【 기술 노트 12 】

인텔 HEX 파일에 대한 올바른 이해

마이크로프로세서에서 어셈블리나 C 언어로 프로그램을 작성해본 사람은 누구나 이를 번역하고 링크하여 최종적으로 인텔(Intel) HEX 파일을 생성하게 된다. 이 프로그램을 타겟 보드에서 실행하기 위하여 ROM에 굽거나 또는 RS-232C 직렬통신으로 다운로드를 수행할 때는 거의 예외없이 인텔 HEX 파일을 필요로 하기 때문이다.

그런데, 이때 많은 사람들이 2진수 형식의 파일(binary file)인 오브젝트 파일과 ASCII 파일인 인텔 HEX 파일을 제대로 이해하지 못하는 것을 보게 된다. 이는 보다 근본적으로는 2진수와 ASCII 코드의 차이를 이해하지 못하는데서 기인한다.

1. 2진수의 값과 ASCII 코드

모든 마이크로프로세서는 2진수로 되어 있는 명령이나 데이터만을 이해할 수 있으며, 따라서 어셈블러나 C컴파일러가 소스 프로그램을 번역하고 링커가 이를 링크하여 실행 가능한 오브젝트 파일로 만들 때는 항상 8비트 단위의 2진 파일을 생성한다. 2진 파일은 모든 명령과 데이터가 2진수 값(value)으로 표시되는 파일이다.

그런데, 이들 8비트의 2진수는 00000000B~11111111B와 같이 되어 있어서 읽거나 기억하기가 매우 불편하므로 사람들은 이를 흔히 2자리의 16진수로 표현한다. 이 16진수라는 것은 단지 사람에게 편리하도록 만든 것뿐으로서, 내용적으로 2진수와 다를 것이 없으며 마이크로프로세서는 16진수를 이해하지 못하고 오로지 2진수만을 이해할 수 있다. 2진수와 16진수의 관계는 <표 1>에 보였으며, 8비트의 2진수 값 00000000B~11111111B는 당연히 16진수로는 00H~FFH가 된다.

<표 1> 2진수와 16진수 대비표

2진수	16진수	2진수	16진수
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

한편, 컴퓨터와 컴퓨터 사이나 또는 컴퓨터와 주변장치 사이에 통신을 수행하려면 주고받는 데이터(도형 문자) 이외에 통신을 제어하기 위한 명령(기능제어 문자)이 필요하게 된다. 서로 다른 컴퓨터 기종이나 주변장치 사이에도 통신에 문제가 없도록 통신에서 사용되는 문자들은 표준으로 정해져 있는데, 이러한 표준 코드에는 <표 2>와 같은 ASCII 코드가

가장 널리 사용된다. ASCII(American Standards Code for Information Interchange) 코드는 국제표준기구(ISO, International Standard Organization)에서 제정한 국제표준의 문자 데이터 표현을 위한 코드로서, 컴퓨터 내부에서의 문자표현은 물론 컴퓨터와 컴퓨터 사이의 정보교환 또는 컴퓨터와 주변장치 사이의 정보교환에 가장 일반적으로 사용된다. 이 ASCII 코드는 7비트로 표현되므로 모두 128문자를 나타낼 수 있는데, 이중에서 20H~7EH(총95자)는 도형 문자로서 영문의 대/소문자, 아라비아 숫자, 각종 문장부호 등을 표현하는데 사용되고, 00H~1FH 및 7FH(총33자)는 기능제어 문자로서 주변장치의 제어, 직렬통신제어 등에 사용된다.

<표 2> ASCII 코드표

코드값	문자	코드값	문자	코드값	문자	코드값	문자
00H	NUL	20H	space	40H	@	60H	`
01H	SOH	21H	!	41H	A	61H	a
02H	STX	22H	"	42H	B	62H	b
03H	ETX	23H	#	43H	C	63H	c
04H	EOT	24H	$	44H	D	64H	d
05H	ENQ	25H	%	45H	E	65H	e
06H	ACK	26H	&	46H	F	66H	f
07H	BEL	27H	'	47H	G	67H	g
08H	BS	28H	(	48H	H	68H	h
09H	HT	29H	)	49H	I	69H	i
0AH	LF	2AH	*	4AH	J	6AH	j
0BH	VT	2BH	+	4BH	K	6BH	k
0CH	FF	2CH	,	4CH	L	6CH	l
0DH	CR	2DH	-	4DH	M	6DH	m
0EH	SO	2EH	.	4EH	N	6EH	n
0FH	SI	2FH	/	4FH	O	6FH	o
10H	DLE	30H	0	50H	P	70H	p
11H	DC1	31H	1	51H	Q	71H	q
12H	DC2	32H	2	52H	R	72H	r
13H	DC3	33H	3	53H	S	73H	s
14H	DC4	34H	4	54H	T	74H	t
15H	NAK	35H	5	55H	U	75H	u
16H	SYN	36H	6	56H	V	76H	v
17H	ETB	37H	7	57H	W	77H	w
18H	CAN	38H	8	58H	X	78H	x
19H	EM	39H	9	59H	Y	79H	y
1AH	SUB	3AH	:	5AH	Z	7AH	z
1BH	ESC	3BH	;	5BH	[	7BH	{
1CH	FS	3CH	<	5CH	\	7CH	|
1DH	GS	3DH	=	5DH	]	7DH	}
1EH	RS	3EH	>	5EH	^	7EH	~
1FH	US	3FH	?	5FH	_	7FH	DEL

그러나, IBM PC에서는 ASCII 코드를 8비트로 확장하여 정의하며, 표준 ASCII 문자의 경우에는 최상위 비트인 b7을 0으로 하여 128문자를 수용하고, b7=1인 128문자를 추가하여 확장 ASCII 코드(Extended ASCII code)라 한다. 확장 ASCII 코드에는 각종 외래어 문자나 세미 그래픽 문자 등이 정의되어 있으며, 우리나라의 경우에는 KS 규격에서 이 확장 ASCII

코드 영역을 이용하여 한글이나 한자 코드를 정의하고 있다.

참고로 ASCII 코드중에 기능제어 문자의 의미를 요약하면 <표 3>과 같고, 도형 문자에 정의되어 있는 특수한 기호들의 정확한 명칭을 정리하면 <표 4>와 같다.

<표 3> ASCII 코드에서의 기능제어 문자

ASCII 코드값	기능제어 문자	의 미
00H(0)	NUL	null(all zeros)
01H(1)	SOH	start of heading
02H(2)	STX	start of text
03H(3)	ETX	end of text
04H(4)	EOT	end of transmission
05H(5)	ENQ	enquiry
06H(6)	ACK	acknowledge
07H(7)	BEL	bell(beep)
08H(8)	BS	backspace
09H(9)	HT	horizontal tabulation
0AH(10)	LF	line feed
0BH(11)	VT	vertical tabulation
0CH(12)	FF	form feed
0DH(13)	CR	carriage return
0EH(14)	SO	shift out
0FH(15)	SI	shift in
10H(16)	DLE	data link escape
11H(17)	DC1	device control 1
12H(18)	DC2	device control 2
13H(19)	DC3	device control 3
14H(20)	DC4	device control 4
15H(21)	NAK	negative acknowledge
16H(22)	SYN	synchronous idle
17H(23)	ETB	end of transmission block
18H(24)	CAN	cancel
19H(25)	EM	end of medium
1AH(26)	SUB	substitute
1BH(27)	ESC	escape
1CH(28)	FS	file separator
1DH(29)	GS	group separator
1EH(30)	RS	record separator
1FH(31)	US	unit separator
7FH(127)	DEL	delete

<표 4> ASCII 코드에 정의된 특수문자들의 올바른 명칭

문자	명 칭	문자	명 칭
!	exclamation mark	:	colon
″	quotation mark	;	semicolon
#	number sign(sharp sign)	< >	less than / greater than
$	dollar sign	=	equal sign
%	percent	?	question mark
&	ampersand(and mark)	@	commercial at mark
′	apostrophe(acute accent)	[ ]	opening/closing bracket
( )	opening/closing parenthesis	\	backslash
*	asterisk	^	circumflex(circumflex accent)
+	plus sign	_	underscore(underline)
,	comma	`	single quotation mark(grave accent)
-	minus sign(hyphen, dash)	{ }	opening/closing brace
.	period	|	vertical bar
/	slash(slant)	~	tilde

하지만, 이처럼 ASCII 코드가 7비트로 정의되어 있음에도 불구하고 현실적으로 대부분의 마이크로프로세서에서는 이를 8비트 데이터로 표현할 수밖에 없으므로 여분으로 추가되는 최상위 비트인 b7을 0으로 하여 8비트 코드로 사용한다.

이제 2진수와 ASCII 코드를 비교해보면 컴퓨터에서 다른 컴퓨터나 주변장치와 통신을 수행할 때 왜 2진수를 그대로 전송하지 않고 ASCII 코드로 바꾸어 전송하는지를 쉽게 알 수 있다. 즉, 통신에서는 전송할 데이터 이외에 전송제어 문자가 필요한데, 2진수를 그대로 전송하면 00H~FFH 사이의 2진수중에서 일부가 불가피하게 전송제어 문자와 중복될 수밖에 없고, 그렇게 되면 수신측에서는 이를 데이터로 받아들여야할지 전송제어 문자로 받아들여야할지를 판단할 수 없게 된다.

그러나, ASCII 코드를 사용하면 전체 코드 영역 00H~FFH가 전송제어 문자와 도형문자로 나누어져 있으므로 이 2가지를 문제없이 전송할 수 있게 된다. 이를 위하여 ASCII 코드에서는 모든 2진수 값을 16진수로 표현하여 0~F의 도형문자로 변환되어야 한다. 예를들어 2진수 00010010B는 16진수로 12H가 되므로 "1"이라는 도형문자(ASCII 코드값 31H)와 "2"이라는 도형문자(ASCII 코드값 32H)로 바뀌어 전송된다. 그러므로 2진수일 때는 00H~FFH에 해당하던 값들이 ASCII 코드가 되면 불과 30H("0")~39H("9")와 41H("A")~46H("F")의 16가지 코드만을 사용하여 표현할 수 있게 되는 것이다.

이와 같이 2진수를 ASCII 코드로 변환하면 256가지의 값을 불과 16가지만의 코드로 표현할 수 있으므로 ASCII 코드에는 무려 240개의 코드 영역이 남아있게 된다. 이 공간을 이용하여 다른 알파벳 문자나 각종 기호와 같은 도형문자를 추가로 정의할 수 있고, 33개의 기능제어 문자까지를 정의할 수 있게 되는 것이다. 그러나, 이와 같이 1바이트의 2진수 데이터는 ASCII 코드로 변환하면 2바이트의 문자가 되므로, 2진수를 ASCII 코드로 변환하게 되면 당연히 데이터의 크기가 2배로 늘어난다.

2. 인텔 HEX 파일의 구조

인텔 HEX 파일은 마이크로프로세서용의 오브젝트 파일을 ROM 라이터나 ROM 에뮬레이터와 같은 주변장치에 전송하기 위하여 만든 ASCII 포맷의 파일로서 오늘날 어셈블러나 컴파일러에서 가장 널리 사용된다. 이 HEX 파일은 8비트 마이크로프로세서용으로 사용되는 16비트 선형 어드레스(linear address) 형식은 물론이고, 8086/8088과 같은 16비트 마이크로프로세서를 위한 20비트 세그먼트 어드레스(segmented address) 형식이나 80386 이상의 32비트 마이크로프로세서를 위한 32비트 선형 어드레스 형식까지 지원한다. HEX 파일에는 이밖에도 ASCII-Hex 파일 형식, Motorola HEX 파일 형식, Tektronix HEX 파일 형식 등이 있는데, 이것들은 표현 형식이 서로 매우 다른 구조로 되어 있다.

인텔 HEX 파일은 정확하게 "Intel Hexadecimal Object File Format"이라고 부르는데, 이 파일은 기본적으로 1행씩으로 되어있는 많은 레코드(record)들로 구성되며, 각 레코드는

<그림 1>과 같이 항상 6개의 필드(field)로 이루어진다. 인텔 HEX 파일에 사용되는 모든 문자는 ASCII 문자이며, 각 레코드의 끝에는 구분기호(delimiter)로서 캐리지 리턴(0DH) 및 라인 피드(0AH) 문자가 추가되어 있으나 사람의 눈에는 보이지 않는다. 각 레코드의 첫번째 필드인 ":"(3AH)와 마지막의 구분기호를 제외한 모든 문자들은 수치를 나타내는데, 이 수치들은 모두 16진 ASCII 문자로 표현하므로 1바이트의 2진수가 2바이트의 ASCII 문자로 되는데 유의해야 한다.

RECORD MARK	RECLEN	LOAD OFFSET	RECTYP	INFO or DATA	CHKSUM
1 바이트	1 바이트	2 바이트	1 바이트	n 바이트	1 바이트

<그림 1> 인텔 HEX 파일의 레코드 구성 형식

첫번째의 RECORD MARK 필드(Start Character)는 1문자인 ":"로 구성되는데 이것은 한 레코드의 시작을 의미한다.

두번째의 RECLEN 필드(Byte Count)는 2문자로 구성되는데 이것은 이 레코드에 포함된 데이터의 바이트 수(n)로서, 이 데이터 바이트는 각 레코드의 앞뒤에 붙는 기능적인 필드를 제외한 5번째 필드의 순수한 데이터만을 의미한다. 이 n값은 0～255가 될 수 있으며, 따라서 이 필드는 "00"～"FF"로 표시된다. <그림 2>의 첫번째 레코드에서 이 필드가 "10"인 것은 10H로서 이 레코드에 16바이트의 데이터가 포함되어 있음을 나타낸다.

세번째의 LOAD OFFSET 필드(Address)는 4문자로 구성되는데 이것은 이 레코드에 포함된 데이터를 로드하기 시작할 메모리의 오프셋 번지를 나타낸다. 이것이 16진수로 4자리라는 것은 어드레스를 항상 16비트로 나타낸다는 것을 의미하므로 16비트 어드레스 형식에서는 이것만으로 문제가 없으나 그 이상의 어드레스값이 필요한 경우에는 여기에 추가적인 정보가 필요하며, 따라서 20비트나 32비트 어드레스 형식에서는 나중에 설명하는 방법으로 상위 어드레스를 지정함으로써 필요한 어드레스값을 표현한다. <그림 2>의 첫번째 레코드에서 이 필드가 "80EC"인 것은 이 레코드에 포함된 16바이트의 데이터가 80ECH 번지부터 로드되어야 한다는 것을 나타낸다.

네번째의 RECTYP 필드(Record Type)는 2문자로 구성되는데 이것은 이 레코드의 형식을 나타낸다. 이것이 "00"이면 정상적인 데이터 레코드(Data Record)를 의미하며, 이것이 "01"이면 인텔 HEX 파일의 마지막 행을 표시하는 레코드(End-of-file Record)라는 것을 의미한다. 그러나, 32비트 어드레스 형식의 인텔 HEX 파일에서는 첫번째 레코드의 레코드 형식이 "04"로 되어 있는데, 이것은 이제부터 사용될 16비트 어드레스의 상위에 붙는 16비트

어드레스를 지정하여 이것들의 합으로 32비트 어드레스를 표현하기 위한 것(Extended Linear Address Record)이다.

레코드의 형에는 다음과 같이 6가지가 있는데, 이중에 “00”~“01”형은 항상 기본적으로 사용되는 레코드이지만, “02”~“03”형은 20비트 세그먼트 어드레스 형식에서만 사용되며, “04”~“05”형은 32비트 선형 어드레스 형식에서만 사용된다.

① “00” - Data Record (16/20/32비트 어드레스 형식에 모두 사용)
② “01” - End of File Record (16/20/32비트 어드레스 형식에 모두 사용)
③ “02” - Extended Segmented Address Record(20비트 어드레스 형식에서만 사용)
④ “03” - Start Segmented Address Record (20비트 어드레스 형식에서만 사용)
⑤ “04” - Extended Linear Address Record (32비트 어드레스 형식에서만 사용)
⑥ “05” - Start Linear Address Record (32비트 어드레스 형식에서만 사용)

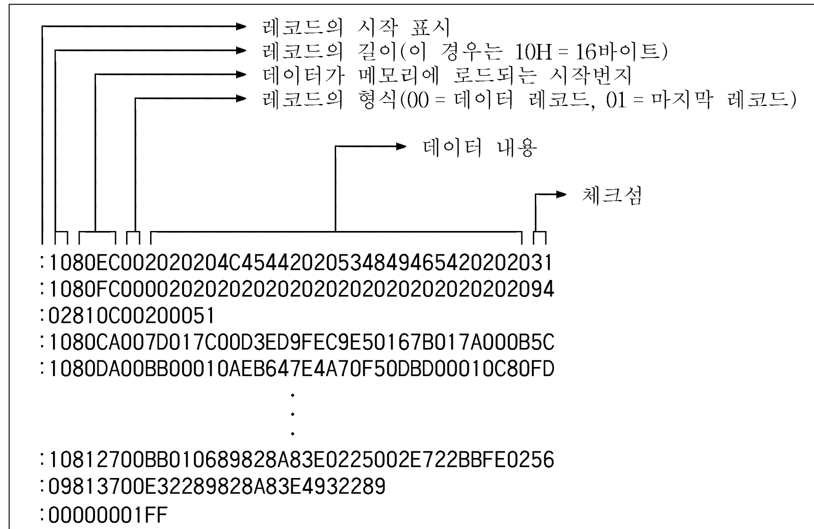
32비트 선형 어드레스 형식은 32비트의 어드레스중에서 하위 16비트는 각 레코드의 3번째 필드인 LOAD OFFSET으로 지정하고, 상위 16비트는 이 레코드 형식 “04”에서 2바이트(4문자)의 데이터로 지정한다. 레코드 형식 “05”는 이같은 32비트 선형 어드레스 형식에서 프로그램의 실행 시작번지를 지정하는 레코드로 사용되는데, 이때는 5번째 필드에서 4바이트(8문자)의 데이터로 32비트 선형 어드레스인 EIP 레지스터의 값을 지정한다.

20비트 세그먼트 어드레스 형식은 20비트의 어드레스를 상위 16비트의 세그먼트 레지스터값과 하위 16비트의 오프셋 값으로 나타낸다. 이때 오프셋값은 각 레코드의 3번째 필드인 LOAD OFFSET으로 지정하고, 상위 16비트 세그먼트값은 이 레코드 형식 “02”에서 2바이트(4문자)의 데이터로 지정한다. 레코드 형식 “03”은 이같은 20비트 세그먼트 어드레스 형식에서 프로그램의 실행 시작번지를 지정하는 레코드로 사용되는데, 이때는 5번째 필드에서 4바이트(8문자)의 데이터중에 앞의 2바이트(4문자)는 16비트 세그먼트 CS 레지스터의 값을 지정하고 뒤의 2바이트(4문자)는 16비트 오프셋 IP 레지스터의 값을 지정하고 한다.

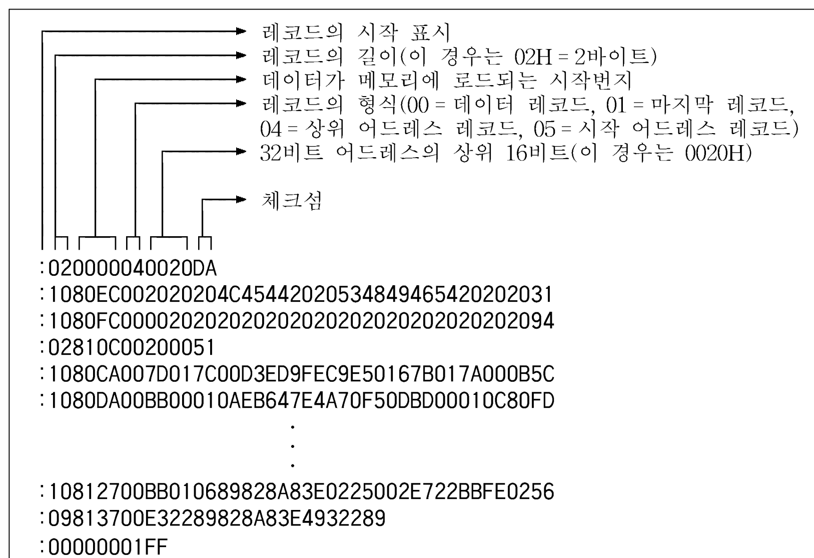
<그림 2>에서 보면 모든 데이터 레코드는 레코드 형식이 “00”으로 되어 있고, 마지막 행만 파일 종료 레코드인 “01”로 되어 있는 것을 볼 수 있다. 그러나, 32비트 어드레스 형식의 인텔 HEX 파일에서는 <그림 3>에 보인 바와 같이 첫번째 레코드에서 레코드 형식이 “04”로 되어 있어서 32비트 선형 어드레스의 상위 16비트에 해당하는 “0020”을 지정하고 있다. 따라서, 이 파일의 데이터는 002080ECH 번지부터 데이터를 로드하게 된다.

다섯번째의 DATA 필드(Data)는 레코드형 “00”에서 Byte Count 필드에서 지정한 바이트수(n)만큼의 데이터로 구성되며, 레코드형이 “01”인 경우에는 이 필드가 존재하지 않게 되고, 레코드형이 “02”~“05”의 경우에는 INFO 필드로서 각각에 해당하는 어드레스값이 된다. 이 바이트수 n은 하나의 레코드를 1행에 표시할 것을 고려하여 보통 10H 또는 20H로 지정

하는 것이 일반적이지만, 불연속된 메모리 데이터나 HEX 파일 마지막의 자투리 바이트가 되면 그 이하의 바이트 수로 처리되기도 한다.



<그림 2> 인텔 HEX 파일의 구조(16비트 어드레스 형식)



<그림 3> 인텔 HEX 파일의 구조(32비트 선형 어드레스 형식)

여섯번째의 CHKSUM 필드(Checksum)는 2문자로 구성되는데 이는 전송중에 에러가 발생했는지의 여부를 체크하기 위한 코드이다. 이것은 첫번째 필드인 ":"를 제외한 2~5번째 필드에서 차례로 2문자씩을 8비트의 2진수로 변환하여 더한 후에 그 총합을 2의 보수로 취한 것이다. 예를 들어서 <그림 2>의 마지막 레코드에서는 00H+00H+00H+01H = 01H가 되

므로 이것의 2의 보수에 해당하는 FFH가 체크섬 값이 되고, <그림 3>의 첫번째 레코드에서는 02H+00H+00H+04H+00H+20H = 26H가 되므로 이것의 2의 보수인 DAH가 체크섬이 된다.

8051이나 80C196KC에서 널리 사용되는 대표적인 인텔 HEX 파일인 16비트 어드레스 형식의 파일을 <그림 2>에 예로 보였고, 32비트 마이크로프로세서에서 주로 사용하는 32비트 선형 어드레스 형식의 HEX 파일 예를 <그림 3>에 보였다.

마이크로프로세서 트레이닝 키트에서 사용자 프로그램을 다운로드하는 모니터 프로그램을 작성하려면 당연히 이러한 인텔 HEX 파일의 구조를 잘 알아야 한다. 모니터 프로그램이 사용자 프로그램을 RS-232C로 수신하여 RAM에 로드하려면 이들 파일의 구조에 따라 필요한 메모리 어드레스와 프로그램 데이터를 추출하고 이를 2진수값으로 변환하여 해당 메모리 번지에 로드해야 하기 때문이다.

3. 바이너리 파일과 인텔 HEX 파일의 비교

흔히 마이크로프로세서의 초보자들은 바이너리 파일과 인텔 HEX 파일의 차이점을 이해하지 못하여 어려움을 겪는 수가 있다. 그러나, 마이크로프로세서 엔지니어라면 이러한 사항들을 잘 알아야 하며, 이 2가지의 파일형식의 차이나 그 장단점을 이해하는 것은 결코 어려운 일이 아니다.

첫째로, 바이너리 파일은 파일의 내용이 8비트 2진수로만 구성되므로 각 바이트는 00H~FFH 범위의 임의의 값을 가질 수 있다. 따라서, 이는 텍스트 에디터로 편집할 수 없으며, 그대로는 모니터나 프린터와 같은 출력장치에 출력할 수도 없다. 또한, 이것은 기능제어 문자와 중복되는 값을 가지므로 ASCII 코드를 기본으로 하는 직렬통신으로 전송하는 것도 불가능하다.

그러나, 인텔 HEX 파일은 ASCII 파일로서 모든 문자는 “0”~“9” 및 “A”~“F”의 도형문자로 이루어지며, 각 레코드의 끝에 캐리지 리턴(0DH) 및 라인 피드(0AH) 정도의 기능제어 문자가 포함된다. 따라서, 이는 필요하면 텍스트 에디터로 직접 편집할 수도 있으며, 그대로 모니터나 프린터와 같은 출력장치에 출력할 수도 있고, 직렬통신으로 전송할 수도 있다.

참고로 HEX 파일이 아닌 일반 ASCII 파일은 여기에 모든 알파벳 대소문자와 각종 기호 및 기능제어 문자의 HT(TAB, 09H) 등의 문자들이 사용된다. 이러한 ASCII 파일을 텍스트 파일(text file)이라고도 부르며, 또한 이를 편집하는 소프트웨어를 텍스트 에디터(text editor)라고 한다. 우리가 컴퓨터에서 어셈블리 언어 또는 C언어로 소스 프로그램을 작성하려면 반드시 이와 같은 텍스트 에디터를 필요로 한다.

둘째로, 바이너리 파일은 모든 데이터를 00H~FFH 사이의 2진수로 표현하는데 비하여 인텔 HEX 파일은 이를 16진수 ASCII 문자로 변환하여 표현한다. 따라서, 8비트 데이터가 16진수 ASCII 문자로 변환되면 2개의 문자가 되므로 HEX 파일은 기본적으로 바이너리 파일에 비하여 길이가 2배로 길어진다.

셋째로, 바이너리 파일은 00H~FFH 사이의 2진수 데이터만으로 구성되며, 이를 로드할 메모리 번지와 같은 기타 어떤 정보도 포함하지 않는다. 이에 비하여 인텔 HEX 파일은 기본적인 데이터의 앞뒤에 여러 가지 필드를 구성하는 정보들이 추가로 포함된다. 따라서, 이러한 점까지 고려하면 인텔 HEX 파일은 바이너리 파일에 비하여 전체 파일의 길이가 2배를 훨씬 넘게 된다.

그러나, 여기에는 전혀 다른 문제가 있다. 즉, 바이너리 파일은 데이터를 로드할 메모리 번지 정보를 포함하지 않기 때문에 무조건 0번지부터 데이터를 로드한다. 따라서, 프로그램이 0번지부터 시작되지 않는 오브젝트 파일을 바이너리로 표현할 경우에는 0번지부터 프로그램의 시작 부분까지를 어떤 임의의 값(일반적으로 00H 또는 FFH)으로 채워두게 된다. 그러나, 인텔 HEX 파일은 각 레코드의 3번째 필드에 어드레스 정보를 가지고 있기 때문에 임의의 어드레스에서부터 데이터를 로드할 수 있으며, 따라서 그 앞부분의 빈 공간을 임의의 데이터로 채워놓을 필요가 없다. 이러한 이유 때문에 프로그램이 0번지부터 시작되지 않는 경우나 프로그램의 중간에 빈 메모리 공간을 포함하여 메모리의 번지가 불연속으로 되어 있는 경우에는 오히려 인텔 HEX 파일보다도 바이너리 파일이 훨씬 더 긴 경우가 많다. 그러기 때문에 이러한 사정을 고려하지 않고 일반적으로 바이너리 파일과 인텔 HEX 파일의 길이를 비교하는 것은 아무 의미가 없다.

자, 그러면 여기서 인텔 HEX 파일을 정확하게 이해하기 위하여 간단한 8051 프로그램을 통하여 생성된 인텔 HEX 파일과 이를 바이너리 파일로 만든 경우를 비교하여 보기로 한다. 다음은 80C32를 이용한 *OK-8051* 키트에서 간단한 어셈블리 프로그램 예를 작성한 것으로서 파일명이 TEST.LST이다. 프로그램의 서두에 인터럽트 벡터를 지정하기 위하여 메모리 번지가 불연속적으로 사용된 것에 주목하기 바란다. 그리고, 프로그램은 0074H 번지에서 끝났다는 것을 기억하라.

```
================================================================================
0000                    ORG     0000H
0000 020030             JMP     START               ; go to boot routine after reset

                ;-------------------------------------------------
                ;       Interrupt Vector Table
                ;-------------------------------------------------
0003                    ORG     0003H
0003 02FF03             LJMP    0FF03H              ; external INT0 interrupt vector
000B                    ORG     000BH
```

```
000B 02FF0B          LJMP    0FF0BH            ; timer 0 interrupt vector
0013                 ORG     0013H
0013 02FF13          LJMP    0FF13H            ; external INT1 interrupt vector
001B                 ORG     001BH
001B 02FF1B          LJMP    0FF1BH            ; timer 1 interrupt vector
0023                 ORG     0023H
0023 02FF23          LJMP    0FF23H            ; serial interrupt vector
002B                 ORG     002BH
002B 02FF2B          LJMP    0FF2BH            ; timer 2 interrupt vector

               ;-----------------------------------------------
               ;       Main Program
               ;-----------------------------------------------
0030                 ORG     0030H
0030 75817F    START: MOV    SP,#7FH           ; initialize temporary SP
0033 120048          CALL    DY100MS           ; time delay for booting
0036 7489            MOV     A,#10001001B      ; A = B = output, C = input
0038 901203          MOV     DPTR,#PPI_CW
003B F0              MOVX    @DPTR,A

003C 901200    LOOP: MOV     DPTR,#PPI_PORTA   ; blink LED1 - LED4
003F E0              MOVX    A,@DPTR
0040 640F            XRL     A,#00001111B
0042 F0              MOVX    @DPTR,A
0043 120050          CALL    DY300MS           ; time delay for 300 ms
0046 80F4            JMP     LOOP

               ;-----------------------------------------------
               ;       Time Delay Subroutines
               ;-----------------------------------------------
0048 C050      DY100MS:PUSH  TIMECOUNT0
004A 755064          MOV     TIMECOUNT0,#100
004D 020057          JMP     DELAY1MS

0050 1148      DY300MS:CALL  DY100MS
0052 1148            CALL    DY100MS
0054 1148            CALL    DY100MS
0056 22              RET

0057 C051      DELAY1MS:PUSH TIMECOUNT1
0059 755163    DELAY1MS1:MOV TIMECOUNT1,#99          ; (2)
005C C051      DELAY1MS2:PUSH TIMECOUNT1           ; 2
005E C051            PUSH    TIMECOUNT1              ; 2
0060 D051            POP     TIMECOUNT1              ; 2
0062 D051            POP     TIMECOUNT1              ; 2
0064 D551F5          DJNZ    TIMECOUNT1,DELAY1MS2    ; 2
0067 00              NOP                             ; (1)
0068 C051            PUSH    TIMECOUNT1              ; (2)
006A D051            POP     TIMECOUNT1              ; (2)
006C 00              NOP                             ; (1)
006D D550E9          DJNZ    TIMECOUNT0,DELAY1MS1    ; (2)
0070 D051            POP     TIMECOUNT1
0072 D050            POP     TIMECOUNT0
0074 22              RET
                     END
==============================================================================
```

이를 Keil사의 매크로 어셈블러 A51.EXE로 어셈블하고 링크하여 최종적으로 만들어진 인텔 HEX 파일 TEST.HEX는 다음과 같다. 프로그램의 서두에서 메모리 번지들이 3번지씩만 사용되면서 불연속적으로 이루어져 있었기 때문에 HEX 파일에서 처음의 7개 레코드들은 각각 데이터 바이트 수가 3바이트씩으로 이루어져 있고, 그 다음부터는 정상적으로 16바이트씩으로 이루어지며, 마지막의 데이터 레코드는 0070H~0074H의 5바이트로 되기 때문에 불가피하게 자투리 레코드로 처리되었다는 점에 주목하라.

```
==========================================================================
:03000000020030CB
:0300030002FF03F6
:03000B0002FF0BE6
:0300130002FF13D6
:03001B0002FF1BC6
:0300230002FF23B6
:03002B0002FF2BA6
:1000300075817F1200487489901203F0901200E0DD
:10004000640FF012005080F4C050755064020057E5
:100050001148114811482 2C051755163C051C05117
:10006000D051D051D551F500C051D05100D550E9F3
:05007000D051D0502228
:00000001FF
==========================================================================
```

이를 바이너리 파일 TEST.BIN으로 변환하여 보면 아래와 같다. 여기서 각 행의 선두에 있는 ( )안의 부분은 실제로 바이너리 파일에는 없는 내용이며, 단지 여러분이 각 데이터의 번지를 알아보기 쉽도록 표시한 것이다. 여기서는 불연속적인 메모리 공간이 모두 00H로 채워져 있음에 주목하라. 인텔 HEX 파일을 바이너리 파일로 변환하는 대부분의 유틸리티에서는 이러한 메모리의 빈 공간을 00H로 채울지 또는 FFH로 채울지 물어오는데, 여기서는 00H로 지정하였다.

```
==========================================================================
(00000)  02 00 30 02 FF 03 00 00 00 00 00 02 FF 0B 00 00
(00010)  00 00 00 02 FF 13 00 00 00 00 00 02 FF 1B 00 00
(00020)  00 00 00 02 FF 23 00 00 00 00 00 02 FF 2B 00 00
(00030)  75 81 7F 12 00 48 74 89 90 12 03 F0 90 12 00 E0
(00040)  64 0F F0 12 00 50 80 F4 C0 50 75 50 64 02 00 57
(00050)  11 48 11 48 11 48 22 C0 51 75 51 63 C0 51 C0 51
(00060)  D0 51 D0 51 D5 51 F5 00 C0 51 D0 51 00 D5 50 E9
(00070)  D0 51 D0 50 22 00 00 00 00 00 00 00 00 00 00 00
==========================================================================
```

이렇게 인텔 HEX 파일을 바이너리 파일로 변환하는 유틸리티 소프트웨어에는 오래전의 MS-DOS 시절에 개발된 HEX2BIN.EXE나 HEXOBJ.EXE 등이 있다.

이 프로그램의 경우에 TEST.HEX는 파일 길이가 349바이트였고 TEST.BIN은 128바이트였다. 그러나, 앞에서 설명한대로 이렇게 프로그램 영역의 중간에 빈 메모리 공간이 존재하는 경우에는 그 공간의 크기에 따라 HEX 파일이 더 클 수도 있고 바이너리 파일이 클 수도 있으므로 서로 파일 길이를 비교한다는 것이 전혀 무의미하다.

【 참고 문헌 】

1. Hexadecimal Object File Format Specification, Intel, 1988
2. 윤덕용, 어셈블리와 C언어로 익히는 8051 마스터, Ohm사, 2001
3. 윤덕용, 어셈블리와 C언어로 익히는 80C196KC 마스터 (I), Ohm사, 2000