Kanda Electronics Blog

MICROCONTROLLERS

INTEL HEX FILES EXPLAINED

DECEMBER 19, 2016 | BLOG ADMIN | LEAVE A COMMENT

We often come across a problem where an understanding of Intel Hex file format will help in solving a programming issue or technical point. Intel Hex format is a standard layout for files produced by assemblers or C compilers when they compile your source code. It is used by device programmers to program the target microcontroller with your code.

ASCII Characters

An assembler or C compiler could just output binary data, which is what a microcontroller needs, and store it as a binary file. If you set your compiler to output a .bin file it will appear as garbage in a text editor, so an Intel Hex file stores data as ASCII characters, which can be read by an editor. The device programmer has to convert these ASCII values into binary data for the microcontroller.

This post is not going to discuss ASCII, data encoding, Unicode or other ways of displaying data in a text editor any further but you should be aware that Intel Hex format uses character encoding so that you can make sense of the output.

Intel Hex Data Layout

Each line in an Intel Hex file has the same basic layout, like this:

:BBAAAATT[DDDDDDDD]CC

where
: is start of line marker
BB is number of data bytes on line
AAAA is address in bytes
TT is type discussed below but 00 means data
DD is data bytes, number depends on BB value
CC is checksum (2s-complement of number of bytes+address+data)

Here is an example.

:10000000112233445566778899AABBCCDDEEFF00F8

The line must start with a colon, :, followed by the number of data bytes on the line, in this case 0x10 or 16 decimal. Each data byte is represented by 2 characters.
Here is an example with only four bytes of data on the line:

:0400100011223344442

Next on the line comes a 2 byte address, represented by 4 characters, with possible values from 0x0000 to 0xFFFF (0 to 64KB). This is followed by the type. This can have a range of values depending on whether the line contains data or not. Type 00 like these examples shows that the line contains data.

The data bytes come next, 2 characters to a byte. The number of data bytes is set at the beginning of the line.

Checksum

Each line must end in a checksum. This is the 2s-complement of the sum of the number of bytes, plus the address plus the data. To do this, add up the number of bytes, the address and all the data and discard any carry to give an 8-bit total. Write this in binary, then invert each digit to give 1s-compliment. Add one to give 2s-compliment.

The checksum in this example line **:0400100011223344442** is 42. This is calculated as follows –

0x04 + 0x00 + 0x10 + 0x11 + 0x22 + 0x33 +0x44 = 0xBE

0xBE is 10111110 in binary.

Invert this value eg by XOR with 0xFF gives 01000001 = 0x41.

Add 1, gives the result 0x42, which matches the checksum shown. The Programmers Calculator in Windows makes this operation simple.

Other Types

The data type 00 indicates that the line contains data bytes. The other type that must occur in every Intel Hex file is 01 as this is the end of file marker. Every file must end with an 01 type, which looks like this – **:00000001FF**.

The other types that you may see are 02 and 04 that show that the line is an extended address line.

Extended Addresses

You may have worked out that with a possible address range of 0x0000 to 0xFFFF, this Intel Hex file format can only store 64KB (0xFFFF) of data. Newer microcontrollers and certainly memory chips can hold much more data than this so how do we include data above 64KB?

The answer is to use a type that means that the line is an extended address rather than a line containing data and both 02 and 04 types do this, in a slightly different way. As the standard addressing is 0 to 0xFFFF, we need a marker that increases the address of the next block by 0x10000, the following block by 0x20000, then 0x30000 and so on.

The first block has addresses in the normal 0..0xFFFF range. Then we have an extended address marker that means add 0x10000 to all subsequent addresses giving possible addresses of 0x10000 to 1FFFF. Then we might have further extended addresses that add 0x20000 to each following address, 0x30000, 0x40000 etc.

Both 02 and 04 types do this.

02 Linear Address Type

This has the following form for each 64KB block:

1st Block – add 0x00000: :020000020000FE
2nd Block – add 0x10000: :020000021000FD
3rd Block – add 0x20000: :020000022000FC
4th Block – add 0x30000: :020000023000FB

The format is shift the value after the 02 marker 4 places left, then 0x1000 becomes 0x10000, 0x2000 becomes 0x20000 and so on. Most files with extended addressing include the first :020000020000FE zero marker but this does nothing – 0x0000 shift left 4 is still zero.

04 Extended Address Type

This does the same thing but has a slightly different format

:020000040000FA
:020000040001F9
:020000040002F8
:020000040003F7
:020000040004F6

This time the value after the 04 marker is shifted left 16. Therefore 0x0001 also becomes 0x10000 etc.

Summary

Each line has the same format, :, number of data bytes, address, type, data and checksum. The different types are used to show data, address or end of file. Some compilers include data lines that are all 0xFF but some omit these. Addresses can be in order but some compilers mix up the addresses so they aren't in order. The number of data bytes is usually 0x10 per line but again this can vary greatly.

PIC Hex File Format post shows how things like configuration bytes, lock bits are stored in an Intel Hex File.

PIC PROGRAMMER