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

* signal.c

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

void my_sig(int signo){
   printf("my_sig called\n");
}

void my_sig2(int signo){
   printf("my_sig2 called\n");
}

int main(void){
   void (*old_p)(int);
   void (*old_p2)(int);
   old_p = signal(SIGINT,my_sig);
   pause();
   old_p2 = signal(SIGINT,my_sig2);
   pause();
   old_p2 = signal(SIGINT,old_p); // or old_p2
   pause();
   for(;;)
          pause();
   return 0;
}
```

ctrl+c 3 번 누르면 종료된다.

old_p 를 old_p2 로 바꾸면 종료되지 않고 계속 my_sig called 가 반복된다.

signal()함수 인수는 시그널 번호, 시그널 처리할 핸들러
리턴값은 이전에 설정된 시그널 핸들러이다.

1. old_p 인 경우

처음 old_p 에는 이전에 설정된 핸들러가 없으므로 NULL 이 전달된다.
old_p2 에는 my_sig 가 전달되고, 다시 my_sig2 가 전달된다.
하지만 signal(SIGINT, old_p)에 의하여 ctrl+c 가 입력된 경우
old_p 가 NULL 이므로 종료된다.

2. old_p2 인 경우

old_p 는 nULL 저장
시그널 작동시 my_sig 작동

old_p2 는 my_sig 저장
시그널 작동시 my_sig2 작동

old_p2 는 my_sig2 저장
시그널 작동시 old_p2 작동(my_sig 작동)
여기서 좀 이상한 부분 old_p2 의 값이 변하지 않는다.
(시그널에 처음 넣어진 값을 상수처럼 취급 하는 것 같다.)
그 밑에 signal(SIGINT,old_p2);를 하면 시그널이
old_p2 가 작동한다.

* signal5.c

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

int main(void){

   signal(SIGINT, SIG_IGN);
   pause();
   return 0;
}
// $ps -ef
// $kill -2 pid
// kill -2 로 할 경우 죽지 않는다.
```

signal()함수에 ctrl+c 와 무시한다는 인자를 넣어주었다.
따라서 ctrl+c 를 입력하여도 무시가 된다.

pause()함수

설명

pause() 함수를 호출하면 시그널을 수신할 때까지 대기 상태로 빠집니다. 어떤 시그널이 발생하기 전까지 대기상태를 유지할 때 사용하면 요긴합니다.

헤더	unistd.h
형태	**int** pause(**void**);
반환	항상 -1을 반환하며 errno에는 ERESTARTNOHAND 로 설정됩니다.

시그널을 수신 할때까지 대기 상태로 빠지게 된다.
signal 상태가 변경되는 것이 있다면 pause 에서 벗어나게 된다.

* signal6.c

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

int main(void){
   int ret;
   char buf[1024]="fefeef";
   if((ret = read(0, buf, sizeof(buf))) > 0)
      goto err;

   return 0;


err :
   perror("read() :");
   exit(-1);
}
```

perror() 함수

기본적으로 현재 상태를 나타내 준다.
에러가 발생하면 에러 내용을 나타내고
정상적인 작동이면 정상작동 하였다고 나타내 준다.

```
* signal6_goto.c

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

void ggg(void){

   goto err;
}

int main(void){
   int ret;
   char buf[1024]="fefeef";
   if((ret = read(0, buf, sizeof(buf))) > 0)
          ggg();

   return 0;


err :
   perror("read() :");
   exit(-1);
}
```

컴파일 자체에서 오류 발생
label err 이 선언되지 않았다고 나온다.

goto 문 자체가 같은 함수 내에서 쓰여야 한다.
다른 함수에서 다른 함수로 라벨링이 불가능하다.

goto 가 스택을 해제할 수 있는 능력이 없다.

* set_jmp.c

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <setjmp.h>

jmp_buf env;

void test(void){
   longjmp(env,1);
}

int main(void){
   int ret;
   if((ret = setjmp(env)) == 0)
         test();
   else if(ret > 0)
         printf("error\n");
   return 0;
}
```

형식

```
#include <setjmp.h>
void longjmp(jmp_buf env, int value);
```

언어 레벨

ANSI

스레드세이프

예

설명

`longjmp()` 함수는 `setjmp()` 함수에서 이전에 *env*에 저장한 스택 환경을 복원합니다. `setjmp()`과 `longjmp()` 함수는 비로컬 이동을 수행하기 위한 방법을 제공합니다. 종종 신호 핸들러에서 사용됩니다.

`setjmp()` 함수에 대한 호출로 현재 스택 환경이 *env*에 저장됩니다. `longjmp()`에 대한 후속 호출은 저장 환경을 복원하고 `setjmp()` 호출에 해당한 프로그램에 대한 위치 제어를 리턴합니다. 처리는 `setjmp()` 호출이 바로 지정된 *value*을 리턴한 것처럼 다시 시작합니다.

제어를 수신하는 함수에 사용 가능한(레지스터 변수 제외) 모두 변수는 `longjmp()`이 호출되었을 때 가지고 있는 값을 포함합니다. 레지스터 변수의 값은 예측 불가능합니다. `setjmp()`에 대한 호출과 `longjmp()` 함수 사이에 변경된 비휘발성 auto 변수도 예측 불가능합니다.

> **참고**
>
> 해당하는 `longjmp()` 함수를 호출하기 전에 `setjmp()` 함수를 호출하는 함수가 리턴하지 않는지 확인하십시오. `setjmp()` 함수를 호출하는 함수가 리턴된 후 `longjmp()`을 호출하면 예측 불가능한 프로그램 작동이 발생합니다.

value 인수는 0(영)이 아닌 값이어야 합니다. *value*으로 0 인수를 지정하면, `longjmp()`은 해당 위치에서 1을 대체합니다.

	longjmp() 함수의 리턴 값은 없다. setjmp()의 리턴값은 longjmp 에서입력한 인자의 값이 튀어나온다.

* setjmp4.c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <setjmp.h>
jmp_buf env1;
jmp_buf env2;
void test1(void){
  int ret;
  if((ret = setjmp(env1)) == 0){
  test2();
  }
  else{
  printf("this is test1 func!\n");
  longjmp(env2,1);
  }
}
void test2(void){
  int ret;
  if((ret = setjmp(env2)) == 0){
  longjmp(env1,1);
  }
  else{
  printf("this is test2 func!\n");
  longjmp(env1,1);
  }
}
int main(void){
	test1 함수와 test2 함수를 계속 반복적으로 수행하는 코드
	
	setjmp() 함수는 전달된 인자에 현재 스택환경을 전달한다.
	초기 setjmp() 선언시 리턴값은 0 을 리턴한다.
	(goto 문의 라벨링 개념)
	
	longjmp()함수는 인자로 스택환경과 어떠한 value 를 받는다.
	인자로 전달된 스택환경으로 복귀함과 동시에 value 를 전달하여
	setjmp()의 리턴값으로 value 를 리턴하게 된다.

```
  test1();
  return 0;
}
```

*setjmp_lec.c

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

jmp_buf env1;
jmp_buf env2;

void test1(void)
{
        longjmp(env1, 1);
}

void test2(void)
{
        longjmp(env1, 2);
}

void test3(void)
{
        longjmp(env2, 1);
}

int main(void)
{
        int ret;
        if((ret = setjmp(env1)) == 0)
```

초기 setjmp(env1)의 리턴값은 0 이므로
this 를 출력하고 test1()함수 실행

test1()함수는 env1 이라는 스택환경으로 복귀하며 리턴값은 1 을 전달

1 을 전달받은 ret 은 else if 문을 통하여 1 을 출력하고
test2()함수를 실행

test2()함수는 env1 이라는 스택환경으로 복귀하며 리턴값은 2 를 전달.

2 를 전달받은 ret 은 else if 문을 통하여 2 를 출력하고 if - else 문을 탈출

그리고 setjmp(env2)를 통하여 현재 스택환경을 env2 에 저장한다.
second label 을 출력한 뒤에 test3()함수 실행.

test3()함수에서 env2 스택환경으로 복귀하며 1 을 전달한다.
ret 은 1 을 전달 받으며, else 문을 통하여 스택환경 env1 로 복귀하고
3 을 전달한다.

env1 스택환경에서 ret 은 3 을 전달 받으며 if-else 문에서
else 문으로 들어간다.
goto letsgo label 출력 뒤에 letsgo 라벨로 이동.

letsgo 라벨에서 err 라벨로 이동.

Error 출력뒤에 종료된다.

```
	{
		printf("this\n");
		test1();
	}
	else if(ret == 1)
	{
		printf("1\n");
		test2();
	}
	else if(ret == 2)
		printf("2\n");
	else
	{
		printf("goto letsgo label\n");
		goto letsgo;
	}

	if((ret = setjmp(env2)) == 0)
	{
		printf("second label\n");
		test3();
	}
	else
		longjmp(env1, 3);
letsgo:
		goto err;

	return 0;

err:
	printf("Error!!!\n");
	exit(-1);
}
```

```
* setjmp5.c

#include <fcntl.h>
#include <stdlib.h>
#include <setjmp.h>
#include <stdio.h>

jmp_buf env;

void test(void){
   int flag = -1;
   if(flag < 0){
          longjmp(env,1);
   }
   printf("call test\n");
}

int main(void){
   int ret;
   if((ret = setjmp(env)) == 0)
          test();
   else if(ret > 0)
          printf("error\n");
   return 0;
}
```

longjmp() 사용할 때 주의사항이 있다.

longjmp() 함수 이후에

아래에 입력된 코드는 의미가 없어진다.

따라서 잘 생각하고 코드를 짜야 한다.

* hackman.c

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

void my_sig(int signo){
   printf("You must insert coin\n");
   exit(0);
}

int main(void){
   char buf[1024];
   int ret;
   signal(SIGALRM, my_sig);
   alarm(3);
   read(0, buf, sizeof(buf));
   alarm(0);
   return 0;
}
```

alarm()함수 지정 시간에 SIGALRM 전송

설명

지정한 시간이 되면 자기 자신에게 SIGALRM 시그널을 전송합니다. 프로세스는 SIGALRM을 받으면 종료되므로 이를 유의하셔야 합니다.
아래 예제 1을 보시면 3초 후에 프로그램이 종료되는 것을 보실 수 있습니다.

헤더	unistd.h
형태	**unsigned int** alarm(**unsigned int** seconds);
인수	**unsigned int** secondsseconds 초 후에 시그널 전송
반환	양수 이전에 alarm() 호출로 시그널이 발생하기 까지 남은 시간 0 이전에 설정된 알람이 없다.

예제 1

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

int main(void)
{
  alarm(3);
  while(1)
    ;
}
```

인자로 second 를 받는다. 인자로 받은 시간이 지나가면 SIGALRM 을 활성화 시킨다.

signal 함수에서 SIGALRM 이 활성화 되면 my_sig 가 실행 되도록 설정하였으므로, 3 초가 지나도록 입력이 되지 않으면
you must insert coin 을 출력하고 프로그램이 종료된다.

alarm(0)을 해줘야 알람이 초기화 된다!!!!!!!
따라서 적절한 입력을 받게 되었다면 꼭 alarm(0)을 통하여 초기화를 해줘야

	나중에 꼬이지 않게 된다!

*quizquiz.c
 	
#include <stdio.h>
#include <signal.h>
#include <fcntl.h>
#include <stdlib.h>
#include <time.h>
#include <setjmp.h>
 	
jmp_buf env;
int cnt;
 	
void my_sig(int signo){
  printf("You must insert number!!!\n");
  exit(0);
}
 	
void quiz(int random, int ans){
  cnt++;
  if(random == ans){
  printf("goooddddddd!\n");
  longjmp(env,2);
  }
	깜짝 퀴즈게임
	
	내가 작성한 코드.
	
	1~100 까지의 랜덤 값을 받고
	
	8 고개 게임을 하는 것이다.
	
	주의할 점은 alarm 의 사용이다.
	alarm 사용후에 원하는 입력이 발생하였다면,
	alarm(0)으로 초기화 하는 것이 좋다.

```
   else if(random > ans){
         printf("random is bigger than your answer!\n");
         longjmp(env,1);
   }
   else if(random< ans){
         printf("random is less !\n");
         longjmp(env,1);
   }

}
int main(void){
   char buf[1024];
   int ret;
   int random;
   int ans;
   srand(time(NULL));
   random = rand()%100+1;

   if((ret = setjmp(env)) == 0){
         scanf("%d",&ans);
         quiz(random, ans);
   }
   else if(ret == 1){
         if(cnt >8){
            printf("you are idiot!!!!\n");
            return 0;
         }
         signal(SIGALRM, my_sig);
         alarm(3);
         scanf("%d",&ans);
         quiz(random,ans);
   }
```

```
  else{
        return 0;
  }
}
```

*quiz_ans.c

```
#include <time.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <stdbool.h>

void sig_handler(int signo)
{
        printf("You lose! Input should be within 1 second!\n");
        exit(0);
}

void make_game(int *data)
{
        *data = rand() % 100 + 1;
}

bool check_correct(int data, int cmp)
{
        if(data == cmp)
                return true;
        else
                return false;
}
```

make_game()
함수를 통하여 랜덤 변수를 생성해 준다.

check_correct()
함수를 통하여 사용자가 입력한 데이터와 랜덤 변수가 같은지 아닌지
확인하여 T 나 F 값을 리턴한다.

stat_game()함수를 통하여
10 번 반복이 되며
check_correct 함수를 통하여 값이 같은지 판단한다.

다르다면 값이 크고 작음을 통해 사용자에게 알린다.

```
void start_game(int data)
{
	char buf[32] = {0};
	bool fin;
	int i, cmp;

	for(i = 0; i < 10; i++)
	{
		signal(SIGALRM, sig_handler);
		printf("숫자를 맞춰봐!\n");
		alarm(1);
		read(0, buf, sizeof(buf));
		alarm(0);
		cmp = atoi(buf);

		fin = check_correct(data, cmp);

		if(fin)
		{
			printf("You Win!!!\n");
			exit(0);
		}
		else
		{
			if(data > cmp)
				printf("%d 보다 크다\n", cmp);
			else
				printf("%d 보다 작다\n", cmp);
		}
	}

	printf("You Lose!!! You Babo\n");
}
```

```
int main(void)
{
	int data;

	srand(time(NULL));
	make_game(&data);

	start_game(data);

	return 0;
}
```