



EDDI

Electronic Design
Development Institute

에디로봇아카데미 임베디드 마스터 Lv1 과정

제 5기

2023. 06. 25

어광선

CONTENTS

1. 인라인 어셈블리 및 예제
2. CMake
3. CMake를 사용한 프로젝트 빌드 과정
4. DDD(Domain Driven Design)
5. eXtreme Programming

Inline Assembly

Inline Assembly는 Assembly 명령어들을 Inline Function으로 작성하는 것을 말한다. Inline Assembly는 실행속도 향상을 위해 사용하며, 시스템 프로그래밍에서 자주 사용된다. C언어에서 `asm` 키워드를 사용하여 어셈블리 명령어를 코딩한다. C언어와 어셈블리 명령어를 혼합할 수 있으므로, 빠른 실행처리는 어셈블리로 수행하고, 결과 데이터는 C언어의 변수들로 확인 가능한 장점이 있다. 어셈블리 언어는 2가지 유형이 있다. 인텔 형식과 AT&T 형식이다. GNU C 컴파일러 즉, GCC는 AT&T 문법이며 우리는 이것을 사용하기로 한다. 인라인 어셈블리 코드는 아래와 같이 2가지 형식이다.

- 1) `asm ("assembly code")`
- 2) `__asm__("assembly code")`

Inline Assembly 예제

```
#include <stdio.h>

#define MAX 6

unsigned int arr[MAX] = { 1, 2, 3, 4, 5 };
//unsigned int other_arr[MAX];

void print_arr_info(unsigned int *arr){
    int i;

    for (i = 0; i < MAX; i++){
        {
            printf("arr[%d] = %d\n", i, arr[i]);
        }
    }
}

void can_I_execute_asm (void)
{
    register unsigned int r1 asm("r1") = 0;
    register unsigned int r2 asm("r2") = 0;
    register unsigned int r3 asm("r3") = 0;
    register unsigned int r4 asm("r4") = 0;
    register unsigned int *r5 asm("r5") = 0;

    r5 = arr;

    asm volatile("mov r1, #3");
    asm volatile("mov r2, r1, lsl #2");
    asm volatile("mov r4, #2");
    asm volatile("add r3, r1, r2, lsl r4");
    asm volatile("stmia r5, {r1, r2, r3}");
    print_arr_info(arr);
}

void yes_you_can_execute_asm (void)
{
    register unsigned int r0 asm("r0") = 0;
    register unsigned int *r1 asm("r1") = NULL;

    r1 = arr;

    asm volatile("mov r2, #0x4");
    asm volatile("ldrr r0, [r1, r2]");

    printf("r0 = %u\n", r0);
}

int main(void){
    can_I_execute_asm();
    yes_you_can_execute_asm();
    return 0;
}
```

c언어 변수 앞에 register만 붙인다고 실제로 register mapping을 해줘야함.

stmia는 Store, Increment After 이라는 뜻이다.
r5가 arr 배열의 base address 이고, base address부터 한 배열
공간마다 r1, r2, r3을 순차적으로 저장한다.

Inline Assembly 예제

```
void can_I_execute_asm (void)
{
    register unsigned int r1 asm("r1") = 0;
    register unsigned int r2 asm("r2") = 0;
    register unsigned int r3 asm("r3") = 0;
    register unsigned int r4 asm("r4") = 0;
    register unsigned int *r5 asm("r5") = 0;

    r5 = arr;

    asm volatile("mov r1, #3");
    asm volatile("mov r2, r1, lsl #2");
    asm volatile("mov r4, #2");
    asm volatile("add r3, r1, r2, lsl r4");
    asm volatile("stmia r5, {r1, r2, r3}");
    print_arr_info(arr);
}
```

asm("r1")과 같이 실제 asm에 mapping 되도록 code를 작성해야 실제 register에 mapping 된다.

어셈블리어 명령어들을 실행하고, 마지막에 stmia r5, {r1, r2, r3}를 통해 r1~r3이 차례대로 r5부터 주소공간에 순차적으로 저장된다.

Inline Assembly 예제

```
void yes_you_can_execute_asm (void)
{
    register unsigned int r0 asm ("r0") = 0;
    register unsigned int *r1 asm ("r1") = NULL;

    r1 = arr;

    asm volatile("mov r2, #0x8");
    asm volatile("ldr r0, [r1, r2]");

    printf("r0 = %u\n", r0);
}
```

r1은 arr 배열의 주소를 가지고 있고,
[r1 + 8] 가리키는 공간의 값을 r0에 저장한다.

loads R0 with the word pointed at by r1+8

r1 = arr 배열의 3번째 공간의 값이 51이므로
r0에는 51이 저장되고,
printf 함수를 통해 51이 출력된다.

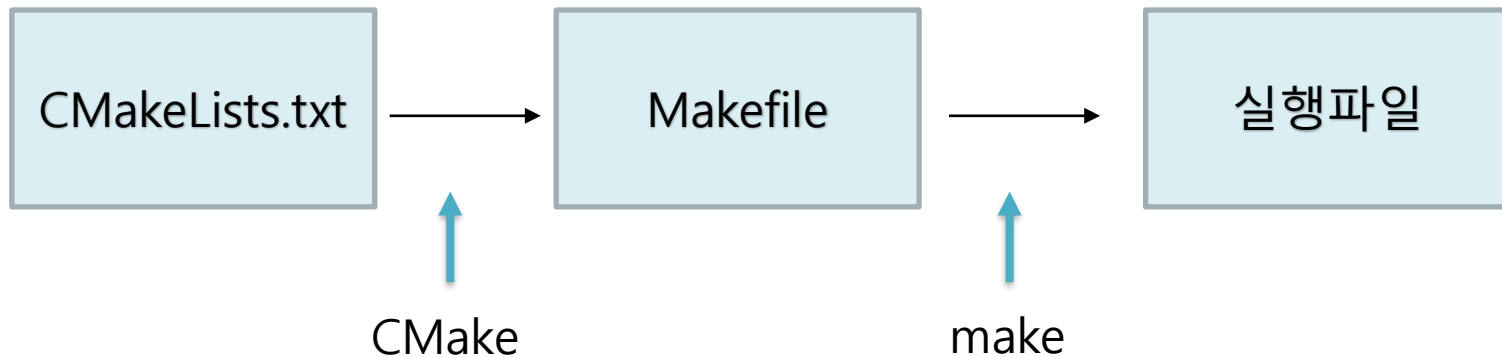
```
root@gseo:/home/eddi/EmbeddedMasterLv1/57/GwangseonEo/lecture/5_week# qemu-arm-static -L /usr/arm-linux-gnueabi ./inline_asm_1.out
arr[0] = 3
arr[1] = 12
arr[2] = 51
arr[3] = 4
arr[4] = 5
arr[5] = 0
r0 = 12
```



Cmake는 빌드 파일을 생성해주는 Program이다. make를 사용한다면 Cmake를 통해서 Makefile을 생성할 것이고, 요즘 핫한 Ninja를 사용한다면 Cmake를 통해서 .ninja 빌드 파일을 만들어 줄 것이다.

Cmake는 빌드 프로그램이 아니라 빌드 파일을 생성하는 프로그램이다.

Cmake는 대부분의 C/C++ 계열 프로젝트에서 사용되고 있다. 사실 요즘에는 Cmake를 안쓰는 곳을 찾아보기 힘들다. Cmake의 역사가 긴 만큼, 올바른 사용 방식이 버전에 따라 많이 바뀌어서 아직도 구시대적인 Cmake 문법을 사용하는 경우가 많다. 마치 C++ 98을 아직도 사용하고 있는 느낌으로 말이다.



CMake를 사용하는 모든 프로젝트에는 반드시 프로젝트 최상위 디렉토리에 CMakeLists.txt 파일이 있어야 한다. 해당 파일에는 Cmake가 빌드 파일을 생성하는 데 필요한 정보들이 들어있다. 따라서 보통의 컴파일 과정은 위와 같이 진행된다.

CMake를 이용한 프로젝트 빌드 과정

```
root@gseo: /home/eddi/Embr × + v
cmake_minimum_required(VERSION 3.0)

project(interface C)
set(CMAKE_C_STANDARD 99)
set(SOURCE main/main.c main/add_on_lib.c
      protocol/protocol_handler.c
      receiver/receiver.c)

add_executable(${PROJECT_NAME} ${SOURCE})
```

프로젝트에서 사용할 Cmake의 최소 버전을 명시

프로젝트 명

C99 표준을 사용

SOURCE라는 변수에 소스 파일들을 할당한다.

실행 가능한 바이너리 이름 : {PROJECT_NAME}
SOURCE는 이전에 set 명령을 통해 정의된 변수로 할당된 소스 파일들을 컴파일하여 해당 바이너리에 링크한다.

DDD(Domain Driven Design)

도메인 주도 설계(Domain Driven Design)은 각각의 기능적인 문제에 대한 영역들을 정의하는 도메인과 그 도메인을 사용하는 비즈니스 로직을 중심으로 설계하는 것을 말함.

DDD의 특징

- Data의 중심이 아닌 도메인의 모델과 로직에 집중
- 동일한 표현과 단어로 구성된 Ubiquitous Language, 보편적 언어 사용
기획-개발-사업까지 단일화된 커뮤니케이션을 지향
- Software Entity와 Domain간 개념의 일치
분석 모델과 설계가 다른 구조가 아닌 도메인 모델부터 코드까지 항상
함께 움직여지는 모델을 지향함.

DDD(Domain Driven Design)

DDD(Domain Driven Development)
임의의 프로토콜이 수신되면
해당 프로토콜 핸들러를 구동시킨다.

to-do-list에 프로토콜 수신, 기능 구현 이런식
으로 적으면 기능 구현하고 끝이다. 실제 field
에 적용하면 issue터짐. 실제 작업을 **기능 관
점**으로 했기 때문이다.

<폭포수 설계, 기능 관점>
프로토콜 수신
각각의 핸들러 구동

도메인 장점 : 내가 누구랑 상호 작용하는 지 알 수 있음.
사람이 실제 버튼을 누르는 건지, 회로에서 전압을 가한
다든지 알 수 있음.

→ 도메인 관점에서 본다면...?

<애자일, 도메인 관점>
사용자(실제 사람/단말기/회로)가
'A' 프로토콜을 전송하면 '무엇'을 한다.

DDD(Domain Driven Design)

<폭포수 설계, 기능 관점>
프로토콜 수신
각각의 핸들러 구동

<애자일, 도메인 관점>
사용자(실제 사람/단말기/회로)가
'A' 프로토콜을 전송하면 '무엇'을 한다.

- 각 프로토콜과 유저 별 스토리 라인들이 만들어짐.
-> 유스케이스를 건다는 말임.
- 유스케이스는 글로 적는다. -> 유연성 ▲
- 유스케이스 = 사용자 스토리 (사용자는 불특정 다수)

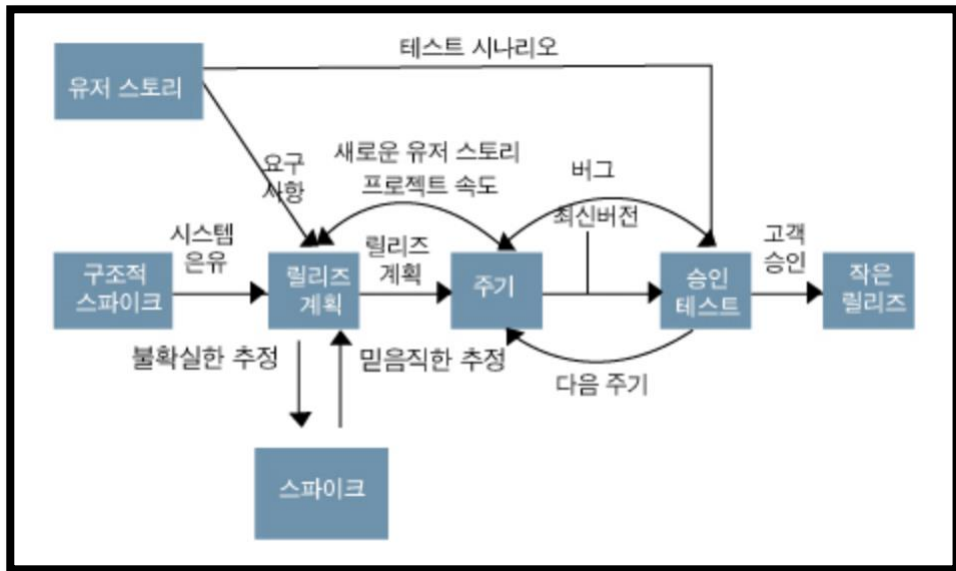
애자일을 잘 달성하려면 DDD+XP를 잘해야함

DDD : Backlog를 작성

XP : 인터페이스 방식.

결정형 함수포인트

XP(eXtreme Programming)




짧은 주기의 반복(Iteration)을 통해 요구 변화에 신속하게 대응하여 위험을 줄이고 고객 관점의 고품질 SW를 빠르게 전달하는 Agile 방법론의 기법

XP의 예시 : Interface 설계

함수포인터를 이용한 Interface 존재.
이와 관련된 table 쪽 나열

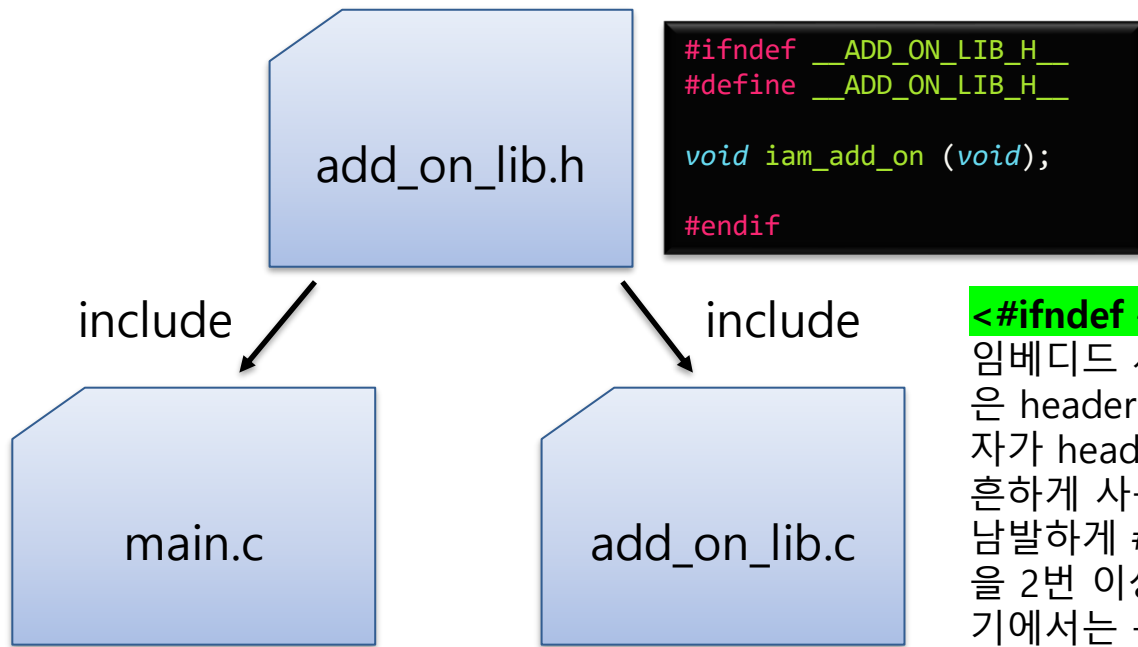
사용할 프로토콜들을 번호로 배정

번호가 들어오면 해당 table에 있는
vector가 실행되도록 한다.



대표적인 예시
"Interrupt Vector"

XP의 예시 : Interface 설계

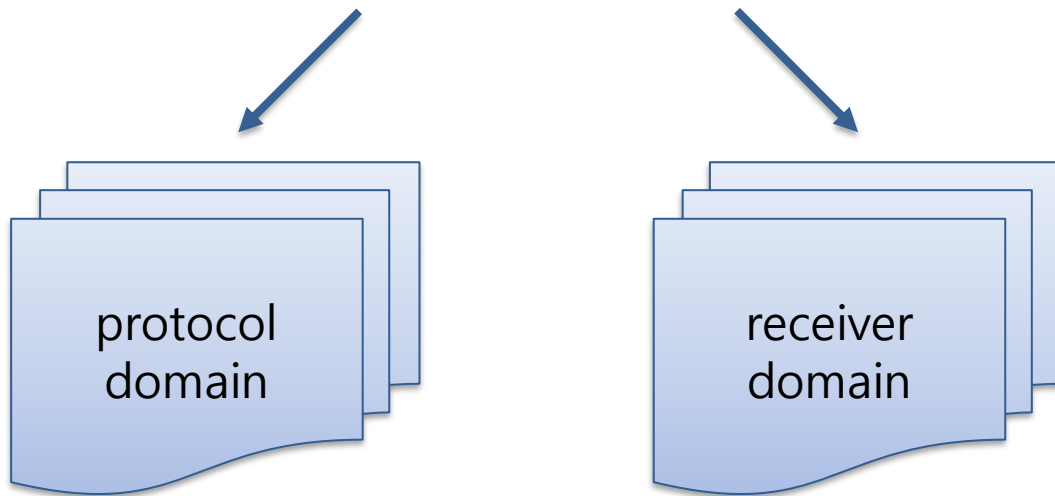


<#ifndef #define ~ #endif>

임베디드 시스템에서는 코드를 보면 많은 header file들을 볼 수가 있다. 개발자가 header file을 작성할 때 `#define`을 흔하게 사용한다. 하지만 header file을 남발하게 `#define`문 같은 정의하는 것을 2번 이상 중첩되는 문제이다. 전처리기에서는 문제가 되지 않지만 컴파일러에서는 이를 에러로 감지하므로 중첩을 방지하기 위해 위 같은 구조는 필수적이다.

XP의 예시 : Interface 설계

'A' 프로토콜을 수신하면...



XP의 예시 : Interface 설계

```
enum protocol
```

```
{
```

```
    CAMERA,  
    DC_MOTOR,  
    BLDC_MOTOR,  
    PMSM_MOTOR,  
    ACIM_MOTOR,  
    POWER_LED,  
    I2C,  
    SPI,  
    LIDAR,  
    CAN,  
    ECAP,  
    END
```

```
};
```

enum(enumerated type) 열거형이고, 서로 연관된 상수들의 집합을 이룬다. point of function(함수포인터)와 table mapping을 설계할 때 유용하다.

→ 전체 프로토콜 COUNT값을 알기 위한 END를 마지막에 작성한다.

```
#define PROTOCOL_CALL_BUFFER_COUNT    (END)  
#define PROTOCOL_CALL_BUFFER        ((END)-1)
```

XP의 예시 : Interface 설계

```
typedef void (* protocol_call_ptr_t) (void);
```

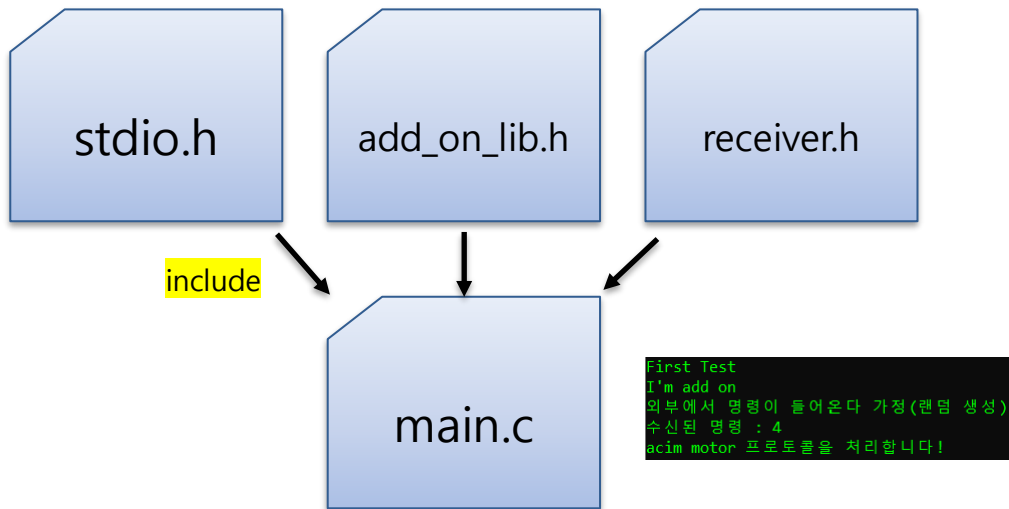
void형 데이터를 매개 변수로 취하고, void형을 return하는 함수포인터를 protocol_call_ptr_t 라는 type을 만든다.

unsigned int의 치환형 예시

```
typedef unsigned int UINT;
```

void (*)(void) : void를 return, void를 인자로 취함, 함수포인터 prototype.
int (*)(int)

XP의 예시 : Interface 설계



`recv_command_from_outbound()` 호출

XP의 예시 : Interface 설계

main.c

receiver.c

recv_command_from_outbound() 호출

protocol_call_table [command]() 호출

```
const protocol_call_ptr_t protocol_call_table[PROTOCOL_CALL_BUFFER_COUNT] =  
{  
    [0 ... PROTOCOL_CALL_BUFFER] = &protocol_not_implemented,  
    #include "protocol_call_table_mapper.h"
```

지정된 초기화자 기능을 사용한 것으로 모든 프로토콜 테이블의 모든 요소를 protocol_not_implemented 함수를 가리키도록 초기화한다.

함수 포인터들로 구성된 배열로, 각 포인터는 프로토콜 요청에 대응하는 함수를 가리킨다. 각 인덱스는 특정 프로토콜 요청에 대응하며, protocol_call_table_mapper.h 파일에서 mapping 된다.

XP의 예시 : Interface 설계

```
#ifndef __PROTOCOL_CALL_TABLE_MAPPER_H__
#define __PROTOCOL_CALL_TABLE_MAPPER_H__

#include "protocol_handler.h"
// 핸들러들 mapping 할것임
#define __PROTOCOL_CALL_TABLE(nr, sym) [nr] = sym,

__PROTOCOL_CALL_TABLE(0, proc_camera)
__PROTOCOL_CALL_TABLE(1, proc_dc_motor)
__PROTOCOL_CALL_TABLE(2, proc_blcd_motor)
__PROTOCOL_CALL_TABLE(3, proc_pmsm_motor)
__PROTOCOL_CALL_TABLE(4, proc_acim_motor)
__PROTOCOL_CALL_TABLE(5, proc_power_led)
__PROTOCOL_CALL_TABLE(6, proc_i2c)
__PROTOCOL_CALL_TABLE(7, proc_spi)
__PROTOCOL_CALL_TABLE(8, proc_lidar)
__PROTOCOL_CALL_TABLE(9, proc_can)
__PROTOCOL_CALL_TABLE(10, proc_ecap)

#endif
```

입력된 nr(인덱스) sym(함수 이름, 즉 심볼) 을 이용해 프로토콜 호출 테이블에 매핑하는 역할을 합니다. 매크로는 [nr] = sym 이라는 코드 조각을 생성한다.

__PROTOCOL_CALL_TABLE 매크로를 사용해 프로토콜 호출 테이블에 핸들러 함수들을 매핑한다. 이런식으로 프로토콜 번호와 핸들러 함수를 연결하여 어떤 프로토콜 요청이 들어왔을 때 해당하는 핸들러 함수를 빠르게 찾아 호출할 수 있다.