**Embedded System Software CSE4116**

**HW#1 Project :**

**Device control with Inter-Process Communication**

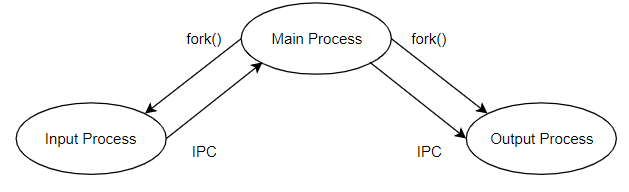
**컴퓨터공학과**

**20131579 윤기영**

1. **개발 목표**

Input process, Main process, Output process를 만들어, 디바이스 컨트롤과 IPC를 이용하여 Clock, Counter, Text editor, Draw board를 구현한다.

1. **개발 범위 및 내용**



* **Input process**

이 프로세스는 계산을 위해 입력을 받는 입력 장치를 관리한다. 주기적으로 입력 장치로부터 새로 들어온 입력이 있는지 확인하고, 있을 경우 IPC를 통해 main process로 전달한다.

* **Main process**

이 프로세스는 실질적인 연산을 수행한다. input process로부터 전달 받은 값으로 현재 모드에 맞는 적절한 처리를 하고, 그 결과를 output process에게 IPC를 통해 전달한다.

* **Output process**

이 프로세스는 출력 장치를 관리한다. main process로부터 전달 받은 값을 장치에 쓰거나 특정 동작을 하게 만든다.

* **IPC (message queue)**

이 프로젝트에서는 여러 프로세스들이 각각의 역할을 가지고 있고, 어떤 정보들을 서로에게 보내주어야 하는데, 이를 Inter-process communication이라고 한다. IPC에는 file, signal, socket, pipe, shared memory등의 여러 방법이 있지만 여기서는 message queue 방식을 사용하였다.

message queue는 메시지를 queue에 관리한다. publisher가 메시지를 queue에 넣으면 subscriber는 queue에서 메시지를 꺼내 읽는 방식이다. message queue는 커널에서 전역적으로 관리되며, 모든 프로세스에서 접근 가능하도록 되어있다. 그래서 만약 같은 message queue key id만 알고 있다면 같은 message queue를 사용할 수 있다.

* **mmap()**

User program이 device에 명령을 내리거나 데이터를 보내는 register의 주소를 자신의 메모리 공간에 mmap을 통해 mapping 한 뒤, 해당 공간에 데이터를 쓰거나 읽어서 device를 직접 제어하는 방식이다. 이 때 register의 주소를 알아야 하고, 어떤 register에 값을 썼을 때 device가 어떻게 동작하는지 등을 다 알아야한다.

* **Device Driver**

특정 하드웨어나 장치를 제어하기 위한 커널의 일부분으로 동작하는 프로그램이다. 드라이버는 장치 칩의 register에 접근하여 하드웨어를 제어한다. 어떤 프로그램이 드라이버의 명령어를 호출하면, 드라이버는 장치에 명령을 전달한다. 장치가 드라이버에게 데이터를 되돌려주면, 드라이버는 원래 요청한 프로그램의 명령어로 데이터를 다시 전달한다.

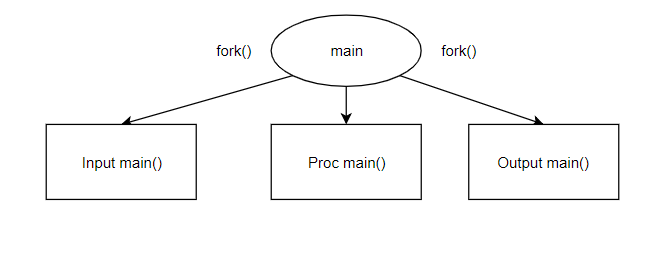
1. **개발 일정**

|  |  |
| --- | --- |
| 날짜 | 개발 내용 |
| 2019.04.11 | - 3개의 프로세스로 나누는 부분 구현 |
| 2019.04.12 | - Input process 구현  - Output process에서 mmap과 driver를 통해 장치를 제어하는 부분 구현 |
| 2019.04.13 | - mode1(Clock) 구현  - mode2(Counter) 구현  - mode3(Text editor) 구현 |
| 2019.04.14 | - mode4(Draw board) 구현  - mode5(Timer 추가구현) 구현  - process exit, killing child process 구현 |

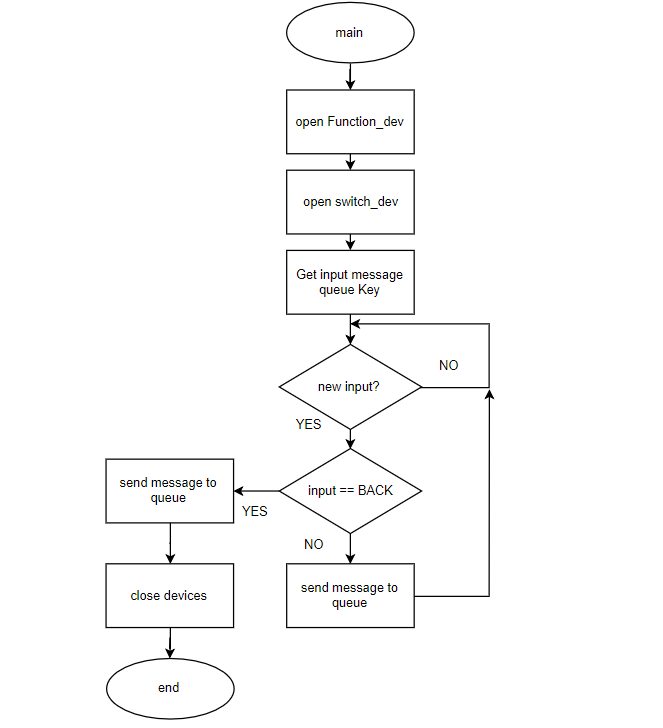
1. **개발 결과**

* **Flow chart**

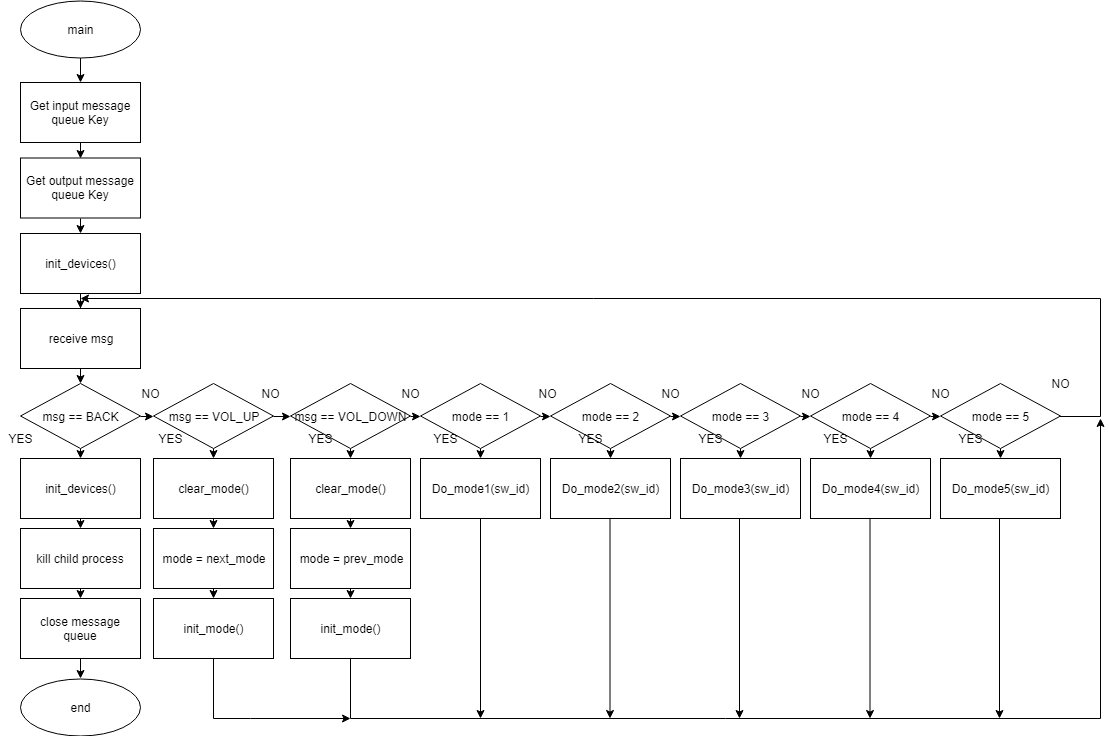
<20131579.c>



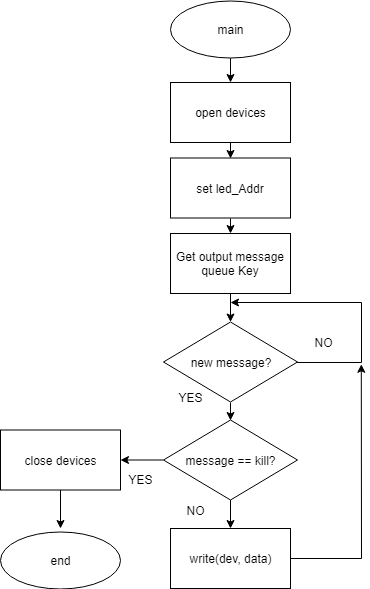
<reader.c>



<processor.c>



<writer.c>



* **Input process**

input process는 오직 입력을 받고 그 입력을 main process로 넘겨주는 역할만 한다. 먼저 헤더에 지정된 키로 message queue의 key id를 얻는다.

device로부터의 입력을 읽기 위해서 function key(readkey)와 switch device를 열고 while문을 돌면서 입력이 있는지 계속 확인한다. default로 입력이 없으면 입력이 오기까지 block되는데, while문에서는 입력을 각각 1번씩 확인해야 하기 때문에, function key에서 입력이 없다고 block이 되어버리면 switch 입력을 못 받는 경우가 생긴다. 따라서 O\_NONBLOCK 옵션을 주어서 입력이 없으면 무시하도록 했다.

function key의 경우에는 키를 뗄 때만 그 입력을 받아 message queue에 보낸다.

switch key의 경우는 키를 뗐다는 입력이 주어지지 않고 현재의 switch 상태를 읽어 오기 때문에 다른 방법을 사용했다. 먼저 switch 중에서 어떤 키가 눌린 상태이면 그 상태를 or연산을 이용해 다른 버퍼에 저장해두고 키가 아무것도 안 눌려 있을 때까지 계속 입력을 받아 버퍼에 더해준다. 즉 키를 땔 때까지 어떤 키가 눌렸는지 확인한다. 그리고 이 정보로부터 눌린 키를 메시지에 담아 message queue에 보내준다.

* **Main process**

input과 output에 해당하는 각각의 message queue key id를 얻는다. while문을 통해서 input message queue로부터 메시지가 와있는지 계속 확인한다. 이 때 msgrcv()를 바로 호출하면 queue에 메시지가 없을 때 에러가 난다. 그래서 msgctl()로 queue에 있는 메시지 수를 확인하고 있을 경우에만 msgrcv()를 호출한다.

일부 모드에서 일정 시간마다 device를 조작해야 하는 경우가 있는데, 이는 매번 while문 안에서 보내야 하는 시간이 되었는지 체크하고 msg를 보내고 보내야 하는 시간을 update 해준다.

BACK버튼 입력이 오면 output process에게 device를 전부 초기화하는 내용의 메시지와 입력을 그만 받으라는 내용이 담은 kill 메시지를 보내고 while문을 탈출한다. 그리고 다른 자식 프로세스들이 죽을 때까지 wait()하고 message queue를 없애고 종료한다.

Vol+-버튼 입력이 오면 모드를 초기화하고 모드를 변경한 후에 다시 해당 모드를 초기화해준다.

switch버튼 입력이 들어오면 각 모드에서 적절한 계산을 하고 그 device에 변경이 필요할 경우 output process에게 메시지를 보낸다.

* **Output process**

사용하는 device들을 모두 열고 while문을 통해 message queue를 계속 읽는다. 메시지에 어떤 device에 write()해야 되는지 msg.fix\_bit를 보고 각 장치에 데이터를 write()한다. 만약 온 메시지가 프로그램이 끝났다는 데이터가 들어있으면 while문을 탈출하고 모든 device들을 닫은 후에 프로세스를 종료한다.

* **IPC**

Linux OS에서 제공하는 message queue를 사용했다. [input process <-> main process], [main process <-> output process] 사이에 사용하는 queue를 2개 사용하였다. 헤더 파일에 input queue와 output queue에서 사용할 key를 정의해 놓고 해당 key를 이용해 queue의 key id를 얻는 방식으로 사용하였다.

* **mode1 Clock**

sw(1) : 보드 시간 변경 모드로 진입, 혹은 변경된 시간으로 저장

sw(2) : 보드 시간을 system 시간으로 변경

sw(3) : 시간을 1시간 증가

sw(4) : 시간을 1분 증가

FND에 시간을 HH:MM형식으로 출력한다. LED는 평상시에는 1번만 켜지고 시간 변경 모드에는 3,4번이 번갈아 가며 켜진다.

* **mode2 Counter**

sw(1) : 진수 변환. 순서는 10 – 8 – 4 – 2 – 10진수

sw(2) : 백의 자리를 1 증가

sw(3) : 십의 자리를 1 증가

sw(4) : 일의 자리를 1 증가

FND에 숫자를 출력한다. 기본은 10진수이고 LED는 각 진수에 따라 2번, 3번, 4번, 1번이 켜진다.

* **mode3 Text Editor**

sw(1~9) : 각 버튼에 해당하는 값을 입력

sw(2)+(3) : LCD를 초기화

sw(5)+(6) : 영어/숫자 입력 모드 전환

sw(8)+(9) : 공백 입력

FND에 스위치 입력 횟수를 출력한다. LCD가 가득 차면 제일 앞 문자가 지워지고 맨 뒤에 새로운 문자가 채워진다. Dot matrix는 현재 영어 입력 모드이면 'A', 숫자 입력 모드이면 '1'이 그려진다.

* **mode4 Draw board**

sw(2, 4, 6, 8) : 각 방향으로 커서 이동

sw(1) : Dot matrix 및 기타 초기화

sw(3) : 커서 깜빡임/숨김

sw(5) : 선택

sw(7) : Dot matrix 초기화

sw(9) : Dot matrix 반전

FND에 스위치 입력 횟수를 출력한다. Dot matrix에는 커서가 그린 그림이 그려진다.

* **mode5 Timer (Additional Implementation)**

sw(1) : 타이머 작동/일시정지

sw(2) : 타이머 1초 증가

sw(3) : 타이머 리셋

FND에 SS:MsMs형식으로 시간을 출력한다. (예를 들어 7.58초면 [0758]로 표시) 타이머가 작동되고 0초에 도달하면 Buzzer가 작동되어 비프 소리를 1회 낸다.

* **mode5 Timer – Detail**

sw(1)은 타이머가 00:00이 아닐 때, 타이머를 작동시킨다. 타이머가 0이 되면 Buzzer에서 비프 소리가 1회 난다. 만약 타이머가 작동 중일 때 sw(1)을 한번 더 누르면 진행 중이던 타이머가 일시 정지된다. 일시 정지되어 있을 때, 다시 한번 누르면 일시 정지가 풀린다.

sw(2)는 타이머를 1초 늘리는 기능을 한다. 만약 타이머가 작동 중이면 타이머를 늘릴 수 없고, 일시 정지인 상태에서만 가능하다.

sw(3)은 타이머를 00:00으로 리셋시키는 기능을 한다. 타이머가 작동 중에도 동작한다.

1. **기타**

* **소감**

Linux PC에서 작성한 프로그램이 arm 프로세서에서 돌아가게끔 cross compile을 사용하는 프로젝트였다. 그러나 개발과정에서 제일 중요했던 것은 쓰레드 프로그래밍 혹은 IPC부분이었다. 두 개 다 해 본 경험이 없어서 이번 프로젝트가 굉장히 흥미로웠고, 시간과 능력상 저급한 방법으로 구현했다고 생각하는 fork나 일정 주기마다 깜빡이는 신호를 보내는 코드에 대해서는 추후에 리뷰해볼 예정이다.