삼성 소프트웨어 멤버쉽 Digital System Design Lab.
배정양(baeyang@secsm.org) 박선화(psh@secsm.org)

**모빌 임베디드 시스템에 사용되는 하드웨어가 저전력, 고밀도, 고성능이어야 한다면, 주저 없이 ARM 계열을 사용하라고 권장하고 싶다.**
**또한 실제로 모빌 시스템은 저전력 설계기술로 널리 알려진, 32-Bit RISC 마이크로 프로세서인 Advanced RISC Machines사의 ARM 코어가 많이 사용되고 있다.**
**이번 호에서는 ARM7로의 포팅에 필요한 프로그래밍을 설명할 것이다.**
**포팅에 앞서 ARM7 프로세서에 대해 알아보자.**

## ARM7 이해하기

### 1. ARM7 프로세서의 개요 및 특성

(1) 32Bit RISC 프로세서

ARM7은 내부적으로 32비트 데이터 버스와 32비트의 어드레스 버스들 제공한다. 내부적이란 말은, ARM이 Core 형태로 여러 종류의 칩에 탑재되기 때문인데, 어떤 칩은 어드레스 라인을 내부적으로는 32비트를 사용하고 외부적으로는 Memory Management Unit을 통해 24비트를 사용하기도 한다.

(2) Big/Little Endian 모드지원

인텔 계열의 CPU에서는 메모리에 여러 바이트의 내용을 저장할 때, 하위 바이트가 먼저 오고 상위 바이트일수록 뒤에 오는데, 이런 형태를 Little Endian이라고 한다. 모토로라 계열의 CPU처럼 최상위 바이트가 먼저 오는 방식이 Big Endian이다. 해당 설정은 ARM7이 구현된 칩에 따라 틀리지만, 보통 칩 외부에서 핀 입력을 받아 결정한다.

(3) High Performance RISC

ARM7의 경우 3V를 사용하고 25MHz의 클록을 사용할 때 약 17MIPS가 나온다.

(4) Fast Interrupt Response

일반적인 인터럽트와 인터럽트 처리를 빠르게 해 줄 수 있는 FAST 인터럽트 기능이 있다. 세 개의 레지스터를 별도로 제공해서, 인터럽트 처리 루틴에서 레지스터를 저장, 복구하는 시간을 줄일 수 있다.

(5) Excellent high level language support

C나 다른 언어에서 일반적으로 사용하는 기능들을 바로 구현할 수 있는 명령어들이 제공된다. 예를 들어 변수의 앞, 뒤에 ++나 --를 붙이는 형식의 오퍼레이션을 제공한다.

(6) Simple & Powerful Instruction Set

ARM의 명령어는 그 종류가 적으면서도, 다양하게 적용시킬 수 있는 특징이 있다. 4개의 그룹과 11개의 명령어 타입을 가지고 있다.

여러분도 나중에 접해 보겠지만, 배우기 쉽고 강력한 기능을 제공한다.

## 2. ARM7의 구조

ARM7의 주요 기능 블록을 그림 1에 나타내었다. 레지스터와 32비트 연산이 가능한 ALU, Booth's 곱셈기, Address Incrementer 등이 있다.

- The read and write data register blocks
- The instruction decoder and control logic
- The multi-port register bank
- The Booth's multiplier
- The barrel shifter
- The Arithmetic Logic Unit  ALU The address register and address incrementer

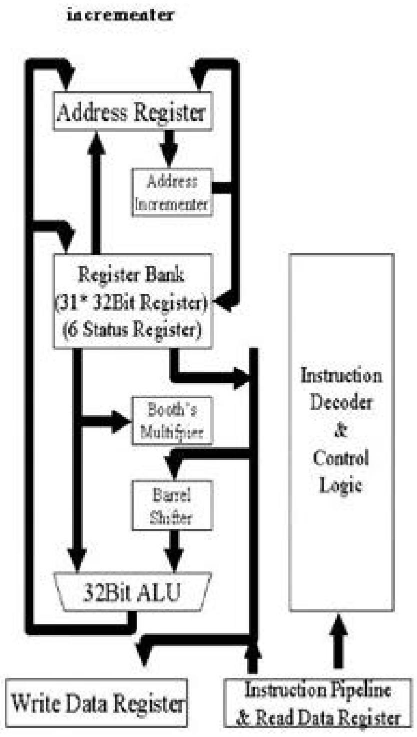

incrementer
Address Register
Address Incrementer
Register Bank (31* 32Bit Register) (6 Status Register)
Booth's Multiplier
Barrel Shifter
32Bit ALU
Instruction Decoder & Control Logic
Write Data Register
Instruction Pipeline & Read Data Register

그림 1. ARM7 Core block diagram

(1) 레지스터

ARM7에는 31개의 32Bit 레지스터가 있다. 동작모드에 따라 6개의 Status 레지스터를 갖는다.

(2) ALU

32Bit 연산이 가능한 ALU가 제공된다. ALU의 한쪽 입력은 Barrel Shifter로 연결되어 있어서 ALU의 인수 하나는 레지스터에서 바로 들어오고, 다른 하나는 레지스터나 버스에서 Barrel Shifter를 거쳐 입력되도록 되어 있다.

이런 이유로 ARM7에서는 제2오퍼랜드를 지정할 때, 해당 값을 쉬프트 시켜서 사용할 수 있다. 보통 다른 CPU에서는 쉬프트 명령이 따로 있는데, ARM7에서는 따로 존재하는 것이 아니라 대부분의 명령에서 옵션으로 적용시킬 수 있다.

(3) Booth's 곱셈기

곱셈 기능을 제공하는 32비트 Booth's 곱셈기가 있다. 곱셈기는 32비트 연산을 지원하며, 32비트의 두 입력을 곱하여 결과가 32비트를 넘더라도 넘는 부분은 버리고 32비트만을 남긴다.

이외에 인스트럭션 디코더와 인크리먼터가 달린 어드레스 레지스터 등이 있고, 32비트의 어드레스와 데이터 버스로 연결된다. ARM7은 파이프라인을 제공하고, 추가적으로 StrongARM과 같은 칩에서 캐쉬 기능과 MMU 기능 등도 제공한다. 상세한 내용은 ARM7 관련 문서를 참고하기 바란다.

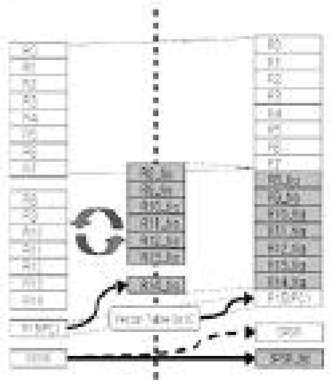


그림 2. ARM7 Exception  동작

(4) Exception

ARM7에서 FIQ(Fast Interrupt reQuest)와 IRQ

(Interrupt reQuest), Abort, Software Interrupt, Undefined Instruction Trap의 5가지 Exception이 있다. 이들 각각의 Exception이 발생하면 CPU는 대응하는 동작모드로 전환된다. 그림 2에 ARM7의 Exception 동작을 나타내었다. 각각의 Exception에 관한 상세한 내용은 잠시 후 언급하겠다.

## 3. 동작모드

ARM7에는 6가지의 동작 모드가 있다. 해당 모드는 User Mode, FIQ Mode, IRQ Mode, Supervisor Mode, Abort Mode, Undefined Mode가 있다. 동작모드에 따른 레지스터의 리맵핑을 그림 3에 나타내었다.

IRQ는 일반적으로 I/O 장치로부터의 입력이 들어오면 반응하는 흔히 말하는 인터럽트 Exception이다. IRQ의 종류로는 내부 타이머나 시리얼, 혹은 외부 IRQ 입력 등이 될 수 있다.

FIQ는 개념적으로 IRQ와 거의 유사한데, 다만 보다 빠른 처리를 할 수 있도록 제공되는 Exception이다. Abort는 CPU가 메모리로부터 인스트럭션을 가져오거나, 혹은 인스트럭션을 동작시키면서 데이터를 가져오려고 할 경우, 해당 메모리를 엑세스할 수 없다면 Abort Exception이 발생한다. Software Interrupt는 프로그램에서 임의로 인터럽트를 호출하는 경우를 말한다.

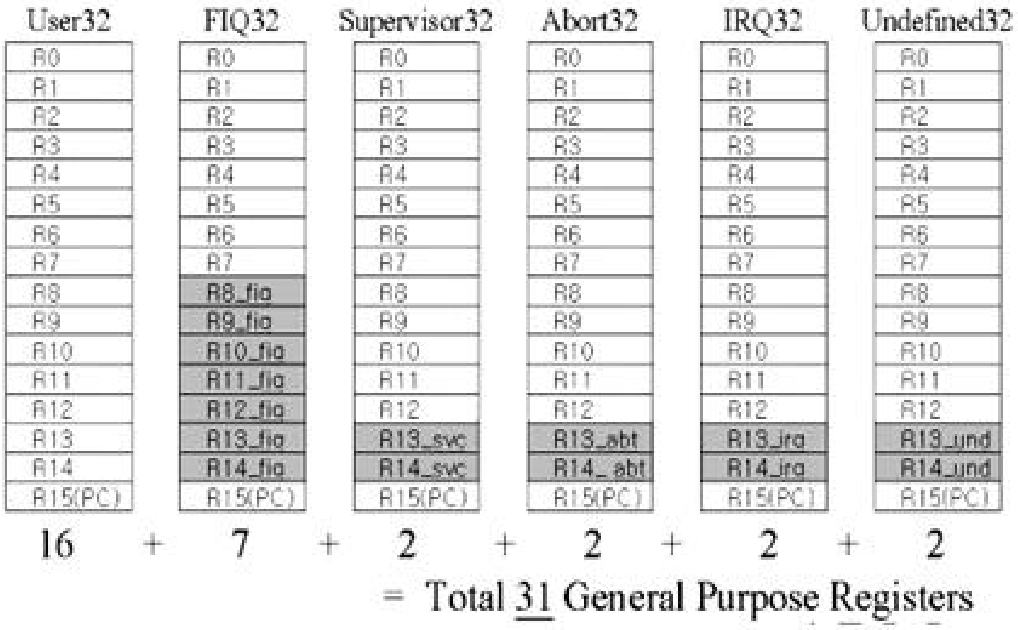
Undefined Instruction Trap은 ARM7에 정의되어 있지 않은 명령어를 만났을경우 발생하는 Exception이다.

(1) 동작모드와 범용 레지스터

ARM7은 각 동작모드에 따라 몇 가지 기능을 제공한다. 가장 큰 것으로 레지스터를 리맵핑 시키는 기능이다. 실제로는 각 모드마다 전용 레지스터가 따고 존재하는데, CPU의 동작모드는 보통 때는 User 모드이다. 이때 R0에서 R15까지의 레지스터를 사용한다. FIQ가 발생했을 경우 동작모드는 FIQ 모드로 바뀌게 된다. 이와 함께 R8에서 R14까지의 7개 레지스터는 FIQ 전용 레지스터로 리맵핑 된다. 즉, User 모드의 레지스터와는 별개였던 레지스터 7개가 R8부터 R14까지의 위치에 배치되는 것이다.

(2) 동작모드와 PSR(Status Register)

범용 레지스터와 비슷하게 PSR 역시 동작모드 마다 분리되어 관리한다. 해당 레지스터는 뒤에 모드이름을 붙여



| User32 | FIQ32 | Supervisor32 | Abort32 | IRQ32 | Undefined32 |
|---|---|---|---|---|---|
| R0 | R0 | R0 | R0 | R0 | R0 |
| R1 | R1 | R1 | R1 | R1 | R1 |
| R2 | R2 | R2 | R2 | R2 | R2 |
| R3 | R3 | R3 | R3 | R3 | R3 |
| R4 | R4 | R4 | R4 | R4 | R4 |
| R5 | R5 | R5 | R5 | R5 | R5 |
| R6 | R6 | R6 | R6 | R6 | R6 |
| R7 | R7 | R7 | R7 | R7 | R7 |
| R8 | R8_fiq | R8 | R8 | R8 | R8 |
| R9 | R9_fiq | R9 | R9 | R9 | R9 |
| R10 | R10_fiq | R10 | R10 | R10 | R10 |
| R11 | R11_fiq | R11 | R11 | R11 | R11 |
| R12 | R12_fiq | R12 | R12 | R12 | R12 |
| R13 | R13_fiq | R13_svc | R13_abt | R13_irq | R13_und |
| R14 | R14_fiq | R14_svc | R14_abt | R14_irq | R14_und |
| R15(PC) | R15(PC) | R15(PC) | R15(PC) | R15(PC) | R15(PC) |

16 + 7 + 2 + 2 + 2 + 2

= Total 31 General Purpose Registers

그림 3. ARM7 General Register

서 SPSR_fiq, SPSR_svc, SPSR_abt, SPSR_irq, SPSR_und와 같은 이름으로 사용한다. 따라서 PSR의 개수는 CPSR을 포함하여 총 6개가 된다. SPSR은 CPSR 값을 저장해 두는 역할을 한다. 범용레지스터가 아예 맵핑이 바뀌는데 반해, 모드가 바뀔 경우, 예를 들어 IRQ가 발생했다면 기존에 User 모드에서 사용하는 CPSR 값을 SPSR_irq에 저장한다. 그리고 IRQ 모드에서는 CPSR과 SPSR_irq를 둘 다 볼 수 있다. 후에 IRQ가 끝나는 시점에서 SPSR_irq의 내용을 CPSR로 복구하면 원래의 CPSR 값이 유지되는 것이다.

(3) Exception 간의 우선순위

ARM7의 경우 IRQ, FIQ를 지원하지 않는다. 해당 인터럽트가 발생하면 CPU는 CPSR의 인터럽트 금지 플래그를 설정한다. 만약 도중에 인터럽트를 받고 싶다면 사용자가 해당 플래그를 클리어 해주어야 한다. 또, 동시에 여러 개의 Exception이 발생한 경우라면 우선순위는 다음과 같다.

1) Reset (가장 높은 우선순위)
2) Data abort
3) FIQ
4) IRQ
5) Prefetch abort
6) Undefined Instruction, Software Interrupt

## 4. ARM7의 레지스터

ARM7의 어셈블러에서 사용하는 범용 레지스터 키워드는 R0에서 R15까지 16개이다. 즉, 사용자가 한번에 사용할 수 있는 레지스터는 16개인 셈이다. 그 중에 몇 개는 프로그램 카운터(PC)나 스택 포인터(SP) 등의 용도로 사용된다. 나머지 레지스터는 CPU 동작모드(Exception)와 관련되어 R0에서 R15로 탑재 맵핑되어 사용된다.

(1) Special Purpose General Register

1) Program Counter(R15)

R15는 다른 CPU에서의 PC와 같은 역할을 한다. 다만 차이가 있다면 R15를 일반 다른 레지스터처럼 오퍼랜드로 사용할 수 있다는 점이고, ARM 어셈블러에서는 PC라는 키워드와 R15를 동일하게 취급한다.

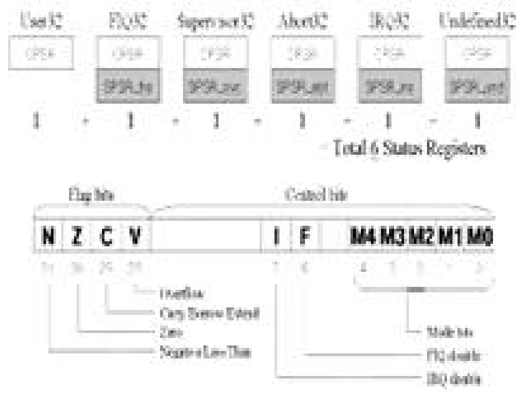


그림 4. ARM7 Status Register

2) Stack Pointer(R13)

ARM7에는 Stack을 위한 명령어가 따로 없기 때문에, SP라는 키워드를 사용하여 R13을 쓸 수 있는데, 묵시적으로 R13을 스택포인터로 사용할 수 있도록 정해 놓은 듯 하다. Push 명령이나 Pop 명령이 없으므로, ARM7에서는 같은 기능을 일반 데이터 전송 명령을 통해 해결한다.

(2)Link Register(R14)ARM7에서는 CALL, RET와 같은 명령이 없다. 대신 Branch with Link라는 명령(BL)이 있는데, 해당 명령을 수행하며 CALL과 비슷하게 다음에 수행될 PC(R15)값을 스택이 아닌 LR(R14)에 넣고 분기번지를 PC(R15) 레지스터에 넣고 분기한다. 즉, 스택을 사용하지 않는 것이다. 복귀할 때는 RET 대신 mov pc, lr이라는 데이터 전송명령으로 복귀한다.

(3) Status Register

그림 4와 같이 ARM7에는 32비트 Status Register가 6개 있다. 그러나 6개 모두를 한꺼번에 사용하지는 않는다. Status Register는 PSR이라고 부르며, 일반적으로 CPSR이라고 하여 Current Processor Status Register로 부르기도 한다. PSR은 크게 Flag Bits 부분과 Control Bits 부분으로 나뉘는데 이들 각각을 잠시 들여다보면 다음과 같다.

1) Flag Bits

- Negative/Less Than Flag : N으로 표기되는 이 플래그는 연산의 결과가 마이너스인 경우에 세트된다.

- Zero Flag : Z으로 표기되는 이 플래그는 연산의 결과

가 0이 되었을 경우에 세트된다.

- Carry/Borrow/Extend Flag : C로 표기되는 이 플래그는 자리올림이나 내림이 발생한 경우, 그리고 Shift 연산 등에서 사용된다.

- Overflow Flag : V로 표기되는 이 플래그는 연산의 결과가 오버플로우 됐을 경우 사용된다.

2) Control Bits

- IRQ / FIQ Disable Bit : ARM7의 인터럽트 중 IRQ와 FIQ를 금지시킬 수 있는 플래그이다.

- Mode Bits : M0에서 M4까지의 모드 비트는 CPU의 6개 동작 상태를 나타낸다. 즉, ARM7은 6개의 동작 모드를 가진다.

### 5. ARM7 인스트럭션의 특징

ARM7은 32비트 Core로, 모든 명령어가 32Bit 하나의 Word로 구성되며 처리된다는 것이다. 그러므로 모든 명령어를 같은 크기로 처리함에 따라 파이프라인 구현이 용이하며, 명령어 해석기를 설계할 경우 예외 처리부분이 없으므로, 쉽고 고속으로 처리할 수 있게 된다. 반면, 코딩 시에 몇 가지 제한이 따른다는 점이 있다. ARM 인스트럭션의 다른 특징으로는 모든 명령어를 조건적으로 실행시킬 수 있다는 것인데, 예를 들어, 8086에서는 jz, jc와 같은 점프 명령을 사용한다. 이는 제로 플래그가 설정되어 있으면 점프하라, 혹은 캐리 플래그가 설정되어 있으면 점프라는 의미이다. ARM의 경우엔 이러한 플래그의 사용이 점프명령에 국한되지 않고, 예외 없이 모든 명령어에 사용할 수 있다는 것이다. 또 다른 특징으로는 대부분의 명령어에 S라는 접미사를 사용하여 플래그 레지스터에 영향을 주는지에 대한 여부를 결정할 수 있다는 것이다.

이외에도 ARM7에 대해 좀더 알아보아야 할 부분이 많지만 이쯤에서 실제 포팅을 위한 프로그래밍에 들어가 보자.

## ARM7으로의 uC/OS 포팅

Target Board로는 StrongARM 아키텍쳐를 사용한 SA1100을 사용하였다. 이 중 특히 인터럽트 처리부분은 8086 계열과 동작방식이 달라, 해당 처리를 위하여 새로운 메커니즘을 사용하였다. 프로세서의 특성에 따른 어셈블리 코드와 C 코드로 작성하였고, 하드웨어 특성에 따라 작성되어야 하는 OS_CPU.H, OS_CPU_A.S, OS_CPU.C 등의 파일을 수정 및 추가하였다. 세부 사항들은 각 파일 별로 구분하여 설명한다.

### 1. OS_CPU.H

OS_CPU.H에서는 #define과 macro 그리고 데이터형에 대한 선언을 해준다. 표 1에서는 일반적인 OS_CPU.H의 일부 코드와 StrongARM 아키텍쳐에 맞는 OS_CPU.H 코드의 일부를 나타내었다.

#define OS_ENTER_CRITICAL( ) *ICMR = 0x00008000

#define OS_EXIT_CRITICAL( ) *ICMR = 0x04008000는 프로그램 수행 중 critical section으로 진입할 때 인터럽트를 disable 시키고, 후에 다시 인터럽트를 enable 시켜주기 위해 Mask Bit를 셋팅 시켜주는 부분이다.

#define OS_TASK_SW( ) OSCtxSw( )는 스케쥴링이 필요할 때 OS_TASK_SW( )가 호출되는데, 뒤쪽에서 설명하겠지만 이의 context switching과, 스택으로의 저장 및 Task Running 작업을 어셈블리 코드로 작성한다.

### 2. OS_CPU_C.C

Task가 생성될 때 표 2와 같은 해당 Task의 stack이 생성되는데, full stack을 사용한다. 아래 표를 참조하여 설명하면, 주석으로 처리된 CPSR은 Flag Register를 저장하고, R0는 각 Task로 넘어오는 인수가 저장된다. Entry Point란 Task가 복귀할 때 필요한 Task의 start pointer가 저장된다.

### 3. OS_CPU_A.S

UC/OS-II를 포팅하기 위해서는 4개의 어셈블리 함수를 작성해야 한다. OSCtxSw( ), OSStartHighRdy( ), OSIntCtxSw( ), OSTickISR( )이 그것이다. In-line

표 1. OS_CPU.H

```
typedef unsigned char   BOOLEAN;

typedef unsigned char   INT8U;
          /* Unsigned   8 bit quantity   */
typedef signed    char  INT8S;
```

```
          /* Signed      8 bit quantity   */
typedef unsigned short INT16U;
          /* Unsigned 16 bit quantity    */
typedef signed    short INT16S;
          /* Signed    16 bit quantity    */
typedef unsigned long   INT32U;
          /* Unsigned 32 bit quantity    */
typedef signed    long   INT32S;
          /* Signed    32 bit quantity    */
typedef float              FP32;
        /* Single precision floating point*/
typedef double             FP64;
       /* Double precision floating point*/
typedef unsigned int    OS_STK;
       /* Each stack entry is 16-bit wide*/
#define BYTE                INT8S
       /* Define data types for backward
                       compatibility ...     */
#define UBYTE              INT8U
       /* . to uC/OS V1.xx.   Not actually
                         needed for ...      */
#define WORD                INT16S
         /* ...   uC/OS-II.                   */
#define UWORD              INT16U
#define LONG                INT32S
#define ULONG              INT32U

#define OS_ENTER_CRITICAL()
                       *ICMR=0x00008000
#define OS_EXIT_CRITICAL()
                       *ICMR=0x04008000
#define   OS_TASK_SW()
                       OSCtxSw()
```

어셈블리가 가능한 컴파일러의 경우는 OS_CPU.C 코드에 해당 함수들을 구현해도 되지만, 여기에서는 어셈블리 파일을 작성하였다.

(1) OSStartHighRdy( ) 함수

표 3의 코드는 StrongARM 아키텍쳐에 맞게 구현된 OSStartHighRdy( ) 함수이다. (2) (4)은 멀티테스킹이 시작됨을 알려주는 코드이고, (5) (6)은 가장 높은 우선순위를 갖는 TCB의 스택포인터를 가져오는 것이다. (10)에서는 Task의 context를 load 한 후, Task를 실행한다.

표 2. task stack 생성

```
Stk   =   (INT32U *)ptos;   /* Load stack pointer */
*(--stk) = (INT32U)0;       /* Dummy Data        */
*(--stk) = (INT32U)task;    /* Entry Point -> PC */
*(--stk) = (INT32U)0;       /*r14        */
*(--stk) = (INT32U)0;       /*r12        */
*(--stk) = (INT32U)0;       /*r11        */
*(--stk) = (INT32U)0;       /*r10        */
*(--stk) = (INT32U)0;       /*r9          */
*(--stk) = (INT32U)0;       /*r8          */
*(--stk) = (INT32U)0;       /*r7          */
*(--stk) = (INT32U)0;       /*r6          */
*(--stk) = (INT32U)0;       /*r5          */
*(--stk) = (INT32U)0;       /*r4          */
*(--stk) = (INT32U)0;       /*r3          */
*(--stk) = (INT32U)0;       /*r2          */
*(--stk) = (INT32U)0;       /*r1          */
*(--stk) = (INT32U)pdata;    /*r0          */
*(--stk) = (INT32U)0;       /* CPSR       */
```

(2) OSCtxSw( ) 함수

OSCtxSW( ) 함수는 Context switching 코드와 Task를 실행해주는 코드로 구현되었다. (1) (4)는 Switching 될 Task의 address, context 그리고 CPSR을 저장하는 부분이다. (9) (17)은 가장 높은 우선순위를 가진 Task를 현재 실행 Task를 가리키게 하는 부분이고, (18) (20)은 실행될 Task의 context를 읽어서 Task를 실행하는 부분으로 구현하였다. 상세한 코드는 표 4에 나타내었다.

(3) OSIntCtxSw( ) 함수

ContextOS_CORE.C 파일에 있는 OSIntExit( ) 함수에 의해 호출되는 함수로, 인터럽트 수행 후에 가장 높은 우선순위의 Task를 찾아내어, 찾아낸 Task가 인터럽트 호출 이전의 Task인지를 구분하고, 기존의 Task가 아니면, 저장할 필요 없이, context switching을 하고 다시 인터럽트를 enable하기 위해 Mask bit를 세팅 시켜준 후, stack의 내용을 pop하고 Task를 실행시켜주는 함수이다. 상세한 코드는 표 5에 나타내었다.

(4) OSTickISR( ) 함수

이 함수는 1/100초마다 Timer Clock을 세팅 해주는

표 3. OSStartHighRdy( )

```
; Call user defined task switch hook
(1) BL      OSTaskSwHook

; Indicate that multitasking has started
(2) LDR   r0,=OSRunning
(3) MOV   r1,#1
(4) STRB  r1,[r0]

; r0 <= &OSTCBHighRdy
(5) LDR   r0,=OSTCBHighRdy
; r0 <=   OSTCBHighRdy
(6) LDR   r0,[r0]

; sp <=   OSTCBHighRdy->OSTCBStkPtr
(7) LDR   sp,[r0]

; restore SP...
(8) LDMFD sp!, {r0}
(9) MSR   CPSR_flg,r0

; Load task's context & Run task
(10) LDMFD    sp!,{r0 - r12, lr , pc}
```

함수이다. 해당 코드는 표 6에서 보여준다. OSCR은 현재 Time Clock의 값을 나타내고, 현재 Time 값에서 얼마를 더해야만 Time Clock이 세팅되는지를 나타낸다.

## strongARM SA-1100 개발보드 설명

우리가 포팅한 ARM CPU를 사용한 보드는 Intel에서 생산한 strongARM SA-1100개발 보드이다. ARM은 CPU를 직접 생산하지 않고 여려 회사들에게 CORE와 소프트웨어 개발 툴들을 제공한다. CPU인 StrongARM은 ARM7 core를 사용하고 있다. 보드는 onboard memory (SRAM, DRAM, Flash, and ROM), LCD panels, touch screen, Keyboard, audio accessories (telephone jack, microphone, speaker), PCMCIA connector, serial I/O interface(USB port, IrDA infrared support, SDLC port, two UART ports), logic analyzer connectors를 포함하고 있는 다용도 보드이다. RTOS를 보드에 올리기 위한 보드의 사양은 CPU와 주변 메모리, 최소의 I/O장치만 있어도 충분히 가능하

표 4. OSCtxSw( )

```
; push resume address
(1) STMFD sp!, {lr}
; push rest context
(2) STMFD sp!, {r0 - r12, lr}
(3) MRS   r0,CPSR
; push CPSR
(4) STMFD sp!, {r0}
; r0 <= &OSTCBCur
(5) LDR   r0,=OSTCBCur
; r0 <=   OSTCBCur
(6) LDR   r0,[r0]
; OSTCBCur->OSTCBStkPtr = sp
(7) STR   sp,[r0]

; Call user defined task switch hook
(8) BL      OSTaskSwHook

; r0 <= &OSTCBCur
(9) LDR   r0,=OSTCBCur
; r1 <= &OSTCBHighRdy
(10) LDR  r1,=OSTCBHighRdy

; r2 <=   OSTCBHighRdy
(11) LDR  r2,[r1]
; OSTCBCur = OSTCBHighRdy
(12) STR  r2,[r0]

; r0 <= &OSPrioCur
(13) LDR  r0,=OSPrioCur
; r1 <= &OSPrioHighRdy
(14) LDR  r1,=OSPrioHighRdy
; r3 <=   OSPrioHighRdy
(15) LDRB  r3,[r1]
; OSPrioCur = OSPrioHighRdy
(16) STRB r3,[r0]

; sp <=   OSTCBHighRdy->OSTCBStkPtr
(17) LDR  sp,[r2]

; restore SP...
(18) LDMFD sp!,{r0}
(19) MSR  CPSR_flg,r0
; Load task's context & Run task
(20) LDMFD    sp!,{r0 - r12, lr , pc}
```

다. 그런데 주변에 ARM을 구하기가 쉽지 않게 때문에 여러 루트로 알아보아 테스트할 수 있도록 구하다 보니 이런 종합적인 개발 보드를 입수해 프로그래밍하게 되었다.

그림 5에서는 보드에 연결된 주변 장치들의 모습을 보여주는 것이고, 그림 6은 실질적으로 기본 장치들이 어떻게

표 5. OSIntCtxSw( )

```
ADD      sp,sp,#4
; r0 <= &OSCTBCur
LDR      r0,=OSTCBCur
; r0 <=  OSCTBCur
LDR      r0,[r0]
; OSTCBCur->OSTCBStkPtr = sp
STR      sp,[r0]
; Call user defined task switch hook
BL       OSTaskSwHook

; r0 <= &OSTCBCur
LDR      r0,=OSTCBCur
; r1 <= &OSTCBHighRdy
LDR      r1,=OSTCBHighRdy

; r2 <=  OSTCBHighRdy
LDR      r2,[r1]
; OSTCBCur = OSTCBHighRdy
STR      r2,[r0]

; r0 <= &OSPrioCur
LDR      r0,=OSPrioCur
; r1 <= &OSPrioHighRdy
LDR      r1,=OSPrioHighRdy

; r3 <=  OSPrioHighRdy
LDRB     r3,[r1]
; OSPrioCur = OSPrioHighRdy
STRB     r3,[r0]

; sp <=  OSTCBHighRdy->OSTCBStkPtr
LDR      sp,[r2]

;*ICMR=0x04008000-> OS_EXIT_CRITICAL();
LDR      r0,=ICMR
LDR      r1,=0x04008000
STR      r1,[r0]

; restore SP...
LDMFD   sp!,{r0}
MSR      CPSR_flg,r0
; Load task's context & Run task
LDMFD    sp!,{r0 - r12, lr , pc}
```

표 6. OSTickISR( )

```
; Set Next Timer Clock about 1/100 sec..
LDR      r0,=OSCR
LDR      r0,[r0]
LDR      r1,=0x8fd2
ADD      r0,r0,r1
LDR      r1,=OSMR0
STR      r0,[r1]

; Notify uC/OS-II of ISR
LDR      r0,=OSIntNesting
LDRB     r1,[r0]
ADD      r1,r1,#1
STRB     r1,[r0]

; Process system tick
BL       OSTimeTick
; Notify uC/OS-II of end of ISR
BL       OSIntExit

LDMFD   sp!,{r0}

MSR      CPSR_flg,r0
LDMFD    sp!,{r0 - r12, lr , pc}
```

연결되어 있는지 보여주는 블럭도이다. 보드에 장착된 메모리는 512K의 SRAM이 있고 개발의 편의를 위해 ROM을 보드에 직접 부착하여 사용하지 않고 ROM 에뮬레이터를 사용하였다. 컴파일 한 프로그램은 롬 에뮬레이터에 다운 로딩하여 작동시켰다. 많은 주변 장치가 부착되어 있으나 첫 번째 목표가 uC/OS를 작동하도록 포팅하는 것이고 다른 주변장치를 인터페이스 하려면 많은 시간적인 투자가 이루어져야 하므로 보드에 장착된 LED와 직렬 통신으로 컴퓨터와 연결하여 컴퓨터의 통신 프로그램에 메시지를 표시하는 것으로 목표를 정했다. ARM보드에 대한 이야기를 조금 더 하자면 시중에서 일반적으로 쉽게 구할 수 있는 ARM보드는 삼성에서 Arm7 core를 이용해서 만든 KS32C50100 보드가 있다. 이더넷도 장착되어 있어 많은 것들을 해볼 수 있는 보드이다. SA-1100과 같은 CPU core를 사용하기 때문에 이번 연재에 나온 내용을 적용해 보기에 알맞은 보드라고 생각한다.

## 1. 개발 툴(ARM SDT)

소프트웨어 개발 툴은 ARM SDT(Software Devel-

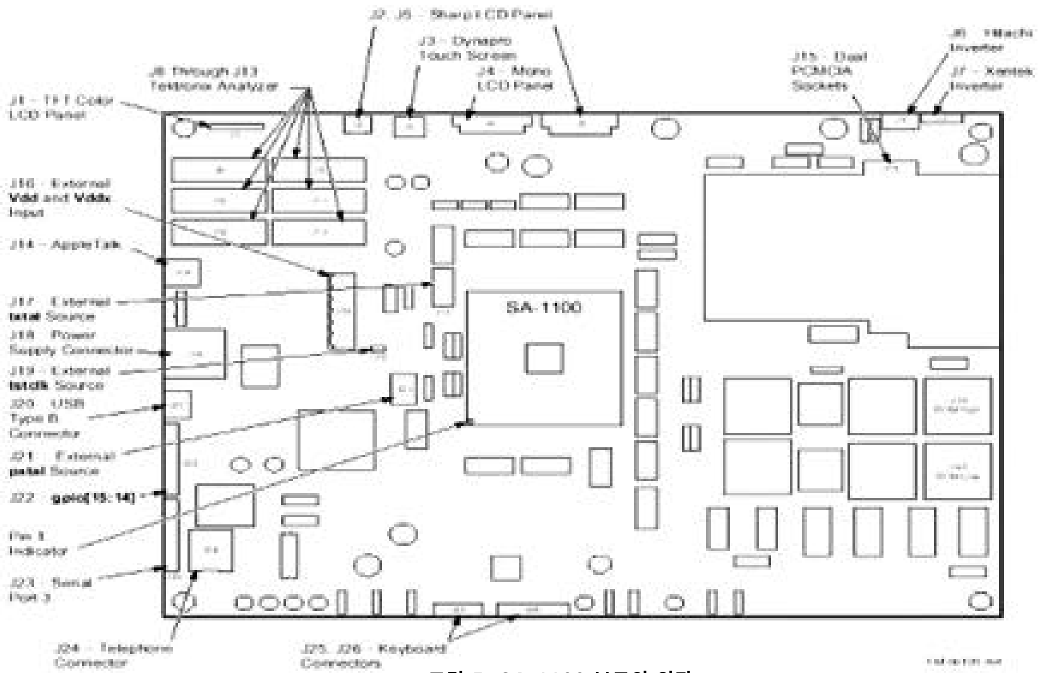


그림 5. SA-1100 보드의 외관

opment Toolkit) v2.5 Evaluation version을 사용하였다. 한달 간 사용할 수 있는 버전이다. 이 툴은 ARM홈페이지를 통해서 구할 수 있다.

ARM SDT는 윈도우용과 도스용 개발 툴을 지원하고 C, ASM도 지원한다. 우리는 이중에서 윈도우용 개발 툴인 ARM Project Manager를 사용하여 윈도우에서 컴파일 하였다.

툴의 사용법은 다른 여러 프로그램 툴들과 비슷하니 쉽게 사용할 수 있을 것이다. 여기서는 CPU와 보드의 메모리 맵에 따라 바꾸어 주어야 하는 중요하면서도 기본인 설정과 파일의 tree구조를 설명하겠다. 아래에서 나오는 수치는 SA-1100보드의 설정 값을 적어 놓은 것이다.

(1) ARM project Manager의 설정

1) C Compiler option

이곳에서는 프로세스의 종류와 주소 지정모드 그리고 주소 접근 방식의 설정을 사용하고 있는 보드의 상태와 동일하게 정의해 준다.

* <cc>=armcc option 선택

Tools -> Configure -> <cc>=armcc

* 선택값

Addressing Mode : 32 Bit

Processor : ARM7TDMI

Byte Sex : Little Endian

2) Link Option

주로 메모리 설정, 즉 ROM과 RAM이 보드에 어떻게 맵 되어 있는지에 대한 설정과 보드의 초기화 프로그램을 연결시켜주는 것이다.

* armlink option 선택

Tools -> Configure -> armlink

* Output page

Output Format : ARM ELF image fromat

* Entry and Base page 메모리 설정

Read-only:0x8000

* imageLayout page 설정

-object file : init.o

-are Name : init

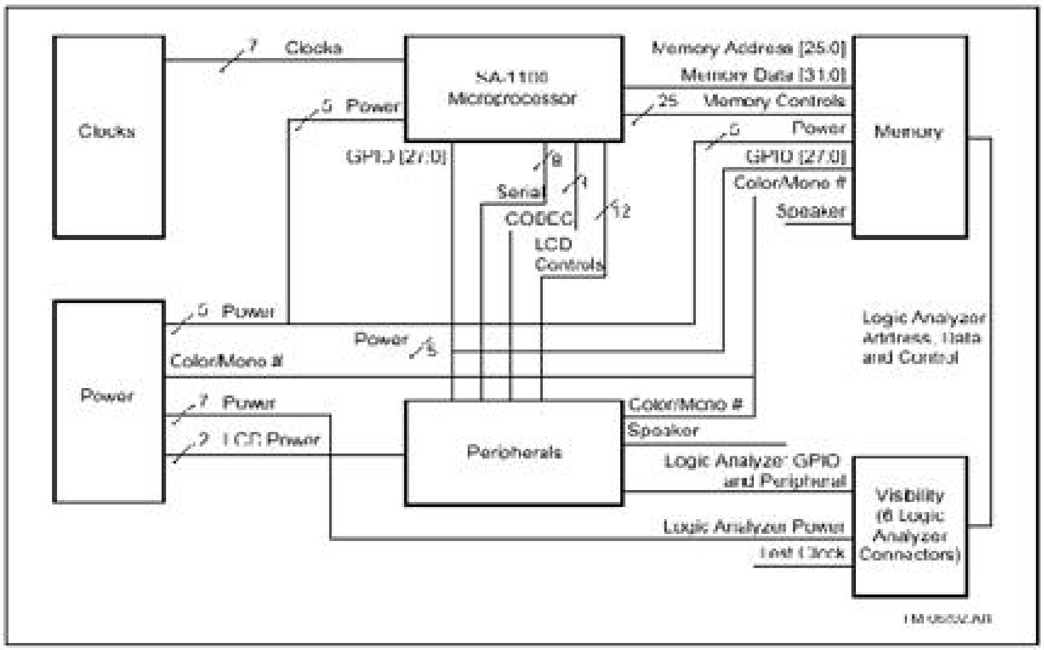


그림 6. Block diagram

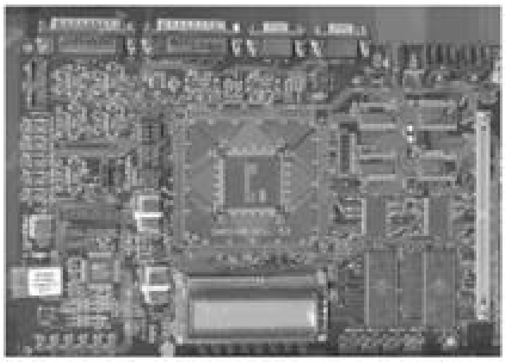


그림 7. KS32C50100 보드 실물사진

init.o file 안에는 시스템의 전반적인 설정이 들어 있는 파일이다. 이 파일을 수정함으로써 기본적인 하드웨어 설정을 바꿀 수 있다. UC/OS를 포팅하는 데 있어서 하드웨어와 밀접한 부분을 바꾸려면 이 파일을 참조하여 작성해야 한다. 꼭 한번 봐으면 한다. 메모리맵, 특수레지스터의 정의 인터럽트, UART, 주변 장치 들에 대한 설정이 모두 들어 있다.

## 2. 프로그램 개발 과정

사용하고자 하는 보드에 대한 기본 설정을 해주고 난 뒤 프로그램의 작성에 들어가게 된다. ARM Project Manager에서 새로운 프로젝트를 생성하고 그 안에 수정된 uC/OSII에 대한 소스들을 추가 시켜주고 task를 돌리기 위해 직접 작성한 source, 초기화에 필요한 파일 등을 연결 시켜준 뒤 컴파일 한다. 이렇게 해서 작성된 실행 파일을 롬 에뮬레이터에 다운로딩하여 실행시킨다.

## 3. 포팅한 소스의 구조

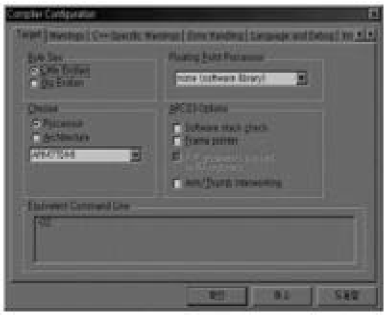
그림 12에서 생성된 프로젝트의 소스 구조를 보여 주고 있다. 이중 대부분은 지난 호까지 보아온 uC/OS에 대한 소스이고 ARM을 위해 새로 추가된 소스들만을 설명하겠다.

(1) sa1100.a

ARM에서 사용하는 레지스터들이 선언되었다.

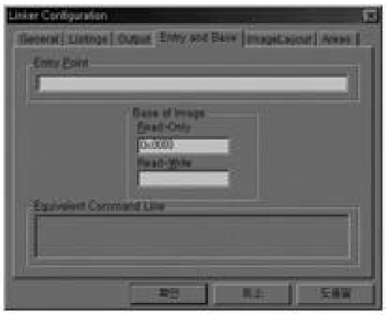
(2) EX11.c (그림 13)

실행될 task들이 기술되어 있는 소스이다. 이 소스에서



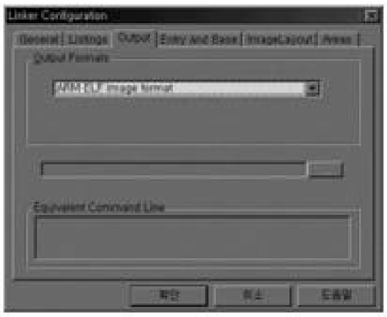
```
Compiler Configuration
Target | Warnings | C++ Specific Warnings | Error Handling | Language and Debug | ...
Byte Sex
  (•) Little Endian
  ( ) Big Endian
Floating Point Processor
  none (software library)
Choose
  (•) Processor
  ( ) Architecture
APCS3 Options
  [ ] Software stack check
  [ ] Frame pointer
  [ ] Arm/Thumb interworking
Equivalent Command Line
확인   취소   도움말
```

그림 8. Compiler Configuration



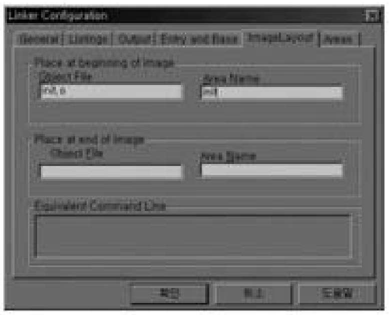
```
Linker Configuration
General | Listings | Output | Entry and Base | ImageLayout | Areas |
Entry Point
Base of Image
  Read-Only
  0x8000
  Read-Write
Equivalent Command Line
확인   취소   도움말
```

그림 10. Linker Configuration - Enrty and Base



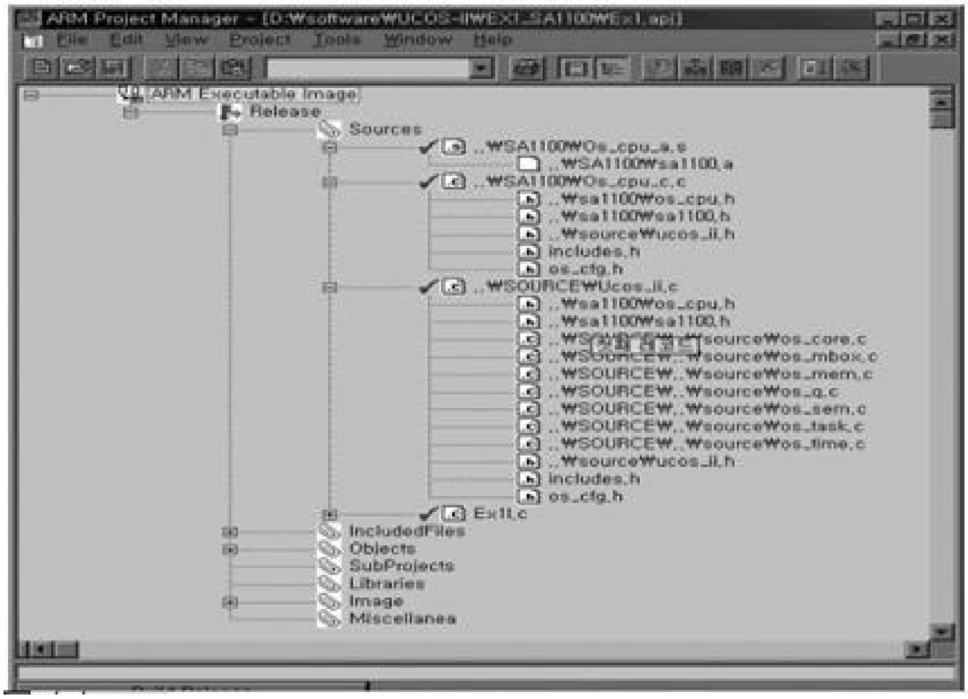
```
Linker Configuration
General | Listings | Output | Entry and Base | ImageLayout | Areas |
Output Formats
  ARM ELF image format
Equivalent Command Line
확인   취소   도움말
```

그림 9. Linker Configuration - Output



```
Linker Configuration
General | Listings | Output | Entry and Base | ImageLayout | Areas |
Place at beginning of image
  Object File          Area Name
  init.o               init
Place at end of image
  Object File          Area Name
Equivalent Command Line
확인   취소   도움말
```

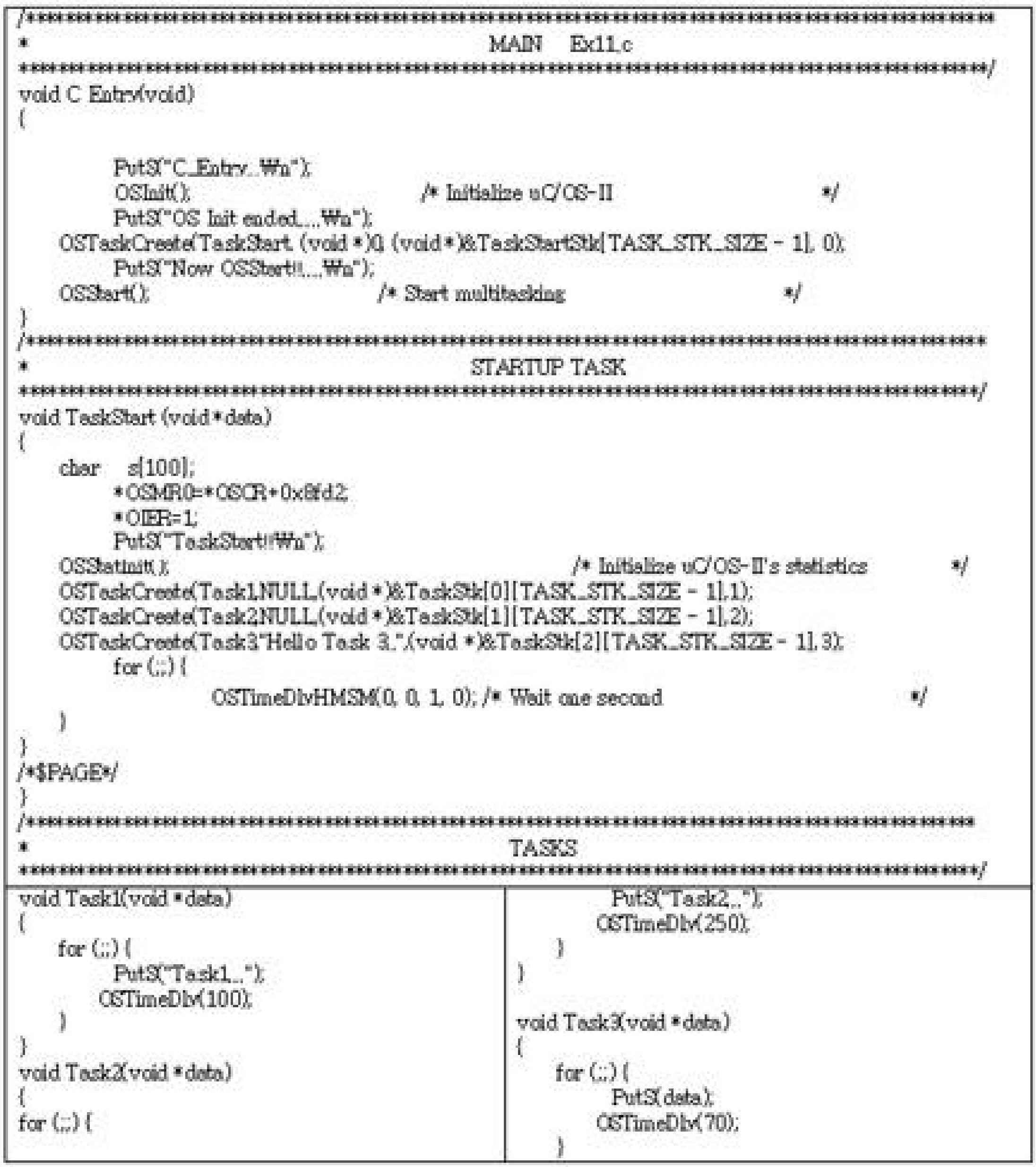
그림 11. Linker Configuration - Image Layout Page



```
ARM Project Manager - [D:₩software₩UCOS-II₩EX1-SA1100₩Ex1.apj]
File  Edit  View  Project  Tools  Window  Help

ARM Executable Image
  Release
    Sources
      ..₩SA1100₩Os_cpu_a.s
        ..₩SA1100₩sa1100.a
      ..₩SA1100₩Os_cpu_c.c
        ..₩sa1100₩os_cpu.h
        ..₩sa1100₩sa1100.h
        ..₩source₩ucos_ii.h
        includes.h
        os_cfg.h
      ..₩SOURCE₩Ucos_ii.c
        ..₩sa1100₩os_cpu.h
        ..₩sa1100₩sa1100.h
        ..₩SOURCE₩..₩source₩os_core.c
        ..₩SOURCE₩..₩source₩os_mbox.c
        ..₩SOURCE₩..₩source₩os_mem.c
        ..₩SOURCE₩..₩source₩os_q.c
        ..₩SOURCE₩..₩source₩os_sem.c
        ..₩SOURCE₩..₩source₩os_task.c
        ..₩SOURCE₩..₩source₩os_time.c
        ..₩source₩ucos_ii.h
        includes.h
        os_cfg.h
      Ex1.c
    IncludedFiles
    Objects
    SubProjects
    Libraries
    Image
    Miscellanea
```

그림 12. 생성된 프로젝트와 소스 구조



```
/*****************************************************************************************
*                                            MAIN    Ex11.c
*****************************************************************************************/
void C_Entry(void)
{

        PutS("C_Entry..\n");
        OSInit();                           /* Initialize uC/OS-II                   */
        PutS("OS Init ended....\n");
    OSTaskCreate(TaskStart, (void *)0, (void *)&TaskStartStk[TASK_STK_SIZE - 1], 0);
        PutS("Now OSStart!!....\n");
    OSStart();                          /* Start multitasking                    */
}
/*****************************************************************************************
*                                          STARTUP TASK
*****************************************************************************************/
void TaskStart (void *data)
{
    char    s[100];
        *OSMR0=*OSCR+0x8fd2;
        *OIER=1;
        PutS("TaskStart!!\n");
    OSStatInit();                                          /* Initialize uC/OS-II's statistics        */
    OSTaskCreate(Task1,NULL,(void *)&TaskStk[0][TASK_STK_SIZE - 1],1);
    OSTaskCreate(Task2,NULL,(void *)&TaskStk[1][TASK_STK_SIZE - 1],2);
    OSTaskCreate(Task3,"Hello Task 3..",(void *)&TaskStk[2][TASK_STK_SIZE - 1],3);
        for (;;) {
                OSTimeDlyHMSM(0, 0, 1, 0); /* Wait one second                    */
    }
}
/*$PAGE*/
}
/*****************************************************************************************
*                                             TASKS
*****************************************************************************************/
void Task1(void *data)
{
    for (;;) {
        PutS("Task1..");
        OSTimeDly(100);
    }
}
void Task2(void *data)
{
for (;;) {
        PutS("Task2..");
       OSTimeDly(250);
    }
}

void Task3(void *data)
{
    for (;;) {
        PutS(data);
       OSTimeDly(70);
    }
```

그림 13. Ex11.c

는 3개의 예제 task를 작동시킨다. 메인 문에서는 OS가 시작된다는 내용은 PutS()라는 함수를 사용해서 직렬 포트로 컴퓨터의 통신 프로그램에 전송한다. 그리고 startup task 에서는 1초마다 LED를 점멸하여 테스트가 작동하고 있음을 보여준다.

예제 task 1, task 2는 직렬 전송으로 task가 작동할 때마다 Task1 Taks2 이라고 화면에 찍어준다. Task3는 Hello Task 3.. 이라는 문구를 task가 걸릴 때마다 화면에 찍어주게 된다. 간단하게 task가 돌아가는 지만 확인하도록 작성하였다. 이 함수들에 각자 하고자 하는 일들 작

성해 주면 되는 것이다.

## 4. 마치며

이번 연재에서는 앞의 두 번의 연재와 비슷하게 uC/OS를 ARM에 포팅하는 방법에 대해 알아보았다.

uC/OS가 x86을 기본으로 하여 제작된 프로그램이어서 ARM과는 상당한 CPU 특성 차이가 있어 많은 부분에서 의 변경이 필요했다. ARM에만 있는 여러 가지 모드라던가, 인터럽트가 작동했을 때의 동작의 다른 점 등이 포팅을 어렵게 하였다.

이번까지 연재를 하면서 여러 프로세서에 RTOS를 포팅하였는데 열심히 설명하려 했지만 이해가 잘 되었는지는 모르겠다. 실질적으로 긴 소스를 프린트해서 한 줄 한 줄 보면서 해야 하는 것이 정석인 것 같다.지금까지 여러 프로세서에 포팅하는 것에 중점을 두어서 설명하느라 간단한 예제 task만을 작성하여 설명하였다.

다음 호에서는 Network application을 작성하여 RTOS에서 동작시켜 봄으로써 좀더 난이도가 높은 실제 사용 예를 보여주려 한다. EL

### -- 참고 --

ARM사의 ARM Core 중 주목받고 있는 것들을 살펴보면 크게 ARM7, ARM9, ARM10계열이 있다. ARM7 계열에서는 ARM7TDMI, ARM7TDMI-S, ARM710T, ARM720T, ARM740T 등이 있다. 또한 Intel의 StrongARM도 이보다 약간 개선된 것이다.

모델명에 붙은 T는 Thumb를 뜻하는 것으로 메모리, 버스 등을 8,16Bit로 구성할 수 있는 명령어를 제공한다. 이로써 저렴한 시스템 구축이 가능해진다. 현재 ARM7TDMI가 시장의 주류를 형성하고 있고 곧 ARM9 Core를 사용하는 프로세서들이 등장할 것이다. ARM9부터는 전형적인 Harvard구조를 가지게 되며 주요 모델로는 ARM920T, ARM940T들이 있다. 차세대 400MIPS이상의 성능을 내는 ARM10 Thumb이 진행되고 있으며 현재 실제적으로 사용할 수 있는 급은 ARM7 계열이다.

몇 군데 주요 제조 업체들을 보면 Intel이 내놓는 StrongARM 계열에서 SA1100, SA1110, SA1111 들이 있으며 현재 추세는 SA1110이다. Cirrus, Atmel 등이 굴지의 전자 부품 회사들은 거의 ARM을 생산하거나 이용하고 있으며 국내에서도 Samsung, Hyundai에서 이를 제작하고 있다.

### -- 정보 --

-인텔 홈페이지
http://www.intel.com
-인텔 StrongArm SA-100 문서
http://developer.intel.com/design/strong/
-ARM 홈페이지
http://www.arm.com
-ARM SDT
Free Evaluation version Download
http://www.arm.com/sitearchitek/
devtools.ns4/html/dev_tools?OpenDocument
&style=Dev_Tools
-삼성 반도체 홈페이지
http://www.samsungsemo.com
-삼성 ks50100 board 문서, Source
http://www.intl.samsungsemo.com/
System_LSI/Communication/
Embedded_Controller/KS32C50100/
KS32C50100.htm