

삼성 청년 SW 아카데미

임베디드 C언어

<알림>

본 강의는 삼성 청년 SW아카데미의 콘텐츠로
보안서약서에 의거하여
강의 내용을 어떠한 사유로도 임의로 복사,
촬영, 녹음, 복제, 보관, 전송하거나
허가 받지 않은 저장매체를
이용한 보관, 제3자에게 누설, 공개,
또는 사용하는 등의 행위를 금합니다.

Day4-1. Byte 단위, 데이터 파싱하기

챕터의 포인트

- Byte 단위, 데이터 파싱 목표
- 구조체
- typedef
- union
- 구조체와 union QUIZ

Byte 단위, 데이터 파싱 목표

목표

- 임베디드 SW 개발자에게 Byte 단위 데이터 파싱을 어떻게 할 수 있는 지 학습한다.
- 구조체와 Union 이 어떻게 동작하는 지 학습한다.

데이터 신호를 주고 받을 때,
일반적으로 1 Byte 단위가 자주 사용되곤 한다.

- 코드를 동작 시켜본다.
- 전달 받은 데이터(target[7]) 을 byte 단위로 파싱 한다.

```
Head = AB  
Body = 1234ABCDFE  
Tail = A0
```

<https://gist.github.com/hoconoco/80e825e51c9bf44a7cc8aed11d341217>

```
//전달 받은 데이터  
uint8_t target[7] = {0xAB, 0x12, 0x34, 0xAB, 0xCD, 0xFF, 0xA0};  
  
//Byte 단위 파싱하기  
union _Data_  
{  
    uint8_t receiveData[7];  
  
    struct{  
        uint8_t head;  
        uint8_t body[5];  
        uint8_t tail;  
    }msg;  
}data;  
  
memcpy(&data, target, 7);
```

Byte 단위 파싱을 위해 구조체 / 유니온에 대해 학습한다.

- 구조체와 유니온을 적절히 섞었을 때 어떤 결과가 나오는 지 학습한다.

- **리틀 엔디안**을 항상 잊지 말자!

주소는 Byte 단위!!

구조체

목표

- Byte 단위 데이터 파싱에서 구조체를 다루기 위해 기초부터 확실히 다진다.

Struct

- 기본 타입들을 모아 새로운 타입을 만드는 문법
- 여러 타입들을 한 데 묶을 수 있다.
- 타입을 하나 생성하여, Trace로 살펴보자.

Expression	Type	Value
▼ hoho	struct ABC	{...}
(x)= a	int	10
(x)= b	int	20
+ Add new expression		

<https://gist.github.com/hoconoco/e5573185fa3b06aff4b7480dc1fd64f4>

```
struct ABC{
    int a;
    int b;
};

int main(){
    struct ABC hoho;

    hoho.a = 10;
    hoho.b = 20;

    printf("%d %d", hoho.a, hoho.b);

    return 0;
}
```

구조체에서 쓰이는 용어

1. 구조체이름

2. 멤버변수

3. 구조체변수

```
struct ABC{  
    int a;  
    int b;  
};  
  
int main(){  
    struct ABC hoho;  
  
    hoho.a = 10;  
    hoho.b = 20;  
  
    printf("%d %d", hoho.a, hoho.b);  
  
    return 0;  
}
```

C++과 혼동하면 안된다.

C와 C++ 에서의 방법이 다르다.

- ABC 구조체 변수 t를 생성할 때 초기화 가능
- 만들고 난 뒤, 한꺼번에 초기화는 불가능
ex) `t = { 4, 5 };`
 - C++에서는 가능한 문법
 - C언어에서는 불가능한 문법
- 초기화할 때 가능
ex) `t = { 4, 5 };`

```
int main(){  
    //가능  
    struct ABC hoho = {10, 20};  
  
    struct DEF hi;  
    //불가능  
    hi = {20,30};  
  
    //가능  
    hi.a = 10;  
    hi.b = 30;  
  
    return 0;  
}
```

구조체 특정 멤버를 선택해서 초기화 한다.

- C++에서는 안되는, C언어 문법이였다?! (C++20 부터 된다.)

```
struct ABC{
    int a;
    int b;
};

int main(){
    struct ABC hoho = {.a = 10, .b = 20};

    return 0;
}
```

<https://gist.github.com/hoconoco/a4720a4f822276cae71e1287c16227ce>

구조체 변수 만드는 방법

- 타입을 만들자마자 변수를 만들 수 있다.

<https://gist.github.com/hoconoco/7d35e32bbd74e2382d48863ec759631f>

```
#include <stdio.h>

struct ABC {
    int a;
    int b;
} t1, t2;

int main()
{
    t1.a = 10;
    t1.b = 20;

    t2.a = 30;
    t2.b = 40;

    return 0;
}
```

구조체 내부에 구조체를 만들 수 있다.

- 멤버 : a, b, c

구조체 이름은 필요하지
않기 때문에 생략함

```
struct ABC {  
    int a;  
  
    struct {  
        int b1;  
        int b2;  
    } b;  
  
    int c;  
};
```


중첩된 구조체 초기화하기

- Trace를 해보자.
- 두 가지 방법으로 초기화할 수 있다.

```
int main() {  
    struct ABC hi = {.a = 10,  
                     .b = {.b1 = 20, .b2 = 30},  
                     .c = 40};  
  
    struct ABC ho;  
    ho.a = 10;  
    ho.b.b1 = 20;  
    ho.b.b2 = 30;  
    ho.c = 40;  
  
    return 0;  
}
```

<https://gist.github.com/hoconoco/b64a779800458b2dae39da05aa581407>

C언어에서는 **반드시** struct를 붙여야 한다.

- C++ 에서는 struct를 안 붙여도 가능하다.
- 타입명 : **struct ABC**

```
int main(){  
    //C언어 사용법  
    struct ABC hoho;  
  
    //C++에서는 가능한 문법  
    ABC hi;  
  
    return 0;  
}
```

typedef

목표

- typedef 으로 변수와 구조체를 재정의한다.
- typedef 과 구조체를 같이 사용할 경우 일어나는 휴먼 에러를 조심한다.

기존 타입을, 원하는 이름으로 정의(definition) 하는 방법

- 변수에 쓰이는 것은 크게 다를 게 없으나 구조체와 유니온에서 혼동이 온다.
- int를 BBQ로 재정의했다.

```
#include<stdio.h>

typedef int BBQ;

int main(){

    BBQ t = 100;
    printf("%d", t);

    return 0;
}
```

<https://gist.github.com/hoconoco/e0e24cd7cffe74b1c39fd6f8d0b3bf59>

구조체 ABC를 typedef 으로 faker 로 재정의

- 새로운 이름 faker 로 hi 라는 구조체 변수 생성 후 초기화

```
struct ABC{  
    int a;  
    int b;  
};  
  
typedef struct ABC faker;  
  
int main(){  
    faker hi = {10,20};  
  
    printf("%d %d\n", hi.a, hi.b);  
  
    return 0;  
}
```

<https://gist.github.com/hoconoco/53b3c6cbdb64a37b10eae267c880ab25>

구조체와 typedef 을 한번에 쓴다.

- 기존 구조체 변수 자리에 typedef 으로 재정의 된 구조체 이름이 온다!
- 어떤 방식으로 코드가 작성되더라도 이해할 수 있어야 한다!

```
typedef struct ABC{  
    int a;  
    int b;  
}faker;  
  
int main(){  
  
    faker hi = {10,20};  
  
    printf("%d %d\n", hi.a, hi.b);  
  
    return 0;  
}
```

<https://gist.github.com/hoconoco/42aa7030edf134ac27d2d4fda400a6e5>

union

목표

- 임베디드 업계에서 파싱에 사용하는 union에 대해 학습한다.

생긴 건 구조체와 비슷하지만, 다르다.


- **멤버끼리 값을 공유**하는 특징이 있다.
- 코드를 작성하고 빌드한다.
- Trace 하면서 x 의 값을 살펴본다.

```
int main(){  
    union ABC{  
        int a;  
        char b;  
    };  
  
    union ABC x = {.a = 0x12345678};  
  
    return 0;  
}
```

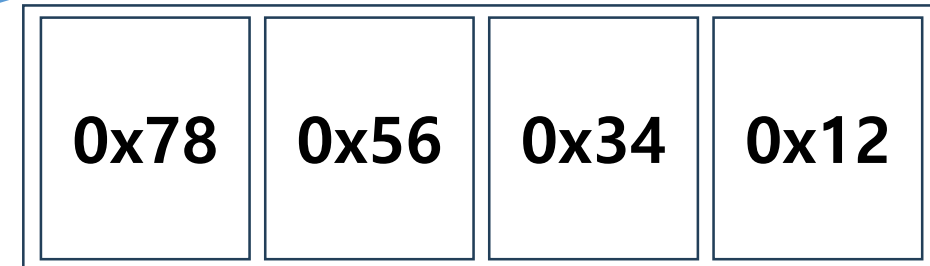
<https://gist.github.com/hoconoco/940b37847165ff6dbb8b30d32211adb6>

union 안에 있는 두 변수의 시작 메모리의 주소가 같다.

- 주소를 공유한다는 의미는 멤버의 시작 메모리 주소가 같다는 것!
- 리틀 엔디안이 동작한다!

Expression	Type	Value	Address
▼  x	union ABC	{ }	0x61ff1c
(x)= a	int	0x12345678 (Hex)	0x61ff1c
(x)= b	char	0x78 (Hex)	0x61ff1c
+ Add new expression			

char b
int a



주소는 Byte 단위!!

구조체에서 적용되던 방식과 똑같다!

- ho 는 union 변수가 아니라 재정의한 이름이다.
- union 변수는 x이다.

```
#include<stdio.h>
#include<stdint.h>

typedef union _ABC_ {
    int a;
    char b;
} ho;

int main(){

    ho x = {.a = 0x12345678};

    return 0;
}
```

<https://gist.github.com/hoconoco/5ad7bf0a9e0e70278155d84ba5ac0b56>

고정길이정수로 byte 단위로 본다.

- 코드를 빌드한 뒤 Trace 한다.
- ACON 은 재정의된 _ABC_ union 이다.

```
typedef union _ABC_  
{  
    uint32_t a;  
    uint8_t b[4];  
}ACON;  
  
int main()  
{  
    ACON data = {.a = 0x1234ABCD} ;  
  
    return 0;  
}
```

<https://gist.github.com/hoconoco/fe6b99beff6840f507dacd249a5af7dd>

메모리 뷰로 확인한다.

- data.a 에 0x12345678 값을 할당
- 리틀엔디안이 적용되서 메모리에 저장
- data.b는 메모리에 적인 값을 읽어온다.

Expression	Type	Value	Address
data	ACON	{...}	0x61ff1c
(x)= a	uint32_t	0x1234abcd (Hex)	0x61ff1c
b	uint8_t [4]	0x61ff1c (Hex)	0x61ff1c
(x)= b[0]	uint8_t	0xcd (Hex)	0x61ff1c
(x)= b[1]	uint8_t	0xab (Hex)	0x61ff1d
(x)= b[2]	uint8_t	0x34 (Hex)	0x61ff1e
(x)= b[3]	uint8_t	0x12 (Hex)	0x61ff1f
+ Add new expression			

Monitors					
&data.a	New Renderings...				
Address	0 - 3	4 - 7	8 - B	C - F	
0061FF10	30194000	00000000	00803200	CDAB3412	
0061FF20	0CFF6100	00803200	50FF6100	88124000	
0061FF30	01000000	B82E1F00	581D1F00	02000000	
0061FF40	00000000	50FF6100	4D3E1777	00003200	

Union 을 쓰는 이유

- Union 과 Struct를 섞어서 쓰면,
바이트 단위로, 자유 자재로 파싱이 가능하다.
- Union 과 Struct를 섞어서 쓰면,
비트 단위로, 자유 자재로 파싱이 가능하다.
- 리틀 엔디안에 주의한다!

주소는 Byte 단위!!

구조체와 union QUIZ

목표

- Union 과 구조체가 섞일 경우 어떻게 데이터가 저장될 지 예측한다.
- 리틀 엔디안을 항상 신경 쓴다.

주소는 Byte 단위!!

union 안에 변수가 있다.

x.a와 x.b의 값은?

```
union ABC{
    uint8_t a;
    uint8_t b;
};

int main(){
    union ABC x = {.a = 0xAB};
```

union 안에 변수가 있다.

x.a와 x.b의 값은?

- x.a = 0xAB
- x.b = 0xAB
- 메모리를 공유 한다.

```
union ABC{
    uint8_t a;
    uint8_t b;
};

int main(){
    union ABC x = {.a = 0xAB};
```

<https://gist.github.com/hoconoco/cca6bfc09f43949c5c47a689ed0a7545>

union 안에 변수와 구조체가 있다.

a,c,d 의 값은?

```
union ABC{
    uint16_t a;

    struct{
        uint8_t c;
        uint8_t d;
    }b;
};

int main(){
    union ABC x = { .a = 0x11FF };

    return 0;
}
```

union 안에 변수와 구조체가 있다.

a,c,d 의 값은?

- a = 0x11ff
- c = 0xff
- d = 0x11
- 리틀 엔디안!

<https://gist.github.com/hoconoco/49a7b040e987737e34cd3668139c1e82>

```
union ABC{
    uint16_t a;

    struct{
        uint8_t c;
        uint8_t d;
    }b;
};

int main(){
    union ABC x = { .a = 0x11FF };

    return 0;
}
```

union 안에
union 안에
변수와 구조체가 있다.

a,b,c,d 의 값은?

```
union ABC{
    uint16_t a;

    union{
        uint16_t b;
        struct{
            uint8_t c;
            uint8_t d;
        }dd;
    };
};

int main(){
    union ABC x = { .a = 0x11FF };

    return 0;
}
```

union 안에
union 안에
변수와 구조체가 있다.

a,b,c,d 의 값은?

- a = 0x11ff
- b = 0x11ff
- c = 0xff
- d = 0x11

<https://gist.github.com/hoconoco/e3abe195778657a41e92b28d932af97a>

```
union ABC{
    uint16_t a;

    union{
        uint16_t b;
        struct{
            uint8_t c;
            uint8_t d;
        }dd;
    };
};

int main(){
    union ABC x = { .a = 0x11FF };

    return 0;
}
```

union이 typedef 되었다.
변수와 구조체가 있다.

a, b[0], b[1] 의 값은?

```
typedef union _ABC_  
{  
    uint16_t a;  
  
    struct{  
        uint8_t b[2];  
    };  
}ACON;  
  
int main()  
{  
    ACON data = { 0xABCD } ;  
  
    return 0;  
}
```


union이 typedef 되었다.
변수와 구조체가 있다.

a, b[0], b[1] 의 값은?

- a = 0xABCD
- b[0] = 0xCD
- b[1] = 0xAB

<https://gist.github.com/hoconoco/40a5b002f2f4fc86ad41f471a87643b2>

```
typedef union _ABC_  
{  
    uint16_t a;  
  
    struct{  
        uint8_t b[2];  
    };  
}ACON;  
  
int main()  
{  
    ACON data = { 0xABCD } ;  
  
    return 0;  
}
```

Day4-2. Bit 단위, 데이터 파싱하기

챕터의 포인트

- Bit 단위, 데이터 파싱 목표
- Padding
- #pragma
- 비트필드
- Datasheet 이해하기

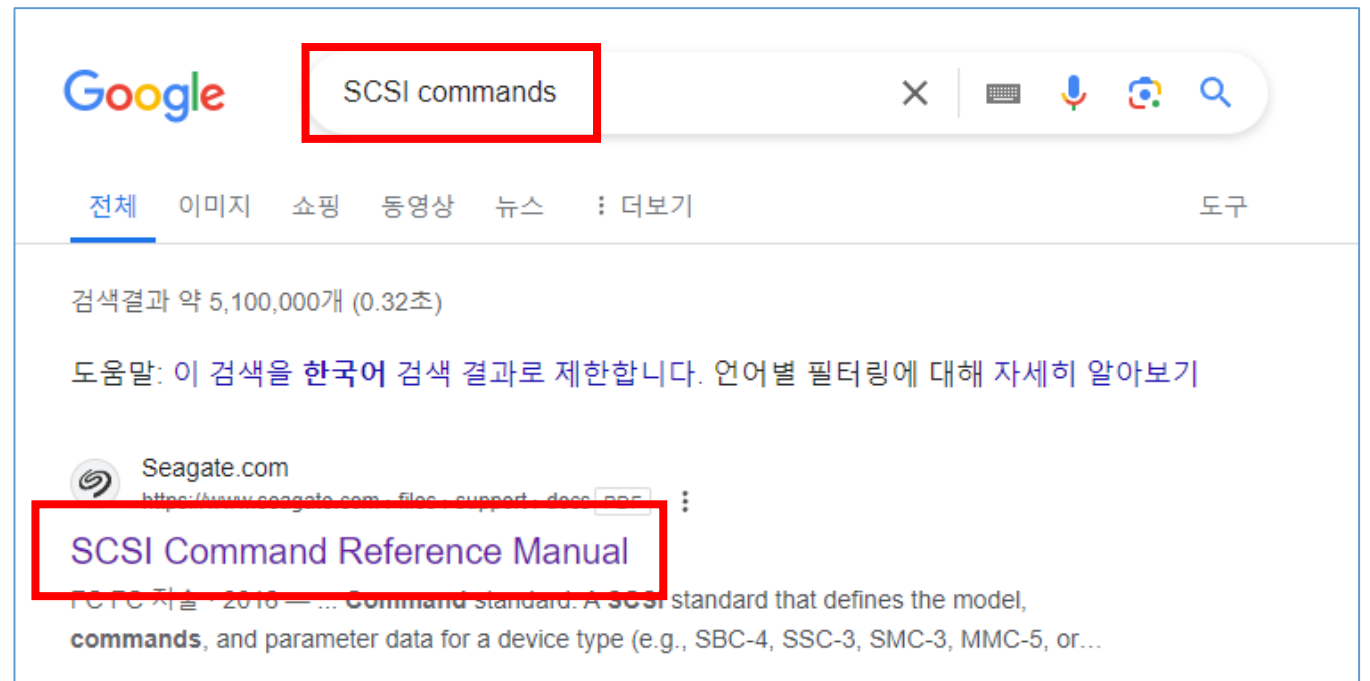
Bit 단위, 데이터 파싱 목표

목표

- Bit 단위의 데이터 파싱이 왜 필요한 지 알아본다.
- Bit 단위 데이터 파싱 코드를 살펴본다.

구글을 켜고 검색한다.

- SCSI Commands (스카시 / 스커지 커맨드)
- Small Computer System Interface
 - 컴퓨터와 주변 장치 간에 데이터 전송을 위한 표준 인터페이스



page 134. Read Six Command

- 데이터 전송 명령어
- Data를 읽어올 때 사용한다.

LBA

- (Logical Block Address)
- 데이터 시작 위치를 나타내는 논리 블록 주소

3.15 READ (6) command

This command has been declared obsolete by the T10 committee. However, it is included because it may be implemented on some products.

The READ (6) command (see table 95) requests that the device server read the specified logical block(s) and transfer them to the data-in buffer. Each logical block read includes user data and, if the medium is formatted with protection information enabled, protection information. Each logical block transferred includes user data but does not include protection information. The most recent data value written, or to be written if cached, in the addressed logical blocks shall be returned.

Table 95 READ (6) command

Bit Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE (08h)							
1	Reserved			(MSB)				
2	LOGICAL BLOCK ADDRESS							
3								
4	TRANSFER LENGTH							
5	CONTROL							

6 Byte의 데이터

- 각각의 Byte 마다 데이터가 들어 있다.

Table 95 **READ (6) command**

Bit Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE (08h)							
1	Reserved			(MSB)				
2	LOGICAL BLOCK ADDRESS							
3								
4	TRANSFER LENGTH							
5	CONTROL							

1. CPU가 주변 장치로 Read(6) Command 보낸다.
2. 주변장치가 응답한다. (6 Byte)

• 6 Byte에 대한 데이터가 들어오면, 다음과 같이 파싱을 해야 한다.

- OPERATION CODE : 8 bit
- Reserved : 3 bit
- Logical Block Address (LBA) : 21 bit
 - 1번 Byte ~ 3번 Byte 까지 분포
- Transfer Length : 8 bit
- CONTROL : 8 bit

Bit Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE (08h)							
1	Reserved			(MSB)				
2	LOGICAL BLOCK ADDRESS							
3								
4	TRANSFER LENGTH							
5	CONTROL							

Bit + Byte 단위 파싱으로
어떤 데이터라도 원하는 결과를 뽑아낸다.

<https://gist.github.com/hoconoco/2e737d60e5c0887258b6f919598fd5c5>

```
#pragma pack(1)
union Node{
    uint8_t ori[6];

    struct {

        uint8_t opcode;

        //LBA -> Bit 파싱
        uint8_t lba_part1 : 5;
        uint8_t reserved : 3;

        //LBA
        uint8_t lba_part2;
        uint8_t lba_part3;

        uint8_t length;
        uint8_t control;

    }field;
}node;
```

Padding

목표

- Bit 단위 파싱을 위해 Padding 에 대해 학습한다.
- CPU가 데이터를 읽는 방식에 대해 이해한다.

구조체 내부에 변수가 있다.

구조체의 크기는?!

- $\text{int} - 4 + \text{char} - 1 = 5$ 가 아니다!
- 놀랍게도 8 이다!!!!

<https://gist.github.com/hoconoco/5c134cf821a3745217b8349ad81df0dd>

```
#include<stdio.h>
#include<string.h>

int main(){
    struct Node{
        int a;
        char b;
    }c;

    printf("%d\n", sizeof(c) );
    return 0;
}
```

CPU는 4 byte 단위로 데이터를 읽는다.

b의 시작 주소에 따라 CPU는 두 번 읽어야 한다.

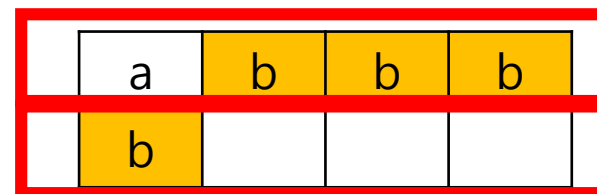
- 시간이 오래 걸린다.

Padding을 사용해서 맞춰서 해결한다!

- 근데 여기서 또 하나의 문제가 생긴다 ^^ “역시 그 세계”



<패딩 사용!>
CPU가 한 번의 동작으로
변수 b의 값을 읽을 수 있다.



<패딩 미사용!>
CPU가 두 번의 동작으로
변수 b의 값을 읽을 수 있다.
→ 느리다.

sizeof(타입)

- sizeof는 함수라고 부르면 안된다.
c언어에서는 연산자 같은 문법으로 취급한다. [암기]
- sizeof 연산자로, 메모리를 차지하는 크기를 측정할 수 있다.

```
struct Node {  
    int a;  
    char b;  
}kfc;  
  
printf("%d", sizeof(kfc));
```

#pragma

목표

- CPU의 성능을 올리기 위해 Padding 을 사용하지만, 문제가 생긴다.
- 어떤 문제가 생기는 지 이해하고 Padding 을 없애는 다양한 시도에 대해 알아본다.

Padding의 존재

- kfc.a 의 주소는 0xAB04 라고 가정하자.
- kfc.b 의 주소는, 0xAB05 가 아닐 수 도 있다.
- 패딩이 붙으면, 패딩이 들어간 만큼 메모리 위치가 달라진다.
- 매 번 이것을 생각할 수 없다.

<https://gist.github.com/hoconoco/7f4a9a45cf59f27dfe29ce16d8948617>

```
int main(){
    struct Node{
        int a;
        char b;
    };

    struct Node kfc;

    printf("%X\n", &kfc.a);
    printf("%X\n", &kfc.b);
    return 0;
}
```

컴파일러마다 Padding 을 붙이는 방식이 다르다.

- 우리가 Bit 파싱을 해야 하는 데, 매 번 Padding 을 생각하면서 코드를 짤 순 없다.
- 시스템이 달라질 때마다 코드를 고치라는 것은 야근을 의미한다.

그래서 Padding 을 없애고 파싱을 해야 한다.

컴파일러에게 패딩을 넣지 말라고 지시하는 명령어

- (임베디드용) ARM의 DS 컴파일러 (ADS, Arm Development Studio)

- 방법 1 : `#pragma pack(1);`
- 방법 2 : `__attribute__((packed))`

- (임베디드용) IAR 컴파일러

- 방법 1 : `__packed struct { ... };`
- 방법 2 : `#pragma pack(1);`
- 방법 3 : `#pragma pack(push, 1) ~ #pragma pack(pop)`

- MSVC 컴파일러 (Visual Studio)

- 방법 1 : `#pragma pack(1)`
- 방법 2 : `#pragma pack(push, 1) ~ #pragma pack(pop)`

- GCC

- 방법 1 : `#pragma pack(1)`
- 방법 2 : `#pragma pack(push, 1) ~ #pragma pack(pop)`
- 방법 3 : `__attribute__((packed))`

어디서든 즐기는 `#pragma pack(1)` 을 학습한다.

#pragma pack(1)

- 해당 코드 밑으로 모든 구조체들은 패딩을 사용하지 않는다.

#pragma pack(4)

- 다시 패딩을 사용한다.

<https://gist.github.com/hoconoco/1566a8c6f23eb21f2e846598036d9da0>

```
#pragma pack(1)
//여기서부터 패딩 사용안함
    struct Node1{
        char a;
        int t;
    };
//여기까지
#pragma pack(4)
//이제 패딩 다시 사용함
    struct Node2{
        char a;
        int t;
    };
```

이제 패딩을 고려하지 않기 위해
꼭 `#pragma pack(1)` 을 사용한다.

비트필드

목표

- Byte를 쪼개는 비트필드에 대해 학습한다.

멤버 변수의 특정 bit만 사용하는 데이터 형식

- a : uint8_t 중에서 1 비트만 사용
- b : uint8_t 중에서 5 비트만 사용
- c : uint8_t 중에서 2 비트만 사용
- 도합 1 byte

```
#pragma pack(1)
struct ABC{
    uint8_t a : 1;
    uint8_t b : 5;
    uint8_t c : 2;
};
int main(){

    struct ABC X;

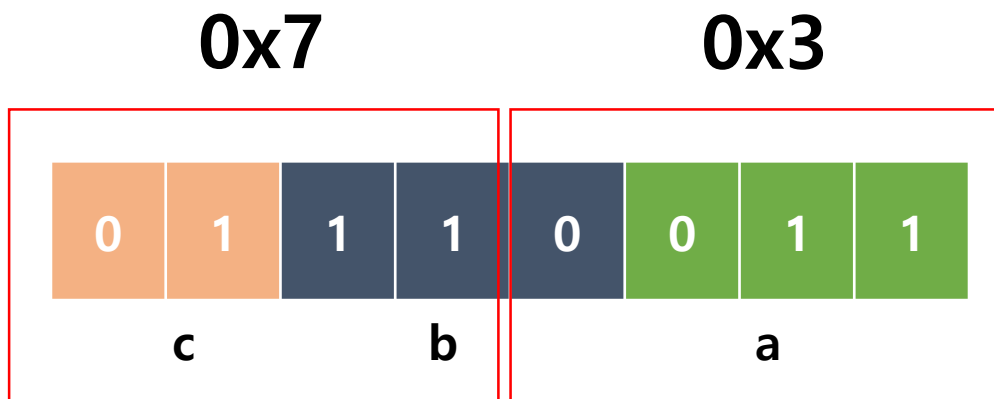
    printf("%d", sizeof(struct ABC));

    return 0;
}
```

<https://gist.github.com/hoconoco/ac2fb76058648e0f114982f64da56f39>

0번 bit부터 순차적으로 값을 넣는다.

- Trace 해서 조사식에서 살펴본다.
- 메모리 뷰로 확인한다. (&k)



```
int main(){
    struct Node{
        uint8_t a : 3;
        uint8_t b : 3;
        uint8_t c : 2;
    };

    struct Node k = { 0x3, 0x6, 0x1};

    printf("%X\n", k.a);
    printf("%X\n", k.b);
    printf("%X\n", k.c);
    return 0;
}
```

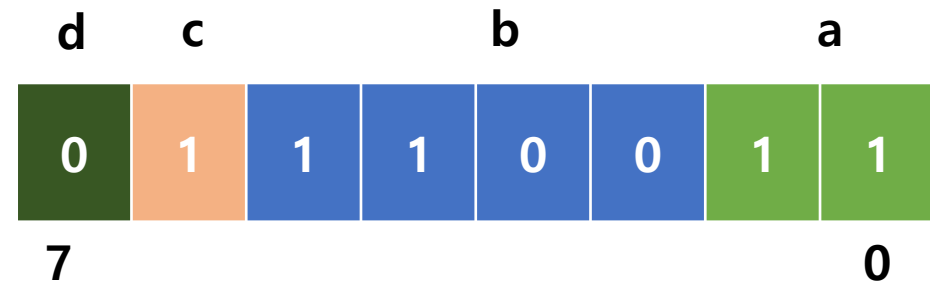
특정 비트를 지칭할 때 다음과 같이 지칭한다.

- **[큰bit 번호 : 작은 bit 번호]**
- 매우 중요하다. 모든 데이터시트에 써 있는 방식이다.

Bit는 반대!!!!

오른쪽 그림은 4개로 구분된다.

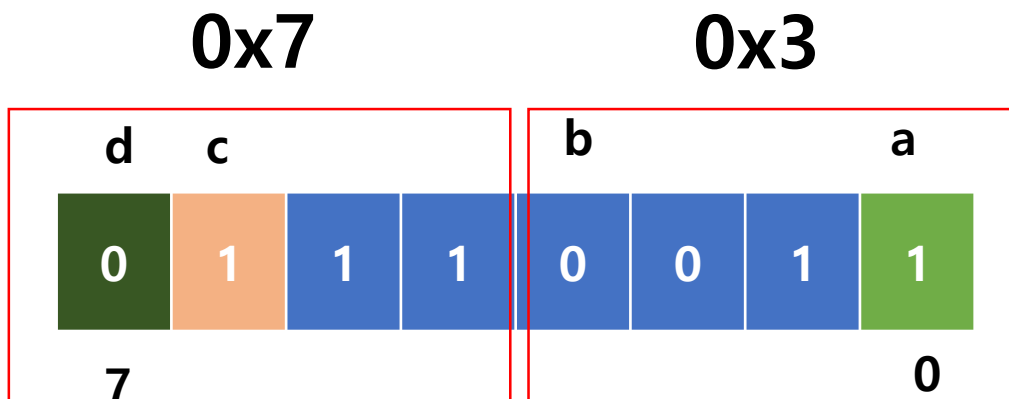
- a [1 : 0]
- b [5 : 2]
- c [6 : 6]
- d [7 : 7]



0x73 을

다음과 같은 비트 필드에 넣는다면?

- 고민할 필요가 없다.
- 그냥 비트 개수 만큼 변수에 나누면 된다.



<https://gist.github.com/hoconoco/a44497f25e6a3806e05ba695d745e726>

```
#pragma pack(1)

uint8_t data = 0x73;

struct Node {
    uint8_t a : 1;
    uint8_t b : 5;
    uint8_t c : 1;
    uint8_t d : 1;
};
```

union 과 struct 를 같이 사용한다.

<https://gist.github.com/hoconoco/4cff4f971e35defa14766f4f6ec1f5be>

```
#pragma pack(1)
//파싱할 데이터
uint8_t data = 0x73;

union Node{
    uint8_t origin;

    struct {
        uint8_t a : 1;
        uint8_t b : 5;
        uint8_t c : 1;
        uint8_t d : 1;
    }field;
}k;

//직접 할당
k.origin = data;
```

Datasheet 이해하기

목표

- 데이터 시트를 보고 어떻게 데이터를 파싱하는 지 학습한다.

page 1076.

디바이스 전자 서명

- 이 챕터에서는 장치를 식별할 수 있도록 특정 주소에 작성된 데이터에 대한 설명이 나와 있다.
- 서명에 포함된 내용
 - ID 값
 - 전기 특성들
- 전자서명은 플래시 메모리에 저장되어 있다.
- JTAG 으로 읽을 수 있다.

Device electronic signature

RM0008

30 Device electronic signature

Low-density devices are STM32F101xx, STM32F102xx and STM32F103xx microcontrollers where the Flash memory density ranges between 16 and 32 Kbytes.

Medium-density devices are STM32F101xx, STM32F102xx and STM32F103xx microcontrollers where the Flash memory density ranges between 64 and 128 Kbytes.

High-density devices are STM32F101xx and STM32F103xx microcontrollers where the Flash memory density ranges between 256 and 512 Kbytes.

XL-density devices are STM32F101xx and STM32F103xx microcontrollers where the Flash memory density ranges between 768 Kbytes and 1 Mbyte.

Connectivity line devices are STM32F105xx and STM32F107xx microcontrollers.

This section applies to the whole STM32F10xxx family, unless otherwise specified.

The electronic signature is stored in the System memory area in the Flash memory module, and can be read using the JTAG/SWD or the CPU. It contains factory-programmed identification data that allow the user firmware or other external devices to automatically match its interface to the characteristics of the STM32F10xxx microcontroller.

파악해야 하는 것

1. Base address : 0x 1FFF F7E0
 2. 메모리 전체 사이즈 : 2 Byte, [15:0]
- 0x 1FFF F7E0 에서 2Byte 를 읽으면, Flash Memory Size 를 알 수 있다.

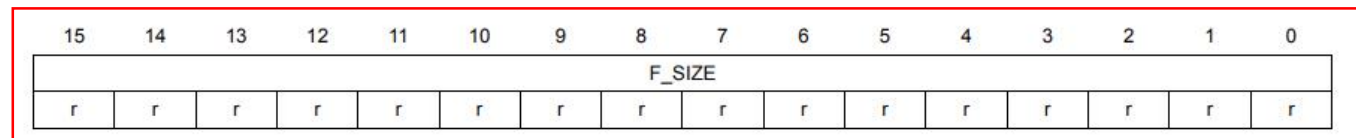
- Read Only Data
- KB 단위로 기록 됨

30.1 Memory size registers

30.1.1 Flash size register

Base address: 0x1FFF F7E0

Read only = 0xXXXX where X is factory-programmed



Bits 15:0

F_SIZE: Flash memory size

This field value indicates the Flash memory size of the device in Kbytes.
Example: 0x0080 = 128 Kbytes.

유니크 디바이스 ID (96 bits)

- 총 96 bits 의 데이터이다.
- ID 모음집
 - 시리얼 넘버 / 보안 키 등
다양한 용도로 사용될 수 있다.
- 사용자에 의해 변경될 수 없다.
 - (=factory programmed)
- Base 주소 : 0x1FFF F7E8

30.2 Unique device ID register (96 bits)

The unique device identifier is ideally suited:

- for use as serial numbers (for example USB string serial numbers or other end applications)
- for use as security keys in order to increase the security of code in Flash memory while using and combining this unique ID with software cryptographic primitives and protocols before programming the internal Flash memory
- to activate secure boot processes, etc.

The 96-bit unique device identifier provides a reference number which is unique for any device and in any context. These bits can never be altered by the user.

The 96-bit unique device identifier can also be read in single bytes/half-words/words in different ways and then be concatenated using a custom algorithm.

Base address: 0x1FFF F7E8

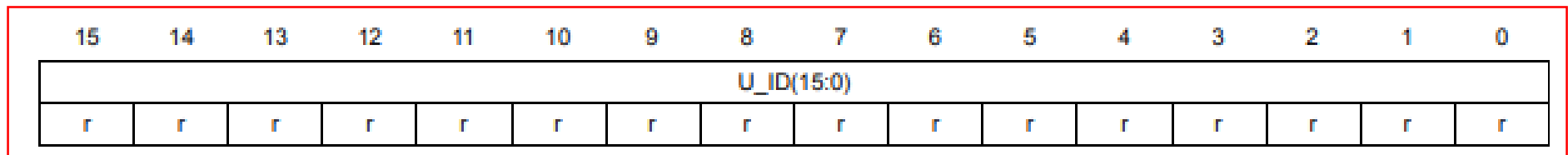
Unique ID Bits

- Base address + offset (0x00) 한 뒤, 2 Byte를 읽으면, unique ID 를 알 수 있다.

Base address: 0x1FFF F7E8

Address offset: 0x00

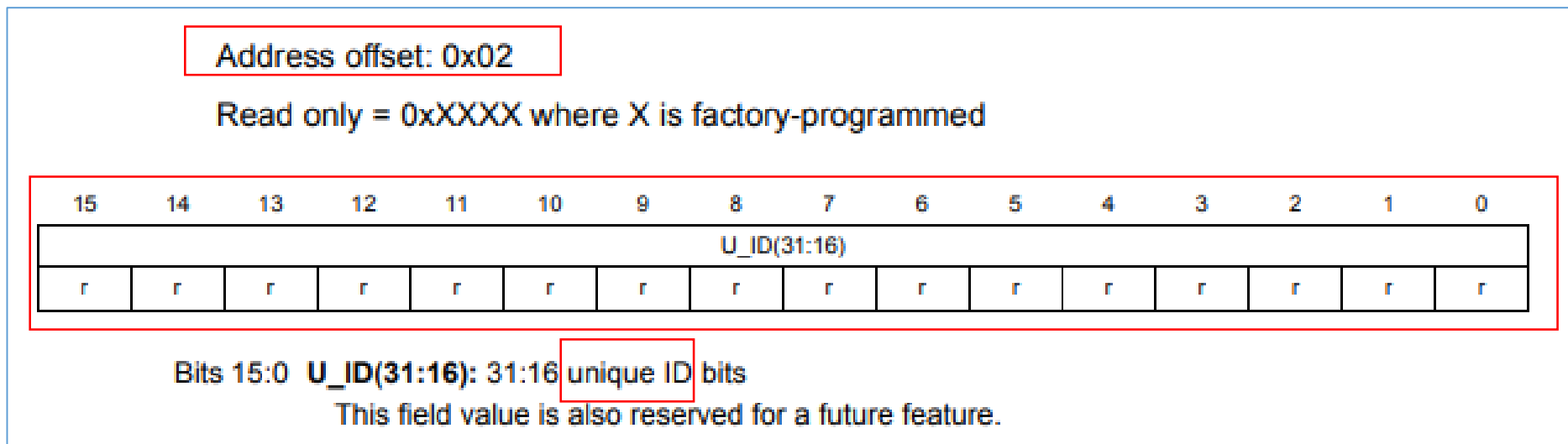
Read only = 0XXXXX where X is factory-programmed



Bits 15:0 U_ID(15:0): 15:0 unique ID bits

Unique ID Bits

- Base address + offset (0x02) 한 뒤, 2 Byte를 읽으면, unique ID 를 알 수 있다.
- 이 필드는 유니크 ID 필드이며, 미래를 위해 남겨두는 Reserved 영역이다.

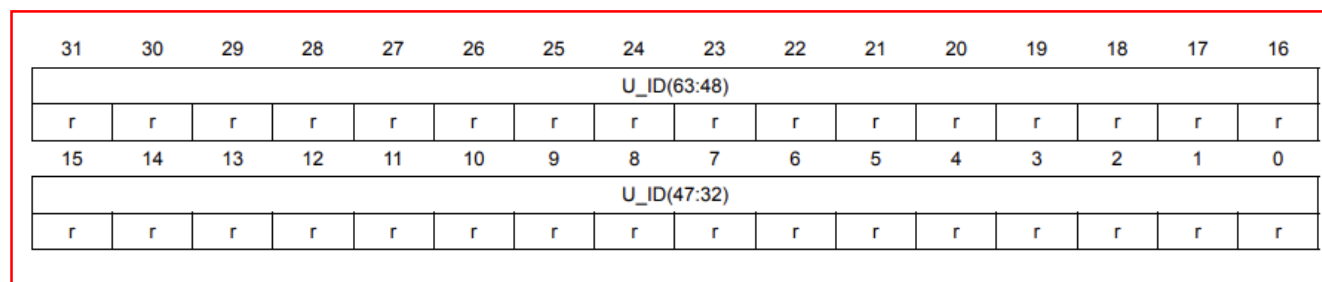


나머지 영역도 모두 유니크 ID

- Base + offset (0x04)
 - 4 Byte
- Base + offset (0x08)
 - 4 Byte

Address offset: 0x04

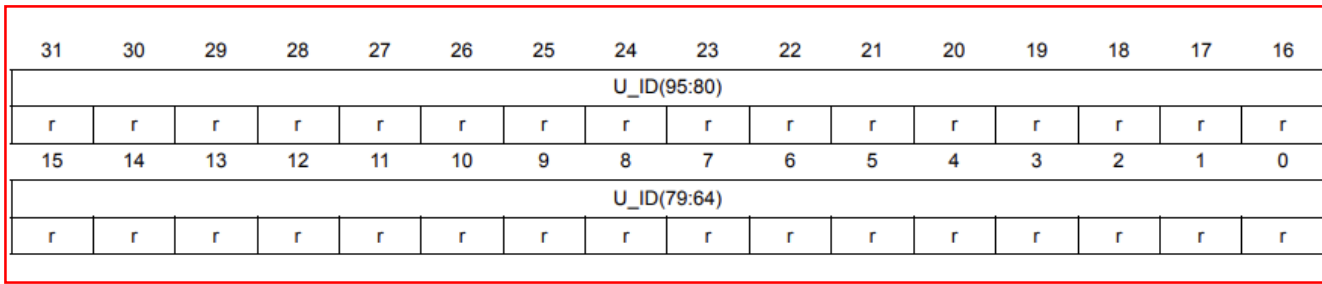
Read only = 0XXXXX XXXX where X is factory-programmed



Bits 31:0 U_ID(63:32): 63:32 unique ID bits

Address offset: 0x08

Read only = 0XXXXX XXXX where X is factory-programmed



Bits 31:0 U_ID(95:64): 95:64 Unique ID bits.

Day4-3. 도전

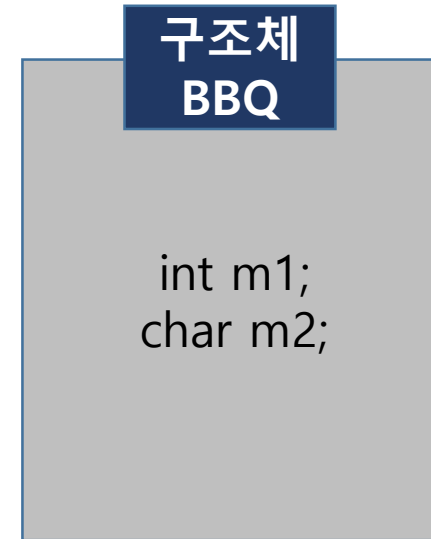
구조체 변수를 생성하고 초기화한다.

- 값은 적당히 넣는다.



typedef 로 구조체 이름을 재정의하고 변수를 생성한다.

- 멤버 값은 적당히 만들어서 초기화한다.



Union 을 사용하여

64bit에 저장된 데이터를 8 bit 단위로 파싱한다.

- 리틀 엔디안 형태로 출력한다.

```
uint64_t g = 0xABCD12345678CD01;  
uint8_t buf[8];
```

```
for (int i = 0; i < 8; i++) {  
    printf("%02X", buf[i]);  
}
```

공용체를 이용하여 Data Parsing 하기

- 다음과 같은 데이터를 받았다 “1234567890ABCDEF987654”
- 데이터 규칙은 다음과 같다.
 1. Header 정보 : 5 Byte
 2. Body 정보 : 4 Byte
 3. Tail 정보 : 2 Byte

프로토콜에 맞게

바이트 파싱을 해보자.

```
int main(){
    uint8_t data[11] = {0x12,0x34,0x56,0x78,0x90,0xab,0xcd,0xef,0x98,0x76,0x54};

    return 0;
}
```

공용체를 이용하여 Data Parsing 하기

- 다음과 같은 데이터를 받았다 “ABCDEF1209AFAF”
- 데이터 규칙은 다음과 같다.
 1. Header 정보 : 6 Byte
 2. Body 정보 : 4 Byte
 3. Tail 정보 : 4 Byte

프로토콜에 맞게

바이트 파싱을 해보자.

```
int main() {  
    uint8_t data[7] = {0xab, 0xcd, 0xef, 0x12, 0x09, 0xAF, 0xAF};  
  
    return 0;  
}
```

1. 직접 비트필드를 사용해서 값 세팅

- $a[1:0] \rightarrow 0x1$
- $b[3:2] \rightarrow 0x0$
- $c[5:4] \rightarrow 0x2$
- $d[7:6] \rightarrow 0x3$
- 오른쪽 필드에 데이터를 채우고 원본 데이터를 구한다.

--	--	--	--	--	--	--	--

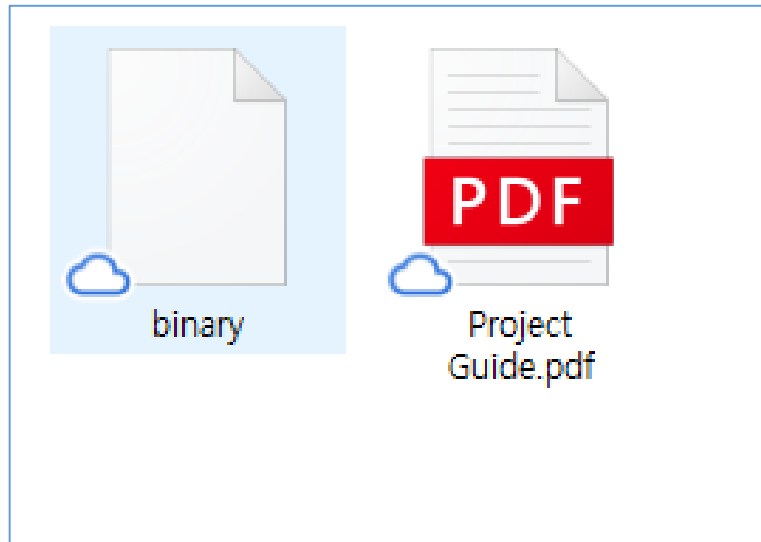
2. 메모리 확인하기

- 비트필드로 원본 데이터를 파싱한다.
- 메모리 뷰와 trace 결과를 같이 캡처해서 제출한다.

Datasheet Project 소개

binary 파일을 읽는 것이 목표

- Project Guide.pdf 파일에는 어떻게 data 가 저장되어 있는 지 해설이 있다.
- 우선, binary 파일을 읽도록 한다.



Device electronic signature

개인정보가 들어있는 파일에 대한 Datasheet 이다. 이를 파악하여 각 항목을 출력하자. 출력해야 하는 항목은 몸무게 1개 / 유니크 ID 정보 (47bit) 이다. 모든 데이터는 리틀엔디언으로 저장되어 있으며, 출력 시 10진수로 변환하여 출력해야 한다.

1. 몸무게 레지스터

A. Base address : 0x00

B. 단위 : KG (ex. 0x0080 = 128 KG)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F_SIZE															
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Bits 15:0 F_SIZE: Flash memory size
This field value indicates the Flash memory size of the device in Kbytes.
Example: 0x0080 = 128 Kbytes.

2. 유니크 ID

A. Base address : 0x02

B. 96 bit에 유니크 ID에 대한 정보를 기록한다.

C. 총 4개의 정보를 포함한다.

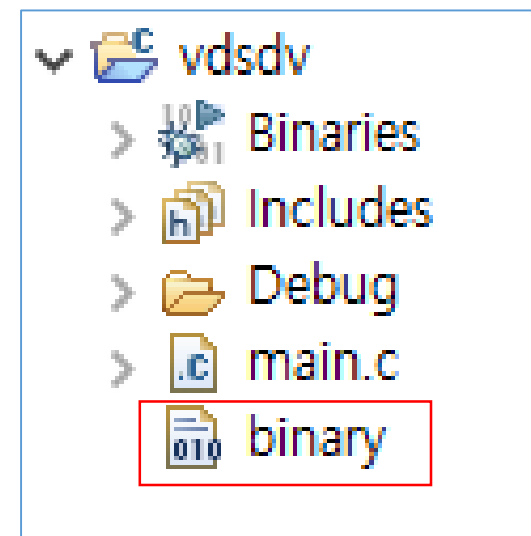
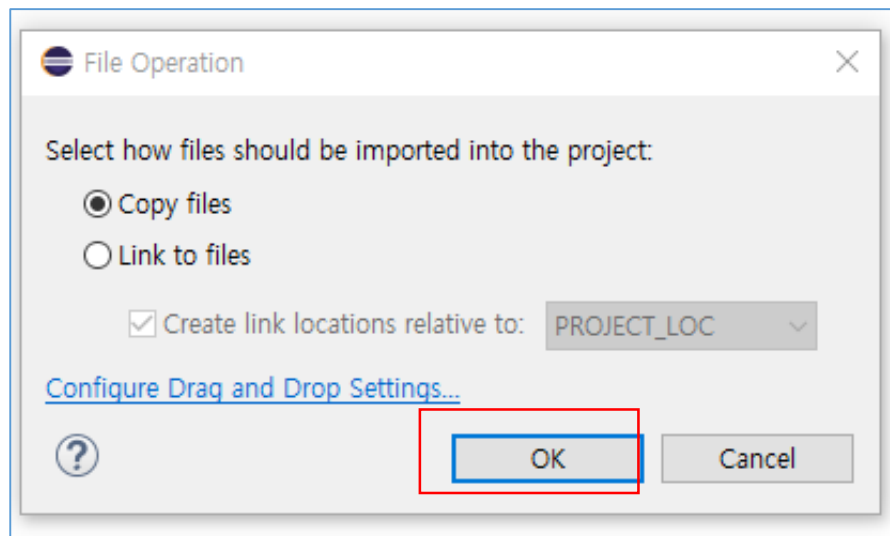
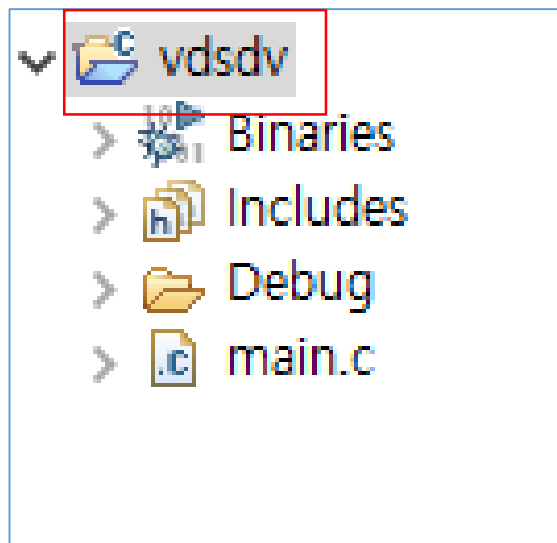
I. 공장 비밀번호 (offset : 0x00)

Address offset 0x00															
Read only = 0xXXXX where X is factory programmed															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
U_IDbits															
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Bits 15:0 U_IDbits(0): 16-bit unique ID bits

binary 파일을 프로젝트 디렉토리에 옮긴다.

- Copy files → OK 클릭



fopen / fread를 통해 binary 파일을 buf에 담는다.

- buf 값을 Trace 한다.





```
int main(){  
    FILE *fp = fopen("binary", "r");  
    uint8_t buf[14]; //16 + 96 = 112 bit / 8 = 14 Byte  
    fread(buf, 1, 14, fp);  
    fclose(fp);  
  
    return 0;  
}
```

<https://gist.github.com/hoconoco/b333c275ebff07908eb8678d1d6f4587>

Expression	Type	Value	Address
buf	uint8_t [14]	0x61ff0e	0x61ff0e
(x)= buf[0]	uint8_t	0x50 (Hex)	0x61ff0e
(x)= buf[1]	uint8_t	0x0 (Hex)	0x61ff0f
(x)= buf[2]	uint8_t	0xa (Hex)	0x61ff10
(x)= buf[3]	uint8_t	0xc (Hex)	0x61ff11
(x)= buf[4]	uint8_t	0xe8 (Hex)	0x61ff12
(x)= buf[5]	uint8_t	0xfd (Hex)	0x61ff13
(x)= buf[6]	uint8_t	0x49 (Hex)	0x61ff14
(x)= buf[7]	uint8_t	0x47 (Hex)	0x61ff15
(x)= buf[8]	uint8_t	0x4f (Hex)	0x61ff16
(x)= buf[9]	uint8_t	0x47 (Hex)	0x61ff17
(x)= buf[10]	uint8_t	0x49 (Hex)	0x61ff18
(x)= buf[11]	uint8_t	0x50 (Hex)	0x61ff19
(x)= buf[12]	uint8_t	0x4f (Hex)	0x61ff1a
(x)= buf[13]	uint8_t	0x43 (Hex)	0x61ff1b
+ Add new expression			

데이터 시트 파싱하기

- 이제 간부와 함께 명세서(pdf 파일) 을 읽고 프로젝트를 진행한다.
- 출력 결과를 제출한다.

```
weight :   
pass :   
salary :   
food :   
drink : 
```

내일 방송에서 만나요!

삼성 청년 SW 아카데미

[과제1] p.79 코드와 결과물 캡처해서 제출

[과제2] p.80 코드와 결과물 캡처해서 제출

[과제3] p.81 코드와 결과물 캡처해서 제출

[과제4] p.82 코드와 결과물 캡처해서 제출

[과제5] p.83 코드와 결과물 캡처해서 제출

[과제6] p.84 코드와 결과물 캡처해서 제출

[과제7] p.89 코드와 결과물 캡처해서 제출