<Power Programming>

소프트웨어 구조와 모듈 설계

Version	Document maturity (draft / valid)	Date of Issue (200x-MM-DD)	Author/Owner	Check/Release	Description
1.0		2008-02-04	W. Lee		Initial version
1.1		2012-04-30	W. Lee		Separate from ADPro_A

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

그림 목차 항목을 찾을 수 없습니다.

LIST OF CODE

그림 목차 항목을 찾을 수 없습니다.

<Power Programming>

소프트웨어 구조와 모듈 설계

Chapter 01
마이크로컨트롤러용 소프트웨어의 구조

Version	Document maturity (draft / valid)	Date of Issue (200x-MM-DD)	Author/Owner	Check/Release	Description
1.0		2008-04-14	W. Lee		Initial version
1.1		2012-04-30	W. Lee		Separate from ADPro_A

1 소프트웨어 구조의 예

1.1 서론

- 지난 수년간 "소프트웨어 구조"에 대하여 많은 연구
  - 대규모 프로그램을 위한 소프트웨어 구조
- 이와 같은 표준적인 소프트웨어 구조는 마이크로컨트롤러용으로 사용하기에 부적합
- 마이크로컨트롤러 프로그램에는 소프트웨어 구조가 필요없다?
  ➔ 더욱 더 잘 설계된 소프트웨어 구조 필요
  - 비기능적 요구사항에 부합하는 소프트웨어 구조
  - 표준적인 구조에서 필수적인 요소만 채택
  - 자원을 낭비할 수 있는 요소는 과감히 생략

[핵심]

- 일반적인 표준 구조는 일반적으로 마이크로컨트롤러에 적용하기에 너무 크고 무겁다.
- 마이크로컨트롤러를 위한 소프트웨어 구조는 표준적인 구조에서 필수적인 요소들만 선택적으로 채택하여 오버헤드가 최소화 될 수 있도록 하여야 한다.

1.2 소프트웨어 구조

- 마이크로컨트롤러 소프트웨어를 위한 구조를 제시
  - 절대적이라기 보다는 참고로 여길 것
  - 필요에 따라 변형하여 사용

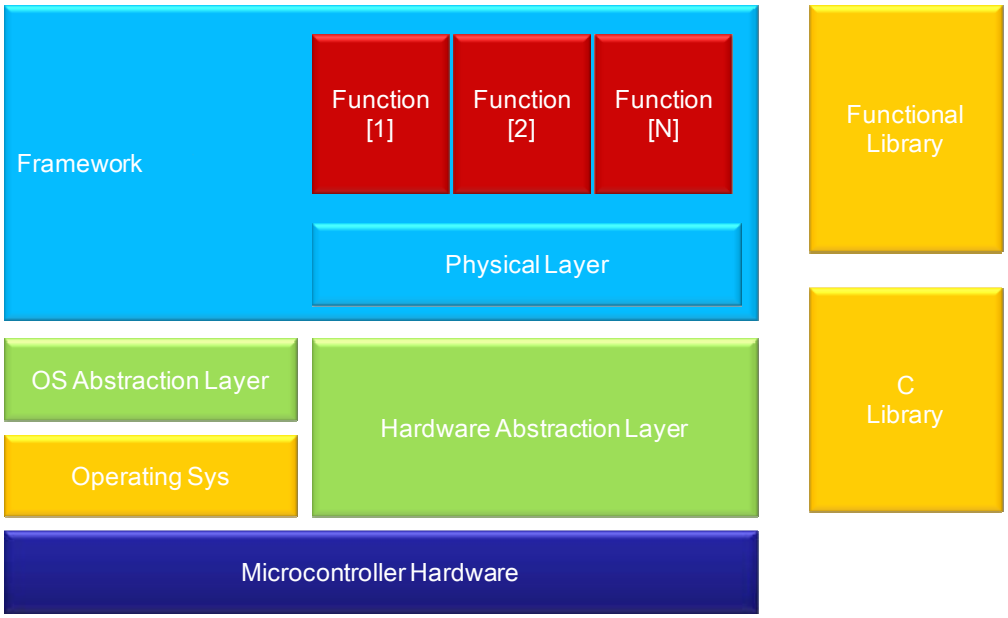

FIGURE 1-1 마이크로컨트롤러용 소프트웨어 구조의 예

- 마이크로컨트롤러와 운영체계는 주어진 것으로 간주
- 마이크로컨트롤러
  - CPU 와 많은 입출력 장치들로 구성

- 나머지 소프트웨어 부분이 마이크로컨트롤러의 입출력 장치들을 효과적으로 사용하여 원하는 동작 수행하도록 만들어야 함.

■ 운영체계
- 운영체계는 운영체계 자체는 마이크로컨트롤러와 상당히 밀접한 관계를 가지고 있음
- 보통 필요한 기능을 구현하기 위하여 고급 프로그래밍 지식을 필요로 하고 전문적인 팀에 의하여 체계적으로 개발하여야 함.
- 일반적으로 소프트웨어 개발시 운영체계는 함께 개발하는 것 보다는 기성품을 사용하거나 별도의 팀에서 개발하여 사용.
- 운영체계를 어떻게 다른 소프트웨어 부분과 연결하는 지가 중요한 사항

■ 하드웨어 추상계층과 운영체계 추상 계층
- 하드웨어와 운영체계를 활용하기 위하여 특별히 고안
- 이 계층 자체는 재사용이 불가능
- 이 계층을 통하여 상위의 소프트웨어 컴포넌트에 대한 비기능정 요구사항을 만족시킬 수 있음

■ 이식성 향상
- 가능한 한 많은 소프트웨어 컴포넌트를 이식가능하게 할 수 있음
- 표준 C 언어로 프래임워크, 기능블럭, 물리계층, 기능 라이브러리를 프로그래밍 할 것
- 특정 컴파일러 혹은 CPU 의 관련 기능은 C 라이브러리에 모아둘 것

■ 재사용성 향상
- 이식이 가능한 부분은 동시에 재사용이 가능해짐
- 재사용성을 더욱 향상 시키기 위해 물리계층을 도입함
- 하드웨어에 의존하는 각종 값과 이 값을 사용해야 하는 기능 블록 사이에 물리계층을 도입하여 물리량과 하드웨어의 값의 변환을 담당; 기능블럭의 재사용성을 더욱 높여줄 수 있음

■ 유지보수성 향상
- 소프트웨어를 용도 및 성격에 따라 모듈과 계층으로 나누어 구성
- 상호 의존성을 최대한 낮추고 각각의 고유한 기능을 명확히 하여줌
- 모듈 설계를 통하여 유지보수성을 더욱 향상시킬 수 있음

[핵심] 마이크로컨트롤러용 소프트웨어 구조

■ 마이크로컨트롤러 부분과 운영체계를 정의하고 이 컴포넌트를 연결할 수 있어야 한다.

■ 하드웨어 추상화 계층과 운영체계 추상화 계층을 사용하여 마이크컨트롤러와 운영체계 컴포넌트를 연결할 수 있다

■ 이와 같은 소프트웨어 구조를 사용함으로 이식성, 재사용성, 유지보수성을 높일 수 있다.

1.3 하드웨어 추상화 계층 (HAL: Hardware Abstraction Layer)

■ 마이크로컨트롤러의 하드웨어 부분과 소프트웨어의 나머지 부분을 분리하는 목적
- 마이크로컨트롤러의 하드웨어/컴파일러 의존적인 부분을 한 곳을 모으자.

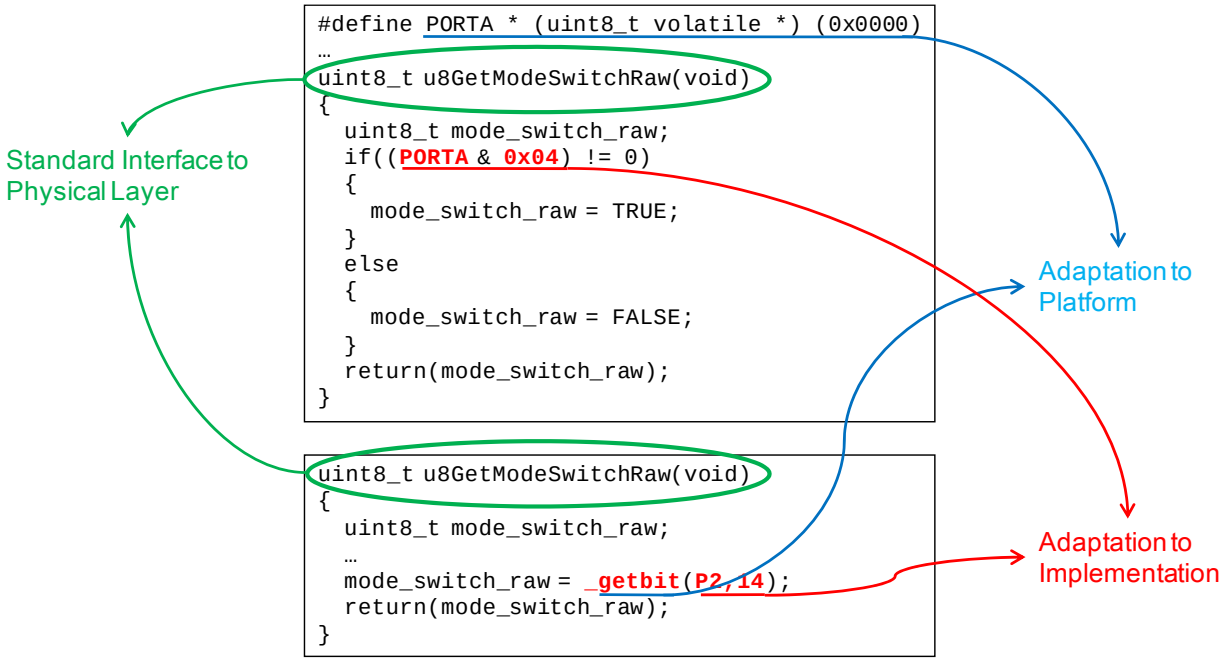

FIGURE 1-2 하드웨어 추상화 계층의 예

- 스위치의 상태를 판별하는 함수
- 물리계층에 표준적인 인터페이스를 제공
- 하드웨어 의존적인 부분, 혹은 컴파일러 의존 적인 부분을 접근하는 방법으로 내부 구현
- 구현되는 하드웨어에 따라 변경이 있는 부분을 내부 구현

■ 하드웨어 추상화 계층을 만들 때 염두에 둘 것들

- 마이크로컨트롤러 하드웨어의 전 범위를 다뤄야 한다; 일부만 추상화 하면 결국은 추상화 되지 않은 부분에 의해 의미가 없어진다.
- 간단하고 일관성 있는 인터페이스를 설계해야 한다.; 모듈명을 최대한 활용
  1. 장치 초기화를 위한 간단한 초기화 함수 제공
     `void Can_Init(void), void Sci_Init(void)`
  2. 간단한 자료를 주고 받을 경우는 “get”, “set”을 활용
     `uint8_t Dio_u8GetSwitchRaw(void)`
  3. 버퍼를 전달할 경우는 버퍼 포인터와 버퍼 길이를 함께
     `void Sci_vLoadTxBuffer(uint8_t * TxBuffer, uint8_t Length)`
  4. 특정 동작을 시작시키고자 할 때 단순한 시작함수로.
     `void Sci_vTxStart(void)`

[핵심]

■ 하드웨어 추상화 계층은 컴파일러/CPU 등의 일반적이지 않은 요소들을 포함한다.
■ 하드웨어 추상화 계층은 하드웨어의 특이한 요소들을 포함한다.
■ 하드웨어 추상화 계층은 다음의 일반적인 함수들로 구성될 수 있다.; 초기화, 버퍼 다루기, 개별 자료 접근, 트리거 동작 등등

1.4 운영체계 추상화 계층 (OS: Operating System Layer)

■ OS 추상화 계층은 일반적으로 널리 사용되지는 않는다.
■ 운영체계의 아주 간단한 기능만을 사용하는 경우

- 추상화 계층이 필요없다.

■ 운영체계의 복잡한 기능을 사용하는 경우
- 이벤트 방식의 태스크 수행
- 세마포어
- 태스크간의 통신 등등

[해결책 1] 표준적인 운영체계를 사용한다.
- 자동차업계의 OSEK

[해결책 2] 운영체계 추상화 계층을 사용한다.
- 하드웨어 추상화 계층과 같이 특정 운영체계에 대한 의존성을 감소시킨다.

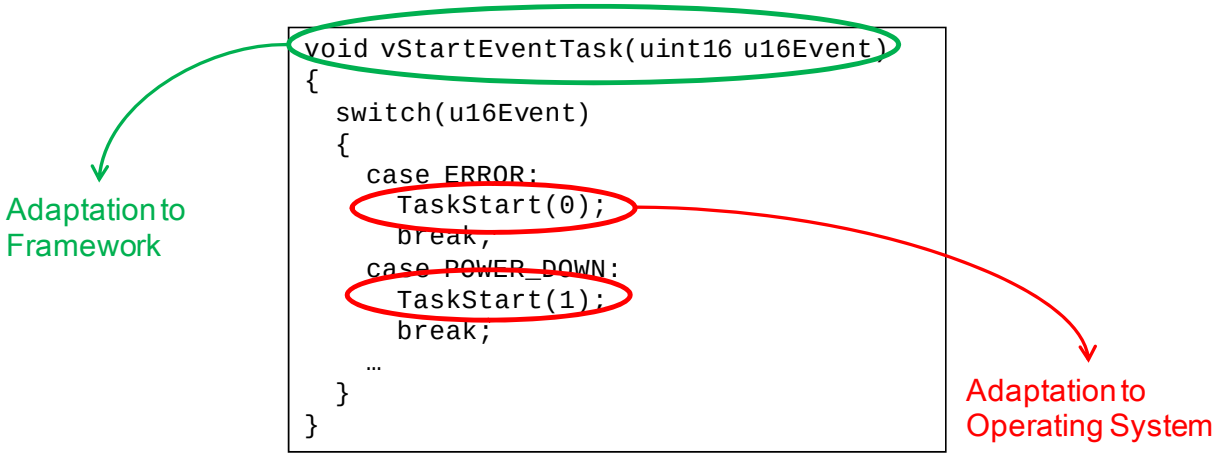

FIGURE 1-3 OS 추상화 계층의 예

[핵심]
■ 운영체계 추상화 계층은 사용하지 않을 수도 있다.
■ 운영체계에 따라 달라질 수 있는 특징들, 세마포어 혹은 태스크간 통신 등,을 사용할 때는 운영체계 추상화 계층을 사용하는 것이 유익하다.
■ 운영체계 추상화 계층은 여러 플랫폼 혹은 여러 운영체계 상에서 입출력 장치를 다룰 때 매우 도움이 된다.

1.5 물리계층 (Physical Layer)

■ HAL 을 통해서 얻은 값은 마이크로컨트롤러에서 직접적으로 사용하는 값이다. → 프로그래머가 바로 이해하기에는 어려운 값
- 직접적인 DIO 입력
- timer 의 tick 값으로 표현된 시간
- ADC 의 변환값

■ 물리계층에서 응용프로그램에서 사용하기 위한 물리량, 가능하다면 정규화된 값으로 변환한다.
- 물리량으로 변환시 충분한 정밀도를 갖도록 하여야 한다.
  : 예를 들어 1V 의 정밀도는 불충분, 0.01V 의 정밀도 활용
- 값의 변환을 위하여 Shift 연산을 최대한 활용하여 연산의 속도를 높인다.
  : (operand) >>1 은 /2 의 효과, <<1 은 *2 의 효과

(ex) (ADC_value * 125) >> 6

[예] 8~16V 의 전압을 5V 의 입력범위를 갖는 10bit ADC 를 사용하여 읽어들이기

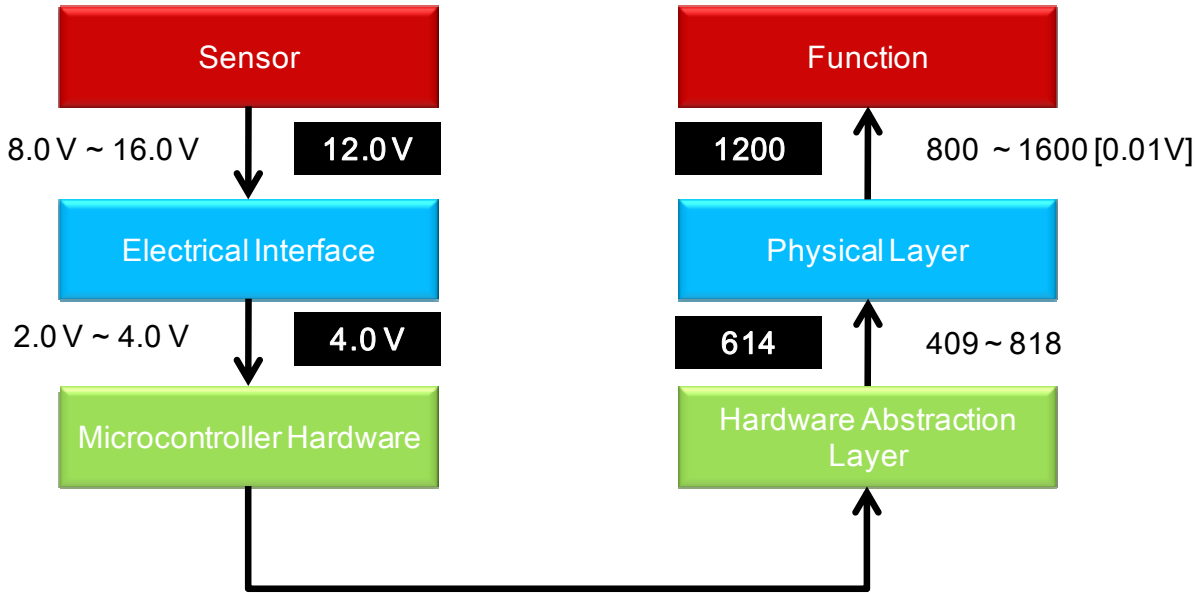

FIGURE 1-4 물리계층과 물리량

- 디지털 입력의 디바운싱 혹은 필터링과 같은 부가기능을 추가할 수 있다.

```
uint16_t u16GetSwitchState(void)
{
  static uint16_t FirstState, SecondState, ThirdState;
  uint16_t switch_state_raw, switch_state;
  switch_state_raw = u16GetSwitchStateRaw(void);
  FirstState = SecondState;
  SecondState = ThirdState;
  ThirdState = SwitchStateRaw;
  if((ThirdState==SecondState)&&(SecondState==FirstState))
  {
    switch_state = switch_state_raw;
  }
  return(switch_state);
}
```

Interface to HW Abstraction Layer

FIGURE 1-5 물리계층과 디바운싱

[핵심]

- 물리계층은 기능과 관계된 컴포넌트를 이식가능하게 만드는데 결정적인 도움을 준다.
- 물리계층은 하드웨어 추상화 계층의 가공되지 않은 물리값을 정규화된 물리값으로 변환하여 준다.
- 물리계층은 디바운스나 필터링과 같은 부가의 기능을 포함할 수 있다.

1.6 프래임워크(Framework)

- 프래임워크는 마이크콘트롤러 소프트웨어 구조의 기본 구성요소
- 간단한 형태로 수행하고자 하는 함수들을 위한 틀을 제공
- 개발자는 이 틀안에 필요한 기능을 프로그래밍
- 이 프래임워크를 통하여 개발자는 한 모듈에서 전체 시스템의 제어흐름을 설계하고 구현할 수 있다.

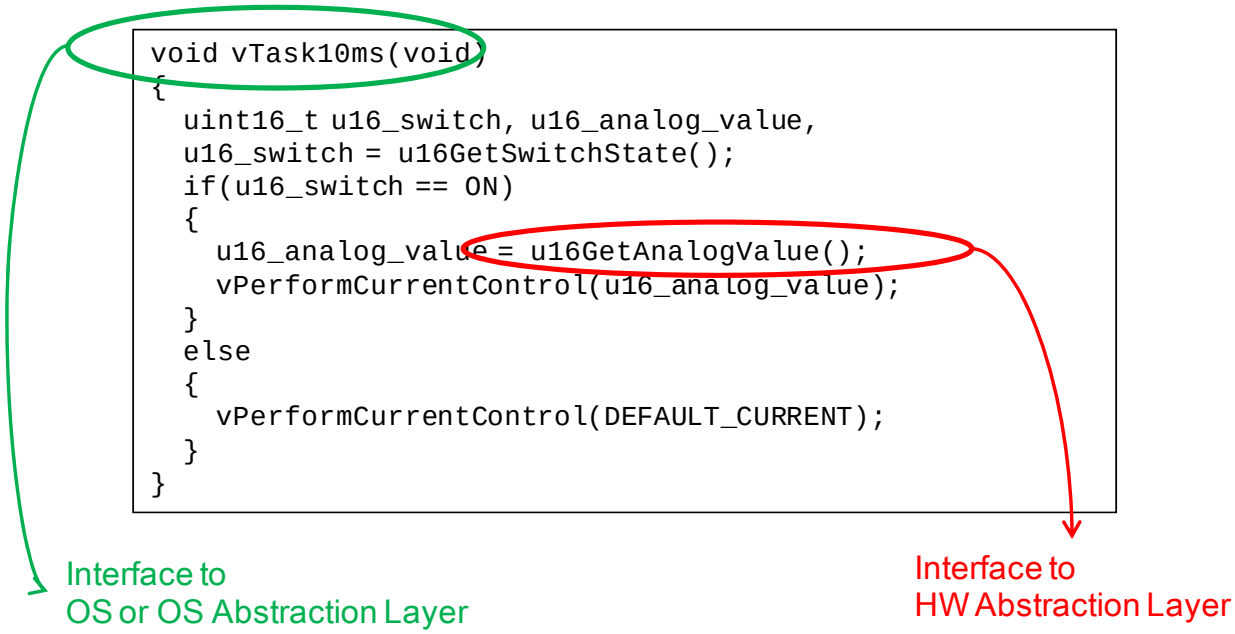

FIGURE 1-6 프래임워크와 하위계층과의 연결

- 하나의 모듈별로 프레임워크를 구성하고 다음의 기능을 갖도록 한다.
  1. 초기화 함수: 다른 모듈의 초기화 함수를 호출, 시스템 초기화에서 이 함수가 호출되도록 구성
  2. 주기적 태스크: 주기적 동작을 연결, OS 에서 주기적으로 호출되는 태스크, (주의) 빠른 주기 태스크에서 느린 주기 태스크를 호출하는 방식으로 설계하지 말 것
  3. 이벤트적 태스크: 이벤트에 의해 시작되어야 하는 동작 연결
  4. 백그라운드 태스크: 시작 제약 사항이 없거나 낮은 동작 연결
  5. 인터럽트 서비스는 이 프레임워크를 통해서 연결하지 말 것. 인터럽트 동작은 가능하면 HAL 에서 처리

[핵심]

- 프래임워크는 하나의 모듈에 있어야 한다.
- 프래임워크는 각각의 주기적/이벤트적 태스크들을 수행시키는 함수를 가지고 있어야 한다.
- 프래임워크에 있는 각각의 함수는 태스크들의 제어흐름을 담고 있어야 한다.
- 자료들은 프래임워크 상에 흩어져 있으면 안된다. 프래임워크는 제어흐름에 영향을 줄 수 있는 자료들만 담고 있어야 한다.

1.7 라이브러리

- 라이브러리는 소프트웨어 재사용의 가장 일반적이며 효율적인 형태
- 라이브러리 활용의 장점
  - 소프트웨어의 재사용
  - 시험 커버리지를 높일 수 있다.
  - 문서화 노력을 줄일 수 있다.
- 기능라이브러리와 C 라이브러리로 구분할 수 있다.

1.7.1 기능라이브러리

- 기능 라이브러리의 특징
  1. 동작이 복잡하고 크다. 그리고 여러 곳에서 사용되어진다. ➔ 공통으로 추출함으로 ROM 사용을 줄일 수 있다.

2. 표준 C 언어로만 프로그래밍 한다. 즉 PC 시스템에서도 수행할 수 있다.
3. 정적인 데이터를 가질 수 있고, 이 경우 사용할 때마다 개별 인스턴스를 사용하도록 한다.

- 정적 데이터를 사용하는 경우 주의 사항
  - 정적인 변수는 모듈별로 선언되어지지, 함수별로 선언되지 않음
  - 복수개의 정적변수를 활용하는 함수를 설계하고자 할 경우 개별 인스턴스를 선언하여 사용
  - 함수에 인스턴스를 전달하는 매개변수를 포함하여야 함.

```
void vInitSensor(uint8_t u8_channel)
{
  …
}

uint8_t u8CheckSensor(uint8_t u8_channel)
{
  …
}
```

FIGURE 1-7 기능라이브러리와 인스턴스

1.7.2 C 라이브러리

- C 라이브러리의 특징
  1. 기능이 작고 간단하며 정적데이터가 필요 없는 경우
  2. 공통으로 여러 곳에서 사용되어진다.
  3. CPU/컴파일러의 특별한 특징을 은닉할 수 있다. ➔ CPU 특징적인 기능과 구문을 추상화 하여 사용

[핵심]

- 기능라이브러리와 C 라이브러리는 기능 컴포넌트 구현에 필요한 일반적인 기능들을 담고있다.
- 기능라이브러리는 표준 C 스타일로 프로그래밍 하여야 한다.
- 기능라이브러리는 정적변수 혹은 여러 개의 인스턴스를 포함하는 복잡한 함수도 가질 수 있다.
- C 라이브러리는 컴파일러/CPU 의 특정 구문과 코드등 포함하고 이를 은닉시켜 기능 컴포넌트에 표준적인 인터페이스를 제공한다.
- C 라이브러리는 정적변수를 사용하지 않는 단순한 기능들 만을 포함하도록 한다.

1.8 결론

- 제안한 구조에서 추가될 수 있는 요소
  - 통신
  - 메모리 처리
  - 시스템 서비스
- 제안한 구조를 활용하여 플랫폼 구성
  - 표준 하드웨어
  - 표준 OS
  - 표준 HAL
  - 표준 물리계층
  - 표준 프래임워크
- 플랫폼이 가져야 하는 중요한 특징
  - Scalability, Flexibility

■ 응용 분야별로 응용 소프트웨어 부분도 계층화, 표준화가 가능하지만, 지나치게 표준화하는 것이 해결책은 아님

[예 1] AUTOSAR

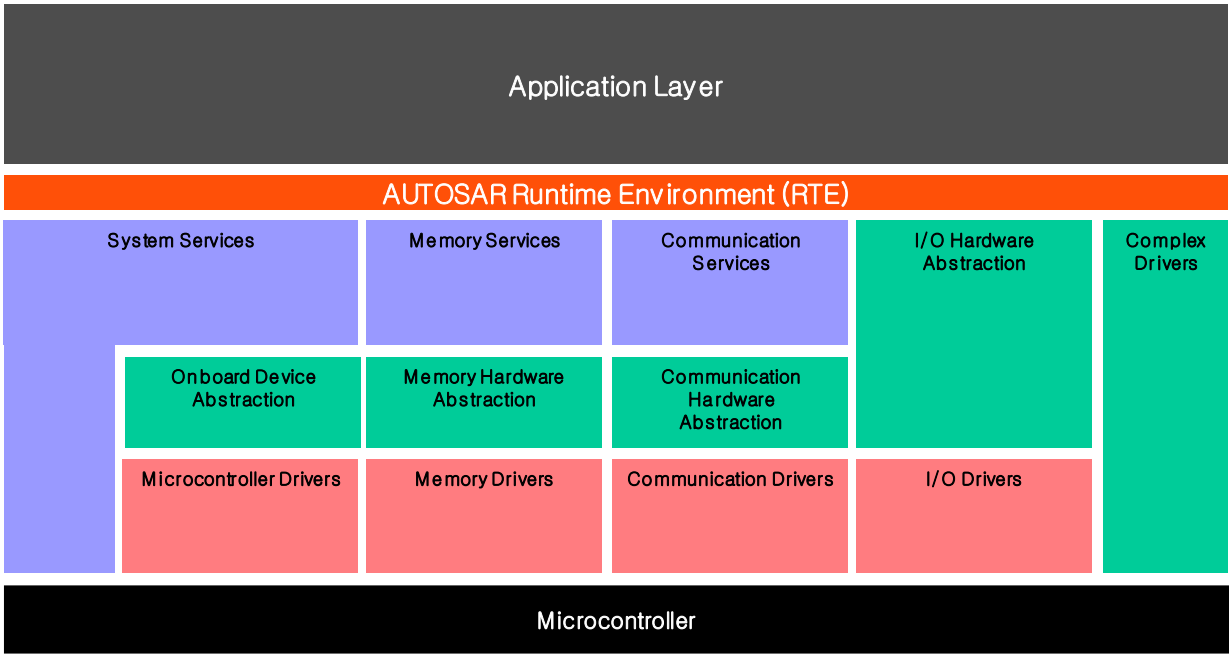

FIGURE 1-8 AUTOSAR SW Architecture

[핵심]

■ 소프트웨어 구조를 통하여 일부의 비기능적 요구사항을 만족시켜줄 수 있다. 다른 것들은 모듈 설계를 통하여 해결할 수 있다.

<Power Programming>

소프트웨어 구조와 모듈 설계

Chapter 02
마이크로컨트롤러 구조와 운영체계가 소프트웨어 구조에 미치는 영향

Version	Document maturity (draft / valid)	Date of Issue (200x-MM-DD)	Author/Owner	Check/Release	Description
1.0		2008-05-13	W. Lee		Initial version
1.1		2012-04-30	W. Lee		Separate from ADPro_A

2 마이크로컨트롤러 구조와 운영체계

2.1 서론

- 마이크로컨트롤러와 OS 의 선택은
  - 시스템의 동적 동작을 결정하는 중요한 요소
  - 비기능적 요구사항과 깊은 연관
- 마이크로컨트롤러의 구분
  - 일반 CPU 를 사용한 마이크로컨트롤러
  - 전용 마이크로컨트롤러
- OS 의 구분
  - 협력적인 운영체계
  - 선점형 운영체계
- 마이크로컨트롤러와 OS 의 다양한 조합이 가능
- 각 조합의 특징을 이해하고 적절한 응용분야를 선정하는 것이 중요

2.2 마이크로 컨트롤러 구조

2.2.1 일반 CPU 를 사용한 마이크로컨트롤러의 구조

- 대표적 사례: PowerPC, ARM core

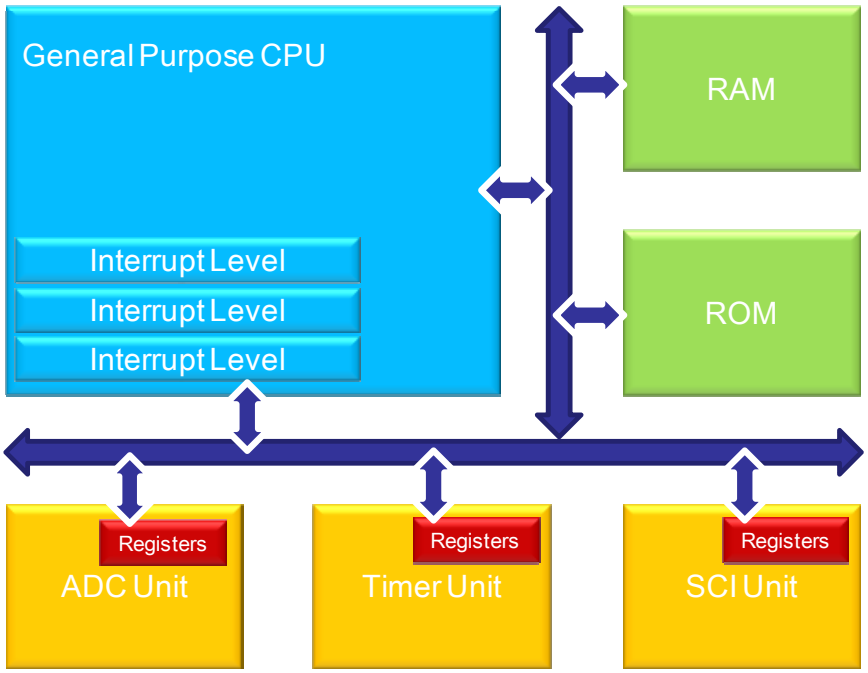

FIGURE 2-1 일반 CPU 를 사용한 마이크로컨트롤러의 구조

[핵심]

- 주변장치들이 매우 강력하며 CPU 와 독립적으로 동작한다.
- 주변장치들이 내부 버스를 통하여 CPU 와 연결되며 필연적으로 시간지연을 갖고 반응하게 된다.
- 주변장치의 프로그래밍이 상대적으로 복잡하고 여러 제한조건을 갖고 있다.
- 주변장치의 각종 이벤트들을 개별적인 인터럽트로 할당하기가 불가능하다.

- 제공되는 인터럽트 레벨의 개수가 제한되어 있다.
- CPU 코어를 대량 생산하는 모델이 일반적이어서 상대적으로 가격이 저렴하다.

2.2.2 전용 마이크로컨트롤러의 구조

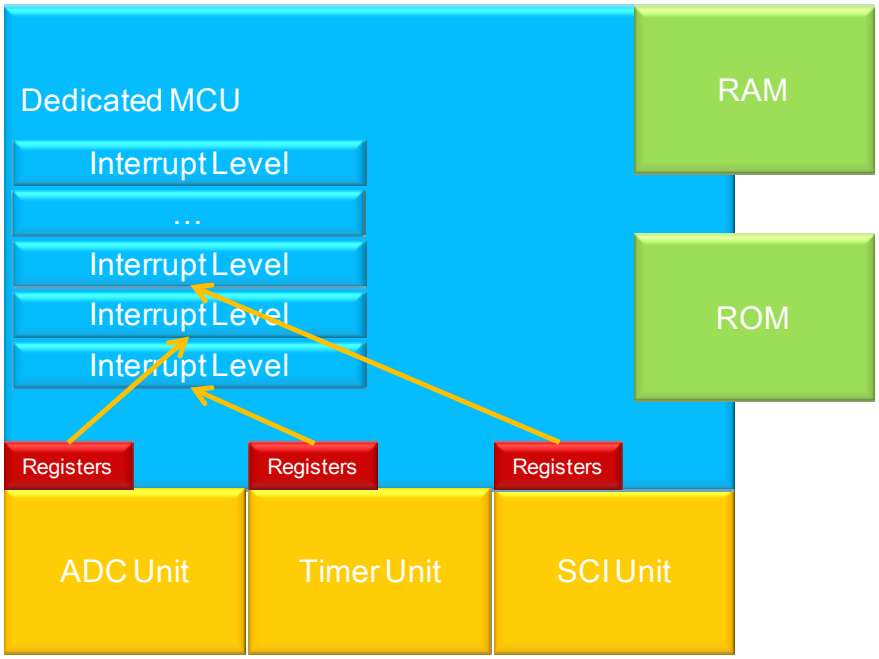

FIGURE 2-2 전용 마이크로컨트롤러의 구조

[핵심]

- 제어레지스터가 내부버스 번지에 할당되고 매우 빠른 속도로 접근 가능하게 된다.
- 최적의 입출력 처리를 위하여 매우 많은 인터럽트 레벨을 가지고 있다.
- 주변장치의 각종 이벤트를 개별적인 인터럽트에 할당할 수 있어, 최적의 입출력 처리가 가능하다.
- 전용 마이크로컨트롤러는 매우 효율적으로 프로그래밍 할 수 있으며, 안정적인 응용시스템 개발이 가능하다.

2.3 운영체계

- 운영체계에서 제공해야 하는 서비스
  - 일정 시간 (주기) 마다 태스크를 수행
  - 특정 이벤트 발생시 태스크를 수행

2.3.1 협력적인 운영체계

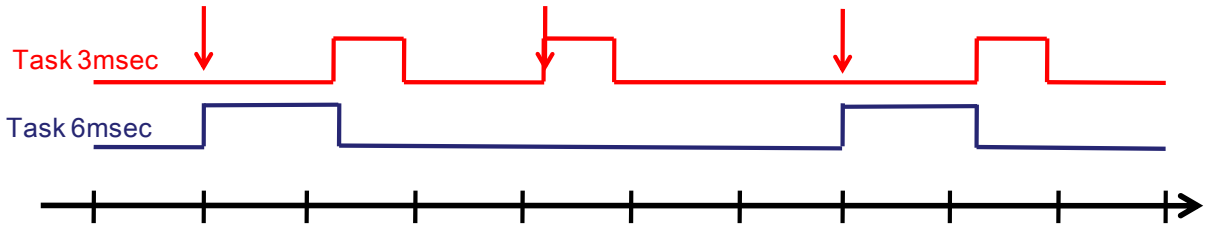

FIGURE 2-3 협력적인 운영체계의 예

2.3.2 선점형 운영체계

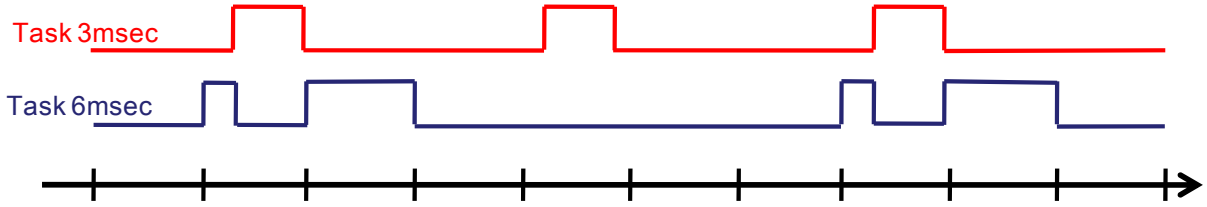

FIGURE 2-4 선점형 운영체계의 예

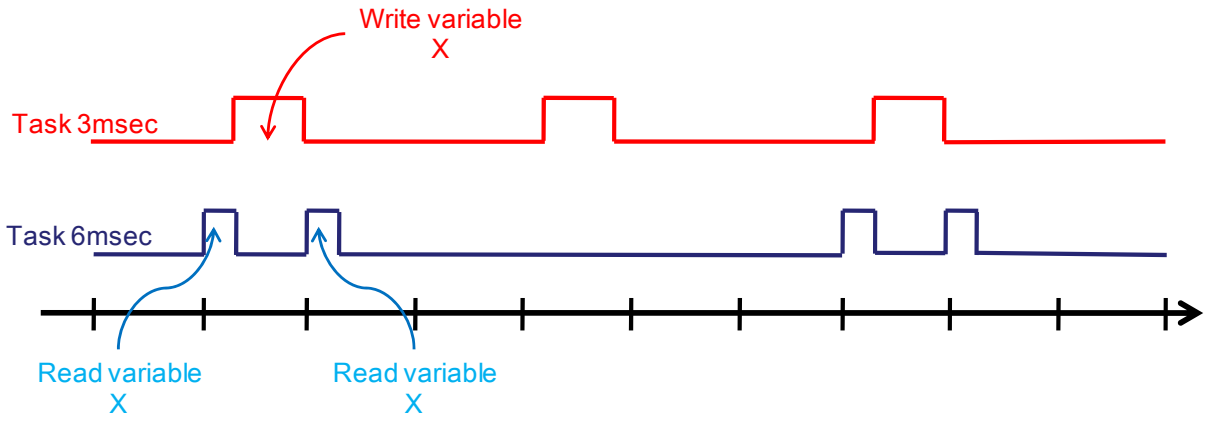

FIGURE 2-5 선점형 운영체계에서 자료공유의 문제

2.3.3 협력적인 운영체계와 선점형 운영체계의 비교

[핵심]

- 협력적인 운영체계는 다른 태스크를 중단시키는 동작을 하지 못한다.
  - 장점: 다른 태스크가 자료(전역변수)의 예상치 못한 변경을 막아준다
  - 단점: 태스크의 수행시간이 부정확해질 수 있고, 그러므로 자료획득 시스템이나, 피드백 제어시스템에 사용하기가 어렵다.
  - 단점: CPU 부하를 최대한 높이기 어렵다
- 선점형 운영체계
  - 장점: 이론적으로 CPU 부하를 100% 까지 높일 수 있다. 태스크의 수행시간을 정확하게 유지시켜 줄 수 있다.
  - 선점형 운영체계를 사용할 경우 충분한 스택 메모리 공간과 인터럽트 레벨이 필요하다.

2.3.4 정적인 주기형 스케쥴링 방법

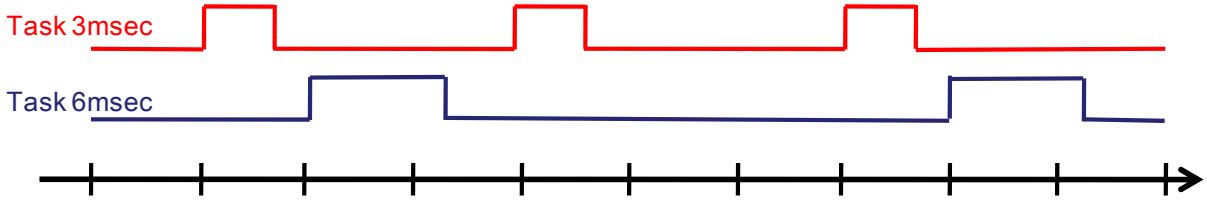

FIGURE 2-6 정적인 주기형 스케쥴링 방법의 예

2.4 구조의 동적 측면들

2.4.1 운영체계와 마이크로컨트롤러의 조화

- 이론적으로는 4 가지의 조합이 가능
  - MCU 종류 2 가지 X OS 종류 2 가지
- 일반적으로
  - 일반적인 CPU 를 사용한 MCU 는 협력적인 OS 를 채택
  - 전용 MCU 는 선점형 OS 를 활용

[핵심]

- 일반 CPU 에 근거한 마이크로컨트롤러는 협조적인 운영체계를 사용하는 것이 좋다.
- 정확한 샘플링 타임을 지켜야 하는 시스템에는 선점형 운영체계를 채택한 전용 마이크로컨트롤러를 사용하자.
- CPU 자원을 최대한 사용하기 위해서는 선점형 운영체계를 채택하는 것이 바람직하다.
- 선점형 운영체계를 사용한 시스템은 내성이 좋다.

<Power Programming>

소프트웨어 구조와 모듈 설계

Chapter 03
객체지향 프로그래밍

Version	Document maturity (draft / valid)	Date of Issue (200x-MM-DD)	Author/Owner	Check/Release	Description
1.0		2008-05-24	W. Lee		Initial version
1.1		2012-04-30	W. Lee		Separate from ADPro_A

3 객체지향 프로그래밍

(질문) 객체지향 프로그래밍은 꼭 객체지향 프로그램 언어를 사용해야만 한다?
: 아니다. 비 객체지향언어인 C 언어를 사용해서 객체지향 프로그래밍을 할 수 있다.
: 물론, 객체지향의 고급 기능들은 지원할 수 없지만 기본 기능은 구현할 수 있다.

3.1 서론

- 객체지향프로그래밍은 1980 년대에 등장
  - C++, Java, SmallTalk 등의 언어가 탄생함
  - 객체지향 설계 방법이 등장하게됨 ➔ UML 등
- 임베디드 시스템에서는 객체지향언어보다는 기존의 Assembler 와 C 언어가 아직까지 대세
  - 수행속도, 메모리 사용의 문제
  - C 언어를 사용한 객체지향프로그래밍은 가능
- 기본적인 객체지향의 개념: Microsoft Press 에서 소개
  - ADT(Abstract Data Types)의 개념 소개
  - 간단한 C 언어의 구문: 구조체를 활용한 자료의 추상화와 함수 호출을 사용한 자료 접근
  - 소프트웨어를 객체로 나누어 프로그래밍 하는 방법
- 오늘날의 객체지향 개념
  - 상속, 다형성 등등 다양한 개념이 기본 객체지향개념에 추가됨
  - 객체지향 프로그래밍 언어를 사용하더라도 객체지향적으로 프로그래밍 되는 것은 아니다.
- 기본적인 객체지향프로그래밍의 개념을 정확히 아는 것이 중요

3.2 객체지향 프로그래밍이란?

3.2.1 객체지향의 원리

- 객체지향과 관계된 몇몇 정의들

"**An object is anything with a crispy defined boundary**" (Brad J. Cox, Object-oriented Programming: An Evolutionary Approach (1991), Addison-Wesley)

"**Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics**" (Crady Booch, Object-oriented Analysis and Design with Applications, 2nd Edition, Benjamin Cummings, Redwood City)

"**Encapsulation(Information Hiding): A principle, used when developing an overall program structure, that each component of a program should encapsulate or hide a single design decision. The interface to each module is defined in such a way as to reveal as little as possible about its inner workings**", (Peter Coad and Jill Nocola, Object-oriented Programming, Prentice Hall PTR, 1st Edition)

- 객체지향의 중요한 개념
  1. 객체란 실제로 물리적인 객체에 근거하여 정의하여야 한다.
     : 마이크로콘트롤러 시스템에서; 센서, 스위치, 액츄에이터, 필터, 적분기 등등
  2. 객체란 명확한 경계와 인터페이스를 가지고 있어야 한다.
     : 인터페이스의 종류와 그 내용이 명확해야 한다.

3. 외부로 보일 필요가 없는 내용은 숨겨져야 한다.
   : 자료 은닉의 기본 개념

- 객체의 예: 사무실 의자
  - 사용자 인터페이스: 등받이, 받침, 팔걸이
  - 시스템 인터페이스: 5 개의 바퀴
  - 설정 인터페이스: 높이 조절 레버
  - 숨겨진 속성들: 나사들, 조절 메커니즘, 충격완충 장치, 받침의 소재 등등

[핵심] 객체지향 소프트웨어의 기본적인 특징

- 소프트웨어를 실생활의 객체로 구조화 하는 것이다. 추상화된 객체를 활용하면 더욱 좋은 객체지향 설계를 할 수 있다.
- 자료와 함수(혹은 메소드)는 객체의 두 부분이고, 이것들은 객체 안으로 숨겨져야 한다.
- 정보의 은닉성은 철저히 지켜야 한다.
  - 외부에서 접근해야 하는 것은 인터페이스를 갖도록 하고
  - 그럴 필요가 없는 자료들은 철저히 숨겨져야 한다.

3.2.2 객체지향 프로그래밍 언어

- 객체지향의 첫번째 사례는 C 언어를 사용한 ADT 이다.
  - 이것은 객체지향 프로그래밍 언어가 시장에 나오기 전에 C 언어를 활용한 객체지향의 좋은 사례이다.
- 최근에 나온 객체지향 언어의 추가적인 기능은 객체지향프로그래밍을 보조적으로 도와주는 방법을 제공하는 것이지, 객체지향 프로그래밍 자체는 아니다.
  - 다형성, 상속, 오버로딩 등등
  - 이러한 보조적인 기능들은 객체지향프로그래밍 자체를 혼란스럽게 할 수 있다.
- 객체지향 언어인 C++을 사용하여 비객체지향적으로 프로그래밍 하는 사례들
  - 하나의 큰 클래스를 만들고 변수를 모두 public 으로 선언하는 것
  - 프랜드 클래스를 여러 개 만들어 변수를 자유롭게 접근하도록 하는 것
  -

[핵심]

- 객체지향 언어를 사용했다고 해서 객체지향적인 소프트웨어가 되는 것은 아니다.
- 많은 객체지향 특징은 객체지향 기본 원리에 나중에 덧붙여 진 것이다.
- 객체지향 설계와 원칙을 고수함으로써 객체지향 소프트웨어를 만들 수 있다.

3.3 절차지향 소프트웨어와 객체지향 소프트웨어

- 소프트웨어는 자료와 자료를 변환하는 함수의 결합
- 절차지향 설계법:
  - 자료변환과정에 집중하여 설계하는 방법
  - 제어흐름을 구성하는 것이 소프트웨어 설계
  - 제어흐름 자체가 소프트웨어의 구조가 됨
- 객체지향 설계법: 변환되는 자료에 집중하여 설계하는 방법
  - 자료흐름을 구성하는 것이 중요
  - 자료흐름 중심으로 소프트웨어 구조 생성

[전동식 안전벨트 제어기의 예]

- 개념적인 이야기 보다는 예를 통한 설명 방식으로 진행
- 소프트웨어 동작의 이해를 돕기 위하여 자료흐름도 제시
- 구조
  - Device Driver: Adc_XXX, Can_XXX, Dio_XXX, Pwm_XXX 등등
  - 응용프로그램: 디바이스 드라이버를 사용하여 값을 변환하고 제어하고 진단하는 동작 단위로 구분하여 구성, 총 7 개의 동작
- 자료전달 방법
  - Device Driver 를 사용하여 HW 에 접근하는 경우 직접적으로 함수를 호출하는 방식, 파라미터와 반환값을 사용하여 전달
  - 응용소프트웨어 영역의 경우 각 동작 사이에 자료 객체를 두고 이를 통하여 전달
- 다양한 수행주기
  - 요구사항에 따라 각 동작이 다른 수행주기, 혹은 수행조건을 갖게됨
  - 기본 수행주기: 10msec, 대부분의 동작이 이 주기에 따름
  - 통신 수행주기: 5msec, 외부시스템과 연계 관계를 고려한 네트워크 설계 요구사항에 따름
  - 신호처리 주기: 1msec, 입력신호의 필터링을 위하여 기본 수행기기보다 10 배 빠르게

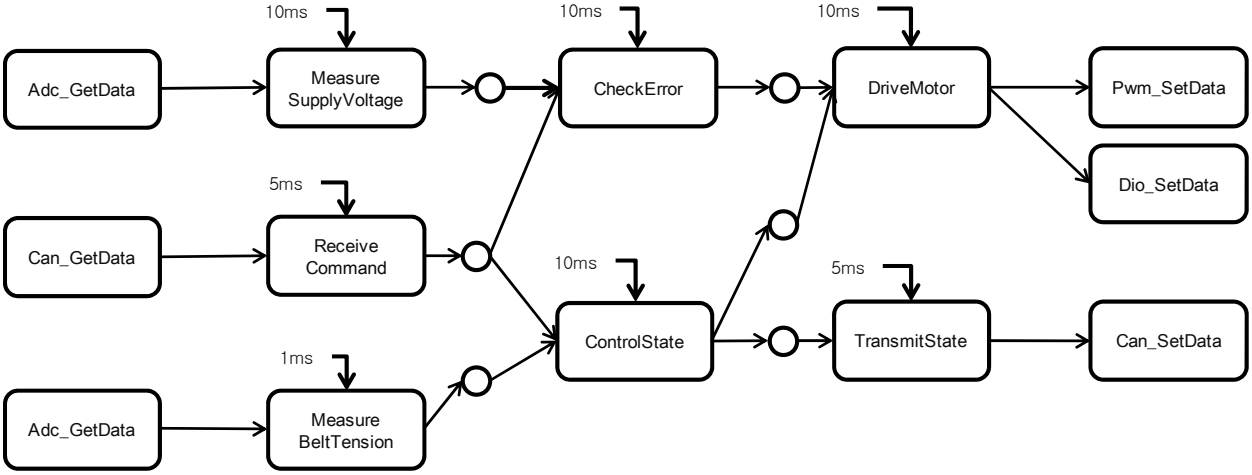

FIGURE 3-1 자료흐름도의 예

3.3.1 제어흐름 중심의 접근법

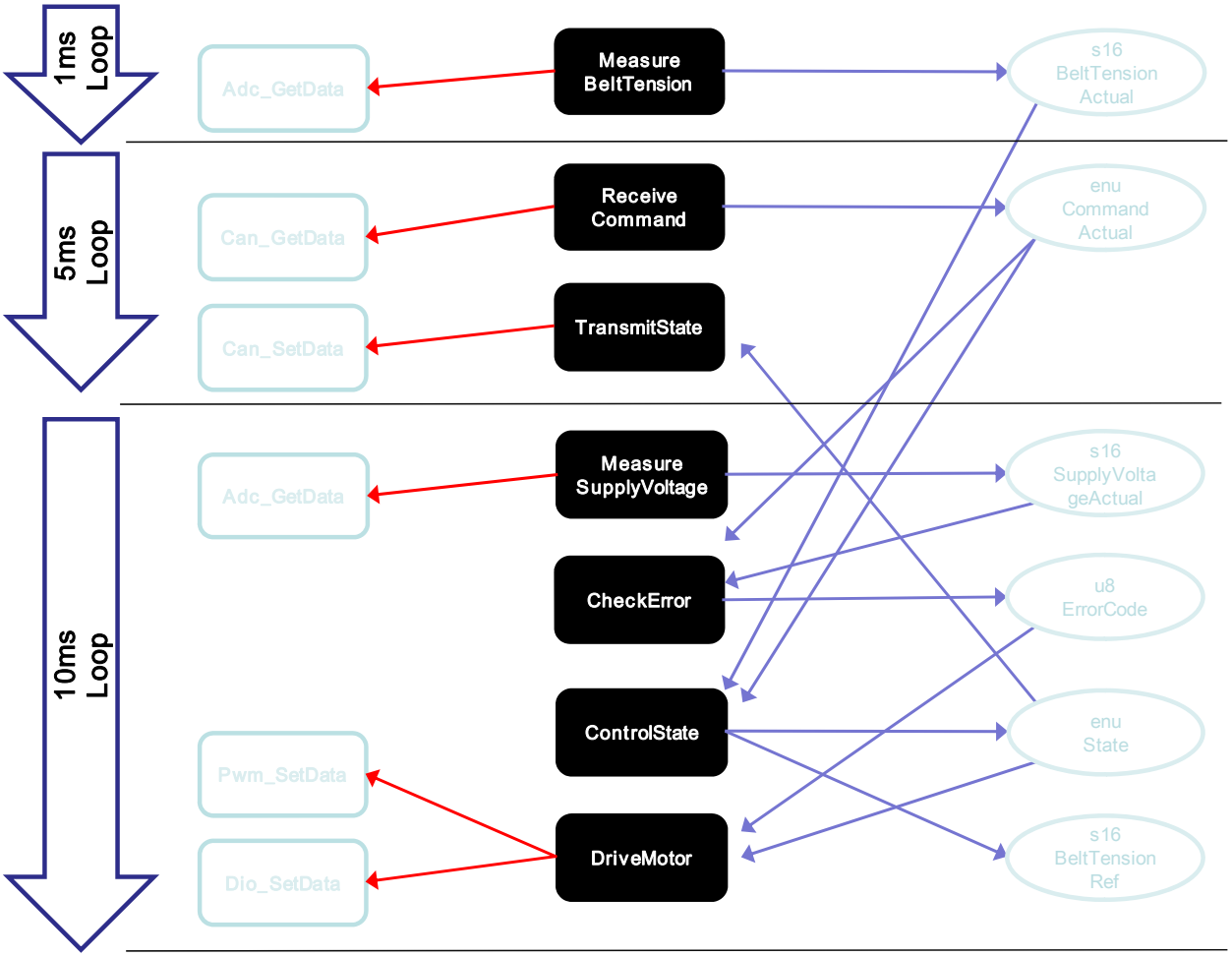

FIGURE 3-2 절차지향 프로그래밍의 제어흐름도

[핵심]

- 제어흐름, 프로세스 중심의 개발법은 원하는 자료를 얻기 위하여 특정 행위를 하는 동작에 집중하여 개발하는 것이다.
- 제어흐름(프로세스)는 프로그램의 기본 구조가 되며 자료흐름을 결정하게 된다.
- 자료는 프로세스의 부산물? 상대적으로 자료에는 관심을 적게 기울이게 된다. 그러므로 자료들은 대부분 전역변수 형태로 만들게 된다. ➔ 전역변수 사용으로 발생되는 많은 문제점 야기
- 자료흐름을 파악하기 매우 어려워지며, 제어흐름을 조금 수정하는 것이 자료흐름을 흩으러 놓을 수 있다.
- 단일 수행 주기일 경우 자료의 흐름을 어느 정도 파악할 수 있지만
  - 여러 수행주기를 갖을 경우 제어흐름만을 가지고서 자료의 흐름을 전혀 예상할 수 없다.
  - 자료흐름을 파악해 보기 위하여 제어흐름 옆에 함께 그려보았으나, 상당히 어려움.
- 함수들이 객체로 그룹화 하기가 어려우며, 여러 모듈에 산재하게 된다.

3.3.2 자료흐름 중심의 접근법

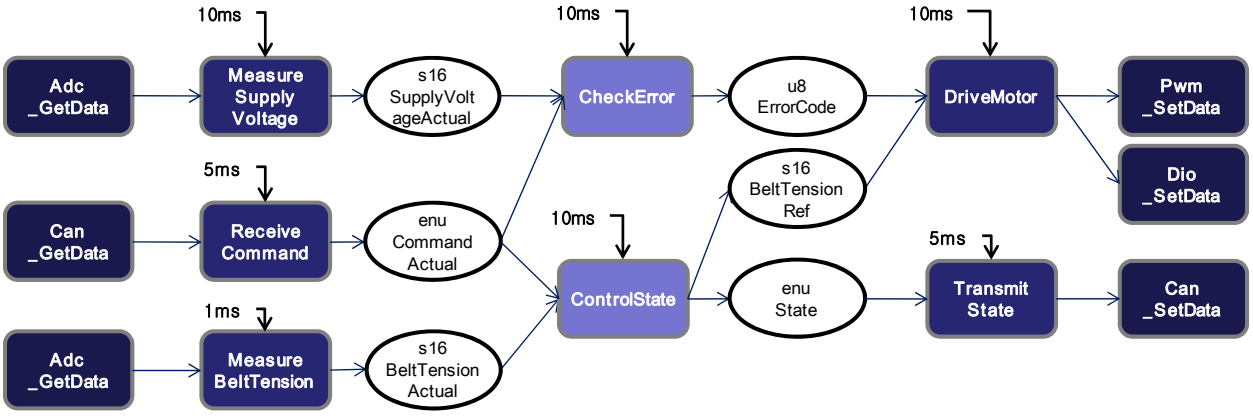

FIGURE 3-3 자료흐름도의 예(상세)

[핵심]

- 자료흐름 중심의 개발법은 원하는 자료를 중심으로 생각하고, 입력자료를 활용하여 중간 자료들을 생성하고 재 배열하여 최종 자료를 만드는 방식
- 자료와 그 자료를 만드는 프로세스를 하나로 묶어 객체로 만들어 구성할 수 있다.
- 자료흐름 중심의 개발법으로 객체지향 프로그래밍 하는 것이 용이하다.

3.3.3 객체 만들기

- 객체가 가져야 하는 물리세계와의 연관성, 명확한 경계, 불필요한 자료의 은닉을 고려
- 자료흐름도에 근거하여 자료와 동작을 하나로 묶음
- 객체는 객체를 나타내기에 필요한 모든 것을 담고 있어야 한다.

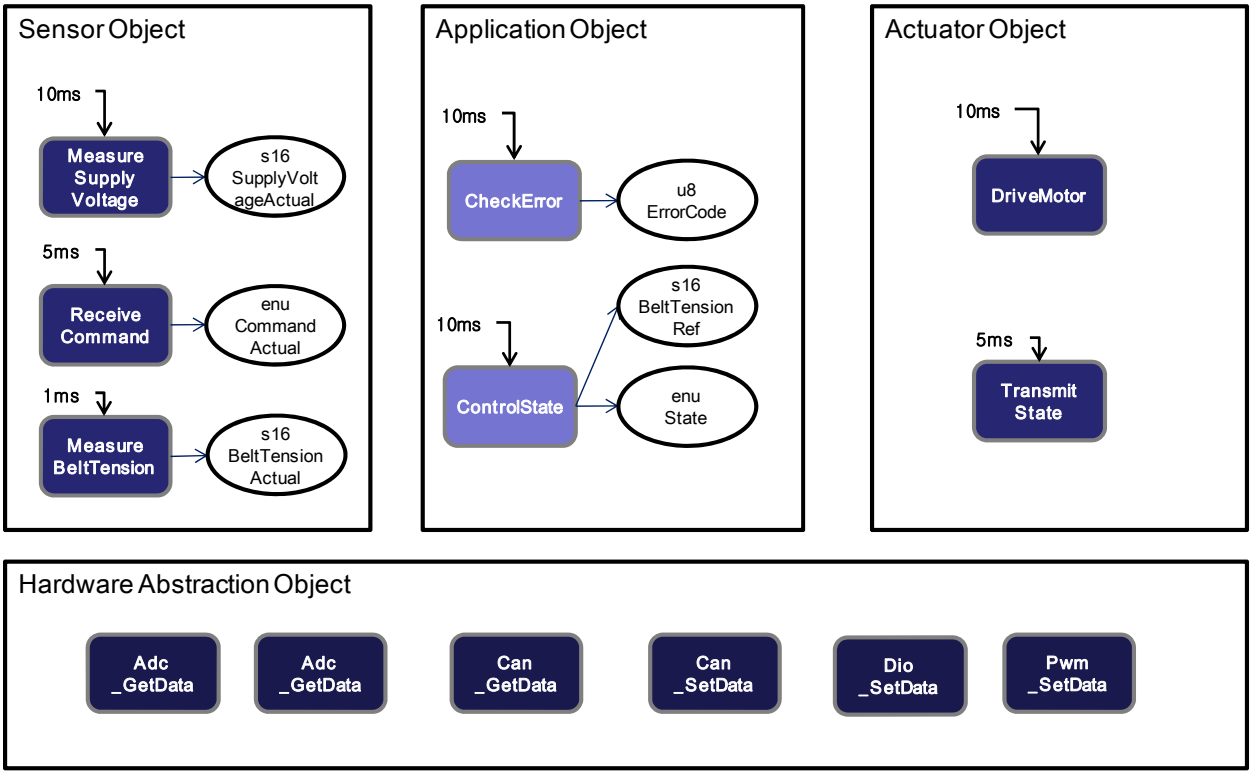

FIGURE 3-4 객체구성

3.3.4 객체와 계층적 구조

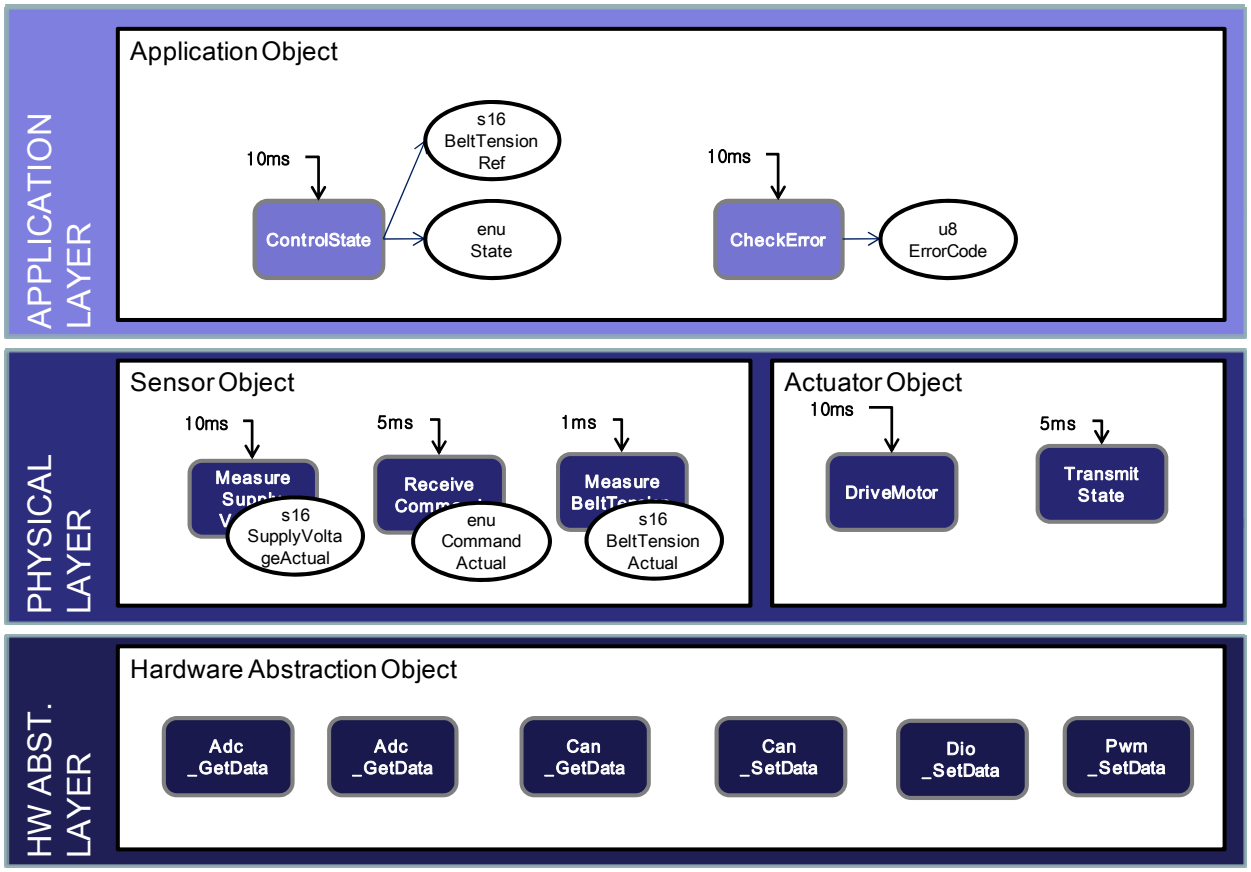

FIGURE 3-5 객체와 계층적 구조

[핵심]

- 객체를 만드는 것이 계층구조를 깨뜨리는 것은 아니다. 오히려 객체들을 모아서 계층을 만들 수 있다.
- 객체 접근을 제한하는 규칙을 사용할 수 있다. 이 규칙을 지킴으로 계층을 바이패스하는 등의 오류를 피할 수 있다.

<Power Programming>

소프트웨어 구조와 모듈 설계

Chapter 04
자료, 자료흐름, 그리고 인터페이스

Version	Document maturity (draft / valid)	Date of Issue (200x-MM-DD)	Author/Owner	Check/Release	Description
1.0		2008-05-24	W. Lee		Initial version
1.1		2012-04-30	W. Lee		Separate from ADPro_A

4 자료, 자료흐름, 그리고 인터페이스

4.1 서론

- 적합한 자료를 정의(변수와 인터페이스)하는 것은 중요한 사항임에도 불구하고 소프트웨어 설계에서 많이 간과되는 영역
- 적합한 자료를 정의하는 것은 비기능적 요구사항을 만족시킬 때 중요
- 아울러 시스템의 성능과도 직접적인 연관: RAM 의 사용, 수행시간 등등
- 많은 프로그래머들이 자료흐름(data flow)에 대하여 크게 인식하지 않은 상태에서 SW 개발
  - 결과적으로 프로세스 중심, 혹은 제어흐름 중심적으로 프로그래밍
- 자료흐름 중심적인 프로그래밍 객체지향적 프로그래밍에 유리

4.2 자료의 정의

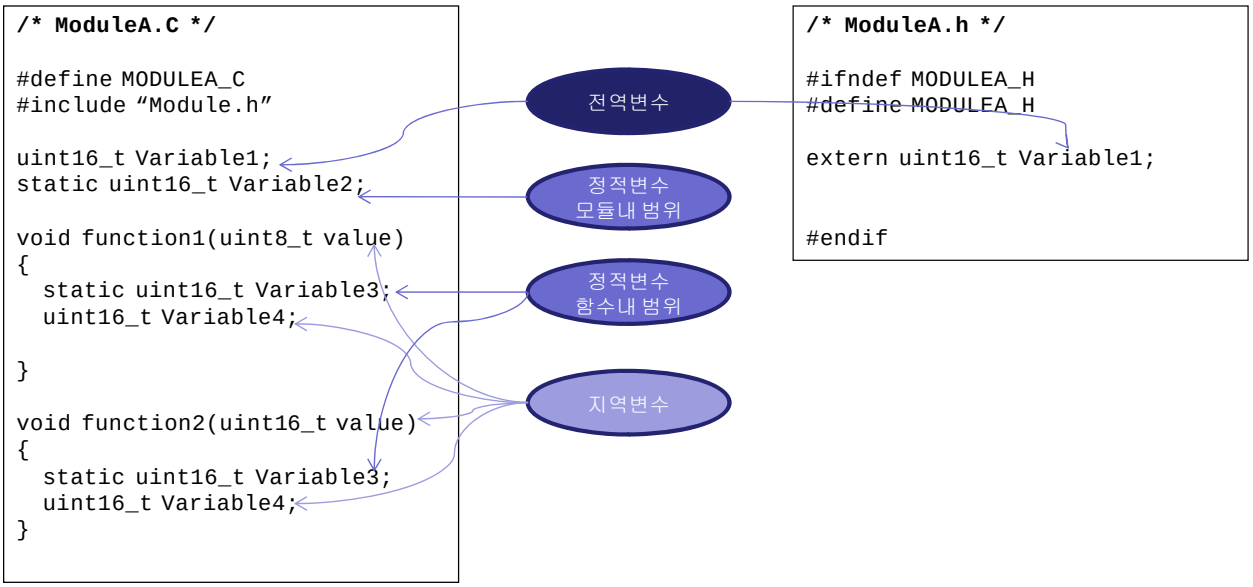

FIGURE 4-1 자료정의의 예

4.2.1 지역변수

- 첫째: 함수의 파라메터
- 둘째: 함수내에 선언된 변수
- 함수외부와 전혀 연결이 없음 ➔ NO LINKAGE
- 함수내 범위를 갖기 때문에 다른 함수에서 같은 이름의 변수를 사용할 수도 있다.
  - 그러나 같은 이름을 사용하는 것은 오류를 범할 가능성을 높임
  - 가능하면 다른 이름을 사용하는 것이 바람직 ➔ MISRA-C
- "automatic" 변수라 불리기도 함
  - 일단 CPU 의 Register 에 변수를 할당하고자 함
  - Register 의 공간이 부족하면 Stack 에 변수를 선언함
- 지역변수의 장점
  - RAM 을 사용하지 않음
  - Register 를 사용하는 경우 수행속도가 빨라짐

4.2.2 함수내 범위를 갖는 정적변수

- 함수내에 선언된 정적변수는 함수내 범위를 갖음
- 정적으로 변수값을 유지해야 하므로 RAM 을 사용
  - Register 를 사용할 때와 같은 속도의 개선을 기대할 수 없음
- 변수의 선언과 동시에 초기화 가능
  - 초기화 값은 시스템의 부팅시 롬에 저장된 초기값을 램의 정적변수 영역으로 복사
- 정적변수의 초기화는 명시적으로 해주는 것이 바람직
- 다른 모듈에서 같은 이름의 변수를 선언할 수 있음
  - 지역변수에서 언급한 바와 같이 가능한한 다른 이름을 지어서 사용하는 것이 바람직

4.2.3 모듈내 범위를 갖는 정적변수

- 함수외부에 선언된 정적변수는 모듈(파일) 범위를 갖음 ➔ INTERNAL LINKAGE
- 다른 사항은 함수내 범위를 갖는 정적변수와 같음

4.2.4 전역변수

- 함수외부에 선언된 모든 변수(static 없이)는 전역변수로 다른 모듈에서 자유롭게 읽고 쓸 수 있다.
- 다른 모듈에서 이 변수에 접근하고자 한다면 “extern” 사용하여 변수를 선언하여야 한다.

[핵심]

- 지역변수는 함수(블록) 내에서 접근 가능하며 함수 내에서만 값이 유효하다. 일반적으로 지역변수는 레지스터 혹은 스택에 위치한다.
- 정적변수는 메모리에 지속적으로 존재하게 된다. 함수내에 선언한 정적변수는 함수내에서 접근가능하며, 함수외부에 선언한 정적변수는 모듈내의 어느 곳에서든 접근 가능하다.
- 전역변수는 정적변수와 마찬가지로 메모리에 지속적으로 존재하게 된다. 모듈내에서 어느 곳에서든 접근 가능하며 “extern” 선언을 사용하여 다른 모듈에서 접근할 수 있다.

4.2.5 좋은 설계를 위한 자료 정의의 원칙

- 가능한한 자료는 지역변수로 선언한다.
- 자료의 값을 다음 수행 주기에서도 사용해야 한다면 함수내 범위를 갖는 정적변수로 선언한다.
- 모듈내의 여러함수에서 공통으로 접근해야 한다면 모듈내 범위를 갖는 정적변수로 선언한다.
- 전역변수의 사용을 정말로 피해보자. 전역변수를 자주 사용해야 한다면 그것은 잘못된 설계이다.

4.2.6 덧붙이는 말: 시스템에서 초기화는 어떻게 이루어지나?

[지역변수의 초기화]

- 변수의 선언과 초기화를 동시에 할 수도 있고 개별적으로 수행할 수도 있다.
- 지역 변수의 경우 두가지 동작이 동등하다.
  - 실제 선언과 초기화를 동시에 하였을 경우, 컴파일러가 변수를 선언하고 초기화 하는 동작으로 나눠서 수행한다.
- 지역변수를 선언하면 ‘0’으로 초기화 되는 것은 아니다.
  - 지역변수가 Register, 혹은 Stack 에 할당되었을 경우 이전의 수행결과 값이 남아있을 수 있다.
  - 지역변수를 꼭 초기화 하는 습관을 갖도록 하자.

```
/* Local Variable Initialization */

sint16_t variable = 10;
```

```
/* Local Variable Initialization */

sint16_t variable;
variable = 10;
```

FIGURE 4-2 지역변수의 초기화

[정적변수(함수범위, 모듈범위)와 전역변수의 초기화]

- 정적변수와 전역변수 모두 RAM 공간에 할당되어지고 프로그램의 시작부터 종료시까지 그 값이 유지된다.
- 변수를 접근할 수 있는 범위만 다르다.
- 정적변수와 전역변수의 초기화 절차
  1. 링커가 정적변수와 전역변수를 모두 묶어서 RAM 의 한 공간에 할당한다.
  2. 동시에 링커가 각 변수의 초기값 또한 모두 묶어서 ROM 에 할당한다.
  3. 프로그램 시작시 ROM 의 초기값을 RAM 에 순차적으로 복사하여 초기화 한다.
- 초기값이 없는 변수들의 처리
  - '0'으로 초기화 될 것이라는 예상은 금물
  - 현재 많은 컴파일러가 초기값이 없는 정적변수와 지역변수를 기본값, '0',으로 초기화
  - 이러한 기능은 C 언어의 표준이 아님
  - 명백하게 초기화하는 것이 바람직한 습관

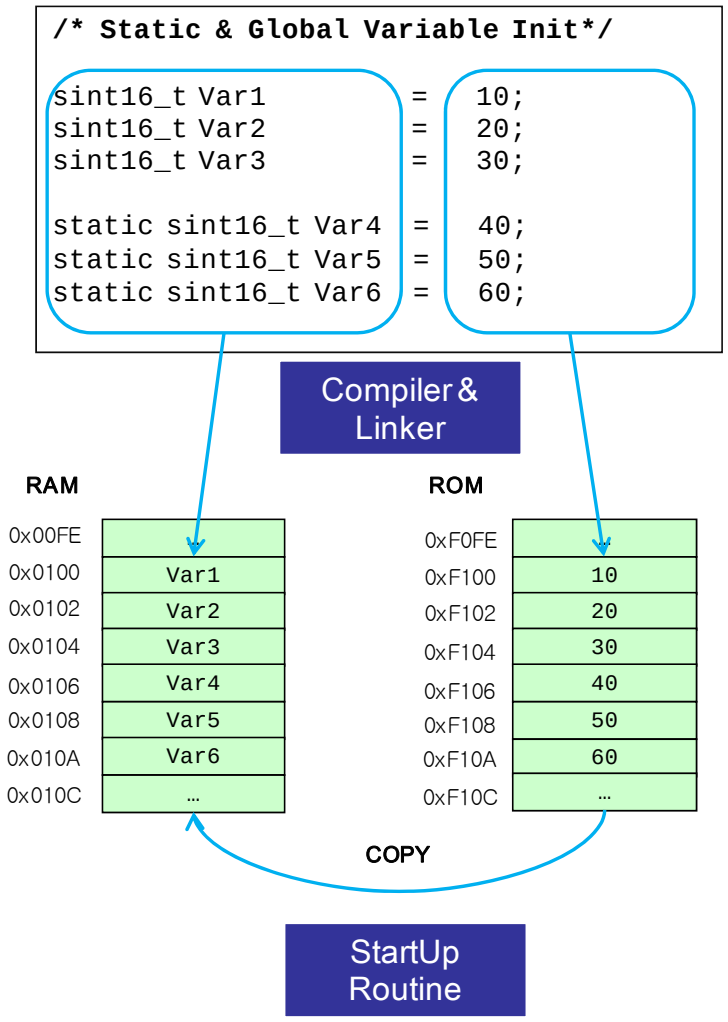

FIGURE 4-3 정적변수와 전역변수의 초기화

[여기서 잠깐: Start-up routine]

- Start-up 루틴은 마이크로컨트롤러가 시작되면서(Reset 직후) 최초로 수행되는 작은 프로그램이다.
- 역할
  - 마이크로컨트롤러의 초기화: 특별한 기능을 가진 레지스터 초기화, 버스 설정, 메모리 watistate 설정 등등
  - 정적변수와 지역변수의 초기화: ROM 에 저장되어 있는 초기값을 각 변수의 RAM 번지로 복사
  - main 프로그램의 호출

[핵심]

- 변수 정의시 초기화는 “start-up” 과정에서 구현되어야 한다. “start-up” 단계에서 ROM 에 있는 변수 초기값의 정보들을 RAM 의 변수 위치에 복사하여 초기화 한다.
- 고 안정 시스템을 만들고자 할 때는 이런 초기화를 사용하지 말고 사용자가 명시적으로 초기화한다.

4.3 “전역변수는 효율적이다”라는 미신을 날려버리자.

```
/* C Source Code */

void LimitSignal()
{

  sw_ay = sw_ay_iir;


  if(sw_ay < -8191)
  {

    sw_ay = -8191;
  }

  else if(sw_ay > 8191)
  {


    sw_ay = 8191;
  }
  else
  {
    …
  }
}
```

```
/* Pseudo Assem Code */




Load Reg ← sw_ay_iir
Load sw_ay ← Reg

Load Reg ← sw_ay
Reg Compare -8191

Load Reg ← -8191
Load sw_ay ← Reg

Load Reg ← sw_ay
Reg Compare 8191


Load Reg ← 8191
Load sw_ay ← Reg
```

FIGURE 4-4 전역변수를 사용한 예

```
/* C Source Code */

void LimitSignal(sint16_t sw_ay_iir)
{
  sint16_t sw_ay;
  sw_ay = sw_ay_iir;

  if(sw_ay < -8191)
  {
    sw_ay = -8191;
  }
  else if(sw_ay > 8191)
  {
    sw_ay = 8191;
  }
  else
  {
    …
  }
}
```

```
/* Pseudo Assem Code */



Load Reg ← Reg

Reg Compare -8191

Load Reg ← -8191

Reg Compare 8191

Load Reg ← 8191
```

FIGURE 4-5 지역변수와 파라미터를 사용한 예

[핵심]

- 전역변수를 사용하는 것이 결코 효율적인 자료교환 법이 아니다.
- 최대한 전역변수 사용을 자제하고 지역변수를 활용한다면, 상당량의 ROM 을 절약함은 물론 수행속도도 높일 수 있다.

4.4 자료흐름의 방향

- 극단적으로 제어시스템의 역할은…
  - 주어진 상황에 맞는 출력값을 계산하는 것
- 이전 단계의 계산값을 활용하여야 함
  - output 객체는 processing 객체에서 계산한 값을 사용
  - processing 객체는 input 객체의 계산 값을 사용해야
- 단방향 자료 흐름을 만들어야 시스템의 해설, 설계, 구현이 용이해짐.

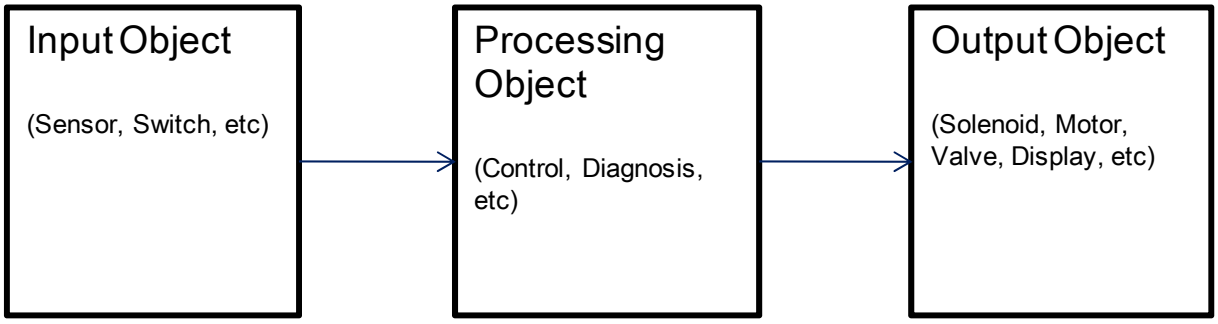

FIGURE 4-6 일반적인 제어시스템의 자료흐름도

4.4.1 객체간 자료전달과 Triggering

- 객체간 자료를 전달하기 위해서는 그 자료를 양쪽간의 어느 한쪽에 할당하고 다른 쪽에서 그 자료에 접근할 수 있는 인터페이스 함수를 만드는 것이 바람직

[인터페이스 함수의 활용]

- 자료전달과 Triggering 을 동시에: Type1
  - 관련 프로시져를 호출(Triggering): 값을 접근하는 함수가 그 값을 사용하는 다른 프로시져를 호출(Triggering)하도록 설계, 자료의 전달과 트리거링 역할을 동시에 수행함
  - 인터페이스 함수의 호출을 통하여 간단한 자료 연결 부를 순차적으로 수행할 수 있음; 간단한 스케쥴링 역할; 프로시져간 동기화
  - 소규모 간단한 프로젝트에서 유리

- 자료전달만: Type2
  - 한 객체에서 프로시져에서 다른 객체의 값을 접근할 경우 단순히 값만 전달하도록 설계
  - 값의 전달과 프로시져의 동작이 분리되어 있으므로 스케쥴러를 사용하여 모든 프로시져를 관리해야함
  - 중규모 이상의 프로젝트에 적합

4.4.2 Pushing 방식 인터페이스

- 이전 단계의 연산 결과를 다음 단계로 전달하는 방식
- 프로시져와 프로시져의 입력값을 묶어서 하나의 객체로 만듬
  - 입력 중심의 사고

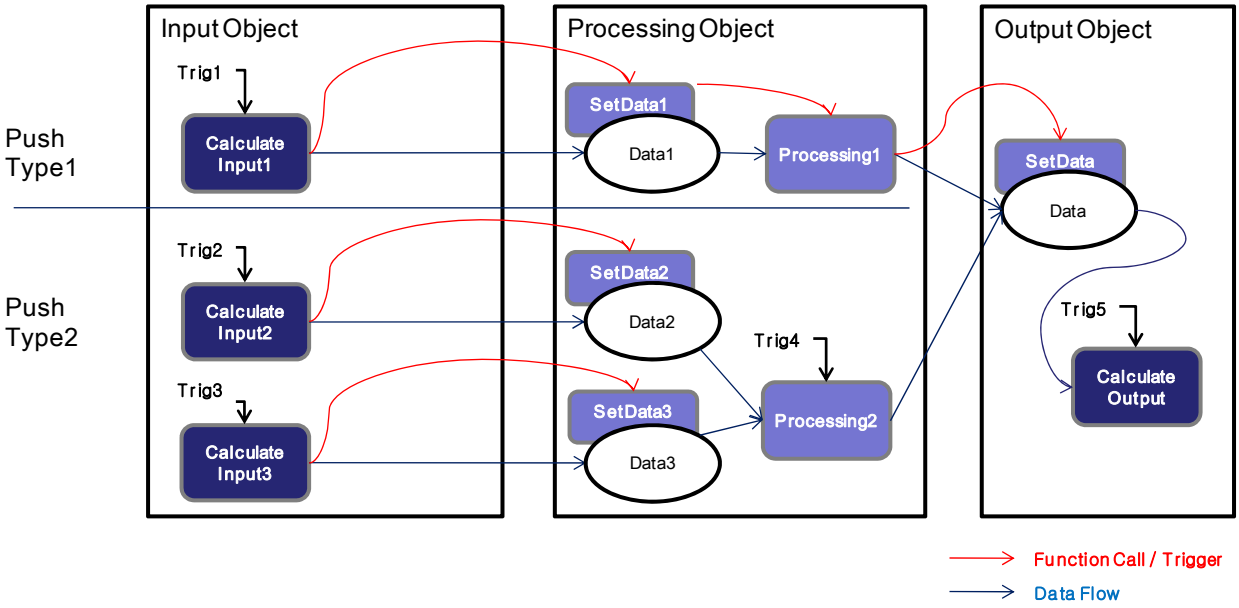

FIGURE 4-7 Pushing 인터페이스의 예

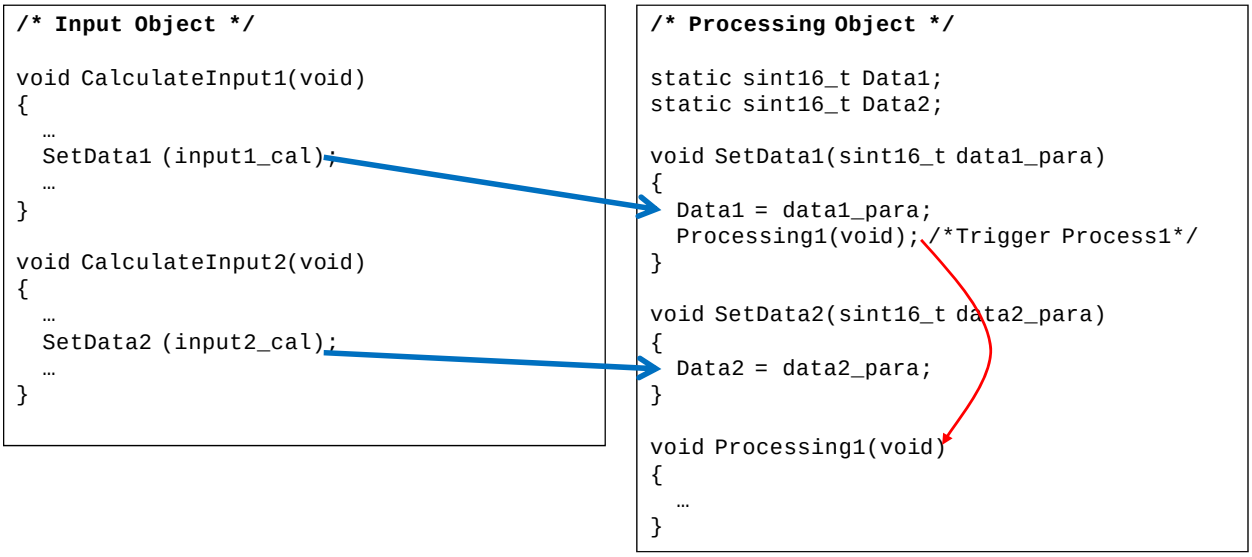

FIGURE 4-8 Pushing 인터페이스의 구현

4.4.3 Pulling 방식 인터페이스

- 이전 단계의 연산결과가 필요할 때 함수를 호출하는 방식
- 프로시져와 출력값을 묶어서 하나의 객체로
  - Pushing 인터페이스를 사용하는 것보다 직관적으로 이해하기 쉬움
- Type1
  - 호출된 함수가 연산을 위해서 다시 이전 단계를 호출할 필요가 있을 경우에는 자동으로 이전단계 호출:
  - 연속으로 연결된 호출관계가 최근의 값을 활용하여 연산할 수 있도록 해줌 ➔ 동기화가 용이
  - 간단한 동작을 연결시켜 동기화 시킬 때 유용
- Type2
  - 모듈간의 인터페이스와 모듈 내에서 수행해야 하는 동작을 구분
  - 모듈 내에서 수행하는 동작은 별도의 트리거를 활용하여 설계 및 관리
  - 복잡한 작업을 수행할 경우 유리

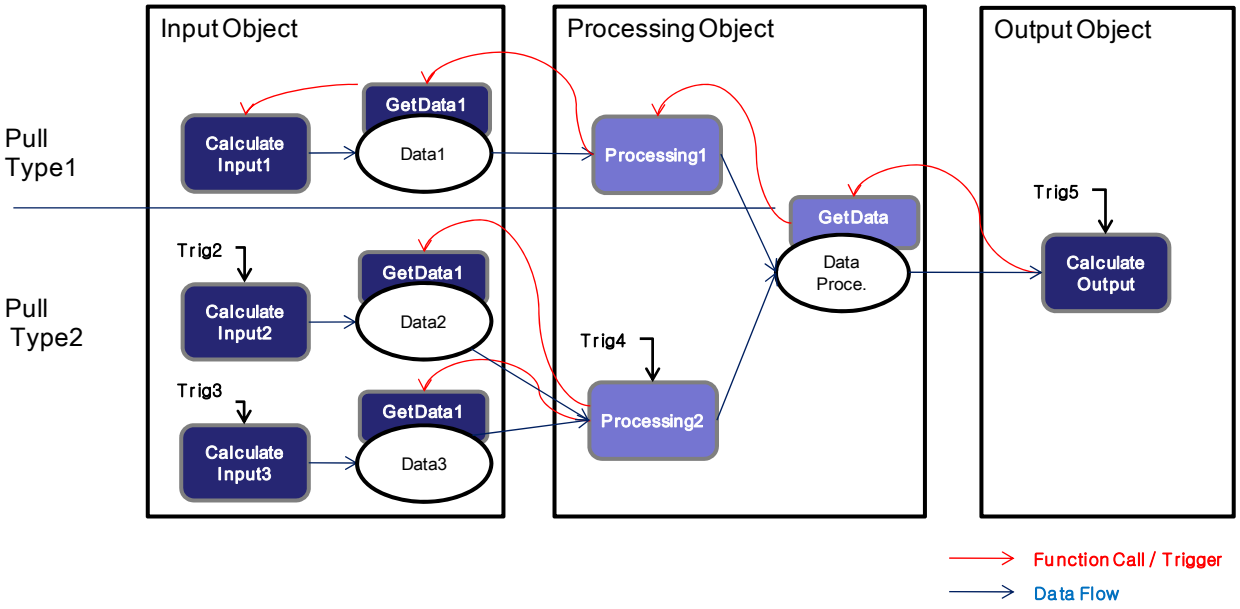

FIGURE 4-9 Pulling 인터페이스

```
/* Input Object */

static sint16_t Data1;
static sint16_t Data2;

void CalculateInput1(void)
{
  …
  Data1 = input1_cal;
  …
}

sint16_t GetData1(void)
{
  CalculateInput1( );
  return(Data1);
}

void CalculateInput2(void)
{
…
  Data2 = input2_cal;
  …
}

sint16_t GetData2(void)
{
  return(Data2);
}
```

```
/* Processing Object */

void Processing1(void)
{
  …
  data1_local = GetData1();
  …
}

void Processing2(void)
{
  …
  data2_local = GetData2();
  …
}
```

FIGURE 4-10 Pulling 인터페이스의 구현

[핵심]

- 자료흐름은 꼭 단방향으로 만들어야 한다. 양방향 자료흐름은 동기화 문제를 만들 수 있다.
- 자료흐름은 “get” 인터페이스를 사용하여 구현한다. 다시 말하여 자료를 저장하는 인터페이스를 사용하지 말고 자료를 읽는 인터페이스를 사용하자.

4.5 객체 인터페이스 디자인

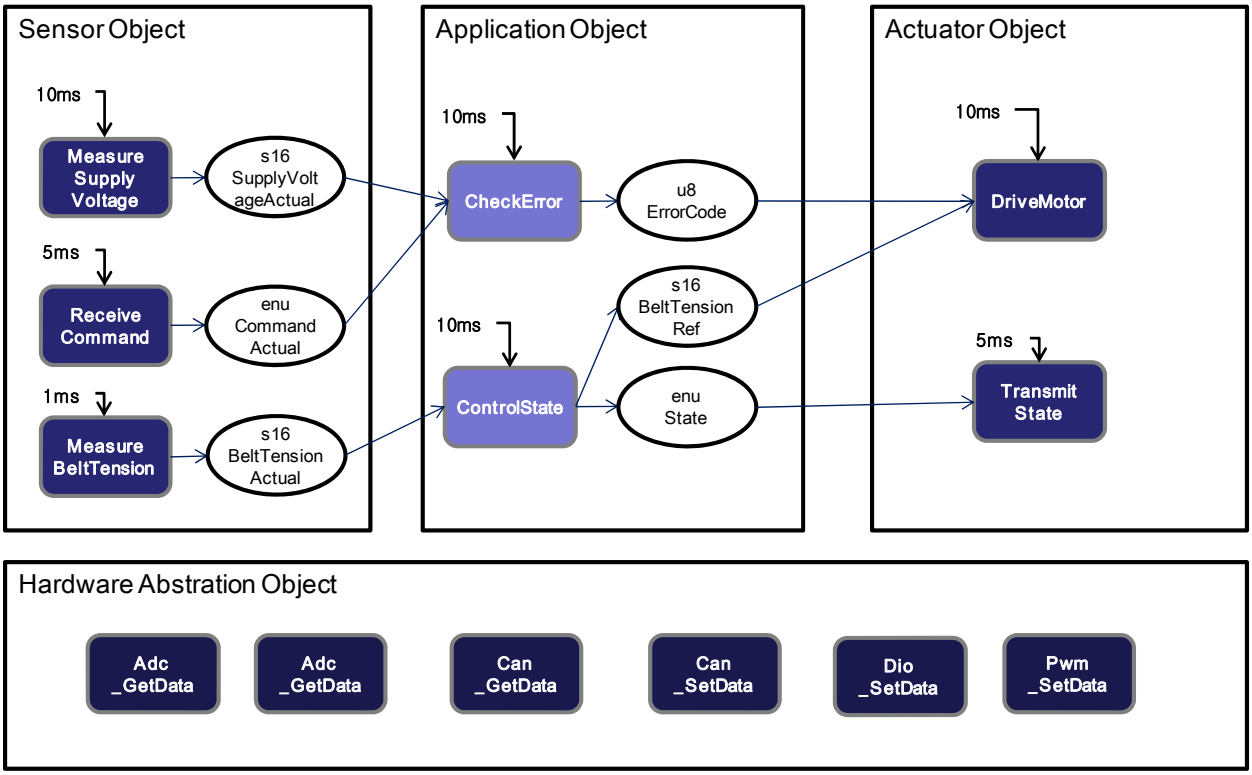

FIGURE 4-11 객체간의 자료 흐름

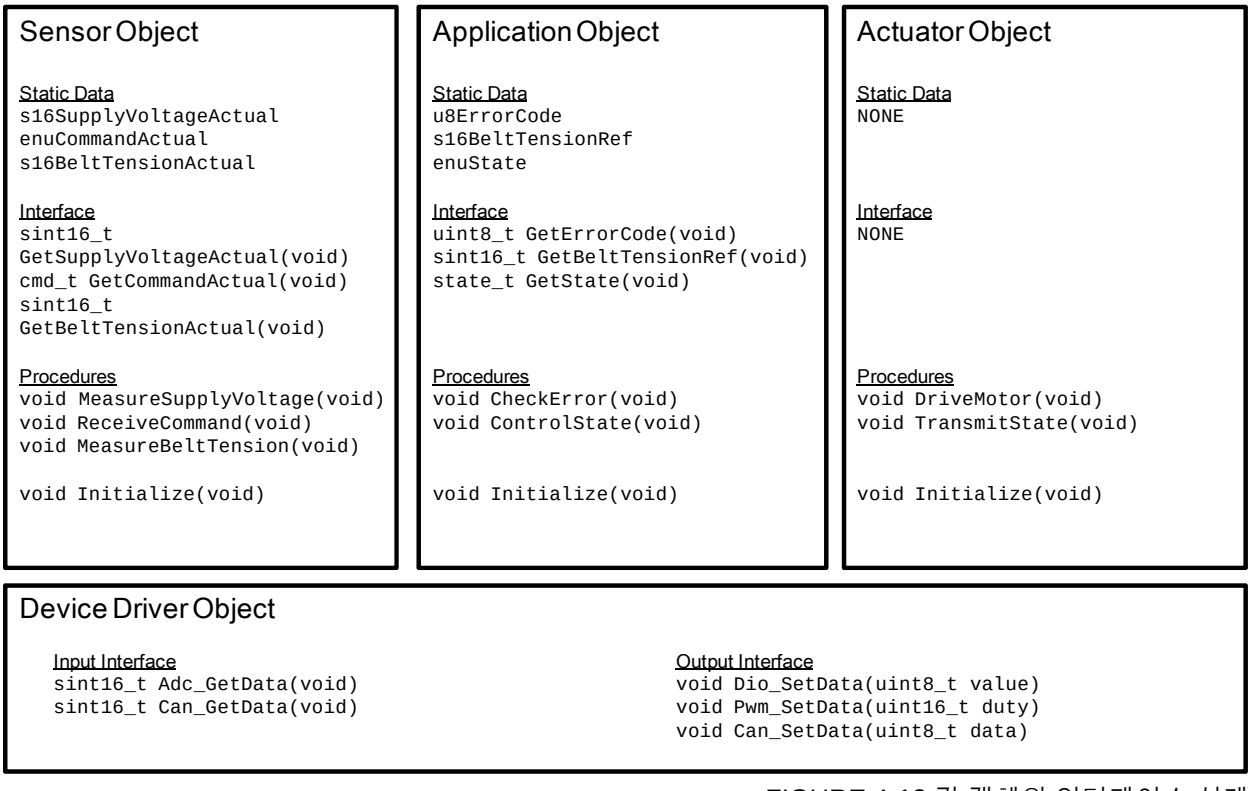

FIGURE 4-12 각 객체의 인터페이스 설계

[핵심]

- 모든 객체는 기본으로 초기화(Initialize) 방법을 가지고 있어야 한다.
- 객체의 인터페이스는 기본 자료형을 사용하여 “get” 스타일로 만들어야 한다.
- 하나의 객체에 10 개 이상의 인터페이스를 만들지 말자.
- “set” 스타일의 인터페이스도 가능하지만 단방향 자료 흐름을 유지할 수 있어야 한다.
- 구조체와 구조체의 포인터도 객체의 인터페이스로 활용할 수 있다.

4.6 객체 스케쥴링

- 인터페이스 함수와 프로시져를 분리하여 설계하는 것이 스케쥴링에 유리하다.
- 스케쥴링을 할 때 Gantt chart 혹은 시퀀스 다이어그램등을 활용한다.
- 본격적인 주기적인 일이 시작되기 전에 항상 초기화를 먼저 한다.

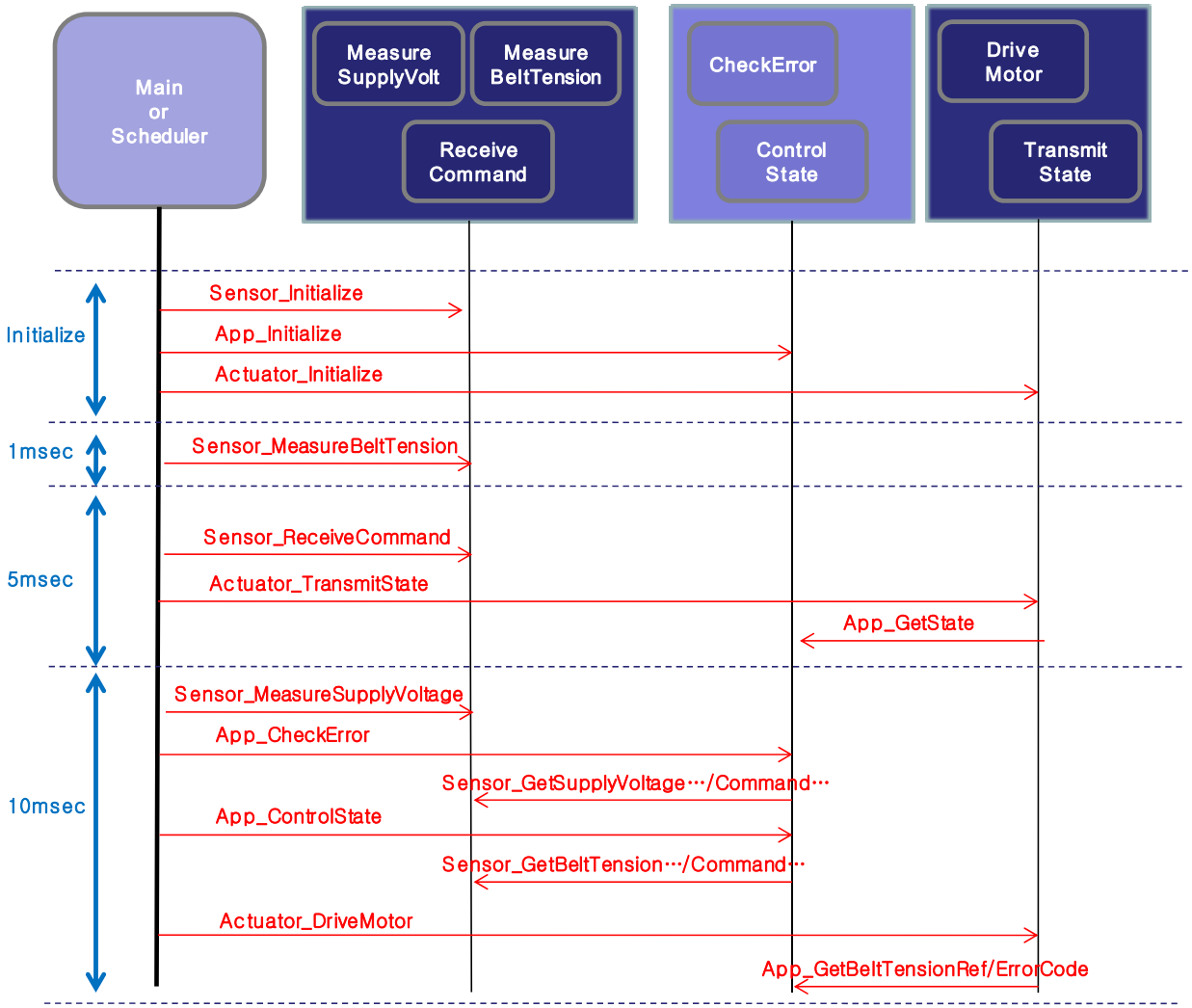

FIGURE 4-13 객체 스케쥴링과 시퀀스 다이어그램

<Power Programming>

소프트웨어 구조와 모듈 설계

Chapter 05
헤더파일의 인클루드 구조

Version	Document maturity (draft / valid)	Date of Issue (200x-MM-DD)	Author/Owner	Check/Release	Description
1.0		2008-05-24	W. Lee		Initial version
1.1		2012-04-30	W. Lee		Separate from ADPro_A

5 헤더파일의 인클루드 구조

- 헤더파일에 대하여서도 많은 경우에 충분히 고려하여 설계하지 않는 경우가 대부분이다.
  - 우연히(?) 만들어 지는 경우가 대부분이고, 이것은 후일에 유지보수에 문제를 야기한다.

5.1 서론

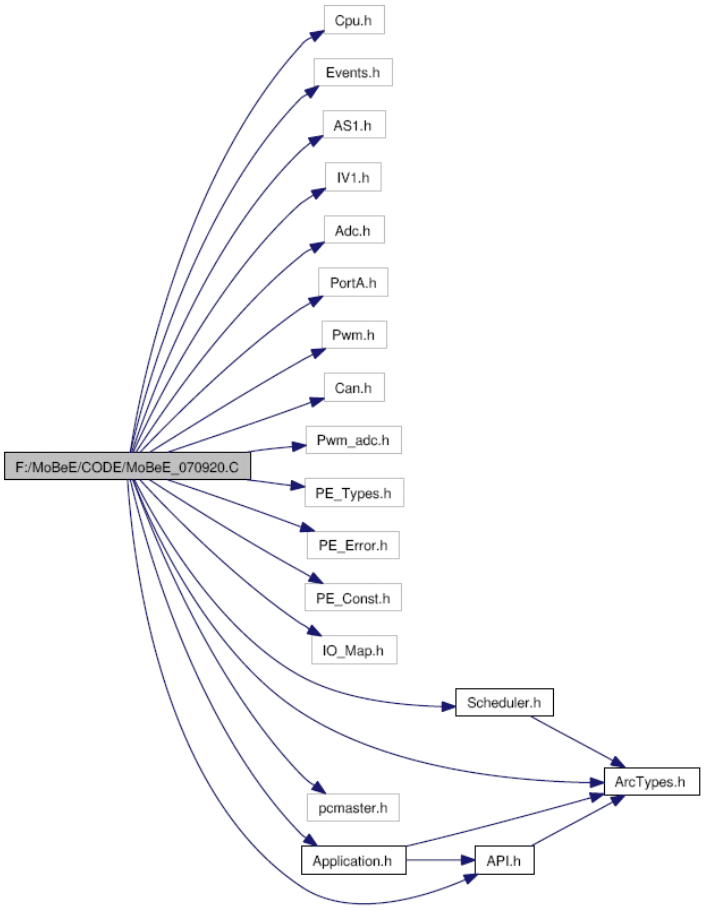

FIGURE 5-1 헤더파일 인클루드 구조의 예

- 상기의 예는 실제 프로젝트의 헤더파일의 구조
  - 어느 정도 정리되어 보기 좋게 구성되어 있음
  - 실제의 경우 좀 더 복잡한 형태로 생성되는 경우가 많음
  - 소프트웨어의 구조가 없는 상태임 ➔ 헤더파일이 지나치게 많이 인클루드 됨.
- 객체지향적 설계에서 헤더파일의 인클루드 구조는 많은 경우에 간과되는 영역
  ➔ 실제 구현을 위하여 바람직한 형태를 언급할 필요가 있음

5.2 좋은 헤더파일 인클루드 구조를 만드는 법칙

1. 헤더파일의 구조를 명확하게 계획하고 설계하여야 한다. 헤더파일과 소스파일의 기본적인 내용으로 템플릿을 구성하고 이것을 반복적으로 사용하도록 한다.
2. 헤더파일의 인클루드 구조를 가능한한 간단하고 논리적으로 구성한다.

3. 가능한한 인클루드 레벨을 낮게 가지고 가도록 한다. 네스팅되거나 상호참조하는 인클루드 형태를 피하자.
4. 헤더파일에는 선언 중심의 내용으로 항상 깨끗하게 유지하자.

5.3 좋은 예

5.3.1 간단한 인클루드 구조

- 모듈헤더 파일은 그 모듈의 외부 모습을 의미한다.
  - 외부에서 참조할 필요가 있는 선언만 모듈헤더 파일에 담도록 한다.
  - 변수와 함수의 정의는 모듈 소스파일 (C 파일)에서 하도록 하자
- 모듈헤더 파일은 모듈에서 사용할 헤더 파일을 모두 인클루드 하도록하고,
  - 모듈소스파일은 이 헤더파일만을 인클루드 하도록 한다.
  - 다른 모듈의 참조가 필요하다면 그 모듈의 헤더를 소스파일에서 인클루드 하도록 한다.
- 모듈의 헤더에서 공통적으로 필요로 하는 헤더파일들는
  - 팀에서 공통적으로 사용할 필요가 있는 부분과
  - 그 프로젝트를 위해서 여러 모듈에서 공통으로 사용하는 부분으로 나눈다.

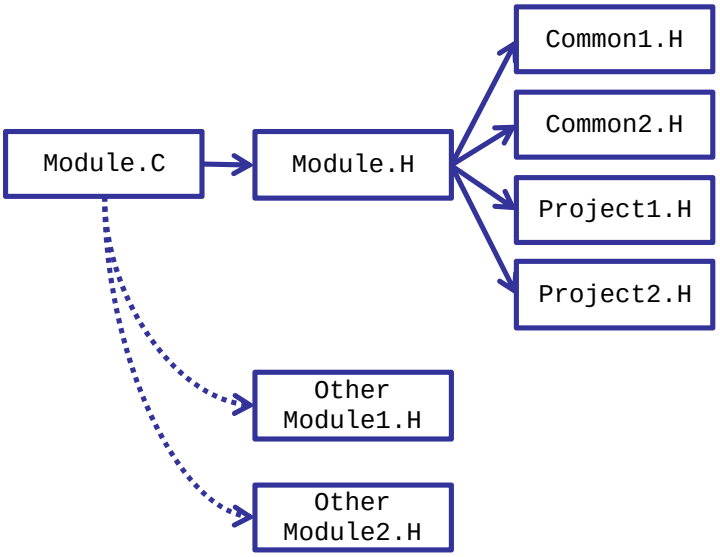

FIGURE 5-2 바람직한 인클루드 구조

```
/* Module.h */

/* Global Include */
#include "GLB_DataTypes.h"          /**< Include Global Data types*/
#include "GLB_Macro.h"              /**< Include Global Macro*/
#include "GLB_FuncLib.h"            /**< Include Function Library*/
#include "GLB_CLib.h"               /**< Include C Library*/

/* Project Specific Include */
#include "XXX_DataTypes.h"
#include "XXX_Macro.h" */
```

FIGURE 5-3 헤더파일의 인클루드 형태

5.3.2 프로젝트에 공통적인 헤더들

5.3.2.1 자료형의 정의를 위한 헤더

- 각 자료형의 크기는 C 언어의 표준이 아님: 컴파일러마다 프로세스마다 다를 수 있음.
- 소프트웨어의 포팅을 돕고, 정수형의 오버플로우를 확인하기 위하여 명시적으로 자료형을 선언하여 사용

```
/* GLB_DataTypes.h */

/**** C type extensions ****/
#define void_t          void
typedef unsigned char   bool_t;
typedef signed char     char8_t;
typedef unsigned char   uint8_t;
typedef char            int8_t;
typedef signed char     sint8_t;
typedef unsigned int    uint16_t;
typedef int             int16_t;
typedef signed int      sint16_t;
typedef unsigned long   uint32_t;
typedef long            int32_t;
typedef signed long     sint32_t;
typedef float           float32_t;
typedef double          float64_t;

/**** C pointer extensions ****/
typedef void*           ptr_t;
typedef unsigned char*  str_t;
```

FIGURE 5-4 자료형 정의를 위한 헤더

5.3.2.2 일반적인 매크로를 위한 헤더

- 참, 거짓과 같은 흔히 사용하는 매크로
- SET, RESET 과 같이 비트 연산을 하는 매크로
- INLINE 과 같은 어셈블러마다 다른 매크로 등

```
/* GLB_Macros.h */

#ifndef TRUE
#define TRUE 1
#endif

#define FALSE 0
#define NULL 0
#define SET |=
#define RESET &=~
#define TEST &

/*Bit Position Mask definition */
#define BYTE_BIT0   0x01
…
#define BYTE_BIT7   0x80

#define WORD_BIT0   0x0001
…
#define WORD_BIT15  0x8000
```

FIGURE 5-5 매크로 정의를 위한 헤더

5.3.2.3 라이브러리를 위한 헤더

- 공통적으로 빈번히 사용되는 함수들
- GNU 에서 Embedded Lib 를 만드는 프로젝트가 별도로 있을 정도.

```
/* GLB_FuncLib.h */

uint8_t GLB_u8GetAbs(sint8_t s8_value);
uint16_t GLB_u16GetAbs(sint16_t s16_value);
uint32_t GLB_u32GetAbs(sint32_t s32_value);

sint16_t GLB_s16SignedShiftRight(sint16_t s16_value, uint8_t u8_shift);
sint32_t GLB_s32SignedShiftRight(sint32_t s16_value, uint8_t u8_shift);
sint16_t GLB_s16SignedShiftLeft(sint16_t s16_value, uint8_t u8_shift);
sint32_t GLB_s32SignedShiftLeft(sint32_t s16_value, uint8_t u8_shift);

…
```

FIGURE 5-6 라이브러리를 위한 헤더

5.3.3 특정 프로젝트 헤더들

- 팀에서 빈번히 사용되지는 않으나
- 개별 프로젝트의 여러 모듈에서 공통적으로 사용되어 지는 것

5.3.3.1 특정 프로젝트의 자료형을 위한 헤더

- DSP 에서 사용하는 Fractional type 등과 같은 경우.
- 여러 프로젝트에서 사용하는 공통적인 자료 구조, 혹은 열거형 선언 등
  - 각 모듈에서 독립적으로 사용되는 것이라면 각 모듈의 헤더에 선언하고 다른 모듈에서 참조

5.3.3.2 특정 객체를 위한 헤더

- 그 모듈을 다른 모듈에서 인터페이스 하고자 할 때 필요한 정보를 모두 담고 있어야 함.
  - external variable, function prototype, data type definition, 등등
- 외부에서 참조할 필요가 없는 것은 숨겨야 한다.

[핵심]

- 헤더파일은 프로젝트 진행 중에 우연히 만들어지는 것이 아니다. 헤더파일은 잘 설계해야 하는 중요한 파일이다.
- 헤더파일의 내용은 명확히 정의 되고, 설계 규칙의 준수 여부를 철저히 검사해야 한다.
- 여러 프로젝트에서 공통적으로 사용할 수 있는 헤더파일들이 있고, 특정 프로젝트에서만 사용하게 되는 헤더파일들이 있다.
- 헤더파일의 인클루드 구조는 단순하고 평면적이어야 한다. 계층적인 인클루드, 의존성이 높은 인클루드는 피해야 한다.

<Power Programming>

소프트웨어 구조와 모듈 설계

Chapter 06
객체지향 프로그래밍을 위한
Source 코드 템플릿

Version	Document maturity (draft / valid)	Date of Issue (200x-MM-DD)	Author/Owner	Check/Release	Description
1.0		2008-05-24	W. Lee		Initial version
1.1		2012-04-30	W. Lee		Separate from ADPro_A

6 객제지향 프로그래밍을 위한 소스코드 템플릿

6.1 서론

- 적절한 종류의 자료를 정의하는 것, 즉 변수 선언 혹은 인터페이스 정의,은 중요한 영역
  - 많은 경우에 구현의 문제로 심도있게 설계되지 않음
- 잘 설계된 자료의 정의 ➔ 유지보수성, 이식성, 시험성 등등을 향상시킴

6.2 모듈 구조의 구성

6.2.1 일반론

- 템플릿의 종류
  - 알고리즘 구현을 위한 소스코드 템플릿
  - 각 소스코드 템플릿과 매칭되는 헤더 템플릿
  - 여러 모듈에서 공통으로 사용되는 헤더 템플릿
- 가시성 종류
  - 전역 vs. 지역

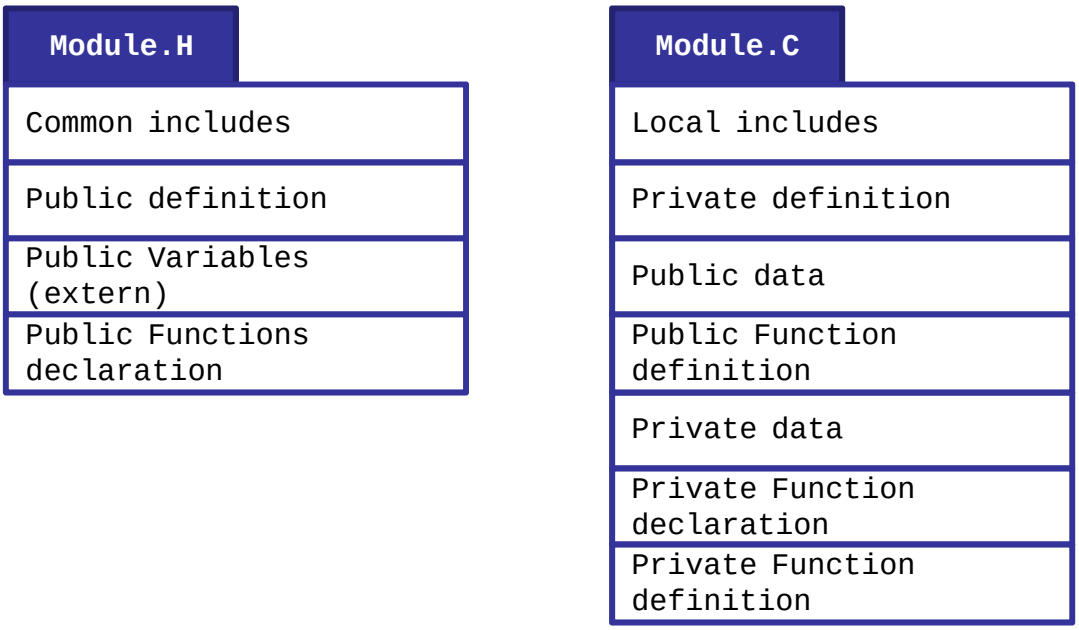

FIGURE 6-1 모듈의 구현

6.2.2 헤더 템플릿

```
/**
 * @defgroup blank blank
 * @{
 */

/******************************************************************************/
/**
 * @file         _blank.h
 *
 * @brief        This is a basic code template
 *
 * @version 1.0  @date Feb/04/2007
 * @author Wootaik Lee <wootaik@changwon.ac.kr>
 * @warning
 * @bug
 * @note
 * <pre>
 * ----------------------------------------------------------------------------
 *              R E V I S I O N   H I S T O R Y
 * ----------------------------------------------------------------------------
 *   Date        Version  Author  Description
 *  ----------   -------  ------  ------------------------------------------
 *  2007-08-03   0.90     Wonbok  - Creation  Prerelease
 *  2008-02-04   1.00     Wootaik - Minor modification and release
 * </pre>
 *
 */
/******************************************************************************/


#ifndef __BLANK_H__
#define __BLANK_H__

/******************************************************************************/
/******************************************************************************/
/*                            MODULES USED                                    */
/******************************************************************************/
/******************************************************************************/

#include "GLB_DataTypes.h"          /**< Include a module of Global Data types*/


/******************************************************************************/
/******************************************************************************/
/*                            DEFINITIONS AND MACROS                          */
/******************************************************************************/
/******************************************************************************/

#define TIMER1                0x01         /**< Bit mask for the task1 */
```

Doxygen: Group 정의

File description

Doxygen 으로 문서화되는 Version 정보, Author, Bug 등의 내용을 포함

Modules include

전처리기를 이용한 Definition 및 Macro 정의

```
/*******************************************************************************/
/*******************************************************************************/
/*                            TYPEDEFS AND STRUCTURES                          */
/*******************************************************************************/
/*******************************************************************************/

/**** C type extensions ****/

/*! \enum enuSchedulerStatus_t
 * enumeration of state of Scheduler @n
 * runngin, elapsed @n
 */

typedef enum
{
  running,                          /**< running state */
  elapsed,                          /**< elapsed state */
}
enuSchedulerStatus_t;

/**** C pointer extensions ****/


/*******************************************************************************/
/*******************************************************************************/
/*                            EXPORTED VARIABLES                               */
/*******************************************************************************/
/*******************************************************************************/

#ifdef __BLANK_C__
  #undefine EXTERN
  #define EXTERN
#else
  #undefine EXTERN
  #define EXTERN extern
#endif

EXTERN enuSchedulerStatus_t Sch_enuSchedulerStatus;

/*******************************************************************************/
/*******************************************************************************/
/*                            EXPORTED FUNCTIONS                               */
/*******************************************************************************/
/*******************************************************************************/

void Sch_InitTimer(void);



#endif

/*******************************************************************************/
/*                                                                             */
/*                              EOF                                            */
/*                                                                             */
/*******************************************************************************/

/** @} */
```

Module 외부에서 사용될 수 있는 type 및 structure 정의

전역 변수 선언

__BLANK_C__가 정의되어 있으면 extern, 정의되어 있지 않으면 공란(Blank)로 처리

전역(Exported) 함수 선언

End of file / End of Doxygen

6.2.3 소스코드 템플릿

```
/**
 * @defgroup blank blank
 * @{
 */

/*****************************************************************************/
/**
 * @file         _blank.c
 *
 * @brief        This is a basic code template
 *
 * @version 1.0  @date Feb/04/2007
 * @author Wootaik Lee <wootaik@changwon.ac.kr>
 * @warning
 * @bug
 * @note
 * <pre>
 * ---------------------------------------------------------------------------
 *               R E V I S I O N   H I S T O R Y
 * ---------------------------------------------------------------------------
 *   Date        Version  Author  Description
 *  ----------   -------  ------  ------------------------------------------
 *  2007-08-03   0.90     Wonbok  - Creation  Prerelease
 *  2008-02-04   1.00     Wootaik  - Minor modification and release
 * </pre>
 *
 */
/*****************************************************************************/


#ifndef __BLANK_C__
#define __BLANK_C__

/*****************************************************************************/
/*****************************************************************************/
/*                           MODULES USED                                    */
/*****************************************************************************/
/*****************************************************************************/

#include "_blank.h"          /**< Include a module of Global Data types*/


/*****************************************************************************/
/*****************************************************************************/
/*                           DEFINITIONS AND MACROS                          */
/*****************************************************************************/
/*****************************************************************************/

#define TIMERMAX    8            /**< The number of Channels used to tasking*/
```

Doxygen: Group 정의

File description

Doxygen 으로 문서화되는 Version 정보, Author, Bug 등의 내용을 포함

Modules include.

전처리기를 이용한 Definition 및 Macro 정의

```
/*****************************************************************************/
/*****************************************************************************/
/*                            TYPEDEFS AND STRUCTURES                        */
/*****************************************************************************/

/**
 * @struct strChannelStatus_t
 * @brief This is a structure managing tasks' status
 */

typedef struct
{
  uint8_t running;      /**< each bit represents a channel (0=Off, 1=running) */
  uint8_t elapsed;      /**< each bit represents a channel (0=off, 1= ready) */
}
strChannelStatus_t;




/*****************************************************************************/
/*****************************************************************************/
/*                            PROTOTYPES OF LOCAL FUNCTIONS                  */
/*****************************************************************************/
/*****************************************************************************/

uint8_t u8ResetTimer(void);


/*****************************************************************************/
/*****************************************************************************/
/*                            STATIC VARIABLES                               */
/*****************************************************************************/
/*****************************************************************************/
#ifdef SIMULATION
  #undef STATIC
  #define STATIC
#else
  #undef STATIC
  #define STATIC static
#endif /* SIMULATION */

STATIC uint32_t u32TimerValueArr[TIMERMAX];    /**< Array of Timer value info */
```

Module 내에서 사용될 type 및 structure 정의

지역(Local) 함수 선언

파일내부(Static)변수 선언

```
/*******************************************************************************/
/*                             EXPORTED FUNCTIONS                              */
/*******************************************************************************/

/*******************************************************************************/
/**
 * @name       Sch_InitTimer
 * @brief      Initialization of the timer for scheduler
 * @pre        none
 * @param      none
 * @return     none
 * @note @n
 *      -Global data: none @n
 *      -Called by     : Application @n
 *      -Description: Initialization of the modulus down counter.
 *                            1 tick is occured every 1 milisecond.
 */
/*******************************************************************************/
void Sch_InitTimer(void)
{
  MCCTL_MCZI = 1;         /**< Enable interrupt */
}

/*******************************************************************************/
/*                            LOCAL FUNCTIONS                                  */
/*******************************************************************************/

/*******************************************************************************/
/**
 * @name         u8ResetTimer
 * @brief        Reset the MDC interrupt request flag
 * @pre          none
 * @param        none
 * @return       u8_timerval
 * @note @n
 *      -Global data: none @n
 *      -Called by     : MDCisr @n
 *      -Description: Reset the MDC interrupt request flag
 *
 */
/*******************************************************************************/
uint8_t u8ResetTimer(void)
{
  uint8_t u8_timerval;       /**< Local variable declaration */
  MCFLG_MCZF = 1;           /**< Reset interrupt request flag */

  return(u8_timerval);
}


#endif

/*******************************************************************************/
/*                                                                             */
/*                              EOF                                            */
/*                                                                             */
/*******************************************************************************/

/** @} */
```

전역(Exported) 함수의 정의

Blank.h 파일 내의 PROTOTYPES OF EXPORTED FUNCTIONS 에서 함수의 선언 필요

지역(Local) 함수의 정의

Blank.c 파일 내의 PROTOTYPES OF LOCAL FUNCTIONS 에서 함수의 선언 필요

End of file / End of Doxygen

<Power Programming>

소프트웨어 구조와 모듈 설계

Chapter 07
객체지향 소프트웨어 설계를 위한 지침

Version	Document maturity (draft / valid)	Date of Issue (200x-MM-DD)	Author/Owner	Check/Release	Description
1.0		2008-05-24	W. Lee		Initial version
1.1		2012-04-30	W. Lee		Separate from ADPro_A

[객체지향 소프트웨어 설계를 위한 지침]

- 소프트웨어의 각 부분을 실제의 객체와 연관 맺어 그룹화 한다.
- 객체를 모아 구조의 각 층에 할당한다.
- 모든 자료들이 객체 내에 포함되도록 하여, 전역변수의 사용을 가능한 한 피한다.
- 자료가 지속적으로 필요한 것이 아니라면 지역변수로 선언하여 RAM 의 낭비를 피한다.
- 객체의 자료는 인터페이스함수를 사용하여 접근하도록 하고, 가능한 한 Pulling 방식을 사용한다.
- 모든 객체는 초기화 함수를 갖도록 한다. 본격적으로 주기적인 태스크들을 수행하기 전에 초기화 함수들을 수행하여야 한다.
- 하나의 main 함수, 혹은 스케쥴링 함수를 사용하여 각 객체가 수행하여야 하는 일을 정리하도록 한다.
- 제어 기능을 수행하기 위하여 주기적인 태스크를 수행할 때, 입력 관계된 객체를 먼저 호출하고, 중간 연산 객체를 호출하고 끝으로 출력 관계된 객체를 호출하게 된다.