

Embedded System Software 과제 1

(과제 수행 결과 보고서)

과목명: [CSE4116] 임베디드시스템소프트웨어

담당교수: 서강대학교 컴퓨터공학과 박 성 용

학번 및 이름: 20171664, 이상윤

개발기간: 2024. 04. 04. -2024. 04. 12.

최종보고서

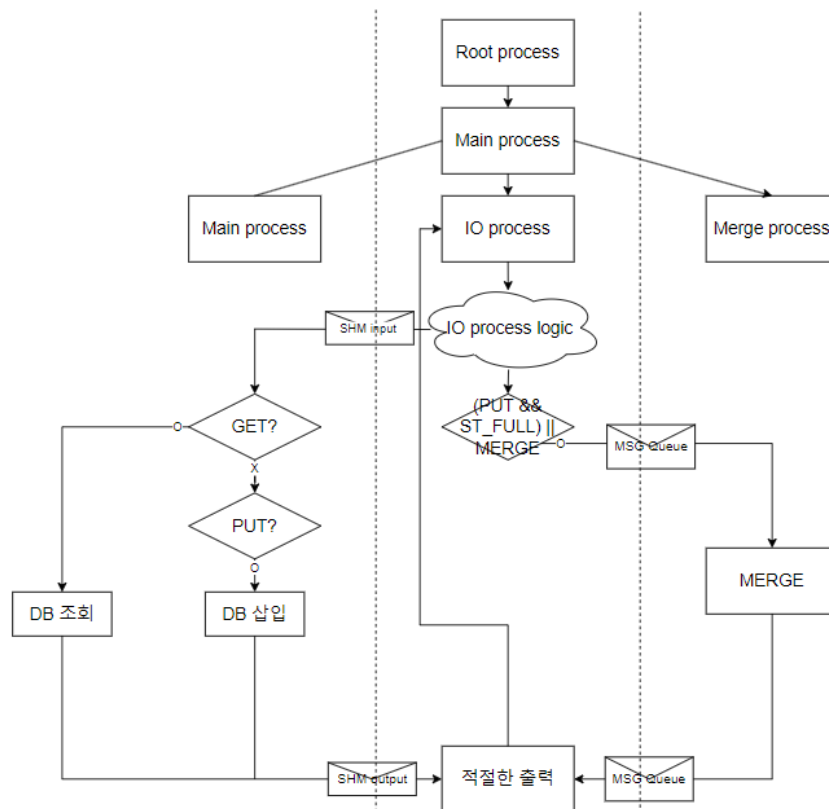
I. 개발 목표

아래의 세가지 목표를 달성하여 Simple Key-Value Store를 만드는 것을 최종 목표로 한다.

1. 멀티 프로세스 환경(main process, io process, merge process)에서 과제 명세에 맞도록 각 프로세스별 로직과 flow를 구현한다.
2. 적절한 shared memory, message queue, semaphore 구현으로 프로세스 간 통신을 가능하게 한다.
3. Device driver를 통해 device I/O 로직을 적절히 추상화하여 구현한다.

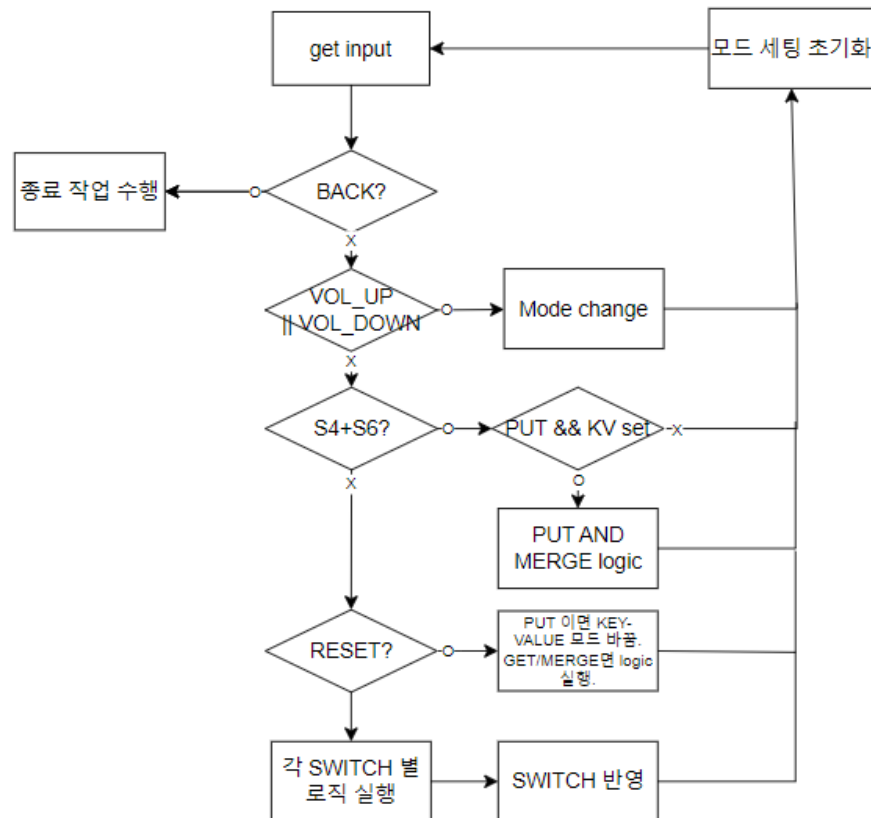
II. 개발 범위 및 내용

가. 개발 범위



<그림1: 프로젝트 플로우 차트>

1. <그림1>에서 점선은 main process, io process, merge process들의 범위를 나타낸다. IO process는 사용자가 입력한 값을 적절한 로직을 통해 받은 다음, 적절한 데이터를 shared memory를 통해 main process로 전달하거나 message queue를 통해 merge process로 전달하여야 한다. Main process와 merge process는 그 요청을 적절히 처리하고 IO process로 결과를 전달할 수 있어야 한다.
2. 이와 같은 프로세스 환경에서 프로세스들 사이에 소통할 수 있는 방법이 필요하다. IPC(Shared memory, message queue, semaphore)를 각 프로세스에서 적절히 사용하여 입력값과 결과값을 주고받을 수 있어야 한다.
3. <그림1>에서 구름 모양으로 표시된 IO process logic은 단순하지 않다.



<그림2: IO process logic>

<그림2>는 IO process logic의 플로우 차트를 대략적으로 나타낸다. 이렇듯 복잡하므로 사용자로부터 직접 입력받는 device 입출력을 어느정도 추상화할 필요가 있다. 물론 그전에 Target board의 여러 입출력 장치들을 제어할 수 있어야 한다. 이 때 LED 장치는 mmap을 통해, 나머지 장치는 device driver를 사용하도록 한다.

나. 개발 내용

```
Press ? for help
.. (up a dir)
/home/solesie/Embedded-System-Software/proj1/
├── common/
│   ├── logging.h
│   └── mode.h
├── device/
│   ├── dev_ctrl.c
│   └── dev_ctrl.h
├── Documentation/
│   └── Embe24_과제1_명세서_v1.5.pdf
├── ipc/
│   ├── payload/
│   │   ├── merge_res.h
│   │   ├── msg.h
│   │   ├── record.c
│   │   ├── record.h
│   │   ├── shm_input.h
│   │   └── shm_output.h
│   ├── ipc_keys.h
│   ├── msgq.c
│   ├── msgq.h
│   ├── semaphore.c
│   ├── semaphore.h
│   ├── shm_database.c
│   ├── shm_database.h
│   ├── shm_io.c
│   └── shm_io.h
├── process/
│   ├── io_process.c
│   ├── io_process.h
│   ├── main_process.c
│   ├── main_process.h
│   ├── merge_processd.c
│   └── merge_processd.h
├── main.c
├── Makefile
└── Readme.md
```

<그림3: 프로젝트 개발 디렉토리 구조>

1. 각 프로세스들의 로직은 process/ directory에 위치한다. main process는 GET PUT을 담당하고, IO process는 입출력 로직과 main process와 merge process에 게 데이터를 보내는 일을 담당한다. merge process는 MERGE를 담당한다.
2. IPC를 위한 structure는 ipc/ directory에 위치한다. 그리고 이번 프로젝트에서 IPC에서 주고받는 메시지들의 규격은 ipc/payload directory에 위치한다. shm는 shared memory를 의미하고 msgq는 message queue를 의미한다. <그림1>에서 IO process에서 main process로 데이터를 전달할 때 SHM input, main process에서 IO process로 그 결과를 전달할 때 SHM output, IO process에서 merge process와 결과를 주고받을 때 MSG queue라 표기한 것을 볼 수 있다. 이때 규격이 각각 ipc/payload/shm_input.h, ipc/payload/shm_output.h, ipc/payload/msg.h를 의미한다.
3. 디바이스 드라이버나 mmap을 사용해서 디바이스 입출력을 관리하거나, 디바이스

입출력을 추상화하는 로직은 device/ directory에 위치한다. ctrl은 controller를 의미한다. 이 파일은 입출력 로직을 가지는 IO process에 의해서만 사용된다.

III. 추진 일정 및 개발 방법

가. 추진 일정

- 2024/04/04: 개발 환경 세팅 및 분석
- 2024/04/05: 프로젝트 구조 구성
- 2024/04/06 ~ 2024/04/08: IPC(ipc/ directory) 구현
- 2024/04/09 ~ 2024/04/10: device IO(device/ directory) 구현
- 2024/04/11 ~ 2024/04/12: IO process, main process, merge process (process/ directory) 구현
- 2024/04/15: 명세서와 어긋나는 로직 수정 및 버그 수정 및 제출

나. 개발 방법

1. 프로젝트의 구성

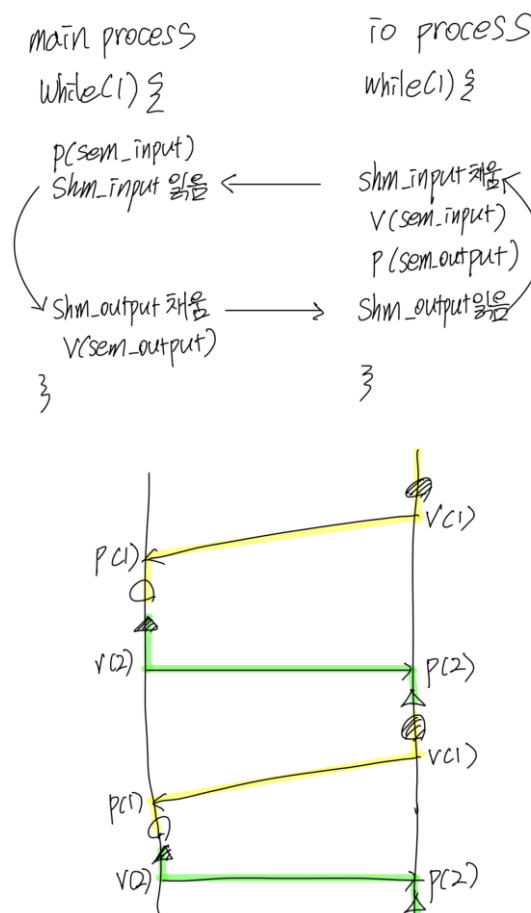
개발 일정 계획이 틀어지지 않기 위해 추후 수정 사항이 생겨도 수정하기 쉬워야 한다. 이에 각 디렉토리 별로 서로 의존적이지 않게끔 설계하는 방법이 필요하다. 따라서 구현 일정상 제일 마지막 단계인 IO process, main process, merge process(process/ directory)는 절차지향적으로 설계하되 device IO(device/ directory), IPC(ipc/ directory)구현은 객체지향적으로 설계한다. 또한 process/ directory를 제외하면 서로 디렉토리를 넘어 헤더파일을 include하는 일은 없도록 한다.

2. IPC(ipc/ directory), process/ directory 구현

우선 이번 프로젝트에서 사용할 system V 에서 제공하는 기본 IPC 기능들인 (mmap도 가능하지만 본 프로젝트에서 IPC에 직접적으로 사용하지 않으므로 제외 하면) shared memory, message queue, semaphore 들은 할당받은 프로세스가 종료되어도 그대로 커널 자원에 남아있다. 이 자원들은 명시적으로 해제해주어야 한다. 그러나 개발을 하다보면 어쩔 수 없이 오류 처리를 하거나 강제종료를 피할 수 없는 경우가 있다. 이에 system V IPC 자원을 관리할 최상위 프로세스인 root process를 <그림1>과 같이 적용한다. 그 후에 명세서대로 main process가 IO process와 merge process를 fork()하도록 한다.

Key-value store database로직(memory table 로직 + storage table logic)은 main process, IO process, merge process 모두 사용한다. 따라서 database 로직을 shm_database.c에 객체 형태로 모으고, 접근은 mutex를 통해 한번에 한 프로세스만 접근 가능하도록 한다. 여기서 read-write problem을 적용할 수 있지만 프로세스는 3개로 한정되었고 구현할 로직 상 동시 접근이 거의 불가능 하므로 적용하지 않는다.

IO process와 main process는 순서가 존재한다. IO process가 shm_input에 데이터를 써야만 main process가 읽을 수 있고 main process가 shm_output에 데이터를 써야만 IO process가 읽을 수 있다. 여기서 아래 <그림4> 와 같이 semaphore를 이용한다.



<그림4: shared memory io semaphore>

그런 일은 거의 없지만, merge process가 동작하는 중에 GET, memory table persist, 종료 등이 일어난다면 어떤 일이 일어날 지 모른다. msgq.c에선 이런일이 일어나지 않도록 구현한다.

3. device/ directory 구현

IO process logic이 복잡하므로 device 입출력을 직접적으로 컨트롤 함에 있어서 추상화를 적용한다. 여기선 시간 로직을 행동 로직으로 바꾼다. 예를 들어 “2초간 꼭 1번 버튼을 누른다”와 “1번 버튼을 누른다”는 모두 하나의 행동으로 취급하도록 한다. 즉, Target board의 어느 버튼을 눌렀다 떴을 때 그 행동이 결정되도록 한다. 출력시엔 motor의 최소 동작시간(2초) 등 뿐만 아니라 일반적인 fnd, text lcd 출력 utility 함수 또한 정의한다. 물론 이미 적재된 device file을 open하는 과정 또한 여기서 구현한다.

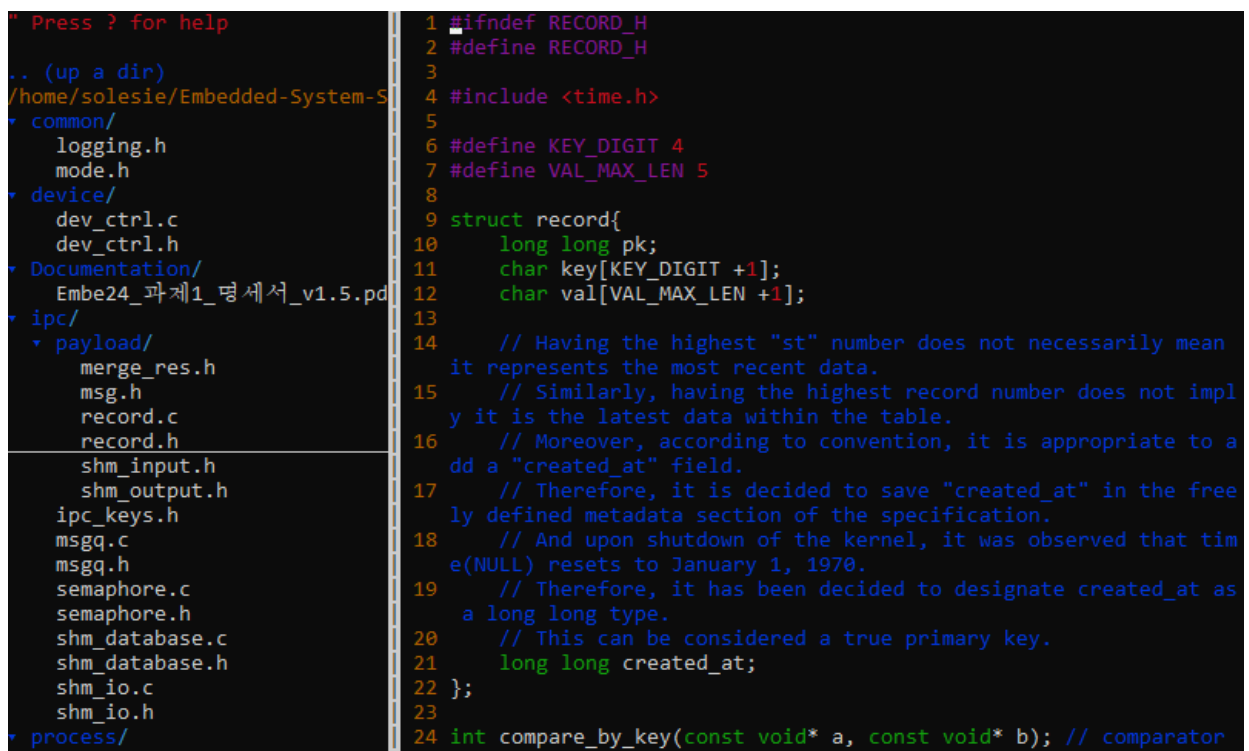
IV. 연구 결과

1. 예외 및 추가 구현 사항

A. Key-Value store metadata

3개의 storage table이 되었다면 merge process는 MERGE를 수행해야 한다.

그리고 GET 시엔 무조건 시간상 최신 입력된 순서대로 조회해야 한다.



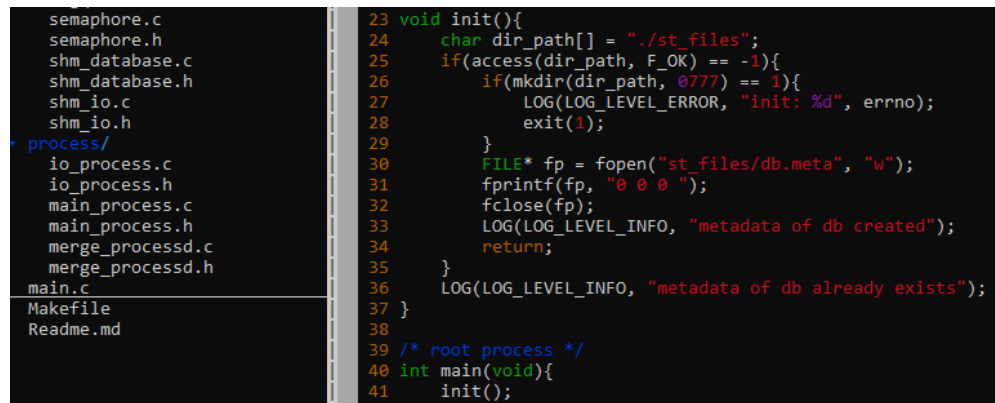
<그림5: record.h>

이렇듯 MERGE를 거치다 보면, 주석에 언급한 바와 같이 높은 번호의 st라고 해서 가장 시간상 최신의 데이터를 들고있지 않게 된다. 따라서 {스테이블명}.meta 라는 파일에 record구조체의 created_at값을 저장하도록 한다. 그러

나 time 함수를 써서 저장하려 하니 Target board를 켜다 키면 시간이 1970/1/1로 초기화 되는 문제가 발생했다. 따라서 이 값은 database에 시간상 저장된 순서를 나타내기로 한다.

마지막에 저장된 key-value의 순서가 몇번째인지, 다음에 할당할 st의 번호는 몇번인지, 현재 st가 몇 개인지 알면 편하다. 이 값을 프로젝트가 실행되는 단계에서 쉽게 읽어올 수 있도록 db.meta라는 파일에 이 값들을 저장하였다.

그리고 이 .st, .meta 파일들은 st_files/ 라는 디렉토리에 따로 저장된다.



```
23 void init(){
24     char dir_path[] = "./st_files";
25     if(access(dir_path, F_OK) == -1){
26         if(mkdir(dir_path, 0777) == 1){
27             LOG(LOG_LEVEL_ERROR, "init: %d", errno);
28             exit(1);
29         }
30         FILE* fp = fopen("st_files/db.meta", "w");
31         fprintf(fp, "0 0 0 ");
32         fclose(fp);
33         LOG(LOG_LEVEL_INFO, "metadata of db created");
34         return;
35     }
36     LOG(LOG_LEVEL_INFO, "metadata of db already exists");
37 }
38
39 /* root process */
40 int main(void){
41     init();
```

<그림6: main.c init(). 디렉토리 생성>

따라서 해당 디렉토리가 없으면 해당 디렉토리를 <그림6>와 같은 과정을 거쳐 생성하도록 하였다. 이때 st_files/db.meta 파일 초기화 또한 실행된다.

B. SIGINT handler

System V system call로 할당받은 IPC 자원은 할당받은 프로세스가 종료된다 하더라도 그대로 커널 자원에 남아있다. 이 자원들은 명시적으로 해제해주어야 한다. 그러나 개발 도중 오류상황에선 강제종료를 피할 수 없는 경우가 있었다. 이에 IPC 자원을 관리할 최상위 프로세스인 root process를 만들어 해당 자원을 정상적으로 해제할 수 있도록 하였다.

msgq.h	68	// create main process
semaphore.c	69	if((main_pid = fork()) == 0){
semaphore.h	70	// set main process as new process group
shm_database.c	71	setpgid(0, 0);
shm_database.h	72	
shm_io.c	73	// create io process
shm_io.h	74	if((io_pid = fork()) == 0){
process/	75	io_process(ids, db, ipc_io, msgq);
io_process.c	76	return 0;
io_process.h	77	}
main_process.c	78	
main_process.h	79	// create merge process
merge_processd.c	80	if((merge_pid = fork()) == 0){
merge_processd.h	81	merge_processd(db, msgq);
main.c	82	return 0;
Makefile	83	}
Readme.md	84	
	85	main_process(ids, db, ipc_io);
	86	
	87	// Normally terminated.
	88	// The main process waits until the IO process finishes the remaining tasks.
	89	// Only the merge process does not know when it should terminate.
	90	waitpid(io_pid, &status, 0);
	91	kill(merge_pid, SIGTERM);
	92	waitpid(merge_pid, &status, 0);
	93	return 0;

<그림7: main.c main()>

그리고 <그림7>에 71번째 줄의 setpgid(0,0)을 통해 main process, io process, merge process는 같은 프로세스 그룹으로 묶어서

```
63 void sem_shm_input_up(struct sem_ids* ids) {
64     if(semop(ids->shm_input, &v, 1) == -1) {
65         LOG(LOG_LEVEL_ERROR, "sem_shm_input_up: %d", errno);
66         killpg(getpgid(), SIGABRT);
67     }
68 }
```

<그림8: semaphore.c sem_shm_input_up()>

이와 같이 이상한 예외상황에선 프로세스 그룹을 같이 죽일 수 있도록 하였다. 이 상황에서 root process는 같은 프로세스 그룹이 아니므로 종료되지 않고 무사히 IPC 자원을 해제할 수 있게 된다.

```
19 void root_sigint_handler(int sig){
20     LOG(LOG_LEVEL_INFO, "to exit normally, must press the BACK button");
21 }
```

<그림9: main.c sig int handler>

또한 root process는 sigint handler를 위와 같이 등록하여 “정상적인 종료를 위해선 BACK버튼을 눌러라”라고 사용자에게 INFO level로 알리며 비정상적인 종료를 방지하였다.

2. 몇몇 함수 및 소스파일 설명

A. common/mode.h

PUT, GET, MERGE 를 나타내는 enum mode; 가 존재한다. 이 값은 ipc/payload에서도 사용하고 main process, io process 에서도 사용하는 값 이므로

common 디렉토리에 위치시켰다.

B. device/dev_ctrl.h

```
14 /* abstracted device Input*/
15 enum input_type {
16     // switch
17     S1_HOLD = 0,
18     S1,
19     S2,
20     S3,
21     S4,
22     S5,
23     S6,
24     S7,
25     S8,
26     S9,
27     S4_AND_S6,
28     // dip_switch
29     RESET,
30     // event
31     BACK,
32     VOL_UP,
33     VOL_DOWN,
34     // nothing pressed
35     DEFAULT
36 };
37
38 /* definition of LED actions to be executed */
39 enum led_action {
40     LED_OFF = 0,
41     LED1_ON,
42     LED3_AND_LED4_TOGGLE,
43     LED7_AND_8_TOGGLE,
44     LED5_ON,
45     LED_ALL
46 };
47
48 struct device_controller;
49 struct device_controller* device_controller_create();
50 void device_controller_destroy(struct device_controller* dc);
51 enum input_type device_controller_get_input(struct device_controller* dc, enum led_action ac
);
52 void device_controller_led_off(struct device_controller* dc);
53 void device_controller_fnd_print(struct device_controller* dc, const char numbers[FND_MAX +
1]);
54 void device_controller_fnd_off(struct device_controller* dc);
55 void device_controller_lcd_print(struct device_controller* dc, const char mode[TEXT_LCD_MAX_
LINE + 1], const char data[TEXT_LCD_MAX_LINE + 1]);
56 void device_controller_lcd_off(struct device_controller* dc);
57 void device_controller_motor_on(struct device_controller* dc);
58 void device_controller_motor_off(struct device_controller* dc);
59 void device_controller_led_all_on(struct device_controller* dc);
60
61 #endif // DEV_CTRL_H
```

<그림10: dev_ctrl.h>

III-나-3 에서 언급한 사용자 입력의 추상화가 enum input_type에 정의되어 있다. 함수들은 함수 이름 그대로 디바이스를 할당받고 컨트롤 할 수 있는 함수들이다. enum led_action에 LED3_AND_LED4_TOGGLE 과 같은 토글 action 이 정의되어 있다. 사용자 입력을 device_controller_get_input()를 통해 추상화 하였는데, 예를 들면 GET 모드에서 123을 입력한 상황에서 5번 스위치를 꺾 눌렀다고 해보자. 이때 5번스위치를 꺾 누르는 도중에도 3번 led와 5번 led가 토글하는게 옳다 생각했다. 따라서 입력 받는 도중의 led 행위를 enum

led_action 값을 통해 받을 수 있도록 하였다.

또한 추가적으로 이 dev_ctrl을 사용하는 io_process와 연관되어 언급할 점은, merge상황에서 모터가 돌아가거나, get/put 이 완료되어 led가 전부 켜질 때엔 입력을 받을 수 없다는 점이다. 그리고 1번스위치를 꼭 누르는 행위는 1번 스위치가 2초가 경과되었을 때부터 적용된다.

C. msg.h, merge_res.h, msgq.h

우선 msgq에 정의된 message queue는 msg.h에 정의된 struct msgbuf를 전달하고, 거기에 직렬화/역직렬화 되는 데이터가 바로 merge_res.h에 정의된 struct merge_res이다. 이러한 연관관계를 가지고 있다.

```
1 #ifndef MSGQ_H
2 #define MSGQ_H
3
4 #include "shm_database.h"
5
6 struct bidir_message_queue;
7
8 struct bidir_message_queue* bidir_message_queue_create();
9 void bidir_message_queue_destroy(struct bidir_message_queue*);
10 void bidir_message_queue_send_merge_req(struct bidir_message_queue* msgq, struct merge_res* res);
11 void bidir_message_queue_on_message_merge(struct bidir_message_queue* msgq, struct database* db);
12
13 #endif // MSGQ_H
```

<그림11: msgq.h>

bidir이라 명명한 이유는, 정말 낮은 확률로(만약 데이터가 엄청 많아진다면) merge가 진행되기 전에 put이 또 진행되어 st가 4개 이상 있을 수도 있다. 결국 동기화가 필요한 시점이 생기는데, 그 동기화를 여기서 진행하기 때문에 bidir이라 명명하였다. 그리고 MERGE 모드일 경우엔 IO process가 단순히 merge process로 메시지를 전송만 하지 않고 merge process로부터 그 결과값을 받아와 text lcd에 그 결과를 출력해야 하기 때문이다.

또한 send_merge_req()는 io process에 의해서만, on_message_merge()는 merge process에 의해서만 불러 진다.

V. 기타

System V IPC 시스템콜을 사용하는 것과 디바이스 드라이버를 직접 사용해 보는 경험은 정말 새롭고 재밌는 경험이었습니다. 그러나 제가 key-value store대해선 미숙하여 merge 과정에 대한 그 목적을 잘 몰랐습니다. 그러다 보니 Get와 merge와 관련된 명세서에 몇몇 부분들이 헛갈렸습니다. System V IPC 관련 자료처럼 key-value store도 간단히 첨부되어 있었다면 좋겠다는 생각이 들었습니다.