TECHNICAL SPECIFICATIONS

MP3 Player

DATE: March 15, 2020
AUTHOR: Kevin Egedy

Feature Set Description

All features for the MP3 player are listed in the `CommandEnum`. Features that begin with `INPUTCOMMAND` are driven by the touch driver. The touch buttons are `PREV`, `PLAY`, and `NEXT` and are the only inputs for the system. These buttons send all instructions to `CommandTask`, the overlying task that controls the display and streaming tasks.

```
typedef enum {
    INPUTCOMMAND_NONE,
    INPUTCOMMAND_NEXTSONG,
    INPUTCOMMAND_PLAY,
    INPUTCOMMAND_PREVSONG,
    SONG_COMPLETE,
} CommandEnum;
```

`INPUTCOMMAND_NONE`: command when a touch input is not mapped to any of the buttons defined. It has no assigned tasks however it is a useful command for debugging.

`INPUTCOMMAND_PREVSONG`: command mapped to the `PREV` button. It brings the streaming buffer pointer `pStream` to the beginning of the current song defined in `curNode`. Future task: if the `PREV` button is hit twice quickly, then go back one song in list. If `PREV` button is held for 1 second, then rewind. Also requires doubly linked list.

`INPUTCOMMAND_PLAY`: command mapped to the `PLAY` button. This button toggles music streaming defined by `STREAMING` and `rxFlags`. Bit 1 in `OSFlagGroup` should be set if `STREAMING` is true. It is the only input that can start or stop streaming.

`INPUTCOMMAND_NEXTSONG`: command mapped to the `NEXT` button. It brings the streaming buffer pointer (`pStream`) to the beginning of the current song, and increments to the next song defined in the linked node list. Future task: if the `NEXT` button is held for 1 second, then fast-forward.

`SONG_COMPLETE`: command mapped to the streaming task. Once streaming task completes (`curNode→data.pos == curNode→data.size`), send a message to the command mailbox `mboxCommand`. Streaming continues to the next song defined in the linked node list.

Input Buttons follow the Spotify design. The play button toggles from pause to play depending if a song is streaming. Streaming is defined by the global variable `STREAMING (OS_FALSE/OS_TRUE)` and bits 0/1 in `rxFlags`. The `PREV` and `NEXT` buttons blink to provide user feedback. Display tasks are more significant than streaming tasks and prevent streaming to continue until it is finished. Before a display task is called, bit 0 in `OSFlagGroup` should be set.

All of the features are achieved using the attached tasks and OS events.

```
// Task prototypes
void CommandTask(void* pdata);     // priority 5
void ProgressTask(void* pdata);    // priority 6
void Mp3Task(void* pdata);         // priority 7
void ProgressTask(void* pdata);    // priority 8
void DisplayTask(void* pdata);     // priority 9

// OS Events
mboxCommand = OSMboxCreate((void*)NULL);
mboxProgress = OSMboxCreate((void*)NULL);
mboxDisplay = OSMboxCreate((void*)NULL);
rxFlags = OSFlagCreate((OS_FLAGS)0, &err);
```

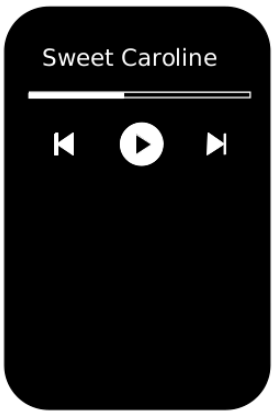

Figure 1: Buttons & Display

Streaming & Song Management

Streaming and song management is defined by structs `SongGroup` and `Node`. The `SongGroup` holds the information necessary per song for the display and streaming tasks. Struct `Node` enables each song to be linked to the next. Each song only knows the song in front of it via the `next` pointer. An example is provided.

```
struct SongGroup{
    char *title;
    INT32U size;
    INT32U pos;
    INT8U *pStart;
    INT8U *pStream;
};

struct Node {
    SongGroup data;
    Node* next;
};

struct SongGroup group0;
struct Node *curNode;
struct Node node0;
struct Node node1;

group0.title = "Train Crossing";
group0.size = sizeof(Train_Crossing);
group0.pos = 0;
group0.pStart = (INT8U*)Train_Crossing;
group0.pStream = group0.pStart;
node0.data = group0;
node0.next = &node1;

curNode = &node0;
```

The streaming task grabs 32 bytes every iteration and uses variable `curNode→data.pos` and `curNode→data.pStream` to track its position. Once streaming completes, `curNode→data.pos == curNode→data.size`, then `curNode` updates to the next node (`curNode = curNode→next)`. After the last song completes, the node list wraps back to the beginning.

Overall, the system uses mailboxes to send its instructions, however `rxFlags` is used to toggle streaming. In most mailboxes, the task pends (`OSMboxPend`) until a message is received in order to perform an instruction. In `Mp3Task`, I wanted to achieve the opposite effect. I designed the streaming task to continue unless told otherwise. I implemented this design using the `OSFlagPend` method and `OS_FLAG_WAIT_CLR_ALL` for `bit0` and `bit1`.

The generator for the progress bar feature is also included in this task. In every iteration, `curNode→data.pos` is sent to `mboxProgress` as an 8 bit integer. `ProgressTask` contains the mailbox and calls `DrawProgress` to accurately draw the song progress

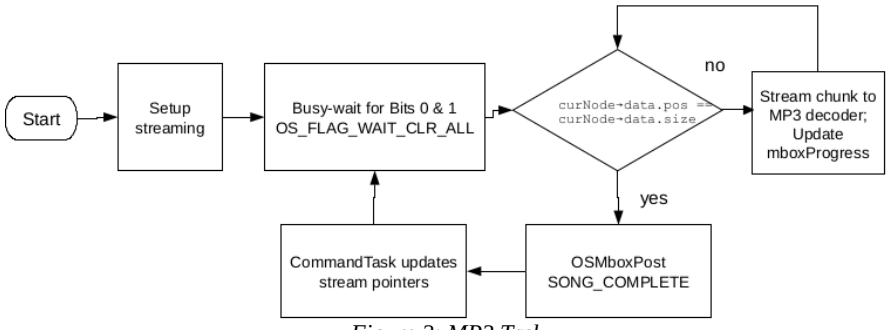

Figure 2: MP3 Task

Input & Display Task

The `TouchInputTask` is the generator for the system and creates messages for the `CommandTask`. Similar to the `Mp3Task`, `rxFlag` is necessary to suspend touch polling until display instructions are finished. This prevents duplicate commands from being processed and provides a good user experience. If no input is received, a non-block time delay `OSTimeDly` is used before polling again.

In the design, I chose `ProgressTask` to be separate from `DisplayTask`. They both update the display, but I created the addition task due to the receive message type. `ProgressTask` receives `INTU8` messages and `DisplayTask` receives `CommandEnum` messages. `CommandTask` is the handler that provides instructions for all the remaining tasks, however `ProgressTask` receives its instructions from the `Mp3Task` directly.

Structs `Point`, and `Grid` were implemented to maintain the display layout. The `CommandTask` calls `DrawLcdContents` to draw the initial layout of the screen. This function uses the attached helper functions in order to achieve this. As mentioned before, the display task handles the instructions from `CommandTask` and then clears bit 0 in `rxFlag` so that streaming and touch polling can continue.

```
// DEFINE LOCATIONS FOR INPUTS
typedef struct{
    int16_t X, Y;
} Point;

typedef struct{
    Point Title;        // Song Title
    Point Progress;     // Song Progress
    Point TL, TM, TR;   // Button Layout
} Grid;
static Grid grid;

// Define Grid; (X,Y) is middle of container
grid.Title.X    = 120; grid.Title.Y    =  30;
grid.Progress.X = 120; grid.Progress.Y =  80;
grid.TL.X       =  50; grid.TL.Y       = 120;
grid.TM.X       = 120; grid.TM.Y       = 120;
grid.TR.X       = 190; grid.TR.Y       = 120;

// Helper functions
void DrawTitle(char *title);
void DrawProgress(int8_t percentage);
void ResetProgress(void);
void TogglePlayBtn(void);
void DrawNextBtn(void);
void DrawPrevBtn(void);
void DefineBtns(Adafruit_GFX_Button *buttonCtrl, int16_t x,
                  int16_t y, CommandEnum CMD);
```

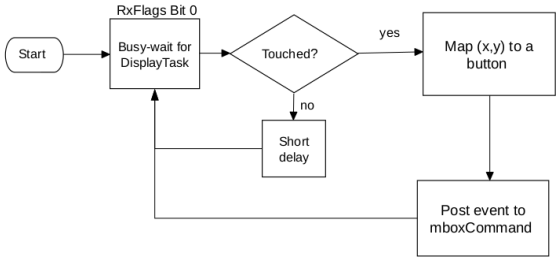

Figure 3: Input Task

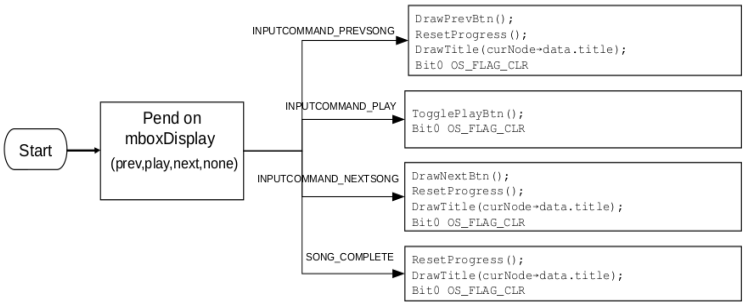

Figure 4: Display Task

Command Task

The main purpose of `CommandTask` is to update `rxFlags` and update the streaming pointers. It pends on `mboxCommand` to receive instructions from `InputTask` and performs a pattern of operations. In every command, `Bit0` in `rxFlags` is set to turn off touch inputs. Then if the command is `INPUTCOMMAND_NEXTSONG`, `INPUTCOMMAND_PREVSONG`, or `SONG_COMPLETE`, it uses the attached code to update the streaming pointers.

```
// update streaming pointers
curNode->data.pos = 0;
curNode->data.pStream = curNode->data.pStart;
curNode = curNode→next;
```

If the command is `INPUTCOMMAND_PLAYSONG`, then `Bit1` in `rxFlags` is toggled according to `STREAMING` status. Lastly, all commands forward their instruction to `mboxDisplay`. See Figure 4 to find how each display instruction is handled. Also, it is important to understand the distinction between `CommandTask` and `Mp3Task`. The `Mp3Task` does the work to decode the file, and `CommandTask` updates the `curNode` pointer for `Mp3Task` to read.

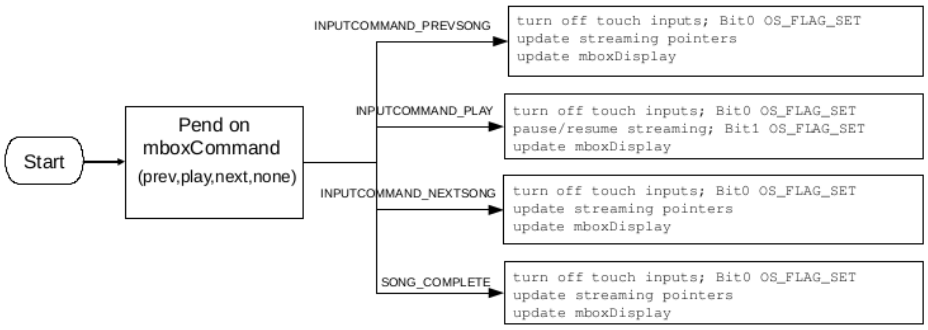

Figure 5: Command Task