2009년 06월 19일

ELF format Object File에 관한 진실. -c option (기계어 세상)

ELF 하면, 요정 엘프가 생각이 납니다. 제가 난생처음 ELF format이라는 걸 알게 되었을 때, ELF format이 도대체 무엇일까 하고 몇 개의 검색 포털에서 ELF를 무작정 찾아본 적이 있었습니다. 정말 '아연하게도' 요정 엘프에 관한 것들이 왜 그리도 많이 나오던지, 원하는 것을 찾기가 힘들었습니다. 지금은 ELF format이라는 것이 검색로봇에 많이 검색되어 검색엔진에서 적당한 자료를 찾을 수 있는 것인지는 잘은 모르겠습니다. 그 당시에는 과연 ELF는 숭고한 요정과 같이 나랑은 거리가 먼 것일까 하고 결론 내어 버린 적이 있었습니다만... 검색엔진이란 참으로 편리하면서도 때로는 어이없다는 생각이 들고 말았습니다.

ELF는 Executable and Linking Format을 의미하며, 말 그대로 실행 가능한 그리고 링크를 하는 형식을 말합니다. 상당히 고무적인 표현이랄까 - 뭔가를 한다는 의미에서 - 하지만, 막연한 느낌이 드는 건 사실이지요. 그래서 말인데, 이번에는 compile 후에 나오는 ELF 형식을 따르는 .o (object file)을 분석해 보겠습니다. - 사실은 Assembler의 output임을 잊지 맙시다 -

spaghetti.c 를 object file로 만들려면 다음과 같이 합니다. (-c option은 tcc에게 linker로 하여금 link를 하지 말라는 의미 입니다. 그렇게 하면 최종 실행 가능한 file이 아니라, link가 가능한 object file로 만들겠지요. 이런 file을 relocatable file이라고 부르는데, 나중에 link를 통해서 재배치가 가능하다는 의미 입니다. 너무 뜬 구름 잡기 식인데, 쉽게 얘기하면, relocatable이란, spaghetti.c만 컴파일 한 것이지, 아직 실행 가능하게 만든 건 아니라는 의미이지요. 결국 spaghetti.c를 Assembler에 통과시키고, 그것을 link가능하게 table형태로 만들어 놓은 것입니다) 다시 말해, ELF format Object file은 Assembler의 output 이지요.

tcc -c spaghetti.c

이렇게 하여 나온 output은 spaghetti.o 라는 파일인데, 이 파일은 인간이 알아볼 수 있는 형식으로 되어 있지 않습니다. 한번 열어 보도록 하겠습니다.

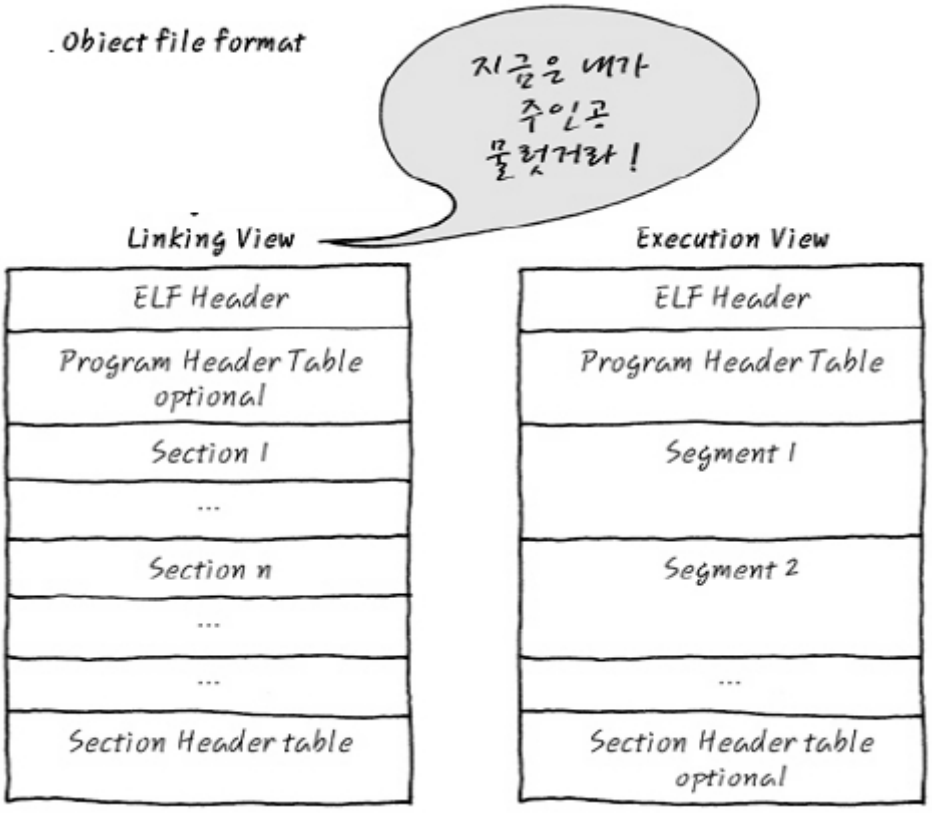
00000000h: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 ; ELF.....
00000010h: 01 00 28 00 01 00 00 00 00 00 00 00 00 00 00 00 ; ..(.....
00000020h: 5C 03 00 00 00 00 00 02 34 00 20 00 00 00 28 00 ; ₩.....4. ...(.
00000030h: 0B 00 01 00 03 20 04 21 40 18 10 18 70 47 00 00 ; ..... .!@...pG..
00000040h: 03 00 00 00 03 00 00 00 28 00 00 00 FF FF FF FF ; ....(...□□□□
00000050h: 01 00 02 7C 0E 0C 0D 00 07 00 07 01 07 02 07 03 ; ...|.....
00000060h: 08 04 08 05 08 06 08 07 08 08 08 09 08 0A 08 0B ; .....
00000070h: 07 0C 08 0E 0C 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000080h: 0A 00 00 00 54 68 75 6D 62 20 43 20 43 6F 6D 70 ; ....Thumb C Comp
00000090h: 69 6C 65 72 2C 20 41 44 53 31 2E 32 20 5B 42 75 ; iler, ADS1.2 [Bu
000000a0h: 69 6C 64 20 38 30 35 5D 00 00 00 00 2D 4F 32 20 ; ild 805]....-O2
000000b0h: 2D 49 43 3A 5C 61 70 70 73 5C 61 64 73 31 32 5C ; -IC:₩apps₩ads12₩
000000c0h: 49 4E 43 4C 55 44 45 00 00 00 00 00 00 00 00 00 ; INCLUDE.....
000000d0h: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 ; .....
000000e0h: 00 00 00 00 02 00 05 00 04 00 00 00 0A 00 00 00 ; .....
000000f0h: 00 00 00 00 02 00 05 00 07 00 00 00 00 00 00 00 ; .....
00000100h: 00 00 00 00 04 00 F1 FF 11 00 00 00 00 00 00 00 ; .....?.....
00000110h: 00 00 00 00 01 00 06 00 18 00 00 00 00 00 00 00 ; .....
00000120h: 00 00 00 00 03 00 06 00 1D 00 00 00 00 00 00 00 ; .....
00000130h: 00 00 00 00 01 00 07 00 26 00 00 00 00 00 00 00 ; .....&.....
00000140h: 00 00 00 00 03 00 07 00 2C 00 00 00 00 00 00 00 ; .....,.....

```
00000150h: 00 00 00 00 01 00 08 00 35 00 00 00 00 00 00 00 ; ........5.......
00000160h: 00 00 00 00 03 00 08 00 3B 00 00 00 00 00 00 00 ; ........;.......
00000170h: 00 00 00 00 03 00 05 00 41 00 00 00 00 00 00 00 ; ........A.......
00000180h: 00 00 00 00 01 00 09 00 50 00 00 00 00 00 00 00 ; ........P.......
00000190h: 00 00 00 00 00 00 F1 FF A0 00 00 00 00 00 00 00 ; ......??........
000001a0h: 00 00 00 00 22 00 00 00 B5 00 00 00 00 00 00 00 ; ....".../?......
000001b0h: 04 00 00 00 11 00 06 00 B8 00 00 00 00 00 00 00 ; ........?.......
000001c0h: 04 00 00 00 11 00 07 00 BB 00 00 00 00 00 00 00 ; ........?.......
000001d0h: 04 00 00 00 11 00 08 00 C4 00 00 00 00 00 00 00 ; ........?.......
000001e0h: 0A 00 00 00 12 00 05 00 C9 00 00 00 00 00 00 00 ; ........?.......
000001f0h: 00 00 00 00 12 00 00 00 D0 00 00 00 00 00 00 00 ; ........?.......
00000200h: 00 00 00 00 12 00 00 00 00 24 74 00 24 64 00 72 ; .........$t.$d.r
00000210h: 65 63 69 70 65 73 2E 63 00 2E 62 73 73 24 37 00 ; ecipes.c..bss$7.
00000220h: 2E 62 73 73 00 2E 64 61 74 61 24 31 30 00 2E 64 ; .bss..data$10..d
00000230h: 61 74 61 00 2E 64 61 74 61 24 31 35 00 2E 64 61 ; ata..data$15..da
00000240h: 74 61 00 2E 74 65 78 74 00 43 24 64 65 62 75 67 ; ta..text.C$debug
00000250h: 5F 66 72 61 6D 65 31 00 42 75 69 6C 64 41 74 74 ; _frame1.BuildAtt
00000260h: 72 69 62 75 74 65 73 24 24 54 48 55 4D 42 5F 49 ; ributes$$THUMB_I
00000270h: 53 41 76 31 24 4D 24 50 45 24 41 3A 4C 32 32 24 ; SAv1$M$PE$A:L22$
00000280h: 58 3A 4C 31 31 24 53 32 32 24 7E 49 57 24 7E 53 ; X:L11$S22$~IW$~S
00000290h: 54 4B 43 4B 44 24 7E 53 48 4C 24 4F 53 50 41 43 ; TKCKD$~SHL$OSPAC
000002a0h: 45 24 50 52 45 53 38 00 4C 69 62 24 24 52 65 71 ; E$PRES8.Lib$$Req
000002b0h: 75 65 73 74 24 24 61 72 6D 6C 69 62 00 7A 69 00 ; uest$$armlib.zi.
000002c0h: 72 77 00 72 65 6C 6F 63 61 74 65 00 6D 61 69 6E ; rw.relocate.main
000002d0h: 00 5F 5F 6D 61 69 6E 00 5F 6D 61 69 6E 00 00 00 ; .__main._main...
000002e0h: 30 00 00 00 02 0B 00 00 34 00 00 00 02 0A 00 00 ; 0.......4.......
000002f0h: 00 2E 73 68 73 74 72 74 61 62 00 2E 73 79 6D 74 ; ..shstrtab..symt
00000300h: 61 62 00 2E 73 74 72 74 61 62 00 2E 63 6F 6D 6D ; ab..strtab..comm
00000310h: 65 6E 74 00 2E 74 65 78 74 00 2E 62 73 73 00 2E ; ent..text..bss..
00000320h: 64 61 74 61 00 2E 64 61 74 61 00 2E 64 65 62 75 ; data..data..debu
00000330h: 67 5F 66 72 61 6D 65 24 24 24 2E 74 65 78 74 00 ; g_frame$$$.text.
00000340h: 2E 72 65 6C 2E 64 65 62 75 67 5F 66 72 61 6D 65 ; .rel.debug_frame
00000350h: 24 24 24 2E 74 65 78 74 00 00 00 00 00 00 00 00 ; $$$.text........
00000360h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000370h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000380h: 00 00 00 00 01 00 00 00 03 00 00 00 00 00 00 00 ; ................
00000390h: 00 00 00 00 F0 02 00 00 69 00 00 00 00 00 00 00 ; ....?..i........
000003a0h: 00 00 00 00 00 00 00 00 00 00 00 00 0B 00 00 00 ; ................
000003b0h: 02 00 00 00 00 00 00 00 00 00 00 00 C8 00 00 00 ; ............?...
000003c0h: 40 01 00 00 03 00 00 00 0D 00 00 00 00 00 00 00 ; @...............
000003d0h: 10 00 00 00 13 00 00 00 03 00 00 00 00 00 00 00 ; ................
000003e0h: 00 00 00 00 08 02 00 00 D6 00 00 00 00 00 00 00 ; ........?.......
000003f0h: 00 00 00 00 00 00 00 00 00 00 00 00 1B 00 00 00 ; ................
00000400h: 01 00 00 00 00 00 00 00 00 00 00 00 84 00 00 00 ; ............?...
00000410h: 44 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; D...............
00000420h: 00 00 00 00 24 00 00 00 01 00 00 00 06 00 00 00 ; ....$...........
00000430h: 00 00 00 00 34 00 00 00 0C 00 00 00 00 00 00 00 ; ....4...........
00000440h: 00 00 00 00 04 00 00 00 00 00 00 00 2A 00 00 00 ; ............*...
00000450h: 08 00 00 00 03 00 00 00 00 00 00 00 40 00 00 00 ; ............@...
00000460h: 04 00 00 00 00 00 00 00 00 00 00 00 04 00 00 00 ; ................
00000470h: 00 00 00 00 2F 00 00 00 01 00 00 00 03 00 00 00 ; ..../...........
00000480h: 00 00 00 00 40 00 00 00 04 00 00 00 00 00 00 00 ; ....@...........
00000490h: 00 00 00 00 04 00 00 00 00 00 00 00 35 00 00 00 ; ............5...
000004a0h: 01 00 00 00 03 00 00 00 00 00 00 00 44 00 00 00 ; ............D...
000004b0h: 04 00 00 00 00 00 00 00 00 00 00 00 04 00 00 00 ; ................
```

000004c0h: 00 00 00 00 3B 00 00 00 01 00 00 00 00 00 00 00 ; ....;.....
000004d0h: 00 00 00 00 48 00 00 00 3C 00 00 00 00 00 00 00 ; ....H...<.....
000004e0h: 00 00 00 00 00 00 00 00 00 00 00 00 50 00 00 00 ; ....P...
000004f0h: 09 00 00 00 00 00 00 00 00 00 00 00 E0 02 00 00 ; ....?..
00000500h: 10 00 00 00 02 00 00 00 09 00 00 00 00 00 00 00 ; ....
00000510h: 08 00 00 00 ; ....

음..... 음.. 상당히 뭔가 있어 보입니다만, 머리 속이 복잡해 지는군요. 인간의 눈으로는 모르는 것이 당연합니다. 이제부터는 기계어 세상이 되어 버렸으니까요. (ELF 규격을 따른) 이 숫자들의 연속인 것을 알아보려면, 앞에서 말한 ELF format에 대하여, 조금 알아야 합니다.

ELF에 대하여, 조금 알아보도록 하지요. ARM ELF specification을 인터넷에서 찾아보시면, (참고로 문서번호는 SWS ESPC 0003 B-02, 8 June, 2001 복잡하다 흡) 그 안에서 다음과 같은 그림을 찾으실 수 있으실 것입니다
.



두 가지의 view가 그려져 있는데, 별건 아니고, Linking View가 relocatable file의 형식입니다. 쉽게 말해서 Link하기전의 object file (.o)은 Linking View이고요, Link가 끝난 후에 완전한 실행 가능한 형태가 된 ELF 형식을 Execution View라고 부릅니다. (Link 하려고 보니까, 다른 object file하고 연결 하려고 여러 가지 정보들을 심어 놓은 것 뿐입죠.) 위에서 만들어 낸 spaghetti.o는 위의 object file format중 Linking View를 따르고 있다고 봐야 합니다. 오, 별거 아닐 거 같다는 생각이 듭니다. 좀 더 파고 들어가자면, 예를 들어, ELF Header중 한 개만 분석해 보면 감이 팍팍 올 것입니다. 아래는 ELF Header가 어떻게 구성되어 있는가를 구체적으로 볼 수 있는 구조체 입니다. ELF Header에는 ELF file의 특징을 결정 짓는 Endian, 운영체제, CPU 정보들 더하기 내부의 각 Section의 시작 위치 (offset)과 크기 정보가 들어 있습니다. 많은 양의 member들이 있습니다만, 겁부터 집어먹으면 안됩니다. 딱 한 개만 해보면 다른 것도 다 똑같은 원리로 분석이 가능하니, 걱정은 전혀 붙들어 매시라니까요.

```
#define EI_NIDENT 16
typedef struct {
        unsigned char e_ident[EI_NIDENT];
        Elf32_Half e_type;
        Elf32_Half e_machine;
```

```
    Elf32_Word e_version;
    Elf32_Addr e_entry;
    Elf32_Off e_phoff;
    Elf32_Off e_shoff;
    Elf32_Word e_flags;
    Elf32_Half e_ehsize;
    Elf32_Half e_phentsize;
    Elf32_Half e_phnum;
    Elf32_Half e_shentsize;
    Elf32_Half e_shnum;
    Elf32_Half e_shstrndx;
} Elf32_Ehdr;
```

이중에서 가장 만만해 보이는 걸 하나 잡아 봅시다. 맨 앞에 e_ident는 (IDENTIFICATION)은 재미 없어 보이니까, 그 다음에 보이는 e_type을 분석해 보기로 마음먹고 자자 레츠고. Object file이 시작하게 되면, 16 byte의 e_ident를 지나 Elf32_Half e_type을 만나게 됩니다. Elf32_Half는 눈치 채신 바와 같이 32 bit의 반, 16 bit이며, 이는 2 byte를 의미합니다. 그리고, ARM Elf specification을 찾아보니, e_type은 다음과 같은 값들을 가질 수 있습니다.

- e_type :

Name	Value	Meaning
ET_NONE	0	No file type
ET_REL	1	Relocatable file
ET_EXEC	2	Executable file
ET_DYN	3	Shared object file
ET_CORE	4	Core file
ET_LOPROC	0xff00	Processor-specific
ET_HIPROC	0xffff	Processor-specific

어허 가만히 보니까, e_type이라는건 file의 종류가 어떤 거냐는 것을 표기해 놓은 것이네요. 1이면 Relocatable file 2면 Executalbe file 등이네요. 그러면, 지금 살펴보던 file은 -c option을 이용하여 relocatable file을 만들었으니까, 값은 1을 가져야 하겠습니다. 진짜로 그런가 확인해 봅시다.

```
00000000h: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 ; ELF.....
00000010h: 01 00 28 00 01 00 00 00 00 00 00 00 00 00 00 00 ; ..(.....
```

자, 앞에 16byte를 빼고서 (e_ident를 건너 띄우고), 2 byte를 읽어 보면 01 00 입니다. 이것을 Little Endian으로 해석해 보면, 00 01 이네요. 결국 1이라는 뜻이죠. 오오, 의외로 간단하네요. 내친 김에 하나 더? 라고 생각해 봤지만, 그다지 쓸모 있는 과정은 아닌 것 같습니다. 정말 그런가만 확인 했고요, 이제 와서 이런걸 자동으로 해주는 tool이 있다고 말하면 너무 무책임한 건가 - 뭐 어쩔 수 없잖아 - 라고 생각합니다. 죄송.
GCC bin utility를 보면, readelf라는 utility가 있는데, 이 녀석을 이용하면, 상당히 편하게 ELF file의 내용물을 들여다 볼 수 있습니다. (위와 같은 수고로움은 readelf가 없을 때, 해야 합니다.) 한번 들여다 봅시다. readelf는 다음과 같이 사용합니다.

```
readelf -h spaghetti.o      (-h 는 header 이라는 option이며, readelf에는 그 외에도 여러가지 option들이 있습니다. )
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
```

```
Type:                              REL (Relocatable file)
Machine:                           ARM
Version:                           0x1
Entry point address:               0x0
Start of program headers:          0 (bytes into file)
Start of section headers:          860 (bytes into file)
Flags:                             0x2000000, Version2 EABI
Size of this header:               52 (bytes)
Size of program headers:           32 (bytes)
Number of program headers:         0
Size of section headers:           40 (bytes)
Number of section headers:         11
Section header string table index: 1
```

헙, ELF header에는 위와 같은 내용들이 들어 있네요. 보시면, Type은 Relocatable이고, Machine은 ARM 이고, 뭐 이러니 저러니 떠들어 놓았습니다. 멍하니 두 눈의 근육을 풀어놓고 보고 있자니, 정말 편한 세상입니다.

자, 이제 link를 해 보기 전에, 이 object file이 link를 하기 위해서 어떤 것들을 해 두었는지를 "연구"해 볼 까요. 우리가 살펴 보았던 ELF header의 크기를 환산해 보면 (다 더해보면), 52 byte입니다. (뭐, readelf를 통해서 알아낸 header의 정보 중 size of this header도 52 (bytes)라고 나와 있군요.) 그렇다면, 52 byte 이후에는 "도대체" 무엇이 있을 까요?

뭐 별거 있겠습니까 - 라고 말했지만 막막하기만 하죠 - 하지만, 길은 무조건 있기 마련이니까 걱정 마시고, 아마도 실제 코드 부분이 있을 거라 가정하고, 진짜 코드가 오지 않을까 확인해 보는 것은 어떨까요? spaghetti.c가 compile된 후에 기계어 (op code)으로는 어떻게 되었을까 확인하는 방법으로는, 실제 object code를 disassemble 해 보는 방법이 있습니다. Utility이름 처럼 fromelf - ELF로 부터 - 뭐든 할 수 있는가 봅니다.

```
fromelf -c spaghetti.o
```

결과를 보시면 많은 정보가 나와있는데, 나머지 정보는 천천히 심심할 때 훑어보시고, disassemble된 부분만 살펴 봐야죠.

```
** Section '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 12 bytes (alignment 4)
    main
    $t
    .text
        0x00000000:    2003        .       MOV     r0,#3
        0x00000002:    2104        .!      MOV     r1,#4
        0x00000004:    1840        @.      ADD     r0,r0,r1
        0x00000006:    1810        ..      ADD     r0,r2,r0
        0x00000008:    4770        pG      BX      r14
    $d
        0x0000000a:    0000        ..      DCW     0
```

오오, main이라는 것이 Section #5 text 에 자리잡고 있네요. 오른쪽을 보면, 'Assembly로 만드는 방법 편'에서 보았던 Assembly가 보입니다 (완전 똑같네요 - 별로 놀라운 일은 아닌가 하는 생각이 문득 드네요.)
자 그러면, 정말 코드 영역은 왼쪽에 늘어선 숫자들인 2003 2104 1840 1810 4770 이겠네요. 16 bit Little Endian으로 다시 늘어 써 보면 0320 0421 4018 1018 7040 이고, 실제 이 값이 object file의 52byte이후에 적혀 있는지 확인해 보시도록 하시죠. (자 이 부분이 compiler 즉, machine에 따라 틀린 값이 되겠지요. ARM과 Intel은 서로 다른 Opcode로 동작 할 테니까요.)

```
00000000h: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 ; ELF.............
```

00000010h: 01 00 28 00 01 00 00 00 00 00 00 00 00 00 00 00 ; ..(.....
00000020h: 5C 03 00 00 00 00 00 02 34 00 20 00 00 00 28 00 ; ₩.....4. ...(.
00000030h: 0B 00 01 00 **03 20 04 21 40 18 10 18 70 47** 00 00 ; ..... **.!@...pG**..
00000040h: 03 00 00 00 03 00 00 00 28 00 00 00 FF FF FF FF ; ....(...□□□□

헉 정말 쓰여져 있네요. main() 함수를 처리하기 위한 op code는 보시는 바와 같이 얼마 되지 않습니다만, 그 외 linker를 통하여, 하나의 executable ELF로 만들기 위하여 만들어 두는 다른 section들이 있으니까, spaghetti.o 는 상당히 큰 file 이 됩니다. 다른 section들 중 몇 개만 readelf를 이용해서 확인한다면, spaghetti.o가 가지고 있는 symbol table과 relocation table입니다.

Symbol의 정의는 앞에 나왔으니까 잘 찾아보세요.

먼저, symbol table은 아래와 같이 형성되는데, 이 내용 중 눈 여겨 볼 것은 main()함수와 zi, rw, relocate 전역변수가 어디에 낑겨 있는지를 확인하는 것이 급선무 이지요.

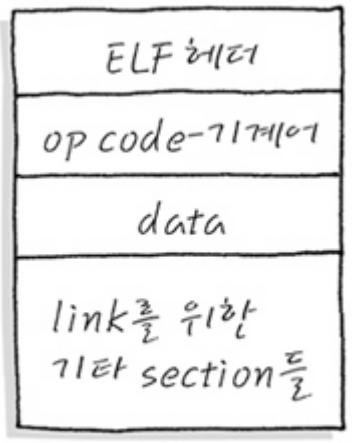
Symbol table '.symtab' contains 20 entries

Num:	Value	Size	Type	Bind	Vis	Ndx	
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	0	FUNC	LOCAL	DEFAULT	5	$t
2:	0000000a	0	FUNC	LOCAL	DEFAULT	5	$d
3:	00000000	0	FILE	LOCAL	DEFAULT	ABS	spaghetti.c
4:	00000000	0	OBJECT	LOCAL	DEFAULT	6	.bss$7
5:	00000000	0	SECTION	LOCAL	DEFAULT	6	.bss
6:	00000000	0	OBJECT	LOCAL	DEFAULT	7	.data$10
7:	00000000	0	SECTION	LOCAL	DEFAULT	7	.data
8:	00000000	0	OBJECT	LOCAL	DEFAULT	8	.data$15
9:	00000000	0	SECTION	LOCAL	DEFAULT	8	.data
10:	00000000	0	SECTION	LOCAL	DEFAULT	5	.text
11:	00000000	0	OBJECT	LOCAL	DEFAULT	9	C$debug_frame1
12:	00000000	0	NOTYPE	LOCAL	DEFAULT	ABS	BuildAttributes$$THUMB_IS
13:	00000000	0	FUNC	WEAK	DEFAULT	UND	Lib$$Request$$armlib
14:	00000000	4	OBJECT	GLOBAL	DEFAULT	6	zi
15:	00000000	4	OBJECT	GLOBAL	DEFAULT	7	rw
16:	00000000	4	OBJECT	GLOBAL	DEFAULT	8	relocate
17:	00000000	10	FUNC	GLOBAL	DEFAULT	5	main
18:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	__main
19:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	_main

Symbol table의 내용 중 Name은 Linker를 위한 symbol 이름이고, Value는 각 section에서의 해당 symbol의 시작 offset 주소를, Size는 Symbol의 크기 (Symbol이 function이나 object가 아닌 경우에는 0), Type은 Function, Object, Section 등을 나타내며, Bind는 Symbol의 scope를 한정 (Local, Global, Weak) 합니다.

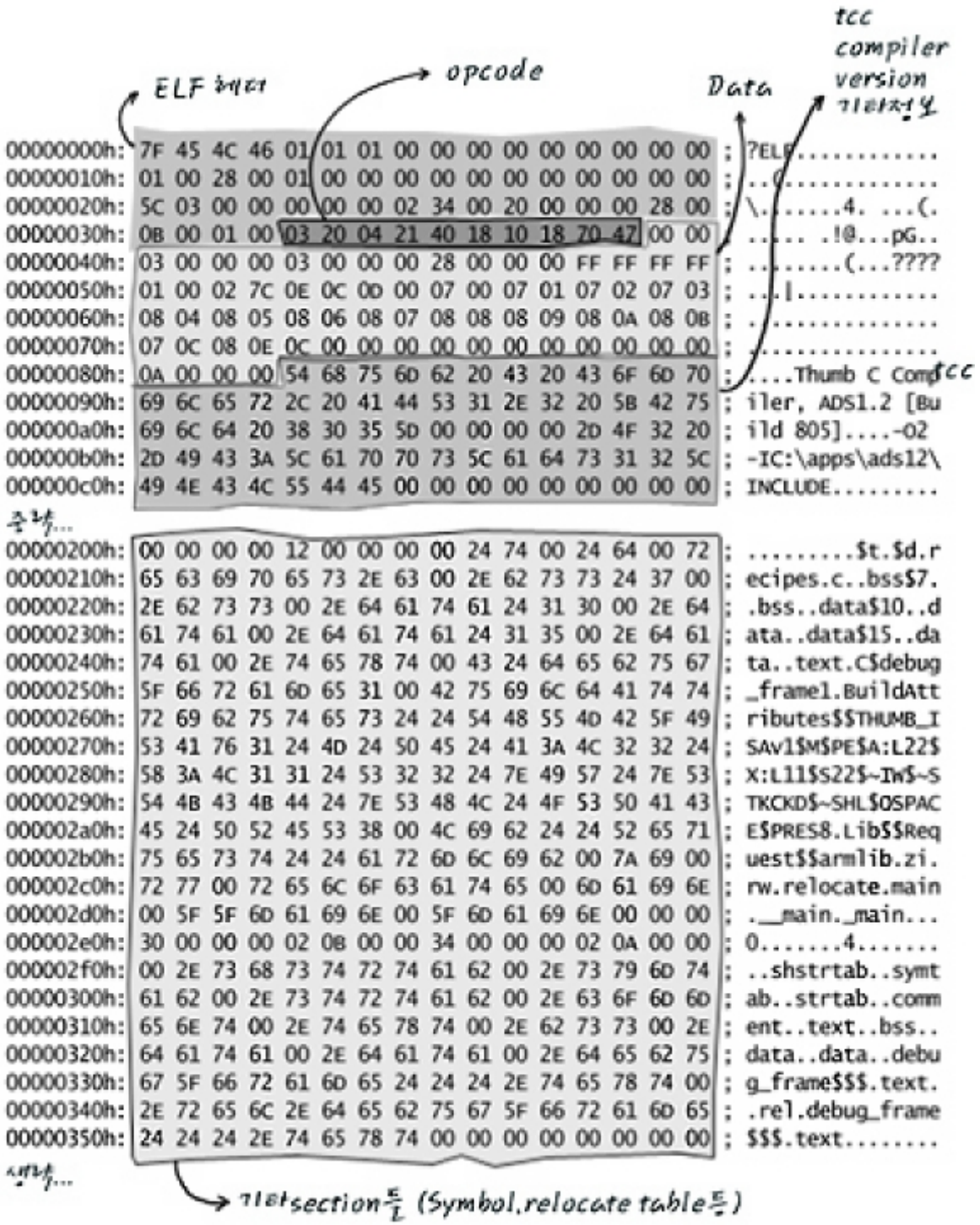
약간의 참고로, Ndx에 보시면, UND는 현재 file에서 사용되고 있지만, 실제 함수의 define이 없는 경우를 의미하며, ABS는 relocate 되어서는 안 되는 것을 의미해요 그리고, Ndx= 1이면,.text section을 의미하고요, Ndx=3이면, .data를 의미하는 거죠. Symbol의 scope를 한정하는 것 중 WEAK의 경우는 LOCAL로만 쓰일 가능성이 큰 symbol을 의미하지요.

자세히 보시면, main()은 FUNC (function)이며, GLOBAL로 처리 되고, zi, rw, relocate 역시, GLOBAL variable로 처리가 되어 있음을 확인할 수 있습니다. 다른 file의 함수 등에서 이 global 변수나, 함수를 만지거나, 호출하게 되는 경우 linker가 바로 이 table을 이용하여, 서로를 엮어주는 역할을 하게 됩니다. compile된 file이 수천 개가 될 경우, 이에 대한 table들을 서로 전부다 엮어야 되니까, linker는 해야 하는 일이 엄청 많겠죠?
이런 식으로 따져 가다 보면, - ELF specification을 전부다 분석하다가 진짜로 무엇을 하려고 했는지, 잊어버릴 수도 있으니까, 실제 분석은 ELF specification을 보시는 것이 도움이 되겠습니다. 덜덜덜 - 실제적인 ELF의 형식을 다음 그림과 같이 표현할 수 있겠습니다.



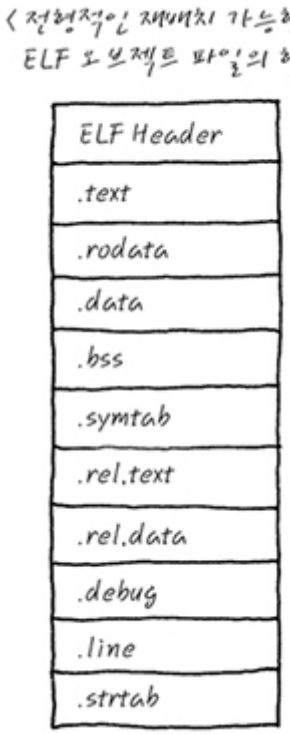
〈 ELF 구조 〉

실제 spaghetti.o의 구조를 위의 ELF 구조에 맞추어서 다시 그려보면, spaghetti.o의 구조가 어떤 것인지, 확연하게 알아보실 수 있으리라 생각하고 있습니다. 진짜 spaghetti처럼 그림이 엉켜 버렸네요.



이런 식으로 object file을 만들어 두면, Linker가 이것들을 보고서 막 연결할 수 있겠다는 자신감이 들기 시작합니다. 이런 식의 자세한 분석에 너무 심각하게 되진 말아주세요. -그리 자세하지도 않지만-

마지막으로 기타 Section들까지 모두 자세히 들여다 본다면



뭐 이런 식이고요

위의 .text는 code, .rodata, .data, .bss는 Data영역에 속하니까 잘 아시겠고 다른 section들의 의미는 뒤에 더 자세히 다루니까 걱정 말고 넘어가 주세요.

 ELF format에 대하여 한가지 더 덧붙이자면, section은 서로 관련이 있는 녀석들끼리 묶이게 마련인데, 이런 관련성이라는 것이 3가지 정도로 나뉘게 되지요. 코드들은 text라는 section, 초기화 된 전역변수는 data라는 section, 그리고, 마지막 초기화 되지 않은 전역 변수들은 bss 라는 section에 같이 묶이게 됩니다. symbol table에도 보면, 오른쪽에 bss, data, text등이 있는 것을 보실 수 있습니다. 그런데, data, text는 무슨 뜻인지 알겠는데, bss는 도대체 어디서 온 말일까? 라는 의문이 제 머리를 사로 잡은 적이 있습니다. bss의 의미는 1950년대 IBM704 assembly에서부터 나온 것으로서 Block Started By Symbol인데 소프트웨어 공학으로 유명한 카네기 멜론대학교의 어떤 사람은 "Better Save Space"라고도 하던데요. 과연 저도 말장난 하나 더 하자면, "Better Save ROM Space" 라고 부르면 혼날까요? 진정한 bss의 의미는 Scatter Loading을 알게 되면 확실하게 감 오실 거에요.
Block Started By Symbol의 의미도 Scatter Loading에서 더 자세히 알게 되실 거에요

 Link에서 ELF 형식에 대해서 더 자세히 볼 것이니 이번 판에 너무 기대 많이 했다면, 다음 판을 기대하세요.

elf, 엘프, arm, software, output, compile, 컴파일, option, format, 기계어, object

Linked at at 2009/10/01 12:32

... ① ELF format Object File의 진실 ... more

Commented by 성용 at 2009/08/04 17:24

아아아~~정말 쉽게 설명해주시는거 같은데요....

제 머리가 돌인 상태로 쭉 지내와서 그런지 이해가 안가네요...

그래도 정말 쉽게 쉽게 써주셔서 감사합니다..

머 몇번 게속 읽어 보면 알겠죠? ㅎㅎ;;;

Commented by 히언 at 2009/08/04 22:40

꺄옥. 어쩌죠. 지 말주변이 없어서요.. T.T
더 친절하면 좋았을텐데.. 그쵸.