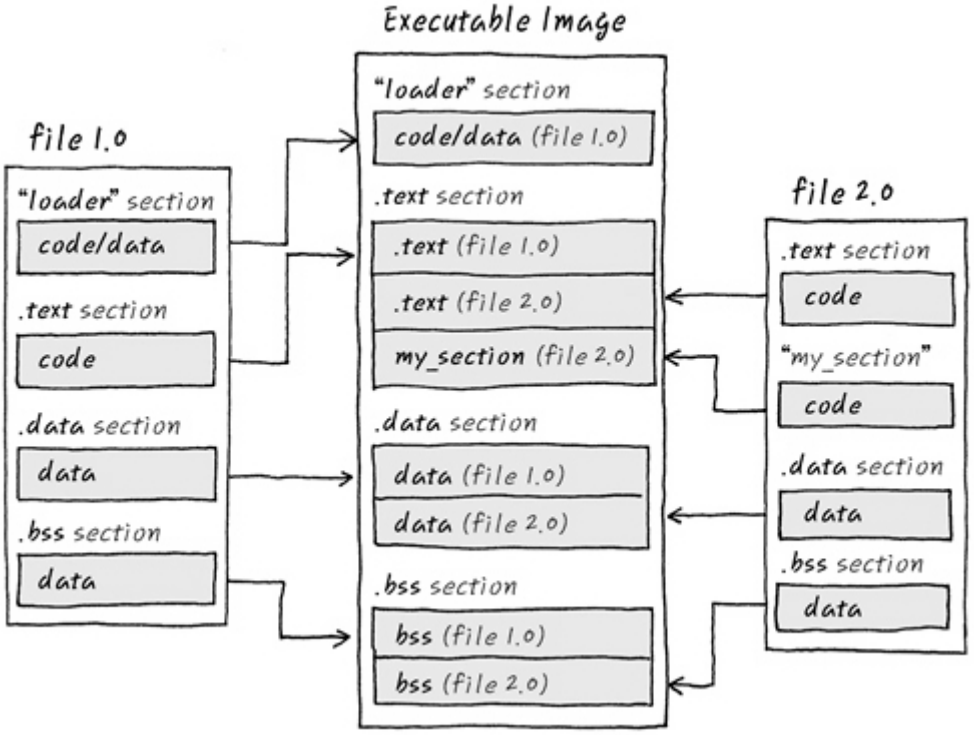
2009년 06월 20일

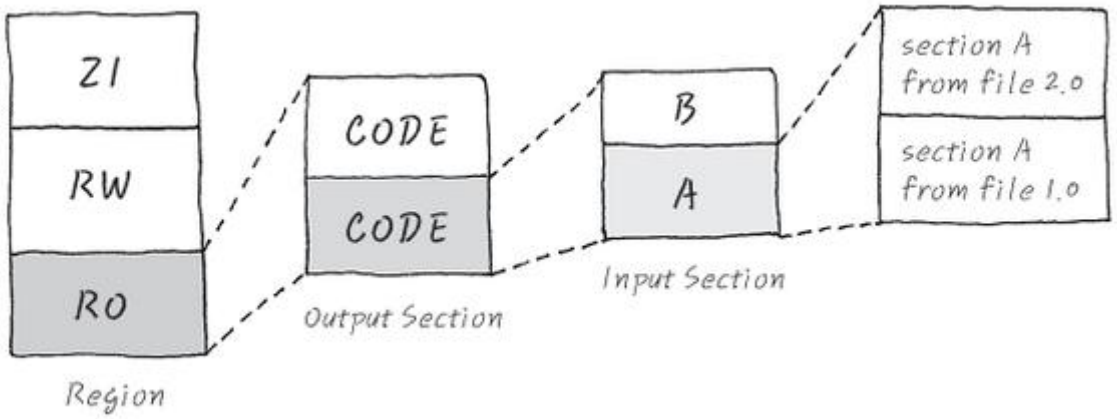
Linker를 마무리 짓자 - ELF와 fromelf 까지!

- Linker란, 결국 Link시에 실제 함수 정의부의 위치와 전역변수들의 위치를 library file과 object file 에서 차례대로 조사한 후에 모두 Table로 간직하고 있다가, 그 주소를 함수호출 코드 부분에 기록해 넣는 것이 Linker가 하는 일 -

아주 거창하게 얘기하자면, Linker를 이용해서 executable ELF format image를 만드는 과정입니다. Linker는 모든 input object file들의 모든 코드와 데이터를 가지는 실행 가능한 새로운 object file을 만들어 냅니다. 그러기 위해서 각각의 object file들이 가지는 text, bss, data를 모두 새로운 text, bss, data에 모으고, 이런 일은 누가 잘 그려놓은 그림으로 표현 되겠습니다. [1]



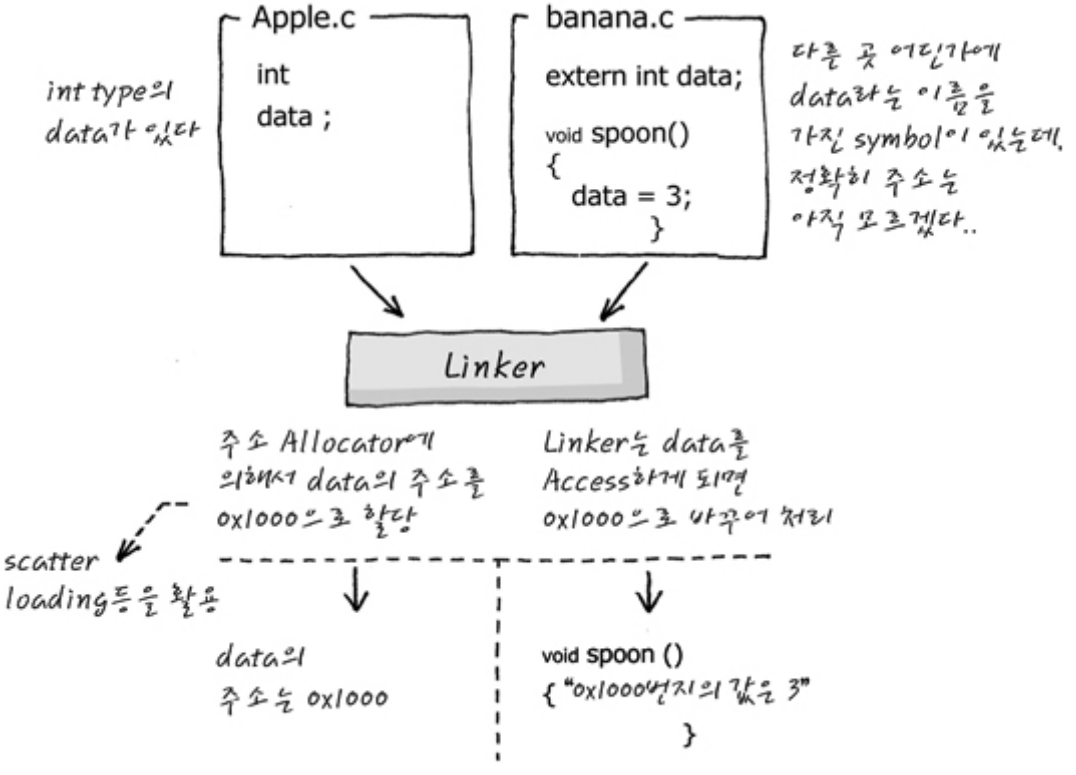
요런걸 Linker Placement Rule이라고 해서, section끼리 모으는 역할을 하죠. output section안에는 많은 input section들이 있을 텐데, 요놈들을 알파벳 순서대로 다시 정렬해서 하나의 output section을 만들죠.



나중에 나오겠지만, 맨 왼쪽이 Region 두 번째가 Output section, 세 번째가 Input Section으로 불리는 거죠. Input Section에는 각 object file의 Symbol들을 알파벳 순서대로 모아서 Input Section들을 만듭니다.

이렇게 같은 속성끼리 묶어 놓는 거로 일이 끝난다면 얼마나 linker가 허무해 하겠사옵니까! 이렇게 모으는 동안 어떤 한 file 내에 선언만 되어 있고 사용되어 지지 않던 변수라던가, 어떤 함수를 불러야 되는데, 막상 그 .c file내에는 없고, 다른 .c file에 있다던가 하는 함수들을 Linker가 하나 하나 찾아 이들을 서로 연결합니다. - 과연 Linker는 부지런한 개미와 같은 일꾼입니다. -

Symbol은 절대 Address를 가질 수 있는 최소의 단위라고 했습니다. (함수, 전역변수) 이런 과정을 유식한 말로, Symbol reference resolving이라고 부르는데 다시 말하면, 여러 개의 object file들은 서로 구멍을 갖고 있다고 보면 쉽습니다. 예를 들어, 어떤 김 아무개 c file내에 handle 이라는 int type의 전역변수가 있다고 하고, 다른 장 아무개 c file에서도 이 handle이라는 전역변수의 값을 최 아무개 함수에서 물컹물컹 만지려고 할 때, 각각의 c file에 대하여 compiler가 object file을 만들어 낼 때는 김 아무개 object file에는 handle을 symbol화 해서 link 시에 이 handle이 위치하는 절대 주소를 가질 수 있게 하며, 장 아무개 object file에는 어디 있는지 모르겠지만 handle이라는 symbol이름만 넣어놓고 구멍을 내 놓게 되는 것이죠. 전체를 아우른 대왕 executable object를 만드는 이때! linker는 이런 구멍을 눈치채고, 장 아무개 object file에 compiler가 구멍을 내 놓은 자리에 절대 번지를 써 넣으면, 비로서 장 아무개 object file 내에 최 아무개 함수는 handle이라는 전역변수를 물컹물컹 만질 수 있게 되는 원리 입니다. 헥헥.

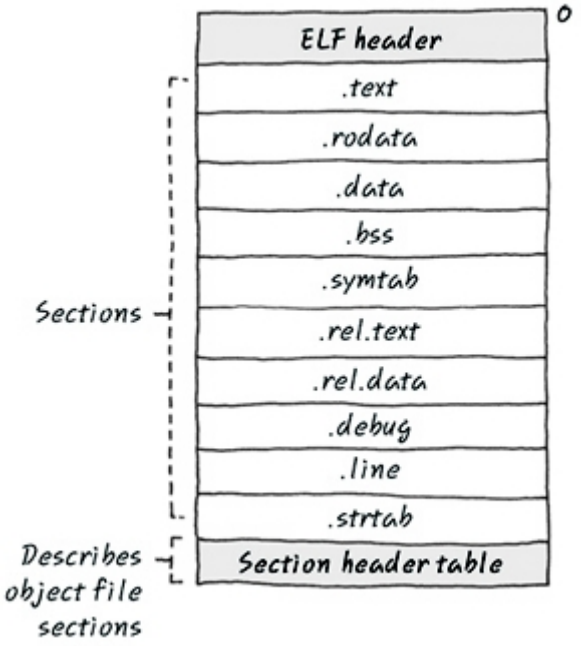


편의상 object file로 만드는 compile과정은 빼고 , c file에 있는 것들이 실제 Linker에서는 어떻게 연결 되는지 만 나타냈습니다. 이해들 해주실 꺼라 믿고, 계속 레츠고. 이런 식이라면, 함수도 마찬가지겠죠? - 함수도 symbol이니까 - 함수의 경우라면, 어느 주소를 실행해야겠는데,
그 주소를 모르니까, 구멍을 내 놓고, link를 할 때, 그 구멍에다가 그 함수의 절대 address를 끼워 넣으면 되겠습니다. 대신에 어딘가 다른 곳에 있을 symbol을 사용하는 c file에서는 relocatable object file을 만들 때, 어딘가 다른 곳에 있을 테니, link 할 때, 꼭 찾아서 link 해 달라고 표시를 해두어야 합니다. 그렇지 않으면, relocatable object file을 만들 때, 선언이 없다면서 compiler는 error를 내고, compile을 멈추게 됩니다. 바로 그 표시가 "extern" 입니다. 그러니까, 이런 경우에는 장 아무개.c 의 handle을 사용하기 전에, 전역변수로 extern int handle; 이라고 선언을 해주어야, compiler가 알아서 나중에 구멍을 메꾸어 줍니다.

결국 Link시에 실제 함수 정의부의 위치와 전역변수들의 위치를 library file과 object file 에서 차례대로 조사한 후에 모두 Table로 간직하고 있다가, 그 주소를 함수호출 코드 부분에 기록해 넣는 것이 Linker가 하는 일이라고 할 수 있겠습니다. 이렇게 많은 일을 하니까, Link시에 RAM도 많이 잡아먹고, 시간도 많이 걸리죠. Link를 효율적으로 할 수 있는 방법을 연구해 보세요. 무궁무진하다니까요.

ELF format relocatable object file의 구조를 자세히 들여다볼 시기가 도래 했습니다. "ELF format object file의 진실"편에서 KTX를 타고 가면서 차창으로 바라보는 풍경처럼 전부다 뭉뚱그려서 ELF format을 슬쩍 맛만 봤습니다. 이젠 무궁화호를 타고 갈 차례라고나 할까요.

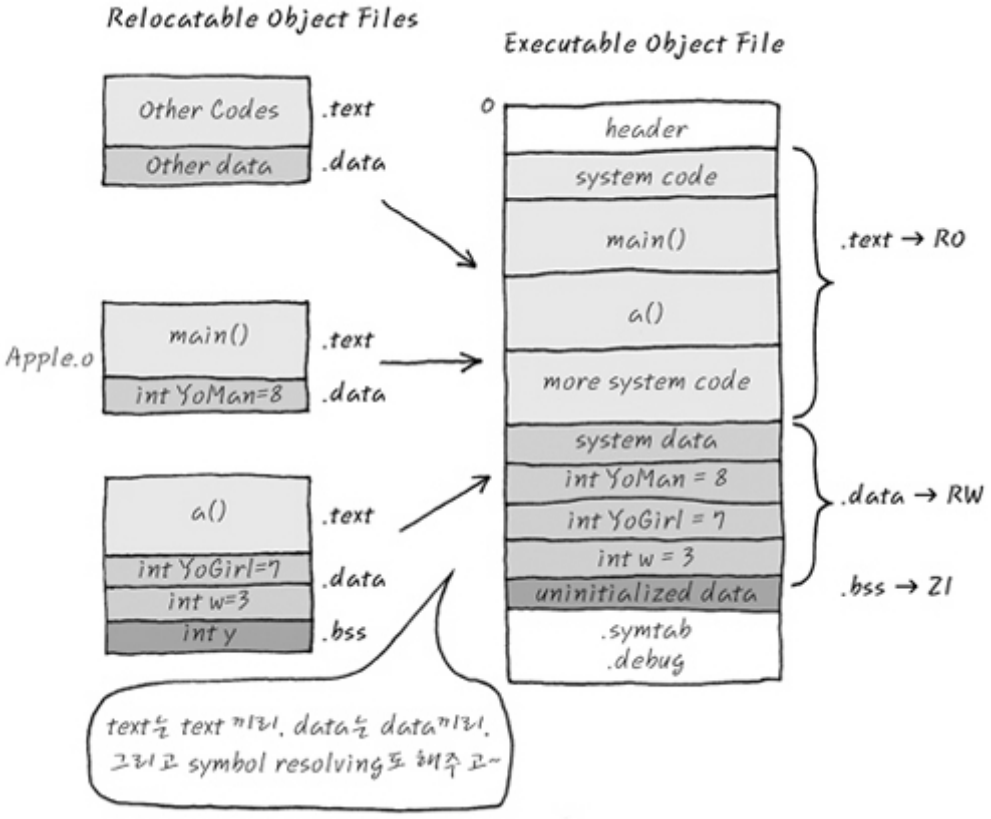
전형적인 ELF relocatable object file의 구조를 자세히 다시 한번 그려보겠습니다.



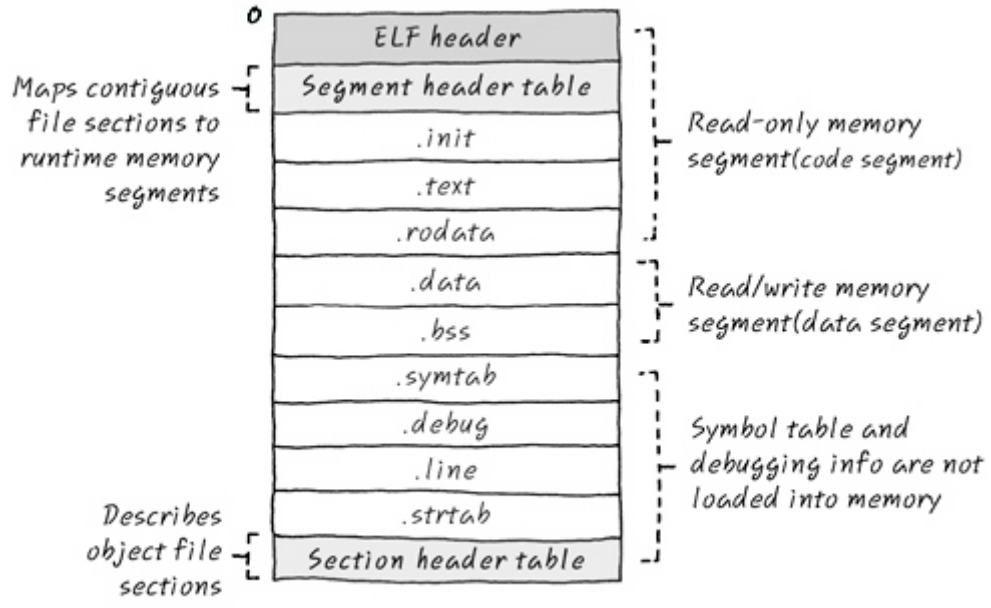
뭉그러져있던 section들이 이제 자세히 보이시는지 모르겠습니다. Linker가 이 object file을 parsing을 하고, 해석을 할 수 있게 하기 위한 내용들이 section들에 들어가 있습니다. 구조는 맨 앞에 ELF header가 있고, 맨 밑에 section header table이 꼭 있습니다. 지붕과 바닥처럼 자리잡고, 그 사이에 section들이 오게 되는데, 각 section에 대하여 간단히 소개하겠습니다. 친절하지 않은 설명입니다.
.rel.text, .text에 들어있는 각 머신 코드의 위치를 나타내고요,
이것들은 나중에 linker가 이 오브젝트 파일을 다른 오브젝트 파일들과 연결시킬 때 필요해요.
.text : 일전에 말했듯이, compile된 기계어 (op code)가 들어 있습니다.
.rodata : read-only data를 의미하며, const로 선언된 바뀌지 않는 data들이 들어 있습니다. 또한 참고로, switch case문에 의한 jump table도 들어 있습니다.
.data : 초기화된 전역변수들이 자리잡고 있습니다.
.bss : 일전에 소개한 대로 초기화 되지 않아, 0으로 초기화 되는 전역변수들이 들어 있습니다. 이런 전역변수들은 실제로 이 section에 자신의 크기만큼 잡히지는 않습니다. - 0으로 초기화 할껀데, 굳이 넣어둘 필요 없겠죠 -
.symtab : symbol table이며 , symbol이란 실제 주소를 가질 수 있는 단위를 말합니다. 보통은 전역변수이름과 함수이름이며, 어떤 사람들은 compile option에 -g option을 꼭 써야, symbol 정보가 생성되는 것으로 알고 있는데, 쓰지 않더라도, 이 section은 꼭 생성됩니다.

.rel.text : relocatable text이며, 말 그대로 op code가 들어 있습니다만, symbol reference resolving에서 언급한 구멍 난 text가 들어 가게 됩니다. executable object에는 없는 section입니다. 결국엔 이것들은 나중에 linker가 이 오브젝트 파일을 다른 오브젝트 파일들과 연결시킬 때 필요해요.
.rel.data : rel.text와 마찬가지로, 구멍 난 전역변수들이 들어 갑니다. 즉, 현재의 파일에서는 정의되어 않고, link시에 참조되는 전역 변수에 대한 재배치 정보를 담고 있고요, extern 전역변수나, extern 함수의 이름들이 들어 있어요.
.debug : 이 section이야 말로 -g option에 의한 debug symbol table입니다. 지역, 전역 변수들에 대한 디버깅 심볼들이 있고요. 컴파일러가 -g 옵션과 함께 수행될 때 생성되죠. 보통 DWARF형식의 디버깅 심볼들이 들어 있어요.
.line : -g option으로 compile했을 때, text section의 opcode와 원본 C의 line을 연결하여, code를 보면서 debugging 가능하게 해줍니다. 만일 이 정보가 잘못된다면, trace32 (Debugger) 등에서 Tracing할 때, symbol을 찾아도, code를 볼 수가 없습니다. 임시방편으로 y.sourcepath + 명령어가 있긴 하지만요.
.strtab : .symtab와 .debug section에 사용되는 const data인 string등을 가지고 있습니다. 그리고, section header의 section 이름들도 들어 있다죠. 헉헉.

linker는 object file을 ram에다가 차곡차곡 쌓아 두면서, 이런 정보들을 가지고, executable elf를 만들어 냅니다. 구멍 난 곳에는 구멍을 메워주고 - 외부 함수로 branch했었어야 되는데 정보가 없어서 그냥 구멍으로 놔주었던 곳에는 진짜 주소를 메워 넣고, 외부 전역변수를 사용했었어야 되는 곳에는 전역변수 주소를 끼워 넣습니다. 맨 먼저 나왔던 그림을 조금 더 아름답게 꾸며 보자면.



사실은 중간에 Linker가 괴물의 모양을 하고서 문어같이 여러 개의 발을 이용해서 같은 종류끼리 모아서 여기저기 깨워 넣는 그림을 상상했는데, 그렇게 그림을 그려 넣는다면, 너무나 끔찍해져 버려서, 참기로 했습니다. - 오탈자 수정 要 system code -> other codes, a(), main()도 바꾸자. - 그림에 표시되어 있는 RO, RW, ZI는 일단 무시하고 레츠고.
이런 의미에서 최종 Executable Object File의 구조를 본다면 약간 다른 모양의 ELF가 탄생합니다. 새로운 용어 중 Segment라는 게 나오는데 Segment = ∑ Sections 라고 보시면 됩니다. 같은 속성의 Section들을 모아서 한 segment안에 다 넣을 거니까 Segment는 같은 속성의 Section들의 모임이라고 보시면 되겠죠!



Typical ELF executable object file

ELF header는 이번에는 이 file이 실행될 때, 시작되어야 하는 entry point address가 있습니다. rel.* section은 모두 없어졌고요, .init section은 두 가지 정도 용도가 있는데, 하나는 OS가 있는 시스템에서 ELF가 실행될 때 실행되기 전에 initializatoin을 하는 작은 code가 들어 있습니다. 또는 Program Header라는 것이 들어가는데, executable file이니까, program header라는 걸 만들어서, 실행하는데 필요한 몇 가지 정보를 넣어 둡니다. (Linker가 친절하게도 만들어 주네요, machine에 따라서.) 결국 Code segment, Data segment 두 개로 나뉘게 됩니다.

한가지, 재미있는 질문을 하나 해볼 게요.
만일, Global symbol들 중 똑같은 이름으로 여러 군데에 선언되어 있다면, 과연 Linker는 어떤 식으로 relocate를 할까요? 보통 우리의 상식으로는 Compile 마지막 단계의 Linking과정에서 duplicate error를 내면서, Link 과정이 안될 거라고 생각하시고 있겠지만, 사실은 다음과 같은 rule로 Linker는 Link를 하게 되므로, 어떨 때는 error를 내지 않고, warning만 내고 executable object file을 만들어 낼 때도 있습니다. 이 과정 때문에, 어떨 때는 생각지도 못한 문제를 야기 시킬 때도 있습니다. 저도 한번은 Linker한테 배신을 당해서 이틀 정도를 고생한 적이 있습니다. 이걸 알아냈을 때, 저 나름대로의 철학이 무너지는 아픔을 겪기도 했지만요.

rule은 간단합니다. Compiler에게는 두 가지 종류의 Global symbol이 있습니다. Strong 또는 Weak인데요, 함수와 초기화가 된 전역변수는 Strong으로 분류하고, 초기화가 되지 않은 전역변수는 Weak으로 분류해 놓습니다. Rule은 간단하게도 다음과 같습니다요.

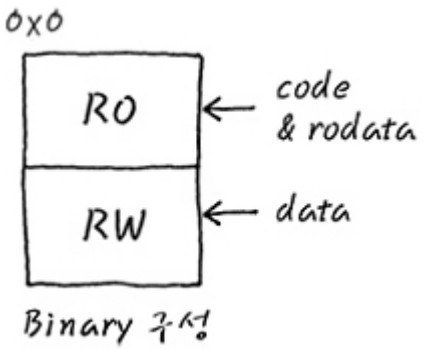
1) 여러 개의 Strong Symbol은 말이 되지 않습니다. → 결국 Link error를 유발 합니다. 같은 이름의 함수정의가 여러 번 되어 있다든지,
   같은 이름의 전역변수가 초기화 되어 선언되어 있다든지, 하는 문제입니다.
2) 하나의 Strong Symbol과 여러 개의 Weak Symbol이 있다면, Strong Symbol을 선택합니다.
3) 여러 개의 Weak Symbol이 있다면, 아무거나 하나 골라서 선택을 합니다.

2번이나, 3번이 벌어졌을 때, - 사실 거의 문제가 되어서는 안되지만 - 곤란한 일이 발생할 때도 있습니다. 예를 들어, Weak symbol만 있는 경우에 (전역변수인 경우겠죠) 한 개를 골라서 compiler가 통합해 버렸다고 했을 때, 전역변수니까, 어떤 함수가 그 변수를 만지게 될지는 아무도 모르게 되는 것이죠. 예를 들어, 1.c와 2.c에 bool weak 라는 함수가 각각 선언되어 있을 때, 1.c는 나름대로 1.c에 맞는 변수 수정을 할 거고, 2.c는 2.c에 맞는 변수수정을 하게 될 것인데, 1.c에서 weak = TRUE라고 set 한 후, 2.c에서 1.c도 모르게 weak = FALSE 라는 statement를 만났을 때, weak = FALSE가 되어버려, 1.c에서는 weak의 값이 엉망친창이 되어 버리는 것입니다. (OS 측면에서 바라보았을 때, 1.c와 2.c가 서로 다른 task에 속해 있다고 가정했을 때 입니다.) 어이 없지요? - 여기서 얻을 수 있는 교훈은 Flag의 남발은 별로 좋지 않다~ 입니다만 -

뭐 가능한 한 이런 식의 flag를 꼭 써야 하는 경우가 있다면, static으로 선언해서 다른 file에서 참조하지 못하도록 원천적으로 막아버리는 테크닉을 쓸 때도 있답니다.

여기까지 하게 되면, 바로 Linux 같은 OS위에서 실행되는 executable object file이 만들어 지게 되는 것입니다만, 우리는 바로

Embedded system을 하니까, 실제로는 ELF header나, symbol table, debug 정보 등의 실제 code 실행에 필요 없는 것들은 빼고, Code segment중 text, rodata를 RO (READ only)라고 칭하고, data를 RW (Read-Write) 라는 두 개 section으로 나눈 binary 형태로 만들어야 비로소 embedded system에서 실행될 수 있는 binary라는 것이 탄생하게 됩니다. 이 binary야 말로, Embedded system의 Flash memory에 burning하는 형태 입니다. (bss와 ZI는 (zero initialized)라는 형태로 만들어 지는데, 이는 RAM을 차지하기 때문에, 굳이, binary에 포함되지 않고, boot up 등에서 따로 그 부분을 처리해 줘야 합니다.) - "Boot up sequence" 편에서 다시 한번 다루기로 하고, 일단 패쓰입니다.



Binary 구성

bss에 관련한 이야기를 하자면, data와 bss에 대한 차이점을 짚고 넘어가야 하는데요, data는 초기화가 되어 있으니까, Flash등의 ROM에 그 초기 값을 가지고 있을 필요가 있습니다. 하지만, bss는 uninitialize되어 있으니까, 모두 0으로 일단 초기화를 하게 됩니다. 그러니까, 굳이 ROM에 그 초기값을 가지고 있을 필요는 없고, 그 시작 주소와 size만 알면, 그 시작주소에서 크기만큼만 RAM에 확보해주면 되는 거죠. 이때 boot sequence시 bss영역을 0으로 초기화 해주니까, 결국엔 uninitialized된 전역변수들은 0으로 initialize된다고 보시면 무방합니다.

예제 하나도 안 해보고 넘어가면 서운하니까, Linker를 이용하여, executable elf를 만들어보고, 그 elf를 binary로 만드는 걸 해보고 마무리 아쉽지만 - 뭐 나름대로 잘 견딘 거나 다름없는 거 아닙니까 - compile의 원리에 대해서는 여기까지 하는 걸로 하는 것이 서로에게 좋겠지요.

한가지 기억해 두어야 할 Rule이 있습니다. Memory Map을 그릴 때는 저 같은 경우에는 0x0를 맨 위에다 그리고, Highest 번지를 아래쪽에 그리는 경향이 있습니다. 이 rule을 잘 기억하고 있으면, stack이나, 다른 memory의 위치를 이해하는데 조금 더 편리할 것이라 생각됩니다. - 제가 굳이 이렇게 하는 이유는 Debugger등을 사용 할 때, 낮은 주소가 위쪽에 표시되니까, 이런 식으로 그림을 이해하고 있다면, Debugger에서 보여주는 Memory 상태 등을
분석할 때, 조금 더 편하게 다가갈 수 있습니다. 어떤 책을 보면 높은 주소를 위에 적어놓고, 낮은 주소를 아래에 적어놔서 점점 더 헷갈려 진다니까요! -

예제는 arm.c와 thumb.c를 compile하고, Library section에서 만들어 두었던 recipes.lib를 link하여, embedded.elf를 만들고, 이 embedded.elf를 embedded.bin으로 만들어 봅시다.
(c compiler는 tcc, linker는 armlink를 이용해 보세요, output file 이름을 정해주고 싶으면 -o option을 주고 그 뒤에 내가 원하는 이름을 써 주면 그대로 output이 생성됩니다.)

```
tcc -c arm.c thumb.c
armlink -elf -o embedded.elf recipes.lib arm.o thumb.o
fromelf -bin -o embedded.bin embedded.elf
```

오, 이제 스스로 감탄할 때 입니다.

그러면, 이제 혹시 이거 할 수 있겠죠?arm.c thumb.c를 armcc로 컴파일 해서, armthumb.lib으로 만든 후 boot.s를 assembler로 컴파일하고 나서,
armthumb.lib과 boot.o를 link하여 recipe라는 이름의 elf와 bin을 만들 수 있겠죠? 간단간단합니다.

```
armcc -c arm.c thumb.c
```

armar -r armthumb.lib arm.o thumb.o
armasm boot.s
armlink -elf -o recipe.elf armthumb.lib boot.o
fromelf -bin -o recipe.bin recipe.elf

꺄옷!


갑자기, Trace32 얘기를 꺼내서 죄송합니다만, 아마 아시는 분한테는 상당히 유용한 정보가 되리라 생각 중입니다. 물론, 조금만 시간이 지난 후, 누구라도 다시 읽어 보시면, 아하! 하실 것이 분명합니다.


DWARF라는 말을 들어본 적이 있다면, 다음과 같이 그 concept을 이해 하시면 좋습니다. Executable ELF중 debug section이 있는데, 이 영역에 어떤 형식으로 debug정보를 넣을 것이냐 하는 규칙이 있습니다. 이 규칙 중 ARM의 경우 사용되는 방식이 DWARF 방식이며, 이 방식을 통해서, ICD (In circuit debugger, 예 - Trace32)으로 Target system을 Debugging할 때 사용합니다.


.axf 라는 file 확장자도 보신 적이 있을 것입니다. axf는 elf의 한가지 변종인데, arm executable format 이며, DWARF2.0 규격을 따르는 format입니다. 그러니까, elf header나, 나머지 .text, .data, .bss는 같은 규격이지만, .debug의 정보가 DWARF2.0을 따른다고 보면 되겠죠? 엘프 나라에는 드워프도 있다니까요.

[1] Quing Li "real time concepts for embedded systems

elf, linker, fromelf, 마무리, Memorymap, compile, arm, 만세

Linked at at 2009/06/20 20:10

... p; ① ELF format Object File의 진실 ⓜ Linker를 마무리 짓자 - ELF와 fromelf까지! ⓝ Scatter Loading - Linker Description Script   ... more

Commented by 히언 at 2009/08/09 21:09

어디가 달라졌을까요~?