도전! RTOS를 만들어 봅시다 ①

RTOS의 정의와 기본구조/개발환경 설정

임베디드 시스템 프로그래머로 활동하고 있는 필자는 그간의 경험을 바탕으로 TI의 OMAP 평가 보드에 ARM9용으로 간단한 커널을 작성해 보았다. 상용 제품만큼 뛰어나지는 않지만, 기본적인 OS의 개념에 충실하게 설계하였으므로 참고자료로서 활용되기를 기대한다. 첫 순서로, 이 달에는 RTOS의 기본 개념 및 설계하려는 RTOS의 범위, 그리고 타깃 프로세서인 ARM9에 대해 간략히 설명한다.

글: 정승민/(주)텔슨전자 중앙연구소 분당연구분소 선임연구원
smjung21@naver.com

[연재순서]

1 RTOS의 정의와 기본구조/개발환경 설정
2 Startup Code와 인터럽트 처리
3 태스크 스케줄러 설계
4 태스크간 동기화 및 통신 설계

글을 시작하면서

운영체제(Operating System)는 주어진 H/W 또는 S/W 자원을 좀더 효율적으로 여러 사용자(또는 작업-태스크)에게 제공하기 위해 시스템을 관리하는 하나의 프로그램이다. 이러한 운영체제는 메인 프레임 또는 개인용 컴퓨터에서 우리가 흔히 접할 수 있는 UNIX, LINUX, Windows 등의 운영체제와 VxWorks, pSOS, Nucleus 등 이른바 내장형 시스템이라 불리는 상대적으로 작은 규모의 시스템에서 동작하는 운영체제로 구분되어 서로 다른 특징을 가지고 나름대로의 영역을 구축하여 발전해 가고 있다.

내장형 시스템을 구분 짓는 기준은 사실 애매한 점이 없지 않다. 단지 주어진 특정한 목적에 충실한 하나의 시스템을 내장형 시스템이라고 포괄적으로 정의할 수 있을 것이다. 그렇다면 특정한 목적에만 사용되는 시스템에 다목적으로 이용되는 컴퓨터에서나 볼 수 있는 운영체제가 왜 필요한 것일까? 기준이 애매한 만큼 정답은 없겠지만 가까운 답은 있다. 하나의 목적을 수행하기 위해 필요로 하는 기능들이 많아지고 있기 때문이다. 이러한 다양한 기능을 한정된 자원을 이용해서 모두 수행하려면 이를 관리, 조정하는 운영체제는 필수적이라 할 수 있을 것이다.

이 글에서는 점차 확산되고 있는 내장형 시스템의 운영체제에 대한 이해를 돕기 위해 간단한 내장형 운영체제를 만들어 보고자 한다. 물론 기능상 상용 운영체제에 비할 바는 못 되지만 덩치 큰 상용 운영체제에 비해 간단하고 운영체제의 기본적인 내용을 이해하는 데는 충분하리라 생각된다. 또한 실제 업무에서 주어진 시스템에 운영체제를 올리는 작업에서 일반적인 참고자료로서 활용될 수 있을 것이다.

더욱이 내장형 시스템의 기능과 종류가 다양해지고 있는 추세에서 운영체제의 일부 기능만을 필요로 하는 제한적인 시스템에서는 시스템에 맞는 운영체제를 직접 개발해야 하는 경우도 발생할 수 있다.

운영체제를 만든다고 하면 조금은 거창하게 들릴 지 모르겠다. 일반적인 상용 내장형 운영체제는 커널 이외에 각종 장치 드라이버, 네트워크 프로토콜, 파일 시스템 등을 포함하고 있어 그 구성이 복잡하고 방대해 보이지만, 이 글에서 다루고자 하는 부분은 운영체제의 가장 근간이 되는 태스크를 중심으로 이의 생성과 단순한 형태의 스케줄링 방법 그리고 태스크간의 통신과 동기화에 관련된 몇 가지 시스템 콜을 구현해 보는 것이다. 아울러 실시간 요소가 포함된 타이머의 구현에 관해서도 알아본다.

우선, 이름을 짓기로 하자. 우리가 만들게 될 운영체제는 기능이 대폭 간략화된 운영체제이므로 SOS(Simplified Operation System)이라고 하기로 한다. 그림 1은 우리가 만들게 될 SOS의 구조를 나타낸다. 그림에서 APPLICATION, PROCESSOR, LIBRARY를 제외한 나머지 부분이 우리가 직접 만들게 될 SOS의 커널 부분이며 애플리케이션에서 접근할 수 있도록 API를 제공한다.

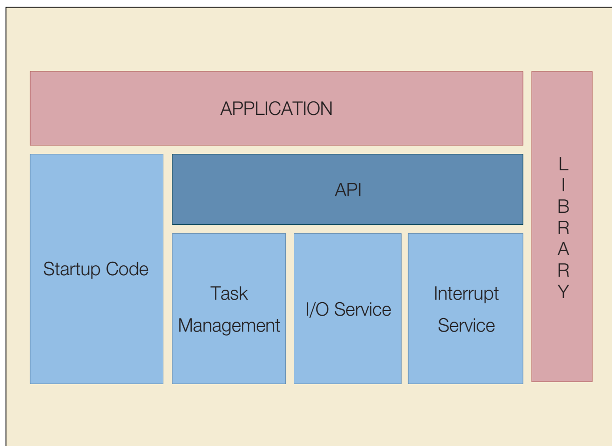


그림 1. SOS 아키텍처

이번 호에서는 운영체제를 만들기에 앞서 개발환경과 설정 방법 등에 대해 알아본다. 작성되는 모든 소스코드는 기본적으로 ANSI-C의 문법을 따르며, C로 코딩이 어려운 커널 내부는 어셈블리 명령어를 사용하여 구현한다.

프로세서

이제 정해진 목표로 나아가기 위한 첫 걸음으로, 우리가 만드는 운영체제를 동작시킬 프로세서를 알아보자.

최근 내장형 시스템 중에서, 특히 통신 단말기에 널리 사용되는 ARM 계열 프로세서는 널리 사용되는 만큼 정보나 자료를 찾고 구하기가 쉽다. ARM 프로세서가 개발된 것은 영국의 ARM(Advanced RISC Machine)사에 의해서 이지만, ARM사는 마이크로프로세서의 IP만을 설계하며 제품을 직접 생산하지는 않는다. 현재 많은 반도체 회사들이 ARM 코어를 라이선스하여 칩을 만들고 있으며, 따라서 칩을 만드는 회사에 따라 특성이 조금씩 차이가 나는 수많은 변종이 생산되고 있다.

이 글에서는 TI(Texas Instruments)에서 개발한 OMAP (Open Multimedia Application Platform)을 사용하기로 한다. OMAP 프로세서는 멀티미디어 애플리케이션의 처리에 적합하도록 Dual-core 구조를 가지고 있다. 즉, MPU(Micro Processor Unit)와 DSP(Digital Signal Processor)를 하나의 칩에 모두 내장하고 있는 것이다. 이 중 MPU는 ARM925T를 그리고 DSP는 TMS320C5510을 사용한다.

우리는 MPU에 해당하는 ARM925T만을 이용하게 될 것이다. ARM925T는 ARM9TDMI의 확장된 형태로 보면 되고 우리가 다루고자 하는 부분은 ARM 계열 프로세서의 일반적인 구조와 구성을 벗어나지 않는다.

운영체제가 개발될 시스템의 사양 중에서 관련된 부분은 다음과 같다.

- Platform: OMAP1509 EVM(ARM925T)
- Memory: SRAM 512KB, SDRAM 128MB
- 32KHz RTC(Real time clock)
- IDE: TI Code Composer Studio 2.0

운영체제는 프로세서를 직접 다루어야 하므로, 그 특성을 파악하는 일이 무엇보다 중요하다. 특히 프로세서 상태와 내부의 레지스터 구조 및 용도, 그리고 인터럽트 처리 방식에 대한 이해를 필요로 한다. 이에 대한 정보는 "*ARM 뛰어넘기*"라는 제목으로 본지에 연재된 기사에 자세하게 나와 있으므로 참고하기로 하고, 여기서는 우리가 직접적으로 사용하게 될 프로세서의 몇 가지 특징에 대해서만 간단하게 알아보자.

운영모드

먼저 ARM은 명령어 세트에 따라 16bit THUMB 명령어와 32bit ARM 명령어를 지원한다. 두 가지 모드의 차이점은 여러 가지가 있겠지만, 가장 큰 특징은 메모리 구성이나 크기 등에 따른 차이점이다. THUMB 모드의 경우 16bit 크기의 명령어 세트를 사용하므로 메모리 크기가 제한적인 시스템에 유리하다. 이 글에서는 ARM 모드를 사용한다.

그리고 ARM에서 지원하는 프로세서의 동작 모드는 총 7가지이다. 이 중에서 우리는 운영체제를 만드는 것이므로 권한이 높은 모드이면서 프로세서 초기화 후 기본 모드로서 별다른 변경 없이 Supervisor 모드를 사용한다.

레지스터

눈여겨 봐야 할 레지스터들이 몇 개 있다. 프로세서의 상태를 알 수 있는 CPSR과 현재 실행되는 코드를 가리키는 PC, 스택을 지정하는 SP(R13), 그리고 복귀주소를 가지는 LR(R14)이다. 이 레지스터들은 태스크의 교환 작업에서 중요한 역할을 수행한다.

인터럽트

ARM에서 인터럽트는 Exception Handler라는 이름으로 운용된다. EVT(Exception Vector Table)는 총 7개로 구성되며 이 중 운영체제의 실시간 처리를 위한 타이머가 IRQ로 발생한다.

메모리 구성

프로세서는 독립적으로 실행될 수 없으며 가장 기본적으로 메모리에 저장된 코드를 읽어야 사용자가 지시한 동작을 수행할 수 있다. 메모리 구성은 운영체제와는 뗄 수 없는 밀접한 관계이므로 반드시 확인하고 운영체제에서 사용할 메모리 영역을 결정하여야 한다. 그럼 첫 단계로 우리가 선정한 시스템의 메모리 구성을 살펴보자(**리스트 1** 참조).

시스템에서 제공하는 메모리 중 우리가 만드는 운영체제의 코드는 BOOT SRAM으로 로딩되어 실행되도록 설정되었다.

주소영역	메모리 종류 및 크기
0x0000:0000 0x0007:FFFF	BOOT SRAM (512KB)
0x0008:0000 0x003F:FFFF	RESERVED
0x0040:0000 0x007F:FFFF	BOOT FLASH (4MB)
0x0c00:0000 0x0DFF:FFFF	USER FLASH (32MB)
0x0E00:0000 0x0FFF:FFFF	RESERVED
0x1000:0000 0x01FF:FFFF	SDRAM (32MB)

리스트 1. 시스템 메모리 구성

이 중에서 EVT는 반드시 0x0000:0000번지부터 0x20만큼을 할당해야 한다. 운영체제를 어느 위치에 넣을 것인가는 시스템에 따라 차이가 많지만, 개발과정에서는 로딩이 쉽고 빠른 SRAM을 사용하는 것이 효율적이다.

그리고 운영체제에서 사용하는 변수, 동적 메모리 등은 모두 SDRAM에 할당된다. 여기서는 우리가 사용하고자 하는 물리적인 메모리의 구성이 어떠한가를 알아보는 것이므로, 이에 대한 자세한 내용은 다음 장 "컴파일과 링크"에서 다루기로 한다.

컴파일과 링크

우리가 사용하게 될 통합 개발환경은 TI CCS(Code Composer Studio)이며, 여기서 컴파일러와 어셈블러 그리고 링커를 호출하여 소스 코드를 실행 파일로 만들어준다. 최종 실행 파일을 COFF(Common Object File Format) 형태로 제공한다.

그림 2는 일반적인 실행 파일 생성 과정을 나타내는 것이다. 우리의 목적은 운영체제를 만드는 것이므로 자세한 설명은 생략하기로 한다.

그림 3에 TI CCS의 실행 모습이 나와 있다. 일반적인 통합 개발 툴과 기능상 크게 차이점은 없고, 앞서 설명했듯이 OMAP 프로세서가 Dual-core를 사용하므로 MPU와 DSP가 모두 지원된다는 점이 특이하다고 할 수 있다. 그러나 우리는 MPU 부분만을 사용할 것이므로 크게 신경 쓰지 않아도 된다.

TI CCS 컴파일러에도 일반적인 컴파일러에서 지원하는 다양한 기능을 사용할 수 있도록 옵션을 제공하지만, 우리는 아래의 한 가지 옵션만 지정해 주는 것으로 충분하다. 컴파일러 옵션의 지정은 TI CCS의 build options 메뉴를 이용한다.

```
-me
```

Memory Format을 Little endian으로 사용하겠다는 의미이다. Memory Format에는 Little endian, Big endian Mode가 있는데 OMAP의 ARM925T에서는 Little endian만 지원한다.

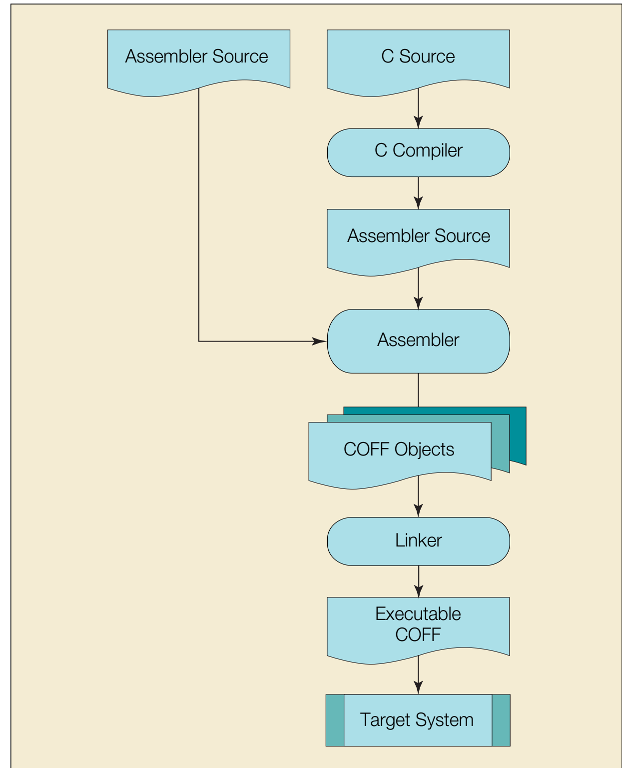


그림 2. 실행 파일 생성 과정

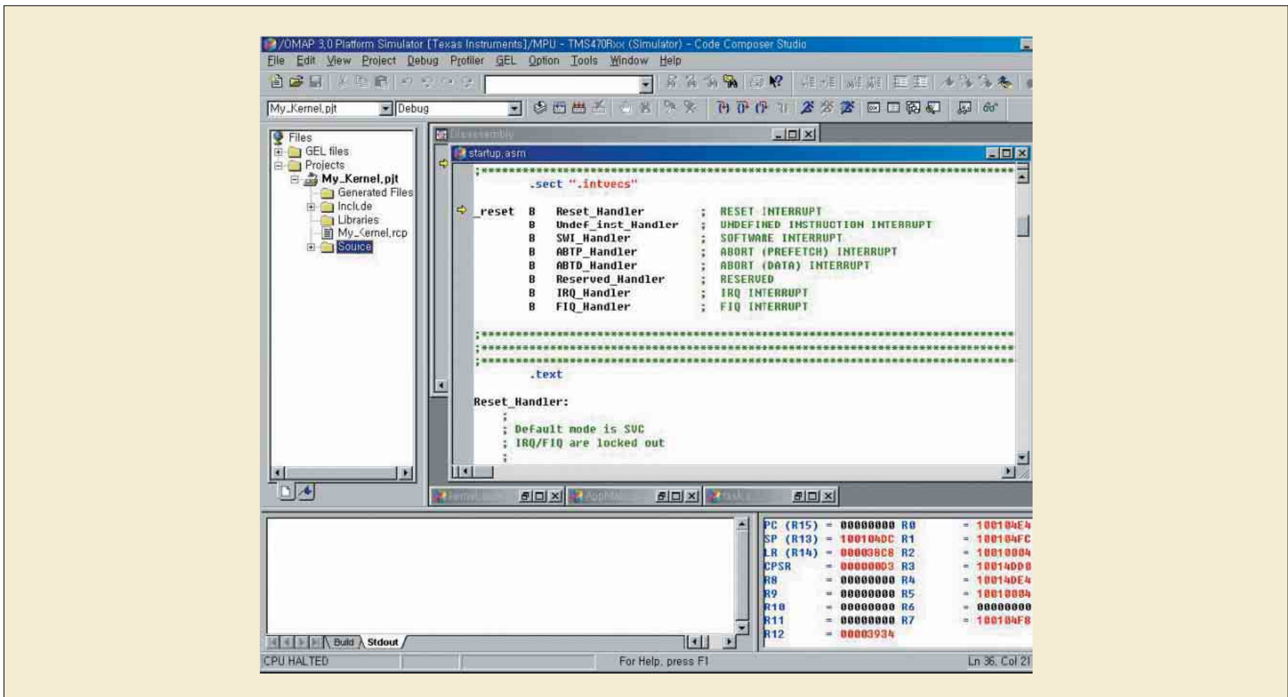


그림 3. TI Code Composer Studio

TI CCS 링커를 사용하기 위해서 가장 먼저 해야 할 일은 링커에서 사용할 옵션과 시스템에서 제공되는 물리적인 메모리 중 우리가 사용할 영역을 알려주는 Linker command file을 작성하는 것이다. 리스트 2와 같은 형태로 설정해 준다.

Linker command file의 내용을 자세히 살펴보자. 우선은 링커에게 지시하는 각종 옵션들이 있다. 링커에서 제공하는 기능에 따라 많은 옵션이 존재하지만, 여기서는 가장 기본적인 옵션 몇 가지를 설명하기로 한다.

```
Linker Command File

/* LINKER OPTIONS */
-l ./lib/rts32e.lib
-o ./out/sos_kernel.out
-m ./out/sos_kernel.map
-e _reset
-stack  0
-heap   0x10000

/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
   E_MEM: org = 0x00000000
          len = 0x00000020
   P_MEM: org = 0x00000020
          len = 0x0007FFE0
   D_MEM: org = 0x10000000
          len = 0x007FC000
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
SECTIONS
{
     .intvecs:
       {} > E_MEM
     .text:
       {} > P_MEM
     .cinit:
       {} > P_MEM
     .const:
       {} > P_MEM
     .sysmem:
       {} > D_MEM
     .stack:
       {} > D_MEM
     .data:
       {} > D_MEM
     .bss:
       {} > D_MEM
}
```

리스트 2. Linker command file

```
-l ./lib/rts32e.lib
```

-l은 사용하고 싶은 라이브러리를 지정하는 링커 옵션이다. 설정된 rts32e.lib는 TI에서 제공하는 ANSI-C 표준 라이브러리이다.

```
-o ./out/sos_kernel.out
```

-o는 최종 실행 파일의 경로를 표시하는 링커 옵션이다. 최종 실행 파일은 COFF(Common Object File Format) 형태로 출력된다.

```
-m ./out/sos_kernel.map
```

-m은 최종 실행 파일에 대한 메모리 구성 파일의 경로를 표시하는 링커 옵션이다. 메모리 구성 파일은 디버깅에 유용하게 쓰인다.

```
-e _reset
```

-e는 시스템 초기화 상태에서 프로그램 카운터(PC)가 가리키게 되는 위치를 표시한다. 시스템이 초기화되면 EVT(Exception Vector Table) 중에서 Reset vector를 수행해야 하므로 _reset으로 정의한다.

```
-stack 0
```

-stack는 스택의 크기를 지정하는 링커 옵션이다. 특이한 것은 크기를 0으로 했다는 점인데, 그렇다면 스택을 사용하지 않는다는 것인가?

그렇지는 않다. 우리는 지금 운영체제를 만들기 위한 설정을 하고 있다는 점을 다시 한번 상기하자. 다음 호에서 자세히 설명되겠지만 ARM 프로세서에서는 동작 모드에 따라 서로 다른 스택을 설정해야 한다. 이 때 별도로 지정한 메모리를 스택으로 사용하게 될 것이다. 이 옵션은 단지 링크하는 동안 발생하는 경고 메시지를 없애기 위한 것이다.

```
-heap 0x10000
```

-heap은 Heap 메모리의 크기를 지정하는 링커 옵션이다. 이 옵션의 경우 -l로 지정한 ANSI-C 라이브러리의 동적 메모리 관련 함수에서 사용하기 위해 정의된다. 여기서는 64KB를 지정하였다.

메모리 관리는 운영체제의 중요한 기능 중 하나이지만, 이 글에서 구현하고자 하는 목표는 태스크 관리가 중심이 되므로 우선은 제공되는 라이브러리의 동적 메모리 관리 함수를 이용하기로 한다.

다음은 메모리를 구성하는 방법에 대해서 알아보자. 메모리 구성에 관한 내용은 프로그램을 실제 동작시킬 시스템의 메모리 사양을 그대로 옮기는 것에 불과하다.

```
MEMORY
{
…
}
```

MEMORY는 링커에게 사용 가능한 메모리 영역을 알려주는 역할을 하며, 다음과 같은 내용이 포함된다.

```
E_MEM:
    org = 0x00000000
    len = 0x00000020
```

주소 0x0부터 크기 0x20의 영역을 E_MEM이라는 이름으로 사용하겠다는 의미이다. 이 영역은 EVT이 들어가는 영역이라는 의미로 E_MEM으로 이름을 지었다.

```
P_MEM:
    org = 0x00000020
    len = 0x0007FFE0
```

주소 0x20부터 크기 0x7FFE0의 영역을 P_MEM이라는 이름으로 사용하겠다는 의미이다. 이 영역은 실제 프로그램이 포함되는 영역이므로 P_MEM으로 이름 지었다. 여기서 잠깐 리스트 1을 보자. 주소 0x00000000로 지정된 영역은 BOOT SRAM(512KB)이다. 따라서 E_MEM와 P_MEM이 BOOT SRAM에 포함되어 있음을 알 수 있으며, 또한 E_MEM과 P_MEM의 크기를 합하면 0x80000(512K)으로 BOOT SRAM의 크기와 동일하다.

```
D_MEM:
    org = 0x10000000
    len = 0x00800000
```

주소 0x20부터 크기 0x7FFE0의 영역을 D_MEM이라는 이름으로 사용하겠다는 의미이다. 이 영역은 프로그램에서 선언되어 사용되는 각종 Data와 변수가 포함되는 영역이므로 D_MEM으로 이름 지었다. 여기서 잠깐 리스트 1을 다시 한번 보자. 0x10000000로 지정된 주소 영역은 SDRAM(32MB)이다. 따라서 D_MEM은 SDRAM에 포함되어 있음을 알 수 있다. 여기서 SDRAM의 크기는 32MB이지만, D_MEM의 크기는 8MB(len=0x00800000)로 잡혀 있음을 볼 수 있다. 실제 물리적으로는 32MB가 존재하지만 운영체제에서는 8MB만 사용한다는 의미가 된다.

마지막으로 섹션을 구성하는 방법에 대해서 알아보자. 우리가 작성한 Source code는 컴파일러에 의해 컴파일 되면 Object code가 생성된다. 여기서 말하는 섹션이란, 여러 개의 Object code에서 동일한 특성을 갖는 영역들을 모아놓은 각각을 의미한다. 이와 같은 섹션은 링커에 의해 생성되는 것도 있지만 사용자에 의해 지정될 수도 있다. 대표적인 섹션의 종류로는 다음의 **리스트 3**과 같은 것들이 있다.

섹션	의미
.cinit	초기화된 전역, 정적 변수의 초기값을 가지고 있는 영역
.const	Const로 선언된 변수들이 저장되는 영역
.text	실행 코드와 상수가 저장되는 영역
.data	어셈블리 코드에 의해 생성되는 변수 영역
.bss	전역, 정적 변수가 생성되는 영역
.sysmem	동적 메모리 할당을 위한 영역

리스트 3. 섹션의 종류와 의미

```
SECTIONS
{
…
}
```

SECTIONS는 링커에게 정의된 섹션들이 MEMORY에서 정의된 메모리 영역의 어느 부분에 포함되는지를 알려주는 역할을 하며, 다음과 같은 내용이 포함된다.

```
.intvecs:
  {} > I_MEM
```

.intvecs 섹션을 I_MEM으로 넣겠다는 의미이다. .intvecs 섹션은 사용자에 의해 정의된 섹션으로 EVT을 의미한다.

```
.text:
  {} > P_MEM
.cinit:
  {} > P_MEM
.const:
  {} > P_MEM
```

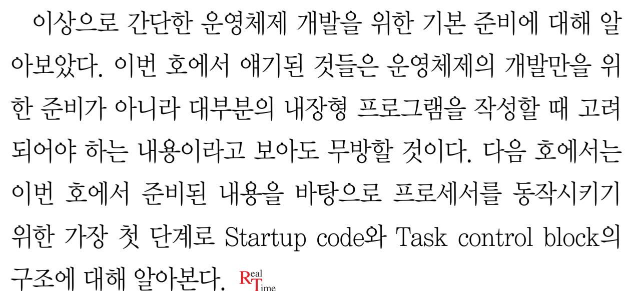
.text, .cinit, .const 섹션을 P_MEM으로 넣겠다는 의미이다. 이 영역들은 실제 시스템에서 ROM 또는 FLASH Memory에 할당되는 영역이며, 실행 코드가 저장된 .text 영역의 경우 실행 속도 등의 문제로 접근 속도가 빠른 RAM으로 옮긴 후 실행되기도 한다. 그러나 앞서 말했듯이 개발과정에서는 로딩이 쉽고 빠른 SRAM을 사용하기로 한다.

```
.sysmem:
  {} > D_MEM
.stack:
  {} > D_MEM
.data:
  {} > D_MEM
.bss:
  {} > D_MEM
```

.sysmem, .stack, .data, .bss 섹션을 D_MEM으로 넣겠다는 의미이다. 앞서 설명했듯이 .stack 섹션은 실제로는 존재하지 않는다. 각 섹션의 의미는 리스트 3을 참조한다.

사용자는 이렇게 작성된 Linker command file을 Linker가 사용할 수 있도록 CCS의 프로젝트 파일에 포함시켜야 한다.

첫 회를 마치며



이상으로 간단한 운영체제 개발을 위한 기본 준비에 대해 알아보았다. 이번 호에서 얘기된 것들은 운영체제의 개발만을 위한 준비가 아니라 대부분의 내장형 프로그램을 작성할 때 고려되어야 하는 내용이라고 보아도 무방할 것이다. 다음 호에서는 이번 호에서 준비된 내용을 바탕으로 프로세서를 동작시키기 위한 가장 첫 단계로 Startup code와 Task control block의 구조에 대해 알아본다.