2009년 06월 21일

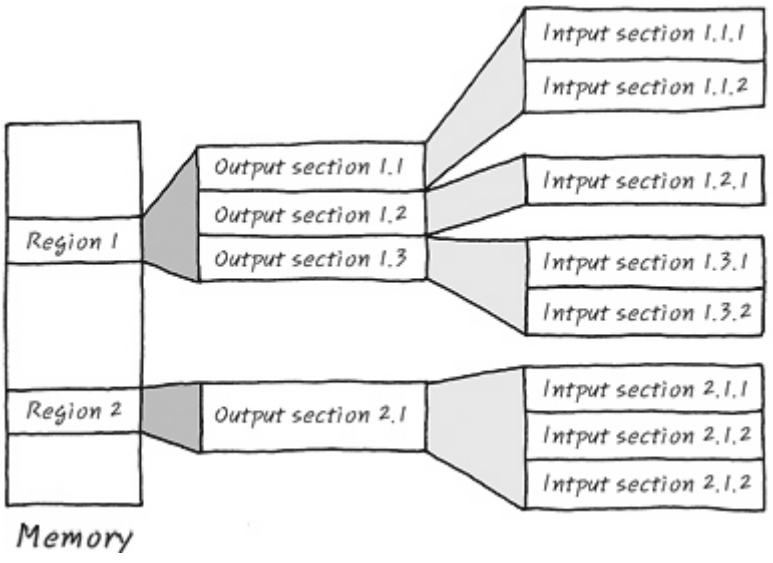
Scatter Loading/ Linker Description Script와 메모리 다루기.

- Linker는 Execution View를 참조하고, Bootloader는 LoadView를 참조해서 Execution View로 만들어주고! -

Symbol과 Linker. 뭐 이런 걸 알게 되었으니, 이제 Linker Description Script를 볼 차례~이런 엔지니어링 세계에서 지금 이 블로그를 읽고 있을 당신에게 찬사를 보내며 XIP 얘기를 꺼내보려고 해요. 앞쪽에 "Memory의 선정과 XIP"라는 Section에서 다음과 같이 XIP를 논한 적이 있어요. " XIP는 Execution In Place라는 말로서, 직접 Software를 실행할 수 있다는 의미인 거죠. 혹자는 Software를 메모리에 Load하여 실행하는 대신 플래시에서 직접 수행하는 기술을 XIP(eXecution-In-Place) 기술이라고 하여, Flash 소자에 그 용례를 국한하기도 합니다만, 저는 모든 메모리 소자들에 XIP라는 용어를 적용하고 싶네요.

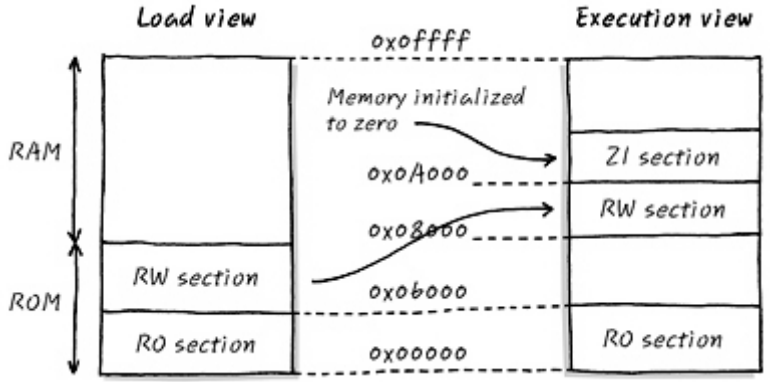
결국에 저는 Word단위의 Access가 가능하여, Software를 Execution(실행)할 수 있는 것을 XIP라 정의하겠습니다. " 이런 얘기를 꺼내는 데는 다 이유가 있겠죠. Execution이 가능하다는 얘기는 Code 영역이 들어갈 수 있는 또는 자리 잡을 수 있는 Memory의 type은 XIP가 가능해야 한다는 말을 하려고 그러는 거죠 머. 만일 XIP가 가능하지 않다면 Code영역은 그곳에 자리잡을 수 없습니다요. 그러면, Linker를 통하여 최종 실행 image를 만들 때는 XIP가 가능한 Memory에는 code를 그리고 Data는 Read Write가 가능한 영역에 mapping시켜서 배치할 수 있어야 하겠죠.

이걸 가능하게 하는게 Linker Descript Script (Scatter Loading)이고요, 컴파일 공장이야기 그림 잘 보시면, Linker에 input으로 *.scl이라는 게 보이죠. 이놈이 바로 ADS에서 scatter loading file이에요. Linker Description Script는 GNU 진영에서 사용하는 용어이고요, ADS쪽에서는 scatter loading file이라고 불러요. 혼용해서 쓰더라도 헷갈리지 말고 용서하소서. 뭐든 뭘 시작하려면 용어를 알아야 해먹으니까, 이것저것 Scatter Loading에 관련한 개념적인 그림 하나 펼쳐 놓고 용어를 좀 알아보시죠.



Memory상에는 Region들이 실제로 올라가게 되는데 Region은 Output section들이 모여서 되고, 이런 Output Section은 Input Section들이 모여서 이루어지게 됩니다. 그럼 이것들의 정체를 알아야 하겠죠. Input Sections Input Section은 이미 앞에서 말한 RO, RW, ZI 중 하나의 속성을 갖는 집합으로서, 하나의 object에도 이런 구역들이 나뉘어져 있겠죠. 다시 한번 comment하자면, RO는 Read Only 니까 const data나, code. RW는 Read Write니까 초기값을 갖는 전역 Data ZI는 Zero initialized니까 초기값을 0으로 갖는 전역 Data 이지요. 이건 Linker한테만 중요하고요. Output으로는 안 나오니까, 어디 가서 찾아보기도 힘들어요. 굳이 찾아보고 싶으면 map file에서 찾아보시면 되어요. Output Sections많은 Input Section들 중에 같은 속성을 갖는 것들을 한 뭉탱이씩 묶어 놓을 수 있는데 이것을 Output Section이라고 부릅니다. 그러니까, ZI 뭉탱이, RW 뭉탱이, RO 뭉탱이 정도 되겠네요. 실제로 Scatter Loading file에 묘사되지는 않고요, Linker가 임의로 만들어내는 뭉탱이라고 보시면 되요. Region Output Section을 한데 묶은 것을 말하는데 실제

메모리에 올라가는 단위가 됩니다. Output section을 묶을 때는 그 속성에 따라 알아서 sorting이 되는데, 만일 ZI, RW, RO 뭉탱이가 하나씩 만나서 하나의 Region을 이루게 되면 RO, RW, ZI순으로 알아서 sorting이 됩니다.
결국에 Scatter Loading에 표시해 줘야 하는 요소는 Input Section과, Region만 표기해 주면 Linker가 알아서 Input Section을 잘 다듬어서 Output section을 만든 후, Region에 맞게 output을 생성해 준답니다. 아.. 이렇게만 말해서는 잘 감이 안 오죠. 그러면 실전 예를 들어가면서 따라가는 게 제일 편하니까, 한번 가봅시다. 이런 Region이나 Section 뭐 이런 거 될 대로 되라는 식으로 냅두고, 한가지 알아두셔야 할 것이 있어요. 뭐냐면, Memory를 내가 원하는 대로 구성할 때 두 가지 관점에서 봐줘야 한다는 사실이죠.
그것이 Load View와 Execution View라는 건데, 뭐 이것도 역시 그림을 펼쳐놓고 얘기하면 편하겠죠. 그림은 너무나 유명한 ADS1.2 Developer Suit에 Linker and Utility Guide의 그림을 널 부러 놓고 갈게요. (실은 위의 그림도 거기 있는 그림이야요)



보시다 시피 왼쪽과 오른쪽으로 나뉜 것이 Load View와 Execution View라는 건데, Load View는 Software가 실행되기 전에 저장매체에 담겨 있을 때의 모습이고요, Execution View는 Software가 막상 실행될 때의 모습이에요. 쉽게 얘기하면 Load View의 경우에는 Flash에 실행 image가 담겨 있을 때의 형태라고 보시면 되요. 쉽게 얘기하면 Loading View는 ROM에 적재되었을 때의 모냥새, Execution View는 실제 Image가 실행될 때의 모냥새를 의미하죠. 보통은 두 개의 View가 같을 거라 생각들 하실 테지만, 막상 그냥 저장되어 있는 상태의 image와 실행될 때의 image가 다른 이유는 몇 가지가 있지요. 첫 번째로는 Load View에서의 RW가 자리잡고 있지만, 이건 실제로는 Readable하기만 한 게 아니라 Writable해야겠죠. 그러니까, 원래 Flash에 자리잡고 있던 RW는 실행 상태에서는 RAM으로 복사되어야 해요. 그러니까 당연히 모양새가 틀리겠죠. 마지막으로 RO의 경우에는 code를 대표하여 XIP를 고려해서 설명해 볼게요. XIP를 고려하게 되면, NOR Flash의 경우에는 Load View와 Execution View가 같을 수 있어요. 왜냐하면 NOR Flash는 Flash에 Loading된 그 상태 그대로 Execution이 가능하죠. 하지만, NAND의 경우에는 얘기가 틀려져요. NAND는 Loading된 상태 그대로 에서 Execution이 불가능하고 XIP가 가능한 SDRAM 같은 메모리로 옮겨져야 하지요. 그러니까 Load View와 Execution View는 달라야겠죠! 이 부분은 Bootloader부분에서 더 자세히 다뤄질 거에요. 두 번째로는 Load View에는 ZI가 따로 잡혀 있지 않아요. 왜냐하면 모두 0이니까 굳이 Load view에 ZI까지 만들어 놓을 이유가 없죠. 하지만 Execution View에서는 당연히 ZI가 자리를 차지하고 앉아 있어야겠죠. 정말 쓰는 것이니까요. 약간의 설명을 토대로 위의 그림을 다시 한번 해석해 보아요. 지금 말하려고 하는 system을 NOR + PSRAM MCP system이라고 치죠. 그러면 NOR는 ROM, PSRAM을 RAM으로 대치해서 생각하시면 돼요. 왼쪽은 Loading View로서, ROM에만 Loading View가 있죠. 뭐, 당연하죠! 하지만 ROM 영역에는 RO, RW만 자리 잡아요. RW는 Data이긴 하지만, 초기값을 갖고 있으니까, 당연히 그 초기값을 저장하고 있어야겠죠. 자, 이제 Execution View로 넘어가서! 실제 Code 영역인 RO 영역은 NOR에서 XIP가 가능하니까, NOR에 그대로 있어도 되고요, 문제는 Data들인데, RW는 초기값을 가지고 있는 녀석들이며 ROM에 있으니까 Read Write를 모두 할 수 있게 하려면 RAM에 복사해야겠고요. ZI는 어차피 ROM에는 자리 잡지 않았었으니까 그냥 ZI영역 크기 만큼 0으로 채워서 RAM에 자리 확보만 하면 되겠죠! 아아.. 간단해라. 그러면, 이렇게 Memory에 자리잡게 하려면 Scatter Loading함수를 어떻게 작성해서 Linker에게 알려줘야 할까요? 지금부터 그것을 살펴보는 순서가 되겠습니다. Scatter Loading의 예를 하나 들어보죠 머. 위 system을 Scatter Loading으로 옮겨 놓은 거에요.

```
LOAD_ROM 0x0
{
    EXEC_ROM 0x0
    {
        spaghetti.o (+RO)
    }
    EXEC_RAM 0x8000
    {
        spaghetti.o (+RW)
    }
```

```
    EXEC_RAM2 0xA000
    {
        spaghetti.o (+ZI)
    }
}
```

위의 Scatter Loading의 예를 구현해 보았어요. 어때요? 일단 큰 그림을 한번 그려 볼까요?

```
LOAD_ROM 0x0
{

}
```

위의 Scatter Loading의 예를 구현해 보았어요. 어때요? 일단 큰 그림을 한번 그려 볼까요? 요렇게 큰 괄호가 하나 있죠? 이게 Load View입니다. 그러니까 Scatter Loading을 보면 큰 괄호가 하나 있어요. 내부에 있는 괄호들은 다 무시되고, 0x0부터 순서대로 그 내용물들을 쌓아서 만들게 되는데 이때 ZI만큼은 무시되는 거에요. 그러니까 아래처럼 생겼다고 보시면 되는 거죠. 위 그림이랑 비슷하죠?

```
LOAD_ROM 0x0
{
    spaghetti.o (+RO)
    spaghetti.o (+RW)
}
```

그러면 Execution View는 어떻게 되는 건가요? 이제 Load View내부에 있는 괄호들이 그 역할을 하는 것입니다요. 내부에 있는 괄호를 무시하면 안 되요. 안에 있는 괄호들을 다시 한번 잘 살펴 보면

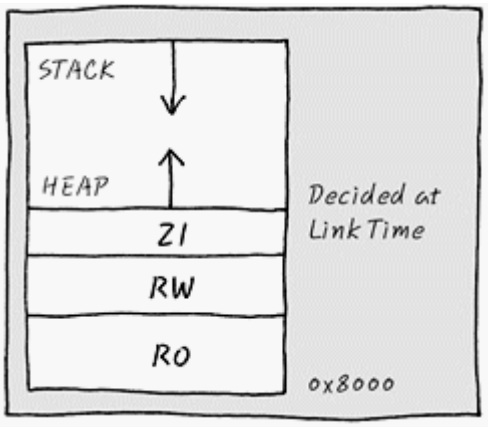
```
    EXEC_ROM 0x0
    {
        spaghetti.o (+RO)
    }
    EXEC_RAM 0x8000
    {
        spaghetti.o (+RW)
    }
    EXEC_RAM2 0xA000
    {
        spaghetti.o (+ZI)
    }
```

요렇게 생겼죠. 그러니까 0x0부터는 spaghetti.o 의 RO가, 0x8000부터는 spaghetti.o의 RW가, 0xA000부터는 spaghetti.o의 ZI가 들어간다~ 고 생각하시면 됩니다. 요 녀석들이 실제 Region이라고 보심 되요. 우와~ 간단하죠? 다시 한번 전체적으로 해석해 보면

```
LOAD_ROM 0x0   : Load Region : Loading 되어 있는 image의 모습
{
    EXEC_ROM 0x0 : Execution Region : Excution할때의 image의 모습
    {
        spaghetti.o (+RO)                  : 실행될때 spaghetti.o의 RO는 여기에, input section
    }
    EXEC_RAM 0x8000 : Execution Region
    {
        spaghetti.o (+RW)                  : 실행될때 spaghetti.o의 RW는 여기에, input sectoin
    }
    EXEC_RAM2 0xA000 : Execution Region
```

```
    {
        spaghetti.o (+ZI)                    : 실행될 때 spaghetti.o의 ZI는 여기에~, input section
    }
}
```

이런 Scatter Loading file을 linker의 input으로 넣으려면 어떻게 해야 할까요? 뭐 또 간단하죠. -scatter 라는 option을 주면 되요. armlink -scatter 파일이름.scl outfile1.o outfile2.o outfile3.o ...이런 식으로 주면 됩니다. 이런 Scatter File을 만들 때 한가지 유의할 점이 있습니다. 뭐냐면, Root Region이라는 게 꼭 있어야 한다는 점이에요. Root Region이라는 건 Loading View와 Execution View의 주소가 같은 Region을 말하는데, 이 Root Region이 꼭 하나씩은 존재해야 합니다. (FIXED Attribute을 이용해서 만들 수도 있어요 → FIXED는 Loading과 Execution을 같게 만들어라 라는 Scatterloading 지시어 에요) 사족 : Default Memory Map : 만일 Scatter Loading file을 쓰지 않으면 아주 간단한 Memory Layout이 Default로 적용이 됩니다. 뭐 별건 아니고, RO → RW → ZI 순으로 낮은 메모리 번지부터 차곡차곡 생기게 되는데요, 아래 그림처럼 쌓아요.



 RO의 시작 번지는 0x8000이고요, RO가 끝나는 지점부터 RW, ZI가 끝나는 지점부터 Stack과 Heap을 잡게 되는 겁니다. ZI의 크기는 Linker가 다 link 해 봐야만 아는 거니까, Heap과 ZI는 Linker가 Link를 다 끝내는 막장에 Heap과 Stack의 위치를 알아서 Automatic으로 정해 주게 됩니다. 그런데, 우리가 만들어낸 image의 ZI에 포함되지 않는 여기서의 Stack과 Heap은 도대체 뭐냐! 지금 말하고 있는 메모리 Model이 Default Memory 모델임을 잊지 마시고, 현재 RO, RW, ZI는 Embedded OS (또는 Kernel)같은 복잡한 것이 porting되지 않은, Application 하나 올린 거라고 생각하시면 됩니다. 그러니까, 그 Application이 사용하는 Stack과 Standard Library가 사용하는 (malloc같은) Heap인 겁니다. Compiler가 알아서 Stack과 Heap영역을 만들어 준다 고나 할까요. 우리가 직접 Stack과 Heap 영역을 만들어주는 case에는 compiler가 이런 stack과 heap영역을 자동으로 만들어 주는 것을 막아야 합니다. 그 방법은 "Scatterloading과 Bootup" 에서 다시 보시죠~ 그리고, RO의 맨 처음 0x8000번지에는 무엇이 들어갈까요. 그건 Symbol중에, __main 이라고 이름 붙은 녀석을 Entry point로 생각하고 Linker가 __main을 제일 앞쪽에 밀어 넣습니다.

 BSS가 실은 Block Started by Symbol의 약자 인데요, 왜 이렇게 부르는지 감이 오시는지요? BSS가 ZI 영역이나 같은 말인데요, RAM에 0으로 채워져서 잡히긴 하지만, ROM에는 실제로는 안 잡히죠. 무엇으로 잡히냐? $$을 이용한 Symbol로 시작과 끝만 알려주고 마는 거지요. 그러니까 ZI는 Symbol로만 잡혀 있으니까 이런 식으로 이름 붙여진 거랍니다.

linker, scatter, script, loading, 메모리, 다루기, Memorymap, arm, CrossCompile, elf