

---

## 4부. 임베디드 소프트웨어 설계

11장. 소프트웨어 개발 툴의 이해와 활용

12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

13장. 시스템 리셋과 부트코드

14장. 하드웨어 제어

15장. 부트로더 개발

# 교육 목표

---

- ❑ 임베디드 시스템 개발에 필요한 툴 사용법을 이해한다.
- ❑ 소프트웨어 동작을 위한 메모리 구조 설계 방법을 이해한다.
- ❑ 임베디드 소프트웨어 최적화 방법에 대하여 이해한다.
- ❑ 부트코드 작성 방법에 대하여 이해한다.
- ❑ 임베디드 하드웨어 제어 방법을 이해한다.
- ❑ 부트로더 개발 방법을 이해한다.

# 목 차

---

## □ 11장. 소프트웨어 개발 툴의 이해와 활용

01. 소프트웨어 개발 툴의 구성 및 사용법

02. 메모리 구조 설계

03. 다운로드와 디버깅

## 12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

## 13장. 시스템 리셋과 부트코드

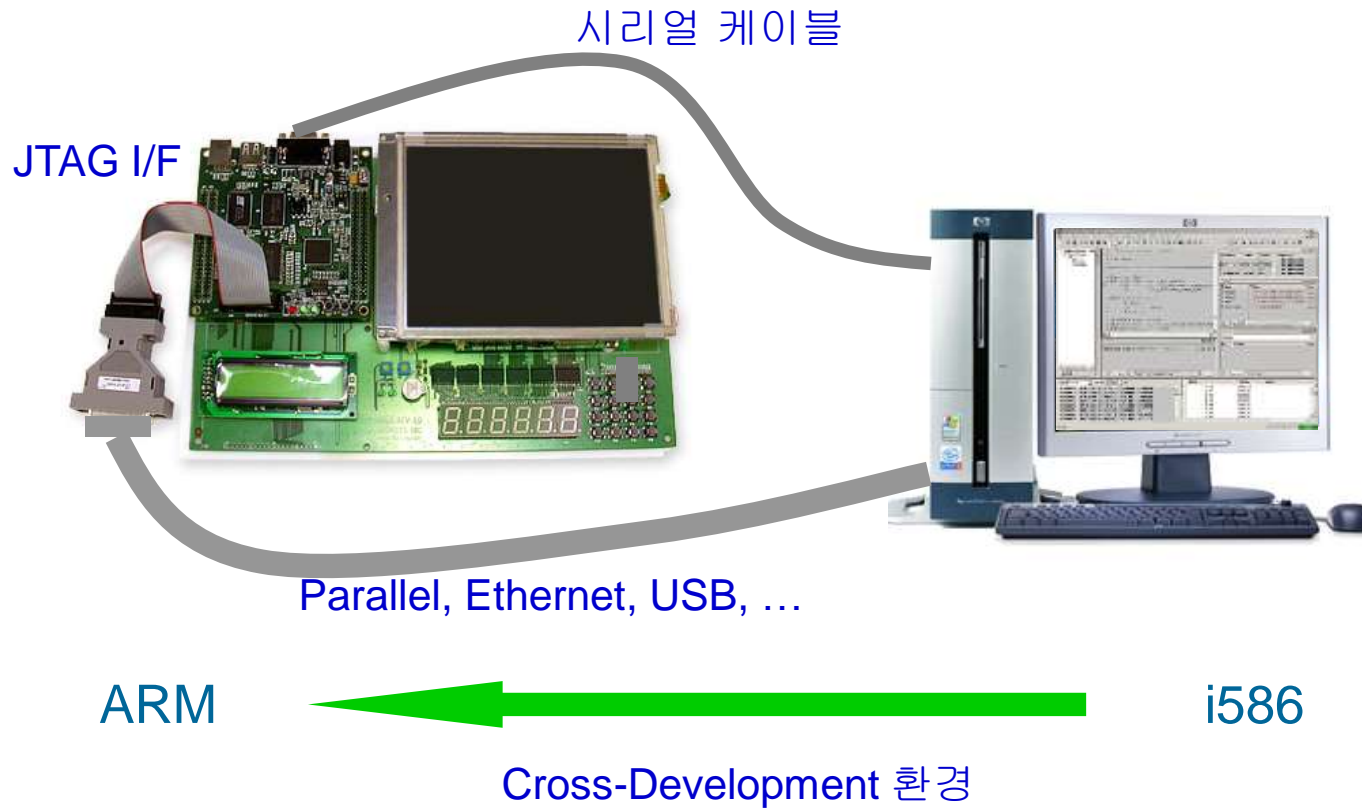
## 14장. 하드웨어 제어

## 15장. 부트로더 개발

# 임베디드 시스템 개발 환경

TARGET 시스템

HOST 시스템



# Target 시스템과 Host 시스템

---

## □ Target 시스템

- ❖ 개발하고자 하는 임베디드 시스템을 말한다.
- ❖ ARM 기반의 PDA 등

## □ Host 시스템

- ❖ Target 시스템을 개발하기 위한 개발환경을 제공하는 시스템
- ❖ 타겟을 위한 어셈블러, 컴파일러, 링커 등의 호스트 툴 과 타겟을 개발하는데 필요한 디버거를 제공한다.
- ❖ ARM 프로세서용 툴과 디버거를 가지고 있는 Pentium PC

# Host 개발 툴

---

## □ Cross-Compiler

- ❖ 개발된 소스 프로그램을 다른 machine의 기계어로 번역하는 프로그램
- ❖ 예를 들면 x86 계열의 호스트에서 개발한 소스 프로그램을 ARM 용 기계어로 번역하는 프로그램을 말한다.

## □ IDE(Integrated Development Environment)

- ❖ 컴파일러, 디버거 와 에디터 등이 통합된 CASE(Computer Aided Software Engineering) 툴을 말한다.

# 디버그 인터페이스

---

## □ ICE 장비

- ❖ In-Circuit Emulator (ICE)는 호스트의 디버거와 함께 Target 시스템의 레지스터나 메모리의 내용을 읽거나 변경할 수 있고, Break Point나 Watch Point를 설정할 수도 있고, 프로그램을 step-by-step으로 실행 할 수 있게 해주는 장치 이다.
- ❖ 호스트와는 Parallel, Ethernet 등을 이용하여 연결되고, 타겟과는 JTAG를 통하여 연결된다.

# ARM Cross 컴파일러

---

## □ ARM Cross 컴파일러

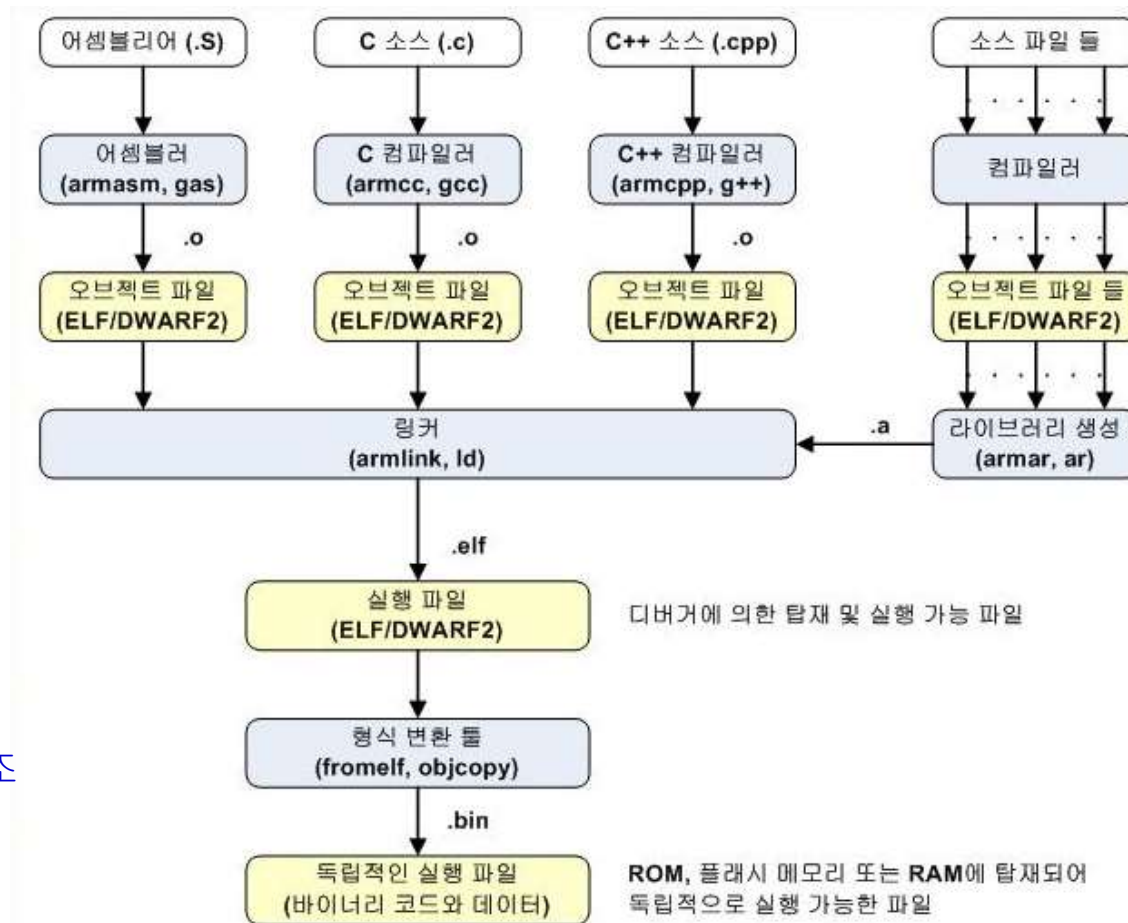
- ❖ PC의 x86 계열의 호스트에서 소스 프로그램을 32비트 RISC 프로세서인 ARM 용 기계어를 생성하는 프로그램

## □ ARM Cross 컴파일러의 종류

- ❖ SDT v2.11, v2.50, v2.51
  - 통합개발환경(IDE) : APM
  - 디버거 : ADW
- ❖ ADS(ARM Developer Suit) v1.01, v1.1, v1.2
  - 통합개발환경(IDE) : CodeWarrior
  - 디버거 : AXD
- ❖ RVDS(RealView Developer Suit) v1.2, v2.0
  - IDE & 디버거 : RVD
- ❖ Code Warrior for ARM v1.2, v2.0
  - IDE & 디버거 : CodeWarrior
- ❖ GNU 교차 툴 – 페이지 299, [표 11-2] 참조
  - 어셈블러 & 컴파일러 : gas, gcc
  - 디버거 : GDB, Insight 디버거



# ARM 교차툴의 활용



페이지 300

[그림 11-1] 참조

# 컴파일러의 레지스터 사용

---

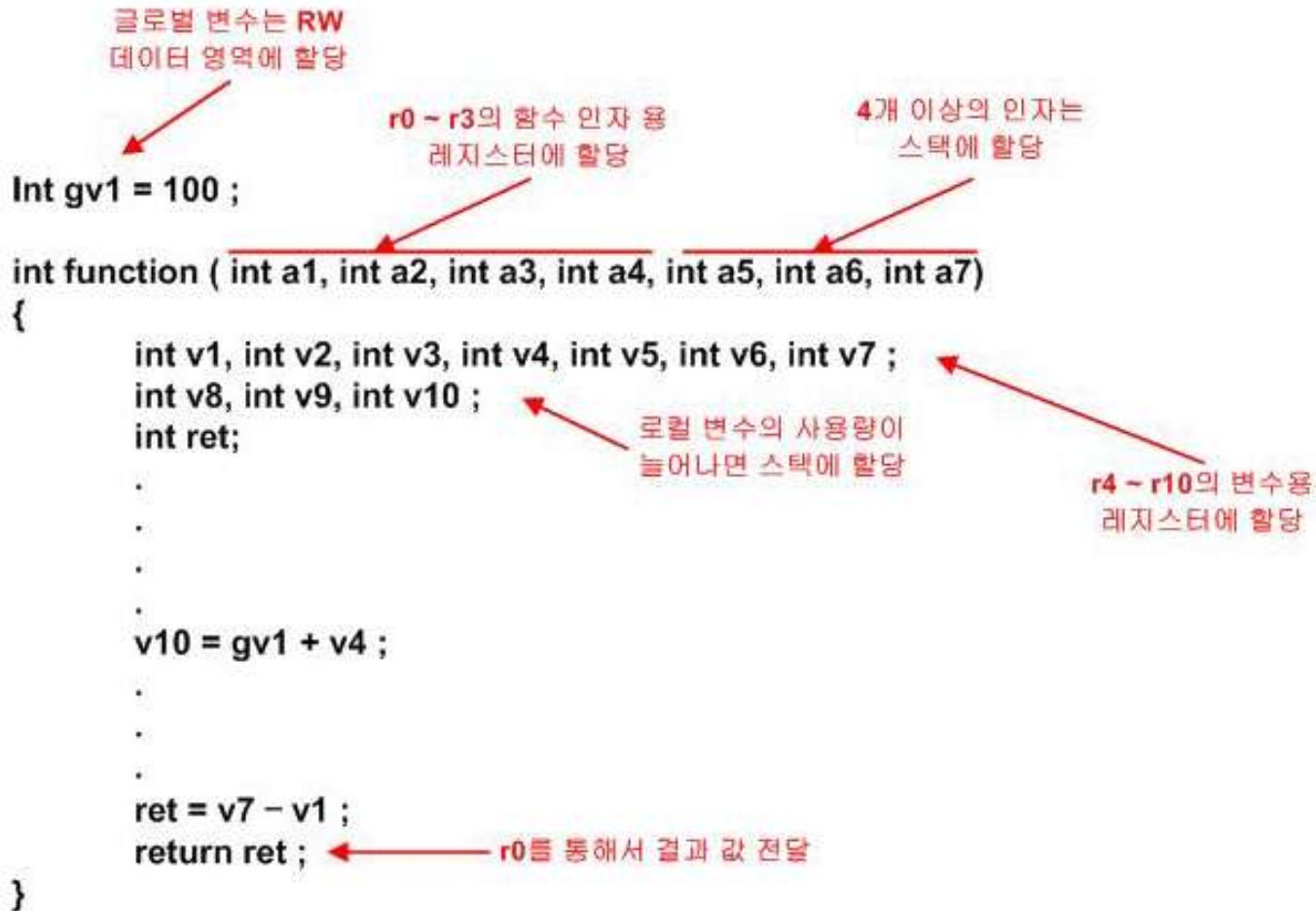
## □ ATPCS or APCS

- ❖ ARM Thumb Procedure Call Standard
- ❖ ARM Procedure Call Standard
- ❖ 컴파일러와 어셈블러에서 레지스터를 사용되는 방법에 대해서 정의
  - Function Argument 사용
  - Result passing(return)
  - Stack 사용
  - General purpose 레지스터의 사용

# APCS

레지스터	APCS	역 할
r0	a1	argumect 1 / interger result / scratch register
r1	a2	argumect 2 / scratch register
r2	a3	argumect 3 / scratch register
r3	a4	argumect 4 /scratch register
r4	v1	register variable 1
r5	v2	register variable 2
r6	v3	register variable 3
r7	v4	register variable 4
r8	v5	register variable 5
r9	sb/v6	Static base / register variable 6
r10	sl/v7	stack limit / register variable 7
r11	fp	frame pointer
r12	ip	scratch reg. / new sb in inter-link-unit calls
r13	sp	Lower end of current stack frame
r14	lr	link address / scratch register
r15	pc	program counter

# APCS 이해

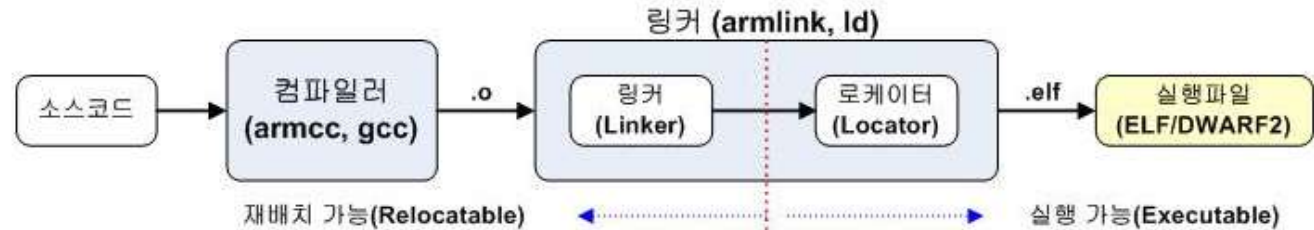


# ARM 컴파일러의 데이터 Type

---

<b>char</b>	<b>8 bit</b>	<b>Byte</b>
<b>short</b>	<b>16 bit</b>	<b>Half-word</b>
<b>int</b>	<b>32 bit</b>	<b>Word</b>
<b>long</b>	<b>32 bit</b>	<b>Integer</b>
<b>float</b>	<b>32 bit</b>	<b>IEEE single-precision</b>
<b>double</b>	<b>64 bit</b>	<b>IEEE double-precision</b>
<b>pointer</b>	<b>32 bit</b>	
<b>long long</b>	<b>64 bit</b>	<b>64 bit integer</b>

# 오브젝트 파일의 자료 구조

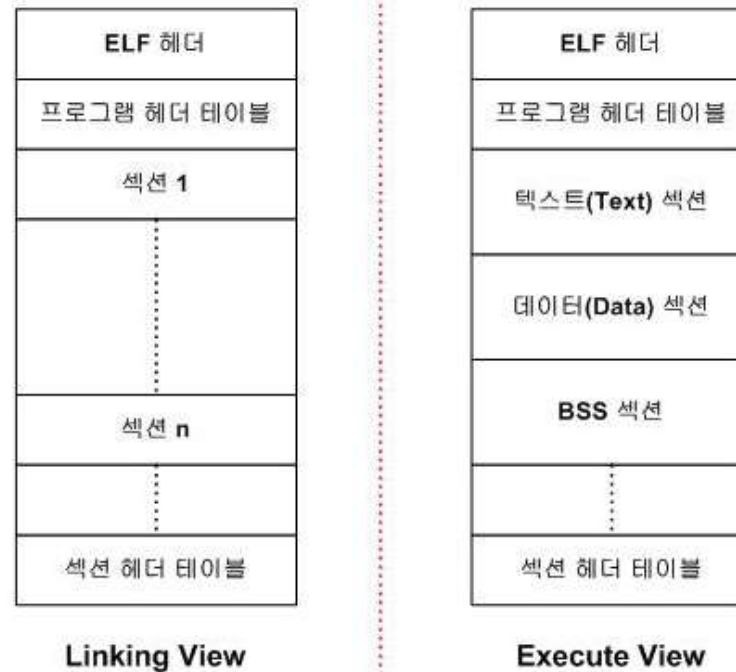


## □ 오브젝트 파일의 종류

- ❖ 재배치 가능한 오브젝트 파일 (relocatable file)
- ❖ 실행 가능한 오브젝트 파일 (executable file)
- ❖ 공유 오브젝트 파일 (shared object file)

## □ 오브젝트 파일의 형식

- ❖ COFF(Common Object File Format)
- ❖ ELF(Executable Linkable Format)



# 실행 가능한 오브젝트 파일의 구성

## □ Object 파일의 자료 구조

- ❖ Header
  - Section을 설명하는 내용
- ❖ Section
  - Text section : 모든 code block
  - Data section : 초기화된 전역변수 및 초기값
  - BSS section : 초기화 되지 않은 전역변수
- ❖ Symbol Table
  - 변수와 함수의 이름과 위치 정보를 가진다.

ELF 헤더
프로그램 헤더 테이블
텍스트(Text) 섹션
데이터(Data) 섹션
BSS 섹션
심볼(Symbol) 테이블
문자열(String) 테이블
섹션 이름 문자열 테이블
디버그 테이블 (ASD / DWARF1.0 / DWARF2.0)
섹션 헤더 테이블

# 링커 (Linker)

---

## □ Linker

- ❖ 불완전한 **object** 파일들을 합쳐 모든 코드와 데이터를 포함하는 새로운 **object** 파일을 생성해 내는 도구
- ❖ 필요에 따라 라이브러리와 초기화(**startup**) 파일을 같이 링크 한다.

## □ Startup 코드

- ❖ 어셈블러로 구성되며 모든 프로그램에는 반드시 필요
- ❖ 일반적으로 **crt0.S**(C-runtime 0 의 약자), **init.S**, **startup.S** 등의 이름을 많이 사용한다.
- ❖ **Startup 코드 동작**
  - ① 시스템 설정
  - ② **Data** 초기화
  - ③ **Stack** 영역 할당
  - ④ **Heap** 영역 할당
  - ⑤ 메인 함수(**main**)를 호출



# 로케이트 (Locate)

## □ 로케이트 (Locate)

- ❖ 메모리에서 실행 가능하도록 코드와 데이터를 배치하여 최종 바이너리 이미지를 생성하는 도구
- ❖ 대부분 링커에 포함되어 있다

## □ Linker script 파일

- ❖ 코드와 데이터의 메모리 배치를 정의한 파일

```
# ROM located at address 0x0
# RAM located at address 0x800000
# alignment directives have been removed for clarity

MEMORY {
    TEXT (RX) : ORIGIN = 0x0, LENGTH = 0
    DATA (RW) : ORIGIN = 0x800000, LENGTH = 0
}

SECTIONS{
    .main :
    {
        *(.text)
        *(.rodata)
    } > TEXT
```

```
# Locate the initialized data to the ROM area at the end of .main.

.main_data : AT( ADDR(.main) + SIZEOF(.main) )
{
    *(.data)
    *(.sdata)
    *(.sbss)
} > DATA

.uninitialized_data:
{
    *(SCOMMON)
    *(.bss)
    *(COMMON)
} >> DATA
```

# 목 차

---

## □ 11장. 소프트웨어 개발 툴의 이해와 활용

01. 소프트웨어 개발 툴의 구성 및 사용법

02. 메모리 구조 설계

03. 다운로드와 디버깅

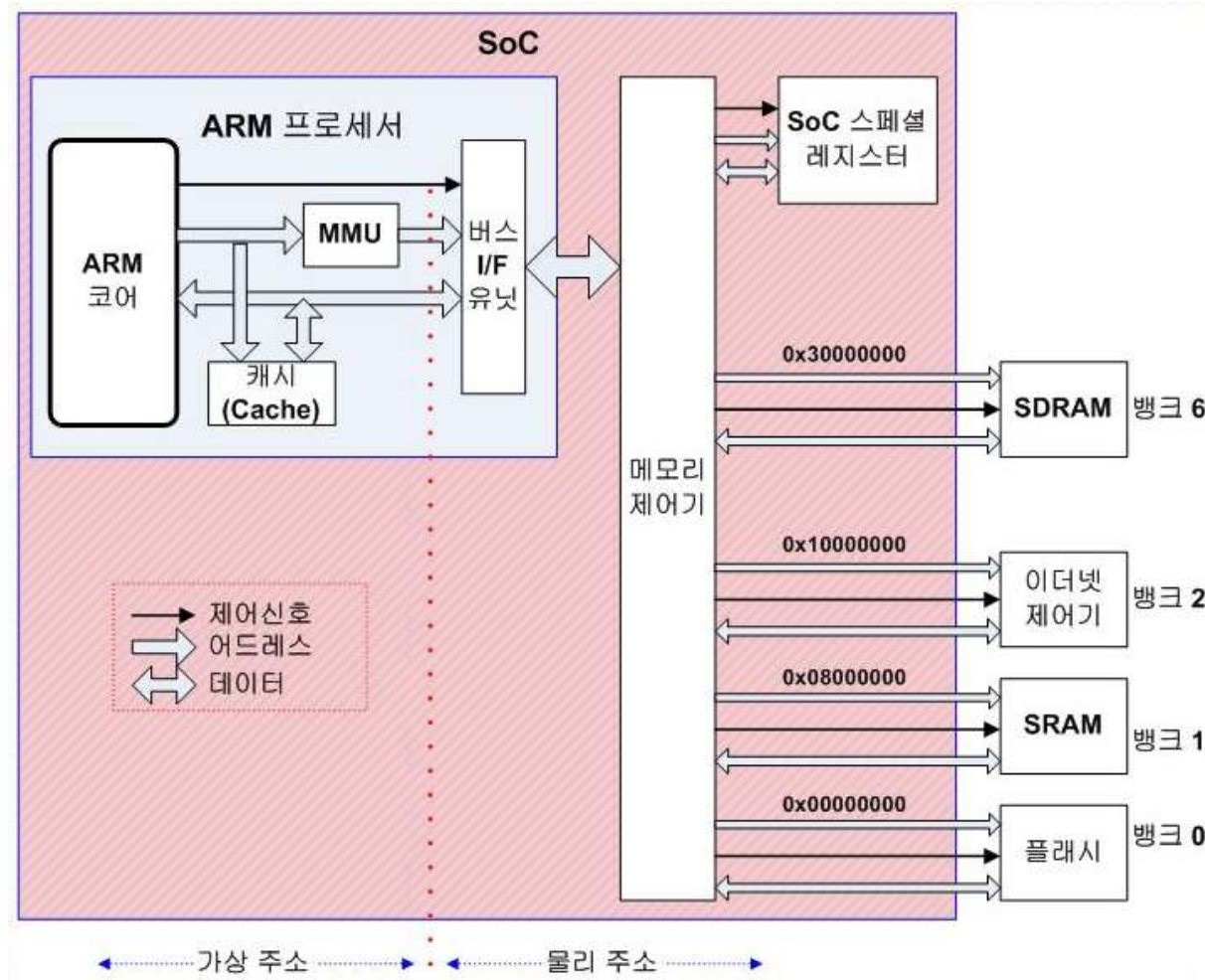
## 12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

## 13장. 시스템 리셋과 부트코드

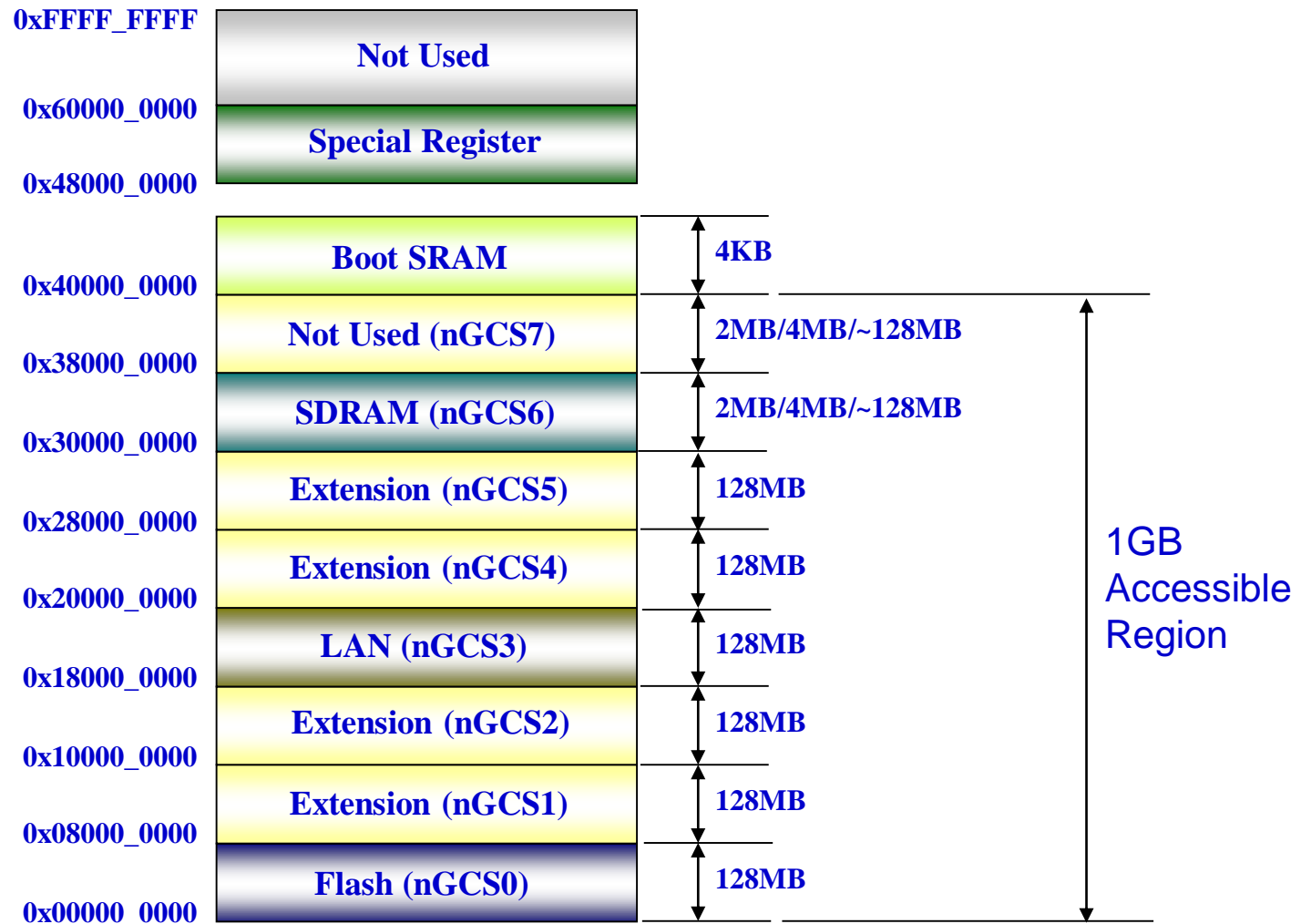
## 14장. 하드웨어 제어

## 15장. 부트로더 개발

# 메모리 제어기와 물리 메모리 할당



# 물리적인 메모리 구조



# 메모리 리매핑(Re-mapping)

---

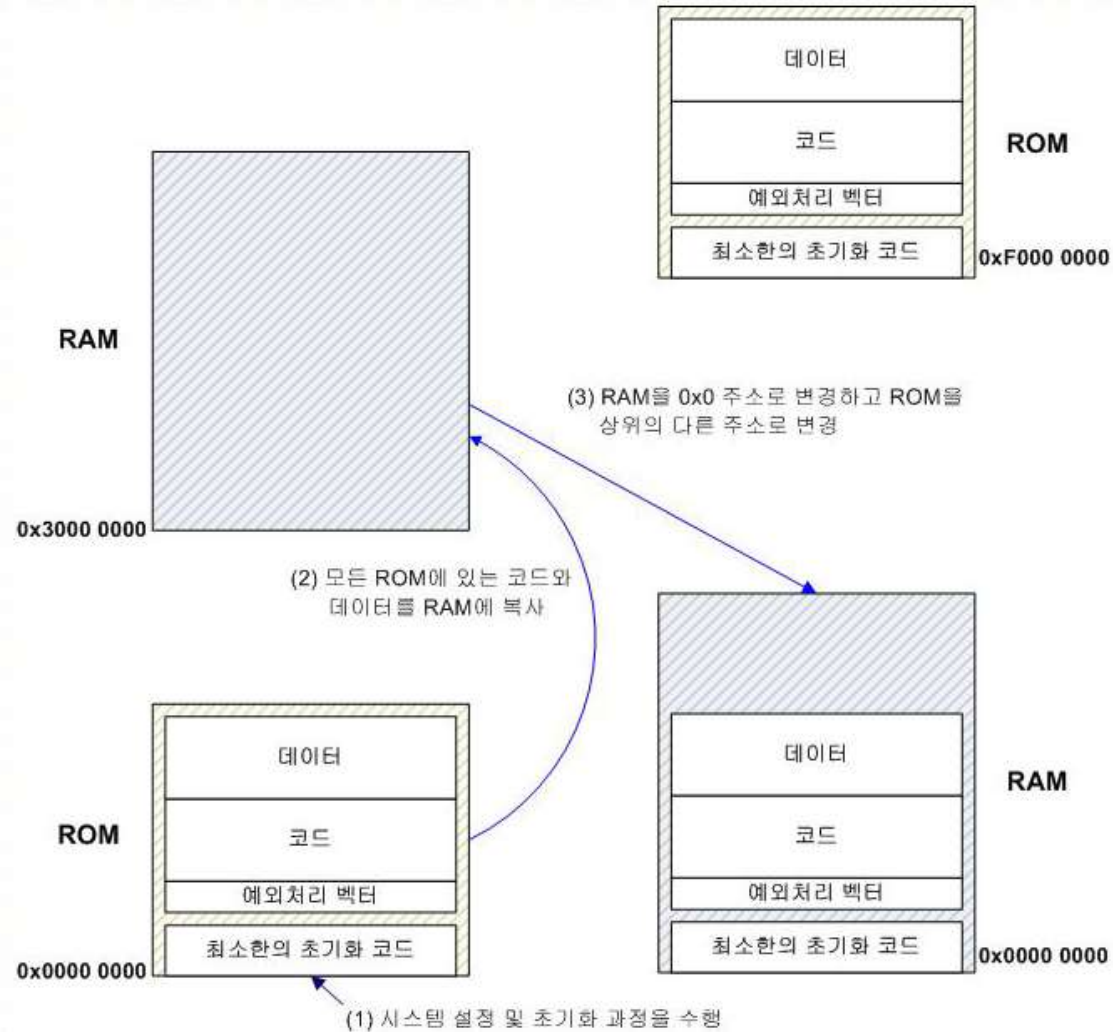
## □ 하드웨어 적으로 메모리의 어드레스 **MAP**을 바꾸는 동작

- ❖ 메모리 컨트롤러에서 지원 되어야 한다.
- ❖ 참고로 실습용으로 사용하는 S3C2410은 Re-mapping을 지원하지 않는다.

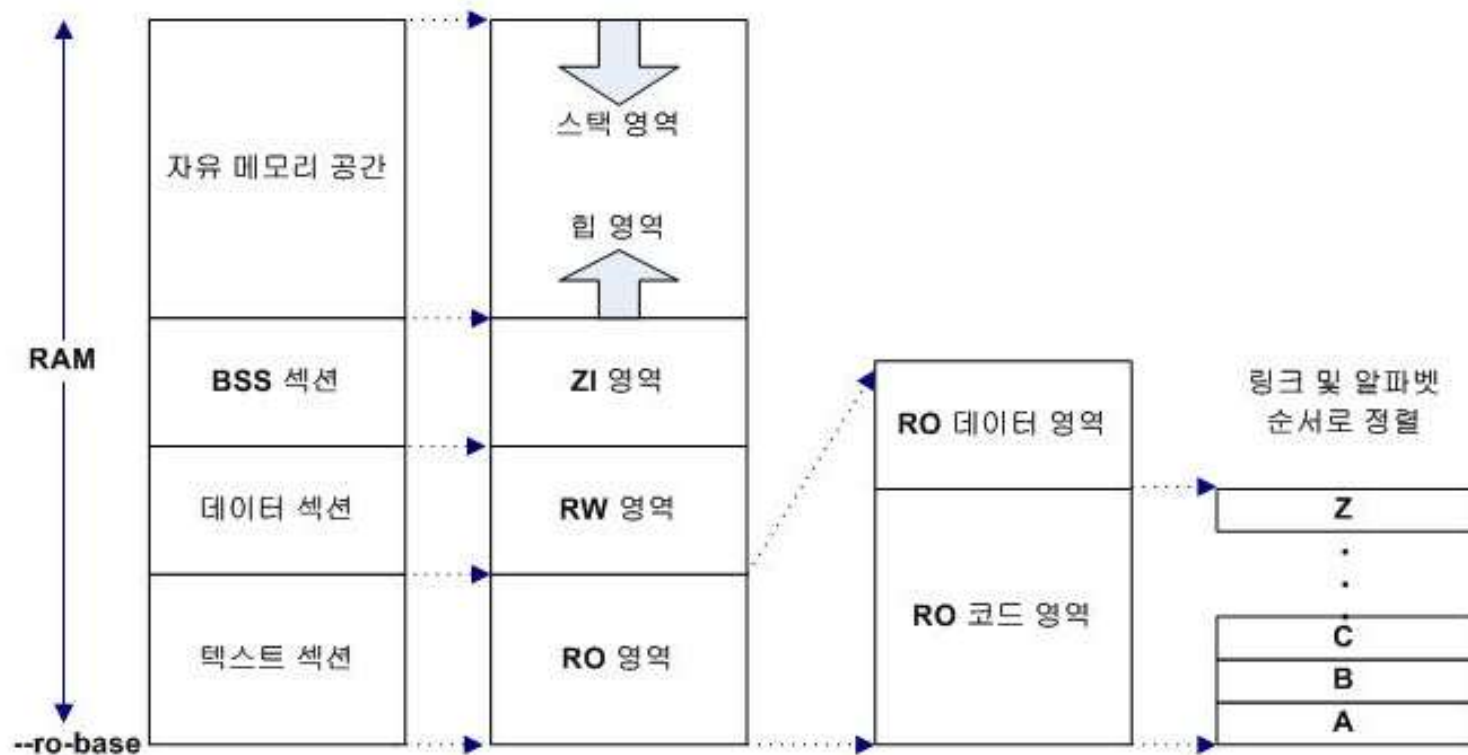
## □ Re-mapping이 필요한 이유

- ❖ ROM이나 Flash 만이 전원이 꺼진 후에도 code나 data가 저장되어 있다.
  - 대부분 0x0 번지에 ROM이나 Flash가 위치한다.
  - ARM의 경우 Reset이 발생하면 무조건 0x0 번지의 명령을 수행한다.
- ❖ ROM 이나 Flash는 DRAM에 비하여 data 버스의 width도 좁고, 속도가 느려서 시스템 동작 중에 성능을 저하 시킬 수 있다.
- ❖ ARM의 경우 Exception Vector Table은 항상 0x0 번지에 있는데, ROM 이나 Flash를 0x0 번지 위치에서 계속 사용하게 되면 Vector Table을 수정하기가 용이하지 않다.
  - 대부분의 OS의 경우 Vector Table을 초기화 중에 다시 설정한다.

# 메모리 리매핑 과정



# 소프트웨어 동작을 위한 메모리 구조 1

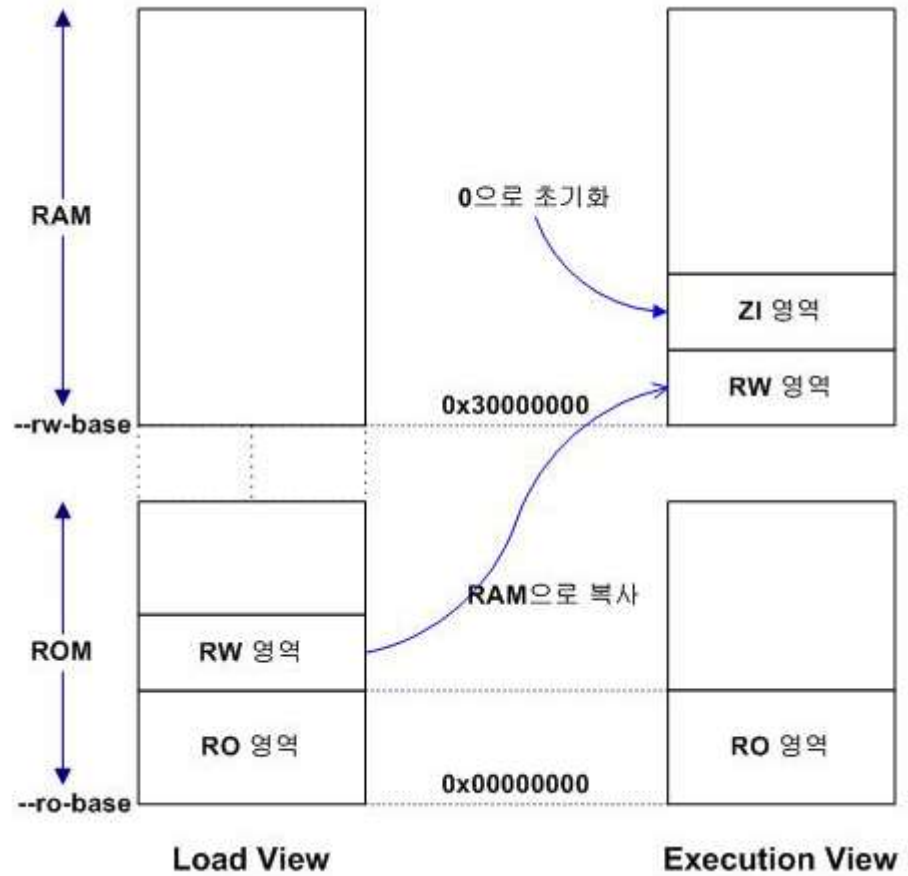


# 소프트웨어 동작을 위한 메모리 구조 2

## 스캐터 로딩 (Scatter Loading)



- 실제 동작에 앞서 프로그램이나 데이터를 재배치
- 메모리를 분산해서 사용
- 로드뷰와 실행뷰가 서로 다르다.





# 목 차

---

## □ 11장. 소프트웨어 개발 툴의 이해와 활용

01. 소프트웨어 개발 툴의 구성 및 사용법

02. 메모리 구조 설계

03. 다운로드와 디버깅

## 12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

## 13장. 시스템 리셋과 부트코드

## 14장. 하드웨어 제어

## 15장. 부트로더 개발

# 다운로드와 디버깅

---

## □ 다운로드

- ❖ 생성된 실행 가능한 바이너리 이미지를 타겟 보드의 메모리에 탑재하는 동작
- ❖ 다운로드 및 실행 방법
  - ROM / Flash에 탑재하여 실행
  - 전용 ICE 없이 DRAM에 탑재하여 실행
  - 전용 ICE를 사용하여 DRAM 탑재하여 실행

## □ 디버깅

- ❖ 타겟 시스템을 실행하면서 프로그램의 실행 상태, 메모리, 변수 등을 프로그래머가 확인하거나 제어 하면서 오류를 찾아 수정하는 동작
- ❖ 실행 가능한 이미지가 반드시 디버깅 정보와 함께 메모리에 탑재 되어 있어야 한다.
- ❖ 전용 디버깅 용 ICE 장비와 디버거 소프트웨어를 사용한다

# 다운로드 방법

---

## ❑ ROM / Flash에 탑재하여 실행

### ❖ ROM Writer를 사용하는 방법

- 전용 ROM writer를 사용하여 ROM을 programming하고 타겟 보드에 삽입

### ❖ JTAG 동글(Dongle)을 이용하는 방법

## ❑ 전용 ICE 없이 DRAM에 탑재하여 실행

### ❖ Target에 모니터 프로그램, 또는 부트로더를 탑재하고, Serial, ethernet, USB 등을 이용하여 다운로드

### ❖ JTAG Dongle을 이용하는 방법

### ❖ 전용 ICE를 이용하는 방법

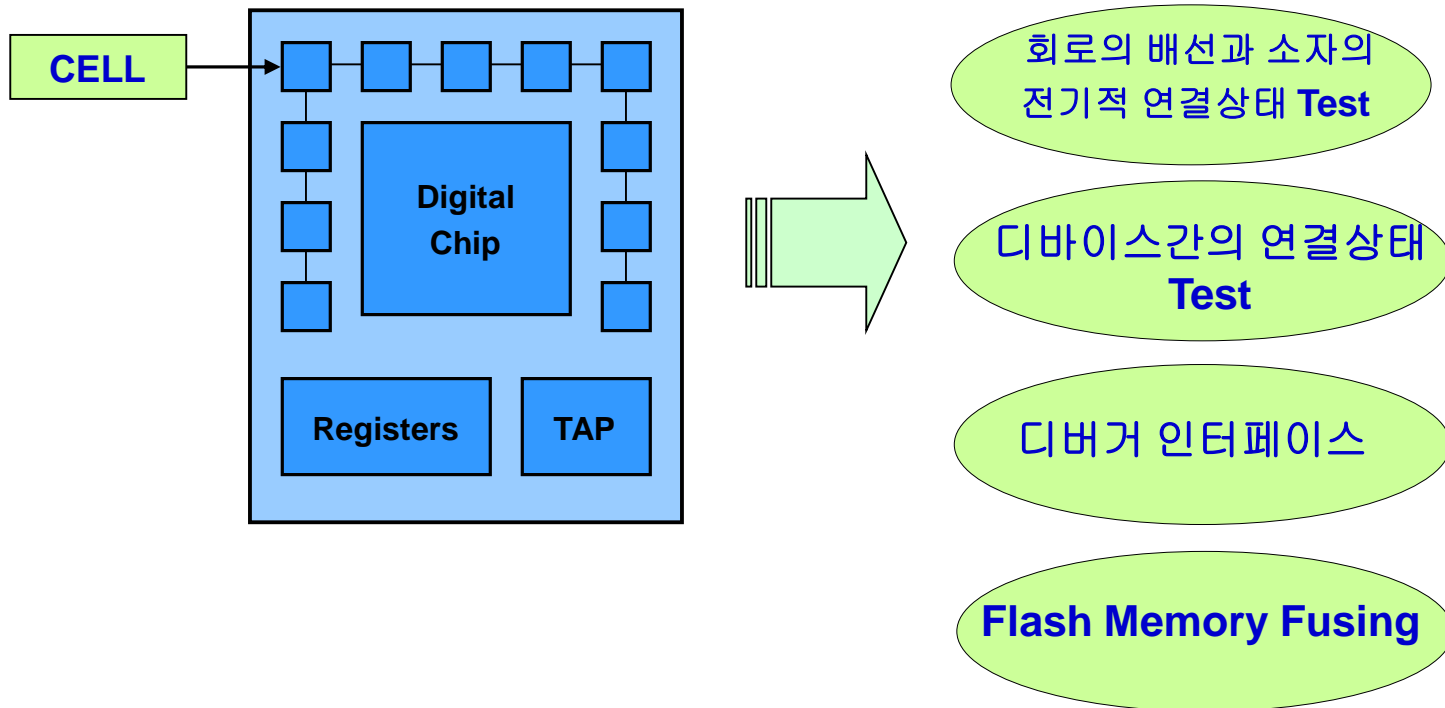
## ❑ 전용 ICE를 사용하여 DRAM 탑재하여 실행

### ❖ 디버깅을 할 때 매우 편리하다.

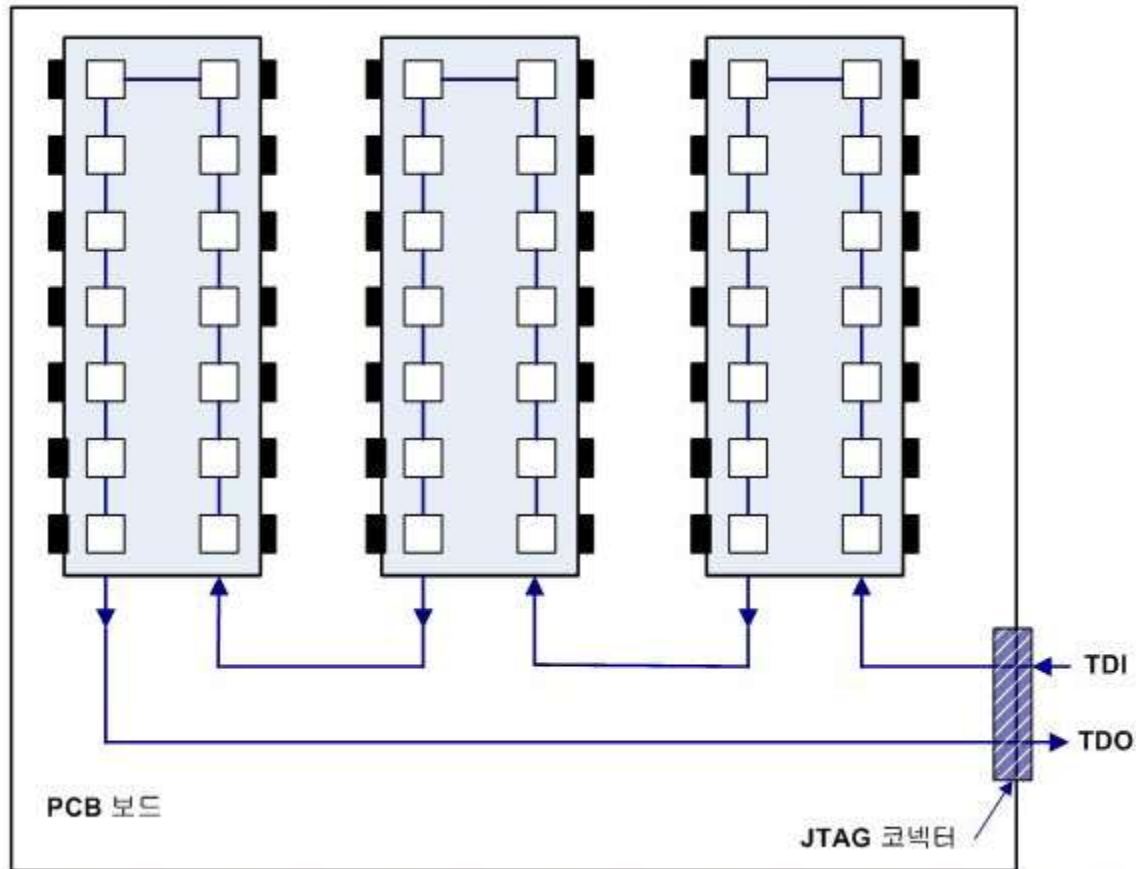
# JTAG(Joint Test Action Group)

## □ Boundary-Scan Test Interface JTAG Diagram

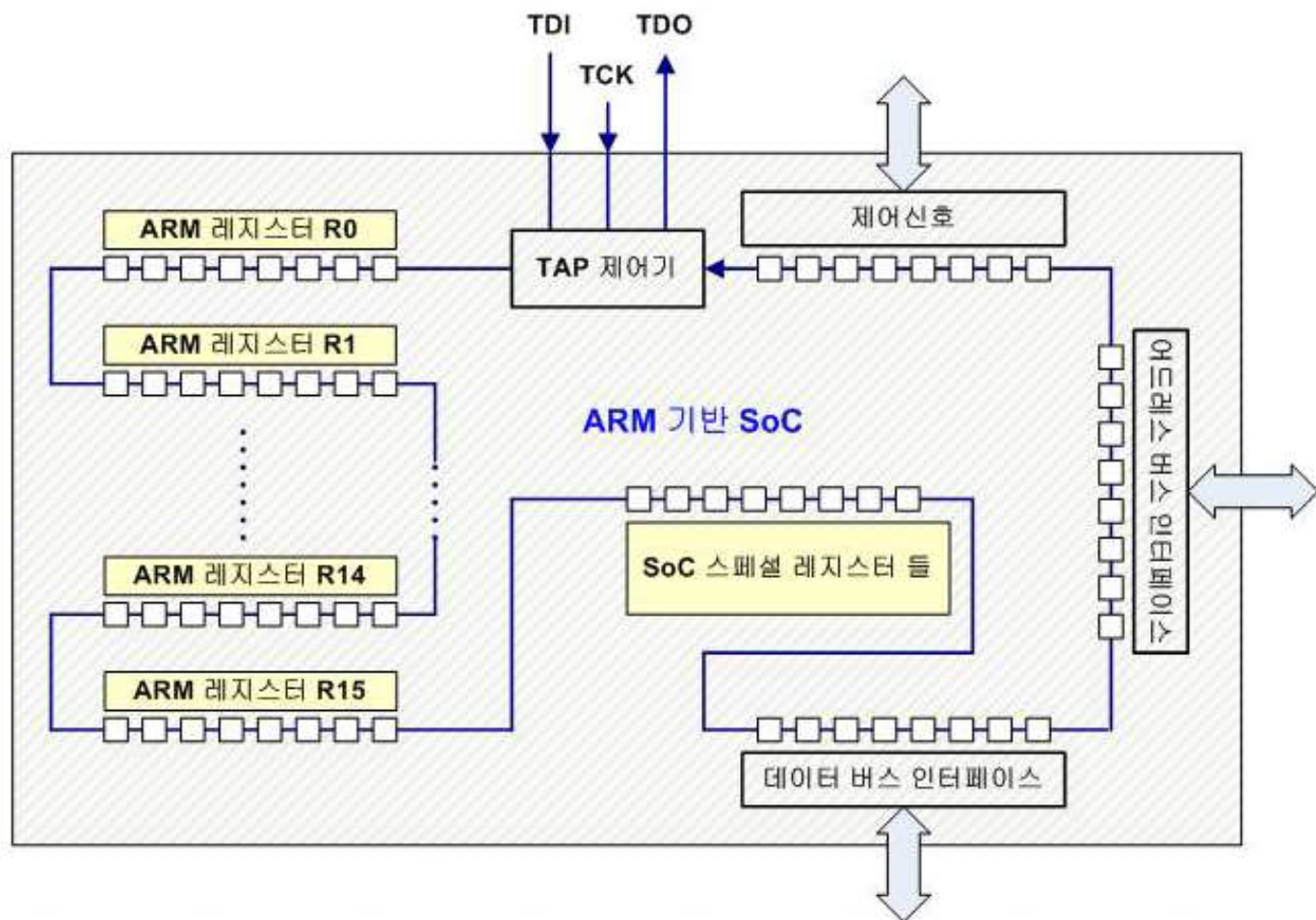
각각의 JTAG cell은 서로 연결되어 있다.



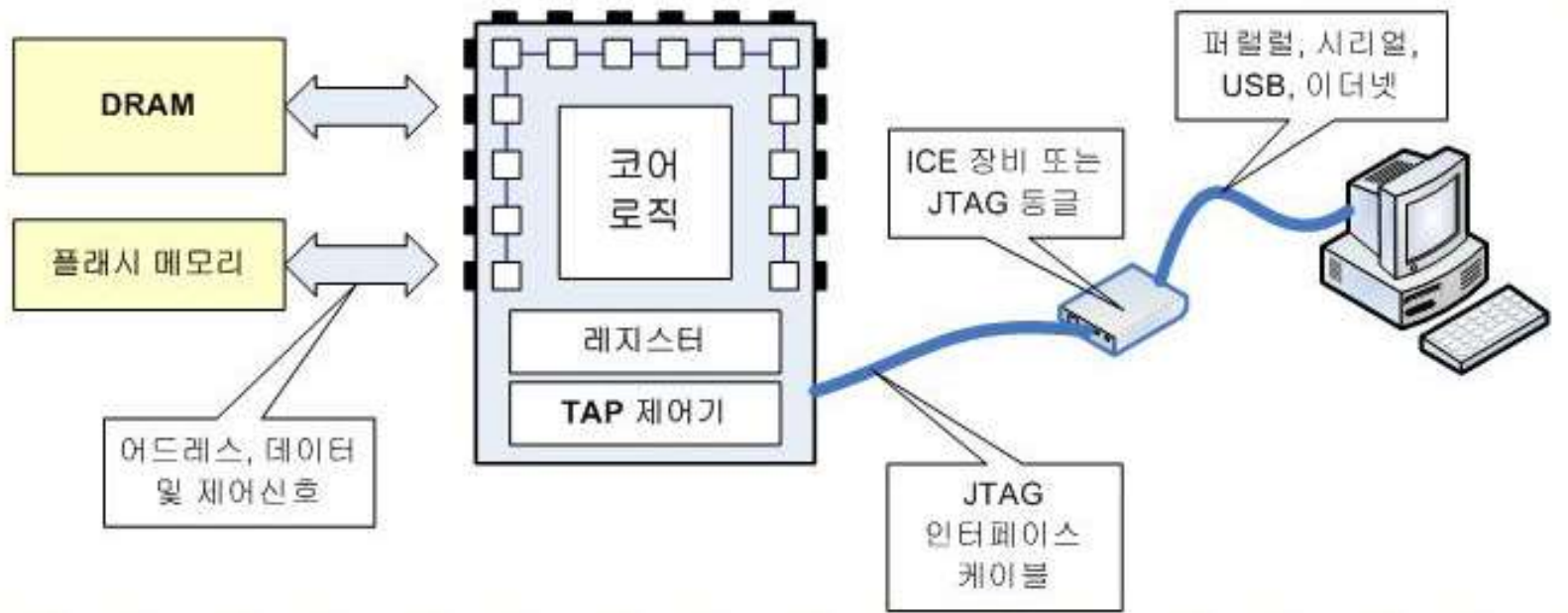
# JTAG을 이용한 테스트 루프 구성



# JTAG 기반의 칩과 버스 제어



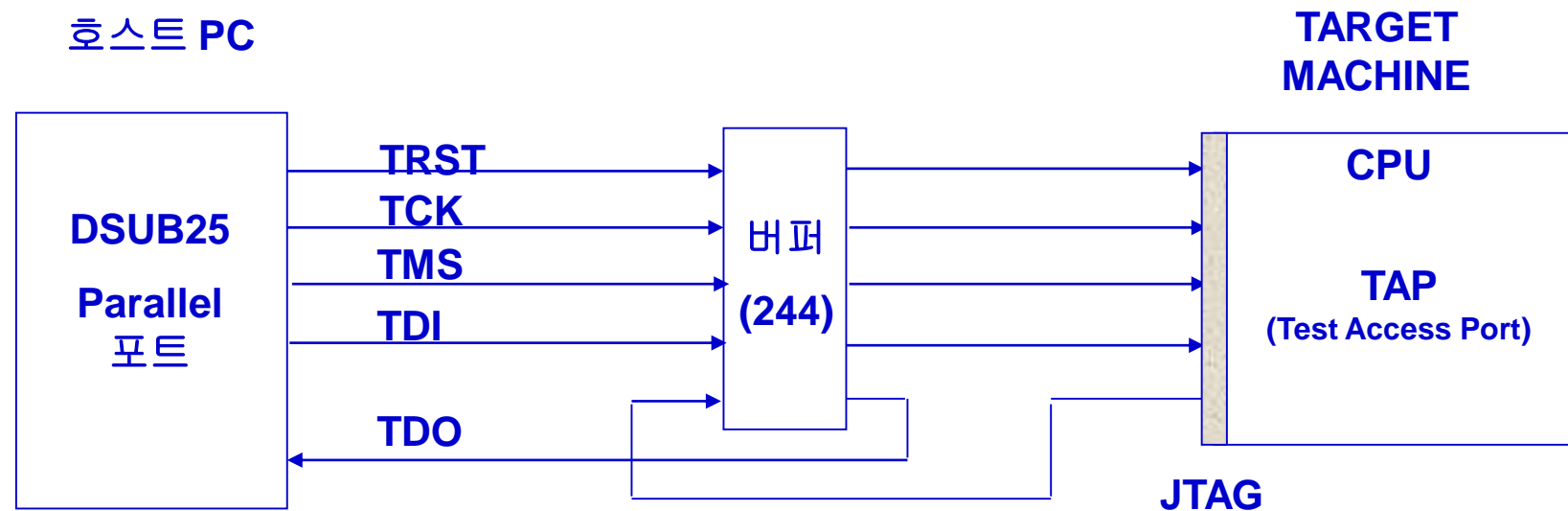
# JTAG 기반의 디버깅 시스템 구성



# JTAG 동글

## □ PC의 Parallel 포트와 JTAG을 연결하여

- ❖ 프로토콜을 변환하고
- ❖ 타겟에 맞게 전원을 변환해 주며
- ❖ 타겟을 JTAG 프로토콜에 의해 제어할 수 있도록 한다.





# 목 차

---

## 11장. 소프트웨어 개발 툴의 이해와 활용

### □ 12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

01. 컴파일러 사용과 옵션 설정

02. 임베디드 C의 구성요소와 프로그램 최적화

03. 나눗셈과 나머지 연산

04. 메모리 참조와 포인터

05. ARM/Thumb 인터워킹

## 13장. 시스템 리셋과 부트코드

## 14장. 하드웨어 제어

## 15장. 부트로더 개발

# 컴파일러 사용과 옵션 설정

---

## □ 컴파일러 사용

ADS/RVCT : armcc [option] file1 file2

GNU : arm-linux-gcc [option] file1 file2

## □ **ARM** 프로세서와 아키텍처 옵션

### ❖ 프로세서 옵션

-proc arm920t 또는 --cpu arm920t

### ❖ 아키텍처 옵션

-proc arm920t 또는 --cpu arm920t

## □ 디버깅 옵션

-g 또는 --debug

# 컴파일러의 최적화 기능

---

## □ 컴파일러의 Optimization Level

### ❖ -O0

- Optimization을 하지 않는다. 따라서 불필요한 code 존재
- Debugging하는 경우에 적합한 optimization 방법이다.
- 소스레벨 디버깅을 하기 위해 '-g' 옵션을 사용하면 optimization level의 선언에 무관하게 O0 레벨이 된다.

### ❖ -O1

- O0와 O2의 중간 단계의 Optimization 수행

### ❖ -O2

- 컴파일러의 default optimization level이다.
- Full optimization을 제공
- Code density가 좋다.

## □ 컴파일러 Optimization 제어

- ❖ Space 최적화 : “-Ospace” 사용, default
- ❖ 실행 speed 최적화 : “-Otime” 사용

# 목 차

---

## 11장. 소프트웨어 개발 툴의 이해와 활용

### □ 12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

01. 컴파일러 사용과 옵션 설정

02. 임베디드 C의 구성요소와 프로그램 최적화

03. 나눗셈과 나머지 연산

04. 메모리 참조와 포인터

05. ARM/Thumb 인터워킹

## 13장. 시스템 리셋과 부트코드

## 14장. 하드웨어 제어

## 15장. 부트로더 개발

# C 프로그램 구성 요소

변수	글로벌 변수	
	로컬 변수	
함수	문장들	
	조건부 분기	if, switch
	무조건 분기	goto
	루프	for, while, do ~ while
	루프 탈출	break, continue
	탈출	break, continue, return
	수식문	
	Null 문	
	복합문	{ ... }

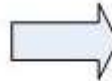
# 변수 사용과 프로그램 최적화

종류	지시자	저장 위치	유효 사용 범위	특징
로컬 (local)	미지정	레지스터, 스택	선언된 블록 ({ ... }) 내에서만 유효	
	<b>register</b>	레지스터		
글로벌 (global)	미지정	메모리	자신의 파일을 포함한 여러 파일에서 유효	초기화된 변수는 <b>RW</b> 영역에, 초기화되지 않는 변수는 <b>BSS</b> 영역에 할당
	<b>Extern</b>	메모리		
	<b>Static</b>	메모리	자신의 파일 내에서만 유효	

# 로컬 변수와 변수 크기

워드(Word) 크기

```
int wordfunc ( void )  
{  
    int a;  
    return a+1;  
}
```



```
wordfunc  
    mov    r3, #10  
    add    r3, r3, #1  
    mov    r0, r3  
    mov    pc, lr
```

하프 워드(Half-word) 크기

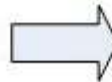
```
short shortfunc ( void )  
{  
    short a;  
    return a+1;  
}
```



```
shortfunc  
    mov    r3, #10  
    add    r3, r3, #1  
    mov    r3, r3, lsl #16  
    mov    r3, r3, asr #16  
    mov    r0, r3  
    mov    pc, lr
```

바이트(Byte) 크기

```
char bytefunc ( void )  
{  
    char a;  
    return a+1;  
}
```

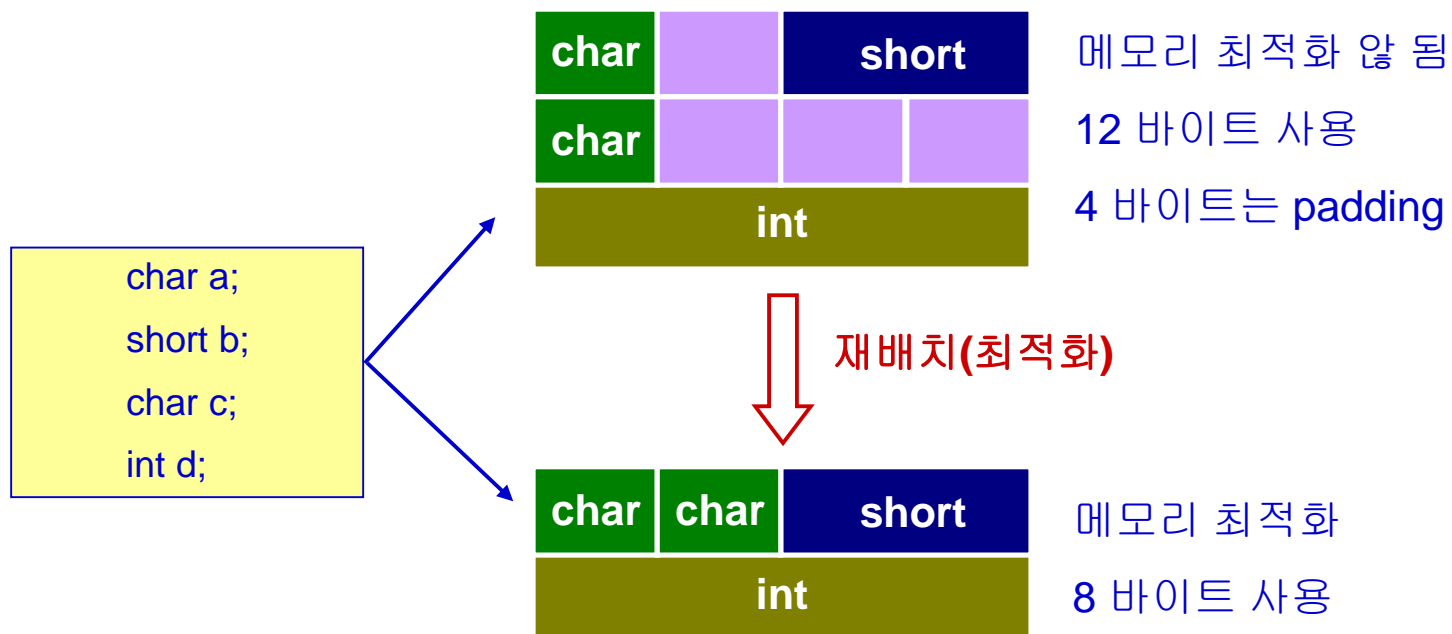


```
bytefunc  
    mov    r3, #10  
    add    r3, r3, #1  
    and    r3, r3, #255  
    mov    r0, r3  
    mov    pc, lr
```

# Global, Static 변수의 메모리 할당과 패딩

## □ Global 또는 static 변수

- ❖ 메모리에 설정, load/store 명령을 이용하여 데이터 전송
- ❖ Align되어 변수가 할당된다.
- ❖ 컴파일러에서는 메모리 사용을 최소화 하도록 변수를 재배치






# 변수의 packing (1)

□ 통신,미디어 프로토콜 등에서는 정해진 데이터 구조 형성

❖ IP 프로토콜의 예

0	15			16	31
Version (4bit)	Header leng. (4)	Type of service (8bit)	Total length(in bytes)		
Identification			Flag (3bit)	Fragment offset (13bit)	
Time to live (8bit)		Protocol (8bit)	Header checksum		
					

□ 데이터를 **packing** 하여야 원하는 데이터 구조를 유지할 수 있다.

❖ “\_\_packed”를 사용하여 구현

# 변수의 packing (2)

## □ 변수 또는 structure의 Packing

```
__packed char a;  
__packed short b;  
__packed char c;  
__packed int d;
```

or

```
__packed struct packed_str {  
    char a;  
    short b;  
    char c;  
    int d;  
}
```



데이터 구조가 packing 됨

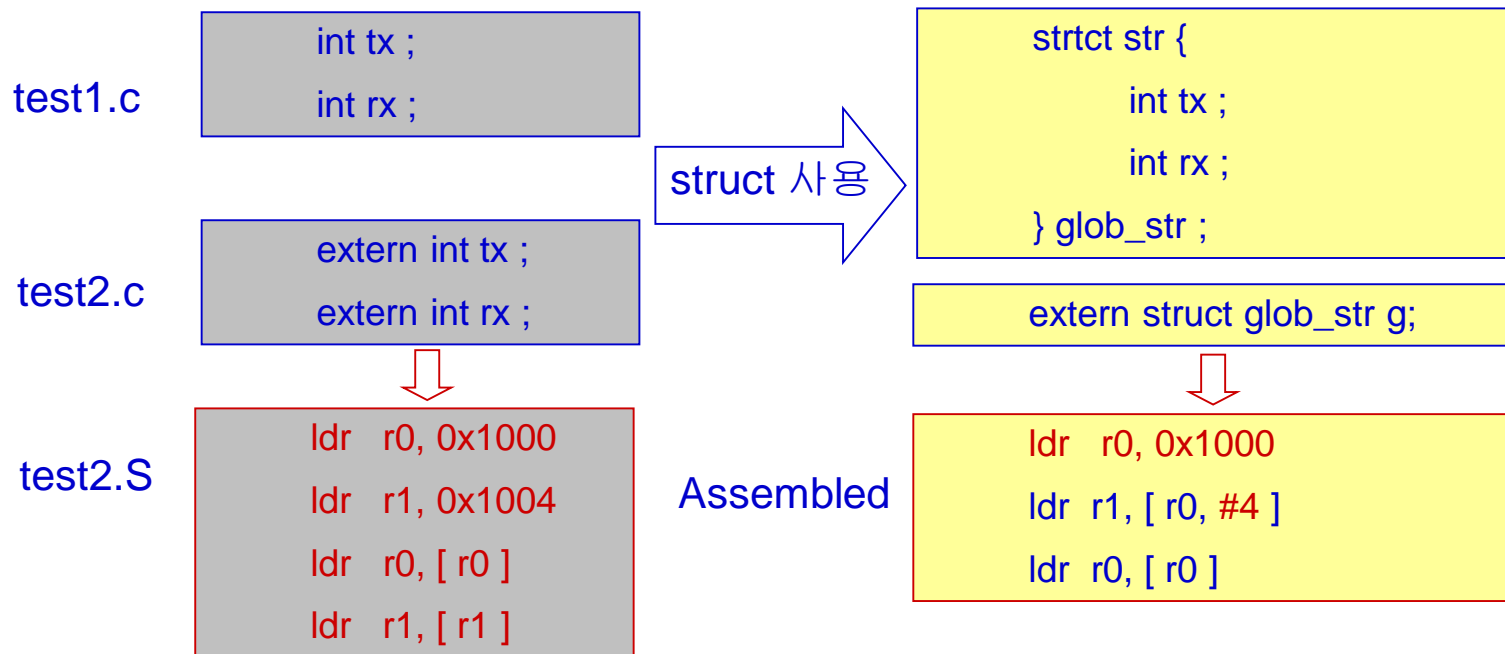
주의 : 변수를 **packing**하면 모든 액세스는 **byte** 단위로 이루어 진다.  
따라서 **structure** 전체를 **packing** 하면 성능을 감소 시킬 수 있으므로 필요한 변수 단위로 **packing** 하는 것이 성능을 향상 시키는데 유리한다.

# 외부 변수와 Base Pointer

## □ Base pointer를 이용한 Global 변수 선언

### ❖ Structure 형태로 선언

- 하나의 base pointer 만을 load하고 다른 변수는 load한 pointer와 offset에 의하여 변수를 load/store
- Global 변수를 사용할 때 성능 향상 -> Base pointer optimization



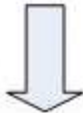
# 변수 형식 지시자 **volatile**

변수 형식 지시자 **volatile**를 사용하지 않은 경우

```
void func(void)
{
    /* 메모리 맵드 입출력 장치 주소 */
    unsigned long *port = (unsigned long *)0x10000;
    unsigned long value=0x12; /* 포트에 저장할 값, r2 */

    /* 입출력 포트 액세스 */
    *port = value; /* 포트 주소에 데이터 write */
    value = *port; /* 포트 주소에서 데이터 read */
}
```

컴파일  
(최적화 레벨 -O2)



변수 port : r3  
변수 value : r2

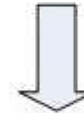
```
func
    mov r3, #65536 @ 주소 0x10000
    mov r2, #18    @ value 변수 0x12
    str r2, [r3]   @ 데이터 write
    mov pc, lr     @ return
```

변수 형식 지시자 **volatile**를 사용한 경우

```
void func(void)
{
    /* 메모리 맵드 입출력 장치 주소 */
    volatile unsigned long *port = (unsigned long *)0x10000;
    unsigned long value=0x12; /* 포트에 저장할 값, r2 */

    /* 입출력 포트 액세스 */
    *port = value; /* 포트 주소에 데이터 write */
    value = *port; /* 포트 주소에서 데이터 read */
}
```

컴파일  
(최적화 레벨 -O2)



변수 port : r3  
변수 value : r2

```
func
    mov r3, #65536 @ 주소 0x10000
    mov r2, #18    @ value 변수 0x12
    str r2, [r3]   @ 데이터 write
    ldr r2, [r3]   @ 데이터 read
    mov pc, lr     @ return
```

# 함수 사용과 프로그램 최적화

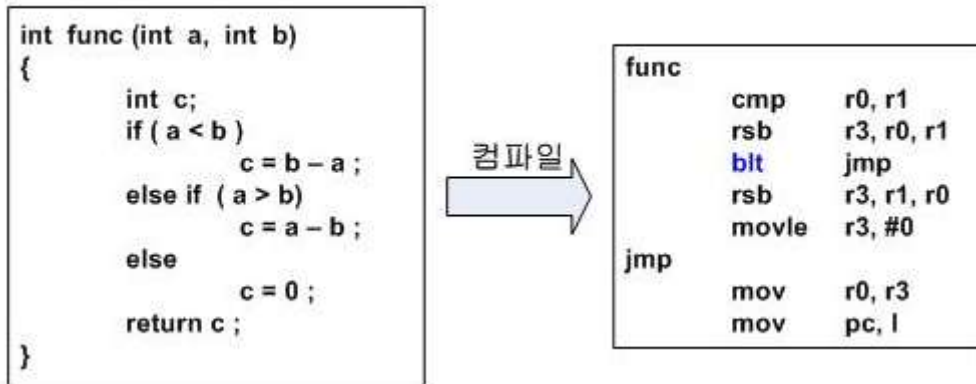
---

- 함수는 기능을 모듈화하여 프로그램 작성을 용이하고 간단하게 하기 위해서 사용
- 함수 사용에 따른 성능 저하
  - ❖ 레지스터 사용이 늘어난다
  - ❖ 스택의 사용이 많아진다.
- 성능 개선
  - ❖ 가능하면 함수의 개수를 줄인다.
  - ❖ 각각의 함수는 간단하고 작게 만들어 사용한다
  - ❖ 인자는 가능하면 4개 이하를 사용한다.
  - ❖ 인라인(**inline**) 함수를 사용한다.

# 분기문 사용과 프로그램 최적화

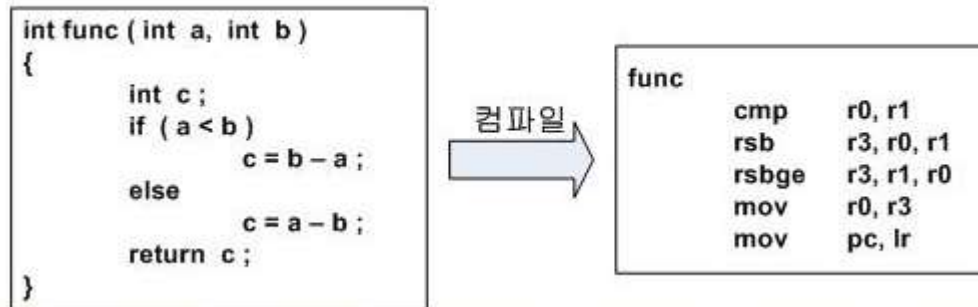
## □ 복잡한 if ~ else는 분기 명령이 사용된다.

비교적 복잡한 if ~ else



## □ 간단한 if ~ else가 소요되면 조건부 실행에 의해서 처리된다.

비교적 간단한 if ~ else



# 루프 사용

- ❑ LOOP 명령 : for ( ), while ( ), do { } while ( ) 등
- ❑ LOOP 명령의 경우 0로 decrement 방법을 사용하여 종료
  - ❖ Code 사이즈가 줄어 들고, 레지스터의 사용이 준다.

업 카운트(up-count) 되는 루프

```
int fact1 ( int n )
{
    int i, fact = 1;
    for ( i = 1 ; i <= n ; i++ )
        fact *= i;
    return fact;
}
```

컴파일

```
fact1
    mov    r2, #1      @ 변수 fact
    mov    r1, #1      @ 변수 i
    cmp    r0, #1      @ 인자 n
    blt    label2
label1
    mul     r2, r1, r2
    add     r1, r1, #1
    cmp     r1, r0
    ble     label1
label2
    mov     r0, r2
    mov     pc, lr
```

다운 카운트(down-count)되는 루프

```
int fact2 ( int n )
{
    int i, fact = 1;
    for ( i = n ; i != 0 ; i-- )
        fact *= i;
    return fact;
}
```

컴파일

```
fact2
    movs    r1, r0      @ 변수 i
    mov     r0, #1
    moveq   pc, lr
label
    mul     r0, r1, r0
    subs    r1, r1, #1 @ 빼기와 flag 셋
    bne     loop
    mov     pc, lr
```

# 목 차

---

## 11장. 소프트웨어 개발 툴의 이해와 활용

### □ 12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

- 01. 컴파일러 사용과 옵션 설정
- 02. 임베디드 C의 구성요소와 프로그램 최적화
- 03. 나눗셈과 나머지 연산
- 04. 메모리 참조와 포인터
- 05. ARM/Thumb 인터워킹

## 13장. 시스템 리셋과 부트코드

## 14장. 하드웨어 제어

## 15장. 부트로더 개발



## 2의 제곱으로 연산

### □ ARM에는 나눗셈 명령이 없다.

❖ 나누기가 사용되면 `__rt_udiv` 라이브러리 사용

### □ 2의 제곱으로 나눗셈 실행

❖ 배럴 쉬프터가 사용되어 한 클럭 동안 나눗셈 완료

부호 없는 경우

```
typedef unsigned int uint;

uint divu ( uint a )
{
    return a / 16 ;
}
```

컴파일

```
divu
mov    r0, r0, LSR #4
mov    pc, lr
```

부호 있는 경우

```
int divs ( int a )
{
    return a / 16 ;
}
```

컴파일

```
divs
cmp    r0, #0
addlt  r0, r0, #0xF
mov    r0, r0, ASR #4
mov    pc, lr
```

# 나머지 연산 변경 사용

## □ 나머지 연산(%)를 if 문으로 변경 예

나머지 연산(%)을 사용하는 경우

```
typedef unsigned int uint;
uint counter1 ( uint count )
{
    return (++count % 60) ;
}
```

컴파일

```
counter1
    stmdb    sp!, {lr}
    add     r1, r0, #1
    mov     r0, #0x3c
    bl      __rt_udiv
    mov     r0, r1
    ldmbia  sp!, {pc}
```

나누기 연산 대신 if 문을 사용하는 경우

```
typedef unsigned int uint;
uint counter2 ( uint count )
{
    if (++count >= 60)
        count = 0;
    return (count) ;
}
```

컴파일

```
counter2
    add     r1, r0, #1
    mov     r0, #0x3c
    movcs   r0, #0
    mov     pc, lr
```

# 목 차

---

## 11장. 소프트웨어 개발 툴의 이해와 활용

### □ 12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

- 01. 컴파일러 사용과 옵션 설정
- 02. 임베디드 C의 구성요소와 프로그램 최적화
- 03. 나눗셈과 나머지 연산
- 04. 메모리 참조와 포인터
- 05. ARM/Thumb 인터워킹

## 13장. 시스템 리셋과 부트코드

## 14장. 하드웨어 제어

## 15장. 부트로더 개발

# 메모리와 입출력 장치 참조

---

## □ 메모리 참조

```
unsigned long func (void)
{
    unsigned long *wdata;
    unsigned long *rdata, value;
    wdata = (unsigned long *) 0x30000100;
    rdata = (unsigned long *) 0x30000200;
    *wdata = 0x12345678;
    value = *rdata;
    return value;
}
```

## □ 입출력 장치 참조

```
unsigned short inw (void)
{
    volatile unsigned short *data_port;
    data_port = (volatile unsigned long *) 0x0x10000304;
    return *data_port;
}
```

# 목 차

---

## 11장. 소프트웨어 개발 툴의 이해와 활용

### □ 12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

- 01. 컴파일러 사용과 옵션 설정
- 02. 임베디드 C의 구성요소와 프로그램 최적화
- 03. 나눗셈과 나머지 연산
- 04. 메모리 참조와 포인터
- 05. ARM/Thumb 인터워킹

## 13장. 시스템 리셋과 부트코드

## 14장. 하드웨어 제어

## 15장. 부트로더 개발

# ARM & Thumb Interworking

## □ Interworking이 필요한 이유

### ❖ Thumb 명령의 장점

- Code density가 ARM 명령 보다 높다
- 좁은 메모리 인터페이스에서 더 좋은 성능을 가진다.

### ❖ Thumb 명령으로 처리할 수 없는 동작

- 모든 exception handling
  - ✓ *Exception이 발생하면 ARM은 무조건 ARM state가 된다.*
- PSR(Program Status Register) Transfer 명령
  - ✓ *인터럽트 제어, 프로세서 모드 변환*
- Coprocessor access
  - ✓ *Cache, MMU 제어 등 포함*

### ❖ ARM 명령의 장점

- 메모리 인터페이스가 32비트 이면 성능 우수
  - ✓ *특히 매우 빠른 처리를 위한 경우 32비트의 내부 SRAM서 처리하는 경우*
  - ✓ *TCM을 사용하는 경우 등*

## □ 아무리 16비트 Thumb 명령을 사용하고자 하여도 32비트 ARM 명령은 반드시 필요

# Interworking 명령

## □ ARM Architecture v4T 이하

- ❖ Core에 'T' 가 있는 경우만 Thumb 명령 사용 가능
  - 예를 들어 Architecture v4는 Interworking이 지원 않됨
- ❖ BX 명령
  - Branch with Exchange 명령

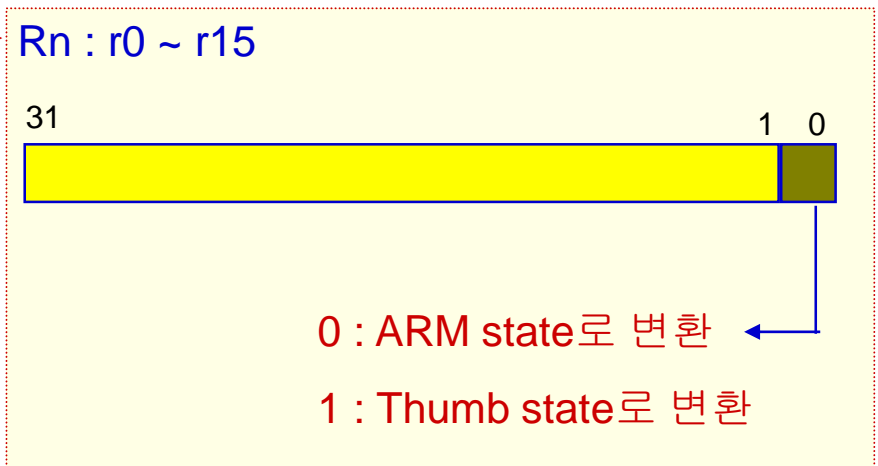
사용법 :  $BX\{cond\}$   $Rn$

$Rn : r0 \sim r15$

## □ ARM Architecture v5T 이상

- ❖ BX명령과 함께 BLX 명령 추가
- ❖ BLX 명령
  - Branch with Link and Exchange

사용법 :  $BLX$   $\langle offset \rangle$



# 목 차

---

11장. 소프트웨어 개발 툴의 이해와 활용

12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

□ 13장. 시스템 리셋과 부트코드

01. 리셋 핸들러

02. 부트코드 작성

14장. 하드웨어 제어

15장. 부트로더 개발



# 프로세서의 **Reset** 과 부트 코드

---

□ 프로세서에 리셋 신호가 입력되면 실행 중이던 명령을 멈추고

~~① SPSR\_svc에 CPSR 값을 복사~~

② CPSR의 값을 변경

- Mode bit M[4:0]를 Supervisor 모드인 10011'b로 변경
- I 비트와 F 비트를 1로 세트하여 인터럽트를 disable
- T 비트를 0으로 클리어하여 ARM state로 변경

~~③ PC 값을 LR\_svc 레지스터에 복사~~

④ PC 값을 Reset Vector 어드레스인 0x00000000으로 변경

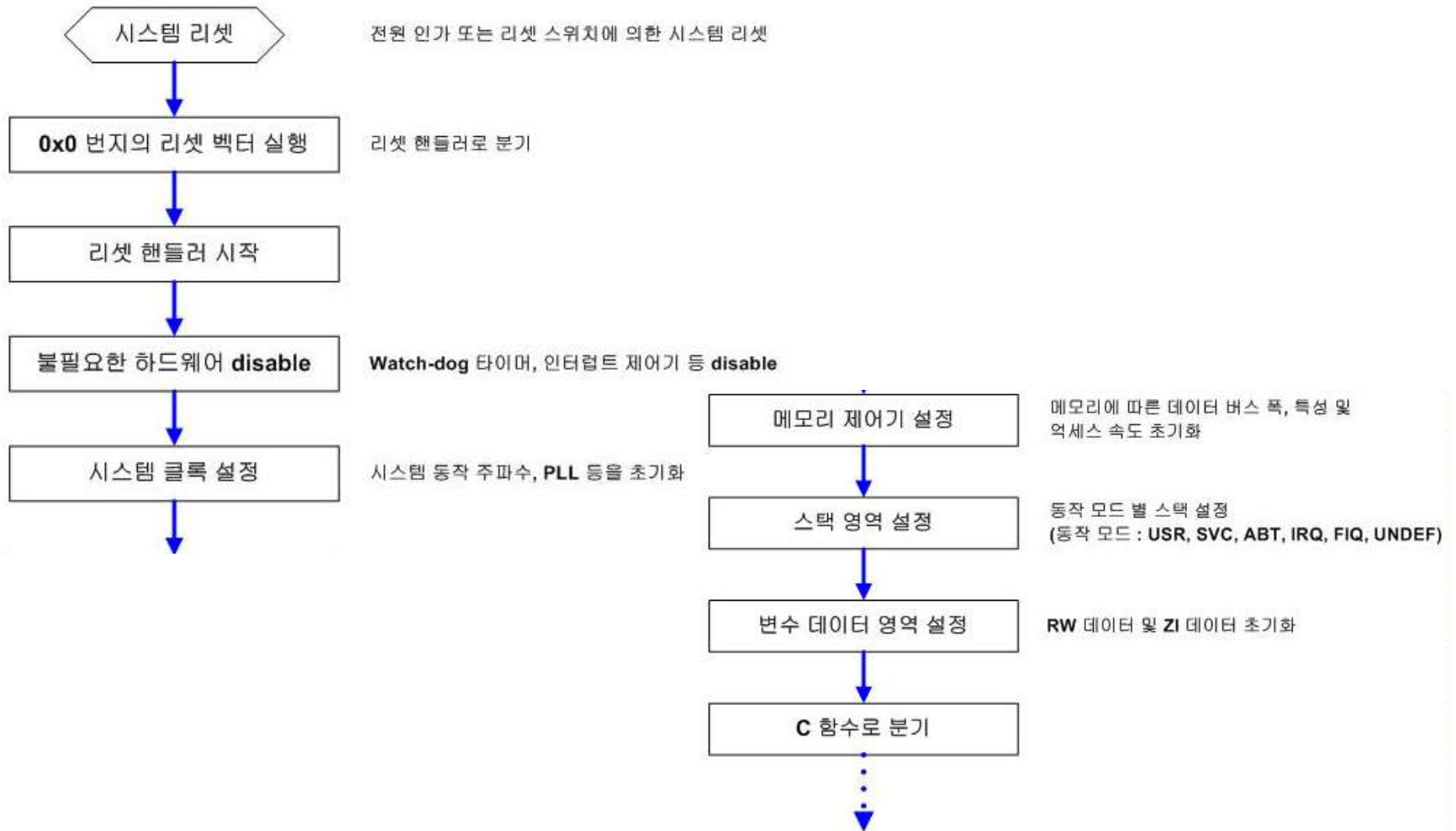
⑤ **Reset** 핸들러로 분기하여 시스템의 초기화 수행

# 리셋 핸들러

---

- ❑ **Reset Vector**에서 분기된 어셈블리어로 작성된 처리 루틴
- ❑ 주요 동작
  - ❖ 하드웨어 설정
    - 불필요한 하드웨어 동작 중지
    - 시스템의 클럭 설정
    - 메모리 제어기 설정
    - 필요에 따라 MMU나 MPU 설정
  - ❖ 소프트웨어 동작 환경 설정
    - 예외처리
    - 스택(stack) 영역 설정
    - 힙(heap) 영역 설정
    - 변수영역 초기화
- ❑ 리셋 핸들러의 마지막에서는 **main()** 함수와 같은 **C**로 구성된 함수를 호출
- ❑ 이 부분을 **startup 코드** 또는 **부트코드**라고 부른다.

# 부트코드 초기화 과정



# 부트 코드

---

## □ 부트 코드를 작성하기 위해서는

- ❖ Programmer's model, 특히 명령어
  - ❖ 시스템 하드웨어 구조 및 기능에 대한 전반적인 사항
- 등을 모두 알아야 한다.

# 목 차

---

11장. 소프트웨어 개발 툴의 이해와 활용

12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

□ 13장. 시스템 리셋과 부트코드

01. 리셋 핸들러

02. 부트코드 작성

14장. 하드웨어 제어

15장. 부트로더 개발

# 부트코드 작성

---

□ 일반적으로 부트코드의 시스템의 초기화 동작은 다음을 포함한다.

- ❖ 엔트리 포인트(Entry point)를 정의
- ❖ 예외처리 벡터 설정
- ❖ 불필요한 하드웨어 동작 중지
- ❖ 시스템 클럭 설정
- ❖ 메모리 시스템 초기화
- ❖ 스택 영역 설정
- ❖ IRQ 예외처리 핸들러 설정 및 IRQ 인에이블(enable)
- ❖ C에서 사용하는 변수 초기화
- ❖ C 코드로 분기

# Entry Point 선언

□ 모든 프로그램은 시작되는 부분을 정의해야 한다.

❖ 어셈블러로 구성되며 모든 프로그램에는 반드시 필요

❖ 일반적으로 crt0.S(C-runtime 0 의 약자), init.S, startup.S 등의 이름 사용

□ Entry Point 선언

❖ SDT, ADS, RVDS or CodeWarrior

➤ ENTRY 어셈블러 디렉티브를 사용한다.

```
AREA Init,CODE,READONLY
ENTRY
ResetHandler
```

❖ CodeWarrior 2.x or GNU

```
.text
.globl _start
.org 0
_start: /* entry point */
```

# 예외처리 벡터 설정

---

## □ 0x0의 ROM에 vector table이 정의 되어 있는 경우

- ❖ ROM에 vector table이 고정 되어 있고 시스템 동작 중에 별도의 vector table의 설정이 필요 없는 경우
- ❖ Re-mapping이 지원되지 않는 경우에는 항상 vector table이 고정 되어 있다.

## □ 0x0에 vector table 없이 초기화 코드만 있는 경우

- ❖ 0x0 번지에 최소한의 reset handler가 동작 하도록 되어있고, 이 reset handler에서 re-mapping 및 vector table 초기화
- ❖ Re-mapping이 지원되는 경우 대부분 시스템 동작 중에 vector table의 초기화 과정이 있다.
- ❖ Linux 등의 대부분 임베디드 OS는 OS 초기화 과정에서 자신의 exception vector table을 초기화 한다.



# 예외처리 벡터 작성 예

---

<b>B</b>	<b>ResetHandler</b>	@ 0x00 번지, 리셋 벡터
<b>LDR</b>	<b>PC, HandlerUndef</b>	@ 0x04 번지, UNDEF 벡터
<b>LDR</b>	<b>PC, HandlerSWI</b>	@ 0x08 번지, SWI 벡터
<b>LDR</b>	<b>PC, HandlerPAbort</b>	@ 0x0C 번지, Prefetch 어보트 벡터
<b>LDR</b>	<b>PC, HandlerDAabort</b>	@ 0x10 번지, Data 어보트 벡터
<b>B</b>	<b>.</b>	@ 0x14 번지, Reserved
<b>LDR</b>	<b>PC, HandlerIRQ</b>	@ 0x18 번지, IRQ 벡터
<b>LDR</b>	<b>PC, HandlerFIQ</b>	@ 0x1C 번지, FIQ 벡터
.....		

@ 리셋 핸들러 시작

**ResetHandler**

@ 불필요한 하드웨어 **disable**

.....

# 불필요한 하드웨어 동작 중지

---

## □ 시스템 초기화를 방해하는 하드웨어 동작 중지

❖ 와치독(Watchdog) 타이머, 인터럽트 제어기 등

❖ 와치독 타이머 중지 예

LDR	r0, =0x53000000	@ R0에 주소 0x53000000 값을 기록한다.
LDR	r1, =0x0	@ R1에 설정할 레지스터 값을 기록한다.
STR	r1, [r0]	@ R0주소에 R1을 기록한다.

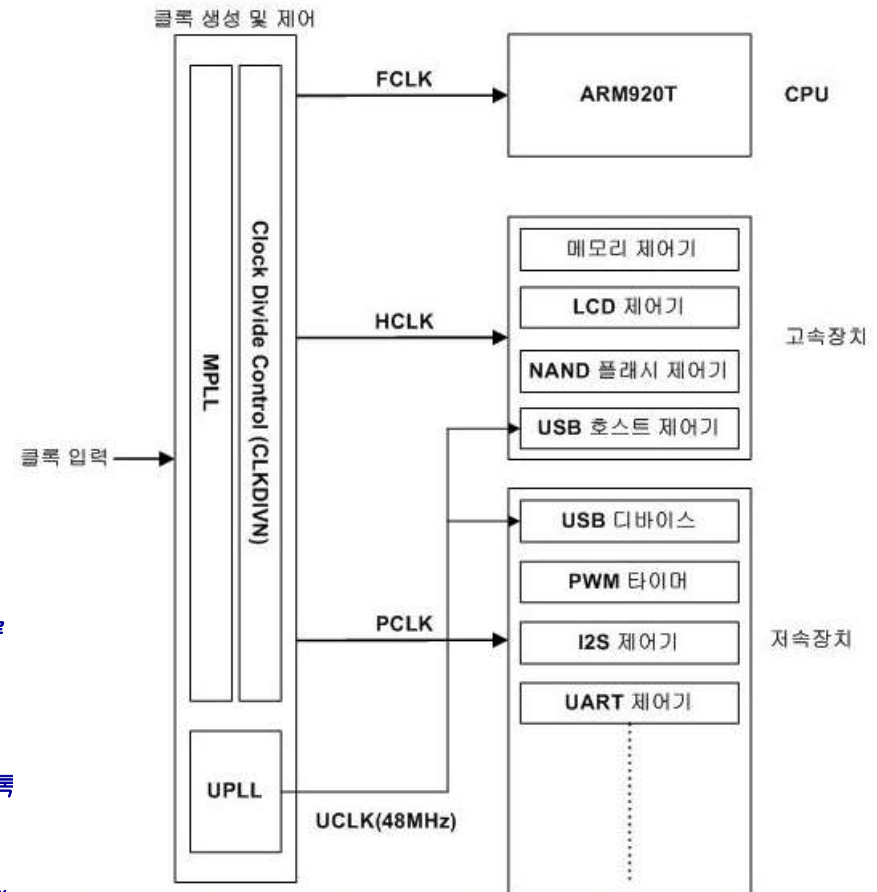
# 시스템 클록 설정

## □ PLL 설정

- ❖ FCLK
- ❖ HCLK
- ❖ UCLK

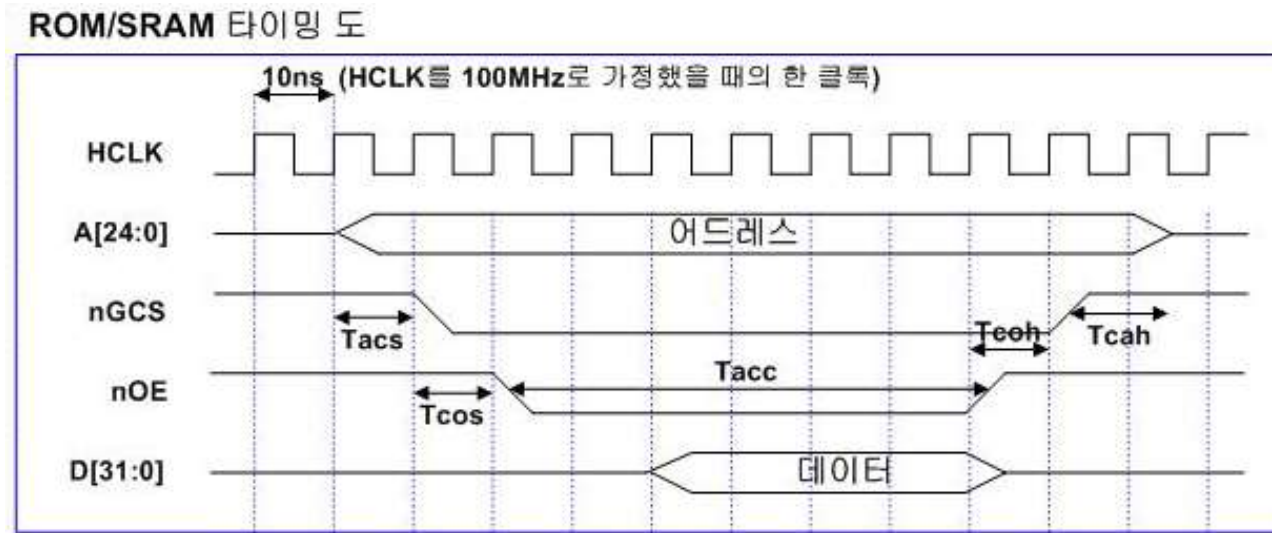
## □ CLOCK 설정 예 ([표 13-3] 참조)

<code>LDR r0, =0x4C000014</code>	@ R0에 CLKDIVN 레지스터의 주소 기록
<code>LDR r1, =0x3</code>	@ R1에 설정할 레지스터 값을 기록한다.
<code>STR r1, [r0]</code>	@ FCLK : HCLK : PCLK = 1:2:4
<code>LDR r0, =0x4C000004</code>	@ R0주소에 R1을 기록, CLKDIVN 설정
<code>LDR r1, =0xa1031</code>	@ R0에 MPLLCON 레지스터의 주소 기록
<code>STR r1, [r0]</code>	@ R1에 설정할 레지스터 값을 기록
	@ 202.8MHz 출력 사용하도록 설정
	@ R0주소에 R1을 기록, MPLLCON 설정



# 메모리 시스템 초기화

## □ 사용되는 메모리의 데이터 버스 폭, 액세스 타이밍 등



## □ 메모리 뱅크 2 설정 예 (372 페이지 참조)

LDR r0, =0x4800000C	@ R0에 BANKCON2 레지스터의 주소를 기록
LDR r1, =0x00002A50	@ R1에 설정할 레지스터 값 기록
STR r1, [r0]	@ R0주소에 R1을 기록, BANKCON2 설정

# 스택 영역 설정

*/\* 스택의 위치 지정, GNU style 선언 \*/*

**#define STACK\_BASE\_ADDR 0x33ff8000**

**#define UserStack       STACK\_BASE\_ADDR - 0x3800** */\* USR 스택 base \*/*

**#define SVCStack        STACK\_BASE\_ADDR - 0x2800** */\* SVC스택 base \*/*

**#define UndefStack      STACK\_BASE\_ADDR - 0x2400** */\* Undef스택 base \*/*

**#define AbortStack      STACK\_BASE\_ADDR - 0x2000** */\* Abort 스택 base \*/*

.....

*/\* ABORT mode stack \*/*

**orr     r1,r0,#0x17** */\* ABORT 모드 값 지정 \*/*

**msr     cpsr\_c,r1** */\* ABORT 모드로 변환 \*/*

**ldr     sp,=AbortStack** */\* ABORT 모드 스택 설정 \*/*

*/\* IRQ mode stack \*/*

**orr     r1,r0,#0x12** */\* IRQ 모드 값 지정 \*/*

**msr     cpsr\_c,r1** */\* IRQ 모드로 변환 \*/*

**ldr     sp,=IRQStack** */\* IRQ 모드 스택 설정 \*/*

.....

# IRQ 예외처리 핸들러와 IRQ 인에이블

## □ 예외처리 핸들러

```
ldr    r0, =HandleIRQ    /* IRQ 예외처리 테이블 주소 기록 */
ldr    r1, =IRQ_Handler  /* IRQ 예외처리 핸들러 주소 기록 */
str    r1, [r0]           /* 예외처리 테이블에 핸들러 주소 기록 */
```

## □ 인터럽트 핸들러

IRQ\_Handler:

```
sub    lr, lr, #0x4        /* 되돌아갈 주소 계산 */
stmfd  sp!, {r0-r12, lr}   /* PUSH */
ldr    r0, =0x4A000014     /* 발생된 인터럽트 번호를 가지는 레지스터 */
ldr    r0, [r0]            /* 인터럽트 번호를 읽어 R0에 기록 */
bl     do_IRQ              /* 인터럽트 번호를 인자로 do_IRQ 호출 */
ldmfd  sp!, {r0-r12, pc}^  /* POP 하면서 CPSR 값 복원 */
```

## □ IRQ 인에이블

```
mrs    r0, cpsr            /* CPSR 값을 R0에 읽는다 */
bic    r0, r0, #0x80       /* I 비트를 클리어 한다 */
msr    cpsr_c, r0          /* CPSR의 c 필드에 R0 값을 저장한다 */
```

# C에서 사용되는 변수 초기화

```
ldr r0, =_etext    /* RO 데이터의 끝을 지정 */
ldr  r1, =_data     /* RW 데이터의 시작을 지정 */
ldr  r3, =_bss_start /* BSS 영역의 시작을 지정 */

/* RW 초기값 데이터 복사, [그림 11-10]과 같은 메모리 구조 */
cmp  r0, r1         /* RO의 끝과 RW의 시작을 비교 */
beq  2f             /* [그림 11-9]와 같은 메모리 구조, 복사 필요 없음 */
1:                  /* 다르면 롬에 있는 RW 초기값을 RW 영역으로 복사 */
cmp  r1, r3         /* ZI 데이터 영역의 시작까지 복사 */
ldrcc r2, [r0], #4   /* R2에 RW 데이터 초기값 LOAD */
strcc r2, [r1], #4   /* RW 영역에 저장 */
bcc  1b
2:
ldr  r1, =_end       /* ZI의 끝 지정 */
mov  r2, #0          /* 초기화할 데이터 0 */
3:
cmp  r3, r1         /* ZI의 끝 지점까지 */
strcc r2, [r3], #4   /* 0으로 초기화 */
bcc  3b
```

# 목 차

---

11장. 소프트웨어 개발 툴의 이해와 활용

12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

13장. 시스템 리셋과 부트코드

□ 14장. 하드웨어 제어

01. GPIO 제어와 LED 점멸

02. UART 장치 제어

03. 타이머 제어

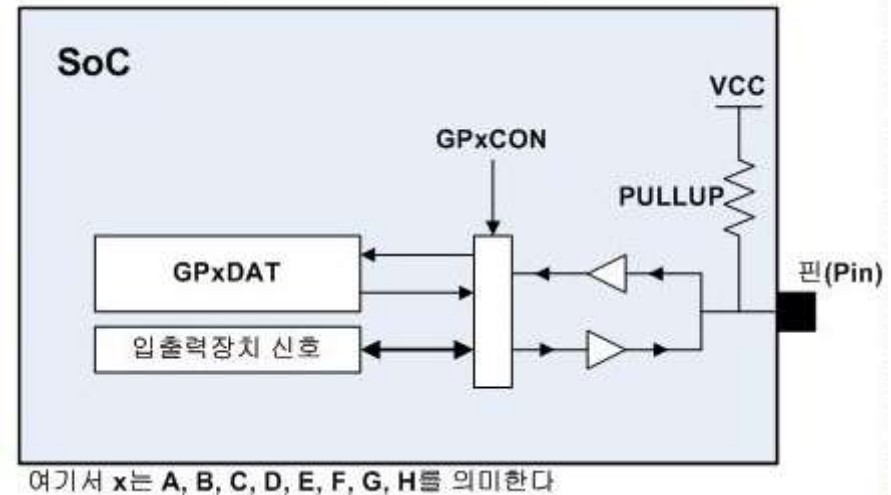
15장. 부트로더 개발



# GPIO (General Purpose Input/Output)

## □ 117 개의 Multiplexed Input/Output

- ❖ Port A (GPA) : 23 output 포트
- ❖ Port B (GPB) : 11 input/out 포트
- ❖ Port C (GPC) : 16 input/out 포트
- ❖ Port D (GPD) : 16 input/out 포트
- ❖ Port E (GPE) : 16 input/out 포트
- ❖ Port F (GPF) : 8 input/out 포트, External Interrupt
- ❖ Port G (GPG) : 16 input/out 포트, External Interrupt
- ❖ Port H (GPH) : 11 input/out 포트



# GPIO 설정을 위한 레지스터 (1)

---

## □ Port Control Register (GPxCON)

- ❖ Where x=A,B,C,D,E,F,G and H
- ❖ GPIO port의 pin 설정

## □ Port Data Register (GPxDAT)

- ❖ Where x=A,B,C,D,E,F,G and H
- ❖ GPIO port의 data 값 지정

## □ Pull-up Disable Register (GPxUP)

- ❖ Where x=B,C,D,E,F,G and H

# GPIO 설정을 위한 레지스터 (2)

---

## □ External Interrupt Control Register (EXTINTn)

- ❖ Where  $n=0,1,2$ 
  - EXTINT0 : external interrupt 0 ~ 7
  - EXTINT1 : external interrupt 8 ~ 15
  - EXTINT2 : external interrupt 16 ~ 23
- ❖ 24개의 외부 인터럽트를 제어하기 위해 사용된다.

## □ External Interrupt Filter Register (EINTFLTn)

- ❖ Where  $n=0,1,2,3$ 
  - EINTFLT0 , EINTFLT1 : reserved
  - EINTFLT2 : external interrupt 16 ~ 19
  - EINTFLT3 : external interrupt 20 ~ 23

## □ External Interrupt Mask register (EINTMASK)

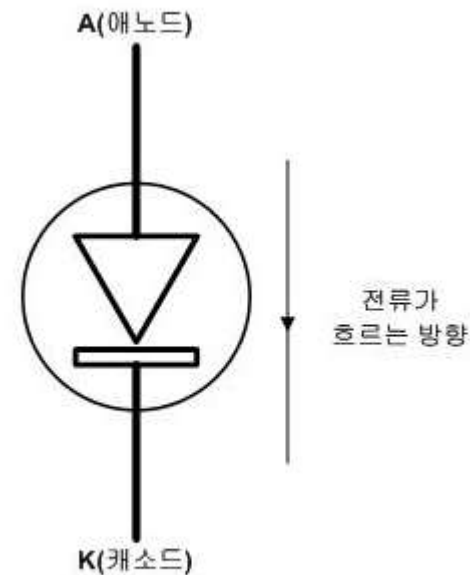
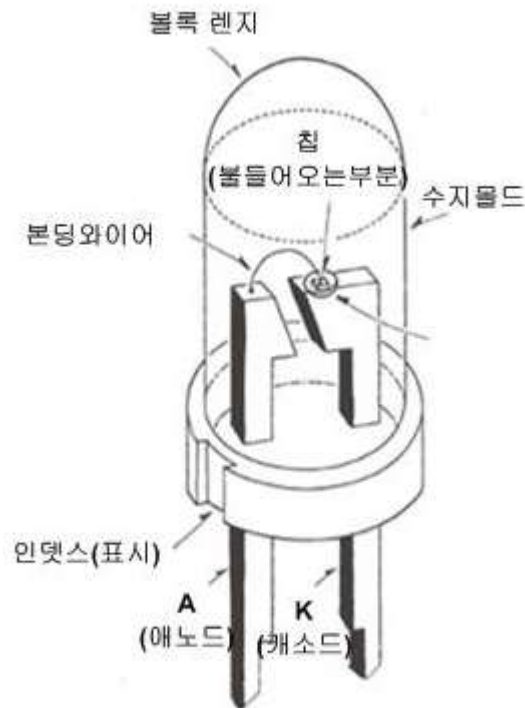
- ❖ EINT4 ~ EINT23 인터럽트의 enable 또는 disable

## □ External Interrupt Pending register (EINTPEND)

- ❖ EINT4 ~ EINT23 인터럽트를 hold

# LED (발광 다이오드)

- 전기를 통해 주면 전자가 에너지 레벨이 높은 곳에서 낮은 곳으로 이동하며 특정한 파장의 빛을 내는 반도체 소자

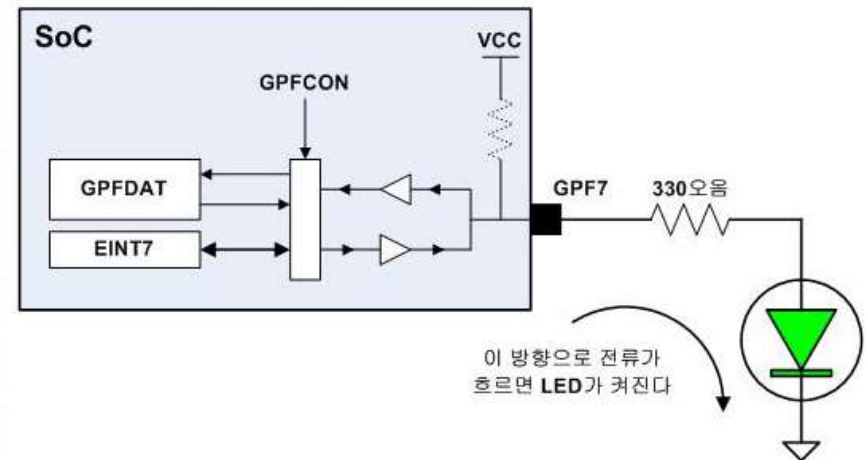


# GPIO 제어에 의한 LED 구동

```
#define rGPFCON      (*(volatile unsigned *)0x56000050)/* 포트 F 제어용 레지스터 */
#define rGPFDAT      (*(volatile unsigned *)0x56000054)/* 포트 F 데이터 레지스터 */
#define rGPFUP       (*(volatile unsigned *)0x56000058)/* 포트 F 풀업 제거 설정 */

void function (unsigned long write_data)
{
    unsigned long up, read_data;
    rGPFCON = 0x4000;          /* GPFCON 주소에 0x4000 값 기록
                                GPF7을 출력 모드로 설정 */
    up = rGPFUP;               /* GPFUP 레지스터에 기록된 값을 읽어 up 변수에 기록 */
    read_data = rGPFDAT;       /* GPFDAT 레지스터에 기록된 값을 읽어 read_data 변수에 저장 */
    rGPFDAT = write_data;     /* write_data 값을 GPFDAT 레지스터에 기록 */
}
```

```
/* LED 구동을 제어하기 위한 메인 프로그램 */
Main()
{
    int i;
    /* 변수 i의 비트 0에 의해서 LED 구동
    * 변수 i의 비트 0가 0이면 LED off, 1이면 LED on
    * Led_Display() 함수에서는 전달되는 변수 i의 비트 0만 사용
    */
    for (i=0; i<10; i++) {
        Led_Display(i); /* Led_Display 함수 호출, 인자는 i */
        Delay(500);     /* 딜레이를 주어 LED를 on 또는 off 유지
        시간을 준다 */
    }
}
```



# 목 차

---

11장. 소프트웨어 개발 툴의 이해와 활용

12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

13장. 시스템 리셋과 부트코드

□ 14장. 하드웨어 제어

01. GPIO 제어와 LED 점멸

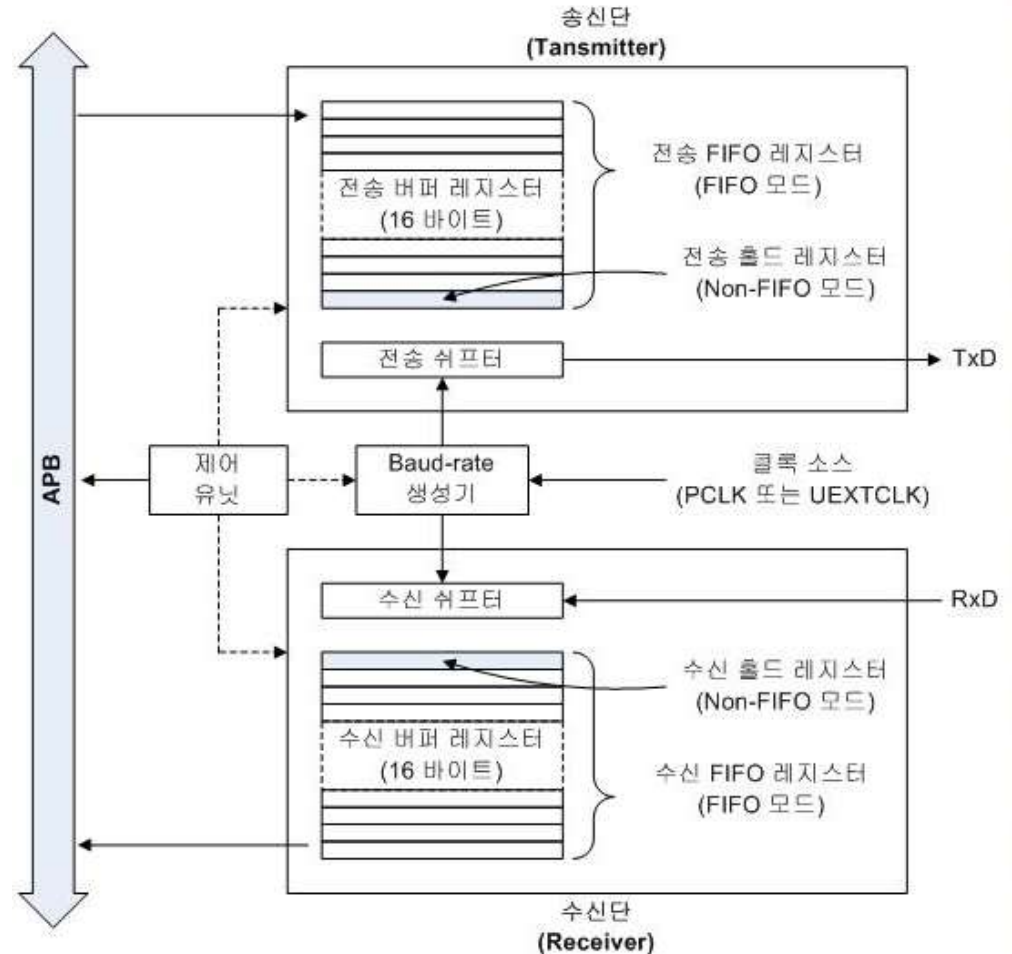
02. UART 장치 제어

03. 타이머 제어

15장. 부트로더 개발

# UART 제어기 구조

- UART는 직렬로 데이터를 교환하는 통신 방식을 제공하고 사용하기가 간단하여 시스템 콘솔 또는 디버깅용으로 많이 사용되는 장치



# S3C2410의 UART

---

- ❑ **UART (Universal Asynchronous Receiver and Transmitter)**
- ❑ **3개의 독립적인 UART 제어기**
- ❑ **DMA 또는 interrupt 방식에 의한 제어**
- ❑ **IrDA1.0 지원**
- ❑ **16-byte Tx/Rx FIFO 지원**
- ❑ **외부 UART clock 지원**
- ❑ **nRTS. nCTS 지원**
  - ❖ **UART channel 0 와 1**



# UART 제어용 레지스터 (1)

레지스터		어드레스	기 능
UART Line Control	ULCON0 ULCON1 ULCON2	0x5000_0000 0x5000_4000 0x5000_8000	Word length, stop bit, parity 모드 설정 Normal 또는 IrDA 선정
UART Control	UCON0 UCON1 UCON2	0x5000_0004 0x5000_4004 0x5000_8004	Tx/Rx 모드 선정, clock source 선정 등
UART FIFO Control	UFCON0 UFCON1 UFCON2	0x5000_0008 0x5000_4008 0x5000_8008	FIFO enable, FIFO trigger level 설정
UART Modem Control	UMCON0 UMCON1	0x5000_000C 0x5000_400C	Auto Flow Control enable/disable RTS software control status
UART Tx/Rx Status	UTRSTAT0 UTRSTAT1 UTRSTAT2	0x5000_0010 0x5000_4010 0x5000_8010	Transmit & Receive status

# UART 제어기 설정

## □ UART0 제어기 설정

- ❖ 통신속도(Baud Rate : bit/sec) 115200
- ❖ 데이터 비트 수 8
- ❖ 패리티 사용 없음
- ❖ 정지 비트 수 1
- ❖ 흐름 제어 없음
- ❖ FIFO 사용 하지 않음

## □ UART 초기화

```
rULCON0    = 0x03;    /* ULCON0 : 8 data bits, 1 stop bit, no-parity, normal 모드 */
rUCON0     = 0x5;     /* UCON0 : Tx/Rx 인터럽트 모드, PCLK 사용 */
rUFCON0    = 0x0;     /* not use */
rUMCON0    = 0x0;     /* not use */
rUBRDIV0   = 26;      /* 115200bps at 202.8MHz */
```

# Baud-rate 생성

## □ Baud-rate Divisor 레지스터

레지스터		어드레스	기 능
UART Baud-rate Divisor	UBRDIV0	0x5000_0028	UAT Baud-rate divisor 값을 저장
	UBRDIV1	0x5000_4028	
	UBRDIV2	0x5000_8028	

## □ Clock 소스

### ❖ PCLK

- Peripheral BUS clock
- $UBRDIVn = (int) ( PCLK / ( bps * 16 ) ) - 1$

### ❖ UCLK

- 외부의 UART 전용 clock
- $UBRDIVn = (int) ( UCLK / ( bps * 16 ) ) - 1$

## □ 예제

- ❖ Baud-rate : 115200 bps
- ❖ UCLK or PCLK : 50.7 MHz
- ❖ UBRDIVn

$$\begin{aligned} UBRDIVn &= (int) ( 50700000 / ( 115200 * 16 ) ) - 1 \\ &= (int) ( 27.5 ) - 1 \\ &= 217 - 1 = 26 \end{aligned}$$

# UART 통신

## □ 데이터 전송

```
void Uart_SendByte( char ch )
{
    while( !(rUTRSTAT0 & 0x04) );    /* 전송 버퍼 및 쉬프트의 데이터의
                                       전송이 완료될 때 까지 대기 */
    rUTXH0=(unsigned char)(ch);        /* 8비트 데이터 기록 */
    if(ch == '\n') {                  /* 만약 리턴이면 \r 추가 송신 */
        while( !(rUTRSTAT0 & 0x04) ); /* 전송완료 대기 */
        rUTXH0=(unsigned char)('\r');
    }
}
```

## □ 데이터 수신

```
char Uart_Getch(void)
{
    while( !(rUTRSTAT0 & 0x01) );    /* 데이터 수신 대기 */
    return (unsigned char)rURXH0;     /* 8비트 데이터를 읽어서 리턴 */
}
```

# 목 차

---

11장. 소프트웨어 개발 툴의 이해와 활용

12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

13장. 시스템 리셋과 부트코드

□ 14장. 하드웨어 제어

01. GPIO 제어와 LED 점멸

02. UART 장치 제어

03. 타이머 제어

15장. 부트로더 개발

# PWM Timer

---

## □ 5개의 16-bit Timer

- ❖ 2개의 8-bit prescalers 와 2개의 4-bit divider 지원

- ❖ 4-channel

  - PWM(Pulse Width Modulation) 출력 신호 제공

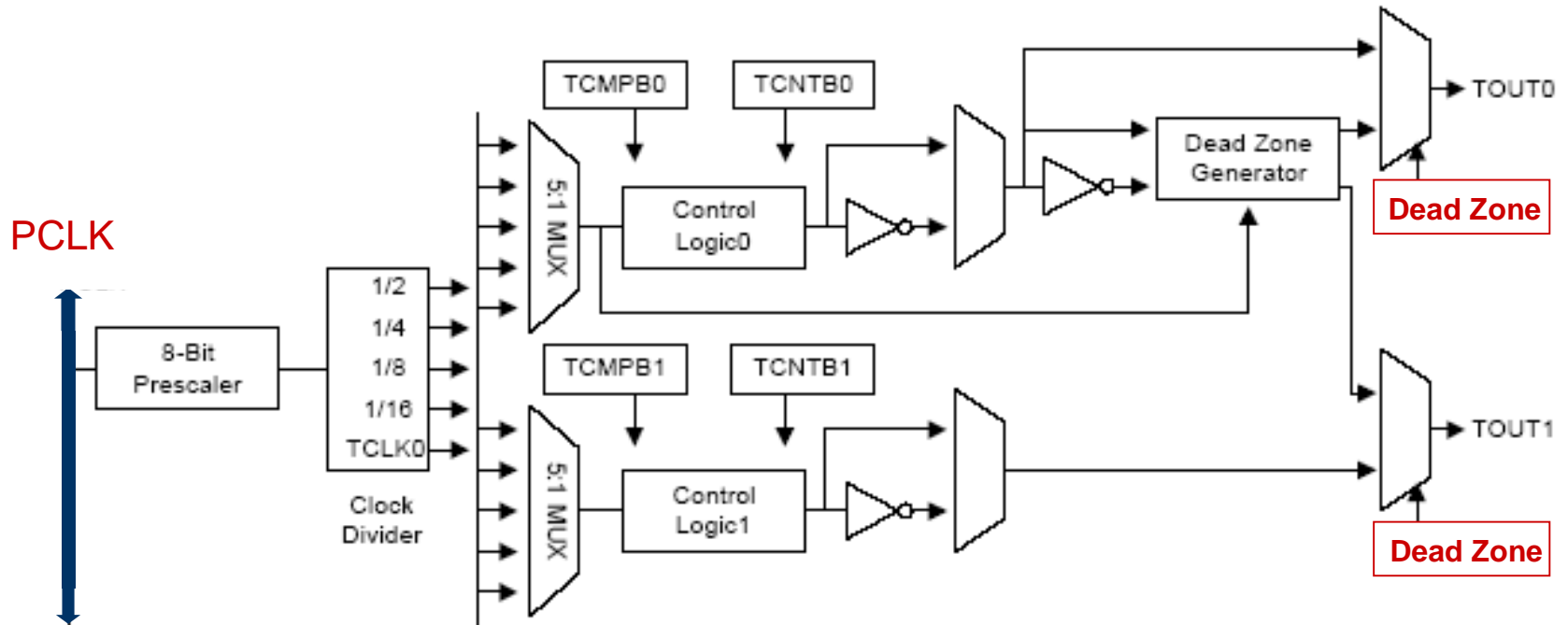
- ❖ 1-channel

  - 내부에서만 사용, 인터럽트 또는 DMA 동작 등

## □ Auto reload mode 또는 one-shot pulse 모드 지원

## □ Dead-zone generator

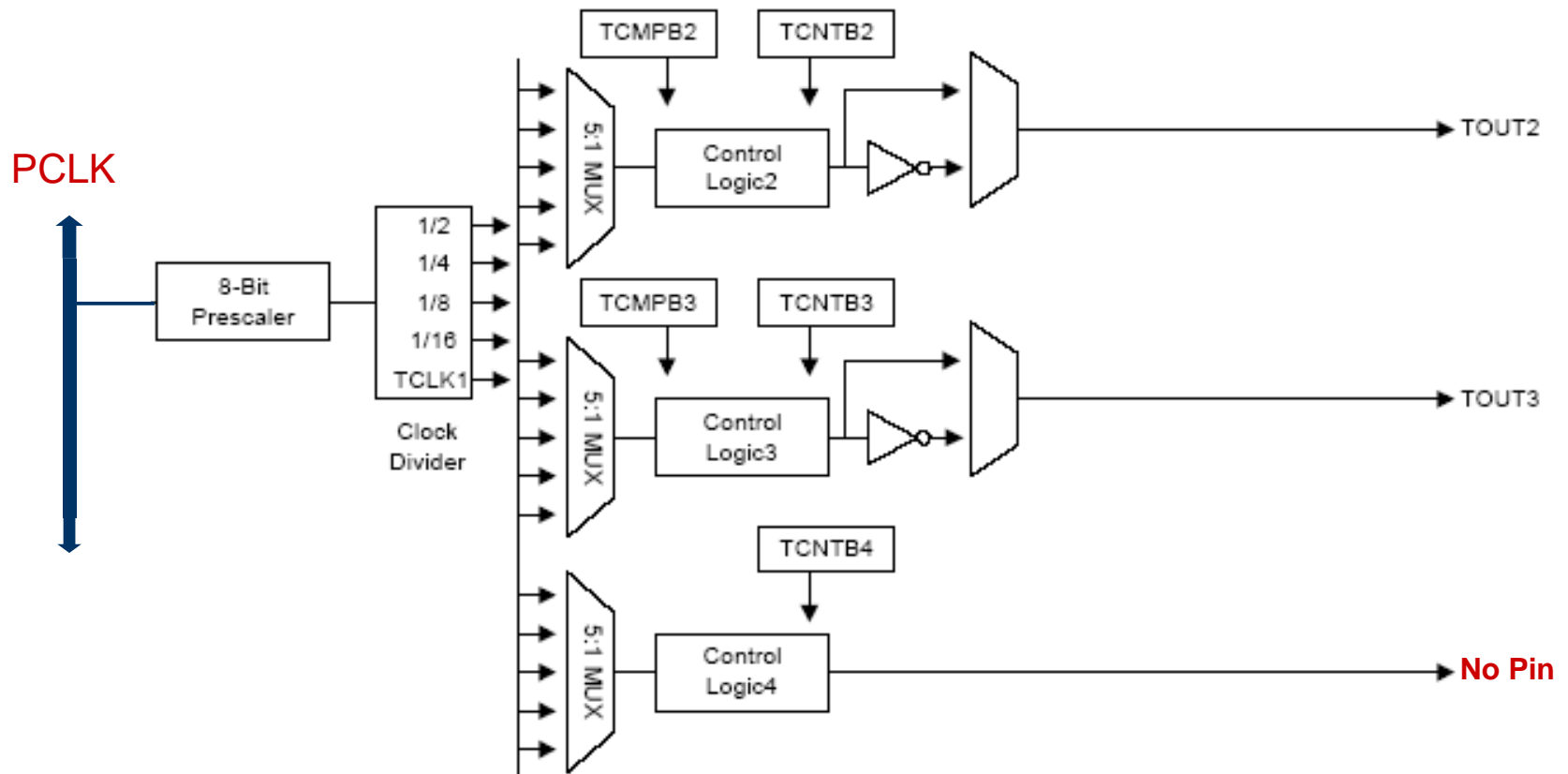
# PWM Timer 0 , 1 구조



TCNTB : Timer Count Buffer

TCMPB : Timer Compare Buffer

# PWM Timer 2,3,4 구조



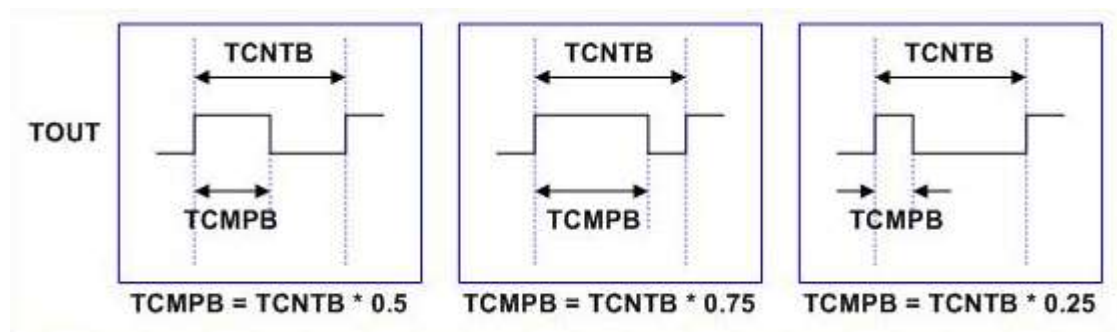


# PWM Timer 레지스터

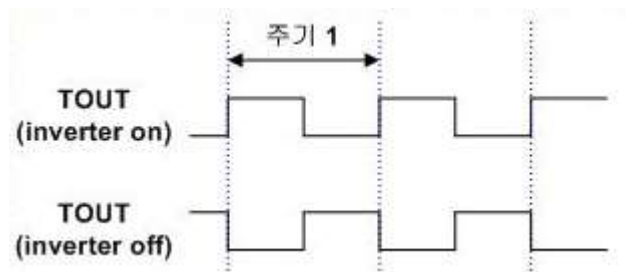
레지스터		어드레스	기 능
Timer Configuration Register 0	TCFG0	0x5100_0000	Timer prescaler0, 1 값 설정 Dead zone length 설정
Timer Configuration Register 1	TCFG1	0x5100_0014	MUX 와 DMA 모드 설정
Timer Control	TCON	0x5100_0008	5개의 Timer 제어, Timer enable, Auto reload, dead zone enable 등
Timer Count Buffer	TCNTB0 TCNTB1 TCNTB2 TCNTB3 TCNTB4	0x5100_000C 0x5100_0018 0x5100_0024 0x5100_0030 0x5100_003C	Timer counter 값을 설정
Timer Compare Buffer	TCMPB0 TCMPB1 TCMPB2 TCMPB3	0x5100_0010 0x5100_001C 0x5100_0028 0x5100_0030	Compare 값을 설정
Timer Count Observation	TCNTO0 TCNTO1 TCNTO2 TCNTO3 TCNTO4	0x5100_0014 0x5100_0020 0x5100_002C 0x5100_0038 0x5100_0040	Timer counter 값을 설정

# 타이머의 PWM 출력 신호 제어

## □ TCNTB와 TCMPB에 의한 PWM 출력 신호



## □ 인버터에 의한 PWM 출력 변화



# 와치독 타이머(WDT)

---

- 시스템에 에러가 발생하여 정상적으로 동작이 불가능한 경우

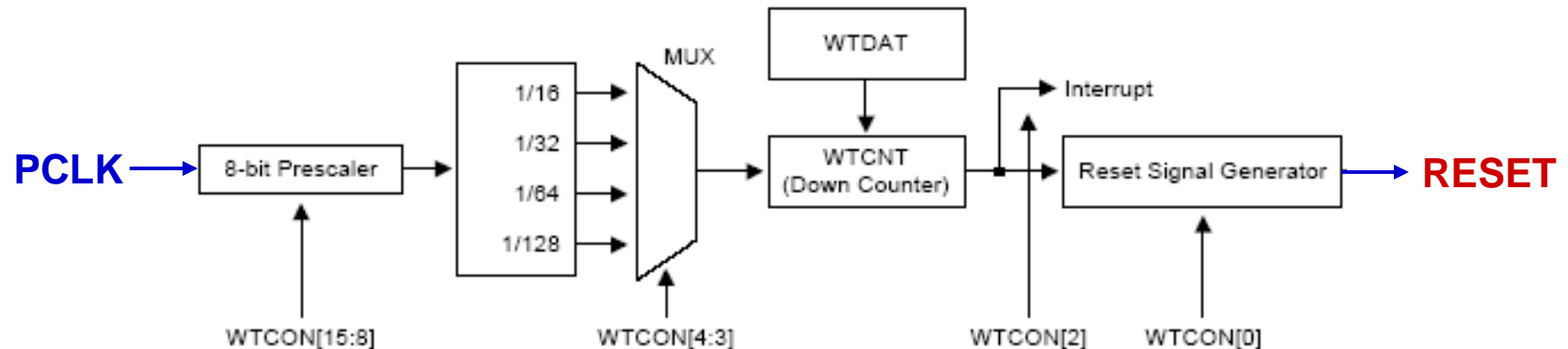
시스템에 일정 시간 동안 **reset** 신호 구동

❖ 128 PCLK 사이클 동안 reset 신호를 구동한다.

- **Normal interval timer** 모드로 설정 가능

# Watchdog Timer의 구성

- ❑ PCLK를 source clock로 사용
- ❑ Watchdog Timer clock 주파수  
$$= 1 / ( \text{PCLK} / (\text{prescaler value} + 1 / \text{division\_factor}) )$$



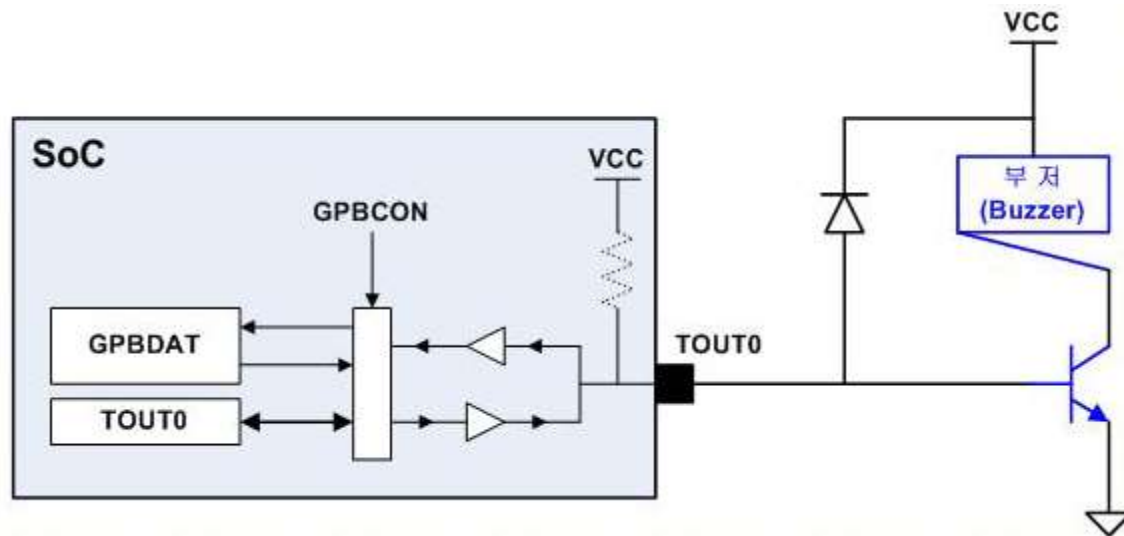
# Watchdog Timer 레지스터

레지스터		어드레스	기 능
Watchdog Timer Control	WTCN	0x5300_0000	Watchdog timer 설정 WDT enable/disable, Reset Enable/disable, 인터럽트 생성, prescaler 값 설정 등
Watchdog Timer Data	WTDAT	0x5300_0004	비트 [15:0] : watchdog timer reload count 값 설정
Watchdog Timer Count	WTCNT	0x5300_0008	비트 [15:0] : watchdog timer 현재 count 값

# 타이머의 PWM 출력에 의한 부저 구동

## □ 압전부저(Piezo Electric Buzzer)를 구동하여 신호음 발생

- ❖ 압전부저는 전압 신호를 음량신호로 바꾸어 주는 소자
- ❖ 입력되는 신호의 높낮음에 의하여 진동판이 진동하여 음을 발생하도록 만들어진 원형 모양의 판
- ❖ 부저에서 음의 높이는 주파수를 높이면 고음이 되고, 주파수를 낮추면 저음이 된다.
- ❖ 음의 강약은 신호 파형의 진폭에 의해 결정된다.



# 목 차

---

11장. 소프트웨어 개발 툴의 이해와 활용

12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

13장. 시스템 리셋과 부트코드

14장. 하드웨어 제어

□ 15장. 부트로더 개발

01. 부트로더 이해

02. U-Boot 빌드와 설치

03. U-Boot 활용

04. U-Boot 포팅

# 부트로더의 역할 (1)

---

## □ 타겟 시스템 초기화

- ❖ 부트로더는 전원이 입력되면 타겟 시스템이 정상동작 할 수 있도록 하드웨어 및 소프트웨어 동작 환경을 설정
- ❖ 불필요한 하드웨어의 동작 중지, 시스템 클럭 설정, 메모리 제어기 설정 및 필요에 따라 MMU나 MPU를 설정
- ❖ 프로그램 동작에 필요한 재배치(relocation), 스택 영역 설정 및 C에서 사용되는 변수 영역을 설정한 다음 C로 작성된 함수 호출
- ❖ 부트로더에는 필요에 따라 IRQ와 같은 예외처리(Exception Handling) 처리 벡터 및 핸들러도 작성 되어야 한다.

## □ 타겟 시스템 동작 환경 설정

- ❖ 부트 방법, 부트 디바이스를 비롯한 네트워크를 이용한 부트를 지원하기 위한 네트워크 설정, IP 주소 설정 등 부트로더 동작에 필요한 정보를 설정
- ❖ 설정된 환경 변수 값은 플래시 메모리 또는 EEPROM에 저장 관리



# 부트로더의 역할 (2)

## □ 시스템 운영체제 부팅

- ❖ 일반적으로 임베디드 시스템의 운영체제는 플래시 메모리에 탑재되어 있고 부팅과정에서 주 메모리(일반적으로 **DRAM**을 사용)에 탑재하여 실행
    - 운영체제를 **DRAM**에 복사하고 제어권을 운영체제의 시작점으로 넘겨주는 기능 필요
  - ❖ 시리얼, 네트워크 또는 **USB**를 이용한 실행하는 방법 지원
    - 네트워크는 **TFTP**(Trivial File Transfer Protocol)을 사용
    - **USB**는 네트워크에 비하여 별도 설정 없이누구나 쉽게 사용 가능
    - 시리얼 장치(**UART**)를 이용한 탑재 방법 사용
- ✓ 속도가 늦어 용량이 큰 운영체제를 탑재하기에는 부적합

## □ 플래시 메모리 관리

- ❖ 임베디드 시스템에서 가장 효율적인 보조 기억 장치
- ❖ 부트로더 탑재, 시스템에 전원이 인가되면 플래시에 저장된 부트로더 실행
- ❖ 부트로더 및 시스템 동작에 필요한 환경 변수 저장
- ❖ 플래시 메모리에는 시스템 운영체제 이미지 탑재

## □ 모니터 기능

- ❖ 시스템의 동작 상태를 감시
- ❖ **POST**(Power-On Self Test)
  - 하드웨어 정상 동작 여부 검사, 메모리 검사 등

# 부트로더의 특징

---

## □ 프로세서에 의존성이 높다

- ❖ 특히 시스템 초기화 과정의 모든 코드는 어셈블러로 작성된다.
- ❖ 프로그래머는 프로세서의 **Architecture**와 **Programmer's Model** 대하여 이해를 하고 있어야 한다.

## □ **TARGET** 시스템의 하드웨어에 의존성이 높다

- ❖ 프로그래머는 **TARGET** 시스템의 하드웨어를 이해하여야 한다.
- ❖ 특히 메모리 인터이스, 주변 장치 등에 대해서는 잘 알고 있어야 한다.

## □ 가능하면 적은 크기 이어야 한다.

## □ 대부분은 사용자가 직접 작성하여 사용한다.

- ❖ 대부분의 부트로더 코드는 사용자가 개발하려는 시스템의 특성에 맞게 작성하여 사용
- ❖ 다운로드 방법, **OS** 시작 방법 등
- ❖ 실습에서는 **U-Boot**를 사용 (이미 개발되어 공개되어 있는 부트로더 임)

# 사용 가능한 다양한 부트로더

---

## ❑ LILO

- ❖ LILO(Linux Loader)는 x86기반의 컴퓨터 기반의 리눅스에서 오래 전부터 사용되고 있는 부트로더이다.

## ❑ GRUB

- ❖ GRUB(Grand Unified Bootloader)는 GNU 프로젝트의 부트로더

## ❑ EtherBoot

- ❖ 디스크 없는 시스템에서 이더넷을 통해 부팅할 수 있게 하는 부트로더
- ❖ 주로 X-터미널 등에서 많이 사용

## ❑ Blob

- ❖ Blob는 ARM 프로세서 기반의 LART 하드웨어를 위한 부트로더로 개발

## ❑ PMON

- ❖ PMON(PROM Monitor)은 MIPS 보드를 지원하기 위해 제작된 부트로더

## ❑ RedBoot

- ❖ 레드햇에서 개발되어 배포된 부트로더

## ❑ U-Boot

- ❖ U-Boot는 PPCBoot와 ARMBot 프로젝트로 기반으로 개발
- ❖ 다양한 PPC 또는 ARM 프로세서 기반의 타겟 보드 지원

# U-Boot

---

## □ U-Boot 란

- ❖ U-Boot는 ARM이나 PowerPC와 같은 프로세서 기반의 임베디드 시스템에서 동작 하도록 작성된 부트로더이다.

## □ 기능

- ❖ 하드웨어 초기화
- ❖ 하드웨어 테스트
- ❖ 소프트웨어 다운로드 및 실행
- ❖ FLASH 관리
- ❖ OS 부팅

# U-Boot의 특징

---

## □ Linux와 유사한 구조

- ❖ U-Boot는 Linux와 유사한 구조를 가지고 있다.
- ❖ 일부 소스는 Linux용을 사용
- ❖ Linux 이미지를 부팅하기 쉽도록 되어 있다.

## □ 확장이 용이하다

- ❖ 새로운 Command 추가가 쉽다
- ❖ Configuration에 의한 다이내믹 한 기능 추가 및 확장 제공

# 목 차

---

11장. 소프트웨어 개발 툴의 이해와 활용

12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

13장. 시스템 리셋과 부트코드

14장. 하드웨어 제어

□ 15장. 부트로더 개발

01. 부트로더 이해

02. U-Boot 빌드와 설치

03. U-Boot 활용

04. U-Boot 포팅

# U-Boot 소스

---

- **U-Boot 소스**는 다음 사이트에서 구할 수 있다

<http://sourceforge.net/projects/u-boot>

- **TARGET MACHINE** 및 보드용 소스 준비

1. U-Boot 소스를 WEB에서 얻는다.
2. 유사한 TARGET MACHINE의 소스를 개발하고자 하는 TARGET에 맞도록 복사하고 설정 환경을 만든다.
3. U-Boot를 Configuration 하고 빌드하여 최종 이미지를 얻는다.
4. TARGET에 탑재하고 실행 하면서 보드에 맞도록 포팅한다.

- **U-Boot 포팅**

- ❖ U-boot는 범용으로 많이 사용 되므로 일부 유명한 프로세서는 포팅되어 있는 경우도 있지만 대부분의 사용자가 사용하고자 하는 프로세서 및 보드에 맞도록 포팅하여 사용 하여야 한다.

# U-Boot 소스 준비

□ 실습에서는 **CD**에 있는 소스를 이용하여 사용한다

❖ 저장 위치 : /mnt/cdrom/Software/ARM\_Linux/BootLoader/u-boot-1.1.1.tar.gz

□ **U-Boot** 소스 설치

\$ tar zxvf /mnt/cdrom/Software/ARM\_Linux/BootLoader/u-boot-1.1.1.tar.gz

```
# tar zxvf /mnt/cdrom/Software/ARM_Linux/BootLoader/u-boot-1.1.1.tar.gz
u-boot-1.1.1/
u-boot-1.1.1/board/
u-boot-1.1.1/board/AtmarkTechno/
u-boot-1.1.1/board/AtmarkTechno/suzaku/
u-boot-1.1.1/board/AtmarkTechno/suzaku/Makefile
u-boot-1.1.1/board/AtmarkTechno/suzaku/config.mk
u-boot-1.1.1/board/AtmarkTechno/suzaku/flash.c
..... 이하 생략 .....
```



# U-Boot 빌드

## □ BUILD 절차는 다음과 같다.

### 1. BUILD Configuration

```
$ cd u-boot-1.1.1
```

```
$ make dtk2410_config
```

### 2. 의존성(Dependency) 검사

```
$ make dep
```

### 3. BUILD U-Boot

```
$ make
```

Configuration 전에 U-boot 디렉토리의 Makefile에 TARGET 설정 정보를 만들어 주어야 한다.

Makefile에 아래 내용을 추가

```
dtk2410_config : unconfig
```

```
@./mkconfig $(@:_config=) arm arm920t dtk2410
```

## □ BUILD 후 생성되는 파일

파일명	설명
u-boot.map	U-Boot memory map 파일
u-boot	ELF binary format의 U-Boot 이미지
u-boot.bin	Plane binary format의 U-Boot 이미지, FLASH에 탑재되는 이미지

# 부트로더 탑재

---

## □ 부트로더 탑재

- ❖ 개발된 부트로더는 **TARGET** 보드의 **FLASH**에 탑재되어 부트로더 기능을 수행
- ❖ **TARGET**에 부트로더 탑재하는 방법
  - ROM 라이터를 사용하는 방법
  - 전용 **ICE** 장비를 사용하는 방법
  - **JTAG Dongle**을 활용하는 방법
  - 기존에 탑재된 부트로더를 이용하여 탑재하는 방법

## □ **JTAGProbe™** 활용

- ❖ **JTAGProbe**는 **Wiggler**와 호환되는 **JTAG Dongle**
- ❖ **JTAGProbe** 기능
  - **ARM**의 내부 레지스터 읽고 쓰기 가능
  - 메모리 내용 읽고 쓰기 가능
  - 프로그램 **RUN** 또는 **STOP**
  - 사용자 프로그램 로딩
  - **FLASH**에 부트로더 탑재 등

# JTAGProbe™ 유틸리티 빌드

## □ JTAGProbe 소스

- ❖ 부록CD의 리눅스 용 툴 폴더에 있다

*\$ tar zxvf /mnt/cdrom/Tools/Linux/JTAGProbe/JTAGProbe.tar.gz*

## □ JTAGProbe 빌드

- ❖ 반드시 루트 권한으로 빌드 해야 한다.

*\$ cd JTAGProbe*

*\$ make*

## □ JTAGProbe 실행

*\$ ./jtagprobe*

## □ JTAGProbe 명령

- ❖ [표 15-2]참조

*# ./jtagprobe*

*JTAGProbe – JTAG interface to the ARM9TDMI/ARM7TDMI  
DIGNSYS Inc. (www.dignsys.com)*

*Jtag - Using port 0x378*

*JTAG interface reset.*

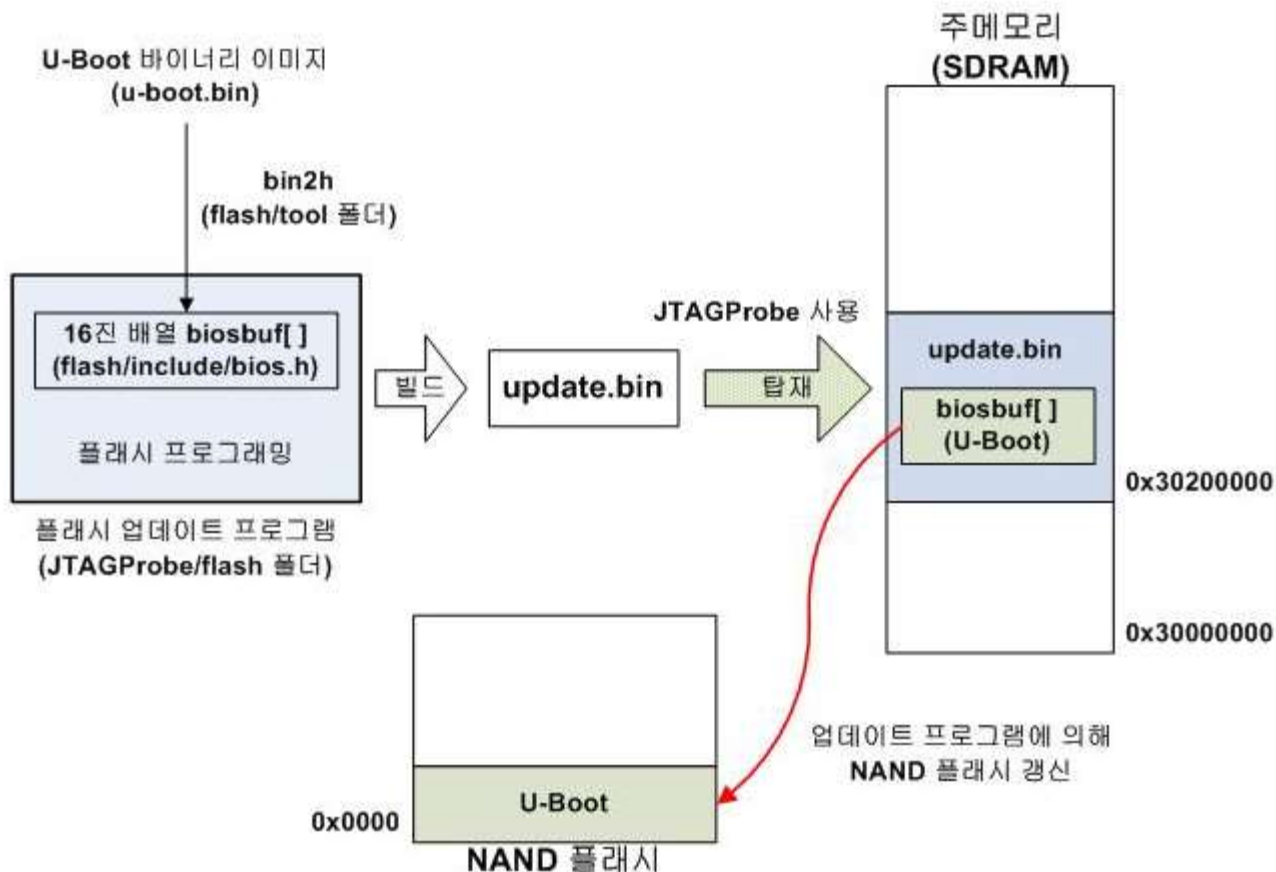
*Device ID..0x0032409D*

*S3C2410 Core Detected!*

*Jtag:>*

# 플래시 업데이트 프로그램

## □ “JTAGProbe/flash” 폴더에 제공



# 플래시 업데이트 프로그램 빌드

```
# cd flash
# make
```

```
make[1]: Entering directory `/root/JTAGProbe/flash/tool'
gcc -O2 -c -o bin2h.o bin2h.c
gcc -O2 -s -o bin2h bin2h.o
rm -f bin2h.o
./bin2h ../../../u-boot-1.1.1/u-boot.bin ../include/bios.h
make[1]: Leaving directory `/root/JTAGProbe/flash/tool'
make -C lib
```

..... 이하 생략 .....

```
arm-linux-gcc -O2 -fomit-frame-pointer -I/root/JTAGProbe/flash/include -Wall -mapcs-32 -
mcpu=arm9tdmi -DELFB -DVERSION=\"1.0\" -DCONFIG_ARCH_S3C2410 -
DCONFIG_NAND_FLASH -D__ASSEMBLY__ -c -o crt0.o crt0.S
make[1]: Leaving directory `/root/JTAGProbe/flash/update'
arm-linux-ld -N -Ttext 0x30200000 -T elf.lds -o update.elf update/crt0.o update/main.o lib/lib.a
`arm-linux-gcc -O2 -fomit-frame-pointer -I/root/JTAGProbe/flash/include -Wall -mapcs-32 -
mcpu=arm9tdmi -DELFB -DVERSION=\"1.0\" -DCONFIG_ARCH_S3C2410 -
DCONFIG_NAND_FLASH --print-libgcc-file-name`
arm-linux-objcopy -I elf32-littlearm -O binary update.elf update.bi
```

# JTAGProbe™ 실행

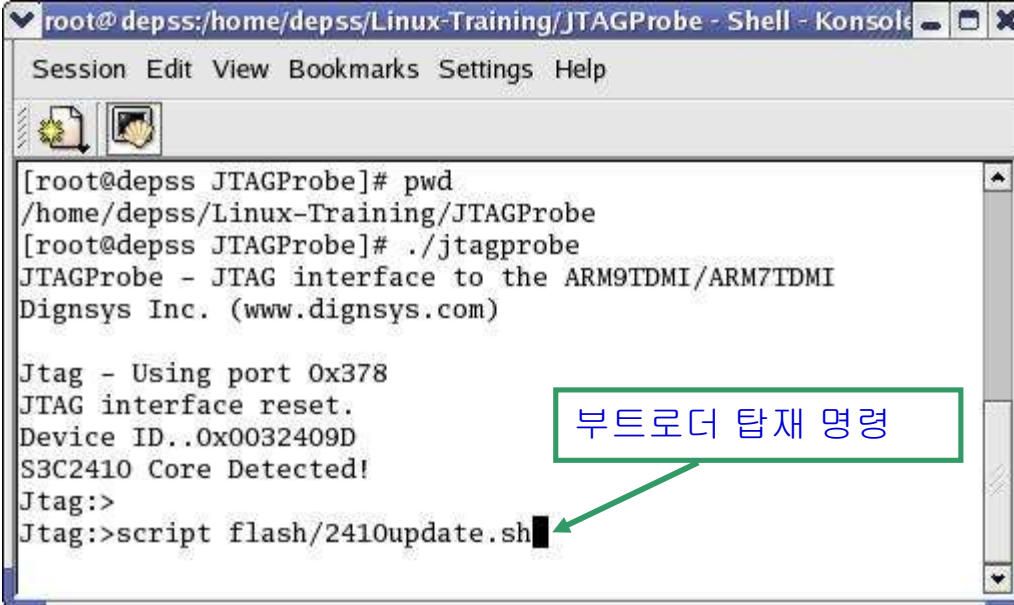
## □ BUILD한 JTAGProbe를 실행한다

- ❖ JTAGProbe는 ROOT 권한으로 실행하고, 차후 부트로더 실습이 완료될 때까지 여러 번 사용되므로 별도의 창을 열어서 사용

\$ su -

Password:

# ./jtagprobe



```
root@ depss:/home/depss/Linux-Training/JTAGProbe - Shell - Konsole
Session Edit View Bookmarks Settings Help

[root@depss JTAGProbe]# pwd
/home/depss/Linux-Training/JTAGProbe
[root@depss JTAGProbe]# ./jtagprobe
JTAGProbe - JTAG interface to the ARM9TDMI/ARM7TDMI
Dignsys Inc. (www.dignsys.com)

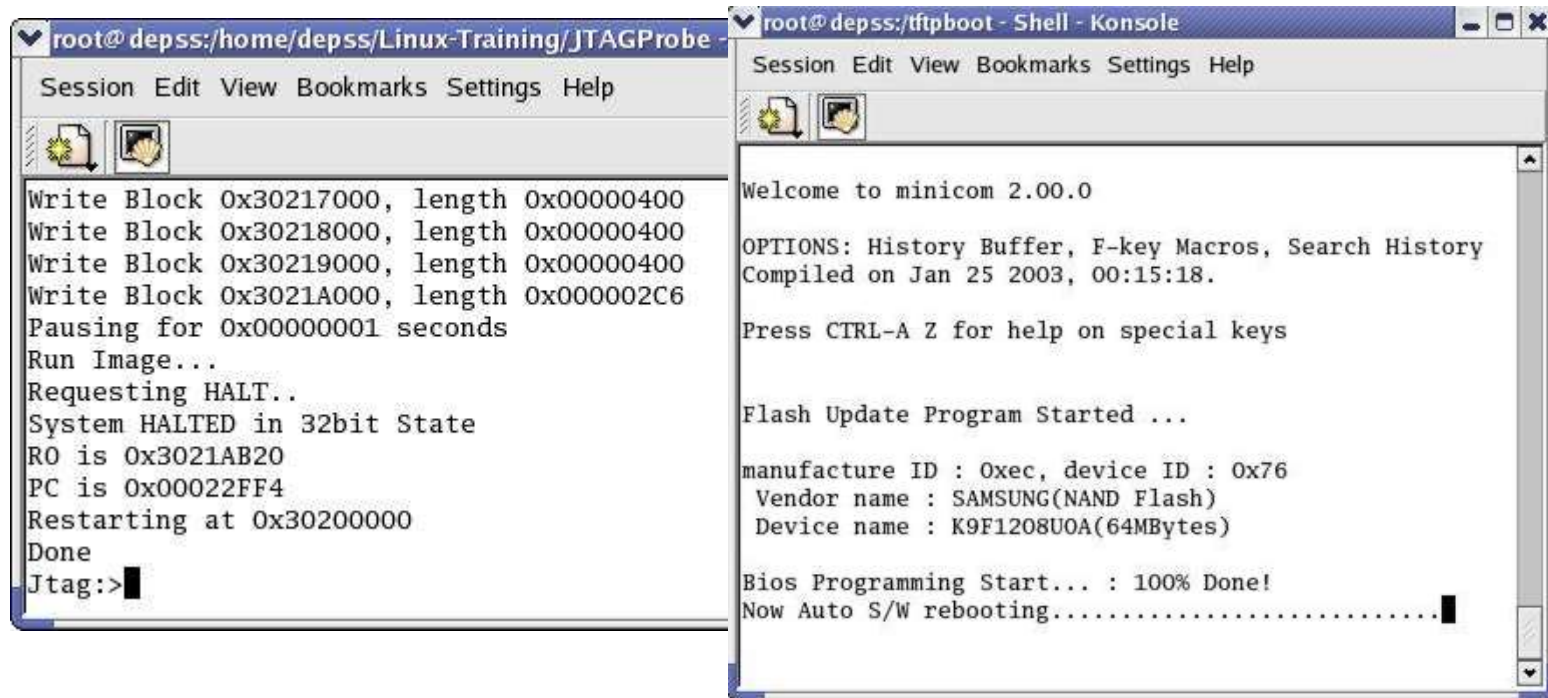
Jtag - Using port 0x378
JTAG interface reset.
Device ID..0x0032409D
S3C2410 Core Detected!
Jtag:>
Jtag:>script flash/2410update.sh
```

# 플래시 업데이트와 부트로더 탑재

## □ JTAGProbe에서 FLASH에 DTK2410 부트로더 탑재

- ❖ Script에 의해서 자동으로 upgrade 프로그램을 다운로드 하고 부트로더를 탑재하는 기능 수행

Jtag:> script flash/2410upgrade.sh



```
root@depss:/home/depss/Linux-Training/JTAGProbe -  
Session Edit View Bookmarks Settings Help  
Write Block 0x30217000, length 0x00000400  
Write Block 0x30218000, length 0x00000400  
Write Block 0x30219000, length 0x00000400  
Write Block 0x3021A000, length 0x000002C6  
Pausing for 0x00000001 seconds  
Run Image...  
Requesting HALT..  
System HALTED in 32bit State  
R0 is 0x3021AB20  
PC is 0x00022FF4  
Restarting at 0x30200000  
Done  
Jtag:>  
  
root@depss:/tftpboot - Shell - Konsole  
Session Edit View Bookmarks Settings Help  
Welcome to minicom 2.00.0  
OPTIONS: History Buffer, F-key Macros, Search History  
Compiled on Jan 25 2003, 00:15:18.  
Press CTRL-A Z for help on special keys  
Flash Update Program Started ...  
manufacture ID : 0xec, device ID : 0x76  
Vendor name : SAMSUNG(NAND Flash)  
Device name : K9F1208U0A(64MBytes)  
Bios Programming Start... : 100% Done!  
Now Auto S/W rebooting.....
```

# 부트로더 부팅

- 탐색이 완료되고 리셋키를 누르면 다음과 같은 메시지를 확인할 수 있다.

```
root@depss:/tftpboot - Shell - Konsole
Session Edit View Bookmarks Settings Help

U-Boot 1.1.1 (Aug 17 2005 - 20:14:18)

U-Boot code: 33F80000 -> 33F98DF0 BSS: -> 33F9CEC4
RAM Configuration:
Bank #0: 30000000 64 MB
Flash: 512 kB
*** Warning - bad CRC, using default environment

In: serial
Out: serial
Err: serial
Hit any key to stop autoboot: 0
DTK2410 #
DTK2410 #
DTK2410 #
DTK2410 #
```

메모리 정보

지정한 프롬프트



# 목 차

---

11장. 소프트웨어 개발 툴의 이해와 활용

12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

13장. 시스템 리셋과 부트코드

14장. 하드웨어 제어

□ 15장. 부트로더 개발

01. 부트로더 이해

02. U-Boot 빌드와 설치

03. U-Boot 활용

04. U-Boot 포팅

# 환경 변수 설정

---

□ [표 15-3]과 [표 15-4] 참조

□ 환경 변수 설정 명령

명령	용도	사용법
printenv	현재 설정된 환경 변수 출력	printenv
saveenv	설정된 환경 변수 저장	saveenv
setenv	환경 변수 설정	setenv [변수명] [설정값]

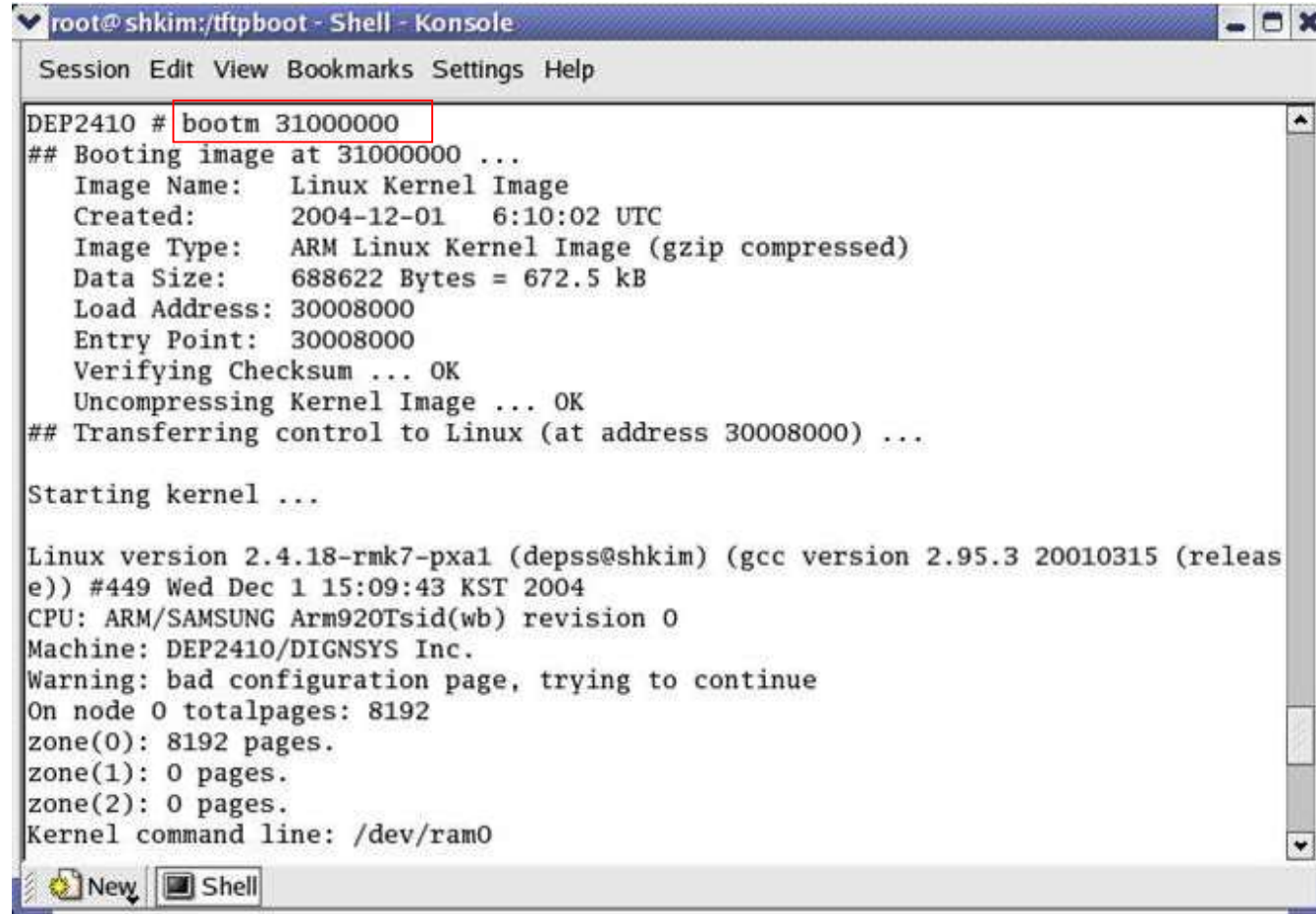
# 다운로드

- 운영체제 커널, 디스크 이미지를 비롯한 파일을 주 메모리에 탑재

명령	용도	사용법	예제
loadb	시리얼로 다운로드	loadb [주소] [속도]	loadb 3100000 115200
tftp	네트워크로 다운로드	tftp [주소] [파일명]	tftp 3100000 ulmage
loadub	USB로 다운로드	loadub [주소] [파일명]	loadub 3100000 ulmage

# 임베디드 리눅스 부팅

## □ “bootm” 명령 사용



```
root@shkim:/tftpboot - Shell - Konsole
Session Edit View Bookmarks Settings Help
DEP2410 # bootm 31000000
## Booting image at 31000000 ...
Image Name:   Linux Kernel Image
Created:      2004-12-01  6:10:02 UTC
Image Type:   ARM Linux Kernel Image (gzip compressed)
Data Size:    688622 Bytes = 672.5 kB
Load Address: 30008000
Entry Point:  30008000
Verifying Checksum ... OK
Uncompressing Kernel Image ... OK
## Transferring control to Linux (at address 30008000) ...

Starting kernel ...

Linux version 2.4.18-rmk7-pxa1 (depss@shkim) (gcc version 2.95.3 20010315 (release)) #449 Wed Dec 1 15:09:43 KST 2004
CPU: ARM/SAMSUNG Arm920Tsid(wb) revision 0
Machine: DEP2410/DIGNSYS Inc.
Warning: bad configuration page, trying to continue
On node 0 totalpages: 8192
zone(0): 8192 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: /dev/ram0
```

# NAND 플래시 관리

- ❑ **NAND** 플래시는 대용량의 데이터 저장을 비롯하여 프로그램 저장 용도로 많이 사용되는 플래시 메모리
- ❑ **NOR** 플래시와 달리 기록할 때뿐만 아니라 읽을 때에도 블록 단위로 처리
- ❑ **NAND** 플래시 관리 명령

명령	용도
nand info	유효한 NAND 디바이스 정보 출력
nand device [dev]	사용되는 디바이스 설정 및 출력
nand read [addr] [off] [size]	NAND에서 주메모리로 데이터를 읽는다
nand write [addr] [off] [size]	주메모리의 데이터를 NAND 플래시에 기록한다
nand erase [off] [size]	NAND의 내용을 지운다
nand bad	잘못된 블록 출력

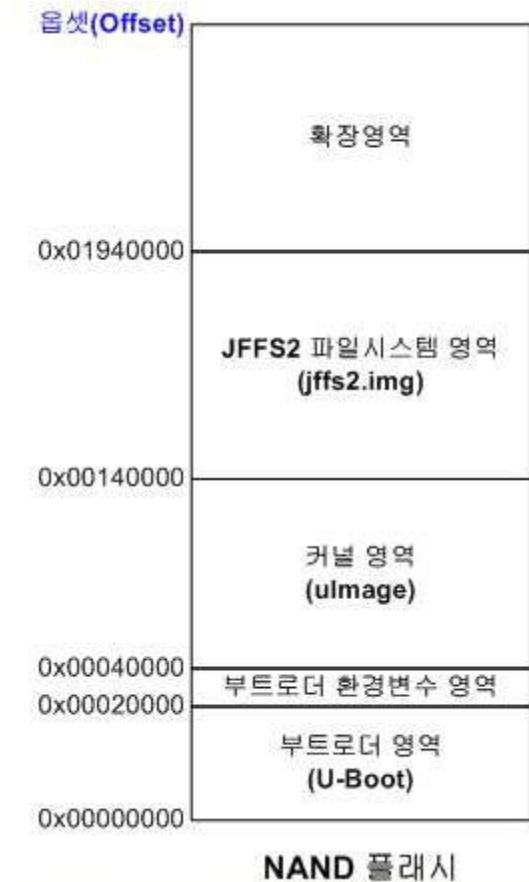
# 시스템 이미지 업데이트

## □ 시스템 이미지는 운영체제 커널과 응용 프로그램을 의미

- ❖ 임베디드 리눅스의 경우 리눅스 커널 이미지와 루트파일시스템을 의미

## □ NAND 플래시 메모리에 커널 이미지와 루트파일시스템 이미지를 탑재하는 방법

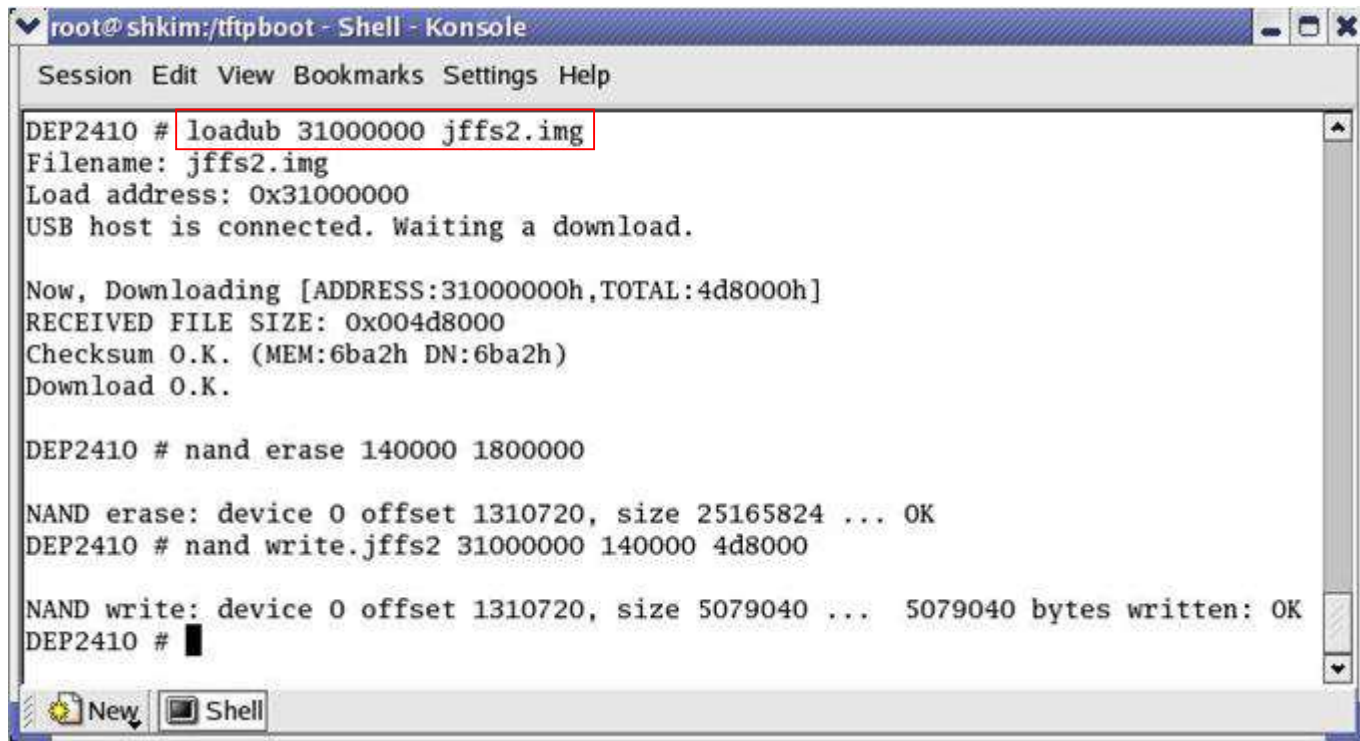
- ❖ 우측과 같은 맵에 시스템 이미지 저장
- ❖ 이미지 다운로드 한 후 기록



# JFFS2 파일시스템 업데이트 예

## □ JFFS2 파일시스템 이미지

❖ “jffs2.img”



```
root@shkim:/tftpboot - Shell - Konsole
Session Edit View Bookmarks Settings Help
DEP2410 # loadub 31000000 jffs2.img
Filename: jffs2.img
Load address: 0x31000000
USB host is connected. Waiting a download.

Now, Downloading [ADDRESS:31000000h,TOTAL:4d8000h]
RECEIVED FILE SIZE: 0x004d8000
Checksum O.K. (MEM:6ba2h DN:6ba2h)
Download O.K.

DEP2410 # nand erase 140000 1800000

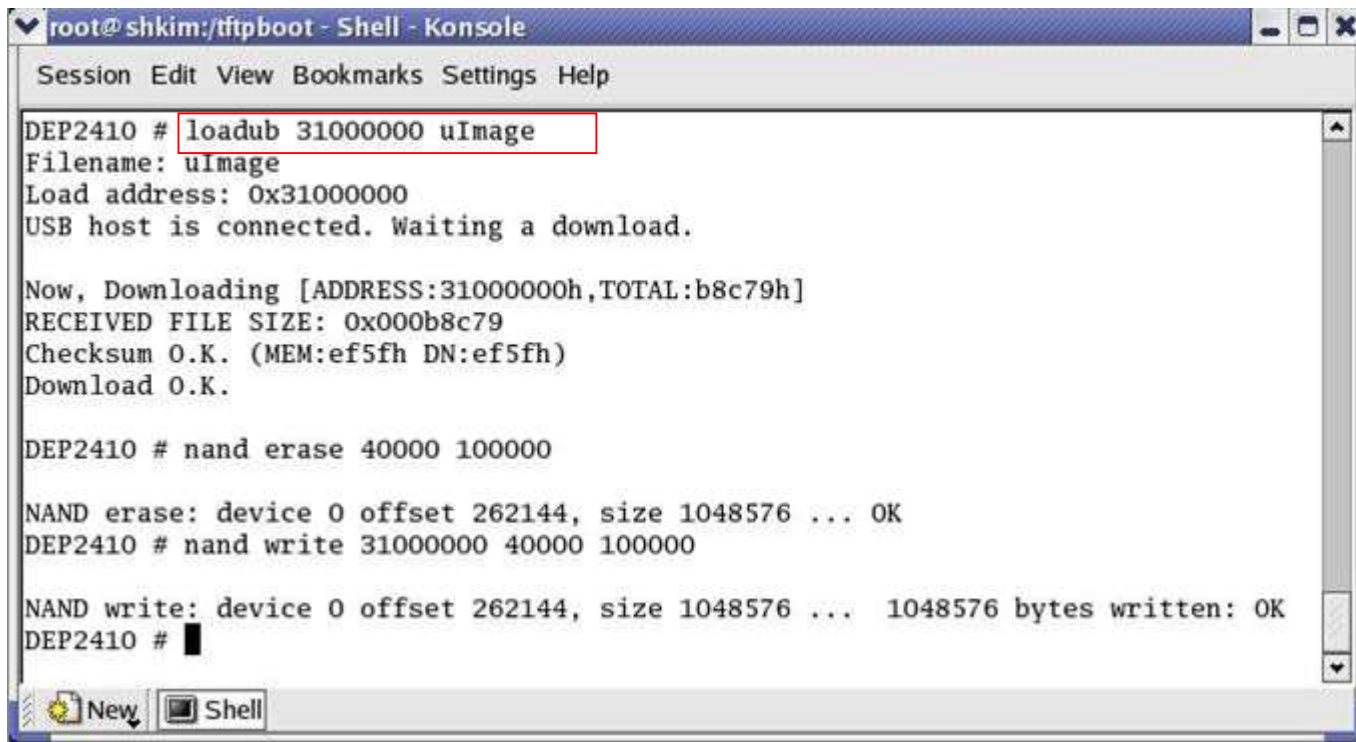
NAND erase: device 0 offset 1310720, size 25165824 ... OK
DEP2410 # nand write.jffs2 31000000 140000 4d8000

NAND write: device 0 offset 1310720, size 5079040 ... 5079040 bytes written: OK
DEP2410 # █
```

# 커널 이미지 업데이트 예

## □ 커널 이미지

### ❖ “uImage”



```
root@shkim:/tftpboot - Shell - Konsole
Session Edit View Bookmarks Settings Help

DEP2410 # loadub 31000000 uImage
Filename: uImage
Load address: 0x31000000
USB host is connected. Waiting a download.

Now, Downloading [ADDRESS:31000000h,TOTAL:b8c79h]
RECEIVED FILE SIZE: 0x000b8c79
Checksum O.K. (MEM:ef5fh DN:ef5fh)
Download O.K.

DEP2410 # nand erase 40000 100000

NAND erase: device 0 offset 262144, size 1048576 ... OK
DEP2410 # nand write 31000000 40000 100000

NAND write: device 0 offset 262144, size 1048576 ... 1048576 bytes written: OK
DEP2410 #
```



# 자동 부트 지원

## □ 자동 부트

- ❖ 전원이 인가되거나 리셋이되면 부트로더가 스스로 플래시 메모리나 호스트 컴퓨터에 있는 이미지를 주 메모리에 탑재하여 실행하는 것

## □ 자동 부팅과 관련된 환경 변수

### ❖ “bootcmd”

- U-Boot의 초기화 완료 후 자동으로 실행되는 명령을 지정하며

### ❖ “bootdelay”

- bootcmd를 실행하기 전 대기 시간을 지정

```
DEP2410# setenv bootcmd 'nand read 31000000 40000 100000; bootm 31000000'  
DEP2410# saveenv
```

# 목 차

---

11장. 소프트웨어 개발 툴의 이해와 활용

12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

13장. 시스템 리셋과 부트코드

14장. 하드웨어 제어

□ 15장. 부트로더 개발

01. 부트로더 이해

02. U-Boot 빌드와 설치

03. U-Boot 활용

04. U-Boot 포팅

# U-Boot 폴더 구조(1)

디렉토리	DESCRIPTION
<b>board</b>	보드 관련된 파일이 있는 디렉토리
<b>board/dep2410</b>	<b>DEP2410</b> 보드에 관련된 디렉토리, <b>TARGET</b> 추가 시 만들어 주어야 한다.
<b>common</b>	<b>Architecture</b> 나 <b>TARGET</b> 에 무관하도록 구성되는 소프트웨어로 <b>Command</b> 등이 이 디렉토리에 만들어져 있다.
<b>cpu</b>	프로세서나 <b>CPU</b> 관련된 파일을 가지고 있는 디렉토리
<b>cpu/arm920t</b>	<b>ARM920T</b> 프로세서나 <b>S3C2410 CPU</b> 관련된 파일을 가지고 있는 디렉토리
<b>disk</b>	디스크 드라이버 및 파티션 관련된 정보를 가지고 있는 디렉토리
<b>doc</b>	<b>U-Boot</b> 관련된 문서를 가지고 있다

## U-Boot 폴더 구조(2)

디렉토리	DESCRIPTION
<b>drivers</b>	각종 디바이스 드라이버를 가지고 있는 디렉토리
<b>examples</b>	<b>U-Boot stand-alone</b> 응용 프로그램을 가지고 있는 디렉토리
<b>include</b>	<b>U-Boot</b> 에서 사용되는 헤더 정보
<b>Include/config</b>	보드 <b>Configuration</b> 을 위한 헤더 파일을 가지고 있는 디렉토리
<b>net</b>	네트워킹 코드
<b>post</b>	<b>Power On Self Test</b>
<b>rtc</b>	<b>Real Time Clock</b> 드라이버
<b>tools</b>	<b>S-Record</b> 나 <b>U-Boot</b> 이미지를 생성하기 위한 툴

# 대상 시스템에 필요한 소스 (1)

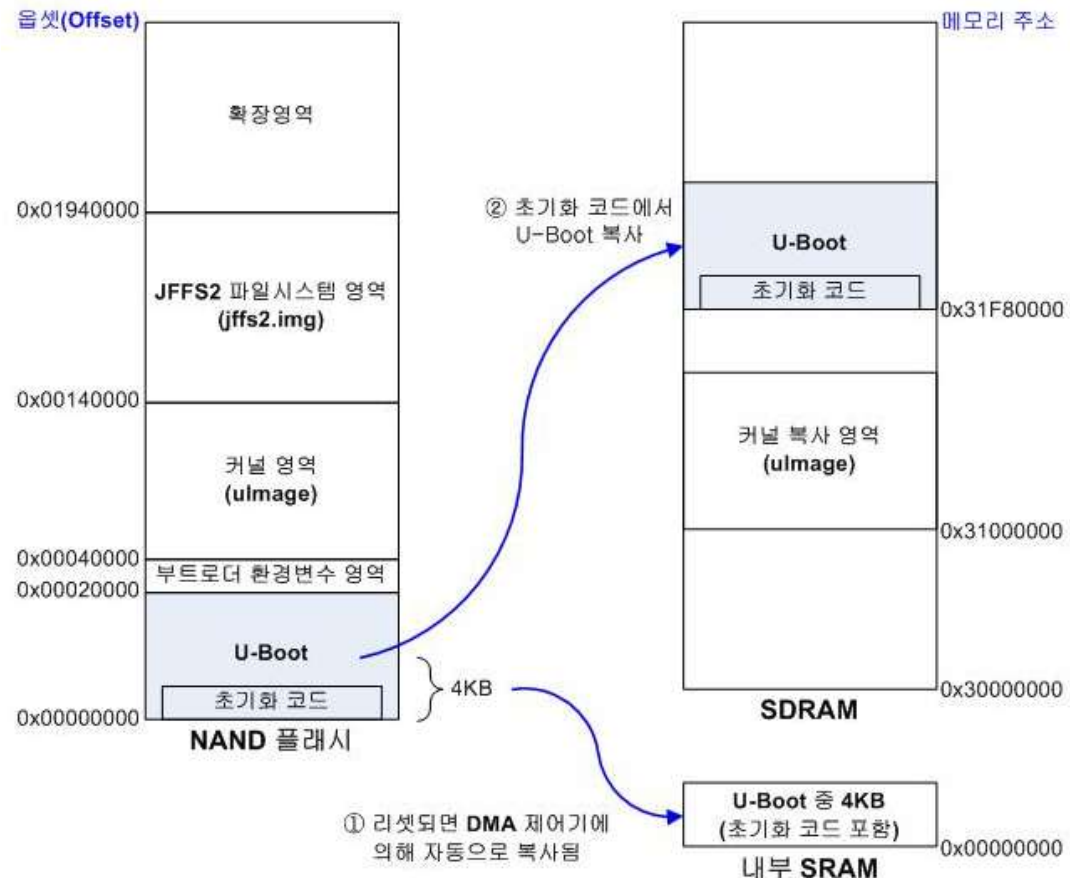
디렉토리와 파일			DESCRIPTION
include	s3c2410.h, s3c24x0.h		S3C2410 SoC(MCU)의 레지스터를 정의, 매크로 선언 등
	configs	dep2410.h	메모리 정보, 부트 형태 등 Configuration 정보를 설정한다.
cpu	arm920t	start.S	U-Boot의 시작부분, Exception 핸들러를 가지고 있으며 시스템 초기화를 시작한다.
		serial.c	부트로더에서 사용되는 시리얼 드라이버
		interrupt.c	인터럽트 제어기를 관리
		cpu.c	Cache와 같은 CPU의 자원을 제어하기 위한 소스

## 대상 시스템에 필요한 소스 (2)

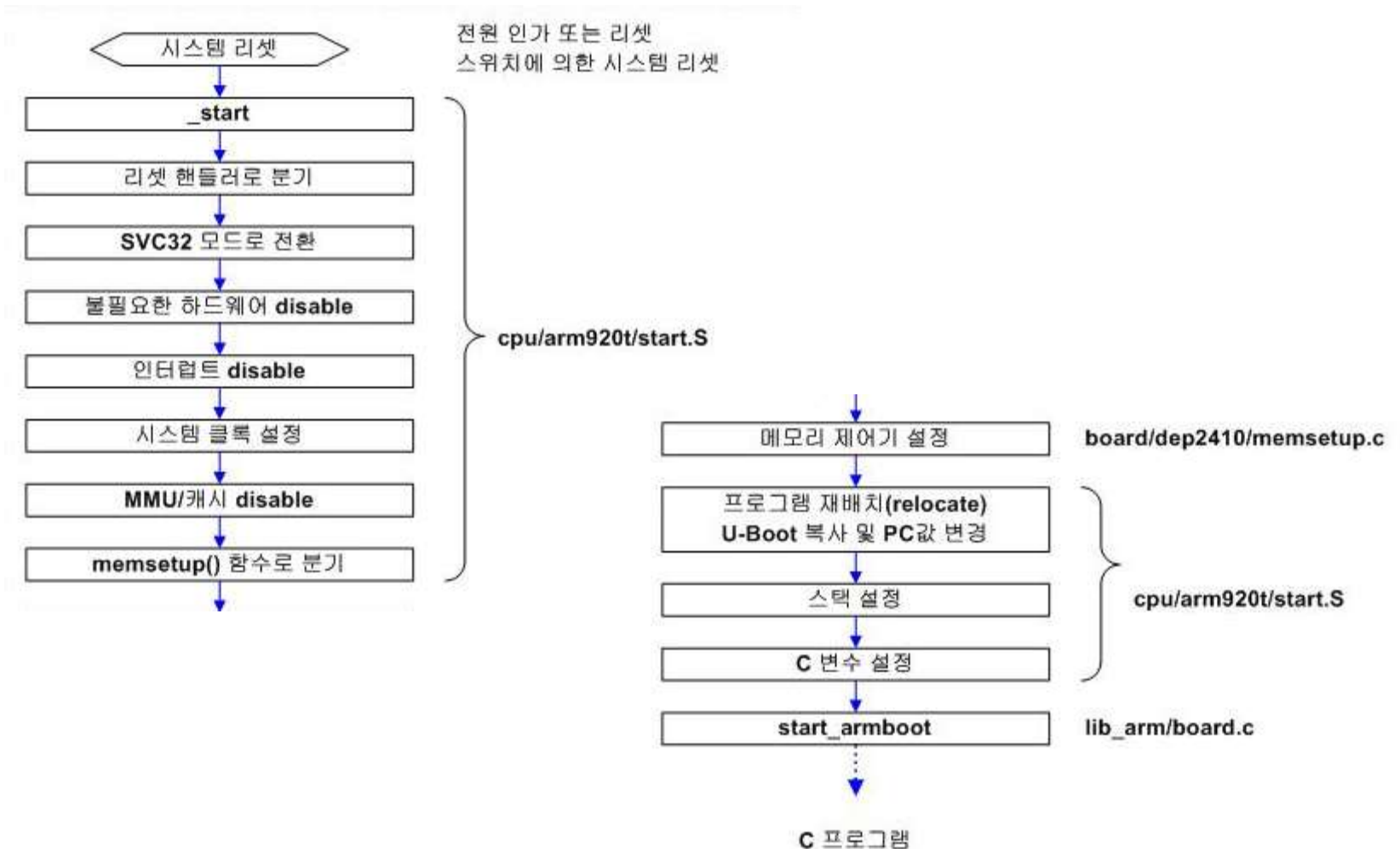
디렉토리와 파일			DESCRIPTION
board	dep2410	config.mk	text(code)의 BASE 어드레스 지정
		u-boot.lds	U-Boot용 linker script, 프로그램 동작을 위한 메모리 배치 관리
		memsetup.S	TARGET 보드에 맞도록 메모리 제어기 설정
		dep2410.c	TARGET 보드 초기화
		flash.c	FLASH 메모리를 제어
drivers	smc91111.c		U-Boot를 위한 LAN91C111 Ethernet 드라이버, TFTP 로딩에 사용된다.
	smc91111.h		LAN91C111 Ethernet 드라이버를 위한 헤더 정보

# U-Boot의 구조와 동작 이해

- U-Boot는 시스템에 리셋 키가 입력되거나 전원이 인가되면 실행되는 부트코드
- 부트코드 동작과 메모리 배치
  - ❖ 옆 그림 참조

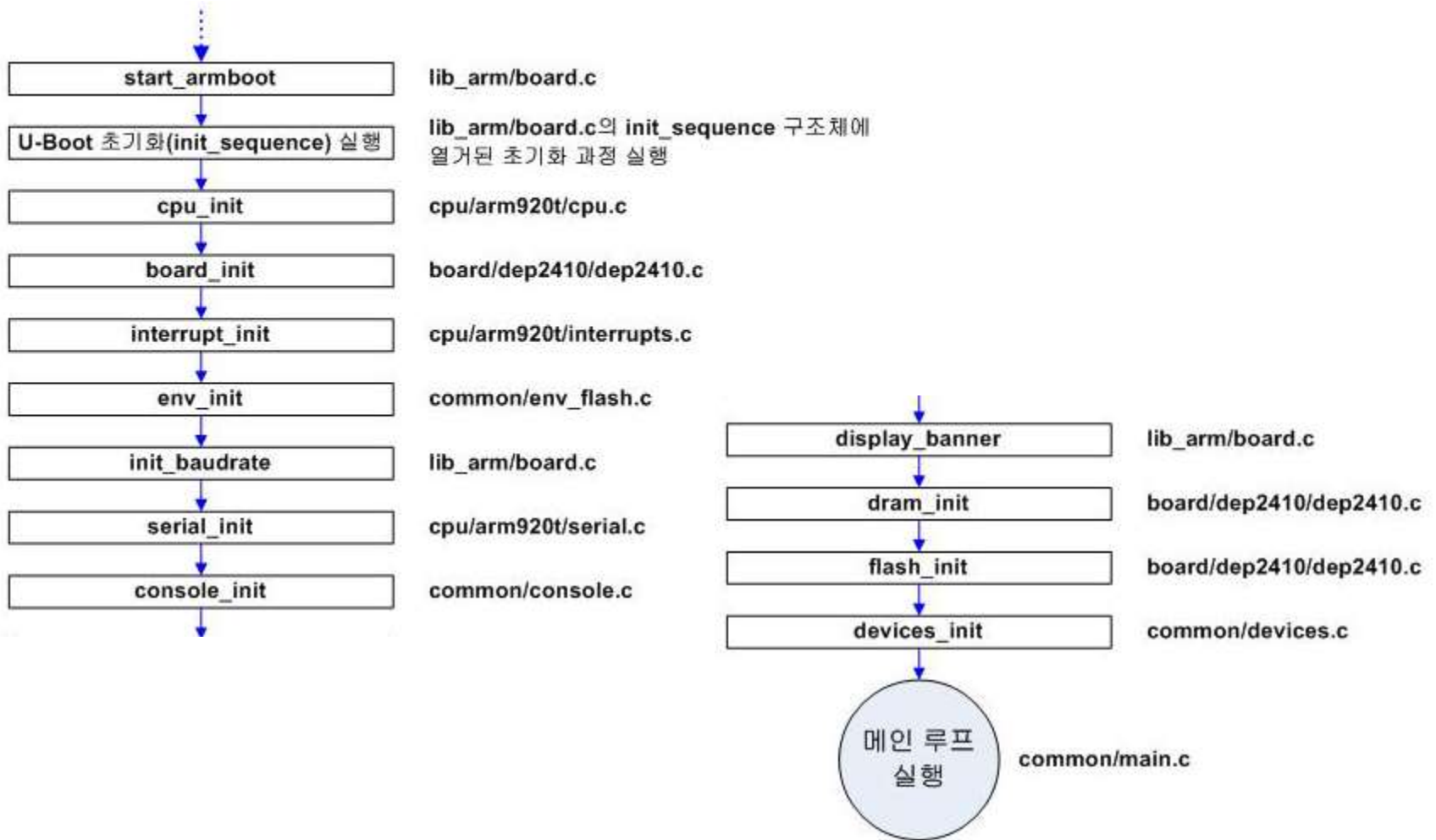


# U-Boot 초기화 과정 (1)





## U-Boot 초기화 과정 (2)



# U-Boot 포팅

---

□ 새로운 시스템 보드에서 정상 동작 하도록 부트로더를 개발 하는 과정

□ 실습용 **DTK2410** 보드의 사양

❖ 보드명 : DTK2410

❖ SoC (CPU) : S3C2410 (삼성)

❖ 프로세서 : ARM920T

□ **DTK2410** 추가

❖ 현재 실습에 사용되는 **DTK2410**은 삼성의 **S3C2410**을 사용하고, **S3C2410**은 **ARM920T** 프로세서를 사용한다.

❖ **S3C2410**으로 포팅 된 소스를 쉽게 적용하여 사용할 수 있다.

❖ 하지만 **U-Boot**에 새로운 프로세서와 **SoC**를 추가하려면 실습 동안 추가되는 소스의 많은 부분은 새로운 프로세서 및 **SoC**에 맞도록 포팅 하여야 한다.

# 새로운 대상 보드용 소스 추가 (1)

디렉토리와 파일			추가 방법
include	s3c2410.h, s3c24x0.h		이미 S3C2410용으로 준비되어 있는 파일로 기존 파일을 그대로 사용
	configs	dtk2410.h	같은 S3C2410을 사용한 smdk2410.h를 dtk2410.h로 복사
cpu	arm920t	start.S	그대로 사용
		serial.c	그대로 사용
		interrupt.c	그대로 사용, 단 get_tbclk() 함수의 CONFIG_SMDK2410으로 선언된 부분에 CONFIG_DTK2410 추가
		cpu.c	그대로 사용

## 새로운 대상 보드용 소스 추가 (2)

디렉토리와 파일		추가 방법
board		“smdk2410” 디렉토리를 “dtk2410” 디렉토리로 복사
	config.mk	복사된 파일 그대로 사용
	u-boot.lds	복사된 파일 그대로 사용
	memsetup.S	복사된 파일 그대로 사용
	dtk2410.c	기존에 복사된 smdk2410.c를 dtk2410.c로 변경
	flash.c	복사된 파일 그대로 사용
	Makefile	“OBJS := dtk2410.o flash.o”로 변경
drivers	smc91111.c	그대로 사용
	smc91111.h	그대로 사용

# 소프트웨어와 하드웨어 설정

---

- ❑ **C Preprocessor**를 이용한 변수를 정의하여 설정
- ❑ **TARGET MACHINE**의 하드웨어와 소프트웨어 관련된 설정은
  - ❖ “u-boot-1.1.1/include/configs/dtk2410.h”
- ❑ **U-Boot**는 2개 **Class**의 **Configuration** 변수를 가진다
  - ❖ Configuration OPTIONS
    - 사용자가 소프트웨어를 설정할 때 사용하는 변수로 “CONFIG\_”로 시작되는 변수 들이 이에 해당한다
  - ❖ Configuration SETTINGS
    - 하드웨어와 관련된 설정을 할 때 사용하는 변수로 “CFG\_”로 시작 되는 변수들이 이에 해당한다.

# “dtk2410.h” 파일 (1)

---

## 1. 보드 Configuration

- ❖ SMDK2410 보드의 Configuration 파일을 복사 했으므로 DTK2410 보드에 대한 Configuration이 되도록 수정한다.

```
CONFIG_SMDK2410 => CONFIG_DTK2410
```

## 2. 하드웨어 드라이버

- ❖ DTK2410은 SMC91111을 사용하므로 이에 맞게 수정한다.

```
#define CONFIG_DRIVER_SMC91111  
#define CONFIG_SMC91111_BASE      0x10000300  
#undef CONFIG_SMC91111_EXT_PHY  
#undef CONFIG_SMC_USE_32_BIT
```

# “dtk2410.h” 파일 (2)

## 3. 부팅과 네트워크 인터페이스 설정

- ❖ 시스템 부팅에 필요한 변수 및 네트워크 인터페이스를 설정한다.

```
#define CONFIG_BOOT
#define CONFIG_BOOTDELAY      3
#define CONFIG_BOOTARGS      "root=ramfs devfs=mount console=ttyS0,115200"
#define CONFIG_ETHADDR        08:00:3e:26:0a:5b
#define CONFIG_NETMASK        255.255.255.0
#define CONFIG_IPADDR         192.168.1.11
#define CONFIG_SERVERIP       192.168.1.10
#define CONFIG_BOOTFILE        "ulmage"
#define CONFIG_BOOTCOMMAND    "tftp; bootm"
```

## 4. 명령 프롬프트를 보드 이름으로 수정

```
#define CFG_PROMPT            "DTK2410 #"
```

---

# 질의 응답