D2  .€Ku..×œ…¼.\.ðùÒ
00000150  32 D2 44 04 B7 47 9E 03 45 FD 23 92 9E 28 75 D1  2ÒD.·Gž.Eý#’ž(uÑ
00000160  D0 0D FC 96 FC C8 10 88 BD D2 26 25 3E 38 48 3C  Ð.ü–üÈ.ˆ½Ò&%>8H<
00000170  7D 93 58 8A 61 1F 03 84 8A 93 50 7D 18 8B 3B B4  }“XŠa..„Š“P}.‹;´
00000180  B9 AC C6 F0 CA D4 61 BB 5C F6 B4 6A 58 86 07 7C  ¹¬ÆðÊÔa»\ö´jX†.|
00000190  30 7C EB F5 B7 65 DC CB 24 05 F7 93 59 7A 00 E9  0|ëõ·eÜË$.÷“Yz.é
000001A0  E6 ED 57 53 6A 34 9A 9D 48 08 00 E4 D5 96 C9 D3  æíWSj4š.H..äÕ–ÉÓ
000001B0  6E A1 34 C5 20 78 E4 D3 D1 78 A7 FF 14 D5 02 02  n¡4Å xäÓÑx§ÿ.Õ..
000001C0  13 E3 F4 72 3B 2B DB 3A 9C ED 65 AE 21 EC F6 A4  .ãôr;+Û:œíe®!ìö¤
000001D0  0E 96 03 60 75 FB 25 39 87 12 03 4F F1 3E 51 B7  .–.`uû%9‡..Oñ>Q·
000001E0  B2 26 75 E3 11 3B 9C 41 A6 8E 4D 48 9D CB C9 7A  ²&uã.;œA¦ŽMH.ËÉz
000001F0  9D 76 B9 0F 06 D3 00 3E 6F FD C1 74 8B 1D D4 E1  .v¹..Ó.>oýÁt‹.Ôá
00000200  E9 D3 45 23 35 2F A6 06 EA D9 3C 96 9F 3D 52 01  éÓE#5/¦.êÙ<–Ÿ=R.
00000210  5A DF 4B 00 94 2D 2A C9 E4 6A AD C6 70 9E 43 80  ZßK.”-*Éäj.ÆpžC€
00000220  90 89 64 91 62 11 19 82 1E BD C4 E2 70 11 74 09  .‰d‘b..‚.½Äâp.t.
00000230  38 F3 44 43 55 75 AA 64 79 8B A5 FF 24 B0 2D B7  8óDCUuªdy‹¥ÿ$°-·
00000240  37 6C 67 6F C9 33 66 4F 2B 30 90 D7 4D D6 E6 87  7lgoÉ3fO+0.×MÖæ‡

Offset(h): 0
```

Binario (8 bit)		11111110
Int8	go to:	-2
UInt8	go to:	254
Int16	go to:	-19202
UInt16	go to:	46334
Int24	go to:	-3951362
UInt24	go to:	12825854
Int32	go to:	935572734
UInt32	go to:	935572734
Int64	go to:	-6763517924742023938
UInt64	go to:	11683226148967527678
AnsiChar / char8_t		þ
WideChar / char16_t		뷾
UTF-8 code point		Invalid code unit
Single (float32)		2,33300925174262E-5
Double (float64)		-3,06836884006807E-144
OLETIME		30/12/1899
FILETIME		Invalid
Data DOS		30/07/2070
Ora DOS		Invalid
DataOra DOS		Invalid
time_t (32 bit)		25/08/1999 09:18:54
time_t (64 bit)		Invalid
GUID		{37C3B4FE-2848-A223-7D85-3F
Disassembly (x86-16)		Invalid

Byte order: [✓ Little endian] Big endian

☐ Show integers in hexadecimal base

Overwrite

Is 0x0 the right address to which I should write my embedded flash memory? Unfortunately the answer is usually no. The reason is that microcontrollers are "mapping" all their devices, including embedded flash into a single map, resulting in big addresses. Can you imagine if your memory would be mapped at start address 0x20000000? You would have a huge binary bigger than 512MB with dummy data from 0x0 up to 0x20000000. There are several drawback with this file:

- Slowing down transfer between host pc and FlashRunner 2.0
- Wasting FlashRunner 2.0 storage memory

Although in some cases file output are looking like this, nowadays this practice is not common anymore. The alternative is the following: binary will contains data since the very beginning (i.e. from address 0x0) and start address is agreed and communicated separately. With this additional information, it's possible to indicate to FlashRunner that binary data must be written starting from target address 0x20000000 (and source file address 0x0). This method also has drawbacks: keeping this two information separated can bring misalignments, loss of information. For this reason some more clever method has been designed, and are presented below.

5. Intel Hex

Intel HEX has been designed in 1975 and is by far one of the most common data output file formats. Almost every silicon producers IDE has Intel Hex file format as output option.

Unlike binaries, Intel Hex is a text file. A text file is based on common ASCII character encoding standards. ASCII standard agrees that every number and alphabet letter commonly used in western world correspond to a number. This information is defined by ASCII table, which can be read here https://en.wikipedia.org/wiki/ASCII

So, being that hexadecimal notations contains both numbers and letters, it turns out that they can be written in ASCII. This could be a little bit tricky, so here an example:

0xFE is a hexadecimal number, it can be written as "FE" inside a text file using ASCII standard. In ASCII table we find out that letter "F" is 0x46 and letter "E" is 0x45.

That's what Intel Hex is, lines of ASCII text that are separated by line feed or carriage return characters. Each text line contains hexadecimal characters. An example:

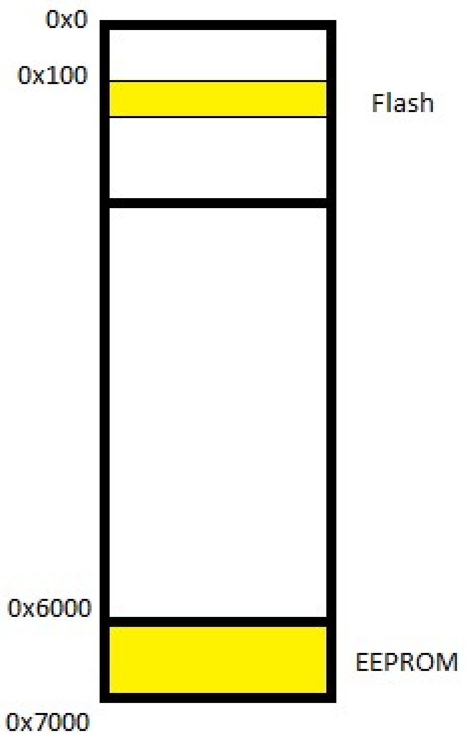
:10010000214601360121470136007EFE09D2190140

What's the purpose of all this? The purpose is to bring inside this file much more information compared to binary file. In fact, this information is placed using a specific format which divides in "fields" each line. Each field has a specific size.

:	"start code", every line must start with this character
10	byte count field. It tells you that this line contains 16 bytes of data
0100	address field. It tells you that data must be written starting from address 0x100
0021…	data to be programmed
40	checksum of this line

As you can see there is are two major benefits by using Intel Hex. The most evident is that compared to binary file format you have addresses embedded, so you know exactly to which address data must be flashed.

The second benefit is that this information is repeated for each line, so you can have “holes” between data. Let’s see an example:



This image above could represent a typical device memory map. This device has two different embedded memories on die, i.e. a flash memory and an EEPROM memory. They are mapped in two different regions, flash begins at address 0x0 and ends at address 0x1000, EEPROM begins at address 0x6000 and ends at address 0x7000. Firmware has data in both areas and are marked in yellow:

Flash has data at address 0x100 for 0x20 bytes

EEPROM has data at address 0x6000 for 0x1000 bytes

How can this be translated in an Intel Hex file? Here’s a possible description:

:10010000021460136012147013600 7EFE09D2190140

:100110002146017E17C20001FF5F16002148011928

:10600000E13AE66C8EF1358991EF438A847DC805CB

:106010003EDAFB16AEA9FAB643D49ADFC7448F3DE9

…

:106FF000111670ADE84496BE39F1D478EAD20A662B

As you can see, the first two rows are defining data at 0x100 for 0x10 bytes and at 0x110 for 0x10 bytes, resulting in a unique block starting at 0x100 for 0x20 bytes which will be targeted into Flash memory.

Then you have a new line which starts at 0x6000 address, defining data for EEPROM memory. Can you see the advantage? You don’t have any dummy data in between: this means that there is no waste of data, as it happens for binaries.

Although seems that Intel Hex bring only benefits, they also have a drawback: while as explained above one hexadecimal digit requires 4 bits, every ASCII character requires instead 8 bits 8 (an unsigned char). So, Intel Hex requires at least double space compared to the firmware needed to be flashed. It could be a minor issue, but nowadays memories are coming up with 256GB size, if data to be flashed would be in Intel Hex, file would have 512GB size.

5. Motorola SREC

Motorola SREC is quite similar to Intel Hex, except from the fact that the field specifications are slightly different. Here's an example of a SREC line:

S1137AF00A0A0D000000000000000000000000000061

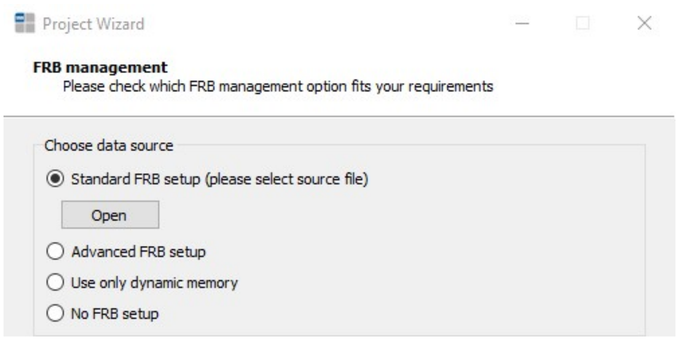
S1	is the "start code", every line must start with this character
13	is the byte count field. It tells you that this line contains 16 bytes of data
7AF0	is the address field. It tells you that data must be written starting from address 0x100
0A0A…	data to be programmed
61	is the checksum of this line

6. What is an FRB file

SMH Technologies provides a unique, common file format which let you combine together all advantages of all file format descripted above with not drawbacks.

FRB stays for FlashRunner Binary and can be created using FlashRunner 2.0 Workbench software, included in the System software freely downloadable from our website, or through command line utility frb_converter.exe.

FlashRunner 2.0 Workbench provides you two options to create an FRB file: Standard FRB setup and Advanced FRB setup. Both options are described in details in Programmer's Manual. Both paths accept Binary, Intel Hex, Motorola SREC file types.


Project Wizard

FRB management
Please check which FRB management option fits your requirements

Choose data source

● Standard FRB setup (please select source file)

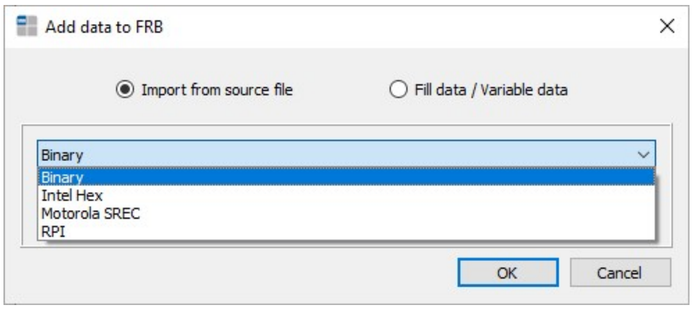
Open

○ Advanced FRB setup

○ Use only dynamic memory

○ No FRB setup

Using Advanced FRB setup you'll have the possibility to add more input files into the same FRB, resulting in a single file containing a merge of all data to be written.



Let's assume we would like to convert into an FRB an Intel Hex file like the following one:

:1001000021460136012147013600 7EFE09D2190140

:100110002146017E17C20001FF5F16002148011928

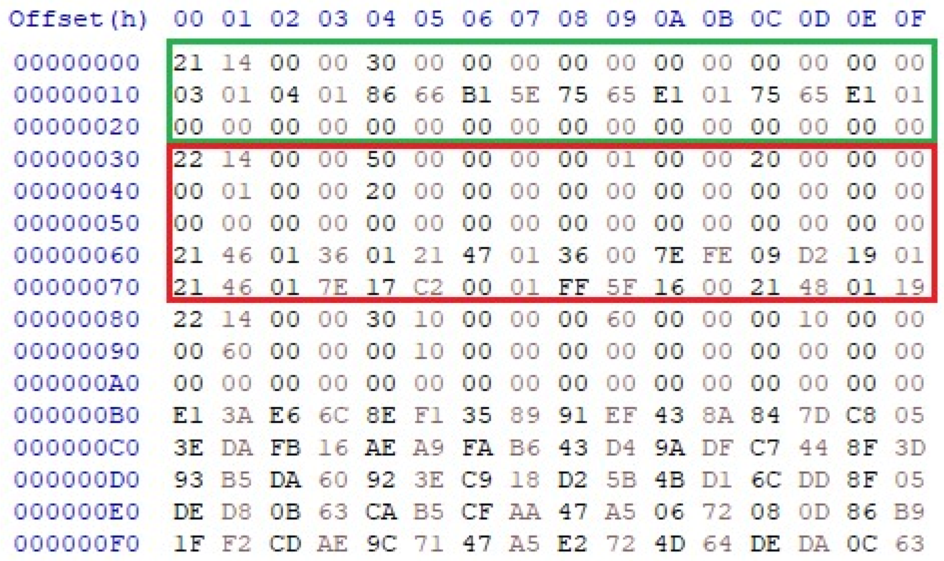
:10600000E13AE66C8EF1358991EF438A847DC805CB

:106010003EDAFB16AEA9FAB643D49ADFC7448F3DE9

…

:106FF000111670ADE84496BE39F1D478EAD20A662B

Using HxD editor we can open the FRB file created. The result will be the following:



```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   21 14 00 00 30 00 00 00 00 00 00 00 00 00 00 00
00000010   03 01 04 01 86 66 B1 5E 75 65 E1 01 75 65 E1 01
00000020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030   22 14 00 00 50 00 00 00 00 01 00 00 20 00 00 00
00000040   00 01 00 00 20 00 00 00 00 00 00 00 00 00 00 00
00000050   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060   21 46 01 36 01 21 47 01 36 00 7E FE 09 D2 19 01
00000070   21 46 01 7E 17 C2 00 01 FF 5F 16 00 21 48 01 19
00000080   22 14 00 00 30 10 00 00 00 60 00 00 00 10 00 00
00000090   00 60 00 00 00 10 00 00 00 00 00 00 00 00 00 00
000000A0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000B0   E1 3A E6 6C 8E F1 35 89 91 EF 43 8A 84 7D C8 05
000000C0   3E DA FB 16 AE A9 FA B6 43 D4 9A DF C7 44 8F 3D
000000D0   93 B5 DA 60 92 3E C9 18 D2 5B 4B D1 6C DD 8F 05
000000E0   DE D8 0B 63 CA B5 CF AA 47 A5 06 72 08 0D 86 B9
000000F0   1F F2 CD AE 9C 71 47 A5 E2 72 4D 64 DE DA 0C 63
```

FRB is using headers to store additional information aside from data to flash. Headers have 48 byte size (0x30) until 4GB file size. If data exceed this limit, headers will take 96 byte size (0x60). Information are stored in Little Endian representation (for more information about endianness please visit https://en.wikipedia.org/wiki/Endianness). In the table below is represented the field sequence, filling out with 0 value to reach 96 byte size.

h_type	0x1421 MAIN HEADER 0x1422 DATA HEADER 0x1423 FILL HEADER

block_len	<32bit integer>				
start_addr	<32bit integer>				
len	<32bit integer>				
main	version	data	orig_start_addr	fill	value
	time		orig_len		byteWord
	file_crc32		offs		<null>
	data_crc32		byteWord		<null>
	crypted		absFrbOffs		<null>
	filledByte		<null>		<null>
	numHeaders		<null>		<null>

While h_type is declaring which header type it is, next 3 fields are block_len, start_addr, len and they are all in common between all headers type. After that, each header type has specific information.

MAIN HEADER:

Every FRB starts with a "main header" indicated inside the green box. Important fields belonging to this header are:

- version: indicates converter version used to create FRB file
- time: timestamp of the FRB conversion
- data_crc32: CRC32 calculation over data block

CRC32 is an additional security feature compared to compared to Intel Hex files. Instead of a simple checksum there's a CRC32 over the whole data. Changing a single byte inside an FRB will corrupt it and file will not be accepted anymore by FlashRunner 2.0.

DATA HEADER:

Each data block is initiated by a data header. Let's take back our first example: each data header block will contain the biggest block of **contiguous** data.

Block1:

```
:10010000214601360121470136007EFE09D2190140
:100110002146017E17C20001FF5F16002148011928
```

Block2:

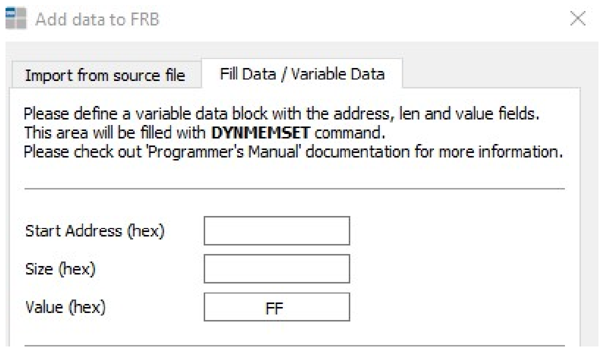
```
:10600000E13AE66C8EF1358991EF438A847DC805CB
:106010003EDAFB16AEA9FAB643D49ADFC7448F3DE9
…
```

:106FF000111670ADE84496BE39F1D478EAD20A662B

This principle will maintain the benefit of having only defined data inside the file, not wasting space and resources to fill with dummy values between disjointed blocks. After header, real data to flash will be appended.

FILL HEADER:

If you want to fill with some values a certain region you can actually do it anyway, using the fill feature using Advanced FRB setup included in FlashRunner 2.0 Workbench.



Please define a variable data block with the address, len and value fields.
This area will be filled with **DYNMEMSET** command.
Please check out 'Programmer's Manual' documentation for more information.

Start Address (hex)
Size (hex)
Value (hex) FF

This header contains standard information i.e. start_addr, len plus a single field dedicated to store the value which will be used as fill. This way, filling option will keep small file size and can be moved or resized quite easily.

NULL HEADER

At the end of the FRB file a “null header” is appended, which is a series of zeroes aligned to 16 bytes. They will not be included in the flashing data block, they are like a “termination” character, which will inform FlashRunner 2.0 that FRB is finished.

6. Conclusions

Having an overview of how FRB works is an opportunity to appreciate the benefits of using them as file format for your projects. As a resume, they are briefly reported here:

- Combine together multiple input source file in one single file, satisfying one common production environments desiderata. Input file can be mixed.

- Including only “real” data: by using headers, all additional information can be stored in the FRB. No need to fill data. Only input source file information is defined and then programmed.

- Filling is achieved by only a dedicated header file, a triplet of start_addr, len, value. No need to fill out huge data blocks wasting space and slowing down file transfer operation.

- CRC32 calculated and stored inside the file will guarantee that no tampering has been achieved on the FRB file, flashed data will be 100% matching original input file data.