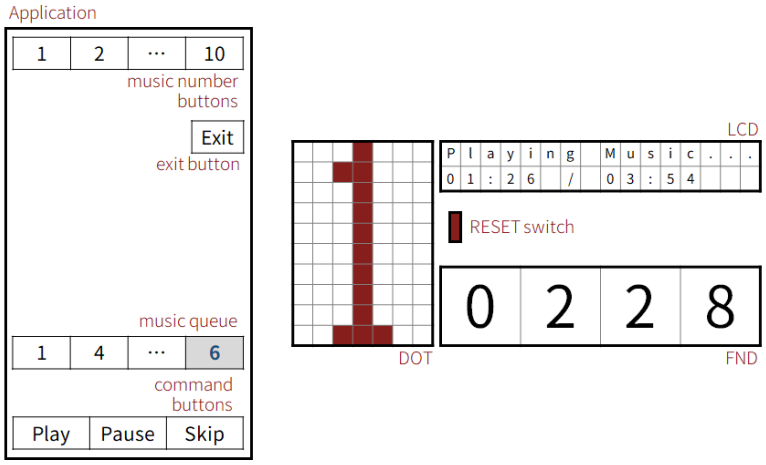
임베디드시스템소프트웨어 프로젝트 보고서

컴퓨터공학과 20211584 장준영

1. 개발 목표

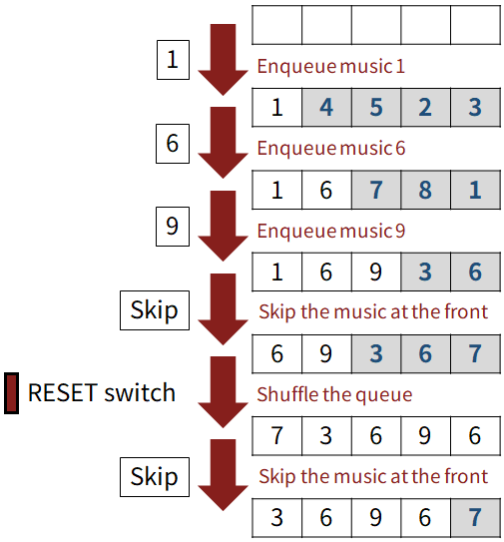
임베디드시스템소프트웨어 과목의 최종 프로젝트로 '음악 재생 프로그램 및 추천 시스템'을 구현한다. 지금까지 수업 시간에 배운 리눅스 커널 모듈, Android SDK, NDK 프로그래밍을 모두 사용하여 종합적인 임베디드 시스템 소프트웨어를 설계한다.

2. 개발 내용

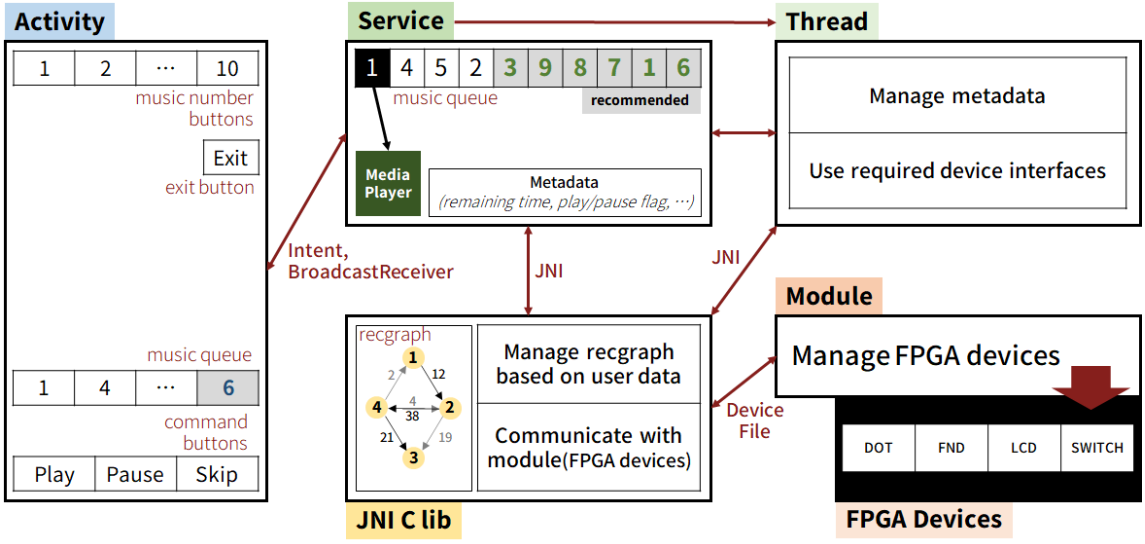


개발한 임베디드 시스템 소프트웨어의 유저 엔드 부분으로, 음악 재생 프로그램의 아주 기본적인 형태를 확인할 수 있다. 사용자는 디스플레이 화면의 여러 버튼과 RESET 스위치를 통해 어플리케이션을 조작하고, 그 결과를 디스플레이 화면의 music queue와 DOT, LCD, FND 디바이스로 확인할 수 있다. 1부터 10까지 번호가 붙은 버튼은 각각 음악의 번호를 나타내며, 해당 버튼을 눌러 재생 목록에 음악을 추가할 수 있다. Play, Pause, Skip, Exit 버튼은 각각의 이름에 해당하는 동작을 한다. Play는 일시 정지 상태에서 재생 상태로, Pause는 재생 상태에서 일시 정지 상태로 변환하고, Skip은 현재 재생 중인 음악을 스킵하고 다음 음악을 재생한다. Exit 버튼을 누르면 프로그램을 종료할 수 있다. 사용자가 조작하여 만들어진 재생 목록은 디스플레이 화면의 music queue에서 확인할 수 있는데, 파란색 폰트(사진에서는 검정색 폰트)로 보이는 번호들은 유저가 직접 추가한 것으로 간주되는 부분이고, 검정색 폰트(사진에서는 파란색 폰트)로 보

이는 번호들 추천 시스템에서 자동으로 추가한 추천 음악 부분이다. DOT 매트릭스에선 현재 재생 중인 음악의 번호를 확인할 수 있고, FND 디스플레이에선 현재 재생 중인 음악의 남은 시간을 확인할 수 있다. LCD 디스플레이에선 현재 음악이 재생 중인지, 만약 그렇다면 음악의 전체 시간 중 어느 정도 재생되었는지를 확인할 수 있다. 사용자가 RESET 스위치를 누르면 현재 재생 목록에 있는 모든 요소를 셔플하여 유저가 추가한 것으로 간주한다.



위 이미지는 유저 조작의 예시로, 번호 버튼을 누르면 차례대로 음악이 추가되고 그에 따른 추천 음악도 추가되는 것을 볼 수 있다. 음악은 유저가 직접 추가한 것이 추천된 음악보다 우선 추가된다. 또, Skip 버튼을 누르면 재생 목록의 맨 앞에서 재생중이던 음악을 넘기고 다음 음악을 재생한다. RESET 스위치를 눌러 재생 목록을 셔플하면, 재생 목록에 있는 모든 음악이 섞이고 유저가 추가한 것으로 간주된다. 빈 재생 목록에는 추천 음악이 추가되지 않는다.



프로그램의 개발 설계도는 위와 같다. (1)Activity는 사용자에게 보여지는 정보(버튼, 재생 목

록)를, (2)Service는 음악 재생을 위한 MediaPlayer 및 재생 목록(queue)과 메타데이터를, (3)JNI C 라이브러리는 음악 추천을 위한 recgraph와 모듈 file descriptor를, (4)모듈에서는 FPGA 디바이스들을 각각 관리한다. 또, Service에서는 일정한 주기마다 필요한 정보를 갱신하기 위해 워커 쓰레드를 별도로 관리한다. Activity에서 Service로 버튼 명령을 전달할 땐 intent를 사용하고, Service에서 Activity로 관리된 재생 목록 정보를 전달할 땐 BroadcastReceiver를 사용한다. 또, Service에서 추천 시스템 및 FPGA 디바이스를 사용해야 할 땐 JNI를 통해 NDK로 빌드된 C native 라이브러리 메써드를 호출한다. C native 라이브러리에선 해당 호출에 응하기 위해, JNI를 통해 Service 클래스의 정보를 확인하거나 디바이스 파일을 통해 리눅스 커널 모듈과 소통한다. 모듈은 디바이스 파일에 행해지는 file operation에 따라 알맞은 동작을 한다. 이렇게 수업 시간에 배운 임베디드 시스템 소프트웨어의 모든 개발 요소를 적극 활용한다. (자세한 동작 및 구현 방법은 **3. 개발 방법**에서 후술한다.) 이를 위해, Android 어플리케이션의 구조와 NDK에 대해 숙지해야 한다.

3. 개발 방법

[1] MainActivity.java (Activity)

어플리케이션의 Activity 부분인 MainActivity 클래스는 onCreate, onDestroy 메써드를 오버라이딩하여 사용한다.

```
public class MainActivity extends Activity
{
    private ArrayList<Integer> queue; /* music queue */
    private int userTop; /* number of user-inserted music */
```

우선 queue(재생 목록)과 userTop(사용자가 직접 추가한 음악의 개수)를 선언한다. 값은 이후 BroadcastReceiver를 통해 Service로부터 받아올 예정이다.

```
/* onCreate */
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    /* Register the receiver to receive broadcast intents with a matching action. */
    IntentFilter filter = new IntentFilter("org.example.musicplayer.QUEUE_STATUS");
    registerReceiver(receiver, filter);
    /* Start the service. */
    Intent intent = new Intent(MainActivity.this, MusicService.class);
    startService(intent);
```

어플리케이션이 시작되어 Activity가 생성되면 호출되는 onCreate 메써드이다. 우선 Service에서 정보를 받아올 BroadcastReceiver를 등록한다. 해당 리시버는 재생 목록의 정보를 담은 QUEUE_STATUS intent만 받아들인다. 이후 Service에도 어플리케이션이 시작됨을 알린다.

```
/* Music buttons (1 ~ 10) */
for (int i = 1; i <= 10; i++)
{
    int buttonId = getResources().getIdentifier("button" + i, "id", getPackageName());
    Button button = (Button)findViewById(buttonId);
    final int finalI = i;
    button.setOnClickListener(new View.OnClickListener()
    {
        @Override
        public void onClick(View v)
        {
            /* Include the music ID in the intent and pass it to the service. */
            Intent intent = new Intent(MainActivity.this, MusicService.class);
            intent.putExtra("buttonId", finalI);
            startService(intent);
        }
    });
}
```

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="1" />
```

이후 버튼 등록이 시작된다. 가장 먼저 1부터 10까지의 번호가 적힌 음악 추가 버튼이 등록된다. 우선 xml 레이아웃 파일의 알맞은 버튼 ID를 받아오고, 해당 버튼에 OnClickListener를 추가한다. 리스너에서는 버튼이 눌리면 버튼 번호를 담은 intent를 Service로 전송한다.

```
/* Play button */
Button playButton = (Button)findViewById(R.id.playButton);
playButton.setOnClickListener(new View.OnClickListener()
{
    @Override
    public void onClick(View v)
    {
        /* Include the "play" command in the intent and pass it to the service. */
        Intent intent = new Intent(MainActivity.this, MusicService.class);
        intent.putExtra("command", "play");
        startService(intent);
    }
});
```

다음으론 Play, Pause, Skip 명령 버튼을 등록한다. (이 세 버튼은 등록 과정이 동일하기 때문에, Play 버튼만 설명한다.) 역시 알맞은 버튼 ID에 OnClickListener를 추가하는데, 여기선 intent에 명령의 종류를 담아 Service로 전달한다.

```
    /* Exit button */
    Button exitButton = (Button)findViewById(R.id.exitButton);
    exitButton.setOnClickListener(new View.OnClickListener()
    {
        @Override
        public void onClick(View v)
        {
            /* Finish the activity. */
            finish();
        }
    });
    /* Update the queue display. */
    updateQueueDisplay();
}
```

Exit 버튼은 눌렸을 때 Activity를 종료한다. 마지막으로 재생 목록 디스플레이를 갱신하는 updateQueueDisplay를 호출하면서 onCreate를 마친다.

```
/* receiver - BroadcastReceiver to receive broadcast intents */
private BroadcastReceiver receiver = new BroadcastReceiver()
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        /* Get the queue and the userTop from the intent. */
        queue = intent.getIntegerArrayListExtra("queue");
        userTop = intent.getIntExtra("userTop", 0);
        /* Update the queue display. */
        updateQueueDisplay();
    }
};
```

BroadcastReceiver는 필터에 의해 걸러진 intent가 도착하면, 해당 intent에서 queue와 userTop의 정보를 받아 클래스 내의 필드에 갱신한다. 이후 재생 목록 디스플레이도 갱신한다.

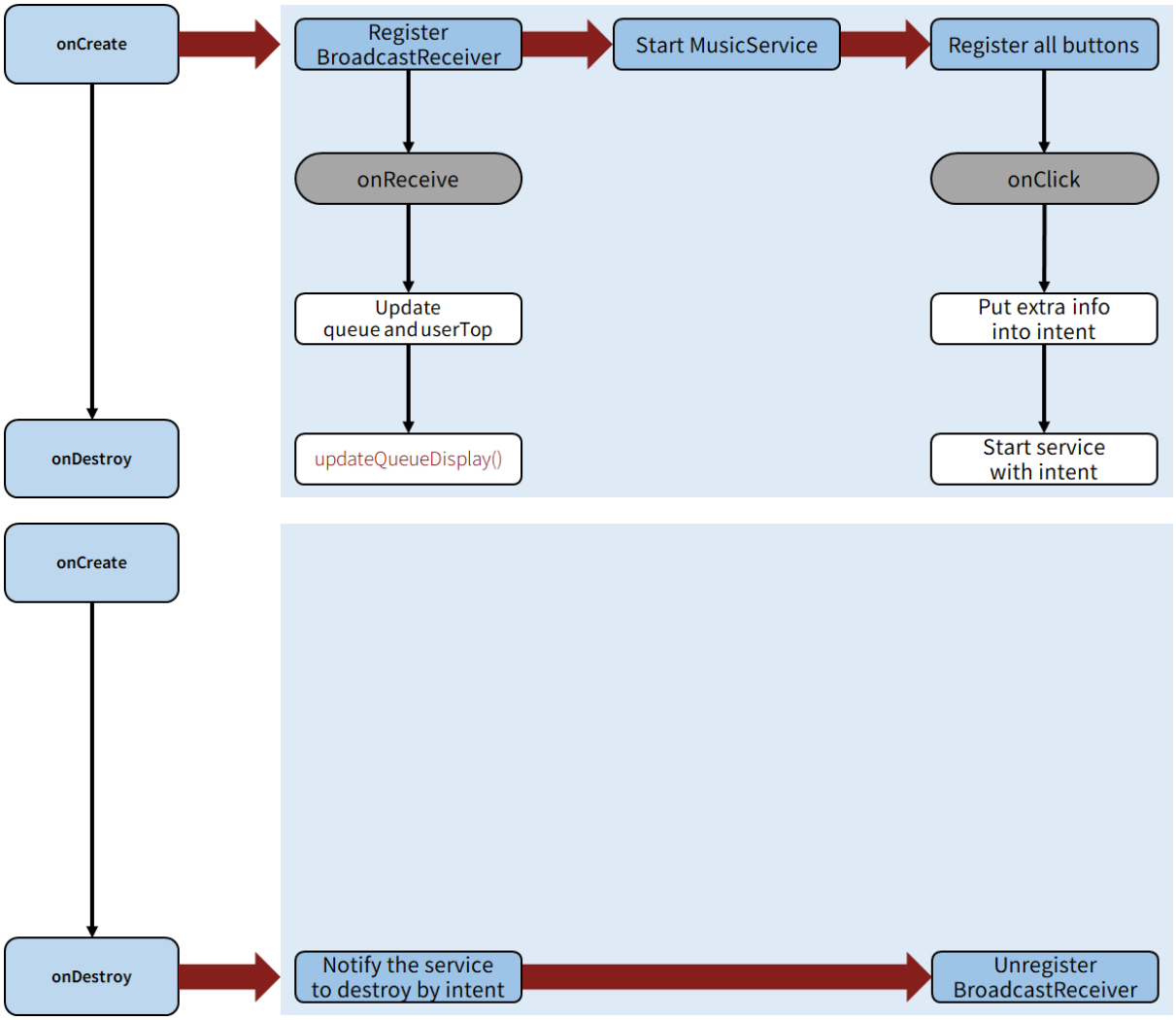
```
/* updateQueueDisplay - Method to update the queue display */
private void updateQueueDisplay()
{
    LinearLayout queueLayout = (LinearLayout)findViewById(R.id.queueLayout);
    /* Remove all views from the queueLayout. */
    queueLayout.removeAllViews();
    if (queue != null)
    {
        for (int i = 0; i < queue.size(); i++)
        {
            TextView textView = new TextView(this);
            textView.setText(String.valueOf(queue.get(i)));
            textView.setLayoutParams(new LayoutParams(LayoutParams.WRAP_CONTENT, LayoutParams.WRAP_CONTENT));
            /* If the music is user-added, set the text color to holo_blue_dark. */
            if (i < userTop)
                textView.setTextColor(getResources().getColor(android.R.color.holo_blue_dark));
            /* Otherwise, set the text color to black. */
            else
                textView.setTextColor(getResources().getColor(android.R.color.black));
            queueLayout.addView(textView);
        }
    }
}
```

updateQueueDisplay는 Service로부터 받아온 정보를 통해 재생 목록 디스플레이를 갱신한다.

우선 레이아웃의 queueLayout을 모두 지우고, 큐의 요소를 읽으며 디스플레이에 추가한다. 유저가 추가한 음악의 개수인 userTop 만큼의 음악은 파란색 폰트로, 나머지 추천된 음악은 검정색 폰트로 설정한다.

```
/* onDestroy */
@Override
protected void onDestroy()
{
    super.onDestroy();
    /* Include the "exit" command in the intent and pass it to the service. */
    Intent intent = new Intent(MainActivity.this, MusicService.class);
    intent.putExtra("command", "exit");
    startService(intent);
    /* Unregister the receiver. */
    unregisterReceiver(receiver);
}
```

Activity가 종료되면 호출되는 onDestroy 메써드이다. Service에게 종료되었음을 알리고, BroadcastReceiver를 등록 해제한다. MainActivity 클래스의 전체적인 구조도는 다음과 같다.



[2] MusicService.java (Service)

어플리케이션의 Service 부분인 MusicService 클래스는 onCreate, onStartCommand, onDestroy 메써드를 오버라이딩하여 사용한다.

```
public class MusicService extends Service
{
    private MediaPlayer mp; /* media player obj */
    private Queue<Integer> queue = new LinkedList<Integer>(); /* music queue */
    private int userTop = 0; /* number of user-added music */
    private boolean isPaused = false; /* boolean to check if the music is paused */
    private int prevButtonId = -1; /* ID of the previously pressed button */
    private int durationTime = -1; /* duration time of the currently playing music */
    private int remainingTime = -1; /* remaining time of the currently playing music */
    private Thread worker; /* worker thread */
```

우선 MediaPlayer, 관리하는 자료 구조 및 메타데이터, 워커 쓰레드를 선언한다. 모두 주석에 설명된 역할을 수행한다.

```
/* native library methods */
static
{
    System.loadLibrary("recgraph");
}
private native void initializeGraph();
private native void updateGraph(int prev, int curr);
private native void recommendMusic();
private native void saveGraph();
private native int resetInput();
private native void deviceOutput();
private native void initDisplay();

/* onBind */
@Override
public IBinder onBind(Intent intent)
{
    return null;
}
```

이후 C native 라이브러리를 불러오고 메써드를 선언한다. 또, Android Service에 필수적인 onBind 역시 오버라이딩한다. 이번 프로젝트에서 bound Service는 사용하지 않기 때문에, 단순히 null을 return 한다.

```
/* onCreate */
@Override
public void onCreate()
{
    super.onCreate();
    /* Initialize recgraph and output devices. */
    initializeGraph();
    initDisplay();
```

서비스가 사용되기 시작하면 호출되는 onCreate 메써드이다. 가장 먼저 추천을 위한

recgraph를 초기화하고, FPGA output device들도 초기화한다. (C native 라이브러리 메써드의 자세한 설명은 후술한다.)

```
/* Create a new media player. */
mp = new MediaPlayer();
/* Set a listener for when the music completes. */
mp.setOnCompletionListener(new MediaPlayer.OnCompletionListener()
{
    @Override
    public void onCompletion(MediaPlayer mp)
    {
        /* When the music completes, remove the song from the queue
         * and play the song at the front of the queue.
         */
        queue.poll();
        userTop--;
        if (userTop < 0)
            userTop = 0;
        playNextSong();
    }
});
```

이후 MediaPlayer를 선언한다. 재생 중인 음악이 끝났을 때 호출되는 OnCompletionListener에선 재생 목록의 맨 앞 음악(종료된 음악)을 삭제하고, userTop을 1 감소시킨다. 이후 변경된 맨 앞 음악을 재생하는 playNextSong 메써드를 호출한다.

```
/* playNextSong - Method to play the song at the front of the queue */
private void playNextSong()
{
    mp.reset();
    if (!queue.isEmpty())
    {
        /* Get the song ID from the queue and load the music corresponding to the ID. */
        int songId = queue.peek();
        int resId = getResources().getIdentifier("music" + songId, "raw", getPackageName());
        mp.reset();
        mp = MediaPlayer.create(this, resId);
        mp.setOnCompletionListener(new MediaPlayer.OnCompletionListener()
        {
            @Override
            public void onCompletion(MediaPlayer mp)
            {
                queue.poll();
                userTop--;
                if (userTop < 0)
                    userTop = 0;
                playNextSong();
            }
        });
        /* Start the music. */
        mp.start();
    }
}
```

playNextSong 메써드에서는 MediaPlayer를 초기화하고, 재생할 음악의 ID에 해당하는 파일을 불러와 등록한다(1 → /res/raw/music1.mp3). 역시 동일한 OnCompletionListener를 추가하고 재생을 시작한다.

```
/* Create a new worker thread. */
worker = new Thread(new Runnable()
{
    @Override
    public void run()
    {
        int prevValue = 0;
        while (true)
        {
            /* When the reset button is pressed,
             * shuffle the queue and play the song at the front of the queue.
             */
            int value = resetInput();
            if (value == 1 && prevValue == 0)
            {
                List<Integer> q2list = new ArrayList<Integer>(queue);
                Collections.shuffle(q2list);
                queue = new LinkedList<Integer>(q2list);
                userTop = queue.size();
                playNextSong();
            }
            prevValue = value;
```

다음으로 워커 쓰레드를 설정한다. 워커 쓰레드는 세 가지 일을 하는데, 첫 번째로 RESET 스위치의 입력을 받는다. resetInput의 결과가 true인 경우 RESET 스위치가 막 눌린 것이므로, 재생 목록을 셔플하고 재생 목록 내의 모든 음악을 유저가 추가한 것으로 간주한다(userTop = queue.size();). 이후 변경된 재생 목록의 맨 앞 음악을 재생한다.

```
            /* If the music is playing, update the duration and remaining time. */
            if (mp.isPlaying())
            {
                durationTime = mp.getDuration() / 1000;
                remainingTime = (mp.getDuration() - mp.getCurrentPosition()) / 1000;
            }
            else
            {
                durationTime = -1;
                remainingTime = -1;
            }
            /* Pass the necessary information to the module and broadcast the queue status. */
            deviceOutput();
            broadcastQueueStatus();
            /* Sleep for 0.1 seconds. */
            try
            {
                Thread.sleep(100);
            }
            catch (InterruptedException e)
            {
                return;
            }
        }
    }
});
/* Start the worker thread. */
worker.start();
```

두 번째로 durationTime과 remainingTime 메타데이터를 관리한다. 만약 MediaPlayer가 재생중이라면 getDuration와 getCurrentPosition 메써드를 이용해 음악의 전체 시간과 남은 시

간을 변경한다. 세 번째로 Service 내의 메타데이터를 모듈로 전송하고, 재생 목록 정보는 Activity로 broadcasting 한다. 이 과정은 0.1초에 한 번씩 실행된다.

```
/* broadcastQueueStatus - Method to broadcast the queue status */
private void broadcastQueueStatus()
{
    Intent intent = new Intent("org.example.musicplayer.QUEUE_STATUS");
    intent.putExtra("queue", new ArrayList<Integer>(queue));
    intent.putExtra("userTop", userTop);
    sendBroadcast(intent);
}
```

broadcastQueueStatus에선 필터를 설정한 intent에 필요한 정보를 담고, broadcasting 한다. 이는 MainActivity의 BroadcastReceiver에 도착한다.

```
/* onStartCommand */
@Override
public int onStartCommand(Intent intent, int flags, int startId)
{
    /* Leave only as many elements in the queue as userTop. */
    if (userTop != 0)
        maintainUserTop();
```

```
/* maintainUserTop - Method to leave only as many elements in the queue as userTop */
private void maintainUserTop()
{
    while (queue.size() > userTop)
        ((LinkedList<Integer>)queue).removeLast();
}
```

startService가 호출되면 실행되는 onStartCommand 메써드이다. 우선 유저가 직접 추가하는 음악이 재생 목록의 앞쪽에 올 수 있도록, 자동으로 추천된 음악을 재생 목록에서 제거한다.

```
/* Get the command from the intent. */
String command = intent.getStringExtra("command");
if (command != null)
{
    /* If the command is "play", start the music. */
    if (command.equals("play"))
    {
        if (isPaused)
        {
            mp.start();
            isPaused = false;
        }
        else
            playNextSong();
    }
    /* If the command is "pause", pause the music. */
    else if (command.equals("pause"))
    {
        if (mp.isPlaying())
        {
            mp.pause();
            isPaused = true;
        }
    }
```

이후 intent의 command를 확인한다. command가 null이 아니라는 것은 Play, Pause, Skip, Exit의 명령 버튼이 눌렸다는 의미이므로 알맞은 동작을 수행한다. Play 버튼이 눌린 경우 MediaPlayer가 일시 정지된 상태라면 이를 해제하고, 그렇지 않다면 playNextSong을 호출해 재생을 시작한다. Pause 버튼이 눌린 경우 재생 중인 음악을 일시 정지한다.

```
    /* If the command is "skip", skip to the next song. */
    else if (command.equals("skip"))
    {
        if (!queue.isEmpty())
        {
            queue.poll();
            userTop--;
            if (userTop < 0)
                userTop = 0;
            isPaused = false;
            playNextSong();
        }
    }
    /* If the command is "exit", destroy the service. */
    else if(command.equals("exit"))
        onDestroy();
}
```

Skip 버튼이 눌린 경우, 재생 목록이 비어있지 않다면 맨 앞의 음악을 제거하고 userTop을 1 감소시킨다. 이후 맨 앞 음악이 바뀌었으므로 일시 정지를 해제하고 playNextSong을 호출한다. Exit 버튼이 눌린 경우 onDestroy를 호출해 서비스를 종료한다.

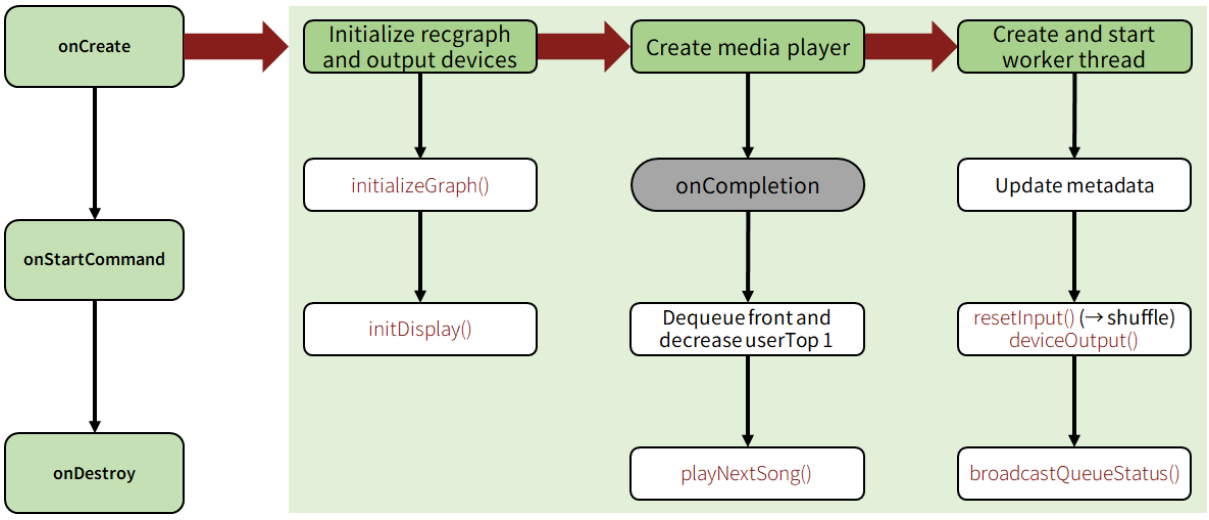
```
    else
    {
        /* If there is no command, get the button ID from the intent. */
        int buttonId = intent.getIntExtra("buttonId", -1);
        if (buttonId != -1 && queue.size() < 10)
        {
            /* Update the graph based on previously pressed and currently pressed buttons. */
            if (prevButtonId != -1)
                updateGraph(prevButtonId, buttonId);
            /* Add the button ID to the queue. */
            queue.add(buttonId);
            userTop++;
            if (userTop > 10)
                userTop = 10;
            prevButtonId = buttonId;
            /* If it is enqueued first into the empty queue, start playback. */
            if (!mp.isPlaying() && !isPaused)
                playNextSong();
        }
    }
    /* Recommend music. */
    recommendMusic();
    return START_STICKY;
}
```

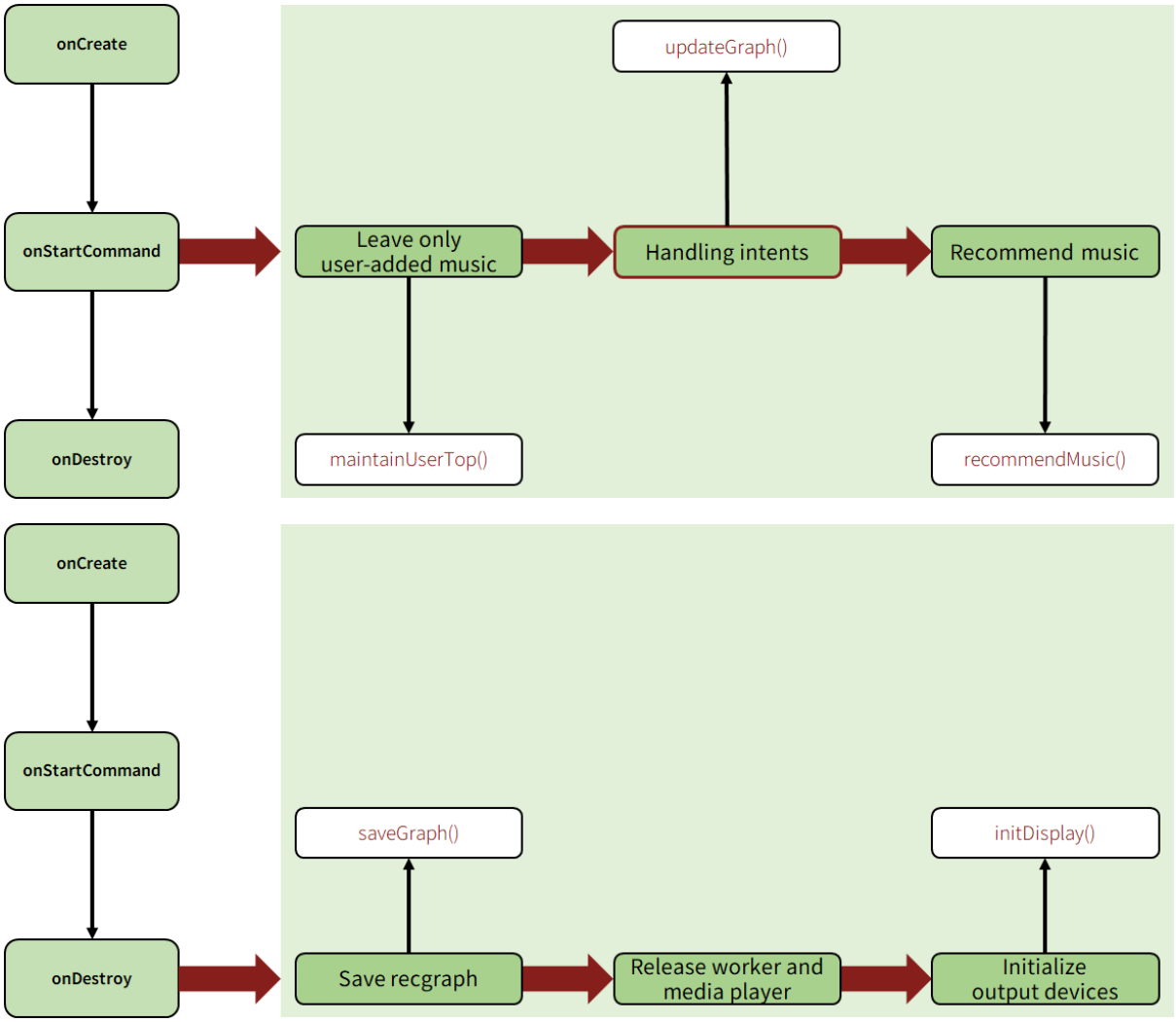
command가 null인 경우 음악 추가 버튼이 눌린 것이다. buttonID를 받아와서, 재생 목록에

추가한다. 이때, 이전에 추가된 음악과 현재 추가된 음악을 통해 추천 시스템의 recgraph를 업데이트한다. userTop을 1 증가시키고, 음악이 재생 목록에 맨 처음으로 들어왔다면 playNextSong을 호출한다. 여기까지의 과정을 마치면, 유저가 직접 추가한 음악만 재생 목록에 남아있는 상태이다. 이 재생 목록을 바탕으로 나머지 빈 칸을 추천 음악으로 채우기 위해 recommendMusic을 호출한다. START_STICKY는 Service가 강제로 종료되었을 때 시스템이 해당 Service를 자동으로 재시작하도록 하기 위해 설정한다.

```java
/* onDestroy */
@Override
public void onDestroy()
{
    super.onDestroy();
    /* Save recgraph. */
    saveGraph();
    /* Interrupt the worker thread to stop. */
    if (worker != null)
        worker.interrupt();
    /* Stop and release the media player. */
    mp.stop();
    mp.release();
    /* Initialize the output devices. */
    initDisplay();
}
```

서비스가 종료되면 호출되는 onDestroy 메써드이다. recgraph를 파일로 저장하기 위해 saveGraph를 호출하고, 워커 쓰레드에 인터럽트를 보내 종료한다. MediaPlayer도 멈춘 후 종료하고, 다시 output device들을 초기화한다. MusicService 클래스의 전체 구조도는 다음과 같다.





Service 내에서 사용하지 않는 메써드도 존재한다.

```
/* getAppDataFilepath - Method to get the file path for the app data */
public String getAppDataFilepath(String filename)
{
    File file = new File(getFilesDir(), filename);
    return file.getAbsolutePath();
}
```

getAppDataFilepath 메써드는 어플리케이션의 데이터를 저장할 수 있는 AppData 파일 위치를 알려주는 메써드이다. 이 메써드는 이후 C native 라이브러리에서 파일을 저장하기 위해 사용된다. Java로 구현된 어플리케이션 컨텍스트에서만 해당 파일 위치를 알 수 있기 때문에, Service 내의 메써드로 구현하였다. 이렇게 하면, Android 어플리케이션의 실질적인 동작은 모두 설계가 완료된다.

[3] recgraph.c (C Native Library)

```c
/* debug preprocessors */
#define DEBUG_LOGGING
#define SAVE_DATA_LOCAL_TMP

/* consts */
#define NUM_OF_NODES 10              /* number of nodes */
#define NODES_IDX (NUM_OF_NODES + 1) /* +1 (idx) */
#define QUEUE_SIZE 10                /* maximum queue size */
#define INT_MAX (int)0x7FFFFFFE

/* dat file path */
#ifdef SAVE_DATA_LOCAL_TMP
#define RECGRAPH_DAT "/data/local/tmp/_recgraph_.dat"
#else
#define RECGRAPH_DAT "_recgraph_.dat"
#endif

/* device file params */
#define MAJOR_NUM 242
#define DEV_NAME "music_driver"
#define DEV_FILE_LOC "/dev/music_driver"

/* IOCTL number */
#define IOCTL_OPTION _IOW(MAJOR_NUM, 20211584, char *)
```

소스 코드 최상단에 전처리 부분이 존재한다. 먼저, DEBUG_LOGGING이 정의되어 있으면 logcat을 통해 그래프 정보, 함수 호출 여부 등을 확인할 수 있다. SAVE_DATA_LOCAL_TMP가 정의되어 있으면, AppData 경로에 _recgraph_.dat을 저장하지 않고, /data/local/tmp/에 저장한다. recgraph와 재생 목록의 크기, 파일 저장 위치, 모듈 정보, ioctl 번호 등을 정의하는 부분도 존재한다.

```c
/* recgraph EDGE */
typedef struct _EDGE
{
    int node;
    int weight;
} EDGE;

/* global vars */
static int **graph;
static EDGE *maxEdge;
int module_fd = -1;
```

recgraph는 graph와 maxEdge로 관리된다. graph는 recgraph의 형태를 저장하는 adjacency matrix이고, maxEdge는 특정 노드로부터 최대 가중치 엣지로 도달할 수 있는 노드를 가중치와 함께 저장하는 1차원 배열이다. module_fd는 디바이스 파일의 file descriptor이다. 이 정의 및 변수들을 바탕으로 함수를 구현한다.

```c
/* printGraph - Function to print recgraph (for debugging). */
static void printGraph()
{
    int i, j;
    char nodes[NODES_IDX + 1] = {0};
    LOGD("[GRAPH]");
    for (i = 0; i < NODES_IDX; i++)
    {
        for (j = 0; j < NODES_IDX; j++)
        {
            nodes[j] = graph[i][j] + '0';
        }
        LOGD("%s", nodes);
    }

    LOGD("[MAX_EDGE]");
    for (i = 0; i < NODES_IDX; i++)
        LOGD("(%d, %d) ", maxEdge[i].node, maxEdge[i].weight);
}
```

printGraph 함수는 디버그 로깅을 위한 함수로, graph와 maxEdge를 logcat으로 확인할 수 있다. 하지만, 릴리즈 버전에서는 앞서 DEBUG_LOGGING이 정의되지 않기 때문에 호출되지 않는다.

```c
/* initializeGraph - JNI function to initialize recgraph. */
JNIEXPORT void JNICALL
Java_org_example_musicplayer_MusicService_initializeGraph(JNIEnv *env, jobject this)
{
#ifdef DEBUG_LOGGING
    LOGD("initializeGraph called.");
#endif
    int i, j;
#ifdef SAVE_DATA_LOCAL_TMP
    FILE *fp = fopen(RECGRAPH_DAT, "r");
#else
    /* Get the AppData path from the Java method. */
    jclass cls = (*env)->GetObjectClass(env, this);
    jmethodID mid = (*env)->GetMethodID(env, cls, "getAppDataFilepath",
                                        "(Ljava/lang/String;)Ljava/lang/String;");
    jstring filename = (*env)->NewStringUTF(env, RECGRAPH_DAT);
    jstring filepath = (jstring)(*env)->CallObjectMethod(env, this, mid, filename);
    const char *filepathStr = (*env)->GetStringUTFChars(env, filepath, NULL);
    FILE *fp = fopen(filepathStr, "r");
#endif
```

initializeGraph는 MusicService의 onCreate 메써드에서 호출된다. 라이브러리를 초기화하는 과정을 담고 있다. 가장 먼저, 저장된 _recgraph_.dat 파일을 AppData에서 연다. cls에 클래스를, mid에 getAppDataFilepath 메써드의 ID를 불러오고, filename에 “_recgraph_.dat”을 UTF-16 포맷으로 바꾸어 저장한다. 이후 마련된 인자들을 이용해 CallObjectMethod를 호출하고, 불러온 파일의 경로 문자열을 다시 C에서 사용할 수 있도록 UTF-8 포맷으로 바꾼다. 이 과정을 거치면, 어플리케이션의 AppData의 _recgraph_.dat 파일을 열 수 있게 된다.

```
    if (fp != NULL)
    {
        /* Allocate memory for the graph and maxEdge. */
        graph = (int **)malloc(sizeof(int *) * NODES_IDX);
        maxEdge = (EDGE *)malloc(sizeof(EDGE) * NODES_IDX);
        /* Read and store recgraph information from _recgraph_.dat. */
        for (i = 0; i < NODES_IDX; i++)
        {
            graph[i] = (int *)malloc(sizeof(int) * NODES_IDX);
            for (j = 0; j < NODES_IDX; j++)
            {
                fscanf(fp, "%d", &graph[i][j]);
            }
            fscanf(fp, "%d%d", &maxEdge[i].node, &maxEdge[i].weight);
        }
        fclose(fp);
    }
    else
    {
        /* Allocate memory for the graph and maxEdge. */
        graph = (int **)malloc(sizeof(int *) * NODES_IDX);
        maxEdge = (EDGE *)malloc(sizeof(EDGE) * NODES_IDX);
        /* Initialize graph and maxEdge. */
        for (i = 0; i < NODES_IDX; i++)
        {
            graph[i] = (int *)malloc(sizeof(int) * NODES_IDX);
            for (j = 0; j < NODES_IDX; j++)
                graph[i][j] = 1;
            maxEdge[i].node = i % QUEUE_SIZE + 1;
            maxEdge[i].weight = 1;
        }
    }
    /* Open the device file. */
    module_fd = open(DEV_FILE_LOC, O_RDWR);
```

이후 해당 파일이 존재하면 파일에서 정보를 불러와 라이브러리 내의 자료 구조에 저장하고, 존재하지 않으면 모든 엣지를 1로 초기화한다. 마지막으로 module_fd에 디바이스 파일의 file descriptor를 저장한다.

```
/* updateGraph - JNI function to update weights. */
JNIEXPORT void JNICALL
Java_org_example_musicplayer_MusicService_updateGraph(JNIEnv *env, jobject this,
                                                       jint prev, jint curr)
{
    /* Increases the edge weight by 1 between consecutively added music. */
    if (graph[prev][curr] < INT_MAX)
        graph[prev][curr]++;
    /* Update the maxEdge. */
    if (graph[prev][curr] > maxEdge[prev].weight)
    {
        maxEdge[prev].node = curr;
        maxEdge[prev].weight = graph[prev][curr];
    }
#ifdef DEBUG_LOGGING
    printGraph();
#endif
}
```

updateGraph는 MusicService에서 재생 목록에 음악이 추가될 때 호출된다. 연속적으로 추가된 음악에 대해, 두 노드를 연결하는 엣지의 가중치를 증가시킨다. 또, 이 과정에서 maxEdge의 갱신이 필요한 경우 수행한다. 예를 들어, 재생 목록에 2가 추가된 이후 바로 3이 추가되었

다면, 2→3 엣지의 가중치가 1 증가하게 된다. 이 정보는 이후 추천에 사용된다.

```
/* recommendMusic - JNI function to recommend music by recgraph. */
JNIEXPORT void JNICALL
Java_org_example_musicplayer_MusicService_recommendMusic(JNIEnv *env, jobject service)
{
    jclass cls = (*env)->GetObjectClass(env, service);

    /* Access the queue field. */
    jfieldID fid = (*env)->GetFieldID(env, cls, "queue", "Ljava/util/Queue;");
    jobject queue = (*env)->GetObjectField(env, service, fid); /* queue */

    /* Access the necessary class and method fields. */
    jclass queueCls = (*env)->GetObjectClass(env, queue);
    jclass integerCls = (*env)->FindClass(env, "java/lang/Integer");
    jmethodID sizeMethod = (*env)->GetMethodID(env, queueCls, "size", "()I");
    jmethodID addMethod = (*env)->GetMethodID(env, queueCls, "add", "(Ljava/lang/Object;)Z");
    jmethodID peekMethod = (*env)->GetMethodID(env, queueCls, "peek", "()Ljava/lang/Object;");
    jmethodID toArrayMethod = (*env)->GetMethodID(env, queueCls, "toArray", "()[Ljava/lang/Object;");
    jmethodID intValueMethod = (*env)->GetMethodID(env, integerCls, "intValue", "()I");
```

recommendMusic은 MusicService의 onStartCommand가 실행될 때마다 마지막에 호출된다. 재생 목록에 추천 음악을 자동으로 추가하는 동작이 담겨있다. 먼저, 추천을 위해 MusicService에서 참조해야 하는 필드와 여러 클래스 및 메써드를 저장한다. 여기에는 재생 목록(queue)과 이를 관리하기 위한 Queue, Integer 클래스, size, add, peek, toArray, intValue 메써드가 포함된다.

```
    /* Fill the queue with recommended music up to QUEUE_SIZE. */
    int size = (*env)->CallIntMethod(env, queue, sizeMethod);
    if (size == 0)
        return;
    while (size < QUEUE_SIZE)
    {
        jobjectArray q2array = (jobjectArray)(*env)->CallObjectMethod(env, queue, toArrayMethod);
        jobject lastElement = (*env)->GetObjectArrayElement(env, q2array, size - 1);
        int last = (*env)->CallIntMethod(env, lastElement, intValueMethod);
        /* Recommend the node with the largest weight that can be reached from last node. */
        int next = maxEdge[last].node;
        jobject nextElement = (*env)->NewObject(env, integerCls,
                                                (*env)->GetMethodID(env, integerCls, "<init>", "(I)V"),
                                                next);
        (*env)->CallBooleanMethod(env, queue, addMethod, nextElement);

        size = (*env)->CallIntMethod(env, queue, sizeMethod);
    }
}
```

이제 이를 통해 재생 목록에 추천 음악을 채워 넣는다. queue의 크기가 최대 크기를 넘지 않을 때까지, queue의 맨 뒤 요소를 확인해 해당 노드에서 최대 가중치 엣지를 통해 도달할 수 있는 노드를 맨 뒤에 넣는다. 이때, queue의 맨 뒤 요소를 확인하기 위해 toArray 메써드를, 새로운 음악을 삽입하기 위해 Integer 생성자(<init>)를 사용한다. 이 과정을 거치면, 유저 데이터를 통해 연속적으로 추가될 확률이 가장 높은 음악이 추천된다.

```c
/* saveGraph - JNI function to save the graph. */
JNIEXPORT void JNICALL
Java_org_example_musicplayer_MusicService_saveGraph(JNIEnv *env, jobject this)
{
#ifdef DEBUG_LOGGING
    LOGD("saveGraph called.");
#endif
    int i, j;
#ifdef SAVE_DATA_LOCAL_TMP
    FILE *fp = fopen(RECGRAPH_DAT, "w");
#else
    /* Get the AppData path from the Java method. */
    jclass cls = (*env)->GetObjectClass(env, this);
    jmethodID mid = (*env)->GetMethodID(env, cls, "getAppDataFilepath",
                                        "(Ljava/lang/String;)Ljava/lang/String;");
    jstring filename = (*env)->NewStringUTF(env, RECGRAPH_DAT);
    jstring filepath = (jstring)(*env)->CallObjectMethod(env, this, mid, filename);
    const char *filepathStr = (*env)->GetStringUTFChars(env, filepath, NULL);
    FILE *fp = fopen(filepathStr, "w");
#endif
    /* Write recgraph information to _recgraph_.dat. */
    for (i = 0; i < NODES_IDX; i++)
    {
        for (j = 0; j < NODES_IDX; j++)
        {
            fprintf(fp, "%d ", graph[i][j]);
        }
        fprintf(fp, "\n%d %d\n", maxEdge[i].node, maxEdge[i].weight);
    }
    fclose(fp);
```

saveGraph는 MusicService의 onDestroy에서 호출된다. recgraph의 정보를 파일에 저장하고, 할당한 메모리를 해제하는 동작을 수행한다. initializeGraph와 동일한 과정을 거쳐 파일을 열고, 해당 파일에 recgraph의 정보를 작성한다.

```c
    /* Free the memory allocated for the graph and maxEdge. */
    free(maxEdge);
    for (i = 0; i < NODES_IDX; i++)
    {
        free(graph[i]);
    }
    free(graph);
    /* Close the device file. */
    close(module_fd);
}
```

이후 동적으로 할당받은 메모리를 해제하고 디바이스 파일을 닫는다.

```c
/* resetInput - JNI function to get input from RESET switch. */
JNIEXPORT int JNICALL
Java_org_example_musicplayer_MusicService_resetInput(JNIEnv *env, jobject this)
{
    return read(module_fd, NULL, 0);
}
```

resetInput은 MusicService의 워커 쓰레드에서 RESET 스위치의 입력을 받기 위해 호출된다. 이 함수는 단순히 디바이스 파일의 read의 결과값을 반환한다.

```c
/* initDisplay - JNI function to initialize the output devices. */
JNIEXPORT void JNICALL
Java_org_example_musicplayer_MusicService_initDisplay(JNIEnv *env, jobject this)
{
    write(module_fd, NULL, 0);
}
```

마찬가지로 MusicService의 onCreate, onDestroy에서 output device들의 초기화를 위해 호출되는 initDisplay도 단순히 디바이스 파일에 write 연산을 한다.

```c
/* deviceOutput - JNI function to pass the necessary information to the output devices by IOCTL. */
JNIEXPORT void JNICALL
Java_org_example_musicplayer_MusicService_deviceOutput(JNIEnv *env, jobject service)
{
    int i;
    jclass cls = (*env)->GetObjectClass(env, service);

    /* Access the queue field. */
    jfieldID queueFid = (*env)->GetFieldID(env, cls, "queue", "Ljava/util/Queue;");
    jobject queue = (*env)->GetObjectField(env, service, queueFid); /* queue */

    /* Access the durationTime field. */
    jfieldID durationTimeFid = (*env)->GetFieldID(env, cls, "durationTime", "I");
    jint durationTime = (*env)->GetIntField(env, service, durationTimeFid); /* durationTime */

    /* Access the remainingTime field. */
    jfieldID remainingTimeFid = (*env)->GetFieldID(env, cls, "remainingTime", "I");
    jint remainingTime = (*env)->GetIntField(env, service, remainingTimeFid); /* remainingTime */
```

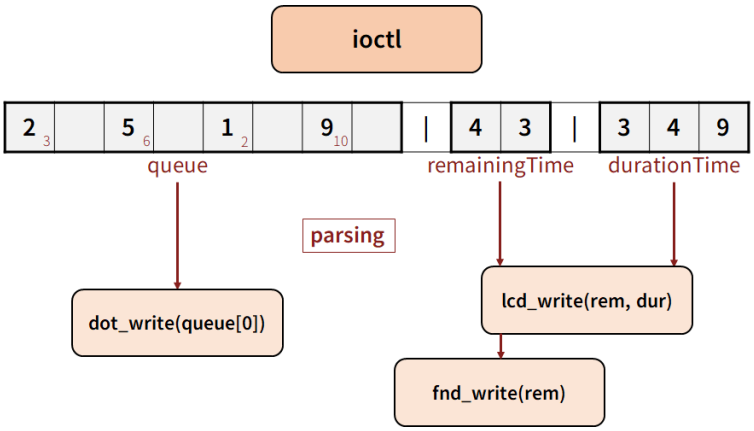
deviceOutput은 MusicService의 워커 쓰레드에서 output device들의 출력을 바꾸기 위해 호출된다. 우선 필요한 정보인 queue, durationTime, remainingTime을 저장한다.

```c
    /* Convert the four elements at the front of the queue to a string.
     * (e.g. |1|4|7|10|5|3|... → "0 3 6 9 ")
     */
    jclass queueCls = (*env)->GetObjectClass(env, queue);
    jmethodID toArrayMethod = (*env)->GetMethodID(env, queueCls, "toArray", "()[Ljava/lang/Object;");
    jobjectArray array = (jobjectArray)(*env)->CallObjectMethod(env, queue, toArrayMethod);
    jsize len = (*env)->GetArrayLength(env, array);
    char queueStr[13] = {0};
    for (i = 0; i < 4 && i < len; i++)
    {
        jobject element = (*env)->GetObjectArrayElement(env, array, i);
        jclass integerCls = (*env)->GetObjectClass(env, element);
        jmethodID intValueMethod = (*env)->GetMethodID(env, integerCls, "intValue", "()I");
        int value = (*env)->CallIntMethod(env, element, intValueMethod);

        char str[3];
        sprintf(str, "%d ", value - 1);
        strcat(queueStr, str);
    }

    /* Prepare the IOCTL arguments.
     * (e.g. "0 3 6 9 |392|983")
     */
    char ioctlArgs[33] = {0};
    sprintf(ioctlArgs, "%s|%d|%d", queueStr, remainingTime, durationTime);

    /* Send the IOCTL command. */
    ioctl(module_fd, IOCTL_OPTION, ioctlArgs);
}
```



ioctl parameter는 위와 같은 형태를 갖는다. 재생 목록의 맨 앞 네 곡, 음악의 남은 시간(초), 전체 시간(초)를 적절히 배치하여 하나의 문자열로 만든다. 해당 문자열을 ioctl을 통해 디바이스 파일에 쓰면서 동작을 마친다.

[4] Module

과제 2, 3에서 구현했던 디바이스 드라이버와 아주 유사한 형태이고, 이번 프로젝트에 필요한 동작만 수정하였다.

```c
void fnd_write(const int rem /* remaining time (sec) */)
{
    int rem_min = (rem / 60) % 100;
    int rem_sec = rem % 60;
    if (rem ≥ INT_MAX || rem < 0)
    {
        rem_min = 0;
        rem_sec = 0;
    }
    /* Concatenate min and sec to unsigned short value. */
    unsigned short int _s_value = ((rem_min / 10) << 12) + ((rem_min % 10) << 8) +
                                  ((rem_sec / 10) << 4) + (rem_sec % 10);
    /* Write value to FND. */
    outw(_s_value, (unsigned int)dev_addr[FND]);
}

void dot_write(const char cur /* current music number */)
{
    int i;
    int playing;
    if (cur == '#')
        playing = 10;
    else
        playing = (int)((cur - '0' + 1) % 10);

#pragma unroll 5
    /* Write pattern to DOT matrix. */
    for (i = 0; i < 10; i++)
        outw(dot_number[playing][i] & 0x7F, (unsigned int)dev_addr[DOT] + i * 2);
}
```

device.c에 구현된 FPGA 디바이스를 관리하는 코드들이다. 먼저, fnd_write는 rem (remaining time)을 표시하고, dot_write는 현재 재생 중인 음악의 번호를 표시한다. 만약 재생 중이 아니라면 FND에는 0000을, DOT에는 빈 칸을 표시한다.

```c
void lcd_write(const int rem /* remaining time (sec) */,
               const int dur /* duration time (sec) */)
{
    int i;
    unsigned char value[33]; /* Buffer to hold the formatted string to be displayed on LCD. */
    /* value[33] :
     * +----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
     * | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 |
     * +----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
     * | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |'\0'|
     * +----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
     * [00, 15] : indication whether music is currently playing or not
     * [16, 31] : played time / duration time (e.g. 02:13 / 04:42)
     */
    /* Initialize the buffer to empty. */
    memset(value, ' ', 32);
    value[32] = '\0';
```

lcd_write에는 음악이 재생 중인지 아닌지와 재생된 시간, 음악 전체 시간을 표시한다. 먼저, 표시할 값이 들어있는 버퍼를 공백으로 초기화한다.

```
    /* NOT PLAYING */
    if (rem < 0 && dur < 0)
        memcpy(value, notplaying, 16);
    else if (rem ≥ INT_MAX || dur ≥ INT_MAX)
        memcpy(value, notplaying, 16);
    /* PLAYING */
    else
    {
        memcpy(value, playing, 16);

        int prog = dur - rem; /* played time (sec) */
        int prog_min = (prog / 60) % 100;
        int prog_sec = prog % 60;
        int dur_min = (dur / 60) % 100;
        int dur_sec = dur % 60;
        value[16] = (char)(prog_min / 10 + '0');
        value[17] = (char)(prog_min % 10 + '0');
        value[18] = ':';
        value[19] = (char)(prog_sec / 10 + '0');
        value[20] = (char)(prog_sec % 10 + '0');
        value[21] = ' ';
        value[22] = '/';
        value[23] = ' ';
        value[24] = (char)(dur_min / 10 + '0');
        value[25] = (char)(dur_min % 10 + '0');
        value[26] = ':';
        value[27] = (char)(dur_sec / 10 + '0');
        value[28] = (char)(dur_sec % 10 + '0');
    }

    /* Write value to LCD. */
    unsigned short int _s_value = 0;
    for (i = 0; i < 32; i += 2) /* Combine two adjacent characters into
                                 * a short int and write it to the LCD.
                                 */
    {
        _s_value = (value[i] & 0xFF) << 8 | value[i + 1] & 0xFF;
        outw(_s_value, (unsigned int)dev_addr[LCD] + i);
    }
}
```

이후 재생 중이 아니라면 재생 중이 아니라는 알림을, 재생 중이라면 재생 중이라는 알림과 함께 시간 정보를 포매팅하여 버퍼에 저장한다. 이후 LCD 디스플레이에 버퍼의 내용을 띄운다.

```
/* m_read - Replace 'read' fop on device file. */
static int m_read(struct file *file, char __user *buffer, size_t length, loff_t *offset)
{
    /* Read the input from the RESET switch and return it. */
    return switch_read();
}
```

```
int switch_read()
{
    unsigned char dip_sw_value;
    unsigned short _s_dip_sw_value;
    /* Read switch value, */
    _s_dip_sw_value = inw((unsigned int)dev_addr[SWITCH]);
    dip_sw_value = _s_dip_sw_value & 0xFF;
    /* and return 1 if switch is on, 0 otherwise. */
    if (!dip_sw_value)
        return 1;
    else
        return 0;
}
```

module.c에는 디바이스 파일에 행해지는 file operation에 대한 처리가 작성되어 있다. read

의 경우 switch_read의 결과를 그대로 반환하는데, switch_read에선 RESET 스위치의 입력을 받아 눌린 경우 1을, 눌리지 않은 경우 0을 반환한다. 이를 통해 스위치가 눌렸는지 눌리지 않았는지를 알 수 있다.

```c
/* m_write - Replace 'write' fop on device file. */
static int m_write(struct file *file, char __user *buffer, size_t length, loff_t *offset)
{
    /* Initialize display devices. */
    m_display_init();
    return 0;
}
```

```c
/* m_display_init - Initialize the output devices. */
static void m_display_init()
{
    fnd_write(0);
    led_write(0);
    dot_write('#');
    lcd_write(-1, -1);
}
```

write의 경우 모든 output device들을 초기화하는 m_display_init 함수를 호출한다.

```c
/* m_ioctl - Replace 'ioctl' fop on device file. */
static long m_ioctl(struct file *file, unsigned int ioctl_num, unsigned long ioctl_param)
{
    int i, j;
    if (ioctl_num == IOCTL_OPTION)
    {
        /* Cast the ioctl parameter to a char pointer. */
        char *u_option = (char *)ioctl_param;
        char opt_buf[34] = {0}; /* buffer to hold the option string */
        /* Copy the user option string to the buffer. */
        if (strncpy_from_user(opt_buf, u_option, 33) < 0)
        {
            printk("ERROR(module.c) : strncpy_from_user failed\n");
            return -1;
        }
```

ioctl에선 앞서 포매팅한 ioctl parameter를 어플리케이션으로부터 받아온다.

```c
char queue[4] = {'#', '#', '#', '#'}; /* array to hold the queue */
int queue_top = 0;                    /* the number of elements of the queue */
/* Loop to parse the queue from the option string. */
for (i = 0; opt_buf[i] != '|'; i = i + 2)
    queue[queue_top++] = opt_buf[i];
i++;

int rem = 0; /* remainingTime */
/* Loop to parse the remaining time from the option string. */
for (; opt_buf[i] != '|'; i++)
    rem = rem * 10 + (int)(opt_buf[i] - '0');
i++;

int dur = 0; /* durationTime */
/* Loop to parse the duration from the option string. */
for (; opt_buf[i] != '\0'; i++)
    dur = dur * 10 + (int)(opt_buf[i] - '0');

/* Write the first element of the queue to the dot matrix. */
dot_write(queue[0]);
/* Write the playing time(dur - rem) and duration to the LCD */
lcd_write(rem, dur);
/* Write the remaining time to the FND */
fnd_write(rem);

return 0;
```

이후 이 문자열을 파싱하여 output device들에게 필요한 값을 전송한다.

4. 연구 결과

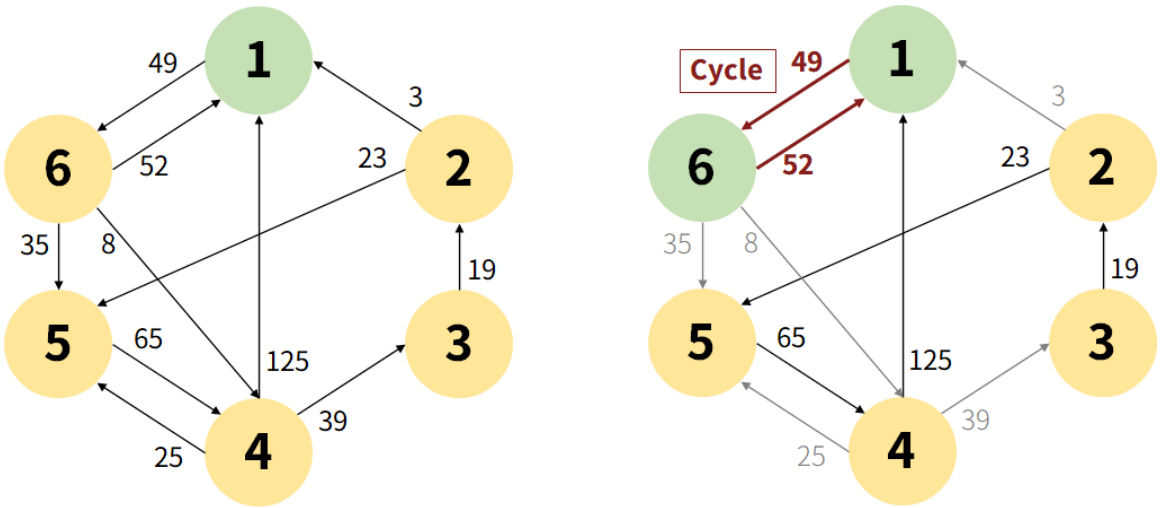
설계한 것과 완전히 동일하게 어플리케이션이 작동하였다. **(자세한 시연은 발표를 참고해주세요. 보고서에서 설명이 힘든 부분은 발표 자료에 설명해두었습니다.)**

5. 후속 연구 계획

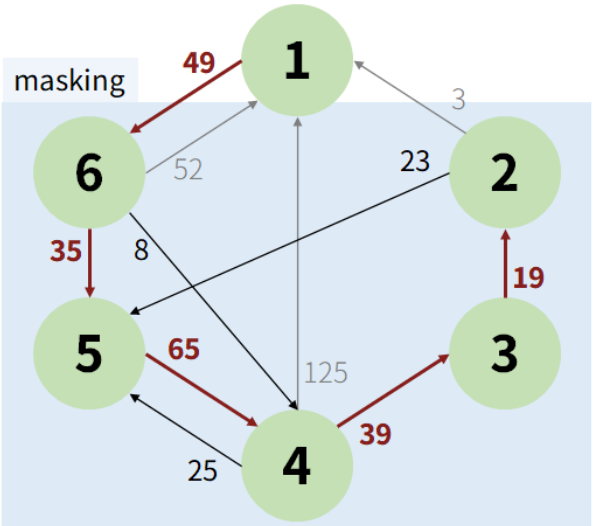
프로젝트를 진행하면서 집중한 부분은 수업 시간에 주로 학습한 (1)리눅스 커널 모듈 프로그래밍, (2)안드로이드 어플리케이션 프로그래밍(Activity, Service), (3)JNI C/C++ 프로그래밍이었다. 이 부분에 집중하여 구현을 하다 보니, 최종 결과물의 나머지 부분에 있어 아쉬움이 많았다.

[1] recgraph 자료 구조 및 알고리즘

recgraph는 adjacency matrix로 관리되는 그래프로, 큐의 마지막 노드에서 시작해 최대 가중치 엣지를 건너가며 음악을 추천한다. 이는 추천 시스템을 아주 간략하게만 구현한 것으로, 네이티브 라이브러리 자체에서 관리하는 자료 구조를 만드는 것에 의의를 두었기 때문에 더 개선하지 않았다. 하지만 현재의 나이브한 알고리즘 구현은 다음과 같은 문제를 초래할 수 있다.



만약 recgraph가 왼쪽과 같다면, 최대 가중치 엣지를 따라간 추천 경로는 오른쪽과 같을 것이다. 이렇게 추천 경로에 사이클이 생기면, 어떤 음악은 영영 추천되지 않는 등 다양하게 음악을 추천할 수 없다.



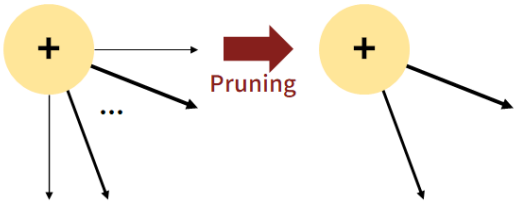
Directed maximum spanning tree
on recgraph excluding 1 (source node)
optimal $O((V+E)\log E)$ *(subquadratic)*

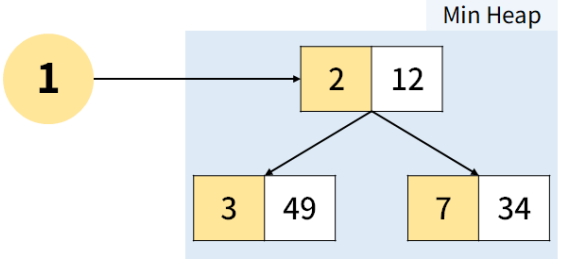
현재의 나이브한 구현에서 가장 쉽게 떠올릴 수 있는 개선 방법은 시작 노드를 제외한 그래프에서 directed MST(Maximum Spanning Tree)를 찾는 것이다. 이렇게 하면 경로에 사이클이 생기지 않아 큐에 특정 패턴이 반복될 일이 없다.

그러나, 애초에 큐의 마지막 음악만을 갖고 추천 경로를 생성하는 것은 좋은 사용자 경험을 주기 힘들다. 아주 단순한 유저 데이터 만으로 static 하게 결정된 그래프에서, 하나의 노드만을 참고하여 음악을 추천하는 것은 항상 동일한 결과를 낼 확률이 높기 때문이다.

Determining weights (Heuristic)

Multiple continuous additions within a short period of time
No continuous addition again after they have been continuous once
Repeated additions after a particular cluster(long-term memory)
…





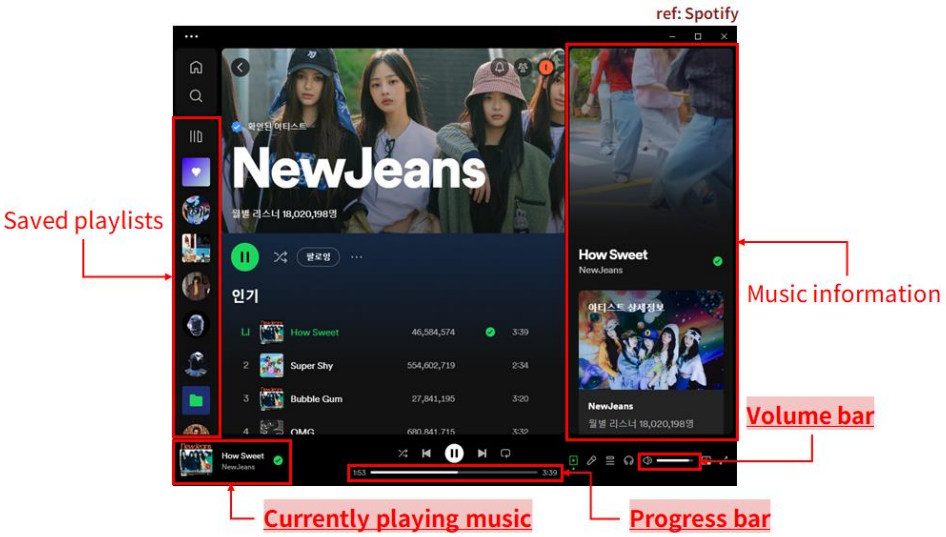
실제 추천 시스템에서는 훨씬 더 복잡한 유저 데이터 및 음악 정보를 활용해 추천을 진행하고, 최근에는 빅 데이터 및 인공 지능을 활용하여 추천 시스템을 평가 및 개선한다. 이를 나의 환경에서 조금이라도 모방해보기 위해서, 유저 데이터를 더 세밀하게 분석하여 해당 metric마다 다른 가중치를 부여하는 휴리스틱을 사용할 수 있다. 짧은 시간 동안 여러 번 연속 추가된 음악이 있다면 해당 음악 사이 엣지에 양의 보정을, 한 번 연속 추가된 이후 다시 추가되지 않고 있다면 흐른 시간에 비례하는 음의 보정을 하는 등의 방법이 있을 수 있다. 또, 연속 추가된 두 노드 간의 관계만 확인하지 않고, 여러 노드를 포함하는 집합에 대한 추천을 진행할 수도 있다. 예를 들어, 세 개 이상의 노드가 서로 SCC(Strongly Connected Components)를 이룬다면, 해당 SCC 내 엣지의 가중치 총합이 클러스터의 점수가 되어 이를 하나의 노드로 취급할 수도 있을 것이다. 또, 이런 다양한 알고리즘 연산을 효율적으로 진행하기 위해 adjacency matrix의 관리 형태를 바꿀 수 있다. 실제 환경처럼 음악이 많아진다면 해당 방법은 시간 및 공간 복잡도에 있어 매우 좋지 않은 효율을 보이기 때문에, 최대 N 개의 상위 가중치 엣지만을 관리하는 adjacency list 등으로 교체한다면 좋은 효율을 기대할 수 있을 것이다.

[2] worker Thread의 synchronization issue

```
/* Create a new worker thread. */
worker = new Thread(new Runnable()
{
    @Override
    public void run()
    {
        int prevValue = 0;
        while (true)
        {
            /* When the reset button is pressed,
             * shuffle the queue and play the song at the front of the queue.
             */
            int value = resetInput();
            if (value == 1 && prevValue == 0)
            {
                List<Integer> q2list = new ArrayList<Integer>(queue);
                Collections.shuffle(q2list);
                queue = new LinkedList<Integer>(q2list);
                userTop = queue.size();
                playNextSong();
            }
            prevValue = value;
            // ...
```

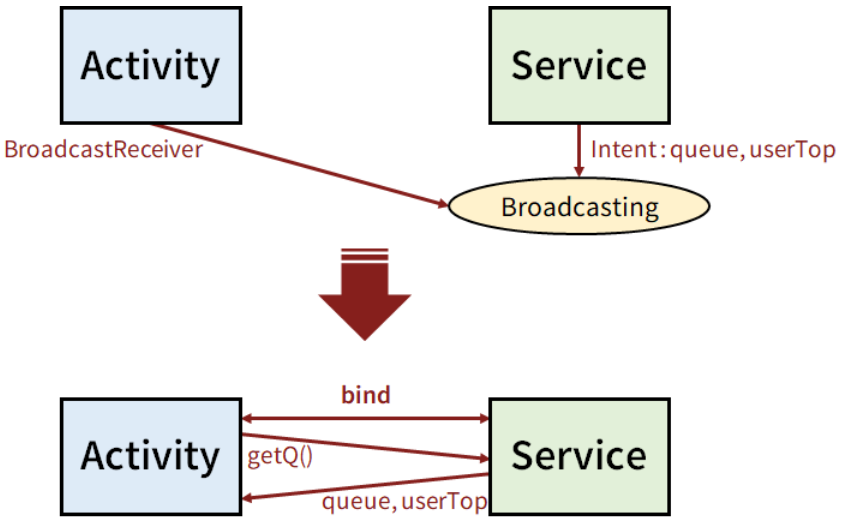
MusicService에서 사용하는 워커 쓰레드의 코드이다. Java의 구동 방식을 정확히 알진 못하지만, 만약 C의 쓰레드와 동일한 방식으로 동작한다면 잠재적인 동기화 문제가 발생하도록 구현되어있다. Java는 동기화를 위해 synchronized()를 사용하는 것으로 알고 있는데, 이에 대해 더 확실하게 공부하여 동기화 문제가 없는 코드로 수정하고 싶다.

[3] 추가 기능 및 UI



위 사진은 현재 가장 많이 사용되는 음원 스트리밍 어플리케이션인 스포티파이의 UI이다. 사운드 볼륨 바, 프로그레스 바, 현재 재생중인 음악의 커버 이미지, 음악 정보 창, 유저가 저장한 플레이스트 목록 등 다양한 기능을 지원하는 것을 확인할 수 있다. 이번 프로젝트에서 구현한 어플리케이션은 꾸밈 없이 음악 목록(1~10), 재생, 일시정지, 건너뛰기, 종료 버튼에 재생 목록 큐만 보여주는 아주 투박한 형태이다. 유저에게 잘 만들어진 어플리케이션처럼 보이기 위해선, 앞서 설명한 추가 기능 및 UI가 필요할 것 같다. 실제로 현재 재생중인 음악의 커버 이미지, 프로그레스 바, 사운드 볼륨 바는 레이아웃에 구현하기 위해 노력했었으나, 시간도 부족하고 Android SDK 버전(Kitkat)과 호환되지 않는 요소가 많아 포기하였다. 상위 버전에서 안드로이드 어플리케이션을 개발할 일이 생긴다면 개선하고 싶다.

[4] Remote/Bound Service 적용



이번 프로젝트에서는 기본적인 unbound service만 이용하여 MusicService 클래스를 구현하였다. 이용하지 않은 remote/bound service를 추가하여 어플리케이션을 개선할 수 있다. 특히, 기존에는 intent broadcasting을 통해 queue의 정보를 service에서 activity로 전송했는데, 이를 bound service를 활용하여 간단한 형태로 바꿀 수 있다. Activity와 service를 bind 하고, activity에서 필요 시에 getQ()와 같은 메써드를 서비스에 호출해 정보를 받아올 수 있다.

***** 한 학기 동안 수고 많으셨습니다. *****