Embedded System Software
HW #1

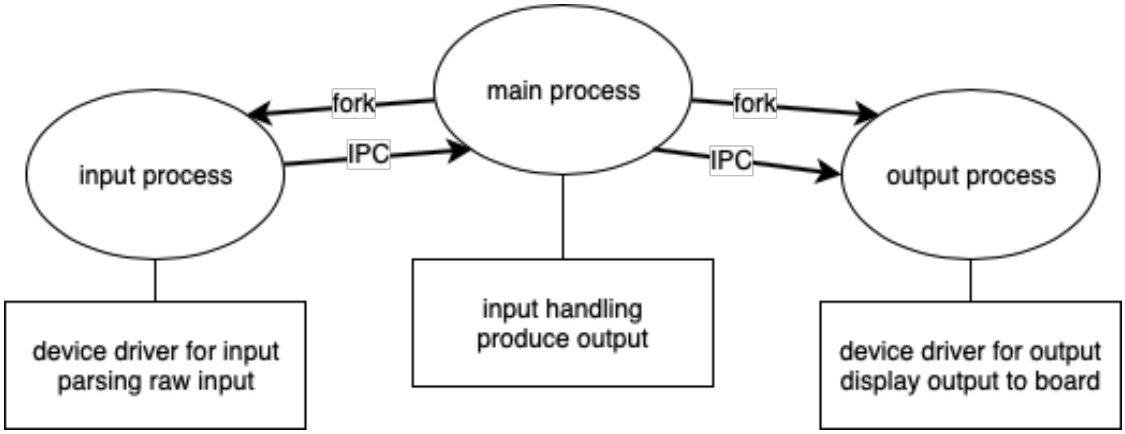
20151623
한상구

1. 프로젝트 목표

디바이스 컨트롤과 IPC 를 이용하여 주어진 Clock, Counter, Text editor, Draw board 를 구현한다.

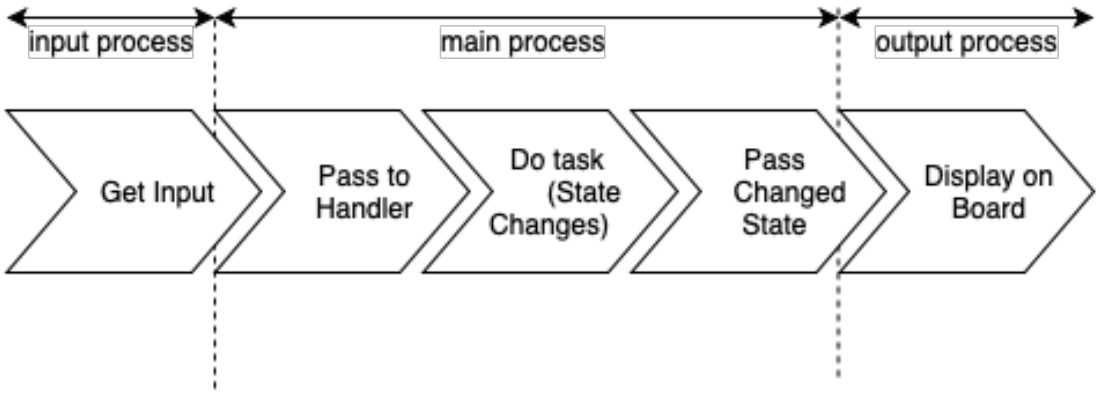
2. 설계

a. Project structure



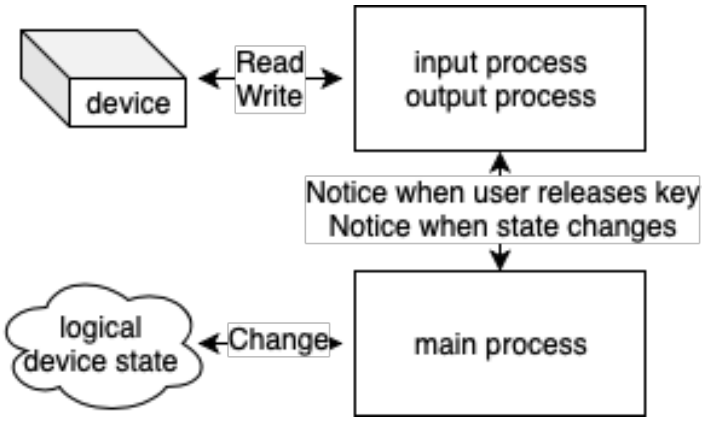
<그림 1 - 전제적인 프로젝트 구조도, 프로세스 중심>

프로젝트 명세서에서 명시하고 있듯이, main process 가 parent process 가 되고, input/output process 는 main process 가 fork 하여 생성되는 child process 가 되는 구조입니다.
message-queue 를 통해 프로세스간 통신을 진행하며, 양방향 통신이 아닌 input process -> main process -> output process 로 이어지는 단방향 통신을 하도록 설계하였습니다.

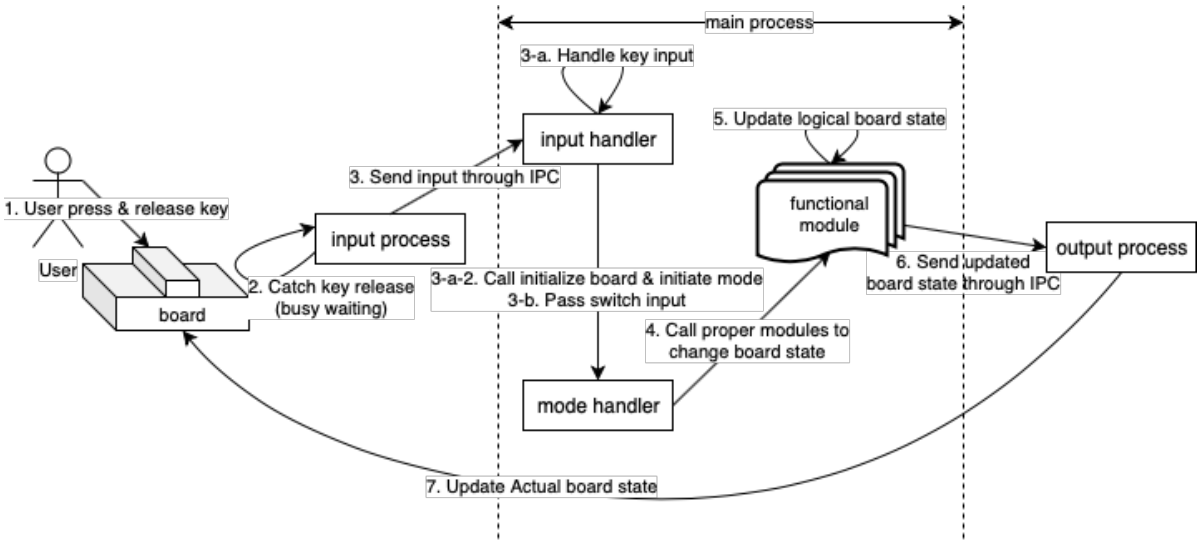


<그림 2 - 전체적인 프로젝트 흐름도>

프로세스별 큰 기능 분담 이후, 프로젝트에서 구현해야 할 기능의 흐름을 구체화 및 세분화했습니다.
크게 사용자의 입력을 받는 것 (input process), 입력 값을 적절한 핸들러에게 넘기고, 그 값에 맞게 보드의 상태를 변경하고, 출력을 위해 변경된 상태를 전달하는 것 (main process), 전달받은 상태를 보드에 표시하는 것 (output process) 으로 생각할 수 있었습니다.
조금 더 깊게 설계를 설명하자면, Synchronous Worker 와 비슷한 느낌으로 전체적인 프로젝트를 설계했습니다. 아래 그림 3, 4 와 함께 기술하겠습니다.



<그림 3 - device, process 간의 관계도>



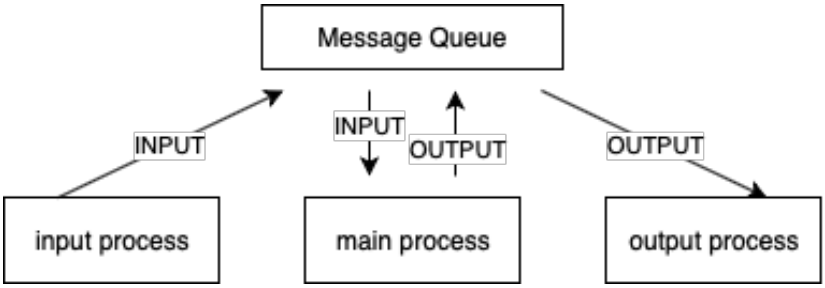
<그림 4 - 프로젝트 흐름도>

본 프로젝트를 진행하는데 있어 필요한 device control 을 정리, atomic 하게 쪼개고 각 atomic operation 에 대해 module 화하는 작업을 우선 진행했습니다. 본 작업을 통해 각 device 에 대해 atomic operation 을 진행할 수 있는 functional module 을 작성했습니다 (프로젝트 내의 operation.c, operation.h 참조).

mode handler 는 현재 모드와 주어진 입력값에 알맞는 board state change 를 위해 위 functional module 을 적절하게 사용하여 간편하게 작성할 수 있었으며 (프로젝트 내의 modes.c, modes.h 참조) 이후 설명할 Message queue 를 통해 input -> main -> output 으로 이어지는 프로세스간 통신을 구현했습니다.

각 프로세스의 설계 및 IPC 의 설계는 이어서 기술하도록 하겠습니다.

b. IPC (Message Queue) (message.c)



<그림 4 - Message queue 도식화, 화살표 위 텍스트는 Message 의 mtype>

Message Queue 를 이용하여 IPC 를 구현하였고, 위 도식화된 그림에서 확인할 수 있듯이 단방향으로 프로세스간 통신이 이루어 지는 것을 확인하실 수 있습니다.

IPC 를 위한 wrapper 를 작성한 message.h 의 코드를 이용해 상세 구현 방법을 기술하겠습니다. (함수 구현은 message.c 에서 확인할 수 있으며, msgsnd(), msgrcv() 외엔 msg_t 에 dtype, mtype 등을 설정해주는 것이 전부이기 때문에 인용하지 않았습니다)

```
/*
 * Message types
 */
#define INPUT         1
#define OUTPUT        2

/*
 * Data types
 */
#define DATA_INPUT    0
#define DATA_FND      1
#define DATA_LCD      2
#define DATA_DOT      4
#define DATA_LED      8
#define DATA_TERM     16

/**
 * struct msg_t - used by processes to communicate w/ message queue
 * @mtype: type for current message
 * @dtype: type for data which current message contains
 * @msg: data which current message contains
 */
typedef struct msg_t {
        long mtype;
        int dtype;
        char msg[MSG_LEN];
} msg_t;
```

Message queue 사용을 위한 자료형 및 플래그를 선언한 부분입니다. mtype 을 통해 어떤 프로세스가 해당 메시지를 가져갈 것인지 명시하며, dtype 을 통해 해당 메시지가 어떤 종류의 데이터를 가지고 있는지 명시하도록 했습니다.

```c
/**
 * get_message_qid - get message queue id
 */
int get_message_qid();
/**
 * enqueue_message - enqueue message with given data
 * @qid: queue id of message queue
 * @mtype: type for message
 * @dtype: type for data which current message contains
 * @msg: data which message will contain
 */
int enqueue_message(int qid, long mtype, int dtype, char *msg);
/**
 * receive_message - receive message with given mtype
 * @qid: queue id of message queue
 * @mtype: type for message
 * @msg: pointer to store receive message
 */
int receive_message(int qid, long mtype, msg_t *msg);
```

Message queue 사용을 위한 함수를 선언한 부분입니다.

`get_message_qid();` 는 `msgget()`을 통해 본 프로그램이 사용하는 message queue 의 id 를 가져올 수 있도록 하는 함수이며, `int enqueue_message(int qid, long mtype, int dtype, char *msg);` 를 통해 메시지의 mtype, dtype 을 설정한 뒤 `msgsnd()`를 통해 enqueue 할 수 있도록 하는 함수입니다.

`int receive_message(int qid, long mtype, msg_t *msg);`를 통해 Message queue 에서 `msgrcv()`를 통해 parameter 로 넘긴 mtype 에 해당하는 메시지를 가져올 수 있도록 함수입니다.

각 프로세스에서 그림 4 와 같이 적절하게 mtype 을 설정, 위 함수들을 이용하여 자신의 역할에 맞는 정보를 가져오고 보낼 수 있도록 설계 및 구현하였습니다.

c. Input Process (process.c - input_process())

key pressed & relased 를 지속적으로 확인하며 입력값이 있을 경우 main process 로 넘겨주기 위한 프로세스입니다.

PROG, BACK 등 key 를 받기위한 device driver (이하 key input)와 SWITCH1-9 를 받기위한 switch device driver (이하 switch input)를 이용합니다.

키가 입력되었다는 사실을 main process 로 알리는 시기는 모든 키가 release state 가 되었을 때 입니다. 그 이전까지 눌린 key 는 전부 기록되며, 모두 release state 가 되기 전까지 눌린 key 들을 main process 로 보내줍니다 (`enqueue_message(qid, INPUT, msg)` 이용) - 이를 통해 누른 시간과 관계없이 한 번만 눌렀다고 판단할 수 있도록 설계했습니다.

본 프로젝트에서 두 개 이상의 키 입력이 동시에 들어오는 경우는 switch input 뿐이기 때문에, 본 상황만 고려한 채로 input handling 이 이뤄집니다. (switch input 중 key input 불가능)

key input 의 경우 `msg[KEY_ENV]`에 input 을 넣고 (`ev[0].code` 그대로 넣음, 해당 값은 globals.h 에 define 되어있음), switch input 의 경우 `msg[FLAG_SWITCH]`에 1, `msg[0]` ~ `msg[8]`에 각각 SWITCH1 ~ SWITCH9 의 값이 들어가도록 했습니다 (눌렸으면 1, 눌리지 않았으면 0).
본 프로세스의 로직은 process.c 의 `input_process()`에 구현하였습니다.

d. Output Process (process.c - output_process(), device.c)

logical board state 를 받아 실제 board state 를 device driver, mmap 을 통해 업데이트 하기위한 프로세스입니다. 실제 board 에 접근해야하기 때문에 driver 들을 관리하며, message queue 를 통해 주고받는 메시지의 structure 인 `msg_t` 에서 `dtype` 을 통해 받은 메시지의 데이터가 어떤 device 의 state 를 바꿔야 하는지 확인할 수 있습니다.
위 정보를 이용해 logical board state 가 업데이트 될 때 마다 전달되는 메시지를 처리하며 적절한 driver 에 state overwrite 작업을 진행합니다.
본 프로세스의 로직은 process.c 의 `output_process()`, driver 연결, 사용등의 로직은 device.c 에 구현하였습니다.

e. Main Process - input handler (main.c)

input process 에서 message queue 를 통해 보낸 입력값을 일차적으로 처리하는 핸들러입니다.
switch input 을 받을 경우, 눌린 스위치를 bit-masking 하여 현재 모드와 함께 mode hanlder 에게 입력값으로 넘깁니다 (SWITCHn 이 눌렸을 경우 LSB 로부터 n-th bit 가 1 이 되도록 bit-masking).
key input 을 받을 경우, 모드의 변경 혹은 프로그램을 종료하는 상황을 처리합니다.
모드가 변경되는 경우 `initialize_board()`를 이용해 logical board state 를 초기화 한 다음, 모드를 변경한 뒤 `initiate_mode(mode)`를 이용해 변경된 모드의 초기 상태로 업데이트 합니다 (그림 4 에서 3-a-2 순서) - 이를 통해 모드 변경 시 항상 모드의 초기 상태가 돌아옴을 보장할 수 있습니다.
프로그램이 종료되는 경우, input process 를 `kill(input_pid, SIGKILL)`을 통해 종료시키고, `initialize_board()`를 통해 보드를 초기화 한 다음 `DATA_TERM` 타입의 메시지를 output process 에게 보내 열려있는 driver 를 모두 닫고 종료하도록 합니다. 이후 IPC 를 위해 생성하였던 message queue 를 삭제한 뒤 main process 를 종료합니다.

f. Main Process - mode handler (modes.c)

input handler 가 현재 모드와 입력 값을 넘겨주면, 이에 맞는 board state 를 만들기 위해 적절한 fuinction module 을 조합하여 호출하는 핸들러입니다.
input handler 에서 넘겨주는 입력 값은 switch input 을 bit-masking 한 값이기 때문에,

```
/*
 * Bit masking for switch input
 */
#define SW_1   0x00000001
#define SW_2   0x00000002
#define SW_3   0x00000004
#define SW_4   0x00000008
#define SW_5   0x00000010
#define SW_6   0x00000020
```

```
#define SW_7   0x00000040
#define SW_8   0x00000080
#define SW_9   0x00000100
```

와 같이 미리 define 해둔 값을 이용하면 `switch (input)` 을 이용하여 쉽게 분기를 나눌 수 있습니다 (예를 들어 2 번과 3 번 스위치를 누른 경우는 `case SW_2 | SW_3`).

각 모드를 담당하는 함수는 `void mode##num(int input)` 형식이며 (num 은 1~5), 이 함수들은 함수포인터인 `static mode_func_ptr modes[5]`로 묶어서 관리하도록 했습니다.

이 외에 1 번모드, 4 번모드에서 사용되는 주기적으로 상태가 변하는 기능을 구현하기 위해 `alarm()`과 `void periodic_control(int signo)`을 이용하였는데, systemcall interrupt 를 통한 주기적인 board state update 를 구현할 수 있었습니다.

각 함수에 관해 어떤 function module 을 조합하여 구현하였는지는 **3. 기능구현**에서 다루도록 하겠습니다.

g. Main Process - functional modules (operation.c)

main process 에서 처리하는 작업을 logical board state 에 진행, state 를 업데이트하고 업데이트 된 결과 state 를 output process 로 보내 실제 board state 에 업데이트 될 수 있도록 하는 함수들입니다.

실제 구현은 간단하므로, 각 device 에 대해 어떤 atomic operation 이 있었는지 정리 및 특징만 서술하도록 하겠습니다 (구현은 operation.c 에서 확인하실 수 있습니다).

```
/*
 * Options for fnd operation
 */
#define FND_INCREASE         0
#define FND_SET_BOARD_TIME   1
#define FND_ADD_MINUTE       2
#define FND_ADD_HOUR         3
#define FND_ADD_BASE         4
#define FND_ADD_SQUARE       5
#define FND_CHANGE_BASE      6
#define FND_RESET            7
```

먼저 FND 와 관련된 작업들입니다. 수를 1 증가시키거나 (Counter-SW(4), Text editor-Any SW, Draw board-Any SW), 표시 시간을 보드의 시간으로 바꾸거나 (Clock-SW(2)), 분, 시간을 더하거나 (Clock-SW(3),(4)), 십의 자리($base$)를 1 증가시키거나 (Counter-SW(3)), 백의 자리($base^2$)를 1 증가시키거나 (Counter-SW(2)), 진수를 바꾸거나 (Counter-SW(1)), 초기화하는 (Change mode) 작업이 필요했습니다.

`static int fnd_curr` 을 이용해 현재 FND 에 표시되는 숫자를 관리하였고, 적절한 모듈러 연산을 통해 진수변환에도 안전하게 작동할 수 있도록 하였습니다.

```
/*
 * Options for lcd opertaion
 */
#define LCD_RESET            0
```

```
#define LCD_ADD_CHAR          1
#define LCD_REPLACE           2
```

다음은 LCD 와 관련된 작업들입니다. LCD 를 비우거나 (Change mode, Text editor-SW(2,3)), 글자를 추가하거나 (Text editor), 마지막 글자를 바꾸는 (Text editor) 작업이 필요했습니다.
현재 LCD 에 올라가 있는 문자열을 관리하며, `MAX_WIDTH` 를 넘어가는 문자열의 경우에는 처음 들어왔던 문자부터 버리는 형식으로 관리하도록 하였습니다.

```
/*
 * Options for dot matrix operation
 */
#define DOT_PRINT_INPUT_MODE 0
#define DOT_RESET            1
#define DOT_CURSOR_UP        2
#define DOT_CURSOR_SHOW      3
#define DOT_CURSOR_LEFT      4
#define DOT_SELECT           5
#define DOT_CURSOR_RIGHT     6
#define DOT_CLEAR            7
#define DOT_CURSOR_DOWN      8
#define DOT_REVERSE          9
#define DOT_CURSOR_HIDE      10
#define DOT_PRINT_1          11
#define DOT_PRINT_2          12
#define DOT_PRINT_3          13
```

다음은 DOT MATRIX 와 관련된 작업들입니다. 현재 입력모드를 출력하거나 (Text editor), Dot matrix 를 비우거나 (Change mode, Draw board-SW(1)), 커서 위치를 조정하거나 (Draw board-SW(2),(4),(6),(8)), 커서 위치의 Dot matrix 를 반전시키거나 (Draw board-SW(5)), 전체 Dot matrix 를 반전시키거나 (Draw board-SW(9)), 커서를 표시/숨기거나 (Draw board-SW(3)), 숫자를 출력하는 (Mode 5) 기능이 필요했습니다.
각 device 가 logical state 를 들고 있지만, Dot matrix 같은 경우 커서를 구현하기 위해 레이어를 두개로 나눠서 작업했습니다. 하나의 레이어는 실제로 선택된 값을 들고 있고 (이하 selected layer), 다른 레이어는 커서의 위치만 표시되어 있습니다 (이하 cursor layer). 두 레이어를 XOR 하여 Dot matrix 의 최종 logical state 를 구할 수 있도록 구현했습니다. (cursor layer 는 커서의 위치를 제외하곤 모두 0 이기 때문에 XOR 시 커서의 위치를 제외하고는 변화가 없기 때문입니다 + 커서를 숨기는 경우 모든 element 가 0 이 되기 때문에 selected layer 가 곧 logical state 가 됩니다)

```
/*
 * Options for led operation
 */
#define LED_RESET            0
```

```
#define LED_SET             1
```

마지막으로 LED 와 관련된 작업들입니다. 모든 모드에서 사용되지만 끄거나 켜는 작업 외에 복잡한 작업은 없기 때문에 LED 를 모두 끄거나, bit-masking 된 입력값 (MSB 로부터 n-th bit 가 n 번째 LED 의 state)에 따라 LED 를 켜는 기능만이 필요했습니다.

위 functional module 을 적절하게 조합하여 본 프로젝트에서 원하는 기능을 구현하였고, 이는 이어서 기술하도록 하겠습니다.

3. 기능 구현

a. Mode 1 (Clock)

```
case 0:
        /*
         * mode 1 - clock
         */
        control_fnd(FND_SET_BOARD_TIME);
        control_led(LED_SET, 1 << 7);
        alarm_flag = FLAG_MODE_1 | FLAG_BOARD_TIME;
        alarm(5);
        break;
```

1 번 모드의 초기 설정을 위해 사용된 function module 입니다 (modes.c - `initiate_mode()`에 위치한 코드입니다). FND 에 현재 보드의 시간을 출력하고, 1 번째 LED 를 켠 뒤 5 초마다 한 번 보드의 시간으로 업데이트 할 수 있도록 `alarm_flag` 에 `FLAG_BOARD_TIME` 를 세팅하고, alarm()을 통해 `periodic_control` 을 호출했습니다.

```
switch (input) {
case SW_1:
        if (alarm_flag & FLAG_BOARD_TIME) {
                alarm_flag ^= FLAG_BOARD_TIME;
        }
        alarm_flag ^= FLAG_BLINK;
        control_led(LED_SET, alarm_flag & FLAG_BLINK ? 1 << 5 : 1 << 7);
        sec = 0;
        alarm(1);
        break;
case SW_2:
        if (alarm_flag & FLAG_BLINK) {
                control_fnd(FND_SET_BOARD_TIME);
        }
        break;
case SW_3:
        if (alarm_flag & FLAG_BLINK) {
```

```
                control_fnd(FND_ADD_HOUR);
        }
        break;
case SW_4:
        if (alarm_flag & FLAG_BLINK) {
                control_fnd(FND_ADD_MINUTE);
        }
        break;
default:
        break;
}
```

1 번 모드에서 입력이 주어졌을 때 적절한 동작을 하기 위해 조합한 function module 입니다. SW(1),(2),(3),(4)가 아닌 다른 입력에 대해서는 아무런 동작도 하지 않으며, SW(2),(3),(4)에 대해서는 SW(1)이 눌린 상태에서만 동작하도록 작성했습니다.

SW(1)이 한번도 눌리지 않은 상태라면 보드의 시간을 실시간으로 가져와야 하지만, 한 번이라도 눌린 경우 수정 완료 이후부터 60 초에 1 분씩 증가시켜야 하기 때문에 `alarm_flag` 에서 `FLAG_BOARD_TIME` 를 제거하는 것을 확인할 수 있습니다.

`alarm_flag` 에서 `FLAG_BOARD_TIME` 이 세팅되어 있는 경우 LED 는 3, 4 번이 1 초를 주기로 깜빡거리며, 사용자가 값을 증가시키지 않는 한 시간의 변동은 없게 됩니다.

b. Mode 2 (Counter)

```
case 1:
        /*
         * mode 2 - counter
         */
        control_led(LED_SET, 1 << 6);
        break;
```

2 번 모드의 초기 설정을 위해 사용된 function module 입니다 (modes.c - `initiate_mode()`에 위치한 코드입니다). 초기 상태는 10 진수로 설정되어 있기 때문에 2 번 LED 만 켜진 상태로 모드를 시작합니다.

```
switch (input) {
case SW_1:
        control_fnd(FND_CHANGE_BASE);
        break;
case SW_2:
        control_fnd(FND_ADD_SQUARE);
        break;
case SW_3:
        control_fnd(FND_ADD_BASE);
        break;
```

```
case SW_4:
        control_fnd(FND_INCREASE);
        break;
default:
        break;
}

base = get_fnd_base();
control_led(LED_SET, 1 << (8 - base));
```

2 번 모드에서 입력이 주어졌을 때 적절한 동작을 하기 위해 조합한 function module 입니다. SW(1),(2),(3),(4) 외의 입력은 무시하며, 각 입력에 대한 function module 은 위와 같이 굉장히 단순합니다. 입력 이후 진수가 바뀌었을 수 있기 때문에 logical state 에서 진수 정보를 가져와 LED state 를 업데이트 하는 부분을 보실 수 있습니다.

c. Mode 3 (Text editor)

```
case 2:
        /*
         * mode 3 - text editor
         */
        control_dot(DOT_PRINT_INPUT_MODE);
        break;
```

3 번 모드의 초기 설정을 위해 사용된 function module 입니다 (modes.c - `initiate_mode()`에 위치한 코드입니다). 초기 상태는 영문 입력 상태이므로 Dot matrix 에 영문 입력 상태임을 출력합니다.

```
int cur, i;

control_fnd(FND_INCREASE);

switch (input) {
case SW_1:
case SW_2:
case SW_3:
case SW_4:
case SW_5:
case SW_6:
case SW_7:
case SW_8:
case SW_9:
        for (i = 0; i < 9; i++) {
                if ((input >> i) & 1) {
                        cur = i;
                }
```

```
                }
                if (numeric) {
                        control_lcd(LCD_ADD_CHAR, '1' + cur);
                } else {
                        if (prev == cur) {
                                cnt = (cnt + 1) % 3;
                                control_lcd(LCD_REPLACE, text_pad[cur][cnt]);
                        } else {
                                cnt = 0;
                                control_lcd(LCD_ADD_CHAR, text_pad[cur][cnt]);
                        }
                        prev = cur;
                }
                break;
        case SW_2 | SW_3:
                prev = -1;
                control_lcd(LCD_RESET, 0);
                break;
        case SW_5 | SW_6:
                numeric ^= 1;
                control_dot(DOT_PRINT_INPUT_MODE);
                break;
        case SW_8 | SW_9:
                control_lcd(LCD_ADD_CHAR, ' ');
                break;
        default:
                break;
        }
```

3 번 모드에서 입력이 주어졌을 때 적절한 동작을 하기 위해 조합한 function module 입니다.
본 모드에서 사용되지 않는 조합의 스위치이더라도 사용자가 누른다면 스위치 누른 횟수를 증가시켜야 하므로, 제일 먼저 FND 의 숫자를 증가시키도록 했습니다.
영문 입력 상태인 경우 편리함을 위해 각 스위치에 해당하는 3 자리 글자를 아래와 같이 선언해 두었고, 이를 참조하는 식으로 구현하였습니다.

```
static char text_pad[9][4] = {
        ".QZ", "ABC", "DEF", "GHI", "JKL",
        "MNO", "PRS", "TUV", "WXY"
};
```

SW(1)~(9)에 대해 이전에 눌렀던 스위치가 또 입력이 되었다면 다음 글자로 치환, 새로운 스위치가 눌렸다면 새 글자를 추가하도록 functional module 을 이용했습니다. (LCD 가 표현 가능한 최대 글자가 넘어가는 경우 등은 functional module 에서 핸들링)
이외에 인풋에 대해서도 적절한 functional module 을 이용, 프로젝트에서 요구한 기능을 적절히 수행할 수 있도록 구현했습니다.

d. Mode 4 (Draw board)

```
case 3:
        /*
         * mode 4 - draw board
         */
        control_dot(DOT_CURSOR_SHOW);
        alarm_flag = FLAG_MODE_4;
        alarm(1);
        break;
```

4 번 모드의 초기 설정을 위해 사용된 function module 입니다 (modes.c - `initiate_mode()`에 위치한 코드입니다). 빈 화면에 커서가 깜빡거리고 있어야 하므로 커서가 보이도록 세팅하고, `alarm_flag` 에 `FLAG_MODE_4` 를 세팅한 뒤 `periodic_control` 을 이용해 커서가 깜빡거릴 수 있도록 구현했습니다.

```
int i;

control_fnd(FND_INCREASE);

switch (input) {
case SW_1:
case SW_2:
case SW_4:
case SW_5:
case SW_6:
case SW_7:
case SW_8:
case SW_9:
        for (i = 0; i < 9; i++) {
                if ((input >> i) & 1) {
                        control_dot(i + 1);
                }
        }
        break;
case SW_3:
        alarm_flag ^= FLAG_MODE_4;
        if (alarm_flag) {
                control_dot(DOT_CURSOR_SHOW);
                alarm(1);
        } else {
                control_dot(DOT_CURSOR_HIDE);
        }
        break;
default:
```

```
        break;
    }
```

4 번 모드에서 입력이 주어졌을 때 적절한 동작을 하기 위해 조합한 function module 입니다. 스위치를 누른 횟수를 세기 위해 FND 의 숫자를 증가시키고, 각 스위치에 맞는 function module 을 사용했습니다. (**2.g Main process - function modules (operation.c)** 에서 확인하실 수 있듯이, 명세서에서 지시한 사항에 맞춰 define numbering 을 하였습니다)
SW(3)의 경우 커서의 보임/숨김 기능이기 때문에 `periodic_control` 을 관리하는 mode handler 가 관여해야 했고, `alarm_flag` 에 `FLAG_MODE_4` 를 세팅/제거 하며 커서 관리를 하도록 구현했습니다.

e. Mode 5 (추가구현)

랜덤하게 나오는 LED 에 맞춰 스위치를 누르는 게임입니다. ((1)번, (2)번, (7)번 LED 에 불이 들어올 경우 SW(1), SW(2), SW(7)을 동시에 눌러야 게임을 종료합니다)

- FND : 게임 중 스위치를 누른 횟수를 출력한다. **초기상태는 0000.**
- LED : 게임의 정답에 맞게 불이 들어온다. 정답이 1, 2, 7 이라면 (1)번, (2)번, (7)번 LED 에 불이 들어온다. 게임 중이 아닐 때에는 모든 불이 꺼져있다. **초기 상태는 모든 LED 가 꺼진 상태.**
- LCD : 게임의 정답에 맞게 스위치를 눌렀을 때 CONGRATS! 라는 문자열을 출력한다. 게임 종료 화면이 아닐 때에는 빈 화면이다. **초기 상태는 빈 화면.**
- Dot Matrix : 게임 시작 시 3 초를 카운트하고 (1 초를 주기로 3, 2, 1 을 차례대로 출력한다), 게임 중에는 모든 불이 켜져있다. 게임 종료 화면이거나 게임중이 아닐 때에는 모든 불이 꺼져있다. **초기 상태는 빈 화면.**
- SW : 게임 시작과 리셋, 게임 중 정답 입력을 받는 버튼. **초기 상태는 게임 시작 전 상태.**

- 게임 시작 전

- SW(5) : 게임을 시작하는 버튼. 누르면 Dot Matrix 에서 1 초를 주기로 3, 2, 1 을 출력하고 게임을 시작한다.

- 게임 중

- SW(1)~SW(9) : 정답 입력 버튼.

- 게임 종료 화면

- SW(9) : 게임을 리셋하는 버튼. 게임 도중, 혹은 게임 종료 이후 누르면 초기 상태로 돌아간다.

`initialize_board()`이후 따로 세팅이 필요한 device 가 없기 때문에 초기 상태 세팅을 위한 function module 사용은 없습니다.

```
if (ingame) {
        control_fnd(FND_INCREASE);
}

switch (input) {
case SW_5:
        if (!ingame) {
```

```
                ingame = 1;
                control_dot(DOT_PRINT_3);
                sec = 2;
                alarm_flag |= FLAG_MODE_5;
                alarm(1);
        }
        break;
case SW_9:
        ans = 0;
        ingame = 0;
        control_led(LED_RESET, 0);
        control_dot(DOT_RESET);
        control_lcd(LCD_RESET, 0);
        control_fnd(FND_RESET);
        break;
default:
        break;
}

if (ingame && input == ans) {
        control_dot(DOT_REVERSE);
        control_lcd(LCD_ADD_CHAR, 'C');
        control_lcd(LCD_ADD_CHAR, 'O');
        control_lcd(LCD_ADD_CHAR, 'N');
        control_lcd(LCD_ADD_CHAR, 'G');
        control_lcd(LCD_ADD_CHAR, 'R');
        control_lcd(LCD_ADD_CHAR, 'A');
        control_lcd(LCD_ADD_CHAR, 'T');
        control_lcd(LCD_ADD_CHAR, 'S');
        control_lcd(LCD_ADD_CHAR, '!');
}
```

`ingame` 플래그를 통해 현재 게임 진행중이면 FND_INCREASE 를 통해 스위치 입력 횟수를 증가시킵니다. 게임 시작 시 3 초간 대기시간이 있는데, 이 때 Dot matrix 에서 남은 시간을 출력하기 위해 `periodic_control` 를 이용했습니다. 본 코드는 아래에서 설명하겠습니다.

게임 진행 중, 입력값이 정답과 같으면 Dot matrix 를 비우고 (게임중에는 꽉 찬 상태이기 때문에 DOT_REVERSE 를 통해 빈 화면을 출력하게 함) LCD 에 CONGRATS!라는 문자열을 출력하게 했습니다.

LED 는 (8)번까지 밖에 없기 때문에, SW(9)를 리셋 스위치로 사용했습니다.

```
if (alarm_flag & FLAG_MODE_5) {
        if (sec) {
                sec -= 1;
                control_dot(DOT_PRINT_1 + sec);
                alarm(1);
        } else {
```

```
                control_dot(DOT_RESET);
                control_dot(DOT_REVERSE);
                srand(time(NULL));
                do {
                        r = rand() % 8;
                        ans |= 1 << r;
                        ans_led |= 1 << (7 - r);
                } while (r % 2);
                control_led(LED_SET, ans_led);
        }
}
```

FLAG_MODE_5 를 통해 mode5 에서 호출 한 것이 확인되면, Dot matrix 를 통해 게임 시작까지 남은 시간을 출력하고, 이후엔 Dot matrix 를 가득 채우게 세팅, 랜덤 값을 정답으로 설정하고 이에 맞게 LED_SET 을 통해 led 를 세팅하는 도록 구현했습니다.