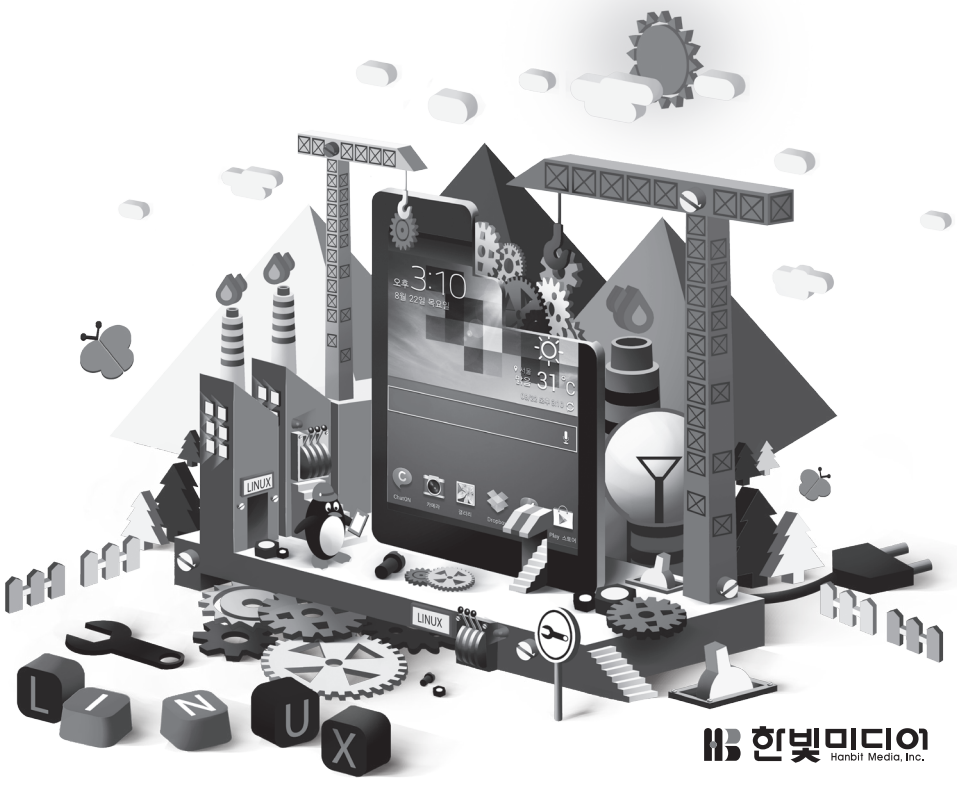
IT EXPERT IT 전문가의 현장실무서

개정판

ARM으로 배우는 임베디드 리눅스 시스템

CPU부터 안드로이드까지 총망라!

안효복 지음



한빛미디어 Hanbit Media, Inc.

연습문제 답안

1장. 임베디드 시스템 개요

1 임베디드 시스템은 일상 생활에서 사용되는 각종 전자 기기로 프로세서, 메모리 장치, 각종 입출력 장치 등의 하드웨어와 하드웨어를 제어하는 소프트웨어가 조합되어 특정한 기능을 수행하는 시스템이다.
임베디드 시스템의 응용 분야는 상당히 광범위하다. 스마트폰을 비롯하여 디지털 카메라, 내비게이션, 휴대용 게임기 등의 모든 휴대용 정보 단말기, 가정에서 사용하는 디지털TV, 냉장고, 세탁기, 오디오, 셋톱 박스, 가정용 게임기 등이 임베디드 시스템에 해당한다.

2 주어진 시간 내에 결과를 얻어야 하는 임베디드 시스템

3 하드웨어 구성 요소에는 프로세서, 메모리 장치, 입출력 장치가 있다. 소프트웨어 구성 요소에는 운영체제, 시스템 소프트웨어, 응용 소프트웨어가 있다.

4 레지스터는 데이터를 일시적으로 보관하는 임시 기억 장치다. 레지스터의 종류에는 범용 레지스터, 프로그램 카운터와 같은 제어용 레지스터 그리고 프로세서의 상태를 나타내는 상태 레지스터가 있다.

5 연산 장치는 덧셈과 뺄셈 같은 산술 연산, 논리 연산, 보수 연산, 시프트 연산 등을 수행한다.

6 제어 장치는 명령을 해석하고, 실행하고, 명령을 읽고 실행하는 데 필요한 프로세서 내부 각 장치 사이의 데이터 흐름을 제어한다.

7 16비트 프로세서는 한 번에 16비트 단위로 자료를 처리할 수 있는 프로세서로, 레지스터, 프로세서 내부 버스, ALU가 16비트 단위로 처리된다. 32비트 프로세서는 한 번에 32비트 단위로 자료를 처리할 수 있는 프로세서로 레지스터, 프로세서 내부 버스, ALU가 32비트 단위로 처리된다.

8 명령은 다음 3가지 단계를 통해 실행된다.

• 페치(FETCH) 단계

명령어 수행은 프로그램 카운터가 지정하는 주기억 장치에 위치한 명령어를 읽어서 명령어 레지스터에 저장한다.

• 디코드(DECODE) 단계

제어 장치가 명령 레지스터 내에 있는 명령어를 해독하여 적절한 제어 신호를 연산 장치, 메모리 등에 보낸다.

• 실행(EXECUTE) 단계

연산 명령의 경우 연산 장치에서 원하는 작업을 수행한 후 결과를 명령어에 명시한 레지스터에 저장한다. 조건 분기 명령어인 경우 분기 조건을 만족하면 PC의 내용을 분기 주소로 바꾼다. 무조건 분기 명령어인 경우 PC의 내용을 분기 주소로 바꾼다.

메모리 참조 명령어인 경우는 참조하려는 메모리의 주소를 데이터용 어드레스 레지스터로 보낸 후 메모리의 내용을 데이터 입력 레지스터로 읽어서 명령어에 지정된 레지스터에 저장한다.

9 CISC와 RISC의 차이는 다음과 같다.

• CISC(Complex Instruction Set Computer) 구조

① 명령어 수가 많다(보통 120~350개).

② 명령어 중에는 특별한 일을 하는 것도 있다.

③ 주소 지정 방식이 다양하다(8~24가지).

④ 명령어의 길이가 가변적이다.

⑤ 명령어 중에는 기억 장치 내의 피연산자를 처리하는 것도 있다.

• RISC(Reduced Instruction Set Computer) 구조

명령어 수와 주소 지정 방식을 최소화하여 제어 장치의 구조를 간단하게 구성한 프로세서 구조다.

① 명령어 수를 최소화하여 시스템을 작고 빠르게 한다.

② 명령어와 주소 지정 방식을 최소화하여 제어 장치 구조를 간단하게 구현한다.

③ 명령어의 길이가 고정되어 해독하기 쉽고 파이프라인 구성이 쉽다.

④ 로드/스토어 구조

10 SRAM(Static RAM)은 고속으로 동작할 수 있고, 전원이 공급되는 동안에는 안전하게 데이터를 저장한다. 하지만 한 비트를 구성하는 4~6개의 트랜지스터를 사용하기 때문에 반도체 생산 효율이 낮다.

DRAM(Dynamic RAM)은 SRAM에 비해 동작 속도가 느리고 전원이 공급되는 동안에도 일정 시간마다 리프레시를 해주어야만 안정되게 데이터를 저장한다. 하지만 1개의 트랜지스터만 이용하면 1비트를 저장할 수 있기 때문에 반도체 효율 면에서는 SRAM보다 유리하다.

SRAM과 DRAM의 차이점

구분	DRAM	SRAM
리프레시와 충전	주기적	필요 없다
액세스 주기	느리다	빠르다
회로 구조	단순하다	복잡하다
칩 크기	작다	크다
가격	싸다	비싸다
용도	일반 메모리	캐시 메모리

11 캐시 메모리는 프로세서에 근접하여 있는 고속의 메모리 장치로, 프로세서에서 읽기 요청이 있을 때 최대한 빨리 명령이나 데이터를 전달해주기 위한 고속 메모리 장치다.

12 메모리 제어기는 입력되는 주소 정보에 따라 어떤 메모리 장치를 액세스할 것인지를 결정(어드레스 디코드)하고 각 메모리 장치의 특징에 따라 제어 신호를 구동한다.

13 어드레스 버스는 칩 내부에서 메모리의 위치를 구분할 수 있는 영역을 선택하도록 하는 물리적인 신호를 말하며, 어드레스는 메모리 구별을 위해 사용하는 주소 그 자체를 말한다.

14 시스템 버스에는 많은 메모리 장치가 같이 연결될 수 있다. 하지만 한 사이클 동안에는 하나의 메모리 장치만을 사용할 수 있다. CE(Chip Enable) 또는 CS(Chip Select)는 물리적으로 연결되어 있지만 메모리 칩 내부에서 외부 버스와 연결을 끊어서 프로세서가 자신이 원하는 메모리 칩에만 연결되도록 하는 기능을 제공하는 칩 선택 신호다.

15 MMU는 프로세서에서 사용되는 가상 주소를 물리적으로 할당된 주소(Physical Address)로 변환하는 장치다. 어드레스 변환 기능과 메모리 보호 기능을 가진다.

16 입출력 장치의 주소 지정 방식은 다음 2가지가 있다.

- **I/O 맵(I/O–mapped peripheral) 방식**

  입출력 장치를 액세스하기 위해 별도의 신호와 in 또는 out과 같은 별도의 명령을 사용한다.

- **메모리 맵(Memory mapped peripheral) 방식**

  대부분의 임베디드 시스템에서 사용하는 방식으로 입출력 장치의 주소 공간을 별도로 할당하지 않고 메모리의 일부를 입출력 장치 공간으로 할당하여 사용한다. 입출력 장치의 접근은 메모리 참조 명령인 LDR/STR 명령을 사용한다.

17 입출력 장치의 자원 관리 방법은 다음 3가지가 있다.

- **폴링 방식**

  프로세서가 하나의 프로그램이나 장치들이 어떤 상태에 있는지를 지속적으로 검사하는 전송 제어 방식이다.

- **인터럽트**

  외부의 입출력 장치에서 프로세서에게 처리를 요청하면 프로세서는 현재 처리하는 프로그램을 일시적으로 멈추고 입출력 처리를 하는 방식이다.

- **DMA(Direct Memory Access)**

  프로세서의 개입 없이 입출력 장치와 기억 장치 사이에 데이터를 전송하는 방식이다.

18 인터럽트 발생 시 프로세서가 처리하기 위한 명령 또는 주소가 저장된 공간을 말한다.

19 인터럽트 제어기는 여러 개의 입출력 장치로부터 구동되는 인터럽트를 처리하는 장치다. 입출력 장치의 인터럽트 사용 여부를 제어하는 마스크(MASK) 레지

스터와 어떤 인터럽트가 발생되었는지를 저장하는 펜딩 레지스터를 가지고 있다. 인터럽트 제어기는 마스크 레지스터에 의해 사용하도록 설정된 인터럽트가 발생하면 프로세서로 인터럽트 요청 신호를 전달한다.

20 폰노이만 버스 구조와 하버드 버스 구조의 차이는 다음과 같다.

- **폰노이만 버스 구조**

명령어와 데이터를 위한 메모리 인터페이스가 하나만 존재하는 버스 구조로, 명령어를 읽을 때 데이터를 읽거나 쓸 수 없는 단점이 있다.

- **하버드 버스 구조**

명령어와 데이터를 위한 메모리 인터페이스가 분리되어 있는 버스 구조로, 명령어를 읽을 때에도 데이터를 읽거나 쓸 수 있고 데이터를 읽고 쓰는 동안에도 명령을 읽을 수 있는 장점을 가지고 있어 폰노이만 구조보다 성능이 우수하다.

2장. 임베디드 시스템 설계

1 용도와 기능과 성능을 분석한다. 분석된 결과에 따라 하드웨어와 소프트웨어의 사양을 결정하고, 결정된 사양에 따라 하드웨어 설계자는 프로세서, 메모리 및 주변 장치를 선정하고, 회로도를 그린 후 PCB를 설계한다. PCB 설계가 완료되면 PCB를 제조하고 사용되는 부품을 실장하여 하드웨어를 조립하고 시험한다. 소프트웨어 설계자는 설정된 사양에 따라 OS 사용 여부 및 OS와 개발 환경을 선정한다. 선정된 OS와 개발 환경에 따라 디바이스 드라이버 및 애플리케이션 소프트웨어를 설계하고 하드웨어에 탑재할 기능이 정상적으로 동작하는지 시험한다.

2 메모리는 크게 코드 및 데이터를 저장하기 위한 메모리와 프로그램 구동 중에 사용되는 메모리로 나뉜다. 코드 및 데이터 저장용 메모리(ROM)는 저장할 코드 특징 및 데이터의 크기 등에 따라 선정되며 EEPROM, NOR 플래시, NAND 플래시가 있다. 프로그램 구동 중에 사용되는 메모리(RAM)로 SDRAM을 많이 사용한다.

3 ROTS와 Non-RTOS의 차이는 다음과 같다.

- **RTOS**

주어진 입력 조건을 이용하여 주어진 한계 시간 내에 처리하여 결과를 내야만 하는 운영체제로, 멀티스레드와 선점성을 지원하며 일반적인 운영체제와 유사한 기능을 수행하지만 시간 제약성, 신뢰성 등을 일반 운영체제보다 중시한다.

- **Non-RTOS**

소프트(Soft) 리얼 타임 시스템에 사용되며 선점성 기능을 지원하지는 않는다. 하지만 멀티프로세싱 기능은 지원한다.

4 커널 포팅(이식)이란 운영체제의 커널이 타깃에서 동작하도록 운영체제의 프로세서 관련 처리, 메모리 관리, 트랩(Trap 또는 Exception) 및 인터럽트 처리, 타이머 등을 구현하여 운영체제의 스케줄러가 정상적으로 동작하도록 하는 과정을 말한다.

5 디바이스를 제어하는 프로그램, 사용자의 애플리케이션, 정형화된 인터페이스를 제공하는 함수와 자료 구조의 집합체를 말한다.

6 PC 기반 호스트 시스템에서 개발된 프로그램을 실제 탑재하려는 타깃에서 동작하도록 머신 코드를 생성하는 개발 환경을 말한다.

7 디버깅은 타깃 시스템을 실행하면서 프로그램의 실행 상태, 메모리, 변수 등을 프로그래머가 확인하거나 제어하면서 오류를 찾아 수정하는 동작이다.

- **와치 포인트(Watch point)**

프로세서의 실행을 제어하여 지정한 위치에 대한 데이터 액세스가 발생하면 프로세서의 정상 동작을 멈추고 디버그 상태로 전환된다.

- **브레이크 포인트(Break point)**

지정한 위치의 명령어가 읽혀지면 프로세서를 멈추고 디버그 상태로 전환된다.

3장. ARM 아키텍처와 동작 원리

1 ARM 아키텍처는 ARM 프로세서의 명령어, 레지스터 구조, 처리되는 데이터의 크기 등과 같은 기본 구성 및 동작 원리를 말한다. ARM 아키텍처에는 v4T, v5TE, v5TEJ, v6, v7 및 v8 등이 있다.

2 ARM 코어 또는 ARM 프로세서 코어는 ARM 아키텍처의 기본 원리를 이용하여 구현한 프로세서의 핵심이다.

3 프로그래머가 프로그램을 작성하는 데 필요한 명령어, 메모리 구조, 데이터 구조, 프로세서 동작 모드, 내부 레지스터 구성 및 사용법, 예외 처리, 인터럽트 처리와 같은 각종 정보를 말한다.

4 데이터 처리 명령은 ALU에 의해서 처리되는 명령으로 OP 코드, 대상 레지스터, 소스 레지스터, 오퍼랜드 2로 구성된다. 소스 레지스터는 A 버스로, 오포랜드 2는 B 버스로 전달되어 ALU에서 계산된 다음 ALU 버스를 통해서 전달된 결과가 대상 레지스터에 저장된다.

5 페치 단계에서 PC값이 지정하는 메모리 위치에 저장된 명령어를 읽어온다. 이어서 디코드 단계에서 읽어온 명령어를 해석하여 필요한 레지스터를 선택하면 실행 단계에서 연산 또는 주소를 계산한다. LDR/STR 명령과 같이 메모리 참조가 필요한 명령은 계산된 주소를 이용하여 MEMORY 단계에서 데이터를 읽어오고 WRITE 단계에서 연산 결과 또는 메모리에서 읽어온 데이터를 대상 레지스터에 저장한다. 프로세서에 따라 명령어 처리 단계가 줄거나 늘 수 있다.

6 ARM은 명령을 읽고, 해석하고, 처리하고, 메모리에서 데이터를 읽고, 그 결과를 저장하는 회로가 별도로 움직일 수 있도록 설계되어 있고, 각 동작은 한 사이클 내에 모두 처리가 가능하다. 명령을 실행하고 있을 때 다음 명령을 디코드하고, 그 다음 명령을 페치(FETCH)한다. 이와 같이 일련의 명령어 처리 동작이 연속적으로 이루어지는 것을 명령어 파이프라인이라 한다.

4장. 프로그래머 모델

1 일반적으로 32비트 ARM 명령과 16비트 Thumb 명령어가 있으며 근래 나오는 v6 이상 아키텍처 중에는 Thumb-2 명령을 지원하는 프로세서도 있다.

2 아키텍처 v6의 트러스트존을 지원하기 전에는 32비트 레지스터 37개를 가지고 있으며, 트러스트존을 지원하기 시작한 이후에는 40개를 가지고 있다.

3 어떤 권한을 가지고 어떤 종류의 작업을 처리하는지를 나타내는 것으로, 트러스트존 지원 이전 아키텍처에서는 USER 모드, SVC 모드, IRQ 모드, FIQ 모드, ABORT 모드, UNDEF 모드, SYSTEM 모드를 지원한다. 트러스트존을 지원하면서 MONITOR 모드가 추가되어 총 8가지의 동작 모드를 지원한다.

4 외부의 요청이나 오류에 의해 정상적으로 진행되는 프로그램의 동작을 잠시 멈추고 프로세서의 동작 모드를 변환한 후 미리 정해진 프로그램을 이용하여 외부의 요청이나 오류에 대한 처리를 하는 동작을 말한다.

5 리셋, 소프트웨어 인터럽트, UNDEF 명령, 프리페치 어보트, 데이터 어보트, IRQ, FIQ 예외 처리가 있다.

6 읽어온 명령을 해석하고 프로그램을 실행할 준비를 하는 단계다.

7 ARM 명령에 비해 코드의 크기를 줄일 수 있고, 좁은 폭의 데이터 버스를 사용하는 경우 성능을 향상시킬 수 있기 때문에 사용된다.

8 R0에서 R15까지 총 16개의 레지스터가 사용될 수 있다.

9 SP, 즉 스택포인터로 사용되는 레지스터는 R13이고, 링크 레지스터 LR로 사용되는 레지스터는 R14다.

10 링크 레지스터는 BL 명령이 사용되는 경우 또는 예외 처리가 발생하는 경우 되돌아올 주소를 저장하는 레지스터다.

11 빅 엔디안은 메모리의 하위 어드레스(Byte 어드레스 0)에 MSB(Most Significant Byte)가 위치하는 메모리 구조이며, 리틀 엔디안은 메모리의 하위 어드레스(Byte 어드레스 0)에 LSB(Least Significant Byte)가 위치하는 메모리 구조다.

12 FIQ는 별도로 할당된 범용 레지스터를 가지고 있어 스택 동작이 IRQ보다 적고 우선순위가 높으며, 예외 처리 벡터 테이블의 맨 상단에 FIQ 벡터가 위치하고 있어 분기 명령 사용 없이 예외 처리 핸들러를 작성할 수 있기 때문이다.

5장. ARM 프로세서 명령어

1 모든 명령이 32비트로 구성되어 있어 Load/Store와 같은 메모리 참조 명령이나 분기(Branch) 명령에서는 모두 상대 주소(Indirect Address) 방식을 사용하고 있다. 이미디어트(Immediate) 상수값도 사용은 가능하지만 32비트 명령어 내에 표시해야 하므로 32비트 상수가 사용될 수 없다. 또한 ARM 명령은 모두 조건부 실행이 가능하다.

2 불필요한 분기 명령의 사용을 줄여서 파이프라인이 깨지는 것을 방지함으로써 성능을 향상시킬 수 있다.

3 PC값을 기준으로 +/– 32MB 이내다.

4 데이터 처리 명령의 두 번째 오퍼랜드는 B 버스를 통해서 전달되는데 레지스터 뱅크 또는 명령어에 포함된 이미디어트 상수를 사용할 수 있고, ALU에 입력되기 전에 배럴 시프터를 통과한다. 따라서 레지스터를 두 번째 오퍼랜드로 사용하는 경우에는 인라인 배럴 시프트 동작을 같이 사용할 수도 있다.

5 LSL, LSR, ASR, ROR, RRX가 있다.

6 8비트로 표현 가능한 숫자로, 짝수만큼 로테이트 라이트(ROR)해서 나올 수 있는 상수여야 한다.

7
```
mrs r0, cpsr @ CPSR 레지스터값을 읽어 R0에 저장
orr r0, r0, #0x80 @ r값의 I비트를 1로 세트
msr cpsr_c, r0
@ r값을 CPSR의 control 필드에 저장
```

8 Pre-index는 메모리에 접근하기 전에 베이스 레지스터와 오프셋을 먼저 계산하는 방식이고 Post-index는 메모리 접근 후에 베이스 레지스터와 오프셋을 계산하여 베이스 레지스터값을 변경하는 방식이다.

9 MOV 명령이나 LDR 명령을 사용하여 대상 레지스터를 PC로 분기한다.

```
사용 예 : mov   pc,  0x3000000
          ldr   pc,  =0x33001200
```

10 블록 단위의 전송 명령에서 사용되는 접미사는 IA, IB, DA, DB 4가지가 있다.

- **IA(Increment After)**

  먼저 베이스 레지스터가 지정하는 주소의 메모리에 데이터를 저장하거나 읽고 어드레스를 자동 증가한다.

- **IB(Increment Before)**

  베이스 레지스터의 위치에서 1워드 자동 증가한 후 데이터를 저장하거나 읽는다.

- **DA(Decrement After)**

  데이터를 읽거나 저장하고 어드레스를 감소시킨다. 하지만 자동 디크리먼트가 없기 때문에 먼저 ALU를 이용하여 어드레스를 계산한 후 어드레스를 증가하면서 데이터를 읽거나 쓴다.

- **DB(Decrement Before)**

  먼저 어드레스를 감소하고 데이터를 읽거나 저장하는 방식이다. ALU를 이용하여 어드레스를 계산한 후 어드레스를 증가하면서 데이터를 읽거나 쓴다.

11 베이스 포인터(BP)는 스택의 맨 처음(bottom) 위치를 나타내고 스택 포인터(SP)는 스택의 최종 위치를 나타낸다.

12 FD(Full Decending) 스택이 사용된다.

13 ARM/Thumb 인터워킹(Interworking)은 ARM과 Thumb의 상태 변환을 의미한다.

14 아키텍처 v4T까지 지원되는 명령은 BX이고 v5TE에 새롭게 추가된 명령은 BLX이다. BLX 명령은 BX 명령이 삽입되는 링커 비니어를 제거하므로 연속적인 브랜치 명령 사용에 의한 파이프라인이 깨지는 현상을 제거하여 시스템의 성능을 향상시킨다.

15 ARM의 Pseudo 명령은 실제로 명령어 디코더에 정의된 명령이 아니고 어셈블러에서 지원하는 명령이다. ADR, ADRL과 같은 실제 ARM 명령어들로 만들어진다.

16 SIMD 명령은 병렬 처리 명령으로, 하나의 명령으로 여러 개의 값을 동시에 계산할 수 있는 명령어들의 집합이다. 이 명령어는 벡터 프로세서에서 주로 사용되며, 비디오 게임 및 그래픽 처리 등 멀티미디어 분야에서 많이 사용한다.

17 ARM 프로세서 중 트러스트존을 지원하는 프로세서는 보호(Security) 영역과 비보호(Non-Security) 영역의 독립적인 공간으로 구분되어 데이터의 보호 및 프로그램을 관리할 수 있다. 모니터 모드는 보호 영역과 비보호 영역 사이의 모드 변경을 처리하기 위한 코드가 처리되는 동작 모드다.

18 Thumb-2 명령어는 ARM 아키텍처 v6 이후에 새롭게 지원되는 16비트 명령어다. 하지만 기존 Thumb 명령어와 달리 자유롭게 Thumb 명령어와 혼용하여 사용할 수 있는 32비트 명령어도 포함한다. ThumbEE 명령어는 Cortex에 새롭게 도입된 명령어로, Thumb-2 명령어를 개선하여 더 최적화된 코드를 생성할 수 있고 다양한 기능을 제공한다.

19 NEON 명령은 Advanced SIMD(Single Instruction Multiple Data) 명령과 VFPv3 명령을 포함하고 있으며, ARM 코어의 레지스터와는 별도로 Advanced SIMD 및 VFPv3 전용의 64비트 및 128비트 레지스터들을 가지고 있다.

20 VSTR은 32비트 또는 64비트 벡터 데이터를 메모리에 저장하고 VSTM은 32비트 또는 64비트 크기의 여러 벡터 데이터를 메모리에 저장한다.

21 VCVT(Vector Convert) 명령을 사용한다.

22 VADD(Vector Add) 명령은 벡터 단위의 더하기 및 빼기 연산을 처리한다. VPADD(Vector Pairwise Add) 명령은 벡터 요소의 인접 쌍을 더하고 결과를 대상 레지스터에 저장한다.

6장. 예외 처리와 시스템 리셋

1 일반적으로 B 명령이 사용된다. 하지만 B 명령은 PC값을 기준으로 32MB를 넘으면 사용하지 못한다. 이 경우에는 MOV 또는 LDR 명령과 함께 PC값을 대상 레지스터로 사용한다.

2 PC값을 LR_〈mode〉로부터 복원한다. 그리고 SPSR_〈mode〉를 CPSR에 복원한다. 이때 두 동작은 하나의 명령에 의해서 동시에 하나의 명령으로 이루어져야 한다.

3 예외 처리에서 복귀할 때는 다음과 같은 명령을 사용한다.

- 데이터 처리 명령에 S 접미사를 사용하고 PC를 대상 레지스터로 지정한다.
- LDM 명령을 사용하면서 레지스터 리스트에 PC가 있고, 레지스터 리스트 뒤에 '^'을 사용한다.
- 아키텍처 v6 이후에는 RFE 명령을 사용한다.

4 FIQ와 IRQ는 모두 외부의 입출력 장치에서 입출력 동작의 처리 요청에 의해서 발생한다.

5 Data Abort가 발생한다.

6 SWI 예외 처리는 명령어를 해석하는 단계, 즉 디코드 단계에서 발생한다.

7 SMC 명령에 의해서 발생하며 Secure 모드와 Non-secure 모드 변환을 위해서 사용된다.

8 SVC 모드의 ARM 상태가 된다.

7장. ARM 프로세서 코어

1 애플리케이션 프로세서는 리눅스, 안드로이드처럼 복잡한 운영체제를 탑재하기에 적합한 프로세서다. MMU를 가지고 있다.

2 임베디드 프로세서는 소형의 저가형 임베디드 시스템에 적합한 프로세서 제품군으로, 주로 RTOS를 사용하거나 운영체제를 사용하지 않고 펌웨어만으로 제품을 구현하여 사용하며, MPU(Memory Protection Unit)와 같은 단순한 메모리 제어 모듈만을 포함한다.

3 ARM9TDMI 프로세서의 이름에서 T, D, M, I는 다음을 의미한다.

- T : Thumb 명령 지원
- D : 디버그 지원
- M : 64비트 결과를 낼 수 있는 32x8 곱셈기 지원
- I : EmbeddedICE 로직

4 A 버스, B 버스, ALU 버스, 데이터 입력, 데이터 출력 버스

5 프로세서 코어와 연결되는 모든 제어 신호, 어드레스 버스, 데이터 버스를 감시하다가 호스트 디버거에서 설정한 디버그 조건이 성립되면 프로세서를 멈추고 디버깅할 수 있다. 프로그래머는 EmbeddedICE 로직을 사용하여 브레이크 포인트와 와치 포인트를 설정한다.

6 ARM9TDMI 코어의 파이프라인은 5단계로 구성된다.

① 페치 단계 : 명령어를 읽는다.

② 디코드 단계 : 명령어를 해석하고 필요한 레지스터의 값을 읽는다.

③ 실행 단계 : 연산을 수행한다. 데이터 전송 명령의 경우 주소를 계산한다.

④ 메모리 단계 : 메모리를 액세스한다.

⑤ 쓰기 단계 : 결과값을 대상 레지스터에 저장한다.

7 LDR 명령에서 대상 레지스터로 사용된 레지스터가 메모리에서 값을 읽어오기도 전에 다음 명령의 연산에서 사용되면 파이프라인을 진행할 수 없고 파이프라인을 지연해야 한다. 이 현상을 인터락이라 한다.

8 ARM920T는 MMU를 가지고 있고, ARM940T는 MPU를 가지고 있다. 캐시는 각각 16KB와 4KB를 사용한다.

9 분기 예측은 분기 실행의 여부를 미리 예측하는 방법이다. 조건에 따라 분기 여부를 결정하는 경우 분기 명령에 필요한 조건 코드의 결과가 분기 명령의 디코드 후에 3 또는 4사이클이 지날 때까지 나오지 않을 경우가 있는데, 분기 예측을 사용하면 분기의 향방을 미리 예측하여 지연을 방지해 성능을 향상시킬 수 있다.

10 SCU(Snoop Control Unit)는 ARM 멀티코어 기술의 핵심 기능으로 코어 간의 상호 연결, 중재, 커뮤니케이션, 캐시-2-캐시 전송, 시스템 메모리 액세스 제어, 캐시 동기화 등 모든 멀티코어 기술이 적용된 프로세서의 기능을 관리한다.

11 파이프라인을 구성할 때 한 번에 하나 이상의 명령어를 처리할 수 있는 파이프라인 구조를 말한다.

12 리얼타임 프로세서에는 Cortex-R4, CortexR4F 등이 있다.

13 Cortex-M0 임베디드 프로세서는 ARMv6-M 아키텍처를 사용하며 Thumb와 Thumb-2 명령을 지원한다.

14 Cortex-M3 프로세서는 ARM 아키텍처 v7-M을 사용하고 32비트 Thumb-2 명령을 포함하는 Thumb 명령을 지원하여 코드 용량을 줄이면서도 높은 성능을 낼 수 있다.

8장. ARM 프로세서

1 코프로세서 15번에 연결되어 있기 때문에 코프로세서와 ARM의 레지스터 사이의 데이터 전송 명령인 MCR, MRC를 사용하여 제어한다.

2 캐시 메모리는 프로세서에서 요구하는 명령이나 데이터를 최대한 빨리 전달해줌으로써 시스템의 속도를 향상시킬 수 있다.

3 매우 빠른 프로세서와 상대적으로 느린 속도의 버스 또는 메모리 장치의 속도 차이를 보상하기 위해서 프로세서가 결과 데이터 저장을 기다리지 않아도 되게 한다.

4 캐시에 있는 데이터를 쓸 때 주메모리에 동시에 기록되는 캐시를 라이트 스루 캐시라 하고, 데이터를 쓸 때 캐시에만 기록하고 주메모리에는 나중에 기록하는 방식을 라이트 백 캐시라 한다.

5 캐시의 락다운은 캐시에 저장된 일정한 명령이나 데이터를 갱신하지 않고 항상 캐시에 존재하도록 하여 일정한 성능을 보장한다.

6 L2 캐시는 L1 캐시와 외부 메모리 사이에 존재하는 캐시 메모리다. 버스 및 외부 메모리 장치로의 액세스 빈도를 줄여 전력 소모를 감소시키고, 시스템 성능을 증가시키기 위해 사용한다.

7 가상 주소를 물리 주소로 변환하는 기능과 메모리 보호 기능을 가진다.

8 ARM 프로세서의 MMU 구성 요소는 다음과 같다.

- TLB(Translation Lookaside Buffer)
- 접근 제어(Access control) 로직
- 변환 테이블 관리(Translation-table-walking) 로직

9 프로그래머가 프로그램에 의해서 4GB의 가상 주소를 1M 섹션 단위 주소 변환 정보, 캐시 및 쓰기 버퍼 사용 정보, 접근 권한 정보를 작성한다.

10 섹션, 슈퍼 섹션 및 페이지 디스크립터가 있다.

11 페이지 단위로 메모리를 관리하려는 경우 사용된다.

12 스몰 페이지 및 라지 페이지 디스크립터가 있다.

13 1차 및 2차 변환 테이블의 C 및 B 비트에 의해서 설정된다.

14 1차 및 2차 변환 테이블의 AP 비트에 의해서 제어된다.

15 주소 변환 정보, 캐시 및 쓰기 버퍼 사용 정보, 접근 권한 정보를 포함하는 변환 테이블을 만들고, 변환 테이블의 주소를 TTB 레지스터에 기록한 다음 제어용 레지스터를 사용하여 MMU를 인에이블하고 캐시와 쓰기 버퍼를 인에이블하는 순서로 설정된다.

16 MPU의 주요 기능은 메모리를 일정한 크기의 보호 영역(protection region)으로 나누고, 각 영역별로 접근 권한(Access Permission)과 캐시 사용 여부, 쓰기 버퍼 사용 여부를 지정하여 관리하기 위해 사용된다.

17 TCM은 캐시의 계속적인 내용 변경에 따르는 성능의 저하를 해결하기 위해 CPU 주변에 배치된 고속 메모리다. TCM이 사용되면 빠르게 처리하려는 명령이나 데이터를 TCM 메모리에 복사해서 사용함으로써 불필요한 버스와 메모리의 액세스를 줄이고 고속으로 명령이나 데이터를 처리한다.

18 명령어 수준에서 병렬 처리를 하는 기법에는 슈퍼스칼라 파이프라인이 있다.

19 스레드 수준에서 병렬 처리를 하는 기법에는 하이퍼스레딩(hyper threading), 동시 멀티스레딩(SMT : Simultaneous Multi Threading)이 있다.

20 UMA 구조와 NUMA 구조의 차이는 다음과 같다.

• UMA 모델

모든 프로세서들이 상호 연결망에 의해 접속된 주기억장치를 공유하는 방식이다. UMA 모델을 사용하여 구성한 대표적인 멀티코어 시스템으로 SMP(Symmetric

Multi–Processor)가 있다.

• **NUMA 모델**

UMA 모델의 한계를 극복하고 더 큰 규모의 시스템을 구성하기 위한 모델로 다수의 UMA 모델들을 상호 연결망에 접속하여 사용하는 방식이다.

21 SMP 멀티코어는 프로세서마다 캐시를 가질 수 있고 필요에 따라 여러 프로세서가 공유하는 별도의 캐시를 하나 더 가질 수도 있다.

22 1 / ((1 – P) + P / N) = 1 / ((1 – 0.6) + 0.6 / 4) = 1.82배 정도 빨라진다.

9장. SoC 구조

1 ARM에서 설계한 시스템 버스 구조로, ARM 프로세서 기반의 SoC를 설계할 때 프로세서를 비롯한 각 장치를 연결하기 위한 방법을 정의하는 규격이다.

2 AXI 버스는 버스트 동작을 기본으로 하고, 버스 사이클은 어드레스 구동 단계(Address Phase), 제어 신호 구동 단계(Control Phase), 데이터 구동 단계(Data Phase)로 분리된다. 정렬되지 않은 데이터(Unaligned data) 전송을 지원한다.

3 AHB 버스의 특징은 다음과 같다.

- 고속(High performance)으로 동작
- 파이프라인 동작 지원
- 여러 개의 버스 마스터 지원

4 APB 버스는 비교적 속도가 느린 주변 장치를 제어하고 전력 소모를 줄일 수 있도록 구성하기 위해 간단한 인터페이스 구조를 가진다. 래치(latch) 타입의 어드레스와 제어 신호를 가지고 있어 전력 소모를 줄일 수 있도록 설계된다.

5 읽기 또는 쓰기 동작을 요청하는 주체가 되며 한 번에 하나의 마스터만이 버스를 사용할 수 있다. CPU, DMA 장치 등이 마스터가 된다.

6 여러 버스 마스터의 요청을 중재하여 한 번에 하나의 마스터만이 버스를 사용하도록 조정하는 기능을 수행하는 제어 블록이다.

7 GPU는 CPU와 함께 사용하여 그래픽의 처리 속도를 향상시키는 장치로, CPU는 일반적으로 직렬로 데이터를 처리하는 반면 GPU는 병렬로 데이터를 처리하여 고속의 데이터 처리를 한다.

8 HKMG는 누수 전류를 줄이고 반도체의 성능을 향상시킨다.

10장. 임베디드 시스템 하드웨어 설계

1 설계 사양에 적합한 프로세서, 메모리 입출력 장치를 선정하고 선정된 소자를 이용하여 회로도를 설계한 다음 PCB를 설계한다. PCB 설계 후에는 PCB를 생산하고 생산된 PCB에 부품을 실장하고 시험한다.

2 애플 iOS, 구글 안드로이드, MS 윈도우 모바일 등이 있다.

3 NOR 플래시와 NAND 플래시가 있다.

4 LPDDR, LPDDR2 등 고속이면서 전력 소모는 적은 모바일 DDR SDRAM이 사용된다.

5 NAND 플래시는 Raw NAND, OneNAND, LBA NAND, BA NAND, eMMC, eSD, Flex OneNAND, ONFI 등이 있다.

6 NAND는 부가적인 회로 없이 블록 단위의 읽기, 지우기, 쓰기 방식을 제공하는 메모리다. One NAND의 내부는 NAND지만 외부와의 통신은 NOR 플래시와 유사하게 어드레스와 데이터 버스를 이용하여 접근이 가능한 메모리다.

7 eMMC는 JEDEC에서 표준으로 정한 휴대용 카드 인터페이스 규약이다. 하지만 휴대용 카드가 아닌 보드에 내장하여 사용하기 때문에 e(embedded)를 MMC 앞에 붙여서 사용한다.

8 디지털 RGB와 LVDS 인터페이스의 차이는 다음과 같다.

- **디지털 RGB 인터페이스**

3원색을 TTL 레벨로 LCD 표시 장치에 전송하는 방식으로 LCD와의 거리가 짧을 때 주로 사용한다.

- **LVDS(Low voltage differential signaling) 인터페이스**

인터페이스는 고속의 디지털 인터페이스로, 고속 데이터 전송 속도를 위한 낮은 전력 소비 및 뛰어난 노이즈 내성을 가지는 인터페이스다.

9 MIPI(Mobile Industry Processor Interface)는 휴대 단말기에서 다양한 인터페이스를 표준화하기 위해서 구성된 공개 협회다.

10 용량이 크고 저온에서 자연 방전 가능성이 있으나 디자인 변형이 자유롭다.

11 가속도 센서는 이동하는 물체의 가속도나 충격의 세기를 측정하는 센서로, 물체의 운동 상태를 상세하게 감지하여 진동, 충격 및 가속도를 측정하기 위해서 사용된다.

12 회로도 설계는 논리적으로 선정된 부품을 연결하는 것이고, PCB 설계는 논리적으로 설계한 회로를 물리적으로 부품을 실장하도록 배치하고 신호를 연결하는 작업이다.

11장. 소프트웨어 개발 툴의 이해와 활용

1 ELF 이미지이며 ELF 헤더를 포함하여 텍스트, 데이터 등 섹션 정보와 심볼 테이블을 포함한다.

2 머신 코드와 데이터를 포함한다.

3 ATPCS 규정에 따라 기본적으로 레지스터에 할당된다. 하지만 지역 변수의 사용이 많아지면 스택이 사용된다.

4 ATPCS 규정에 따라 기본적으로 4개까지의 인수는 레지스터에 할당된다. 하지만 인수 수가 많아지면 스택이 사용된다.

5 링커는 여러 개의 불완전한 오브젝트 파일을 합쳐 모든 코드와 데이터를 포함하는 하나의 새로운 독립형 오브젝트 파일을 생성해내는 도구다. 로케이터(Locator)는 링커에 포함되어 새로운 오브젝트 파일을 생성해낼 때, 메모리에서 실행이 가능하도록 코드와 데이터를 배치하는 도구다.

6 프로세서를 비롯하여 버스 마스터에서 구동한 어드레스에 따라 어떤 메모리 장치를 액세스할 것인지를 결정하고 각 메모리 장치의 특징에 따라 제어 신호를 구동하는 제어 장치다.

7 리매핑은 MMU가 지원되지 않는 SoC의 물리적인 메모리 맵을 바꾸는 방법이다. 메모리 제어기의 어드레스 디코딩 방법을 변경한다. 리매핑은 시스템의 성능을 향상시키기 위해 리셋 동안에는 0x0번지에 ROM이나 플래시를 사용하지만, 초기화가 완료된 후 프로그램에 의해 DRAM을 0x0번지로 사용하도록 메모리 구조를 변경한다. DRAM은 ROM이나 플래시에 비해 비교적 동작 속도가 빠르고 대부분 워드 크기의 데이터 폭을 가지고 있어 여러모로 유리하다.

8 프로그램이 동작되는 메모리를 분산해서 사용하는 방법이다.

9 인터페이스는 nTRST, TCK, TMS, TDI, TDO 5개의 신호로 구성되며, 내부는 TAP 제어기, 레지스터, 바운더리 스캔 셀로 구성된다.

10 개발이 완료되어 ROM에 탑재하는 경우에는 전용 ROM 라이터(Writer)를 이용하여 프로그램을 기록하고 타깃에 삽입하여 사용한다. 플래시 메모리는 전용 ROM 라이터, ICE 장비 또는 간단한 JTAG 동글(Dongle)을 이용하여 탑재가 가능하다.
DRAM에 실행 가능한 바이너리 이미지를 탑재하여 실행하는 경우에는 타깃 시스템의 ROM이나 플래시

메모리에 모니터 프로그램 또는 부트로더를 탑재하고 시리얼 통신, USB 통신 또는 이더넷 통신을 이용하여 탑재하거나 부트로더 없이 디버그 ICE를 이용하거나 JTAG 동글을 이용하여 탑재하는 방법도 있다.

12장. 임베디드 C 프로그래밍과 소프트웨어 최적화

1 컴파일러의 최적화 옵션은 다음과 같다.

- O0 : 최적화를 하지 않는다.
- O1 : 최적화를 제공하지만, 일부 최적화 기능은 디버깅에 사용하지 않는다.
- O2 : 좋은 최적화 기능을 수행한다.
- O3 : –O2와 동일한 최적화를 수행하지만 추가적인 옵션 사용에 따른 제어를 할 수 있다.

2 임베디드 C 프로그램에서 사용되는 변수는 다음과 같다.

- 지역 변수 : 레지스터 또는 스택
- 전역 변수, 스태틱 변수 : 메모리의 RW 또는 BSS 영역

3 구조체를 사용한다.

4 컴파일러가 최적화 과정에서 코드를 제거하지 않도록 하는 지시어로, 프로그래머가 지정한 읽기 또는 쓰기 동작을 반드시 실행하도록 사용한다. 특히 입출력 장치를 제어하는 경우에는 매우 중요하다.

5 스택의 용도는 다음과 같다.

- 함수의 시작과 끝에서 범용 레지스터와 링크 레지스터 저장
- 예외 처리의 시작과 끝에서 범용 레지스터와 링크 레지스터 저장
- 지역 변수 사용이 많아질 때
- 함수의 인수가 4개 이상 사용될 때

6 ARM 상태와 Thumb 상태의 상태 변환 과정을 말한다. v4T까지는 BX 명령만 사용되었으며, v5TE에서는 BLX라는 새로운 명령이 추가되어 사용된다.

7 NEON 명령어를 사용하는 방법은 다음과 같다.

- 어셈블러로 작성
- 인스트린식 사용
- 자동 벡터화(Audo Vectorization)
- NEON 최적화 라이브러리 사용

13장. 시스템 리셋과 부트코드

1 ARM 프로세서에 리셋 신호가 입력되면 프로세서는 다음과 같은 동작을 한다.

- CPSR의 모드 비트 M[4:0]를 슈퍼바이저 모드인 10011'b로 변경한다.
- CPSR의 I 비트와 F 비트를 1로 세트하여 인터럽트를 지원하지 않도록 한다.
- CPSR의 T 비트를 0으로 클리어하여 ARM 상태로 변경한다.
- PC값을 리셋 벡터 주소인 0x00000000으로 변경한다.

2 불필요한 하드웨어를 디스에이블하고 시스템 클록과 메모리 제어기 등을 설정하여 하드웨어가 동작하도록 설정한다. 이어서 스택 영역 및 변수 영역 등을 설정하여 소프트웨어가 동작하는 데 필요한 정보를 설정하고 C로 만들어진 함수로 분기한다.

3 PLL은 위상 고정 루프(Phase Locked Loop)의 약어이다. PLL은 고주파 회로의 오실레이터(주파수 발생 장치)에서 위상을 맞추어 정확한 주파수를 만들어내기 위해 사용하는 방법이다.

15장. 개발 환경과 부트로더

1 개인용 컴퓨터의 BIOS라고 생각하면 된다. 따라서 시스템 초기화 기능, 타깃 시스템 동작 환경 설정 기능, 시스템 운영체제 부팅, 플래시 메모리 관리 기능, 모니터 기능 등이 필요하다.

2 부트로더는 하드웨어 의존성이 강하고, 대부분 C가 아닌 어셈블리어로 작성되는 경우가 많다. 따라서 부트로더를 작성하기 위해서 프로그래머는 프로세서의 구조, 특징, 사용법을 알고 있어야 하며, 특히 부트로더의 시작 부분은 어셈블리어로 작성되기 때문에 명령어 사용법을 알아야 부트로더를 작성할 수 있다.

3 U-Boot는 공개된 부트로더다. U-Boot는 PPCBoot와 ARMBoot 프로젝트를 기반으로 개발되었으며 다양한 PPC 또는 ARM 프로세서 기반의 타깃 보드를 지원한다. U-Boot는 TFTP를 이용하여 운영체제를 부팅할 수 있을 뿐만 아니라 플래시 메모리, IDE, SCSI 등 다양한 매체를 이용하여 부팅 가능하고 JFFS2 파일시스템을 비롯한 다양한 파일시스템을 관리하는 기능도 있다.

4 U-boot는 설정, 의존성 검사 후에 컴파일하는 단계로 빌드된다. 다음의 U-Boot 빌드 과정의 예다.

```
--------명령어 실행 예 시작 --------------
# make dtk4412_config
# make dep
# make
-------- 명령어 실행 예 끝 --------------
```

5 부트로더를 대상 시스템에 탑재하는 방법은 다음과 같다.

- 롬 라이터를 사용하여 프로그래밍 후 보드에 삽입하는 방법
- 전용의 디버깅 툴과 ICE 장비를 사용하여 플래시에 탑재하는 방법
- JTAG 동글을 활용하여 플래시에 탑재하는 방법
- 기존에 탑재된 부트로더를 이용하여 탑재하는 방법

16장. 리눅스 커널

1 운영체제는 컴퓨터 시스템의 전반적인 동작을 제어하고 조정하는 시스템 프로그램들의 집합으로, 하드웨어와 애플리케이션 간의 인터페이스 역할을 하면서 프로세서, 주기억 장치, 입출력 장치 등의 전반적인 컴퓨터 시스템의 자원을 관리한다.

2 마이크로 커널은 QNX나 마크(MACH) 운영체제와 같은 커널의 핵심 기능만 가지고 있는 방식으로, 커널 기능을 최소화하여 매우 간단한 스케줄러와 프로세스 간 통신 방식만을 제공한다. 모놀리틱 커널은 마이크로 커널과 달리 커널이 운영체제의 대부분의 서비스를 가지는 방식으로, 프로그램의 모든 실행을 제어하며 데이터와 파일 관리까지 담당한다. 유닉스, 리눅스 시스템은 모놀리틱 커널의 대표적인 예다.

3 리눅스 커널은 프로세스 스케줄러, 메모리 관리 기능, 프로세스 간 통신 기능, 가상 파일시스템, 네트워크 인터페이스로 구성된다.
프로세스 스케줄러는 CPU를 여러 프로세스가 공평하게 사용하도록 하고, 여러 개의 프로세스가 주메모리를 안전하게 공유해서 사용하도록 메모리 관리 기능을 제공한다. 여러 프로세스 간 통신을 할 수 있는 기능도 제공한다.
가상 파일시스템은 여러 종류의 파일시스템을 동일한 방식으로 접근하도록 인터페이스를 제공하고 있으며, 다양한 네트워크 프로토콜과 디바이스 드라이버를 제공하고, 오랜 기간 동안 서버나 여러 장비에서 검증된 강력하고 안정된 네트워크 기능을 제공한다.

4 스케줄러는 CPU를 여러 프로세스가 공평하게 사용하도록 주기적인 타이머 인터럽트에 의해서 하나의 프로세스에서 다른 프로세스로 전환하는 기능을 수행하여 멀티태스킹이 가능하게 한다.

5 프로세스는 태스크라고도 하며 실행 중인 애플리케이션을 의미한다. 일반적으로 프로세스는 프로그램, 데이터, 스택 정보, 그리고 프로세서 내부의 프로그램 카

운터를 비롯한 다양한 레지스터 정보를 포함한 개체다.

6 프로세스는 clone( ), fork( ) 또는 vfork( ) 시스템 콜을 이용하여 프로세스를 생성한다.

7 디맨드 페이징은 프로세스가 시작하면서 일부 페이지 프레임만 할당하여 사용하다가 새로운 프레임의 요구가 있을 때 새로운 페이지 프레임을 할당하여 사용하는 방식이다.

8 리눅스에서 타이머는 10ms 시간 주기로 인터럽트를 발생하며, 다음과 같은 용도로 사용된다.

- 시스템의 시작부터 경과한 시간을 갱신
- 현재 날짜와 시간 갱신
- 비동기 스케줄링
- 소프트웨어 타이머로 사용되어 일정한 시간 간격을 측정하고 딜레이 동작 처리

9 시스템 콜은 사용자 프로그램에서 커널 자원을 사용하도록 지원하는 방식으로 디바이스의 제어, 시스템 프로그램 실행, 파일 전송 등은 시스템 콜을 이용하여 구현된다.

10 가상 파일시스템은 모든 파일시스템에 대한 공통적인 인터페이스를 사용하도록 하는 표준 인터페이스다. 리눅스에서 VFS로 지원하는 파일시스템은 디스크 기반 파일시스템, 네트워크 파일시스템 및 기타 디바이스 파일과 같은 특수 파일시스템이 있다.

11 exec( ) 시스템 콜이 사용된다.

12 대표적으로 시그널과 세마포어, 메시지 큐 그리고 공유 메모리 방식이 있다.

13 링커 스크립트 파일(arch/arm/kernel/vmlinux.lds)에 기록된다.

17장. 디바이스 드라이버

1 네트워크 디바이스는 이더넷과 같은 통신용 제어 장치로, 네트워크로 연결된 다른 호스트와 데이터를 교환하는 장치를 말하며, 네트워크 디바이스 드라이버는 네트워크 디바이스를 제어하는 프로그램과 애플리케이션과의 통신을 위한 자료 구조를 말한다.

2 디바이스를 파일과 같이 관리하기 위해서 디바이스의 종류를 구분하고 접근 권한을 지정하며, 디바이스에 데이터를 읽고 쓰기 위해서 사용되는 파일이다.

3 일반적으로 인터럽트 방식과 DMA 방식을 많이 사용한다. 인터럽트를 사용하는 경우에는 request_irq( ) 함수를 사용하여 등록한 후 사용할 수 있으며 필요에 따라서 소프트웨어 인터럽트를 사용할 수도 있다.

4 일반적으로 sysfs는 루트 파일시스템의 /sys 디렉터리에 마운트되어 사용되며, 이 디렉터리의 내용을 보면 시스템에서 구동되는 모든 장치에 대한 정보를 알 수 있다. 장치에 대한 정보는 클래스에 따라, 디바이스 정보에 따라 또는 버스에 따라서 다양한 방법으로 관찰이 가능하다.

5 Udev를 사용하면 사용자 영역에서 필요에 따라 원하는 이름으로 디바이스 파일의 이름을 변경하면서 노드를 생성할 수 있다.

6 인터럽트 등록 함수는 request_irq( ), 등록 해제 함수는 free_irq( ) 함수다.

7 구성된 I/O 메모리 맵 테이블을 구성하여 아키텍처를 초기화하는 동안에 iotable_init( ) 함수를 호출하여 MMU의 페이지 테이블을 설정하거나 커널에서 동적으로 가상 주소를 할당받은 경우에는 ioremap( ) 함수를 사용한다.

8 리눅스의 문자 디바이스 드라이버는 VFS 구조에 따라 제어하기 때문에 문자 디바이스 드라이버에서는 파일시스템에 지정된 동작을 구현해야 한다. 파일 동작은 〈linux/fs.h〉의 file_operations 구조체에 선언되며 일반 파일을 사용하는 것과 동일하게 open, close, ioctl, read, write 등의 함수를 사용하여 접근 가능하게 한다.

9 문자 디바이스를 커널에 등록하기 위해서 alloc_

chrdev_region, register_chrdev_region, cdev_alloc, cdev_init, cdev_add 등의 함수가 사용되고 등록을 해지하기 위해서 unregister_chrdev_region 및 cdev_del 함수가 사용된다. 디바이스 드라이버를 등록하기 위해서는 파일 동작에 필요한 함수를 구현하고 file_operation 구조체에 각 함수를 지정한 다음 디바이스 이름, 주번호, file_operation 구조체를 앞에서 설명된 등록 함수를 사용하여 커널에 등록한다.

10 문자 디바이스 제어는 일반 파일을 액세스하는 방법과 유사하게 open( ) 함수를 이용하여 디바이스를 열고 ioctl( ), read( ), write( ) 함수를 이용하여 데이터의 입출력 또는 제어를 한 다음 close( ) 함수로 사용을 종료한다.

11 블록 디바이스는 장치와 교환되는 데이터의 단위가 수 킬로바이트의 블록 단위로 이루어지며, 가상 파일시스템(VFS)에 의한 사용자의 요청뿐만 아니라 커널 내의 파일 동작에서 직접 사용되는 경우도 있다.

12 블록 디바이스 드라이버에서 실질적으로 물리적인 저장 매체와 버퍼 캐시 간 데이터를 읽고 쓰는 처리를 수행하는 동작을 처리한다.

13 ifconfig 명령을 사용하여 활성화 및 비활성화한다. 다음은 사용 예다.

- 네트워크 활성화 : ifconfig eth0 192.168.1.10 netmask 255.255.255.0 up
- 네트워크 비활성화 : ifconfig eth0 down

14 데이터 패킷의 전송은 애플리케이션에서 소켓 API를 사용하여 특정 프로토콜에 맞도록 데이터를 만들어 커널의 네트워크 계층에 전달한다. 커널의 네트워크 계층에서는 프로토콜의 구조에 맞게 데이터를 캡슐화(encapsulation)하고 소켓 버퍼를 할당(alloc_skb)받아 데이터 및 데이터의 길이를 포함하여 전송에 필요한 정보를 설정한 후 원하는 목적지로 전송할 디바이스 드라이버의 전송 함수를 호출(dev → hard_start_xmit)한다. 디바이스 드라이버의 전송 함수에서는 입력되는 소켓 버퍼의 정보를 이용하여 하드웨어 장치에 데이터를 기록하여 데이터가 전달되도록 한다. 데이터 전달 후에는 사용된 소켓 버퍼를 해제(dev_kfree_skb)한다.

15 데이터 패킷의 수신은 대부분 인터럽트를 사용한다. 인터럽트의 서비스 루틴에서는 소켓 버퍼를 할당(dev_alloc_skb)받고 하드웨어 장치로 수신된 데이터를 소켓 버퍼에 채운 다음 채워진 소켓 버퍼와 함께 netif_rx( ) 함수를 호출하여 커널의 네트워크 계층에 수신된 데이터가 있음을 알린다. 커널의 네트워크 계층에 전달된 데이터에서는 애플리케이션에서 요구하는 데이터 형태에 맞도록 캡슐 정보를 제거한 후 애플리케이션으로 데이터를 전달한다.

16 리눅스 커널의 일부 기능이나 디바이스 드라이버를 커널 이미지에 포함하지 않고 리눅스 시스템이 시작된 후에 동적으로 탑재하여 사용하는 방식이다. 모듈 탑재에는 insmod 명령이 사용되고 모듈 제거에는 rmmod 명령이 사용된다.

18장. 리눅스 파일시스템과 애플리케이션

1 리눅스의 파일시스템은 매우 다양하다. 일부 임베디드 시스템에서 많이 사용되는 파일시스템은 ext2, ext3, ext4, FAT, FAT32, NFS, cramfs, romfs, ramfs, jffs2, yaffs2 등이 있다.

2 사용자 프로그램에서 파일 이름과 접근 권한을 가지고 open을 호출했을 때 요청한 파일이 존재하고 접근이 가능하면 파일 디스크립터 테이블에 등록하고 파일 디스크립터의 순차적인 등록 번호를 가지게 되는데 이 값을 파일 디스크립터라 한다.

3 열린 파일 테이블(Open File Table)은 열린 파일을 관리하기 위한 자료 구조로, 파일을 읽기용으로 열었는지 쓰기용으로 열었는지를 나타내는 플래그 정보와 파일 내의 위치를 지정하는 오프셋 정보, vnode

테이블의 엔트리를 지정하는 vnode 포인터 정보로 구성된다. vnode(virtual node) 테이블은 가상 파일시스템에서 파일을 관리하기 위한 자료 구조로, vnode 정보와 inode 정보, 그리고 현재 파일의 크기 정보를 포함한다.

4 하드디스크와 플래시 메모리와의 차이점은 다음과 같다.

- 플래시 메모리는 데이터를 기록하기 전에 반드시 지우는 동작이 필요하다.
- 플래시 메모리는 동일 영역을 지속적으로 지우고 기록하기를 반복하면 매체의 수명이 단축될 수 있다.
- 플래시 메모리는 블록이나 섹터 단위로 지우고 기록해야 한다.
- NAND 플래시는 일부 블록이 손상될 수 있다.

5 플래시 기반의 파일시스템을 사용하는 경우 고려해야 할 사항은 다음과 같다.

- 닦기 균등화(Wear Leveling) : 플래시 메모리 기반 파일시스템을 사용하는 경우에는 일부 블록만을 계속 사용하는 것이 아니고 모든 블록을 골고루 사용하도록 만드는 기술
- 가비지 컬렉션(Gabage Collection) : 무효한 블록에서 유효 데이터만을 새로운 블록에 모아서 사용할 수 있는 방법
- NAND 플래시는 일부 블록이 손상될 수 있다. 따라서 NAND 플래시 기반 파일시스템에서는 손상된 블록을 관리하는 특별한 기능을 제공한다.

6 MTD는 메모리 장치, 특히 플래시 메모리를 사용하기 위한 추상적인 계층을 제공하는 장치로, 메모리 하드웨어 장치와 상위 계층 사이에 표준 인터페이스를 제공한다. MTD에서 지원되는 3가지 형태의 하드웨어 장치는 다음과 같다.

- RAM, ROM, NOR 타입의 플래시 칩 드라이버 : NOR 타입의 플래시는 CFI(Common Flash Interface)를 지원하며, 인텔이나 AMD 사의 플래시 인터페이스 규약을 지원한다.
- 시스템 내(Self-contained)의 DRAM 등을 사용한 MTD 드라이버
- NAND 플래시 디바이스 드라이버를 위한 MTD 드라이버

7 저널링 파일시스템은 파일시스템에 손상이 발생한 경우 빠르게 손상된 파일을 복구할 수 있는 파일시스템이다. 사용자에게는 보이지 않는 특수한 저널 파일을 운영하여 파일시스템에 지우기나 쓰기 동작과 같은 변화가 발생하면 이 저널 파일에 변화 내용을 기록한 후에 저장 장치에 실제 데이터를 기록한다. 시스템 오류로 데이터 손상이 발생하는 경우 fsck와 달리 처음부터 손상된 데이터를 검색할 필요 없이 바로 이 저널 파일을 보고 복구한다.
저널링 파일시스템으로는 ext3, ext4, jffs, jffs2, yaffs, yaffs2, LogFS, XFS, ReiserFS 등이 있다.

8 리눅스 파일시스템에서 루트(ROOT:/) 디렉터리를 가지고 있는 파일시스템을 말하며, 리눅스는 반드시 루트 파일시스템을 필요로 한다. 모든 다른 디렉터리는 루트 디렉터리(/)를 기준으로 트리를 구성하여 배치한다.

9 루트 디렉터리, 디바이스 파일, 라이브러리, 시스템 초기화 파일, 셸, 셸 유틸리티 등이 있다.

10 ELF, a.out 그리고 바이너리 스크립트가 있다.

19장. 커널 포팅 준비

1 커널 빌드 환경을 설정하고 make 명령을 사용하여 빌드한다.

2 셸 스크립트 기반 행 단위 설정, Ncurses를 사용한 컬러/텍스트 메뉴 설정 및 GUI 기반 설정 방법이 있다.

3 리눅스 톱(Top) 디렉터리 밑에 .config라는 특수한 파일이 생성된다.

4 커널 빌드 후 생성되는 파일들

파일	내용
linux/vmlinux	ELF(Executable and Linking Format) 형식의 커널 이미지 오브젝트들은 섹션으로 분리되어 있고 프로그램의 링크, 로딩 및 실행에 대한 정보를 가지고 있다. ELF 형식의 파일은 바이너리 파일로의 형식 변환 없이 타깃 보드의 메모리에 탑재될 수 없다.
linux/ System.map	커널의 심볼 및 주소 정보 커널에서 사용되는 함수, 변수 등의 심볼과 각 심볼들의 메모리 내 위치 정보를 가지고 있다. 향후 커널을 디버깅할 때 유용한 정보를 제공한다.
linux/arch/arm/boot/ Image	압축이 안 된 커널 이미지 vmlinux를 바이너리 형식으로 만든 파일이다.
linux/arch/arm/boot/ zImage	부트스트랩 코드를 포함하는 압축된 커널 이미지 순수 커널 이미지를 압축하고 커널에서 제공하는 부트스트랩 코드와 함께 생성된 새로운 커널 바이너리 파일이다.
linux/include/generated/autoconf.h	커널 설정을 C 선행 지시어로 가지는 파일

5 Image 파일은 압축이 안 된 커널 이미지로, vmlinux를 바이너리 형식으로 만든 파일이다.
zImage는 부트스트랩 코드를 포함하는 압축된 커널 이미지로, 순수 커널 이미지를 압축하고 커널에서 제공하는 부트스트랩 코드와 함께 생성된 커널 바이너리 파일이다.

6 zImage에 포함된 부트스트랩 코드는 압축된 커널 이미지를 커널이 동작하는 메모리 영역으로 압축을 풀어서 설치하고, 제어권을 커널 시작 위치로 변경하는 역할을 수행한다.

7 커널의 빌드된 오브젝트를 제거하기 위한 make 옵션으로 clean, mrproper 및 distclean 등이 있다.

8 각 디렉터리에서 커널의 옵션을 설정하기 위해서 사용되는 파일이다.

9 커널 포팅이 완료되면 플래시 메모리나 SD/MMC와 같은 저장 장치에 리눅스 커널 이미지를 저장하고 부팅할 때 자동으로 주메모리로 커널을 설치하여 실행한다. 커널 포팅 중에는 플래시 메모리나 저장 장치에 커널 이미지를 저장하지 않고 시리얼, 네트워크 또는 USB를 사용하여 주메모리로 탑재하여 실행한다.

20장. 커널 포팅

1 선정된 운영체제의 커널이 타깃에서 동작하도록 운영체제의 프로세서 관련 처리, 메모리 관리, 트랩(Trap 또는 Exception) 및 인터럽트 처리, 타이머 등을 구현하여 운영체제의 스케줄러가 정상적으로 동작하도록 하는 과정을 말한다.

2 3GB 사용자 영역 메모리 크기를 사용하는 경우에 0xC0008000를 사용한다.

3 타깃 머신 추가 절차는 다음과 같다.

① 머신 타입을 arch/arm/tools/mach–types에 추가한다.

② 머신 설정 항목 및 메뉴를 arch/arm/Kconfig에 설정한다.

③ 커널 빌드 옵션을 arch/arm/Makefile에 지정한다.

④ 머신 및 프로세서 소스를 추가한다.

⑤ 머신 및 프로세서의 헤더 정보를 arch/arm 디렉터리의 머신 헤더 디렉터리에 추가한다.

⑥ printk를 위한 시리얼 콘솔을 추가한다.

4 arch/arm/mm/ proc–v7.S 파일에 구성된다.

5 arch/arm/tools/mach–types에서 지정한다.

6 트랩은 ARM 프로세서에서의 예외 처리를 말한다. ARM은 리셋, 언디파인드 명령, 소프트웨어 인터럽트(SWI), 프리패치 어보트, 데이터 어보트, IRQ, FIQ 예외 처리를 제공한다. 트랩 초기화 과정에서는 커널 코드에 포함된 각 예외 처리 벡터를 커널의 벡터 위치에 설정하고 각 예외 처리 벡터에서 분기하여 처리되는 예외 처리 핸들러(handler)를 설정하는 과정이다.

7 –march = armv7–a를 사용한다.

8 콘솔 드라이버는 사용자와의 인터페이스를 제공한다. 임베디드 시스템에서는 시리얼, 즉 UART 장치 기

반의 콘솔을 많이 사용하며, 시리얼 콘솔 드라이버는 디버그 콘솔 및 시스템 표준 입출력 콘솔로 사용된다.

9 커널을 디버깅하기 위해서는 ICE 장비와 함께 전용 디버거를 사용하거나 ICE 장비와 함께 GNU 디버거 GDB를 사용하는 방법, 간단하게 메시지를 프린트하는 방법이 있다.
메시지를 프린트하는 방법은 커널에서 제공하는 printch나 printk를 사용하거나 단순하게 시리얼 UART로 메시지를 출력하는 early_printk를 사용할 수 있다.

10 early_printk 함수를 사용한다.

11 printk는 시리얼 콘솔 설정(console_init) 이후에 커널에서 출력하는 메시지를 출력한다. 그 뿐만 아니라 printk는 인터럽트 서비스 루틴에서 메시지를 출력하도록 함수를 호출하면 인터럽트 종료 후에 콘솔로 메시지로 출력한다. 또한 printk는 메시지 출력의 로그 레벨을 지정하여 제어할 수 있다.

12 부트 파라미터는 커널이 부팅하는 과정에서 필요로 하는 다양한 정보를 말한다. 부트 파라미터 정보는 부트로더에서 전달받거나 커널 설정에서 부트 옵션의 명령행(CONFIG_CMDLINE)에 정적으로 지정할 수도 있다.

13 커널 부트 파라미터에 콘솔 장치와 루트 디바이스를 지정하는 방법은 다음과 같다.

- 콘솔 장치 : console=ttySAC2,115200n81
- 루트 디바이스 : root=/dev/ram0

14 Kernel Feature의 SMP 머신 지원 옵션(CONFIG_SMP)과 CPU 개수를 지정하는 옵션이 사용된다.

15 시스템 부팅 후에 /proc/cpuinfor 파일에서 확인할 수 있다.

16 IPI(Inter-processor Interrupt)는 하나의 프로세서에서 다른 프로세서로 보내는 인터럽트로 멀티 프로세서의 동기화에 사용된다.

21장. 루트 파일 시스템

1 커널 초기화의 마무리 단계에서는 루트 파일시스템에 마운트를 시도하고 만약 구현된 루트 파일시스템이 없으면 시스템 부팅은 멈춘다. 상세 절차는 [그림 5-30]을 참조한다.

2 플래시 메모리, SD/MMC 등이 있다.

3 옵션 변수 root를 사용한다. 램디스크를 루트 디바이스로 사용하는 경우에는 root = /dev/ram0과 같이 사용한다.

4 RAM 디스크를 루트 파일시스템으로 사용하는 경우 initrd(Initial RAM Disk)를 사용한다. 원래 initrd는 컴퓨터에서 일반적으로 사용되는 루트 파일시스템에 마운트하기 이전 초기화 동안에 RAM 디스크에 마운트되어 사용되는 루트 파일시스템이라는 의미로 사용되지만 임베디드 시스템에서는 initrd만으로도 필요한 모든 기능을 구축할 수 있다.

5 루트 파일시스템을 구축하는 절차는 다음과 같다.

① 루트 파일시스템 디렉터리 생성
② 디바이스 파일 생성
③ 라이브러리 설치
④ 초기화 프로그램 및 환경 설정 파일 설치
⑤ 셸과 시스템 유틸리티 설치
⑥ 애플리케이션 설치
⑦ 모듈 오브젝트 설치
⑧ 루트 파일시스템 이미지 생성

6 런 레벨(runlevel)을 조정해서 필요한 소프트웨어를 선택적으로 탑재할 수 있다. 예를 들어 런 레벨 1에서는 네트워크 기능을 지원하지 않지만 런 레벨 5에서는 네트워크 기능을 지원하도록 할 수 있다. 런 레벨은 init에서 사용하는 /etc/inittab 파일의 맨 앞에 "id:3:initdefault" 형태로 지정되며 여기서 숫자 3이 런 레벨을 나타낸다.

7 mknod이다.

8 Busybox는 임베디드 시스템에서 루트 파일시스템을 간단하게 만들수 있도록 하는 애플리케이션 프로그램이다. 다수의 리눅스 명령을 하나의 실행 파일 busybox로 만든 후 명령어 이름을 인수로 사용하여 다양한 명령을 실행하거나 각 명령어를 busybox로 심볼릭 링크해서 실행하도록 구성된다.

9 UDEV(User space device)은 핫플러그 시스템의 일부로, 리눅스 커널의 디바이스를 관리한다. 커널에 등록된 디바이스의 디바이스 파일(노드)을 사용자 영역에서 자동으로 만들어주기 위해서 사용된다.

10 인터넷에서 받은 애플리케이션을 복잡성에 따라 빌드하는 방법이 다를 수 있다. 빌드하는 데 필요한 과정은 대부분 애플리케이션의 README 파일 또는 이와 유사한 문서를 참조하면 알 수 있다. 컴파일러를 ARM 리눅스 타깃으로 지정하고 프로세서와 아키텍처에 따라 최적화된 프로그램을 생성하기 위해서는 아키텍처와 프로세서 정보를 사용하는 프로세서에 맞게 설정해서 빌드한다.

11 커널 모듈은 리눅스 커널의 일부 기능이나 디바이스 드라이버를 커널을 빌드할 때 커널 이미지에 포함하지 않고 리눅스 시스템이 시작된 후에 동적으로 탑재해서 사용하는 방식이다. 모듈 오브젝트는 일반적으로 /lib/module 디렉터리에 설치된다.

12 하나의 파일을 디스크처럼 사용하게 해주는 것을 루프백 디바이스라고 한다.

22장. 디바이스 제어

1 문자 디바이스 드라이버 작성 절차는 다음과 같다.

① 디바이스 드라이버 작성 : open, release 함수를 비롯하여 필요한 파일 동작 함수를 구현하고, file_operation 구조체에 각 함수를 지정한 다음 디바이스 이름과 주번호 그리고 file_operation 구조체를 문자 디바이스 등록 함수를 사용하여 커널에 등록한다.

② 디바이스 드라이버 설정 및 빌드 규칙 지정

③ 커널 옵션 설정

④ 커널 빌드

⑤ 애플리케이션 작성

⑥ 애플리케이션 빌드 및 파일시스템에 탑재

⑦ 리눅스 시스템 부팅

⑧ 디바이스 파일 생성

⑨ 시험

2 ifconfig 명령을 사용한다.

3 drivers/usb/host 디렉터리에 ehci-s5p.c와 ohci-exynos.c 파일에 각각 구현된다.

4 PMIC는 시스템에 전원이나 클록을 공급하는 칩이다.

5 I2C 버스는 필립스에서 제안한 통신 방식으로, SCL와 SDA 2개의 선을 이용하여 장치를 제어하기 위해 사용되는 직렬 버스다.

6 그래픽 LCD로 출력되는 데이터를 보관하는 버퍼다. LCD로 출력할 화상을 이 버퍼에 저장하면 일반적으로 DMA 제어기가 LCD 제어기로 데이터를 전달한다.

7 LCD 표시 장치에서 화상을 표출하기 위해 사용되는 광원(백라이트)을 말한다.

8 Video For Linux(V4L)는 TV 튜너나 USB 웹 카메라 같은 다양한 비디오 캡처 디바이스에 접근할 수 있게 해주는 API다. 1999년에 두 번째 버전인 V4L2 인터페이스 개발이 시작되어 V4L가 가지고 있던 버그를 수정하고 더 많은 디바이스를 지원하게 되었다. 리눅스 커널 2.2부터 지원된다.

9 리눅스 커널에서 분리된 고급 리눅스 사운드 아키텍처(ALSA : Advanced Linux Source Architecture)는 사운드 장치 드라이버를 제공하기 위한 커널의 구성 요소다. 사운드 장치를 자동으로 구성하여 머신에 존재하는 여러 사운드 장치를 효과적으로 관리하는 인터페이스를 제공하고 여러 애플리케이션으로부터의 사운드 데이터를 믹싱(mixing)할 수 있는 인터페이스를 제공한다.

10 DAI는 하드웨어적으로 SoC와 코덱(CODEC) 사이의 인터페이스를 제공해주는 방법으로 AC97, I2S, PCM 방식이 있다.

실습 답안

1부. 임베디드 시스템

〈실습 1–1〉 윈도우 및 리눅스 개발 환경 설정

〈부록 A〉를 참조하여 윈도우 및 리눅스 개발 환경을 설정한다. 리눅스 개발 환경은 5부와 6부를 참고하여 설치한다.

2부. ARM 프로세서 이해

〈실습 2–1〉 ARM 프로세서 명령어 실습

〈부록 B〉에 설명된 절차에 따라 각 명령어 구현을 실습한다.

4부. 임베디드 소프트웨어 설계

4부 실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	윈도우 7 이상
개발 툴	ARM 교차 개발 툴, Teraterm, 이클립스 IDE
실습용 소스	부록 CD Practice₩ARM₩Hello–Practice.zip
완성 소스	부록 CD Complete₩ARM₩Hello–Complete.zip

〈실습 4–1〉 LED 제어 프로그램 작성(C 언어)

main.c와 libc.c 파일에 구현된다.

```
--------소스 내용 [main.c] -------------
#if 1
{
  int i;
  Uart_Printf("\n\nLED Display Test\n");

  /* Initialize GPIO port for LED */
  /*
   * Origen : XmDATA0(D13), XmDATA1(D14),
   * XmDATA2(D15)
   * XmDATA0 : GPY5CON[0][3:0] => 0x1 = output
   * XmDATA1 : GPY5CON[1][7:4] => 0x1 = output
   * XmDATA2 : GPY5CON[2][11:8] => 0x1 = output
   */
  /*
   * 1. Initialize GPY5[0], GPY5[1],
   *   GPY5[2] as output port
   *   GPY5CON [3:0] : 0001, [7:4] : 0001,
   *   [11:8] : 0001
   */
  rGPY5CON = (rGPY5CON & ~0xfff) | 0x1 |
    (0x1<<4) | (0x1<<8);

  /*
   * 2. Disable Pull-up for  GPY5[0:2]
   *   GPY5PULL [0]:1, [1]:1, [2]:1
   */
  rGPY5PULL = rGPY5PULL & ~0x3f ;



  /* LED Blink Test with GPIO */
  /*
   * Display value from 0 to 7
   *     bit value  LED3 LED4 LED5
   * 0x0  000 :    X   X   X
   * 0x1  001 :    X   X   0
   * 0x2  010 :    X   0   X
   * 0x3  011 :    X   0   0
   * 0x4  100 :    0   X   X
   * 0x5  101 :    0   X   0
   * 0x6  110 :    0   0   X
   * 0x7  111 :    0   0   0
   */
```

```
    for (i = 0; i < 8; i++) {
     Uart_Printf("LED Display %d\n", i);
     Led_Display(i);
     Delay();
    }
}
#endif
-----------------------------------


---------소스 내용 [libc.c] ------------
void Led_Display(int data)
{
    /*
     * 3. Setup GPIO DATA
     *   Active High Operation
     */
   rGPY5DAT = (rGPY5DAT & ~0x7) | (data & 0x7) ;
}
-----------------------------------
```

5부. 임베디드 ARM 리눅스

실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, 페도라, 우분투 10.04 LTS
개발 툴	리눅스용 ARM-LINUX 타깃 교차 툴
실습용 소스	부록 CD Practice/Linux 디렉터리
완성 소스	부록 CD Complete/Linux 디렉터리

U-boot 실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, 페도라, 우분투 10.04 LTS
개발 툴	리눅스용 ARM-LINUX 타깃 교차 툴
실습용 소스	부록 CD Practice/Linux/U-boot/u-boot-2010.12-dtk4412.tar.bz2 Practice/Linux/Kernel/zImage
완성 소스	부록 CD Complete/Linux/U-boot/u-boot-2010.12-dtk4412.tar.bz2 Complete/Linux/U-boot/u-boot-2010.12-dtk4412-ram.tar.gz

〈실습 5-1〉 U-boot를 포팅하여 DRAM에서 동작

실습 소스는 플래시에서 동작하는 소스다. 이 소스를 수정하여 DRAM으로 usbdl 유틸리티를 이용하여 전송하고 실행하는 실습을 한다. 이때 새롭게 빌드된 U-boot를 eMMC에 탑재하면 안 된다.

〈실습 5-2〉 U-Boot 사용법 실습

〈부록 A〉를 참조하여 U-boot에서 커널을 usbdl 유틸리티를 이용하여 전송하고 실행하는 방법을 실습한다. 이때 리눅스 커널 이미지는 제공된 소스의 Practice/Linux/Kernel/zImage 파일을 사용해도 무방하고, 루트 파일시스템 마운트 동안에 에러가 발생해도 무시한다.

6부. 커널 포팅 및 디바이스 제어

커널 포팅 준비 실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, 페도라, 우분투 10.04 LTS
개발 툴	리눅스용 ARM-LINUX 타깃 교차 툴
실습용 소스	부록 CD Practice/Linux/Kernel/linux-3.6.9.tar.gz
완성 소스	부록 CD Complete/Linux/Kernel/linux-3.6.9-complete-usb+net+i2c+pmic+fb+socket+ipc-20130413.tar.gz

〈실습 6-1〉 리눅스 소스 설치, 빌드, 탑재 및 실행

리눅스 소스를 설치하고 빌드하여 타깃에 탑재하고 실행한다. 이때 이 커널 이미지는 동작하지 않을 수 있으며, 각 절차는 이 책의 각 단계를 따르고 개발 환경은 〈부록 A〉를 참조한다.

DTK4412 머신용으로 완성된 소스는 Complete/Linux/Kernel/linux-3.6.9-complete-usb+net+i2c+pmic+fb+socket+ipc-20130413.tar.gz 파일을 참조한다.

커널 포팅 실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, 페도라, 우분투 10.04 LTS
개발 툴	리눅스용 ARM-LINUX 타깃 교차 툴
실습용 소스	부록 CD Practice/Linux/Kernel/linux-3.6.9.tar.gz
완성 소스	부록 CD Complete/Linux/Kernel/linux-3.6.9-complete-usb+net+i2c+pmic+fb+socket+ipc-20130413.tar.gz

〈실습 6-2〉 타깃 머신 추가 후 커널 빌드

본문을 따라하면서 실습을 완료한다.

〈실습 6-3〉 디버그 메시지 추가후 실행

본문을 따라하면서 실습을 완료한다.

〈실습 6-4〉 커널 포팅

본문을 따라하면서 실습을 완료한다.

〈실습 6-5〉 멀티코어 지원

DTK4412 머신용으로 완성된 소스는 Complete/Linux/Kernel/linux-3.6.9-complete-usb+net+i2c+pmic+fb+socket+ipc-20130413.tar.gz 파일을 참조한다.

루트 파일시스템 실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, 페도라, 우분투 10.04 LTS
개발 툴	리눅스용 ARM-LINUX 타깃 교차 툴
실습용 소스	부록 CD Practice/Linux/Rootfs디렉터리

〈실습 6-6〉 루트 파일시스템 제작

DTK4412 머신용으로 RAMDISK에서 사용되는 INITRD 루트 파일시스템을 제작한다.

디바이스 제어 실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, 페도라, 우분투 10.04 LTS
개발 툴	리눅스용 ARM-LINUX 타깃 교차 툴
완성 소스	부록 CD Complete/Linux/Driver/dtk-chrdev.tar.gz Complete/Linux/Driver/dtk-extio-plat.tar.gz Complete/Linux/Kernel/linux-3.6.9-complete-usb+net+i2c+pmic+fb+socket+ipc-20130413.tar.gz

〈실습 6-7〉 NFS 서버 설치 및 NFS를 이용한 개발 환경 설정

본문의 내용에 따라 호스트 및 타깃 시스템의 NFS 환경을 설정한다. 단, 현재 포팅된 커널은 네트워크 드라이버를 지원하지 않기 때문에 부록 CD에 제공된 zImage를 사용하여 부팅 후에 실습이 가능하다.

〈실습 6-8〉 문자 디바이스 드라이버 구현

문자 디바이스 드라이버를 구현하여 모듈로 빌드한 후에 NFS로 연결하여 동작 시험을 한다.

문자 디바이스 드라이버는 책에 설명된 내용에 따라 open, release, read 및 write 기능을 구현하고, ioctl을 이용한 LED 제어와 외부 버튼을 이용하여 인터럽트를 사용할 수 있는 소스를 포함한다. 애플리케이션은 디바이스 드라이버에 구현된 기능을 시험하도록 구현된다.

완성된 디바이스 드라이버 소스는 Complete/Linux/Driver/dtk-chrdev.tar.gz을 참조한다.

〈실습 6-9〉 커널에 디바이스 드라이버 추가하기

〈실습 6-8〉에서 구현한 디바이스 드라이버를 커널의 drivers/char 디렉터리에 추가하고, 애플리케이션을 루트 파일시스템에 추가하여 자동으로 실행 가능하게 한다.

디바이스 노드는 부팅 시 udev 유틸리티에 의해서 자동으로 생성된다.

〈실습 6–10〉 플랫폼 드라이버 구현

앞에서 실습한 소스를 수정하고, 플랫폼 디바이스 정보를 추가하여 플랫폼 드라이버 구조로 디바이스 드라이버를 작성하고, 시스템 부팅 시 애플리케이션을 자동실행한다.

완성된 디바이스 드라이버 소스는 Complete/Linux/Driver/dtk–extio–plat.tar.gz를 참조한다.

〈실습 6–11〉 USB 버스 드라이버 포팅

책에 설명된 절차에 따라 USB 버스 드라이버를 지원한다.

완성된 소스는 Complete/Linux/ Kernel/linux–3.6.9–complete–usb+net+i2c+pmic+fb+socket+ipc–20130413.tar.gz를 참조한다.

〈실습 6–12〉 네트워크 인터페이스 드라이버 포팅 및 사용

책에 설명된 절차에 따라 네트워크 인터페이스 드라이버를 포팅하고, 인터페이스를 활성화한 후 PING을 이용한 연결성 시험과 NFS 연결 시험 및 모듈 동작을 시험한다.

완성된 소스는 Complete/Linux/ Kernel/linux–3.6.9–complete–usb+net+i2c+pmic+fb+socket+ipc–20130413.tar.gz를 참조한다.

〈실습 6–13〉 프레임버퍼와 LCD 표시 장치

프레임버퍼 드라이버를 포팅하여 LCD 표시 장치가 정상 동작하게 한다. 이때 I2C 버스및 PMIC 드라이버 등의 추가적인 포팅 작업도 필요하다.

완성된 소스는 Complete/Linux/ Kernel/linux–3.6.9–complete–usb+net+i2c+pmic+fb+socket+ipc–20130413.tar.gz를 참조한다.

〈실습 6–14〉 프레임버퍼 활용

실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, 페도라, 우분투 10.04 LTS
개발 툴	리눅스용 ARM–LINUX 타깃 교차 툴
완성 소스	부록 CD Complete/Linux/application/fb_test/fb_test.c

프레임버퍼를 활용하여 LCD에 [그림 6–47]과 같이 표출한다. 완성된 소스는 Complete/Linux/application/fb_test/fb_test.c에 구현된다.

〈실습 6–15〉 Qt/Everywhere 설치와 실행

실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, 페도라, 우분투 10.04 LTS
개발 툴	리눅스용 ARM–LINUX 타깃 교차 툴
실습 소스	부록 CD Practice/Linux/Qt/qt–everywhere–opensource–src–4.8.0.tar.gz
완성 소스	부록 CD Complete/Linux/Qt/qt–complete.tar.bz2

본문의 순서에 따라 Qt/Everywhere를 컴파일하고 타깃에 탑재하고 실행한다. 빌드된 Qt 바이너리는 Complete/Linux/Qt/qt–complete.tar.bz2에 있다.

7부. 안드로이드 탑재와 활용

〈실습 7–1〉 안드로이드 커널 빌드

실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, 우분투 10.04 LTS
개발 툴	리눅스용 ARM–LINUX 타깃 교차 툴

실습 소스	부록 CD Practce/Android/Kernel/android-kernel-dtk4412.tar.gz
완성 소스	부록 CD Practce/Android/Kernel/android-kernel-dtk4412.tar.gz Complete/Android/zImage

본문의 순서에 따라 안드로이드 커널을 빌드한다.

〈실습 7-2〉 안드로이드 플랫폼 빌드

실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, 우분투 10.04 LTS
개발 툴	리눅스용 ARM-LINUX 타깃 교차 툴, 안드로이드 빌드에 필요한 기타 툴
실습 소스	부록 CD Practce/Android/Vendor/dtk4412_ics_vendor.tar.gz Practce/Android/Vendor/dtk4412_ics_device.tar.gz
완성 이미지	부록 CD Complete/Android//fastboot Complete/Android//zImage Complete/Android/ramdisk.img.ub Complete/Android/system.img

본문의 순서에 따라 안드로이드 플랫폼을 빌드한다.

〈실습 7-3〉 안드로이드 탑재 및 실행

실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, 우분투 10.04 LTS
개발 툴	리눅스용 ARM-LINUX 타깃 교차 툴, 안드로이드 빌드에 필요한 기타 툴
완성 이미지	부록 CD Complete/Android//fastboot Complete/Android//zImage Complete/Android/ramdisk.img.ub Complete/Android/system.img

본문의 순서에 따라 안드로이드 커널 및 램디스크, 시스템 이미지를 타깃 메모리에 탑재하고 실행한다.

〈실습 7-4〉 이클립스 및 안드로이드 SDK 설치

실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, 우분투 10.04 LTS
개발 툴	이클립스, 안드로이드 SDK

본문의 순서에 따라 이클립스 및 안드로이드 SDK를 설치한다.

〈실습 7-5〉 안드로이드 애플리케이션 생성 및 실행

실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, 우분투 10.04 LTS
개발 툴	이클립스, 안드로이드 SDK

본문의 순서에 따라 Hello Android 애플리케이션을 생성하고 안드로이드 가상 디바이스에서 실행한다.

〈실습 7-6〉 안드로이드 애플리케이션 타깃에 탑재 및 실행

실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, 우분투 10.04 LTS
개발 툴	이클립스, 안드로이드 SDK

본문의 순서에 따라 Hello Android 애플리케이션을 타깃 디바이스에서 실행한다.

부록 A. 개발 환경 설치 및 사용법

실습 환경 1

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	윈도우 7 이상
개발 툴	ARM 교차 개발 툴, Tertaterm, 이클립스 IDE
실습용 소스	부록 CD Practice₩ARM₩Hello-Practice.zip
완성 소스	부록 CD Complete₩ARM₩Hello-Complete.zip

실습 환경 2

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	리눅스, FedoreCore 6, 우분투 10.04 LTS
개발 툴	리눅스용 ARM-LINUX 타깃 교차 툴
실습용 소스	부록 CD Practice/Linux 디렉터리
완성 소스	부록 CD Complete/Linux 디렉터리

〈실습 A-1〉 윈도우 기반 개발 환경 설정 및 Hello 프로젝트 설치와 실행

부록에 있는 절차에 따라 각 툴을 설치하고 Hello 프로젝트를 생성한 후 타깃으로 바이너리를 다운로드하고 실행한다.

〈실습 A-2〉 페도라 기반 리눅스 개발 환경 설치

부록에 있는 절차에 따라 페도라 리눅스를 설치하고 GNU 툴을 설치한다. 환경 설정이 완료되면 리눅스 커널과 루트 파일시스템을 빌드하고 타깃으로 다운로드해서 실행한 후 타깃에서 임베디드 리눅스가 부팅되면 NFS를 연결하고 모듈로 드라이버 소스를 구현한 다음에 실습한다. 이 실습에서는 개발 환경만 설치하고 커널 빌드 및 탑재, 실행부터는 5부와 6부를 참고해도 무방하다.

〈실습 A-3〉 우분투 10.04 기반 리눅스 개발 환경 설치

부록에 있는 절차에 따라 우분투 10.04 LTS 리눅스를 설치하고 GNU 툴을 설치한다. 환경 설정이 완료되면 리눅스 커널과 루트 파일시스템을 빌드하고 타깃으로 다운로드해서 실행한 후 타깃에서 임베디드 리눅스가 부팅되면 NFS를 연결하고 모듈로 드라이버 소스를 구현한 다음에 실습한다. 이 실습에서는 개발 환경만 설치하고 커널 빌드 및 탑재, 실행부터는 5부 및 6부를 참고해도 무방하다.

부록 B. ARM 어셈블리 프로그래밍

실습 환경

하드웨어	DTK4412
개발용 PC	펜티엄 프로세서 이상, UART, 랜 및 USB 포트 보유
	윈도우 7 이상
개발 툴	ARM 교차 개발 툴, Teraterm, 이클립스 IDE
실습용 소스	부록 CD Practice₩ARM₩Hello-Practice.zip
완성 소스	부록 CD Complete₩ARM₩Hello-Complete.zip

〈실습 B-1〉 Hello 예제 프로젝트 실행

윈도우 환경을 사용하여 부록 B의 Hello 예제 프로그램 실행 단계1부터 단계4까지 순차적으로 진행한다.

〈실습 B-2〉 입출력 장치 동작 없는 Hello 예제

윈도우 환경을 사용하여 LED 시험을 하지 않도록 소스를 수정하고 빌드하여 다운로드한 후 실행한다.

```
----------소스 내용 [main.c] ------------
#if 0
{
   int i;
   Uart_Printf( " \n\nLED Display Test\n " );
…… 이하 생략 ……
-----------------------------------
```

〈실습 B-3〉 함수의 반환

```
----------소스 내용 [libs.s] ------------
 .global  HOW_TO_RETURN
HOW_TO_RETURN:
 /* IMPLEMENT HERE : return ?
  */
 mov pc, lr
-----------------------------------
```

〈실습 B-4〉 조건부 실행

```
----------소스 내용 [libs.s] ------------
 .global  CONDITIONAL_EXECUTE
CONDITIONAL_EXECUTE:
 /* IMPLEMENT HERE
  * use CMP and MOV instruction only : 4 line
  * result must be stored to r0 as follow ATPCS
  */
 cmp r0, r1
 movlt r0, #1
 movgt r0, #2
 moveq r0, #3

 mov pc, lr /* return *
-----------------------------------
```

〈실습 B-5〉 데이터 처리 명령(산술 연산)

```
----------소스 내용 [libs.s] ------------
  .global DATA_PROCESS1
DATA_PROCESS1:
 /*
  * IMPLEMENT function for calcurate
  * result=(a+b)-c in this location
  * use ADD, SUB and MOV instruction : less
  * than 3 line
  * use register only : r0~r3
  */
  add r0, r0, r1
  sub r0, r0, r2

  mov pc, lr/* return */
-----------------------------------
```

〈실습 B-6〉 데이터 처리 명령(논리 연산)

```
----------소스 내용 [libs.s] ------------
  .global DATA_PROCESS2
DATA_PROCESS2:
 /*
  * IMPLEMENT function for calcurate
  * result=(a<<2) | (b&15) in this location
  * use AND, ORR and MOV instruction : less
  * than 3 line
  * use register only : r0~r3
  */
  and r3, r1, #15
  orr r0, r3, r0, lsl #2

  mov pc, lr/* return */
-----------------------------------
```

〈실습 B-7〉 분기 명령과 연산 명령어를 사용한 데이터 처리

```
----------소스 내용 [libs.s] ------------
 .global  SUM_OF_DEC
SUM_OF_DEC:
 /* IMPLEMENT HERE
  */
 mov r3, #0x0   @ initialize sum
sum_loop:
 add r3, r3, r0 @ add total to r3
 add r0, r0, #1  @ increase 1
 cmp r0, r1     @ compare start and end
 ble sum_loop  @ compare

 mov r0, r3     @ copy to argument
 mov pc, lr /* return *
-----------------------------------
```

〈실습 B-8〉 LDR/STR을 이용한 메모리 복사 구현

```
----------소스 내용 [libs.s] ------------
 .global  MEMCPY_SINGLE
MEMCPY_SINGLE:
 stmfd  sp!, {r4-r5, lr}   @ push
 /* IMPLEMENT HERE */
copy_loop_single:
 ldr r3, [r1], #4    @ load source data
```

```
 str r3, [r0], #4  @ store data to destination
 subs r2, r2, #1  @ decrease size
 bne copy_loop_single

 ldmfd  sp!, {r4-r5, pc} @ pop
-----------------------------------
```

```
 strb r3, [r0], #1  @ store data to destination
 subs r2, r2, #1    @ decrease size
bne copy_loop_single_byte

 ldmfd sp!, {r4-r5, pc}  @ pop
-----------------------------------
```

〈실습 B-9〉 LDM/STM을 이용한 메모리 복사 구현

```
---------소스 내용 [libs.s] ------------
 .global  MEMCPY_MULTIPLE
MEMCPY_MULTIPLE:
 stmfd  sp!, {r4-r5, lr}  @ push
 /* IMPLEMENT HERE */
copy_loop_multiple:
 ldmia  r1!, {r3-r5}
 stmia  r0!, {r3-r5}
 subs r2, r2, #3          @ decrease size
 bne copy_loop_multiple

 ldmfd  sp!, {r4-r5, pc}  @ pop
-----------------------------------
```

〈실습 B-10〉 메모리 복사 함수 구현

```
---------소스 내용 [libs.s] ------------
 .global  MEMCPY
MEMCPY:
 stmfd sp!, {r4-r5, lr}  @ push
 /* IMPLEMENT HERE */
copy_loop_multi_word:
 ldmia r1!, {r3-r5}
 stmia r0!, {r3-r5}
 sub r2, r2, #12
             @ decrease 3 word(12byte)size
 cmp r2, #12
 bge copy_loop_multi_word

copy_loop_single_word:
 ldr r3, [r1], #4  @ load source data
 str r3, [r0], #4  @ store data to destination
 sub r2, r2, #4    @ decrease size
 cmp r2, #4
 bge copy_loop_single_word

copy_loop_single_byte:
 ldrb r3, [r1], #1@ load source data
```