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)

Systein Italia S.r.l.

SMH Technologies®

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: $FE_{16} = 0xFE$.

SYNERGY OF IN-SYSTEM PROGRAMMING LEADERS

Systein Italia S.r.l.

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.
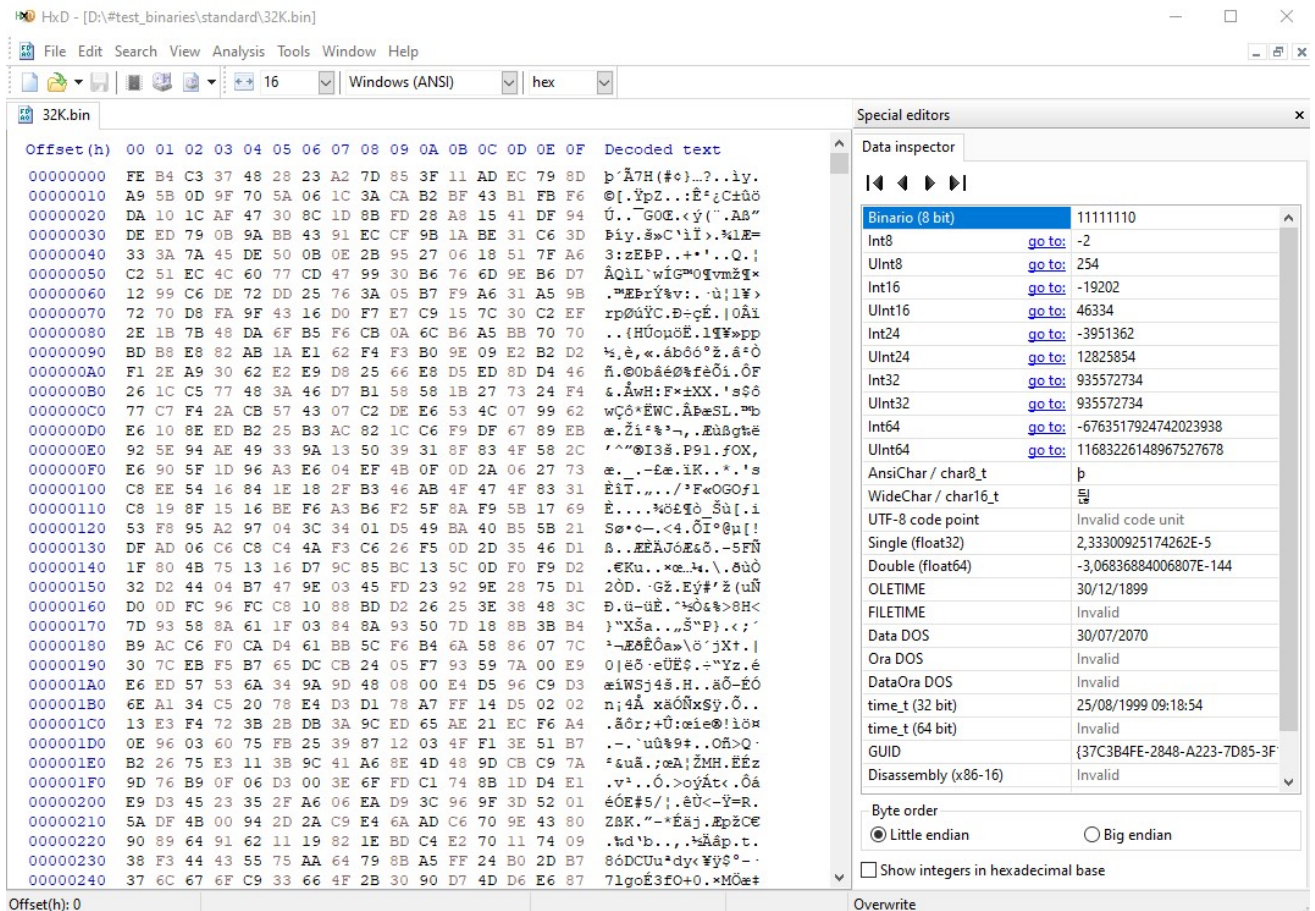
Is 0x0 the right address to which I should write my embedded flash memory? Unfortunately the answer is usually no. The reason is that microcontrollers are "mapping" all their devices, including embedded flash into a single map, resulting in big addresses. Can you imagine if your memory would be mapped 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:

SYNERGY OF IN-SYSTEM PROGRAMMING LEADERS

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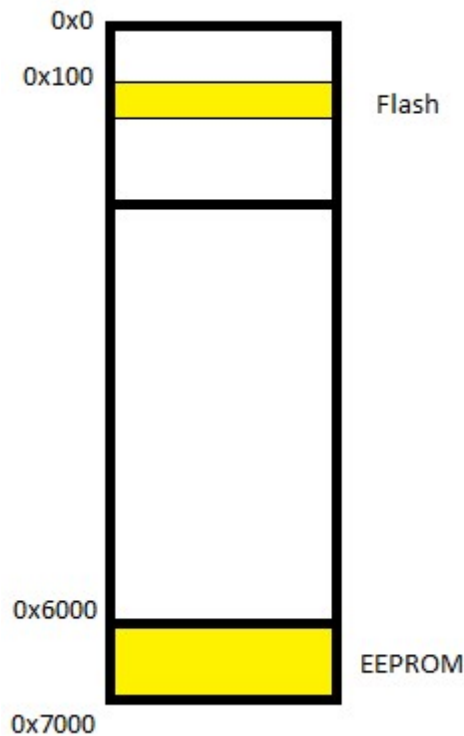
This image above could represent a typical device memory map. This device has two different embedded memories on die, i.e. a flash memory and an EEPROM memory. They are mapped in two different regions, flash begins at address 0x0 and ends at address 0x1000, EEPROM begins at address 0x6000 and ends at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

:10**0100**002146013601214700136007EFE09D2190140

:10**0110**002146017E17C20001FF5F16002148011928

:10**6000**00E13AE66C8EF1358991EF438A847DC805CB

:10**6010**003EDAFB16AEA9FAB643D49ADFC7448F3DE9

…

:10**6FF0**00111670ADE84496BE39F1D478EAD20A662B

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.

SYNERGY OF IN-SYSTEM PROGRAMMING LEADERS

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:

S1137AF00A0A0D0000000000000000000000000061

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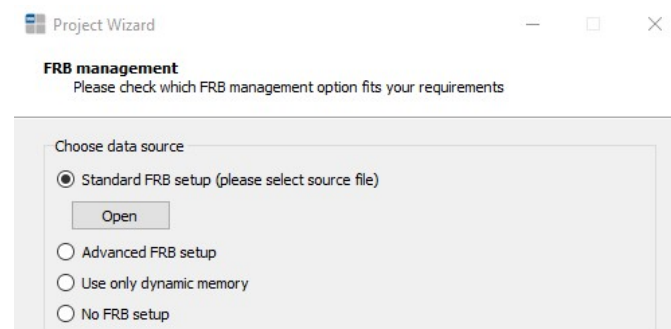
Using Advanced FRB setup you'll have the possibility to add more input files into the same FRB, resulting in a single file containing a merge of all data to be written.

SYNERGY OF IN-SYSTEM PROGRAMMING LEADERS

Systein Italia S.r.l.
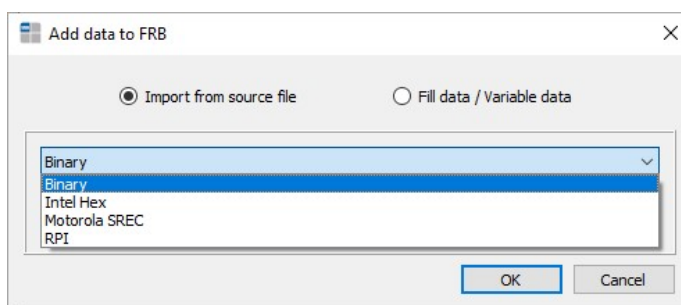
Let's assume we would like to convert into an FRB an Intel Hex file like the following one:

:10010000214601360121470136007EFE09D2190140

:100110002146017E17C20001FF5F16002148011928

:10600000E13AE66C8EF1358991EF438A847DC805CB

:106010003EDAFB16AEA9FAB643D49ADFC7448F3DE9

…

:106FF000111670ADE84496BE39F1D478EAD20A662B

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

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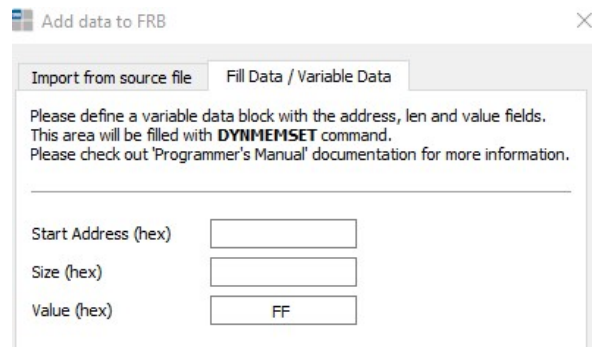
This header contains standard information i.e. start_addr, len plus a single field dedicated to store the value which will be used as fill. This way, filling option will keep small file size and can be moved or resized quite easily.

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.