Mode Privilege

2010.02.08

Processor mode와 Privilege level

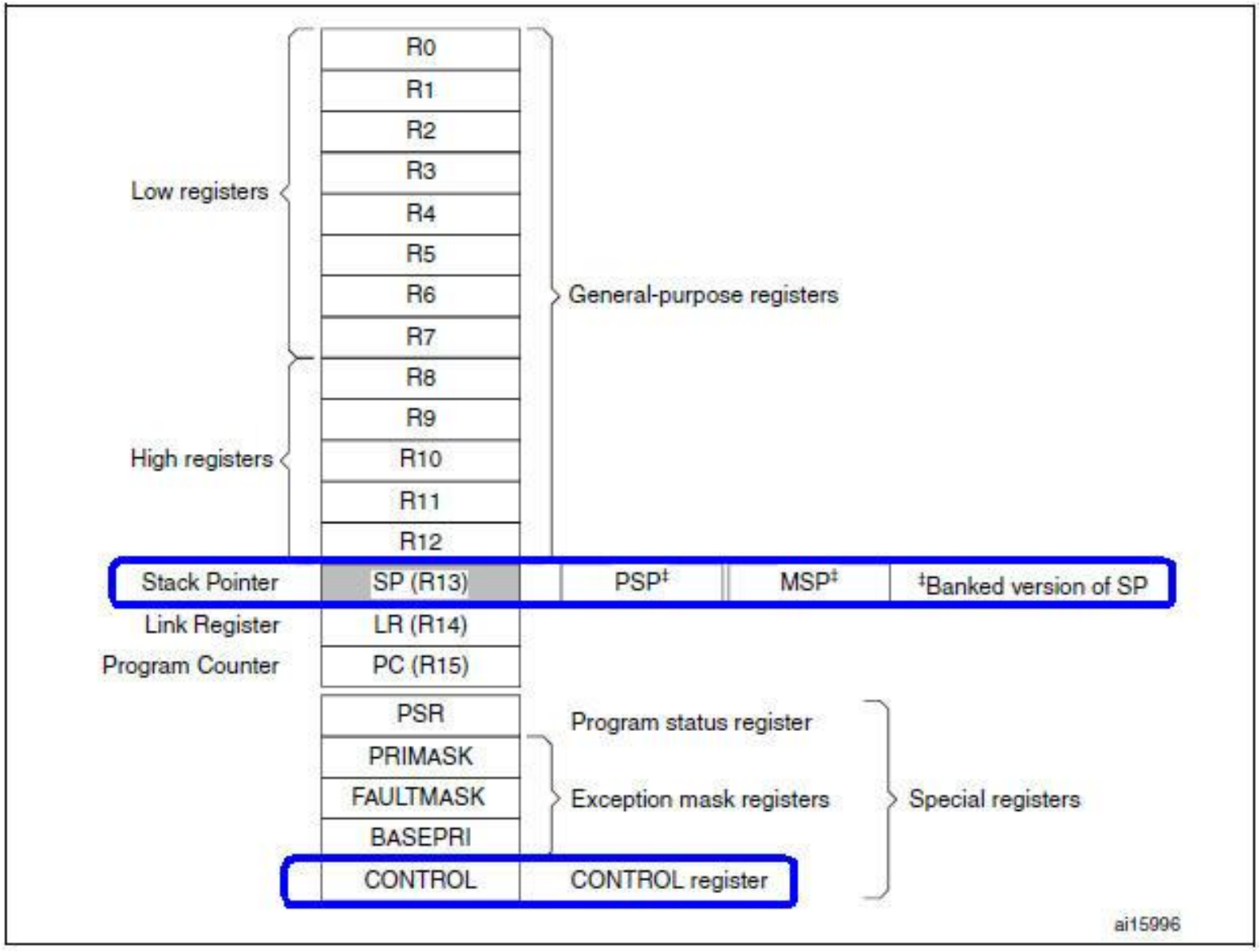
- Processor mode: **Thread mode와 Handler mode**
  - Thread mode: 프로세서가 보통의 Application Program을 실행. 처음 Reset이 되었을 때 프로세서는 Thread mode에 진입.
  - Handler mode: Exception handler를 실행. Exception을 처리할 때 프로세서는 Handler mode로 동작.
  - **Exception 처리를 끝내면 프로세서는 Thread mode로 되돌아 간다.**
- Privilege level: 특권 레벨과 사용자 레벨
  - **Privilege level과 Unprivileged level**
  - 이러한 구분은 기본적으로 보호를 목적: 중요한 부분에 침해가 발생할 수 있는 가능성을 미연에 방지. Critical한 영역으로의 메모리 접근을 보호
  - Privilege level에서는 프로세서가 할 수 있는 모든 일을 할 수 있다
  - 결국 Unprivileged level에서 할 수 없는 일이 무엇인가만 알면 된다.
- Unprivileged level 수행 소프트웨어 특징
  - MSR과 MRS 명령에 제한 (읽는 작업은 되지만 설정하는 작업 안됨)
  - CPS instruction(Change Processor State)을 사용할 수 없다
  - System Timer, NVIC, System Control Block을 접근할 수 없다.
  - memory나 peripherals에 대한 접근이 제한

Privilege level

- Handler mode에서는 모든 프로그램이 Privilege level로 동작
- **Thread mode에서는 Privilege level로 동작할수도 있고, Unprivileged level로 동작할 수도 있다.** 이에 대한 구분이 기록되어 있는 곳이 CONTROL register이다.
- CONTROL register를 Write하는 것은 오로지 Privilege level 소프트웨어만이 가능. 만약 Unprivileged software가 Privileged로 변환되기를 원할 경우 이를 변경시키기 위해서 SVC instruction을 사용해서 supervisor call 인터럽트를 발생시키고 변경 작업을 수행
  - **Unprivileged level로 동작 중인 프로그램은 CONTROL register를 Write해서 Privilege level로 변경시킬 방법이 없다.** 이를 위해 Exception을 발생시켜 CONTROL register를 변경시킬 수 있도록 만드는 것이다. 이 방법이 SVC를 이용하는 것이다.

Processor mode	Used to execute	Privilege level for software execution	Stack used
Thread	Applications	Privileged or unprivileged(1)	Main stack or process stack(1)
Handler	Exception handlers	Always privileged	Main stack

Main Stack과 Process Stack



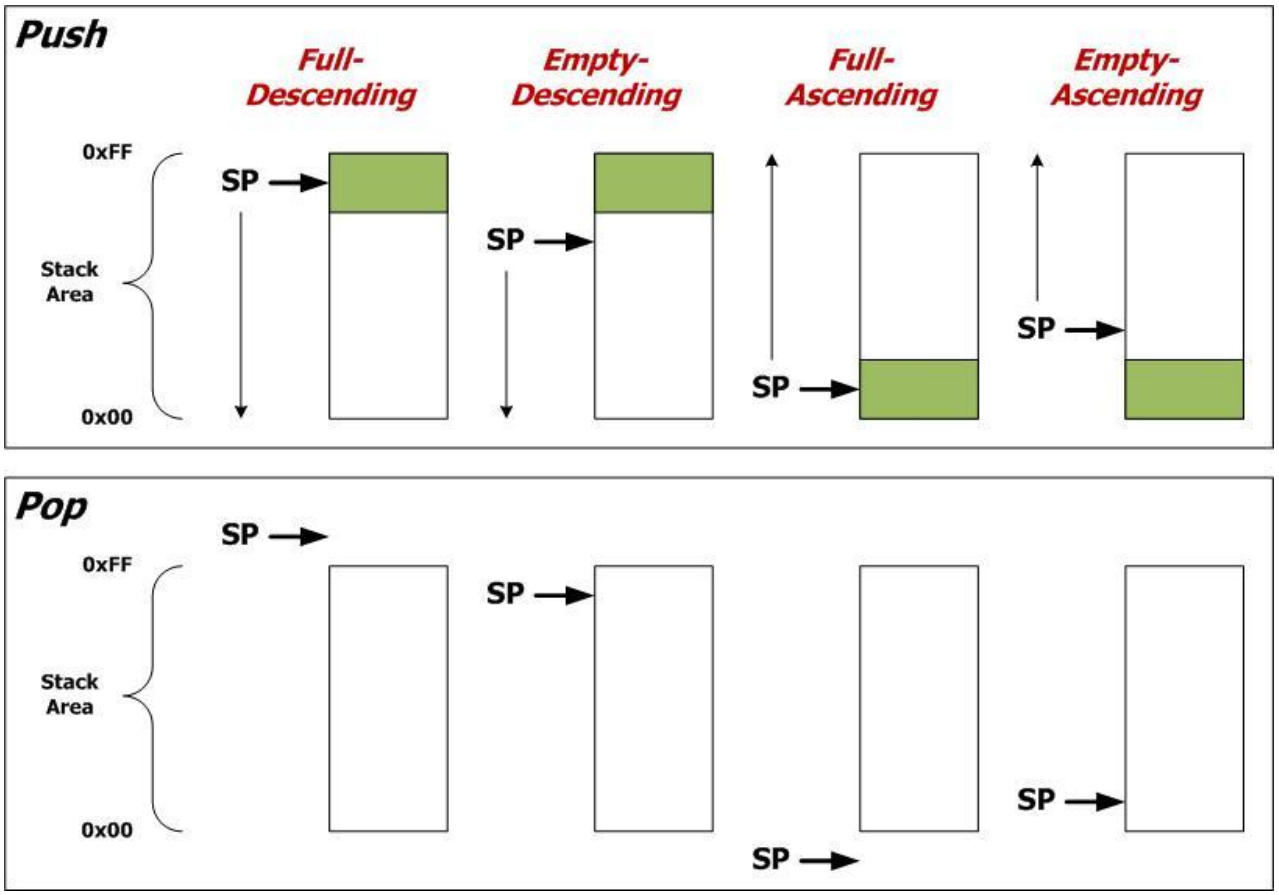
Banked Stack Pointer

- R13이라고 하면 어떤 레지스터가 접근되는 것일까?
  - Stack Pointer는 2개. **Main Stack과 Process Stack**
  - R13이라고 했을 때 접근되는 것은 Main Stack이거나 Process Stack을 가리키게 될 것이다. 이것을 구분하는 것은 CONTROL register.
- Banked의 의미
  - Banked는 한번에 하나의 부분만 볼수 있다는 것
  - 어떤 상태에서 접근할 수 있는 레지스터는 PSP거나 MSP인 것이지 둘 다 볼 수는 없다는 의미 (MRS나 MSR 명령을 사용해서 다른 것도 접근할 수 있다. 당연히 Privilege level이어야 함)
- **Banked 방식 사용 이유? 속도 때문**
  - R13 하나만을 이용한다고 가정하면 이 경우 뭔가 다른 동작을 수행하려고 하면 스텍에 모든 것을 저장하고 복구하는 과정을 수행하지 않을 수 없다.
  - 하지만 이것을 2개를 만들어 놓음으로써 PSP로 동작 중이다가 MSP를 사용해야 하는 시점이 되었을 때 레지스터 접근만을 변경해서 PSP로 동작 중이던 스텍의 상태를 전혀 건드리지 않고, 나중에 MSP 사용을 끝내고 다시 PSP로 돌아왔을 때 아무런 복구 작업 없이 수행이 가능해지게 되는 것이다.

Main Stack과 Process Stack

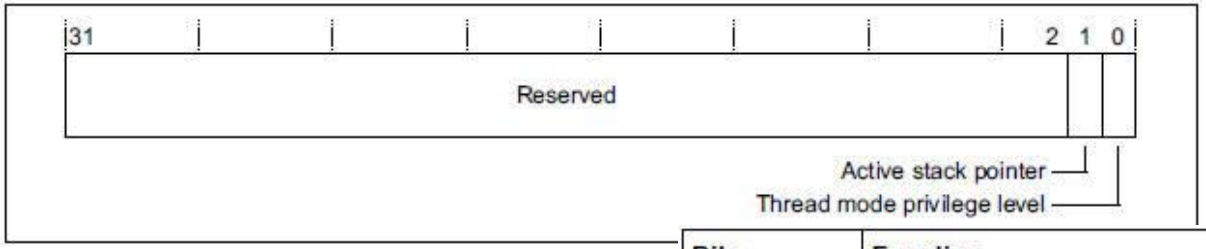
- 스텍은 Main Stack과 Process Stack이 있고, Privilege level과 Unprivileged level처럼 Thread mode에서는 둘 다 사용이 가능하고 Handler mode에서는 Main Stack만 사용
  - 이를 구분하는 것은 CONTROL register의 비트 1번
  - 이 값이 0이면 Main Stack Pointer (MSP)를 사용하는 것이고
  - 이 값이 1이면 Process Stack Pointer (PSP)를 사용하는 것
- 프로세서가 리셋 된 초기 상태에서 항상 Main Stack Pointer (MSP)를 사용하게 된다
  - 리셋 시 프로세서는 **address 0x00000000의 값을 MSP에 로딩**해서 사용
  - address 0x00000000의 값을 적절히 변화시키면 쉽게 Stack의 위치를 변화시킬 수 있다.
- Main Stack Pointer (MSP)는 Default Stack Pointer
  - SP_main으로 불린다. OS Kernel과 Exception Handler에 의해서 사용
  - Privilege level에서 동작하는 Application에 의해서 사용
- Process Stack Pointer (PSP)는 사용자 Application code에서 사용
  - SP_process라 불린다.

Stack Pointer - Full-Descending



- Stack Operation에 대해서 구분하는 것은 4가지
  - Full이냐 Empty이냐를 구분하는 것
  - Ascending이냐 Descending이냐를 구분하는 것

CONTROL register



Bits	Function
Bit 0	**TPL:** Thread mode privilege level Defines the Thread mode privilege level. 0: Privileged 1: Unprivileged.

Bits	Function
Bits 31:2	Reserved
Bit 1	**ASPSEL:** Active stack pointer selection Selects the current stack: 0: MSP is the current stack pointer 1: PSP is the current stack pointer. In Handler mode this bit reads as zero and ignores writes.

- Cortex-M3는 여러 개의 Special Register를 가지고 있다.
  - Program State Register (PSR)
  - Interrupt Mask Register (PRIMASK, FAULTMASK, BASEPRI)
  - Control Register (CONTROL)
- Special Register들은 MSR, MRS를 통해서만 접근이 될 수 있다.
  - MRS <reg>, <special_reg>의 형태로 Special Register를 읽어서 General Register에 저장
  - MSR <special_reg>, <reg>의 형태로 Special Register에 General Register의 값을 Write

CMSIS

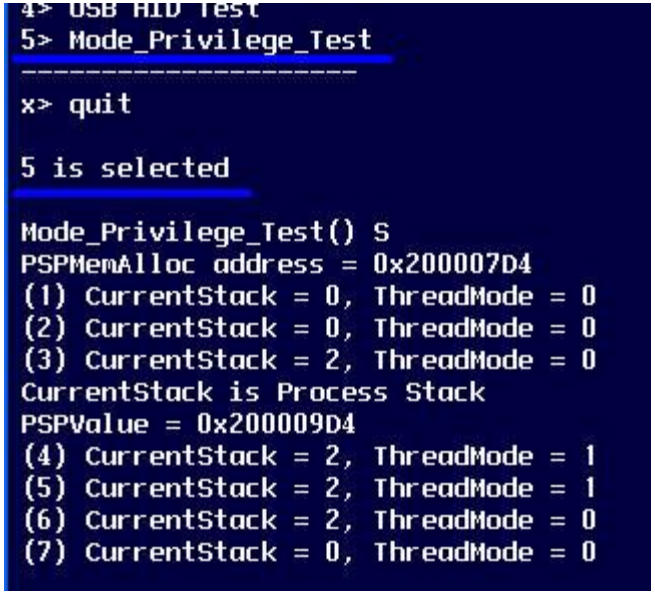
- Cortex microcontroller software interface standard (CMSIS)
- ARM사 제시 표준. 권고 사항. 반드시 따라야 하는 규정은 아니다
  - M3 Core을 사용하는 반도체 회사들 간의 Firmware 호환성을 높임
  - 현재 모든 제조사들은 모든 예제 코드들을 CMSIS에 맞춰서 제공
  - 사용자의 입장에서 다른 회사의 칩을 사용하더라도 코드를 재활용할수 있다.
  - Memory Map에서도 하드웨어적으로 고정. 호환성을 높이려는 시도
- CMSIS 사용 장점
  - peripheral registers에 대한 접근을 할때 공통의 방법을 사용
  - exception vectors의 정의에 대한 공통의 방법을 사용
  - core peripherals registers, core exception vectors 이름 동일하게 사용
  - RTOS kernels interface를 device와 독립적으로 구현할수 있게 된다.
- CMSIS에 compliant하게 개발된 소프트웨어 컴포넌트들을 여러 middleware 제조사로부터 받거나, 혹은 이미 개발된 코드들을 조합해서 쉽게 재 사용이 가능하기 때문에 소프트웨어의 개발에 있어서 매우 높은 효율성을 가져다 주게 된다.

Intrinsic functions – 표준화된 함수

Instruction	CMSIS intrinsic function
CPSIE I	void __enable_irq(void)
CPSID I	void __disable_irq(void)
CPSIE F	void __enable_fault_irq(void)
CPSID F	void __disable_fault_irq(void)
ISB	void __ISB(void)
DSB	void __DSB(void)
DMB	void __DMB(void)
REV	uint32_t __REV(uint32_t int value)
REV16	uint32_t __REV16(uint32_t int value)
REVSH	uint32_t __REVSH
RBIT	uint32_t __RBIT(ui
SEV	void __SEV(void)
WFE	void __WFE(void)
WFI	void __WFI(void)

Special register	Access	CMSIS function
PRIMASK	Read	uint32_t __get_PRIMASK (void)
	Write	void __set_PRIMASK (uint32_t value)
FAULTMASK	Read	uint32_t __get_FAULTMASK (void)
	Write	void __set_FAULTMASK (uint32_t value)
BASEPRI	Read	uint32_t __get_BASEPRI (void)
	Write	void __set_BASEPRI (uint32_t value)
CONTROL	Read	uint32_t __get_CONTROL (void)
	Write	void __set_CONTROL (uint32_t value)
MSP	Read	uint32_t __get_MSP (void)
	Write	void __set_MSP (uint32_t TopOfMainStack)
PSP	Read	uint32_t __get_PSP (void)
	Write	void __set_PSP (uint32_t TopOfProcStack)

실행 결과




```
#define SP_PROCESS_SIZE             0x200   /* Process stack size */
#define SP_PROCESS                  0x02    /* Process stack */
#define SP_MAIN                     0x00    /* Main stack */
#define THREAD_MODE_PRIVILEGED      0x00
                /* Thread mode has privileged access */
#define THREAD_MODE_UNPRIVILEGED    0x01
                /* Thread mode has unprivileged access */
```

소스 코드 분석 (1)

```
#if defined ( __CC_ARM   )
__ASM void __SVC(void)
{
  SVC 0x01
  BX R14
}
#elif defined ( __ICCARM__ )
static __INLINE  void __SVC()          { __ASM ("svc 0x01");}
#elif defined   (  __GNUC__  )
static __INLINE void __SVC()           { __ASM volatile ("svc 0x01");}
#endif
```

- IAR로 빌드를 한 경우는 __ICCARM__가 define이 되어 있고 이 부분의 코드가 수행.
- RIDE7으로 수행했을 경우는 __GNUC__가 define 되어 있기 때문에 이 부분의 코드가 수행
- 컴파일러마다 조금씩 어셈블리를 표현하는 규칙이 다르기 때문에 이와 같은 조건부 컴파일을 사용해야 한다.

소스 코드 분석 (2)

```
void SVC_Handler(void)
{
    /* Switch back Thread mode to privileged */
    __set_CONTROL(2);
}
```

- SVC, Supervisor Call 명령어
- 사용하는 방식은 SVC{cond} #imm 의 형태
  - ‘cond’는 optional condition code
  - ‘imm’은 8 비트의 0에서 255까지 값을 가질 수 있다. 하지만 프로세서는 이 값을 무시.
  - 이 값을 사용하려면 **Stack 되어 있는 PC 값을 이용해서 추출** 해낼 수 있다. 이러한 특별한 상황이 아니라면 뒤에 적히는 숫자는 큰 의미는 없다.
- __SVC 함수는 SVC exception이 발생하게 만들어서 SVC_Handler가 호출되도록 만든다.

소스 코드 분석 (3)

```
    /* Switch Thread mode Stack from Main to Process */
    /* Initialize memory reserved for Process Stack */
    for(Index = 0; Index < SP_PROCESS_SIZE; Index++)
    {
        PSPMemAlloc[Index] = 0x00;
    }
    printf("PSPMemAlloc address = 0x%0X₩n", PSPMemAlloc);
    printf("(1) CurrentStack = %d, ThreadMode = %d₩n",
            Get_Current_Stack(), Get_Current_ThreadMode());
```

```
PSPMemAlloc address = 0x200007D4
(1) CurrentStack = 0, ThreadMode = 0
```

- PSPMemAlloc에 들어있는 모든 내용을 0으로 초기화 한 이후에 그 주소 값을 찍어 본다.
- 현재 Stack Pointer와 Thread Mode에 대한 내용 출력
- 리셋 후 처음 동작을 수행하는 것은 MSP를 사용하게 되고 privileged 레벨이 된다.

소스 코드 분석 (4)

```
/* Set Process stack value */
__set_PSP((uint32_t)PSPMemAlloc + SP_PROCESS_SIZE);
printf("(2) CurrentStack = %d, ThreadMode = %d₩n",
        Get_Current_Stack(), Get_Current_ThreadMode());
```

(2) **CurrentStack = 0**, ThreadMode = 0

- __set_PSP()는 MSR 명령을 통해서 PSP 레지스터에 값을 저장
  - 전달되는 값을 그저 복사하는 작업 외에는 수행하지 않는다.
  - 현 상태가 privileged 레벨이기 때문에 PSP 레지스터 값을 변경하는 것은 가능하지만 Stack Pointer와 Thread Mode에 대한 내용은 변한 것이 없다. 즉, PSP 값을 변경시킬 수는 있지만 이것이 실제로 사용하는 Stack Pointer나 privileged 레벨을 변화시킬 수는 없다.
- PSP에 저장하는 값이 SP_PROCESS_SIZE를 더해서 저장하는 이유?
  - Cortex-M3의 Stack이 Full-Descending 방식을 이용하고 있기 때문

  __IO uint8_t PSPMemAlloc[SP_PROCESS_SIZE];
  - 위 데이타 구조를 Process Stack으로 이용. PSPMemAlloc + SP_PROCESS_SIZE는 위 공간을 벗어나는 위치. 하지만 이렇게 값을 설정해야 Full-Descending 방식을 이용해서 스텍에 저장하기 전에 Stack Pointer 값을 먼저 줄이고 그 줄인 Stack Pointer 값의 위치에 값을 저장

소스 코드 분석 (5)

```
/* Select Process Stack as Thread mode Stack */
__set_CONTROL(SP_PROCESS);
printf("(3) CurrentStack = %d, ThreadMode = %d₩n",
        Get_Current_Stack(), Get_Current_ThreadMode());
```

(3) **CurrentStack = 2**, ThreadMode = 0

- 드디어 **__set_CONTROL을 통해서 사용하는 Stack Pointer를 PSP로 변경**하는 작업을 수행
- SP_PROCESS는 0x02로 정의되어 있고, 이것은 Stack Pointer를 PSP로 변경하게 된다.
- 결과를 보면 CurrentStack이 2로 바뀌어 있다.

소스 코드 분석 (6)

```
/* Get the Thread mode stack used */
if(Get_Current_Stack() == SP_MAIN) {
    printf("CurrentStack is Main Stack₩n");
} else {
    printf("CurrentStack is Process Stack₩n");
    /* Get process stack pointer value */
    printf("PSPValue = 0x%0X₩n", __get_PSP());
}
```

```
CurrentStack is Process Stack
PSPValue = 0x200009D4
```

- 동작과는 무관하게 현재의 상태를 읽어본 것
- 현재 Stack이 Process Stack임을 알수 있고, __get_PSP를 통해서 PSP의 값을 읽어보고 있다.
- __get_PSP는 MRS를 통해서 PSP 레지스터의 값을 전달해 주는 역할
- 읽은 값은 0x200009D4. PSPMemAlloc 주소를 읽어본 적이 있었고, 그 값은 0x200007D4. 0x200만큼 차이가 나는 것을 알수 있다. 설정한 값 그대로 이다.

소스 코드 분석 (7)

```
    /* Switch Thread mode from privileged to unprivileged */
    /* Thread mode has unprivileged access */
    __set_CONTROL(THREAD_MODE_UNPRIVILEGED | SP_PROCESS);
    /* Unprivileged access mainly affect ability to:
    - Use or not use certain instructions such as MSR fields
    - Access System Control Space (SCS) registers such as NVIC and Sy
sTick */
    printf("(4) CurrentStack = %d, ThreadMode = %d₩n",
            Get_Current_Stack(), Get_Current_ThreadMode());
```

```
(4) CurrentStack = 2, ThreadMode = 1
```

- 이제 privileged 레벨을 **unprivileged로 변경**
- 현재 상태가 privileged 레벨이기 때문에 아무런 문제 없이 이 작업을 수행할 수 있다.
- ThreadMode가 1로 변경된 것을 확인 할 수 있다.

소스 코드 분석 (8)

```
    /* Switch back Thread mode from unprivileged to privileged */
    /* Try to switch back Thread mode to privileged (Not possible, this
can be
    done only in Handler mode) */
    __set_CONTROL(THREAD_MODE_PRIVILEGED | SP_PROCESS);
    printf("(5) CurrentStack = %d, ThreadMode = %d₩n",
            Get_Current_Stack(), Get_Current_ThreadMode());
```

(5) CurrentStack = 2, **ThreadMode = 1**

- 위 코드는 privileged 레벨을 privileged로 변경하려고 시도 한다.
- 하지만 이 시도는 실패할 수밖에 없다.
- 왜냐하면 위에서 이미 상태를 **unprivileged**로 변경했기 때문에 이제 다시는 **privileged**로 돌아갈 수 없기 때문이다.
- 결국 찍어보면 전혀 변하지 않은 것을 알수 있다.

소스 코드 분석 (9)

```
    /* Generate a system call exception, and in the ISR switch back Thr
ead mode
    to privileged */
    __SVC();
    printf("(6) CurrentStack = %d, ThreadMode = %d₩n",
            Get_Current_Stack(), Get_Current_ThreadMode());
```

(6) CurrentStack = 2, **ThreadMode = 0**

- 여기서 **__SVC()를 호출**한다.
- __SVC()는 SVC_Handler가 불리도록 만들어 주게 되고 SVC_Handler의 내용에 이미 __set_CONTROL(2)가 수행되도록 해놓았기 때문에 정상적으로 모드를 변경하는 작업을 할 수 있다.
- __SVC()를 호출한 이후에 다시 출력을 해보면 ThreadMode가 0으로 정상적으로 변한 것을 알 수 있다.

소스 코드 분석 (10)

```
    /* Select Main Stack */
    __set_CONTROL(SP_MAIN);
    printf("(7) CurrentStack = %d, ThreadMode = %d₩n",
            Get_Current_Stack(), Get_Current_ThreadMode());
```

```
(7) CurrentStack = 0, ThreadMode = 0
```

- 이제 test 함수를 벗어나기 전에 변경했던 Stack Pointer를 다시 Main Stack으로 환원한다.