

## 【 기술 노트 20 】

## AVR의 내부 SRAM 및 외부 데이터 메모리의 사용 방법

AVR은 프로그램 메모리로 MCU 내부에 플래시 메모리를 가지고 있으며, 사용자가 외부에 프로그램 메모리를 추가로 확장하는 기능이 없다. 그러나, 데이터 메모리는 내부에 SRAM을 가지고 있지만 이것의 용량이 모자라는 경우에는 사용자가 외부에 추가로 데이터 메모리로서 SRAM을 확장할 수 있다.(AVR은 하버드 구조로 되어 있으므로 프로그램 액세스용 버스와 데이터 액세스용 버스를 따로 가지고 있어야 하지만, 이러한 이유로 AVR의 외부에는 데이터 액세스용 버스 1가지만을 가지고 있다.) AVR의 내부 SRAM은 꽤 용량이 큰 편이므로 대부분의 경우 스택이나 웨만한 정도의 메모리 변수는 이것으로 모두 처리하지만, 특별히 큰 용량의 데이터 메모리를 필요로 하는 응용분야에서는 이것만으로는 부족할 수 있기 때문이다. AVR의 내부에는 이밖에도 불휘발성 데이터 메모리로서 소용량의 EEPROM을 내장하고 있다.

AVR-GCC에서는 사용자가 정의한 메모리 변수나 스택을 기본적으로 내부 SRAM에 할당하여 사용하므로 사용자는 이를 별로 의식하지 않고 프로그램을 작성하게 된다. 그러나, 외부에 데이터 메모리를 확장하여 사용하고자 하는 경우에는 사용자가 링커에게 이에 대해 올바르게 지시해야 한다. 그러나, 이와 같이 사용자가 **외부 데이터 메모리를 사용하는 방법**에 대하여는 잘 알려져 있지 않으므로 여기에서 이를 체계적으로 정리하여 보기로 한다.

◀참고▶ AVR 모델 중에 규모가 작은 대부분의 제품들은 외부 버스 신호선들을 가지고 있지 않으므로 외부에 데이터 메모리를 확장하는 기능이 없다. 그러나, 비교적 규모가 큰 ATmega8515, ATmega161, ATmega162, ATmega64, ATmega128과 같은 모델들은 외부 데이터 메모리 확장 기능을 가지고 있다.

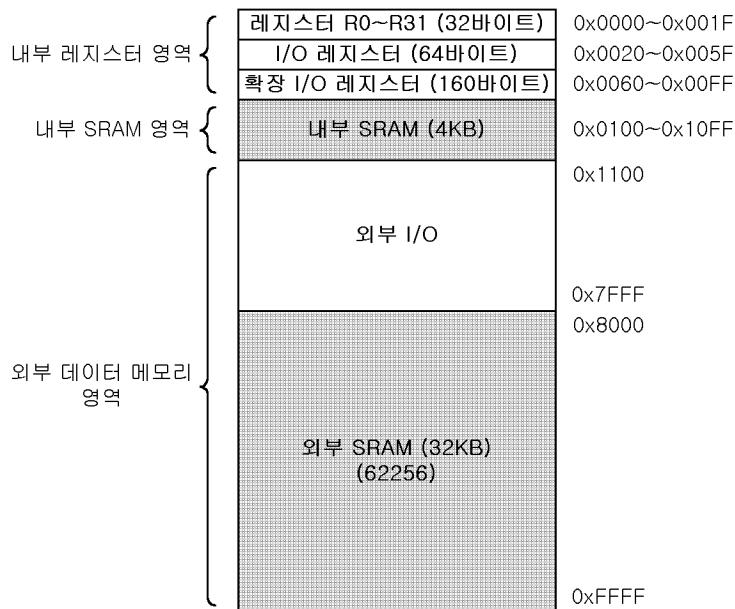
### 1. ATmega128 및 OK-128 키트에서의 내부 SRAM과 외부 데이터 메모리 확장

AVR에서 데이터 메모리는 무조건 0x0000~0xFFFF 번지를 사용하는데, 이중에서 가장 아래쪽은 레지스터 영역이고, 그 위에 내부 SRAM이 위치하며, 다시 그 위는 외부 데이터 메모리 영역이 된다. 그러나, AVR의 모델에 따라 레지스터의 수나 내부 SRAM의 용량이 달라지므로 당연히 모델에 따라 외부에 확장할 수 있는 데이터 메모리의 용량도 달라지게 된다. 그러므로, 여기서는 ATmega128 및 OK-128 키트를 중심으로 설명한다. 하지만, 다른 AVR 모델의 경우에도 메모리 번지가 달라지는 것을 제외하면 이와 똑같은 방법으로 적용이 되므로 원리를 이해하고 사용하는데 문제는 없을 것이다.

ATmega128을 사용한 OK-128 키트의 데이터 메모리 맵은 <그림 20.1>에 보인 바와 같다. 즉, ATmega128은 0x0100~0x10FF 번지에 4KB의 SRAM을 내장하고 있으므로, 사용자

가 외부 데이터 메모리는 0x1100~0xFFFF 번지에 약 60KB 정도를 확장할 수 있다. 그러나, OK-128 키트에서는 0x1100~0x7FFF 번지를 외부 I/O 확장 영역으로 사용하고, 0x8000~0xFFFF 번지에 32KB의 SRAM인 62256을 확장하여 사용한다.

혹자는 내부 SRAM에 이어서 0x1100 번지부터 외부 SRAM을 사용해야만 내부 데이터 메모리와 외부 데이터 메모리가 연속되어 있어서 편리할 것이라고 생각할 수 있지만 이는 그렇지가 않다. AVR-GCC에서는 4KB 내부 SRAM의 처음인 0x0100 번지부터 변수 데이터 영역으로 사용하고 이 영역의 끝 부분을 스택으로 사용하기 때문에 변수 데이터가 많아지더라도 스택 부분을 건너뛰어 외부 메모리를 사용해야 하므로 어차피 메모리 변수는 불연속으로 처리될 수밖에 없다. 스택을 내부 메모리가 아니라 외부 데이터 메모리에 할당하여 사용한다면 이런 문제를 피할 수는 있으나, **스택은 매우 빈번하게 사용하는 데이터 메모리이므로 내부 SRAM을 사용하는 것이 바람직하며 AVR-GCC에서도 이렇게 하도록 강력히 권장하고 있다.** 따라서, OK-128 키트에서는 어드레스 디코딩이 용이하고 32KB를 손쉽게 접속할 수 있도록 0x8000~0xFFFF 번지를 외부 데이터 메모리로 사용하고, 그 아래의 자투리 영역인 0x1100~0x7FFF 번지를 외부 I/O 확장 영역으로 사용하는 것으로 설계하였다.



<그림 20.1> ATmega128 및 OK-128 키트에서의 데이터 메모리 맵

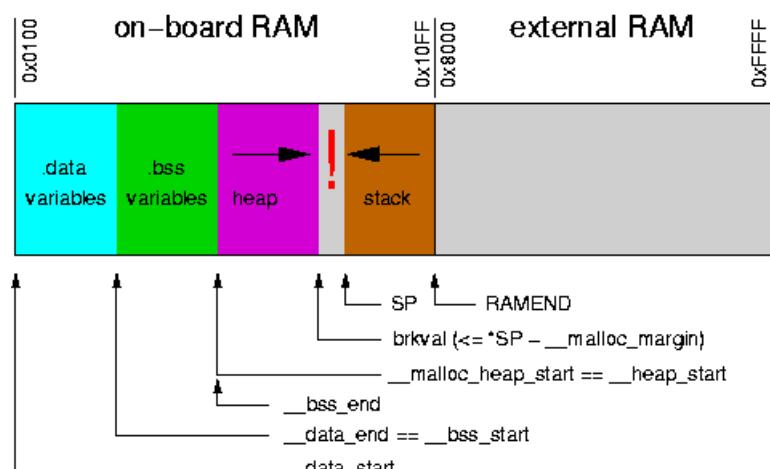
이와 같이 ATmega128의 외부에 데이터 메모리와 I/O를 확장하는 경우에는 반드시 MCUCR 레지스터에서 SRE 비트를 1로 설정하고, MCUCR, XMCRA, XMCRA 레지스터 등을 사용하여 외부 어드레스 영역을 액세스하기 위한 버스 신호들의 동작을 올바르게 초기화해

**주어야 한다.** OK-128 키트에서는 외부 I/O 확장 영역에는 다소 느린 소자가 사용될 수 있는 것을 고려하여 1개의 웨이트 사이클을 부여하고, 외부 데이터 메모리 확장 영역에는 빠른 SRAM을 사용하므로 제로 웨이트로 사용하고 있다.

## 2. AVR-GCC에서 섹션의 사용

AVR-GCC에서는 C컴파일러가 기본적으로 .text 섹션, .data 섹션, .bss 섹션, .noinit 섹션, .eeprom 섹션 등과 같은 5개의 섹션을 사용한다. 당연히 .text 섹션은 플래시 메모리를 사용하는 실행 코드용의 섹션이고, .eeprom 섹션은 EEPROM을 사용하는 불휘발성 데이터 메모리용의 섹션이며, 나머지 섹션들은 모두 SRAM을 사용하는 데이터용 섹션이다. 이밖에도 SRAM은 malloc() 함수 등을 위한 힙(heap) 메모리로 사용될 수 있다.

AVR-GCC에서 특별히 데이터 메모리의 각 섹션 위치를 지정하지 않으면 기본적으로 내부 SRAM의 낮은 번지에서부터 .data 섹션, .bss 섹션, .noinit 섹션, 힙, 스택의 순서로 배열된다. ATmega128의 경우 이를 그림으로 보이면 <그림 20.2>와 같다.



<그림 20.2> ATmega128에서 기본적인 데이터 메모리 할당

데이터 메모리의 섹션들이 이와 같은 순서로 배열되는 것은 기본적으로 링커 스크립트 파일에서 이렇게 할당되도록 지정되어 있기 때문이다. 즉, WinAvr\avr\lib\ldscripts\avr5.x 파일에는 <파일 1>처럼 지정되어 있는 것을 볼 수 있다.

<파일 1> AVR-GCC의 링커 스크립트 파일 avr5.x

```
/* Default linker script. for normal executables */
OUTPUT_FORMAT("elf32-avr", "elf32-avr", "elf32-avr")
```

```

OUTPUT ARCH(avr:5)
MEMORY
{
    text    (rx)    : ORIGIN = 0, LENGTH = 128K
    data    (rw!x)   : ORIGIN = 0x800060, LENGTH = 0ffa0
    eeprom  (rw!x)   : ORIGIN = 0x810000, LENGTH = 64K
}
SECTIONS
{
    /* Read-only sections, merged into text segment: */
    .hash      : { *(.hash) }
    .dynsym    : { *(.dynsym) }

    .

    /* Internal text space or external memory */
    .text :
    {
        *(.vectors)
        ctors start = . ;
        *(.ctors)
        ctors_end = . ;
        .
        .
    } > text

    .data    : AT (ADDR (.text) + SIZEOF (.text))
    {
        PROVIDE (__data_start = .) ;
        *(.data)
        *(.gnu.linkonce.d*)
        . = ALIGN(2);
        edata = . ;
        PROVIDE (__data_end = .) ;
    } > data

    .bss    SIZEOF(.data) + ADDR(.data) :
    {
        PROVIDE (__bss_start = .) ;
        *(.bss)
        *(COMMON)
        PROVIDE (__bss_end = .) ;
    } > data
        data load start = LOADADDR(.data);
        __data_load_end = __data_load_start + SIZEOF(.data);

    /* Global data not cleared after reset. */
    .noinit  SIZEOF(.bss) + ADDR(.bss) :
    {
        PROVIDE ( noinit_start = .) ;
        *(.noinit*)
        PROVIDE ( noinit_end = .) ;
        end = . ;
        PROVIDE ( __heap_start = .) ;
    } > data

    .eeprom  : AT (ADDR (.text) + SIZEOF (.text) + SIZEOF (.data))
    {
        *(.eeprom*)
        eeprom end = . ;
    } > eeprom
}

```

```

.
.
.
}
```



그러나, 사용자가 링커 옵션을 이용하면 이들 섹션의 배열 순서나 각 섹션의 시작번지를 이와 다르게 임의로 지정할 수 있으며, 메모리 변수가 많아져서 내부 SRAM의 용량이 부족할 경우에는 외부에 추가로 확장한 데이터 메모리를 사용할 수도 있다. 단, 이 경우에도 스택은 항상 내부 SRAM의 가장 뒷부분을 사용하도록 그냥 두고 움직이지 않는 것이 좋다. 스택을 외부 SRAM에 할당하면 시스템의 속도가 현저하게 저하되며, 일부 모델에 따라서는 시스템의 동작에 문제를 일으킬 수도 있다.

스택의 용량은 충분히 확보하는 것이 좋으나 내부 SRAM의 용량은 일정하게 한정되어 있으므로 이와 같이 스택의 용량을 크게 확보할수록 사용자 변수용 데이터 메모리의 공간은 줄어들게 되므로 외부에 SRAM을 확장해야 하는 필요성은 증대된다.

**데이터 메모리에서 이와 같이 섹션들을 할당할 때 항상 주의할 사항은 힙과 스택이 용량이 커져서 서로 충돌하지 않도록 하는 것이다.** 이를 방지하기 위하여 힙과 스택 사이에는 32바이트 이상의 공간(margin)을 두는 것이 바람직하다. 힙과 스택이 충돌할 가능성이 있는 경우에는 힙을 충분한 용량을 가지는 외부 확장 메모리에 옮겨 할당하는 것이 좋다.

### (1) .text 섹션

.text 섹션은 **프로그램의 실행 코드로 이루어진 섹션**으로서, 여기에는 사용자가 작성한 프로그램의 실행 코드는 물론이고 서두 부분에 .initN 섹션과 말미에 .finiN 섹션을 함께 포함하고 있다.

.initN 섹션은 스타트업 코드(startup code)를 정의하는 부분으로서 리셋 후에 main() 함수를 실행하기 전의 사용자 프로그램 준비 동작을 수행한다. 이것은 0~9의 10단계 동작으로 이루어지는데, 주요 기능을 보면 리셋 후에 특정한 번지로 점프하고, 스택을 초기화하며, .data 섹션을 플래시 메모리에서 SRAM으로 복사하여 초기화한 후에 main() 함수로 점프하는 것이다.

.finiN 섹션은 main() 함수에서 리턴한 후에 실행될 탈출 코드(exit code)를 정의하기 위한 부분으로서 사용자 프로그램 종료 후의 동작을 수행한다. 이것은 9~0의 10단계 동작으로 이루어지는데, 주요 기능을 보면 사용자 프로그램을 종료하고 exit() 함수를 실행한 후에 아무 기능도 수행하지 않는 무한 루프로 들어가는 것이다. MCU에서는 이와 같이 하지 않으면 프로그램이 미로에 빠져서 예측할 수 없는 결과를 초래할 수 있기 때문이다.

.text 섹션은 디폴트로 **플래시 메모리**의 0x0000 번지에서 시작하지만, 사용자는 **-Wl,**

`-Ttext=address` 옵션을 사용하여 링커에게 .text 섹션의 시작번지를 임의로 지정할 수 있다.

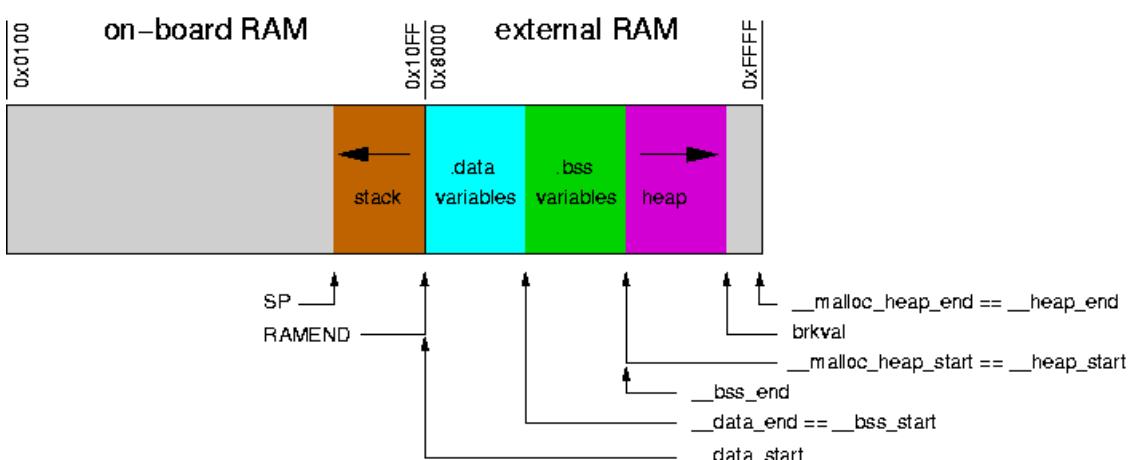
플래시 메모리에 있는 .text 섹션의 뒤에는 이밖에도 .data 섹션을 초기화하기 위한 내용과 .eeprom을 초기화하기 위한 내용을 추가로 가지고 있다. 따라서, **플래시 메모리를 차지하는 용량을 계산할 때는 .data 섹션을 초기화하기 위한 데이터 공간을 고려하여 .text 섹션에 .data 섹션의 크기(와 .eeprom 섹션의 크기)를 합해야 하며, SRAM 용량을 계산할 때는 .data 섹션과 .bss 섹션을 합해야 한다.**

## (2) .data 섹션

.data 섹션은 초기값이 정의된 전역변수나 함수의 파라미터처럼 정적인 데이터(static data)를 포함하는 섹션으로서 이는 반드시 SRAM 영역에 할당된다.

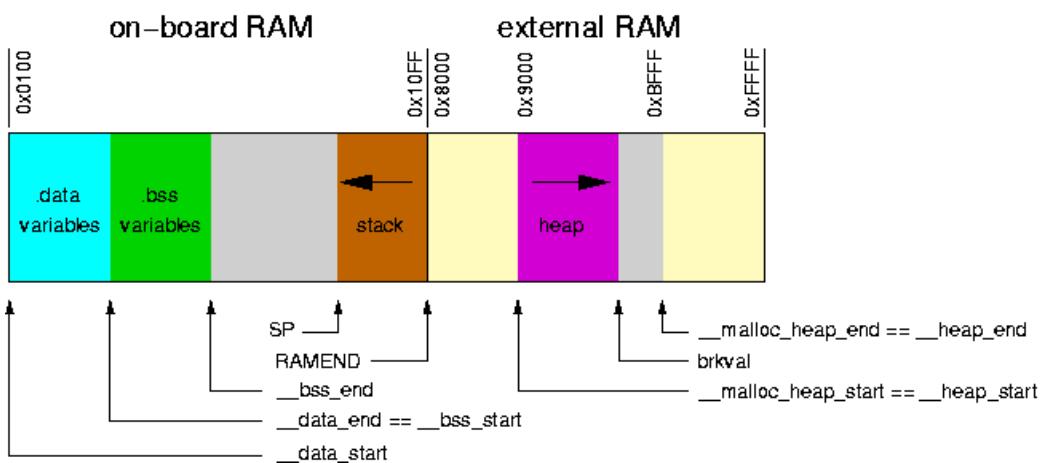
.data 섹션은 기본적으로는 내부 SRAM의 첫 번째 번지부터 시작하도록 할당되지만, 사용자는 `-WI, -Tdata=address` 옵션을 사용하여 링커에게 .data 섹션의 시작번지를 임의로 지정할 수 있다. 그런데, 여기서 `address`를 지정할 때는 실제의 SRAM 물리번지에 `0x800000`을 더하여 나타내야 한다. 즉, .data 섹션을 시작할 실제의 SRAM 번지가 `0x8000`이라면 여기에 SRAM을 나타내는 옵셋값 `0x800000`을 더하여 `-WI, -Tdata=0x808000`으로 옵션을 지정해야 한다는 것이다.

.data 섹션의 뒤에는 기본적으로 .bss 섹션이나 .noinit 섹션이 이어지는데, .data 섹션의 시작번지를 다른 값으로 지정하면 이들 .bss 섹션이나 .noinit 섹션들도 자동적으로 따라서 움직인다. 그러나, 스택은 이와 함께 움직이지 않는다. 예를 들어 ATmega128의 경우 `-WI, -Tdata=0x808000, --defsym=__heap_end=0x80ffff`로 옵션을 지정하였다면 각 섹션의 할당 위치는 <그림 20.3>과 같다.



<그림 20.3> ATmega128에서 .data 섹션을 0x8000 번지에 할당한 경우의 메모리 맵

그러나, 이와 같이 스택을 제외한 모든 데이터 영역을 외부 SRAM으로 옮겨서 할당하지 않고 일부 섹션만을 외부 메모리에 할당할 수도 있다. 예를 들어 ATmega128에서 힙만을 0x9000~0xBFFF 번지에 할당하겠다면 -WI, --defsym=\_\_heap\_start=0x809000, --heap\_end=0x80bfff로 옵션을 지정해야 하며, 이때 각 섹션의 할당 위치는 <그림 20.4>와 같다.



<그림 20.4> ATmega128에서 힙을 0x9000~0xBFFF 번지에 할당한 경우의 메모리 맵

◀참고▶ .data, .bss, .noinit와 같은 휘발성 데이터 섹션에서는 이를 의미하는 옵셋값으로 0x800000을 사용하는데 비하여 EEPROM 섹션에서는 이를 의미하는 옵셋값으로 0x810000을 사용하도록 되어 있다.

◀참고▶ AVR-GCC에서는 이와 같이 링커 옵션을 사용하여 데이터 섹션을 외부 데이터 메모리에 할당하도록 지정한다고 하여 이들 영역이 외부 메모리에 정상적으로 할당되어 프로그램이 올바르게 동작하는 것이 아니라는 유의해야 한다. 실제로 AVR-GCC에서 테스트하여 보면 .data 섹션을 외부 데이터 메모리로 옮길 경우 .data 섹션과 그 뒤에 이어지는 나머지 섹션들이 모두 외부 메모리의 지정된 번지에 할당되기는 하지만 프로그램은 올바르게 동작하지 않는 것을 확인할 수 있다.

AVR-GCC에서 .data 섹션이 이와 같이 정상적으로 처리되는 않는 것은 이것의 초기화 과정 때문이다. 즉, .text 섹션 내부에 있는 .initN 섹션은 .text 섹션의 마지막 부분에 저장되어 있던 초기화 데이터를 .data 섹션으로 복사하는 작업을 수행하는데 이때는 아직 ATmega128이 외부 버스를 사용하도록 초기화가 되어 있지 않은 상태이기 때문에 CPU가 외부 메모리로 지정된 .data 섹션을 액세스할 수가 없는 것이다. 외부 메모리를 액세스할 수 있도록 ATmega128의 MCUCR, XMCRA, XMCRB 레지스터를 초기화하는 것은 나중에 사용자 프로그램에 의해서 처리된다. 따라서, 이 .data 섹션 초기화 프로그램이 동작하기 전에 이들 레지스터를 먼저 초기화하여 주지 않으면 .data 섹션에는 초기화 데이터가 복사되지 않으며, 따라서 이 데이터를 사용하는 프로그램은 올바르게 동작하지 않는다. 이를 해결하는 방법에 대하여는 나중에 제6장에서 설명하기로 한다.

그러나, .bss 섹션이나 .noinit 섹션은 이와 같이 프로그램에 의하여 초기화되는 과정이 필요 없기 때문에 이를 외부 데이터 메모리에 할당하여도 아무 문제없이 프로그램이 잘 동작한다. 다시 말하면 특별한 경우가 아니라면 프로그램을 작성할 때 .data 섹션과 스택은 내부 SRAM의 디폴트 위치에서 그냥 사용하고, 외부 데이터 메모리로 옮겨서 사용하는 것은 .bss 섹션이나 .noinit 섹션 및 힙에만 적용하는 것이 쉽고 바람직하다는 것이다.

### (3) .bss 섹션

.bss 섹션은 초기 값이 정의되지 않은 전역 변수(global variable)나 정적 변수(static variable)를 포함하는 섹션으로서 이는 반드시 SRAM 영역에 할당된다.

.bss 섹션은 기본적으로는 내부 SRAM에서 .data 섹션의 뒤에 이어서 할당되지만, 사용자는 `-WI, -Tbss=address` 옵션을 사용하여 링커에게 .bss 섹션의 시작변지를 임의로 지정할 수 있다. 그런데, 여기서 `address`를 지정할 때는 실제의 SRAM 물리변지에 `0x800000`을 더하여 나타내야 한다. 즉, .bss 섹션을 시작할 실제의 SRAM 번지가 `0x8000`이라면 여기에 SRAM을 나타내는 옵셋 `0x800000`을 더하여 `-WI, -Tbss=0x808000`으로 옵션을 지정해야 한다는 것이다.

그러나, 이와 같이 사용자가 .bss 섹션을 외부 SRAM으로 옮겨 할당하면 그 뒤에 이어지는 나머지 섹션들도 모두 자동으로 따라서 움직이는데 유의해야 한다. 마찬가지로 .bss 섹션은 디폴트로 항상 .data 섹션의 뒤에 이어지므로 .bss 섹션의 시작변지를 따로 지정하지 않고 .data 섹션의 시작변지를 다른 값으로 지정하면 이에 따라 자동적으로 .bss 섹션의 위치도 함께 움직인다.

### (4) .noinit 섹션

.noinit 섹션은 .bss 섹션의 일부분으로서 역시 초기 값이 정의되지 않은 전역 변수나 정적 변수를 포함하는 섹션으로서 이는 반드시 SRAM 영역에 할당된다.

.noinit 섹션에는 사용자가 프로그램에서 다음과 같이 변수를 이 섹션에 할당한 것들만 포함된다.

```
unsigned char data1 __attribute__ ((section ("noinit")));
```

.noinit 섹션은 기본적으로는 내부 SRAM에서 .bss 섹션의 뒤에 이어서 할당되지만, 사용자는 `-WI, --section-start=.noinit=address` 옵션을 사용하여 링커에게 .noinit 섹션의 시작변지를 임의로 지정할 수 있다. 그런데, 여기서 `address`를 지정할 때는 실제의 SRAM 물리변지에 `0x800000`을 더하여 나타내야 한다. 즉, .noinit 섹션을 시작할 실제의 SRAM 번지가 `0x8000`이라면 여기에 SRAM을 나타내는 옵셋 `0x800000`을 더하여 `-WI, --section-start=.noinit=0x808000`으로 옵션을 지정해야 한다는 것이다.

.bss 섹션의 변수들은 프로그램이 스트트업할 때 0으로 값이 초기화되지만, .noinit 섹션에 할당된 변수들은 0으로 초기화되지 않는다.

이제 이상에서 설명한 섹션들이 실제로 프로그램에서 그와 같이 사용되는지를 예제를 통하여 확인해 보기로 한다. 여러 가지의 섹션들이 사용되는 예를 골고루 확인할 수 있도록

작성된 C언어 프로그램을 <파일 2>에 보였다.

<파일 2> OK-128 키트용 예제 프로그램 test1.c의 소스

```
/*
 * =====
 * Internal & External SRAM Usage 1
 * =====
 * Designed and programmed by Duck-Yong Yoon in 2004. */
#include <avr/io.h>
#include "c:\AvrEdit\0k128c\0k128.h"

unsigned char array1[16] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}; // .data section
unsigned char array2[16]; // .bss section
unsigned char array3[16] __attribute__ ((section (".noinit"))); // .noinit section

int main(void)
{ unsigned char i;

MCU_initialize(); // initialize MCU
Delay_ms(50); // wait for system stabilization
LCD_initialize(); // initialize text LCD module

LCD_string(0x80, "array2[0] = 00 "); // display screen
LCD_string(0xC0, "array3[0] = 00 ");

while(1)
{ for(i=0; i <= 15; i++)
{ array2[i] = array1[i]; // display array2
LCD_command(0x87);
LCD_data(i/10 + '0');
LCD_data(i%10 + '0');
LCD_command(0x8D);
LCD_data(array2[i]/10 + '0');
LCD_data(array2[i]%10 + '0');
array3[i] = array1[i] * 2; // display array3
LCD_command(0xC7);
LCD_data(i/10 + '0');
LCD_data(i%10 + '0');
LCD_command(0xCD);
LCD_data(array3[i]/10 + '0');
LCD_data(array3[i]%10 + '0');
Beep();
Delay_ms(3000);
}
}
}
```

이 소스 프로그램을 AvrEdit 3.6에서 모든 데이터 섹션이 내부 SRAM에 할당되도록 디폴트 조건으로 컴파일한 경우의 메모리 맵 파일을 보면 <파일 3>과 같다. 컴파일되고 링크된 후의 메모리 사용 내역은 항상 메모리 맵 파일을 보면 자세하게 알 수 있다.

---

<파일 3> 모든 데이터 섹션들을 내부 SRAM에 할당한 경우의 메모리 맵 파일 test1.map



.

#### Memory Configuration

Name	Origin	Length	Attributes
text	0x00000000	0x00020000	xr
data	0x00800060	0x0000ffa0	rw !x
eeprom	0x00810000	0x00010000	rw !x
*default*	0x00000000	0xffffffff	

#### Linker script and memory map

Address of section .data set to 0x800100

LOAD C:\AVREDIT\WINAVR\BIN\..\lib\gcc-libavr\3.3.1\..\..\..\avr\lib\avr5\crtm128.o

LOAD test1.o

LOAD C:\AVREDIT\WINAVR\BIN\..\lib\gcc-libavr\3.3.1\..\..\..\avr\lib\avr5\libm.a

LOAD C:\AVREDIT\WINAVR\BIN\..\lib\gcc-libavr\3.3.1\avr5\libgcc.a

LOAD C:\AVREDIT\WINAVR\BIN\..\lib\gcc-libavr\3.3.1\avr5\libc.a

LOAD C:\AVREDIT\WINAVR\BIN\..\lib\gcc-libavr\3.3.1\avr5\libgcc.a

.

.

.text 0x00000000 0x33c

.

.

.text	0x0000000ca	0x25a test1.o
	0x0000000ca	MCU initialize
	0x000001e2	Kev input
	0x00000190	LCD string
	0x000001b2	LCD initialize
	0x0000014a	Beep
	0x0000010c	Delay_ms
	0x0000022a	main
	0x00000174	LCD data
	0x000000f2	Delay_us
	0x00000158	LCD command
	0x00000324	. = ALIGN (0x2)

.

.

0x0000033c \_etext = .

.data	0x00800100	0x32 load address 0x0000033c
	0x00800100	PROVIDE (_data_start, .)

\*(.data)

.data	0x00800100	0x32 test1.o
	0x00800100	array1

\*(.gnu.linkonce.d\*)

0x00800132	. = ALIGN (0x2)
0x00800132	edata = .
0x00800132	PROVIDE (_data_end, .)

.bss	0x00800132	0x10
	0x00800132	PROVIDE (_bss_start, .)

\*(.bss)  
\*(COMMON)

```

COMMON      0x00800132    0x10 test1.o
            0x0 (size before relaxing)
0x00800132
0x00800142
0x0000033c
0x0000036e

.array2
PROVIDE ( .bss end. .)
data load start = LOADADDR (.data)
__data_load_end = (__data_load_start + SIZEOF (.data))

.noinit     0x00800142    0x10
            0x00800142    PROVIDE ( __noinit_start, .)

*(.noinit*)
.noinit     0x00800142    0x10 test1.o
            0x00800142    .array3
            0x00800152    PROVIDE ( __noinit_end, .)
            0x00800152    end = .
            0x00800152    PROVIDE ( __heap_start, .)

.eeprom     0x00810000    0x0 load address 0x0000036e
*(.eeprom*)
            0x00810000    __eeprom_end = .

.
.
.
```

## Cross Reference Table

Symbol	File
Beep	test1.o
Delay_ms	test1.o
Delay_us	test1.o
Key_input	test1.o
LCD_command	test1.o
LCD_data	test1.o
LCD_initialize	test1.o
LCD_string	test1.o
MCU_initialize	test1.o
.	
.	
.	



그러나, 이 소스 프로그램을 AvrEdit 3.6에서 컴파일할 때 <그림 20.5>에 보인 바와 같이 링커 옵션을 -Wl, -Tbss=0x808000, --section-start=.noinit=0x809000으로 설정하여 .bss 섹션과 .noinit 섹션이 외부의 확장 SRAM에 할당되도록 처리하였을 경우의 메모리 맵 파일을 보면 <파일 4>와 같다.

<파일 4> .bss와 .noinit 섹션을 외부 SRAM에 할당한 경우의 메모리 맵 파일 test1.map

```

◆-----.
.
.

Memory Configuration

Name        Origin      Length       Attributes
text        0x00000000  0x00020000  xr
```

data	0x000800060	0x0000fffffa0	rw !x
eeprom	0x000810000	0x000100000	rw !x
*default*	0x000000000	0xffffffffffff	

## Linker script and memory map

```

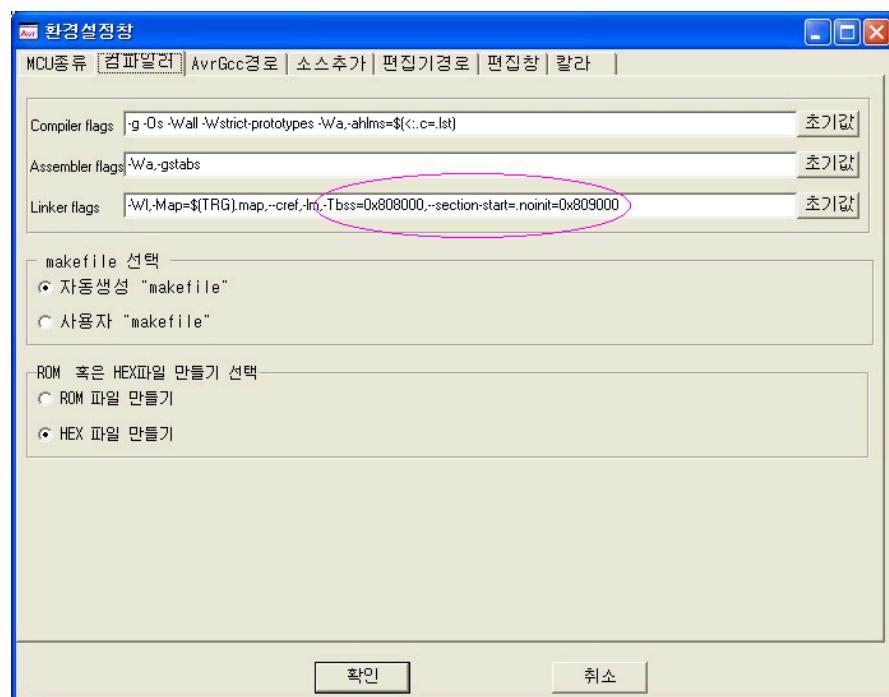
*(.noinit*)
.noinit      0x00809010      0x10 test1.o
              0x00809010
              0x00809020      PROVIDE ( _noinit_end, . )
              0x00809020      end = .
              0x00809020      PROVIDE ( _heap_start, . )

.eeprom       0x00810000      0x0 load address 0x0000036e
*(.eeprom*)
              0x00810000      __eprom_end = .

.
.
.
```

## Cross Reference Table

Symbol	File
Beep	test1.o
Delay_ms	test1.o
Delay_us	test1.o
Key_input	test1.o
LCD_command	test1.o
LCD_data	test1.o
LCD_initialize	test1.o
LCD_string	test1.o
MCU_initialize	test1.o
.	
.	



&lt;그림 20.5&gt; AvrEdit 3.6에서 링커 옵션을 설정하는 화면

### 3. 소스 프로그램에서 포인터를 이용하여 외부 데이터 메모리를 사용하는 방법

외부 데이터 메모리에 변수를 할당하고 이를 액세스하는 가장 쉬운 방법은 <파일 5>의 예제에서 보인 바와 같이 포인터를 사용하는 것이다. 즉, 포인터를 사용하여 액세스하고자 하는 외부 메모리의 번지를

```
#define ext_data1 (*(volatile unsigned char *)0x8000)
```

과 같이 정의하고, 이 번지 ext\_data1을 마치 메모리 변수인 것처럼 사용하는 방법이다. 이 때 링커에는 어떤 옵션도 따로 지정하여 주는 것이 없다.

이 방법은 외부 데이터 메모리 번지에도 물론 사용할 수 있지만, 외부에 추가로 확장한 I/O 번지에 더 많이 사용한다. <파일 5>에서도 I/O 번지로 사용한 예를 보였다.

#### <파일 5> OK-128 키트용 예제 프로그램 test2.c의 소스

```
/*
 * =====
 * Internal & External SRAM Usage 2
 * =====
 * Designed and programmed by Duck-Yong Yoon in 2004. */
#include <avr/io.h>
#include "c:\AvrEdit\0k128c\0k128.h"

#define ext_data1      (*(volatile unsigned char *)0x8000) // external memory
#define ext_data2      (*(volatile unsigned int *)0x9000)

#define LED7_data      (*(volatile unsigned char *)0x2000) // external I/O
#define LED7_digit     (*(volatile unsigned char *)0x2200)

void LCD_2hex(unsigned char number) /* display 2-digit hex number */
{
    unsigned char i;

    i = (number >> 4) & 0x0F;                                // 16^1
    if(i <= 9) LCD_data(i + '0');
    else        LCD_data(i - 10 + 'A');

    i = number & 0x0F;                                     // 16^0
    if(i <= 9) LCD_data(i + '0');
    else        LCD_data(i - 10 + 'A');
}

void LCD_4hex(unsigned int number) /* display 4-digit hex number */
{
    unsigned int i;

    i = number >> 12;                                    // 16^3
    if(i <= 9) LCD_data(i + '0');
    else        LCD_data(i - 10 + 'A');

    i = (number >> 8) & 0x000F;                           // 16^2
    if(i <= 9) LCD_data(i + '0');
    else        LCD_data(i - 10 + 'A');
```

```

i = (number >> 4) & 0x000F;           // 16^1
if(i <= 9) LCD_data(i + '0');
else      LCD_data(i - 10 + 'A');

i = number & 0x000F;           // 16^0
if(i <= 9) LCD_data(i + '0');
else      LCD_data(i - 10 + 'A');

}

int main(void)
{ unsigned char i, x;
  unsigned int j, y;

MCU_initialize();           // initialize MCU
Delay_ms(50);              // wait for system stabilization
LCD_initialize();           // initialize text LCD module

LCD_string(0x80, " 0x8000 = 0x00  ");
LCD_string(0xC0, " 0x9000 = 0x0000");

while(1)
{ for(i=0, i=0; i<=0xFF00; i++, i+=0x10)
  { LCD_command(0x8C);           // display 8-bit i
    ext_data1 = i;
    x = ext_data1;
    LCD_2hex(x);
    LCD_command(0xCC);           // display 16-bit j
    ext_data2 = i;
    y = ext_data2;
    LCD_4hex(y);
    Delay_ms(1000);
    LED7_data = 0xF2;           // display 7-segment LED
    LED7_digit = 0xFF;
    Delay_ms(500);
    LED7_digit = 0x00;
  }
  Beep();
  Delay_ms(3000);
}
}

```

그러나, 위의 <파일 5>에서 사용한 방법은 #define으로 하나의 메모리 번지만을 정의하여 사용할 수 있으므로 많은 외부 메모리 번지를 사용하려는 경우에는 이를 번지를 #define으로 하나씩 일일이 정의해야 하기 때문에 상당히 번거롭게 된다. 이러한 이유 때문에 이 방법은 외부 I/O 번지에 더 많이 사용되는 것이다.

이를 해결하는 방법으로는 #define으로 하나의 포인터를 정의하여 두고, 이 기준 포인터에 오프셋 값을 더하여 메모리 번지를 지정하며 필요할 경우에는 이 오프셋 값을 변경하여 다시 기준 포인터에 더하는 것을 생각할 수 있다. 이를 <파일 6>에 예제로 보였다.

## &lt;파일 6&gt; OK-128 키트용 예제 프로그램 test3.c의 소스

```

/*
 * ===== Internal & External SRAM Usage 3 =====
 * ===== Designed and programmed by Duck-Yong Yoon in 2004. =====
 */

#include <avr/io.h>
#include "c:\AvrEdit\0k128c\0k128.h"

#define ext_data1      ((volatile unsigned char *)0x8000) // external memory
#define ext_data2      ((volatile unsigned int *)0x9000)

void LCD_2hex(unsigned char number)      /* display 2-digit hex number */
{ unsigned char i;

    (o) 함수는 test2.c에서와 동일함.

}

void LCD_4hex(unsigned int number)      /* display 4-digit hex number */
{ unsigned int i;

    (o) 함수는 test2.c에서와 동일함.

}

int main(void)
{ unsigned char i, x;
  unsigned int j, y, offset;

  MCU_initialize();                      // initialize MCU
  Delay_ms(50);                         // wait for system stabilization
  LCD_initialize();                      // initialize text LCD module

  LCD_string(0x80, " 0x8000 = 0x00  ");
  LCD_string(0xC0, " 0x9000 = 0x0000");

  while(1)
  { for(i=0, offset=0; i<=0xFF00; i++, offset+=0x10)
    { LCD_command(0x83);                // display 8-bit i
      LCD_4hex(0x8000 + offset);
      LCD_command(0x8C);
      *(ext_data1 + offset) = i;
      x = *(ext_data1 + offset);
      LCD_2hex(x);
      LCD_command(0xC3);                // display 16-bit j
      LCD_4hex(0x9000 + offset*2);
      LCD_command(0xCC);
      *(ext_data2 + offset*2) = j;
      y = *(ext_data2 + offset*2);
      LCD_4hex(y);
      Delay_ms(1000);
    }
    Beep();
    Delay_ms(3000);
  }
}

```

그러나, 이 방법은 아래 <파일 7>에 보인 바와 같이 좀더 간편하게 사용할 수도 있으며, 이를 다시 포인터와 배열을 결합시켜 <파일 8>처럼 사용할 수도 있다.

<파일 7> OK-128 키트용 예제 프로그램 test4.c의 소스

```
/*
 * =====
 * Internal & External SRAM Usage 4
 * =====
 * Designed and programmed by Duck-Yong Yoon in 2004. */

#include <avr/io.h>
#include "c:\AvrEdit\0k128c\0k128.h"

#define read_XRAM(address)      (*(volatile unsigned char *) (0x8000+address))
#define write_XRAM(address,value) ((*(volatile unsigned char *) (0x8000+address))=value)

void LCD_2hex(unsigned char number)      /* display 2-digit hex number */
{ unsigned char i;

    (이 함수는 test2.c에서와 동일함.)

}

void LCD_4hex(unsigned int number)      /* display 4-digit hex number */
{ unsigned int i;

    (이 함수는 test2.c에서와 동일함.)

}

int main(void)
{ unsigned char i, x, offset;

    MCU_initialize();                      // initialize MCU
    Delay_ms(50);                         // wait for system stabilization
    LCD_initialize();                      // initialize text LCD module

    LCD_string(0x80, " 0x8000 = 0x00  ");           // display screen
    LCD_string(0xC0, "          ");               

    while(1)
    { for(i=0,offset=0; i<=0xF0; i++,offset++)
        { LCD_command(0x83);
          LCD_4hex(0x8000 + offset);
          LCD_command(0x8C);
          write_XRAM(offset, i);                  // write external memory
          x = read_XRAM(offset);                 // read external memory
          LCD_2hex(x);
          Delay_ms(1000);
        }
        Beep();
        Delay_ms(3000);
    }
}
```

## &lt;파일 8&gt; OK-128 키트용 예제 프로그램 test5.c의 소스

```
/*
 * ===== Internal & External SRAM Usage 5 =====
 * ===== Designed and programmed by Duck-Yong Yoon in 2004. =====
 */
#include <avr/io.h>
#include "c:\AvrEdit\0k128c\0k128.h"

#define read_XRAM(address) (((volatile unsigned char *)0x8000)[address])
#define write_XRAM(address,value) (((volatile unsigned char *)0x8000)[address]=value)

(이하는 test4.c와 동일함.)
```

이상에서 설명한 바와 같이 포인터를 사용하여 외부 데이터 메모리를 액세스하는 방법은 모든 것을 소스 프로그램에서 처리하고 링커에는 어떤 특별한 옵션도 주지 않는다. 이는 다시 말하면 소스에서 사용하는 외부 데이터 메모리 영역을 링커가 전혀 인식하지 않는다는 의미이다. 그러나, 바로 이러한 이유 때문에 포인터에 의한 방법과 외부 메모리에 데이터 섹션을 할당하는 방법을 혼용하는 경우에는 주의가 필요하다. 즉, 위의 <파일 5>~<파일 8>에서 외부 메모리에 .noinit 섹션을 할당하기 위하여

```
unsigned char array4[256] __attribute__ ((section (".noinit")));
```

라고 정의하는 부분을 소스에 추가하고 링커 옵션을 -Wl, --section-start=.noinit=0x808000으로 설정하였다면 메모리 변수 array4[256]이 외부 데이터 메모리의 0x8000~0x80FF 번지에 들어가게 되므로 #define으로 정의하여 사용하는 메모리 번지와 중복되는 결과가 된다. 따라서, 이를 피하려면 링커 옵션을 -Wl, --section-start=.noinit=0x80C000과 같이 설정하여 사용하는 메모리 영역이 서로 중복되지 않도록 해야 한다. 즉, **포인터를 사용하는 방식에서는 외부 데이터 메모리의 번지를 링커가 관리하지 않으면 이는 프로그램을 작성하는 사용자의 몫**이라는 것이다.

#### 4. 링커 옵션으로 .bss 또는 .noinit 섹션을 외부 데이터 메모리에 할당하는 방법

이것은 메모리 변수를 특정한 번지에 정의하지 않고 임의의 데이터 메모리 번지에 할당되도록 하는 방법이다. 프로그래머는 다만 초기화된 전역 변수는 .data 섹션에 저장되고, 초기화되지 않은 전역 변수는 .bss 섹션에 저장된다는 것을 알면 된다. 그러나, 특별히 초기화되지 않은 변수를 .bss 섹션이 아니라 .noinit 섹션에 저장되도록 지정할 수도 있다.

사용자의 소스 프로그램에서는 이와 같이 데이터 변수의 종류에 따라 이것들이 어떤 섹션에 저장되는지만 알면 되며, 이를 메모리 섹션이 실제 외부 데이터 메모리의 어느 번지에 할당되는지는 나중에 링커 옵션으로 지정한다. 아래의 <파일 9>는 초기화되지 않은 2개의 배열을 각각 .bss 섹션과 .noinit 섹션에 1개씩 정의하고 이것들을 외부 데이터 메모리에 할당하여 사용하는 예제 프로그램이다. 이를 번역할 때 링커 옵션은 -Wl, -Tbss=0x808000, --section-start=.noinit=0x809000으로 설정하기 바란다.

&lt;파일 9&gt; OK-128 키트용 예제 프로그램 test6.c의 소스

```
/*
 * ===== Internal & External SRAM Usage 6 =====
 * Designed and programmed by Duck-Yong Yoon in 2004. */
#include <avr/io.h>
#include "c:\AvrEdit\Ok128c\Ok128.h"

unsigned char array1[100]; // .bss section
unsigned char array2[100] __attribute__ ((section (".noinit"))); // .noinit section

void LCD_2d(unsigned int number) /* display 2-digit decimal number */
{
    LCD data(number / 10 + '0'); // 10^1
    LCD_data(number % 10 + '0'); // 10^0
}

int main(void)
{
    unsigned char i;

    MCU initialize(); // initialize MCU
    Delay_ms(50); // wait for system stabilization
    LCD_initialize(); // initialize text LCD module

    LCD_string(0x80, "array1[00] = 00 "); // display screen
    LCD_string(0xC0, "array2[00] = 00 ");

    for(i=0; i<=99; i++) // initialize array
    {
        array1[i] = i;
        array2[i] = i;
    }

    while(1)
    {
        for(i=0; i<=99; i++)
        {
            LCD command(0x87); // display array1
            LCD_2d(i);
            LCD command(0x8D);
            LCD_2d(array1[i]);
            LCD command(0xC7); // display array2
            LCD_2d(array2[i]);
            Delay_ms(1000);
        }
    }
}
```

```

    Beep();
    Delay_ms(3000);
}

```

초기화된 전역변수나 배열은 .data 섹션에 저장되는데 이는 작은 용량의 내부 SRAM에 할당되므로 많은 변수나 큰 배열을 처리하는데는 부적합하다. 이와 같은 경우에는 소스 프로그램에서 이 배열을 초기화되지 않은 배열로 정의하여 .bss 섹션이나 .noinit 섹션에 저장되도록 하고 <파일 9>에서 한 것처럼 나중에 프로그램으로 초기값을 부여하는 방법으로 처리하면 된다.

## 5. 링커 스크립트 파일로 .bss 또는 .noinit 섹션을 외부 데이터 메모리에 할당하는 방법

이것은 링커가 오브젝트 파일들과 라이브러리 파일들을 링크할 때 기본적으로 참조하는 링커 스크립트 파일을 수정하여 .bss 섹션이나 .noinit 섹션을 사용자가 원하는 외부 데이터 메모리에 할당하도록 설정을 변경하는 방법이다. 그러면 사용자는 프로그램을 작성할 때 소스 프로그램에서 변수들이 .bss 섹션이나 .noinit 섹션에 포함되도록 처리하여 주기만 하면 되며 링커 옵션에는 별다른 사항을 지정할 필요가 없다.

링커가 링크를 수행할 때 기본적으로 참조하는 링커 스크립트 파일은 <파일 1>에서 보였던 WinAvr\avr\lib\ldscripts\avr5.x 파일이다. 이를 수정하면 링커는 이에 따라 링크 작업을 수행한다. 그러나, 이 원본 파일을 수정하는 것은 나중에 이를 원상태로 되돌릴 때 어려움을 야기할 수 있으므로 여기서는 이 파일을 avr5xram.x로 복사해서 수정하여 사용하기로 한다. 이렇게 수정한 링크 스크립트 파일을 <파일 10>에 보였는데, 링커가 이렇게 다른 이름의 링크 스크립트 파일을 참조하도록 하려면 링커 옵션에 **-WL, -Tldscripts/avr5xram.x**와 같이 링커 스크립트 파일명을 지정하여야 한다.

<파일 10> 링커 스크립트 파일 avr5.x를 수정하여 만든 avr5xram.x 파일

```

/*
Default linker script, for normal executables */
OUTPUT FORMAT("elf32-avr", "elf32-avr", "elf32-avr")
OUTPUT ARCH(avr:5)
MEMORY
{
    text    (rx)    : ORIGIN = 0, LENGTH = 128K
    data    (rw!x)   : ORIGIN = 0x800060, LENGTH = 0x7fa0
    xram    (rw!x)   : ORIGIN = 0x808000, LENGTH = 32K
    eeprom  (rw!x)   : ORIGIN = 0x810000, LENGTH = 64K
}
SECTIONS
{

```

```

/* Read-only sections, merged into text segment: */
.hash          : { *(.hash)           }
.dynsym        : { *(.dynsym)         }

.

/* Internal text space or external memory */
.text :
{
    *(.vectors)
    ctors start = . ;
    *(.ctors)
    ctors_end = . ;
}

.

.

} > text

.data   : AT (ADDR (.text) + SIZEOF (.text))
{
    PROVIDE (__data_start = .) ;
    *(.data)
    *(.gnu.linkonce.d*)
    . = ALIGN(2);
    edata = . ;
    PROVIDE (__data_end = .) ;
} > data

.bss /* SIZEOF(.data) + ADDR(.data) */ :
{
    PROVIDE (__bss_start = .) ;
    *(.bss)
    *(COMMON)
    PROVIDE (__bss_end = .) ;
} > xram
    data load start = LOADADDR(.data);
    __data_load_end = __data_load_start + SIZEOF(.data);

/* Global data not cleared after reset. */
.noinit /* SIZEOF(.bss) + ADDR(.bss) */ :
{
    PROVIDE ( noinit_start = .) ;
    *(.noinit*)
    PROVIDE ( noinit_end = .) ;
    end = . ;
    PROVIDE ( __heap_start = .) ;
} > xram

.eeprom : AT (ADDR (.text) + SIZEOF (.text) + SIZEOF (.data))
{
    *(.eeprom*)
    eeprom end = . ;
} > eeprom
.

.

}

```



이렇게 링커 스크립트 파일 avr5.x를 avr5xram.x로 복사하여 수정하고, 링커가 수정된 링크 스크립트 파일을 참조하도록 하려면 링커 옵션을 -Wl, -Tldscripts/avr5xram.x로 지정한 후에 <파일 9>의 예제 test6.c를 컴파일하여 생성된 test6.map 파일을 보면 <파일 11>과 같다. 여기서 보면 배열 array1[100]은 .bss 섹션에 저장되고 배열 array2[100]은 .noinit 섹션에 올바르게 저장되어 있으며, 이들 섹션은 수정된 링커 스크립트 파일에서 지정한 대로 외부 데이터 메모리에 차례로 할당되어 있는 것을 알 수 있다.

<파일 11> 링커 스크립트 파일에 의하여 .bss 및 .noinit 섹션을 외부 SRAM에 할당한 경우의 메모리 맵 파일 test6.map

```
◆
.
.
.

Memory Configuration

Name      Origin      Length      Attributes
text      0x00000000  0x00020000  xr
data      0x00800060  0x00007fa0  rw !x
xram      0x00808000  0x00008000  rw !x
eeprom    0x00810000  0x00010000  rw !x
*default* 0x00000000  0xffffffff

Linker script and memory map

Address of section .data set to 0x800100
LOAD C:\AVREDIT\WINAVR\BIN..\lib\gcc-lib\avr\3.3.1\..\..\..\..\avr\lib\avr5\crtm128.o
LOAD test6.o
LOAD C:\AVREDIT\WINAVR\BIN..\lib\gcc-lib\avr\3.3.1\..\..\..\..\avr\lib\avr5\libm.a
.
.

.text      0x00000000  0x33a
.
.

.text      0x000000ca  0x248 test6.o
0x0000022a  LCD 2d
0x000000ca  MCU initialize
0x000001e2  Key input
0x00000190  LCD string
0x000001b2  LCD initialize
0x0000014a  Beep
0x0000010c  Delay_ms
0x00000260  main
0x00000174  LCD data
0x000000f2  Delay_us
0x00000158  LCD command
0x00000312  . = ALIGN (0x2)
.
.

0x0000033a  _etext = .
```

```

.data          0x00800100    0x22 load address 0x0000033a
               0x00800100    PROVIDE (__data_start, .)
*(.data)
.data          0x00800100    0x22 test6.o
*(.gnu.linkonce.d*)
               0x00800122    . = ALIGN (0x2)
               0x00800122    edata =
               0x00800122    PROVIDE (__data_end, .)

.bss          0x00808000    0x64
               0x00808000    PROVIDE (__bss_start, .)
*(.bss)
*(COMMON)
COMMON        0x00808000    0x64 test6.o
               0x0 (size before relaxing)
               0x00808000    array1
               0x00808064    PROVIDE ( __bss_end, .)
               0x0000033a    data load start = LOADADDR (.data)
               0x0000035c    __data_load_end = (__data_load_start + SIZEOF (.data))

.noinit       0x00808064    0x64
               0x00808064    PROVIDE (__noinit_start, .)
*(.noinit*)
.noinit       0x00808064    0x64 test6.o
               0x00808064    array2
               0x008080c8    PROVIDE ( __noinit_end, .)
               0x008080c8    end =
               0x008080c8    PROVIDE ( __heap_start, .)

.eeprom        0x00810000    0x0 load address 0x0000035c
*(.eeprom*)
               0x00810000    __eprom_end = .

.
.
.
```

Cross Reference Table

Symbol	File
Beep	test1.o
Delay_ms	test1.o
Delay_us	test1.o
Kev_input	test1.o
LCD_2d	test1.o
LCD_command	test1.o
LCD_data	test1.o
LCD_initialize	test1.o
LCD_string	test1.o
MCU_initialize	test1.o
.	
.	
.	



## 6. 초기화가 필요한 .data 섹션을 외부 데이터 메모리에 올바르게 할당하는 방법

앞의 제2장에서도 설명하였듯이 데이터 섹션들 중에서 .data 섹션만은 링커 옵션을 사용하여도 외부 데이터 메모리에 올바르게 할당되지 않는다. 이는 AVR-GCC에서 .data 섹션을

초기화하는 과정 때문이다. .text 섹션 내부에서 .initN 섹션은 스타트업 코드를 정의하는 부분으로서 리셋 후에 main() 함수를 실행하기 전의 사용자 프로그램 준비 동작을 수행한다. 이것은 다음과 같이 0~9의 10단계 동작으로 이루어진다.

.init0 – \_\_init()와 관련이 있다. 만약 사용자가 \_\_init() 함수를 정의하면 리셋 후에 즉시 그 곳으로 점프된다.

.init1 – 사용되지 않는다. 사용자가 정의하여 사용할 수 있다.

.init2 – C언어 프로그램의 경우에만 사용된다. 스택을 초기화하는 기능과 관련이 있다.

.init3 – 사용되지 않는다. 사용자가 정의하여 사용할 수 있다.

.init4 – 플래시 메모리에서 SRAM으로 .data 섹션의 초기화 데이터를 복사하며, .bss 섹션을 할당하고 초기값을 0으로 클리어한다.

.init5 – 사용되지 않는다. 사용자가 정의하여 사용할 수 있다.

.init6 – C++언어 프로그램의 경우에만 사용된다.

.init7 – 사용되지 않는다. 사용자가 정의하여 사용할 수 있다.

.init8 – 사용되지 않는다. 사용자가 정의하여 사용할 수 있다.

.init9 – main()으로 점프한다.

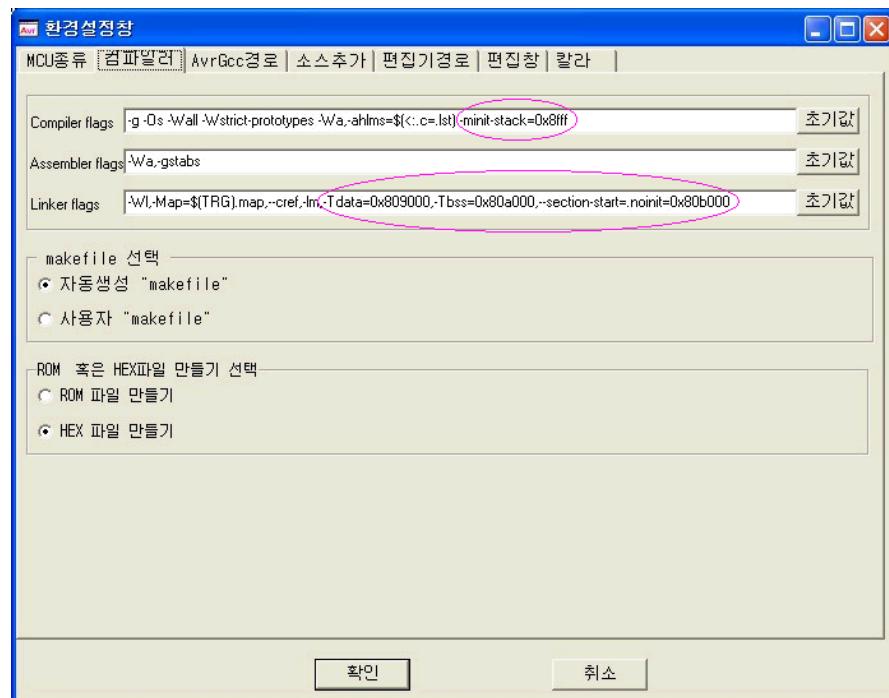
위에서 보듯이 .data 섹션이 초기화되는 것은 .init4의 단계인데, 이때는 아직 사용자 프로그램에서 외부 버스를 사용할 수 있도록 ATmega128의 MCUCR, XMCRA, XMCRA 레지스터를 초기화하기 전이다. 따라서, .data 섹션이 내부 SRAM을 사용하는 경우에는 아무 문제가 없지만, .data 섹션을 외부 데이터 메모리 영역에 할당하고자 하는 경우에는 정상적인 액세스가 수행될 수 없으므로 이 초기화 과정이 올바르게 처리되지 못한다.

따라서, 이 문제를 해결하는 한가지 방법은 사용자가 ATmega128의 MCUCR, XMCRA, XMCRA 레지스터를 초기화하는 루틴을 작성하여 .init4 보다 앞에 있는 단계에 이를 링크해주는 것이다. 예를 들어 이러한 루틴을 사용자 프로그램에서 함수 XBUS\_initialize()로 작성하여 .init1 단계에 링크하려면 아래와 같이 처리하면 된다.

```
void XBUS_initialize(void) __attribute__ ((naked)) __attribute__ ((section ("._init1")));
```

이와 같은 방법으로 .data 섹션을 외부 메모리에 할당하여 사용하는 예제 프로그램을 작성하여 보면 <파일 12>와 같다. 만약 .data 섹션을 0x9000 번지부터 할당하면 이에 이어서 .bss 섹션, .noinit 섹션, 힙 등도 모두 자동으로 외부 SRAM으로 이동되지만, 이 예제에서는 링커 옵션을 -Wl, -Tdata=0x809000, -Tbss=0x80a000, --section-start= .noinit=0x80b000으로 지정하여 각 데이터 섹션들의 위치를 별도로 할당하였다. AvrEdit 3.6에서 이렇게 옵션을 설

정하는 화면은 <그림 20.6>에 보였고, 컴파일한 후에 생성된 메모리 맵 파일은 <파일 13>과 같다.



<그림 20.6> AvrEdit 3.6에서 컴파일 옵션 및 링커 옵션을 설정하는 화면

<파일 12> OK-128 키트용 예제 프로그램 test7.c의 소스

```
/*
 * =====
 * Internal & External SRAM Usage 7
 * =====
 * Designed and programmed by Duck-Yong Yoon in 2004. */

#include <avr/io.h>
#include "c:\AvrEdit\0k128c\0k128.h"

unsigned char array1[16] = {0.1.2.3.4.5.6.7.8.9.10.11.12.13.14.15}; // .data section
unsigned char array2[16]; // .bss section
unsigned char array3[16] __attribute__ ((section (".noinit"))); // .noinit section

void XBUS_initialize(void) __attribute__ ((naked)) __attribute__ ((section (".init1")));

void XBUS_initialize(void) /* initialize external bus */
{
    MCUCR = 0x80; // enable external memory & I/O
    XMCRA = 0x44; // 0x1100-0x7FFF=1 wait. 0x8000-0xFFFF=0 wait
    XMCRB = 0x80; // enable bus keeper, use PC0-PC7 as address
}
```

```

void LCD_2d(unsigned char number)          /* display 2-digit decimal number */
{
    LCD data(number / 10 + '0');           // 10^1
    LCD_data(number % 10 + '0');           // 10^0
}

void LCD_4hex(unsigned int number)         /* display 4-digit hex number */
{ unsigned int i;

    i = number >> 12;                   // 16^3
    if(i <= 9) LCD data(i + '0');
    else      LCD_data(i - 10 + 'A');

    i = (number >> 8) & 0x000F;           // 16^2
    if(i <= 9) LCD data(i + '0');
    else      LCD_data(i - 10 + 'A');

    i = (number >> 4) & 0x000F;           // 16^1
    if(i <= 9) LCD data(i + '0');
    else      LCD_data(i - 10 + 'A');

    i = number & 0x000F;                 // 16^0
    if(i <= 9) LCD data(i + '0');
    else      LCD_data(i - 10 + 'A');
}

void Test_SP(void)                      /* test stack pointer */
{
    LCD string(0xC0, " SP = 0x0000    ");
    LCD command(0xC9);
    LCD 4hex(SP);
    Beep();
    Delay_ms(3000);
}

int main(void)
{ unsigned char i;

    MCU initialize();                  // initialize MCU
    Delay_ms(50);                     // wait for system stabilization
    LCD_initialize();                 // initialize text LCD module

    while(1)
    { LCD string(0x80, "array2[00] = 00 ");      // display variable in external memory
        LCD string(0xC0, "array3[00] = 00 ");
        for(i=0; i<15; i++)
            { array2[i] = array1[i];             // display array2
                LCD command(0x87);
                LCD 2d(i);
                LCD command(0x8D);
                LCD 2d(array2[i]);
                array3[i] = array1[i] * 2;        // display array3
                LCD command(0xC7);
                LCD 2d(i);
                LCD command(0xCD);
                LCD 2d(array3[i]);
                Delay_ms(1000);
            }
    }
}

```

```
        }
        Beep();
        Delay_ms(2000);

        LCD_string(0x80, " SP = 0x0000    ");
        // display satck pointer
        LCD_string(0xC0, "      ");
        LCD_command(0x89);
        LCD_4hex(SP);
        Beep();
        Delay_ms(3000);
        Test_SP();           // stack pointer in function
    }
}
```

<파일 13> 모든 데이터 섹션을 외부 SRAM에 할당한 경우의 메모리 맵 파일 test7.map

## Memory Configuration

Name	Origin	Length	Attributes
text	0x00000000	0x00020000	xr
data	0x00800060	0x0000ffa0	rw !x
eeprom	0x00810000	0x00010000	rw !x
*default*	0x00000000	0xffffffff	

## Linker script and memory map

```
Address of section .data set to 0x800100
LOAD C:\AVREDIT\WINAVR\BIN\..\lib\gcc-lib\avr\3.3.1\..\..\..\..\avr\lib\avr5\crtm128.o
LOAD test7.o
LOAD C:\AVREDIT\WINAVR\BIN\..\lib\qcc-lib\avr\3.3.1\..\..\..\..\avr\lib\avr5\libm.a
Address of section .data set to 0x809000
Address of section .bss set to 0x80a000
Address of section .noinit set to 0x80b000
LOAD C:\AVREDIT\WINAVR\BIN\..\lib\qcc-lib\avr\3.3.1\avr5\libqcc.a
LOAD C:\AVREDIT\WINAVR\BIN\..\lib\qcc-lib\avr\3.3.1\..\..\..\..\avr\lib\avr5\libc.a
LOAD C:\AVREDIT\WINAVR\BIN\..\lib\gcc-lib\avr\3.3.1\avr5\libgcc.a

.

.

.

.text          0x00000000          0x3ea
.

.

.

*(.init0)
*(.init1)
.init1        0x00000008c      0xe test7.o
               0x00000008c      XBUS_initialize
*(.init2)
.init2        0x00000009a      0xc C:\AVREDIT\WINAVR\BIN\.....\avr\lib\avr5\crtm128.o
*(.init3)
*(.init4)
.init4        0x0000000a6      0x1a C:\AVREDIT\WINAVR\BIN\.....\avr\lib\avr5\crtm128.o
               0x0000000a6      __do_copy_data
```



```
.eeprom      0x00810000    0x0 load address 0x0000043e
*(.eeprom*)  0x00810000    __eprom_end = .
```

## Cross Reference Table

Symbol	File
Beep	test7.o
Delay_ms	test7.o
Delay_us	test7.o
Kev_input	test7.o
LCD_2d	test7.o
LCD_4hex	test7.o
LCD_command	test7.o
LCD_data	test7.o
LCD_initialize	test7.o
LCD_string	test7.o
MCU_initialize	test7.o
Test_SP	test7.o
<b>XBUS_initialize</b>	test7.o
.	
.	
.	



또한, 위의 <파일 12> 예제에서는 스택을 외부 데이터 메모리에 할당하여 사용하는 방법도 포함하고 있다. 이 예제에서는 스택을 외부 메모리의 0x8000~0x8FFF 영역에 할당하여 사용하는 것으로 가정하였다.

스택을 사용자가 원하는 메모리 영역에 할당하여 사용하려면 스택 포인터 SP의 초기값을 지정하여야 한다. 이는 컴파일러 옵션에서 `-minit-stack=0x8fff`와 같은 방법으로 지정하거나, 링커 옵션에서 `-Wl, --defsym=__stack=0x808fff`처럼 지정하면 된다. 이와 같이 컴파일러와 링커에서 스택 포인터의 초기값은 `__stack` 심볼로 지정되는데, 사용자가 이를 별도로 지정하지 않으면 ATmega128의 경우 이것이 디폴트로 내부 SRAM의 마지막 번지인 0x10FF로 처리된다.

위의 예제에서는 <그림 20.6>에서도 보였듯이 컴파일러 옵션으로 스택 포인터를 지정하는 방법을 사용하였다. 스택을 0x8000~0x8FFF 영역으로 사용하기 위하여 스택 포인터 SP의 초기값으로 0x8FFF를 지정하였으므로 이 프로그램을 실행하면 처음에 표시되는 SP값은 0x8FFF이지만 `Test_SP()` 함수를 호출하여 그 내부에서 표시되는 SP값은 0x8FFD인 것을 볼 수 있다. C언어에서 함수를 호출하면 이 함수는 어셈블리 언어의 서브루틴에 해당하므로 스택으로 사용되는 메모리의 2개 번지에 16비트의 복귀주소(return address)를 저장하게 되고 따라서 스택 포인터가 2만큼 감소하기 때문이다.

## 7. 어셈블리 언어 프로그램의 경우

AVR에서 내부 SRAM이나 외부 확장 데이터 메모리를 어셈블리 언어에서 액세스하여 사용하는 방법은 C언어에 비하여 훨씬 더 쉽다. C언어가 사용자 지향적(user-oriented)인 언어인데 비하여 어셈블리 언어는 이보다 훨씬 기계 지향적(machine-oriented)인 언어이기 때문이다. 어셈블리 언어를 사용한 내부 SRAM 및 외부 데이터 메모리 액세스 예제를 <파일 14>에 보였다.

<파일 14> 어셈블리 언어를 사용한 내부 SRAM 및 외부 SRAM 사용 예제 test8.asm

```
-----[REDACTED]-----  

;  

; Internal & External SRAM Usage 8  

;  

; [REDACTED]  

;  

; Designed and programmed by Duck-Yong Yoon in 2004.  

;  

;-----  

; Include Header File  

;-----  

;.include "MEGA128.INC" ; include ATmega128 definition file  

;.include "OK128DEF.INC" ; include OK-128 I/O definition file  

;  

;-----  

; Define I/O & Data Memory Area  

;-----  

.eau LED7 DATA      = 0x2000    ; external I/O  

.equ LED7_DIGIT     = 0x2200  

;  

.equ XRAM          = 0x8000    ; external memory  

;  

.dseg  
    .org 0x0100        ; data segment  

DATA1: .bpte 256  
    .org 0x8000        ; start internal memory  

DATA2: .byte 256  
    .org 0x0200        ; start external memory  

;  

;=====  

; Main Program  

;=====  

;cseg  

    .org 0x0000  

LDI  AH, high(RAMEND)    ; initialize SP  

LDI  AL, low(RAMEND)  

OUT SPL, AH  

OUT SPL, AL  

;  

CALL INIT_OK128          ; initialize ATmega128 CPU & OK-128 kit  

CALL D50MS                ; wait for system stabilization  

CALL INIT_LCD              ; initialize text LCD  

;  

;-----  

; Test External I/O (0x2000/0x2200)
```

```

;-----
LOOP: CALL LCD HOME1           ; display screen
      CALL LCD STRING
      .db   " 7-Segment LED ", 0, 0
      CALL LCD HOME2
      CALL LCD STRING
      .db   " ", 0, 0

      LDI   AL. 0b11110010    ; output '3'
      STS   LED7 DATA, AL
      LDI   AL. 0b11111111    ; select all digits
      STS   LED7 DIGIT, AL
      CALL D2SEC
      LDI   AL. 0b00000000    ; turn off all digits
      STS   LED7_DIGIT, AL

;-----
; Test External SRAM (0x8000)
;-----
CALL LCD HOME1           ; display screen
CALL LCD STRING
.db   " External SRAM ", 0, 0
CALL LCD HOME2
CALL LCD STRING
.db   " 0x8000 = 0x00 ", 0, 0

LDI   CL. 0x00
LOOP1: LDI   LCD BUFFER. 0xCC    ; display value
      CALL LCD COMMAND
      STS   XRAM. CL          ; write external memory
      LDS   LCD BUFFER, XRAM   ; read external memory
      CALL LCD 2HEX
      CALL D1SEC
      INC   CL
      CPI   CL. 0x10
      BRNE  LOOP1
      CALL BEEP
      CALL D2SEC

;-----
; Test Internal SRAM (0x0100-0x01FF)
;-----
CALL LCD HOME1           ; display screen
CALL LCD STRING
.db   " Internal SRAM ", 0, 0
CALL LCD HOME2
CALL LCD STRING
.db   " 0x0100 = 0x00 ", 0, 0

LDI   CL. 0x00
LDI   XH. high(DATA1)
LDI   XL. low(DATA1)
LOOP2: LDI   LCD BUFFER. 0xC3    ; display address
      CALL LCD COMMAND
      MOV   AH. XH
      MOV   AL. XL
      CALL LCD 4HEX
      LDI   LCD BUFFER. 0xCC    ; display value
      CALL LCD COMMAND
      ST    X. CL              ; write internal memory
      LD    LCD_BUFFER, X+       ; read internal memory

```

```

CALL    LCD 2HEX
CALL    D1SEC
INC     CL
CPI    CL. 0x00
BRNE   LOOP2
CALL    BEEP
CALL    D2SEC

;-----;
;      Test External SRAM (0x8000-0x80FF)
;-----;

        CALL    LCD HOME1           ; display screen
        CALL    LCD STRING
        .db    " External SRAM ", 0, 0
        CALL    LCD HOME2
        CALL    LCD STRING
        .db    " 0x8000 = 0x00 ", 0, 0

        LDI    CL. 0x00
        LDI    XH. hiqh(DATA2)
        LDI    XL. low(DATA2)
LOOP3: LD1    LCD BUFFER. 0xC3       ; display address
        CALL   LCD COMMAND
        MOV    AH. XH
        MOV    AL. XL
        CALL   LCD 4HEX
        LDI    LCD BUFFER. 0xCC       ; display value
        CALL   LCD COMMAND
        ST     X. CL                ; write external memory
        LD     LCD BUFFER, X+         ; read external memory
        CALL   LCD 2HEX
        CALL   D1SEC
        INC    CL
        CPI    CL. 0x00
        BRNE   LOOP3
        CALL   BEEP
        CALL   D2SEC

        JMP    LOOP

;=====;
;      Include User Subroutine File
;=====;

.include "OK128SUB. INC"          ; include OK-128 subroutine file

```

우선 내부 SRAM과 외부 데이터 메모리의 번지를 가장 손쉽게 지정하는 방법은 예제에서 XRAM처럼 **.equ** 지시어를 사용하여 번지를 상수값으로 정의하는 것이다. 이 번지값은 나중에 **LDS** 또는 **STS** 명령에서 데이터 직접 번지 지정으로 사용된다. 이와 같이 **.equ** 지시어를 사용하여 메모리 번지를 정의하는 방법은 C언어에서 **#define**을 사용하는 방법과 매우 유사하며, 1개의 **.equ** 지시어로는 오직 1개의 번지만을 사용할 수 있으므로 많은 메모리 번지를 사용하려는 경우에는 이를 일일이 **.equ** 지시어로 정의해 주어야 하는 불편이 있다. 또한, 이는 링커에 의하여 데이터 메모리의 번지로 인식되지 않으므로 나중에 **.dseg** 지시어로 데

이터 세그먼트를 정의할 경우 메모리 번지가 중복될 수 있다는 점에 주의하여야 한다. 따라서, 이 방식은 데이터 메모리 번지에도 사용되지만 외부 I/O 번지를 지정하는데 더 적합하다고 할 수 있다. <파일 14>의 예제에서는 이를 I/O 번지 지정에 사용하여 7세그먼트 LED 를 점등하는 예를 보였다.

많은 데이터 메모리 변수를 사용하는 경우에는 **.dseg 지시어를 사용하여 이를 데이터 세그먼트의 안에 정의하는 것이 좋다.** 이는 C언어에서 .bss 섹션이나 .noinit 섹션에 변수를 정의하는 것에 비유될 수 있으며, 링커에 의하여 이를 메모리 번지가 인식되므로 많은 변수를 정의하면 이것들이 데이터 메모리 영역에 차례로 할당된다. 이 방법에서도 물론 각각의 변수를 **LDS 또는 STS 명령**으로 액세스할 수 있지만, <파일 14>에서 사용한 것처럼 변수가 데이터 테이블(배열)일 경우에는 **X, Y, Z 레지스터에 의하여 데이터 메모리 간접 지정을 사용하는 것이 편리하다.**

이 방법은 내부 SRAM이든 외부 데이터 메모리든 상관없이 원하는 시작번지를 **.dseg 세그먼트 안에서 .org 지시어로 지정하기만 하면 된다.** <파일 14>에서는 내부 SRAM의 시작번지인 0x0100 번지에서부터 256바이트의 테이블을 정의하고, 외부 SRAM의 시작번지인 0x8000 번지에서부터 256바이트의 테이블을 정의하는 예를 보였으며, 이 예제는 AVR Studio V4.10으로 정상으로 동작하는 것을 확인하였다. 그러나, AVR Studio V4.10에는 새로 매크로 어셈블러 V2.0이 추가되었는데 이것은 그 이전의 V1.x에 비하여 많은 사항들이 개선되고 달라졌다. 그래서, 이 예제를 어셈블할 때 Project→AVR Assembler Setup 메뉴에서 Use AVR Assembler 2.0 Beta 항목을 체크하고 어셈블하면 2개의 경고 메시지가 발생된다. 이는 내부 SRAM은 0x0100~0x10FF 번지에만 존재하는데 테이블 DATA2는 0x8000 번지부터 정의되었으므로 링커는 이것이 내부 SRAM 영역을 벗어났다고 판단하기 때문이다. 하지만, 메모리 맵 파일을 보면 어셈블러는 이런 경고 메시지에도 불구하고 이를 올바르게 처리하며, 실제로 이 프로그램을 실행하면 외부 데이터 메모리에 있는 테이블 DATA2의 액세스가 정상적으로 수행된다.

### 【 참고문헌 】

1. Doxygen, avr-libc Reference Manual 1.0, Sep. 22, 2004
2. 윤덕용, AVR 마스터 시리즈 ① - AVR ATmega128 마스터, Ohm사, 2004
3. 윤덕용, AVR 마스터 시리즈 ② - AVR ATmega162 마스터, Ohm사, 2004
4. 윤덕용, AVR 마스터 시리즈 ③ - AVR ATmega8515 마스터, Ohm사, 2004