

**TI DSP, MCU 및 Xilinx Zynq  
FPGA  
프로그래밍 전문가 과정**

강사 – Innova Lee(이상훈)

[gcccompil3r@gmail.com](mailto:gcccompil3r@gmail.com)

학생 – 문한나

[mhn97@naver.com](mailto:mhn97@naver.com)

### 예제 1)

**<test>**

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

```
void gogogo(int voidv){
```

```
printf("SIGINT Accur!\n");
exit(0);
```

$$\}$$

```
int main(void){
```

```
signal(SIGINT,gogogo);//시그널 등록
```

```
for(;;){
```

```
printf("kill Test\n");
sleep(2);
```

$$\}$$

```
return 0;
```

$$\}$$

// &->백그라운드에서 실행 혹시라도 끝 일 있으면 끄라고 pid 값 알려줌

**<kill.c>**

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
```

```
int main(int argc, char *argv[]){
```

```
if(argc < 2)
    printf("Usage : ./exe pid\n");
```

else

```
kill(atoi(argv[1]),SIGINT);//argv 는 스트링이니까 인트로 바꿔줌
```

```
return 0;
```

$$\}$$
[illegible]

```
mhn@mhn-Z20NH-AS51B5U: ~/linux/27
mhn@mhn-Z20NH-AS51B5U:~/linux/27$ ./kill 5534
mhn@mhn-Z20NH-AS51B5U:~/linux/27$
```

실행할 때 &를 쓰면 백그라운드에서 실행이 된다. 그리고 PID 값도 알려주는데 그 이유는 혹시라도 끝 일 있으면 끄라고 알려주는 것이다.

test 파일에서 SIGINT, 즉 CTRL+C 가 입력되면 gogogo 함수로 가라는 행동지침이 입력되었다.

그리고 for 문을 돌면서 2 초마다 kill test 가 출력되고 있다.

여기서 CTRL+C 를 입력해도 죽고 KILL.C 파일로 KILL PID 를 입력해도 죽는다.

주목해야 할 점은 함수포인터를 사용하여 다른 프로세스를 죽였다는 점이다

## 예제 2)

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

```
struct sigaction act_new;
struct sigaction act_old;
```

```
void sigint_handler(int signo){
```

```
    printf("Ctrl + C\n"); //act_old 에 암것도 없음 그래서 두번 누르면 꺼짐
    printf("If you push it one more tmie then exit\n");
    sigaction(SIGINT, &act_old, NULL);
```

```
}
```

```
int main(void){
```

```
    act_new.sa_handler = sigint_handler; //시그널 액션 핸들러 핸들러에 값을 넣어주고 있다(시그널의 두번째 인자)
```

```
    sigemptyset(&act_new.sa_mask); // sigemptyset -> 시그널 아무것도 안막을 꺼야~~~ 근데 시그널을 막아야할 경우? 엄청 중요한 작업이 있어서 그걸 먼저 처리해야할 경우 이때는 막아야함
```

```
    sigaction(SIGINT, &act_new, &act_old); //act_old 가 들어오면 act_new 를 실행하겠다 근데 여기서 act_new 는 아무것도 없다
```

```
    while(1){
```

```
        printf("sigaction test\n");
        sleep(1);
```

```
    }
```

```
    return 0;
```

```
}
```

//시그널이랑 동일하게 사용 가능

//차이점은 시그액션은 리턴값을 act\_old 에 넣음

//남의 코드를 빠르게 볼 수 있는 꿀팁?

//&를 보자! &가 들어있다는 것은 주소를 전달한거임 즉 포인터를 전달

//즉 &보낸 애들을 변경시킬 수 있다는 것

//함수는 오직 리턴이 한개. 하지만 포인터를 쓰면 여러개의 처리를 같이할 수 있음

//&을 보면 함수를 통해 값을 변경시키거나 뭔가를 받아서 오겠구나!

## sigaction()

sigaction() 함수는 signal()보다 향상된 기능을 제공하는 시그널 처리를 결정하는 함수이다.

signal()에서는 처리할 행동 정보로 시그널이 발생하면 호출이될 함수 포인터를 넘겨 주었다.

그러나 sigaction()에서는 struct sigaction 구조체 값을 사용하기 때문에 좀더 다양한 지정이 가능하다.

struct sigaction 구조체

```
struct sigaction {  
    void (*sa_handler)(int); //시그널을 처리하기 위한 핸들러. SIG_DFL, SIG_IGN 또는 핸들러 함수  
    void (*sa_sigaction)(int, siginfo_t *, void *); //밑의 sa_flags가 SA_SIGINFO 일때 a_handler 대신에 동작하는 핸들러  
    sigset_t sa_mask; // 시그널을 처리하는 동안 블록화할 시그널 집합의 마스크  
    int sa_flags; // 아래 설명을 참고하세요.  
    void (*sa_restorer)(void); // 사용해서는 안됩니다.  
};
```

형태

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

인수

**int signum**                      시그널 번호

**struct sigaction \*act**        설정할 행동. 즉, 새롭게 지정할 처리 행동

**struct sigaction \*oldact** 이전 행동, 이 함수를 호출하기 전에 지정된 행동 정보가 입력됩니다.

반환

**0**      성공

**-1**     실패



```
mhn@mhn-Z20NH-AS51BSU:~/linux/27$ ./a.out  
sigaction test  
  
sigaction test  
  
sigaction test  
  
sigaction test  
sigaction test  
sigaction test  
sigaction test  
sigaction test  
sigaction test  
sigaction test  
sigaction test  
  
sigaction test  
^C  
If you push it one more tmie then exit  
sigaction test  
sigaction test  
sigaction test  
sigaction test  
^C  
mhn@mhn-Z20NH-AS51BSU:~/linux/27$ vi sigaction1.c
```

처음 행동지침을 지정해 주고 while 문을 돌면서 sigaction test 를 계속 찍는다

그리고 Ctrl + C 를 누르게 되면 함수가 호출되어 If you push it one more tmie then exit\n 를 찍고 한번 더 누르면 종료된다. 종료되는 이유는 act\_old 가 널값이기 때문이다

### 예제 3)

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

```
void *task1(void *X){
```

```
    printf("Tread A Complete\n");
```

```
}
```

```
void *task2(void *X){ //뭐든지 리턴 할 수 있고 뭐든지 받을 수 있음
```

```
    printf("Tread B Complete\n");
```

```
}
```

```
int main(void){
```

```
    pthread_t ThreadA, ThreadB; //pid_t 랑 같은 의미 thread 만들꺼야
```

```
    pthread_create(&ThreadA,NULL,task1,NULL); //스레드주소보내니까 뭔가를 바꿀 것이다
```

```
    pthread_create(&ThreadB,NULL,task2,NULL); //create 는 값을 채운다 테스트 2 는 threadB 가 구동시킬  
    것이다. thread 의 생김새를 만들어놓은 것이고 메모리에 올려지진 않았다
```

```
    pthread_join(ThreadA,NULL); //메모리에 올려졌다
```

```
    pthread_join(ThreadB,NULL);
```

```
    return 0;
```

```
}
```

//컴파일이 안되는 이유?

//뒤에다가 옵션을 주자! gcc thread.c -lpthread

//병렬처리는 thread 사용

//아주 빠른 처리는 dsp

//cpu 는 순차처리 하지만 그래픽 카드 안에 cpu 가 많아서 병렬처리를 해줌 그래서 그래픽카드는 클럭보다 밴드위  
스랑 숫자가 중요함

```
mhn@mhn-Z20NH-AS51BSU:~/linux/27$ gcc thread.c
/tmp/cckoseWC.o: In function `main':
thread.c:(.text+0x60): undefined reference to `pthread_create'
thread.c:(.text+0x7b): undefined reference to `pthread_create'
thread.c:(.text+0x8c): undefined reference to `pthread_join'
thread.c:(.text+0x9d): undefined reference to `pthread_join'
collect2: error: ld returned 1 exit status
mhn@mhn-Z20NH-AS51BSU:~/linux/27$ ./a.out
bash: ./a.out: No such file or directory
mhn@mhn-Z20NH-AS51BSU:~/linux/27$ gcc thread.c -lpthread
mhn@mhn-Z20NH-AS51BSU:~/linux/27$ ./a.out
Tread A Complete
Tread B Complete
mhn@mhn-Z20NH-AS51BSU:~/linux/27$
```

ThreadA, ThreadB 를 만든다.

그리고 pthread\_create 로 값을 채운다. 여기서 &ThreadB 는 task2 로 구동시키도록 만들었다. NULL 은 기본 특  
성으로 지정한 것이다. 그리고 함수가 호출되어 Tread A Complete 와 Tread B Complete\n 를 찍고 pthread\_join  
로 종료되는 것을 기다린다.

헤더파일

```
#include <pthread.h>
```

**pthread\_create** 는 새로운 쓰레드를 생성한다.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
void *(*start_routine) (void *), void *arg);
```

새로운 쓰레드는 start\_routine 함수를 arg 아규먼트로 실행시키면서 생성된다. attr 을 NULL 로 할경우 기본 특성으로 지정된다.

반환값

성공할경우 쓰레드식별자인 thread 에 쓰레드 식별번호를 저장하고, 0 을 리턴한다.

실패했을경우 0 이 아닌 에러코드 값을 리턴한다.

**pthread\_join**

```
int pthread_join(pthread_t th, void **thread_return);
```

pthread\_join 는 식별번호 th 로 시작된 쓰레드가 종료되는걸 기다린다.

반환값

성공할경우 쓰레드식별자인 thread 에 쓰레드 식별번호를 저장하고, 0 을 리턴한다.

실패했을경우 0 이 아닌 에러코드 값을 리턴한다.

**예제 4)**

<basic\_client.c>

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

```
#include <stdlib.h>
```

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

```
#include <unistd.h>
```

```
#include <arpa/inet.h>
```

```
#include <sys/socket.h>
```

```
typedef struct sockaddr_in si;
```

```
typedef struct sockaddr * sap;
```

```
void err_handler(char *msg){
```

```
    fputs(msg,stderr);
```

```
    fputc('\n',stderr);
```

```
    exit(1);
```

```
}
```

```
int main(int argc,char **argv){
```

```
    int sock;
```

```
    int str_len;
```

```
    si serv_addr; //서버주소
```

```
    char msg[32];
```

```
//192.168.0. -> 사설
```

```
    if(argc != 3){
```

```
        printf("use: %s <ip> <port>\n",argv[0]);
```

```
        exit(1);
```

```
    }
```

sock = socket(PF\_INET, SOCK\_STREAM, 0); //파일디스크립터 얻어옴(네트워크상) 내가 통신을 할 수 있는... sock 는 fd 소켓은 오픈이랑 같다!

```
if(sock == -1)
    err_handler("socket() error");
```

```
memset(&serv_addr,0,sizeof(serv_addr)); //서버 어드레스 초기화
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(argv[1]); //아이피주소 들어감
serv_addr.sin_port = htons(atoi(argv[2])); //포트번호 들어감
```

```
if(connect(sock, (sap)&serv_addr,sizeof(serv_addr)) == -1)
    err_handler("connect() error"); //sock 자기자신에 대한 fd 서버어드레스에는 서버 ip 랑 어드레스
어디에 접속할 지는 포트번호 보고 찾을
```

```
str_len = read(sock,msg,sizeof(msg)-1); //서버에서 메시지쓴거 읽음 read 는 블로킹 메시지 들어올때까지
안움직임
```

```
if(str_len == -1)
    err_handler("read() error!");
```

```
printf("msg from serv: %s\n",msg); //메시지에 서버에서 보낸 메시지 들어있음
close(sock);
```

```
return 0;
```

```
}
```

```
<basic_server.c>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
```

```
typedef struct sockaddr_in si;
typedef struct sockaddr * sap;
```

```
void err_handler(char *msg){

    fputs(msg,stderr);
    fputc('\n',stderr);
    exit(1);

}
```

```
int main(int argc,char **argv){
```

```
    int serv_sock;
    int clnt_sock;
```

```
    si serv_addr; //위에 구조체
    si clnt_addr;
    socklen_t clnt_addr_size; //4 바이트 소켓길이
```

```
char msg[] = "Hello Network Programming"; //전달하려는 문자열
```

if(argc !=2){ //포트번호 입력 7777 이 통로, 80 이 웹 / 20,21 은 ftp / 22 는 ssh 포트번호 알고 있으면 특정 역할을 알 수 있다. 우리가 웹 들어가면 무조건 포트번호 80 임 7777 은 우리가 만든 전용 커스텀 포트임 예를들어 게임은 또 전용 포트가 있음 이것을 서비스 번호라고도 한다 두개 잘 입력하면 소켓으로감 소켓도 파일임

```
printf("use : %s <port>\n",argv[0]);  
exit(1);  
}
```

//127.0.0.1 로컬호스트

```
serv_sock =socket(PF_INET, SOCK_STREAM,0); //파일 디스크립터 넘어옴 ip ver4, tcp/ip 를 사용하겠
```

다

```
if(serv_sock == -1)  
err_handler("socket() error");
```

```
memset(&serv_addr,0,sizeof(serv_addr)); //서버어드레스 구조체(si) tcp/ip 설정
```

```
serv_addr.sin_family = AF_INET; //구조체 안에 들어있는거
```

```
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY); //그냥 패턴 외우자
```

```
serv_addr.sin_port = htons(atoi(argv[1])); //지금당장 필요없음
```

```
if(bind(serv_sock, (sap)&serv_addr, sizeof(serv_addr)) == -1) //바인드?서버의 ip 주소세팅 127.0.0.1 세
```

팅됨

```
err_handler("bind() error");
```

```
if(listen(serv_sock,5)==-1) //5 명받겠음 여기서 클라이언트 기다림
```

```
err_handler("listen() error"); //이상받으면 안됨
```

```
clnt_addr_size = sizeof(clnt_addr); //32
```

```
clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_addr,&clnt_addr_size); //서버 소켓이 클라이언트  
의 접속을 기다리고 있음 여기서 클라이언트 접속허용해줌
```

```
if(clnt_sock == -1)  
err_handler("accept() error");
```

write(clnt\_sock, msg, sizeof(msg)); //클라이언트 소켓의 파일디스크립터 넘어옴(원격에 있는 파일) 원격  
으로 세마포어 만듬 원격에 있는 클라이언트에 라이트하고있음(메시지쓰고 있다) sock

```
close(clnt_sock);
```

```
close(serv_sock);
```

```
return 0;
```

```
}
```

```
basic_client.c  convert_endian.c  kill  read_client.c  socket.c  test.c  
mhn@mhn-Z20NH-AS51B5U:~/linux/27$ gcc -o serv basic_server.c  
mhn@mhn-Z20NH-AS51B5U:~/linux/27$ ./serv 7777  
mhn@mhn-Z20NH-AS51B5U:~/linux/27$ █  
Page 8 of 8 2,262 words, 6,837 characters Default Style
```

```
mhn@mhn-Z20NH-AS51B5U:~/linux/27$ gcc -o clnt basic_client.c  
mhn@mhn-Z20NH-AS51B5U:~/linux/27$ ./clnt 127.0.0.1 7777  
msg from serv: Hello Network Programming  
mhn@mhn-Z20NH-AS51B5U:~/linux/27$ █
```



## socket()

### <헤더>

```
#include <sys/types.h>
#include <sys/socket.h>
```

### <형태>

**int socket(int domain, int type, int protocol);**

### <인자>

**int domain** 인터넷을 통해 통신할 지, 같은 시스템 내에서 프로세스 끼리 통신할 지의 여부를 설정합니다.

domain	domain 내용
PF_INET, AF_INET	IPv4 인터넷 프로토콜을 사용합니다.
PF_INET6	IPv6 인터넷 프로토콜을 사용합니다.
PF_LOCAL, AF_UNIX	같은 시스템 내에서 프로세스 끼리 통신합니다.
PF_PACKET	Low level socket 을 인터페이스를 이용합니다.
PF_IPX	IPX 노벨 프로토콜을 사용합니다.

**int type** 데이터의 전송 형태를 지정하며 아래와 같은 값을 사용할 수 있습니다.

type	type 내용
SOCK_STREAM	TCP/IP 프로토콜을 이용합니다.
SOCK_DGRAM	UDP/IP 프로토콜을 이용합니다.

**int protocol** 통신에 있어 특정 프로토콜을 사용을 지정하기 위한 변수이며, 보통 0 값을 사용합니다.

### <반환>

- 1 이외 : 소켓 식별자
- 1 : 실패

## bind()

bind() 함수는 소켓에 IP 주소와 포트번호를 지정해 줍니다. 이로서 소켓을 통신에 사용할 수 있도록 준비가 됩니다.

<헤더>

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

<형태>

```
int bind(int sockfd, struct sockaddr *myaddr, socklen_t addrlen);
```

<인수>

**int** sockfd : 소켓 디스크립터

**struct** sockaddr \*myaddr : 주소 정보로 인터넷을 이용하는 AF\_INET 인지 시스템 내에서 통신하는 AF\_UNIX 에 따라서 달라집니다.

인터넷을 통해 통신하는 AF\_INET 인 경우에는 struct sockaddr\_in 을 사용합니다.

```
struct sockaddr_in {
    sa_family_t    sin_family;           /* Address family          */
    unsigned short int sin_port          /* Port number             */
    struct in_addr  sin_addr;            /* Internet address        */

    /* Pad to size of `struct sockaddr'. */
    unsigned char   __pad[__SOCK_SIZE__ - sizeof(short int) -
        sizeof(unsigned short int) - sizeof(struct in_addr)];
};
```

시스템 내부 통신인 AF\_UNIX 인 경우에는 struct sockaddr 을 사용합니다.

```
struct sockaddr {
    sa_family_t sa_family; /* address family, AF_XXX */
    char sa_data[14]; /* 14 bytes of protocol address */
};
```

socklen\_t addrlen : myadd 구조체의 크기

반환

0 : 성공

-1 : 실패

## connect()

connect() 함수는 생성한 소켓을 통해 서버로 접속을 요청합니다.

<헤더>

**#include** <sys/types.h>

**#include** <sys/socket.h>

<형태>

**int** connect(**int** sockfd, **const struct** sockaddr \*serv\_addr, socklen\_t addrlen);

<인자>

**int** sockfd : 소켓 디스크립터

**struct** sockaddr \*serv\_addr : 서버 주소 정보에 대한 포인터

socklen\_t addrlen : **struct** sockaddr \*serv\_addr 포인터가 가르키는 구조체의 크기

반환

0 : 성공

-1 : 실패

Network 프로그래밍에서 가장 메인이 되는 것은 CS(Client Server)와 토콜로지(네트워크 구성도)이다.

그리고 추가적으로 TCP/IP 프로토콜이 있는데 이 프로토콜은 7 계층으로 되어있다. 하지만 이건 이론적인 얘기이고 실제 구현은 4 계층으로 한다.

<맥통신>

만약 같은 대역에 있는 장치끼리 통신을 하고 싶으면 아이피 주소로 스위치에 요청을 한다.

스위치는 그 요청을 받고 자신의 대역폭에 있는 모든 장치에 찾는 아이피를 뿌린다. 그 후 반응이 오면 통신이 시작된다. 다른 대역에 있는 장치끼리 통신을 하려면 라우터를 거치게 된다. 라우터가 통신을 하고 싶은 장치의 아이피를 알려주면 그 아래에 있는 스위치가 또 방송을 한다. 그리고 반응을 하면 통신을 시작할 수 있게 된다.

결국 네트워킹도 IPC 인 것이다.

#### 예제 4)

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

```
int main(void){
```

```
    unsigned short host_port = 0x5678; //2 바이트
```

```
    unsigned short net_port;
```

```
    unsigned long host_addr = 0x87654321; //4 바이트
```

```
    unsigned long net_addr;
```

```
    net_port = htons(host_port); //host to network short 2 바이트
```

```
    net_addr = htonl(host_addr); //host to network long 4 바이트
```

```
    printf("Host Ordered Port: %#x\n", host_port);
```

```
    printf("Network Ordered Port: %#x\n", net_port); //리틀엔디안은 크로스매칭, 2 바이트씩 읽음
```

```
    printf("Host Ordered Address: %#x\n", host_addr);
```

```
    printf("Network Ordered Address: %#x\n", net_addr);
```

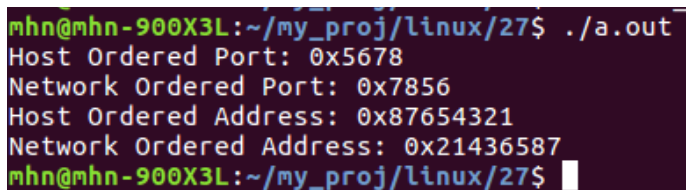
```
    return 0;
```

```
}
```

//이걸 하는 이유? cpu 마다 엔디안 정보가 다름

//통일된 포맷을 위해

//아니면 정보가 꼬일 수 있음



```
mhn@mhn-900X3L:~/my_proj/linux/27$ ./a.out
Host Ordered Port: 0x5678
Network Ordered Port: 0x7856
Host Ordered Address: 0x87654321
Network Ordered Address: 0x21436587
mhn@mhn-900X3L:~/my_proj/linux/27$
```

#### 엔디안이란?

컴퓨터에서 데이터가 저장되는 순서!

컴퓨터에서 데이터 저장은 byte 단위로 저장이 된다. 그런데 이 단위 저장을 할때 각 제조업체(CPU)에 따라서 저장이 되는 순서가 서로 다르다. 가장 낮은 바이트부터 저장을 하는 방식이 있고, 가장 높은 바이트부터 저장을 하는 방식이 존재한다. 전자를 Little Endian 이라고 하며, 후자를 Big Endian 이라고 한다.