Xilinx Zynq FPGA, TI DSP, MCU 기반의 프로그래밍 및 회로 설계 전문가 과정

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – 장성환
redmk1025@gmail.com

*** typedef keyword**

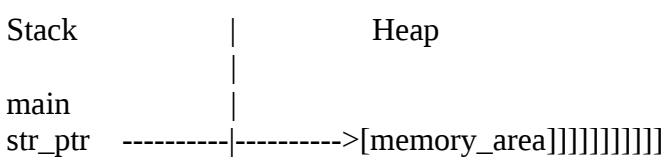
해당 함수 및 데이터 형 그리고 구조체 등을 원하는 이름으로 선언이 가능하게 만들어 준다.

주의! 배열을 typedef 선언 하고 싶을 때,
ex) typedef int INT[5]

INT arr={1,2,3,4,5}; 이런 식으로 작성이 가능하다.

*** malloc (memory allocation) , free()**

메모리 구조상 힙 영역에 data 를 할당하고 해제한다.
데이터의 크기를 알 수 없는 경우에 사용한다.
(전역변수 처럼 사용이 가능하다. 또한 함수로 생성해도 지워지지 않고 삭제도 가능하다.)



스택과 힙이 나누어 지며 스택에서 힙영역을 제어가 가능하게 된다.
여기서 부터 자료구조가 시작이 된다.
(함수에서 동적 메모리 할당 → 자료 생성 → 주소값만 저장 → 주소값으로 제어 등등?)

*** Struct**

다양한 자료를 한번에 처리하기 위한 방법 (구조체도 주소다!)

Struct 도 주소이기 때문에 구조체 포인터를 선언하여 해당 주소를 받아서 배열 처럼 처리가 가능하다. 하지만 구조체 내에 모두 같은 데이터 타입이 선언 되었을 경우만 가능하다.
ex) 구조체 {int a; int b;} 일때 구조체 포인터로 해당 주소를 받은 뒤(이름을 pa 라고 하면)

pa[0] pa[1] 이런 식으로 변수 값 접근 등 처리가 가능하다.
하지만 구조체{int int double} 등 다른 데이터가 섞일 경우 위와 같은 방식으로 처리하면 큰 에러 발생

***enum**
통신 프로토콜을 만들 때 가장 많이 사용하는 방법이다.

*** 함수 포인터**
함수의 주소를 저장할 수 있는 포인터

void (*signal(int signum, void (*handler)(int)))(int); → 함수

함수 프로토 타입이란 ?

리턴, 함수명, 인자에 대한 기술서
그렇다면 위 함수에 대한 프로토 타입은 뭘까?
이전에 배웠던 int(*p)[2] → int (*)[2]p

따라서
void (*)(int) signal(int signum, void (*handler)(int))
리턴 : void (*)(int) ← 이게 함수 포인터다.
함수명 : signal
인자 : int signum 그리고 void (*handler)(int)

토대로 해석해보자

```
#include <stdio.h>

void aaa(void){
   printf("aaa called\n");
}
```

```
void bbb(void(*p)(void)){
    p();
    printf("bbb called\n");
}
```
// void(*p)(void) : void 를 리턴하고 void 를 인자로 취하는 함수의 주소값을 저장할 수 있는 변수 p

```
int main(void){
    bbb(aaa);
    return 0;
}
```
// bbb 함수에 aaa 라는 void(*p)(void) 형 함수의 주소값을 매개변수로 전달하여 해당 요소를 실행한다.
즉 bbb 함수의 역할은 p 함수 실행과 printf 실행.

```
#include <stdio.h>

typedef struct test_class
{
        int in1;
        int in2;
        double dn1;
        double dn2;

        int (*int_op)(int, int); //리턴은 int 함수 이름은 int_op 라는 포인터 변수 매개변수는 int, int
        double (*double_op)(double, double);
        // 리턴은 double 이고 double_op 라는 함수 포인터 이름 그리고 double, double 매개변수
} tc;

int iadd(int n1, int n2)
{
        return n1 + n2;
}

int imul(int n1, int n2)
```

```
{
	return n1 * n2;
}

double dadd(double n1, double n2)
{
	return n1 + n2;
}

int main(void)
{
	int res;
	double dres;
	tc tc_inst = {3, 7, 2.2, 7.7, NULL, NULL};

	tc_inst.int_op = iadd;
	res = tc_inst.int_op(tc_inst.in1, tc_inst.in2);
	//tc_inst 라는 tc 구조체의  int_op 라는 함수포인터에 iadd 함수 주소값 전달 및 해당 구조체의 in1, in2 값을 전달하여 실행
	printf("res = %d\n", res);

	tc_inst.int_op = imul;
	res = tc_inst.int_op(tc_inst.in1, tc_inst.in2);
	printf("res = %d\n", res); //유사함

	tc_inst.double_op = dadd;
	dres = tc_inst.double_op(tc_inst.dn1, tc_inst.dn2);
	printf("dres = %lf\n", dres); //유사함

	return 0;
}


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

```
void aaa(void){
    printf("aaa called\n");
}
int number(void){
    printf("number called\n");
    return 7;
}
void (* bbb(void))(void){
    printf("bbb called\n");
    return aaa;
}
// 리턴 : void (*) void /이름: bbb /인자 : void

void ccc(void(*p)(void)){
    printf("ccc : I can call aaa!\n");
    p();
}
//리턴: void /이름: ccc /인자: void (*p)(void) → 인자로 aaa 함수만 받을 수 있다.

int (* ddd(void))(void){
    printf("ddd : I can call number\n");
    return number;
}
//int (* ddd(void))(void) → int (*)(void) ddd (void) / 리턴: int (*) (void) 이름 : ddd 인자 : void

int main(void){
    bbb()(); → aaa()로 바뀐다.
    ccc(aaa); → aaa 가 인자로 전달된다.
    ddd()();
    return 0; //결과 예측: “bbb called” → “aaa called” → “I can call aaa!” → “aaa called” → “I can call number” → ?(답은 number called)
} //중요한 사실! 함수가 선언이 되면 해당 함수의 이름은 어디서든 불러와 쓸 수 있다.(해당 함수의 주소값을 해당 함수 이름을 써서 언제든 불러올 수 있다.)
//ddd 함수를 int(*)(void) ddd(void) 라고 가독성 좋게 쓰면 에러가 난다. 결합순서의 차이로 그렇게 되는것 같다.
```

함수 포인터를 해석하기 위해서 '(*' 을 만나면 함수명을 먼저 찾고 함수 명 앞에 나온 식이라면 다음과 같이 해석한다.
뒤에서 새로운 괄호를 지나쳐 ')' 만날때 까지 뒤까지 * 앞까지 가져온다. (새로운 괄호 없다면 바로 ')'까지
ex) int (*ddd(void))(void) → int (*)(void) ddd (void) :
함수를 만들기 위해서는 가독성이 좋게 리턴형 + 이름+ 인자로 만들고 함수 생성 법칙을 통해서 실제 함수로 구현하면 된다.
ex) int (*)(void) ddd (void) → int (*ddd(void))(void) :

```
#include <stdio.h>

void aaa(void){
    printf("aaa called\n");
}

void (*bbb(void(*p)(void)))(void){
    p();
    printf("bbb called_\n");
    return aaa;
}

// void (*bbb(void(*p)(void)))(void) → void (*)(void) bbb(void(*p)(void))
//리턴 void(*)(void) / 함수명 bbb / 인자 void (*p)(void)

int main(void){
    bbb(aaa)();
    return 0;
} // bbb 함수에 aaa 라는 함수의 주소값을 매개 변수로 전달하여 aaa 함수를 실행 "aaa called" 이후 "bbb called" 이후 aaa 함수 주소 리턴하여 다시 aaa 실행
```

함수를 가리키기 위하여 변수를 선언하는 법을 알아보자.

1. 일반적으로 함수를 사용하기 위해서 함수이름() 과 같은 방법을 사용한다.

- 함수의 선언은 (리턴형-반환 변수의 이름은 생략) + (함수이름) + (반환형-매개 변수의 이름도 같이 넣음)

2. 해당 함수를 가리키는 함수 포인터를 선언하여 가리키고 싶은 함수의 이름을 넣는다.

- 함수 포인터 선언은 (리턴형-반환 변수의 이름은 생략) + (*함수포인터이름) + (반환형-매개 변수의 이름도 같이 넣음)

<결론>

해당 함수를 포인터 변수로 받을 때는 (*함수 포인터 이름)과 같은 형식을 쓰면된다.

원하는 리턴과 인자값을 통하여 함수를 만들 시에는 가독성이 좋게 변환하여 만들고 이전 페이지에 서술한 함수 변환공식을 따라서 함수를 만들면 된다.

```
#include <stdio.h>
int (* aaa(void))[2]{
    static int a[2][2] = {10, };
    printf("aaa called\n");
    return a;
}
// int (*)[2] aaa(void)
// 반환형은 int 형 배열 포인터 이름은 aaa 인자는 void

int (*(* bbb(void))(void))[2]{
    printf("bbb called\n");
    return aaa;
}
// int (*(* bbb(void))(void))[2] → int (*)[2] (*)(void) bbb (void)
// 리턴형은 함수 포인터 인데 리턴이 int (*)p[2] 배열 포인터 이며 인자는 void 이다. → aaa 함수를 반환 가능하다는 걸 알 수 있다.
// 이름은 bbb
// 인자는 void

int main(void){

    int (*ret)[2];
// ret 이라는 이름을 가진 int 형 배열[2]를 가리키는 배열 포인터
    int (*(*(*p[][2])(void))(void))[2] = {{bbb, bbb}, {bbb, bbb}};
```

```
//  int (*(*(*p[][2])(void))(void))[2] → int (*)[2](*)(void) (*)(void) p[][2]
// 원하는 바는  bbb 의 함수포인터를 지정하는 2*2 형태의 2 중 포인터배열을 선언한다는 뜻이다.

    int (*(*(*(*p1)[2])(void))(void))[2] = p;
// p 라는 변수는 2*2 포인터 배열이다. bbb 함수를 가리킬 수 있는 (*p1)[2] 라는 변수를 선언하였으므로
// p1 은 x*2 의 배열 포인터가 된다. p1 에 p 의 주소값을 넣음으로서, p1 을 p 처럼 사용이 가능하다.

    ret = ((*(*(*(*p1)[2])))()());

// ret = (*(*(*(*p1)[2])))()()와 동일
// p1 선언시 int (*(*(*(*p1)[2])(void))(void))[2]
// int 랑 void 떼버리면 (*(*(*(*p1)[2])))[2] // 이렇게 써보면 ? (*(*(*(*p1)[2])(void))(void)) 안됨
// (*(*(*(*p1)[2])))란 뭘까...

//ret = aaa() = bbb()()= (*(*(*(*p1)[2])))()()
//따라서 bbb =(*(*(*(*p1)[2]))) 라는 소린데…
// bbb =*p1 는 아니고...
//(*p1)[2] =bbb
// 포인터로 구현하는 ... 방법에 대하여 알아 보도록 하자.

//ret = (*p1)()(); 불가능
//ret = p[0][0]()(); 가능
//ret = p1[0][0]()(); 가능

    printf("%d\n", *ret[0]); // ret[0]이 가리키는 주소의 참조값 반환 = ret[0][0]과 같다.
    return 0;
}
```

```
#include <stdio.h>

int (* aaa(void))[2]
{
	static int a[2][2] = {{10, 20}, {30, 40}};
	printf("aaa called\n");
	return a;
}

int (*(* bbb(void))(void))[2]
{
	printf("bbb called\n");
	return aaa;
}

int (*(* ccc(void))(void))[2]
{
	printf("ccc called\n");
	return aaa;
}

int (*(* ddd(void))(void))[2]
{
	printf("ddd called\n");
	return aaa;
}

int (*(* eee(void))(void))[2]
{
	printf("eee called\n");
	return aaa;
}

int main(void)
```

```
{
        int (*ret)[2];
        int (*(*(*p[][2])(void))(void))[2] = {{bbb, ccc}, {ddd, eee}};
        int (*(*(*(*p1)[2])(void))(void))[2] = p;
        ret = ((*(*(*(*p1)[3])))()());
        printf("*ret[0] = %d\n", *ret[0]);
        printf("ret[0][0] = %d\n", ret[0][0]);
        printf("*ret[1] = %d\n", *ret[1]);
        printf("ret[1][1] = %d\n", ret[1][1]);
        return 0;
}
```

*** 함수 포인터를 도대체 왜 쓰는가 ?**

1. 비동기 처리
2. HW 개발 관점에서 인터럽트
3. 시스템 콜(유일한 SW 인터럽트 임)

여기서 인터럽트들(SW, HW)은 사실상 모두 비동기 동작에 해당한다.
결국 1 번(비동기 처리)가 핵심리하는 의미다.

그렇다면 비동기 처리라는 것은 무엇일까?

기본적으로 동기 처리라는 것은 송신하는 쪽과 수신하는 쪽이 쌍방 합의하에만 달성된다.
(휴대폰 전화 통화 등등)
반면 비동기 처리는 이메일, 카톡등의 메신저에 해당한다.
그래서 그냥 일단 던져 놓으면 상대방이 바쁠때는 못 보겠지만,
그다지 바쁘지 않은 상황이라면 메시지를 보고 답변을 줄 것이다.

이와 같이 언제 어떤 이벤트가 발생할지 알 수 없는 것들을 다루는 녀석이 바로 함수 포인터다.
사람이 이런데서는 임기응변을 잘 해야 하듯이 컴퓨터 관점에서 임기응변을 잘 하도록 만들어 주는 것이 바로
함수 포인터다.

결론! 비동기 처리 – 함수포인터

숙제

hestit/788

1. (int 를 반환하고 float, double, int 를 인자로 취하는 함수 포인터)를 반환하고
(float 을 반환하고 int 2 개를 인자로 취하는 함수포인터)를 인자로 취하는 함수를 작성하여

프로그램이 정상적으로 동작하도록 프로그래밍하시오.

(힌트 : 함수는 총 main, pof_test_main, pof_test1, pof_test2 으로 4 개를 만드세요)

int (*)(float, double, int) pof_test_main(float (*p)(int, int))
→ int (*pof_test_main(float (*p)(int, int)))(float, double, int)
인자 :float pof_test1(int a1, int b1)
리턴 :int pof_test2(float a2, double b2, int c2)

정답

```
#include <stdio.h>

int pof_test1(float a, double b, int c){

   return (a+b+c)/3.0;
}//return

float pof_test2(int a, int b){

   return (a+b)*0.23573;
}//parameter

int(*pof_test_main(float(*p)(int,int)))(float, double,int){
```

```
    float p1=p(1,2);
    printf("res = %f\n",p1);
    return pof_test1;
}
int main(void){

    int res; //최종 반환값 선언
    res=pof_test_main(pof_test2)(3.7,2.4,7);
    printf("pof_test_main res= %d\n",res);

    return 0;
}
```

---

2. (int 2 개를 인자로 취하고 int 를 반환하는 함수포인터를 반환하며 인자로 int 를 취하는 함수포인터)를 반환하는 함수는 (float 을 반환하고 인자로 int, double 을 취하는 함수 포인터)를 인자로 취한다.

이를 프로그래밍하고 역시 정상적으로 동작하도록 프로그래밍하시오.

(힌트 : 함수는 총 main, pof_test_main, pof1, subpof1, pof2 로 5 개를 만드세요)

```
#include <stdio.h>

//int(*)(int,int)
int pof1(int n1,int n2){
    return n1+n2;
}

//int(*)(int,int)subpof1(int)
int (* subpof1(int n))(int, int){
    printf("n = %d\n",n);
```

```
    return pof1;
}

//float (*)(int,double)
float pof2(int n1,double n2){
    return n1*n2;
}

//int(*)(int,int)(*)(int)pof_test_main(float(*)(int,double))
//int(*)(int,int)(*pof_test_main(float(*)(int,double)))(int)

//int(*(* pof_test_main(float(*)(int,double)))(int))(int,int)
//int(*)(int,int)(*pof_test_main(float(*)(int,double)))(int)
//int(*)(int,int)(*)(int)pof_test_main(float (*)(int,double))


int(*(* pof_test_main(float (*p)(int,double)))(int))(int,int){
    float res;
    res = p(3,7.7);
        printf("res = %f\n",res);
    return subpof1;
}


int main(void){

    int res;
    res = pof_test_main(pof2)(3)(7,3);


    return 0;
}
```

hesit/1858

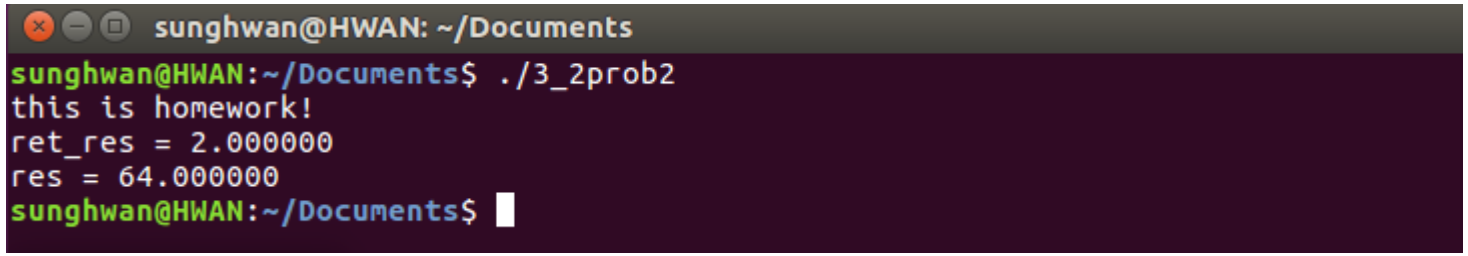
float(*(*test(void(*p)(void)))(float(*)(int,int)))(int,int)프로토 타입 함수 구동되도록 프로그래밍

float(*(*test(void(*p)(void)))(float(*)(int,int)))(int,int) →
float(*)(float(*)(int,int))(*)(int,int) test(void(*p)(void))

parameter: void(*p)(void)
→ void para_func (void)
name: test
return type : float(*)(int,int) (*) (float(*)(int,int)) → float(*)(int,int) ret_func (float(*)(int,int)) → float(*ret_func (float(*p)(int,int)))(int,int)
→ parameter : float 리턴에 (int, int)인자를 받는 함수 포인터 → float para2_func(int a, int b)
return : float 리턴에 (int,int)인자를 받는 함수 포인터 → float ret2_func(int a, int b)
type: 함수 포인터

sunghwan@HWAN: ~/Documents

```c
#include <stdio.h>

float para2_func(int x, int y){
    return x*y;
}

float ret2_func(int a, int b){
    return a*b;
}

float(*ret_func(float(*p)(int,int)))(int,int){
   float res = p(1,2);

   printf("ret_res = %f\n",res);

   return ret2_func;
}

void para_func(void){
    printf("this is homework!\n");
}

float(*(*test(void(*p)(void)))(float(*)(int,int)))(int,int){
    p();
    return ret_func;
}


int main(void){

    float res;
    float(*(*ret_f)(float(*p)(int,int)))(int,int);
    ret_f=test(para_func);

    res=ret_f(para2_func)(2,32);

    printf("res = %f\n",res);

    return 0;

}
```



---

memmove()

memory move 의 합성어

메모리의 값을 복사할 때 사용함

memmove(destination, source, size)

ex)

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

int main(void){
    int i=0;
    int src[5]={1,2,3,4,5};
    int dst[5];

    memmove(dst,src,sizeof(src));

    for(i=0;i<5;i++)
        printf("dst[%d] = %d\n",i,dst[i]);

    return 0;

}
```

---

memcpy()

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

int main(void){
    char src[30]="This is amazing";
    char *dst=src+3;

    printf("before memmove = %s\n",src);
    memcpy(dst,src,3);
    printf("after memmove = %s\n",dst);

    return 0;
}
```

---

main 입출력

```
#include <stdio.h>

int main(int argc, char **argv, char **env){
    int i;

    printf("argc = %d\n", argc);

    for(i=0;i<argc;i++)
        printf("argv[%d]=%s\n",i,argv[i]);
    for(i=0;env[i];i++)
        printf("env[%d]=%s\n",i,env[i]);
```

```
    return 0;
}
```

env 는 도대체 뭘까?
환경변수를 뿌려줌

***strlen()**

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

int main(int argc, char **argv){
    char *str = "This is the string";
    int len = strlen(str);
    printf("len = %d\n",len);
    return 0;
}
```

strlen 은 null 문자를 포함하여 count 하지 않기 때문에
malloc 함수에 strlen+1 의 값만큼 넣어줘서 null 문자를 넣을 수 있도록 해주자!

***strncpy()**

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

int main(void){
    char src[20] = "abcdef";
    char dst[20];

    strncpy(dst,src,3);
```

```
    printf("dst = %s\n",dst);
    return 0;
}
```

strcpy 는 해킹의 위험이 있기 때문에 strncpy 를 쓰자!

*** strncmp()**

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

int main(int argc, char **argv){
    char src[20] = "made in korea";
    char dst[20] = "made in china";

    if(!strncmp(src,dst,8))
        printf("src, dst 는 서로 같음\n");
    else
        printf("str, dst 는 서로 다름\n");

    return 0;
}
```

두개가 같으면 결과가 0 이 나온다. 이것에 주의하자.

*** STACK**

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

#define EMPTY 0

struct node{
```

```
    int data;
    struct node *link;
};

typedef struct node Stack;

Stack *get_node(){
    Stack *tmp;
    tmp=(Stack *)malloc(sizeof(Stack));
    tmp->link=EMPTY;
    return tmp;
}

void push(Stack **top,int data){
    Stack *tmp;
    tmp = *top;
    *top = get_node();
    (*top)->data=data;
    (*top)->link=tmp;
}

int pop(Stack **top){
    Stack *tmp;
    int num;
    tmp = *top;
    if(*top==EMPTY){
        printf("Stack is empty\n");
        return 0;
    }

    num = tmp->data;
    *top =(*top)->link;
    free(tmp);
    return num;
```

```
}

int main(void){
    Stack *top = EMPTY;
    push(&top,10);
    push(&top,20);
    push(&top,30);
    push(&top,40);

    printf("%d\n",pop(&top));
    printf("%d\n",pop(&top));
    printf("%d\n",pop(&top));
    printf("%d\n",pop(&top));
    printf("%d\n",pop(&top));
    printf("%d\n",pop(&top));

    return 0;
}
```

숙제
stack 에서 pop 을 그려 봅시다.