Jean-Philippe Lassonde

Developing a touchscreen driven music player with C++ and FreeRTOS

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Bachelor's Thesis

7 May 2022

Abstract

Author:	Jean-Philippe Lassonde
Title:	Developing a touchscreen driven music player with C++ and FreeRTOS
Number of Pages:	37 pages
Date:	7 May 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Smart Systems
Supervisors:	Keijo Länsikunnas, Principal Lecturer

This thesis describes the research and work done during each step of the development of a touch-screen music player on a Cortex-M microcontroller. Using FreeRTOS and developed in C++ the resulting software perform internal and external peripherals configuration, present a graphical user interface and process inputs on a touchscreen, retrieve files stored on an SD card, parse a tracker music format, control programmable sound generators installed on an expansion board and process the resulting signal with an audio codec.

The creation of the UI engine is thus described, together with an overview of the various other components making up the project and the difficulties encountered and overcome.

Keywords: C++, FreeRTOS, Cortex-M

Contents

List of Abbreviations

ADC:	Analog-to-digital converter
BSP:	Board support package
CMSIS:	Cortex Microcontroller Software Interface Standard
Codec:	Coder-decoder
CPU:	Central processing unit
DMA:	Direct memory access
DSI:	Display serial interface
FMC:	Flexible memory controller
GRAM:	Graphics random-access memory
GUI:	Graphical user interface
HAL:	Hardware abstraction layer
LCD:	Liquid-crystal display
LTDC:	Liquid-crystal thin-film-transistor display controller
MCU:	Microcontroller unit
PCB:	Printed circuit board
PGA:	Programmable gain amplifier
PSG:	Programmable sound generator

RAM:	Random-access memory
SAI:	Serial audio interface
SDMMC:	Secure Digital / MultiMediaCard
SDRAM:	Synchronous dynamic random-access memory

Introduction

Advancements in microcontroller unit (MCU) technologies allowed them both to enter markets traditionally reserved to application processors based systems and provide a better user experience to simpler systems. While their overall performances remain more modest, the increase in clock rates, the integration of memory caches and advanced peripherals, to name a few, now make it possible to design and deploy feature-rich graphical user interface (GUI) applications with a lower total system cost and footprint.

A plethora of free and commercial GUI tools already exist for the microcontroller market. Even though some offer a high level of customization, the level of abstraction introduced by the framework itself does take away some freedom from the developer. The idea of this thesis is both to create a simple, lightweight but expendable GUI engine and implement a functional application, a music player, with it. Thus, the main requirements for the GUI are to be able to present the user with directories and files from which an audio track can be selected, present the information about the track being played and provide the common music player controls (volume settings, previous, next, skip to a specific part).

The chosen music format for playback is Pro Tracker 3.x for the ZX Spectrum 128 home computer. These computers used to run on very limited resources and were equipped with rudimentary audio hardware: programmable sound generators (PSGs). Through a set of registers, these PSGs can be controlled to emit square waves and noise modulated with a simple envelope generator. Tracker formats were eventually developed to both facilitate the creation of music and minimize the memory usage for the audio data, with predefined effects taking parameters and a general design facilitating data reuse. In this case the much smaller size compared to digital audio makes it possible to store an entire file in memory while leaving most of the space for the graphics buffering.

Thus, an interpreter for the tracker format and an expansion board to control two AY-3-891x PSGs are two of the other main components of the project. Finally, an audio coder-decoder (codec) is interfaced for the PSGs output amplification, audio processing and to provide a volume control to the user.

1 Material: hardware and software components

1.1 STM32F769NIH6 microcontroller

The project was targeting the STM32F769NIH6, a microcontroller with an ARM Cortex-M7 processor core which is, currently, the latest and highest performance core of the Cortex-M series. Amongst the improvements of the M7 core compared to its M4 predecessor are the pipeline, the branch predictor, the memory cache and wider data & instruction busses [1].

The 3-stages pipeline (fetch / decode / execute-writeback) is replaced by a dual-issue 6-stages superscalar one. With the dual-issue pipeline and the busses enlarged from 32 bits to 64, when conditions allow, two instructions can be executed in parallel. In the first three stages, in parallel in the absence of inter-dependency, the instructions are fetched, decoded, then split into micro-operations before being issued to the appropriate execute unit. Then for the last 3 stages, they are processed in one of the two Load/Store units, one of the two arithmetic logic unit (ALU), the multiply-accumulate (MAC) unit, the branch unit or the floating-point unit (FPU). Since keeping a throughput of 2 instructions per cycle is not always possible either because of data stall, a mispredicted branch, an interrupt or having only a single MAC and FPU, the performance increase due to this architecture is harder to evaluate and situation specific. Breaking down the pipeline in smaller stages also allows slightly higher clock rates.

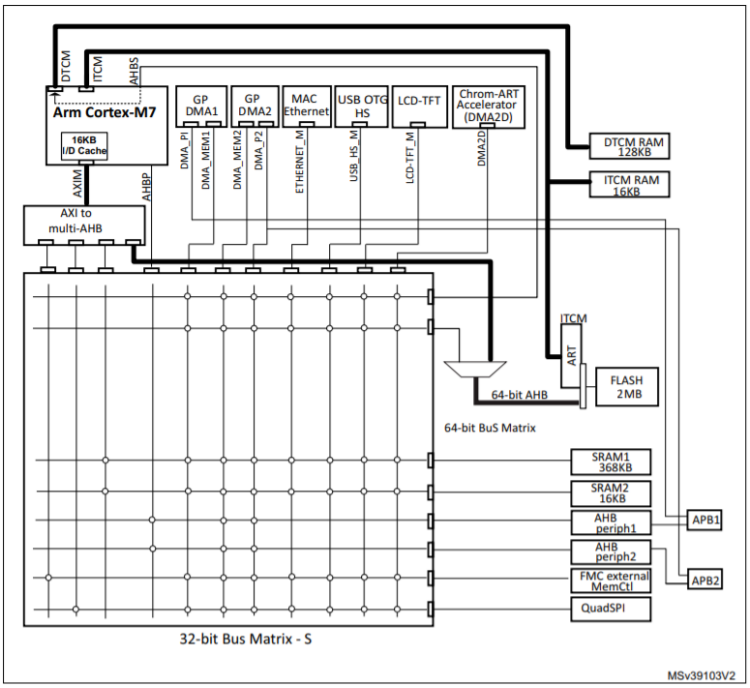
To mitigate the higher branch penalty of a superscalar & deeper pipeline a branch predictor was added, combining static prediction established at compile time and dynamic prediction based on execution history through a branch target address cache (BTAC). Finally, data and instruction layer 1 caches were added

in the core, reducing the flash and SRAM access frequency in loops or often used code and data.

From the application developer standpoint most of these changes occurs transparently, being taken care of either by the compiler or the core itself. The only exception is the data cache which must be carefully managed. In situations where the direct memory access (DMA) controllers are used to transfer data between memory and peripheral without the use of the central processing unit (CPU) the cache coherency must be managed by software, so both the CPU and the DMA controllers see the same data [2]. Thus, before starting DMA transfer from memory to peripheral the modified data stored in cache must be written back to memory through cache flushing and after transfers from peripheral to memory the stale data in cache must be invalidated before usage. It is also possible to disable caching for specific sections of memory, but the increased latency costs make it less desirable in situations where the data is often used by the CPU. A third option on the Cortex-M7 is to globally force write-through [1], synchronizing writes to the cache and the back-store. While not preventing stale data in the cache, it guarantees that a memory to peripheral DMA transfer would always use up to date data. A software managed flush-invalidate strategy was adopted for the most part of this project before and after initiating DMA transfers, except for one circular buffer which was marked as non-cachable.

Beyond the ARM processor, the STM32F769NIH6 contains a variety of peripherals, memory banks and controllers as pictured in the figure on the next page. Thin lines being 32-bit busses while thick lines 64-bit ones.

Figure 1. STM32F769NIH6 bus architecture [3, p. 23]



Bus masters, including the processor, are displayed horizontally on the top part of figure 1. Bus slaves, for their part, are arranged vertically on the right part. Connected to the CPU through 64-bit busses are data and instruction tightly-couple memory blocks (DTCM and ITCM RAM), accessible by the processor without latency [4, p. 8]. Data tightly coupled memory can also be accessed by the other masters via the Cortex-M7's advanced high-performance bus slave (AHBS). Flash memory, for its part, has three interfaces connecting it to the masters. While write access from the processor is done through a 64-bit AHB, read-only accesses can reach higher performance by using the ITCM interface with the adaptive real-time (ART) accelerator which manage prefetch, instruction and constants data caching as well as a branch cache, granting a zero wait state equivalent access to the flash [4, pp. 11-12]. The other masters, for their part, have access to the flash memory through the 32-bit bus matrix. To conclude with the embedded memory, two other internal SRAM blocks (368kB and 16kB) are available to all masters via the bus matrix.

In addition to the processor, four other masters were relevant to this project: the two general purpose DMA controllers, the liquid-crystal thin-film-transistor display controller (LCD-TFT or LTDC) and the Chrom-ART accelerator (DMA2D). The general-purpose DMA controllers operate standard transfers: memory-to-memory (DMA2 only) [4, pp. 18-19], peripheral-to-memory and memory-to-peripheral. The other two are specialized for graphical applications: the LTDC to send data from memory to a display controller and the DMA2D to facilitate the composition of the frames in memory.

During memory transfer to the display peripheral, the LTDC is configurable to operate pixel format conversion, blending multiple layers, perform dithering as well as insert the synchronous timings required by the display: vertical sync, horizontal sync and the inactive zones, the front and back porches [5, pp. 590-600]. Layers are configurable to update only a section of a frame with automatic offset calculations. This way, with the framebuffer start address, its dimensions and pixel format configured, it can transfer a subsection in a single transfer, skipping automatically the addresses of the pixels forming the horizontal and vertical lines not included in that subsection.

The DMA2D has the pixel format conversion, blending and offset calculations features as well, but its purpose is for memory-to-memory transfers, as well as register-to-memory transfer to fill a section with a uniform colour [5, pp. 280-287]. It is used extensively in this project to transfers assets in full or in parts from the flash memory to the framebuffer. The pixel format conversion was particularly useful to store assets like using one byte per pixel, set the desired output colour in the DMA2D register then transfer them to the 32 bytes per pixel framebuffer, considerably reducing the size of the graphic data.

Going back to the slaves, beside the embedded memory discussed before, a configurable flexible-memory-controller (FMC) is used to interface different types of external memory, allowing transparent addressing as if the memory was internal [5, p. 332]. The other peripherals like timers, communication

interfaces and analog-to-digital converter (ADC) are located on the advanced peripheral busses (APB1 and APB2) [3, p. 20].

With this architecture in mind, it makes sense in a transfer-heavy application like this one to use the appropriate controllers for large memory transfers, reducing the processor usage, as well as balance the load between the RAM blocks and the two general purpose DMA controllers. As the different paths of the bus matrix can be used simultaneously without the need for bus arbitration [4, p. 8], different transfers have the potential to be executed at maximal speed if they don't need simultaneous access to the same resource. It is worth noting as well that the ITCM RAM is a good place to place non-cachable data since the processor still have zero-wait access time to it, although it must be used carefully for large transfers, as the simultaneous access by another controller would force bus arbitration [4, pp. 44-45].

1.2 Development board

The STM32f769i Discovery kit was chosen to develop the project as it is relatively low cost, has a small form factor and in addition to the MCU, already contains most of the support components needed for the application, the most obvious being the touchscreen seen in the next figure.

Figure 2. STM32F769i-DISCO front and back view [6]

For the user interaction, a four inches 800x472 liquid crystal display (LCD) is connected to the display serial interface (DSI) host controller of the MCU and a

FocalTech FT2x06 capacitive touch panel controller is connected to the I2C4 bus. For the audio part, Arduino™ connectors are used to support the expansion board housing the PSGs. Two 3.5 mm jacks are connected to the input and output of a Cirrus WM8994 audio codec, which shares the touch panel's I2C4 bus for the control signals and communicates with the MCU via a serial audio interface (SAI) for the digitized audio data. To store the framebuffer data, an external synchronous dynamic random-access memory (SDRAM) chip is present on board and connected to the FMC. Finally, for the external removable memory, a SD card connector is wired to the MCU's Secure Digital / MultiMediaCard (SDMMC) host to retrieve the audio files from external storage.

1.3 STM32 software ecosystem

STMicroelectronics provides a MCU package called STM32Cube with both a low-layer and a hardware abstraction layer (HAL) application programming interface (API), a board support package (BSP) and a set of middlewares, which interact together as shown in the figure below.

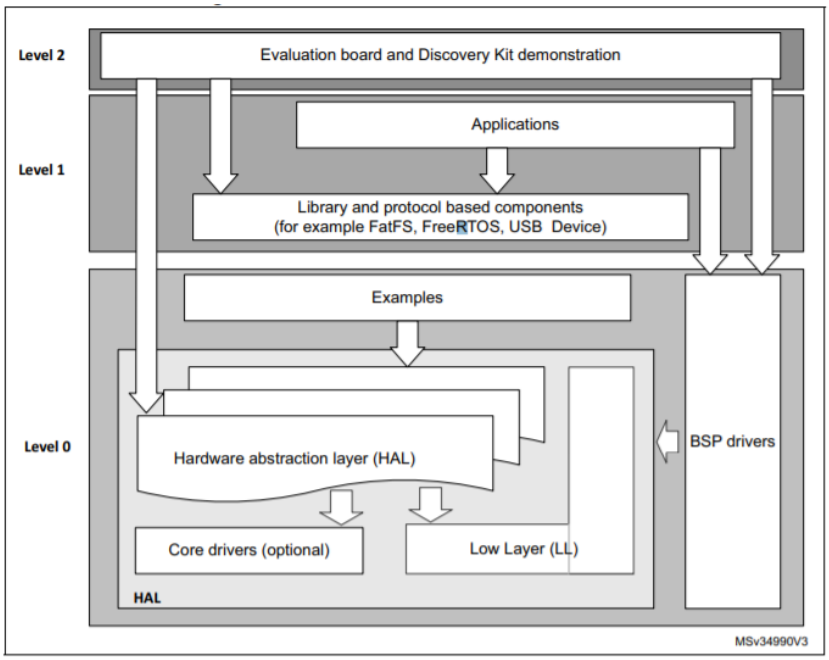


Figure 3. STM32CubeF7 firmware architecture [7, p. 9]

For this project, the package was trimmed down to the APIs and the Cortex Microcontroller Software Interface Standard (CMSIS) components used by the HAL drivers. The HAL is organized in modules, providing functions and data structures to simplify the configuration, initialization and usage of the various peripherals of the MCU. Using components instance handles, it is fully reentrant while able to track the status of a given peripheral and allow instance-specific treatment of events via user implemented callback functions. Throughout the project, the HAL was used to manage the communication peripherals, general input-output pins, interrupts, DMA transfer and error handling and recovery. As for the low-layer APIs, they were not used directly but were needed by the HAL, since the latter is built on top of them.

The BSP was left out of this project since some configurations were sub-optimal, the general structure was restrictive to a limited subset of the hardware features and internal cross-dependency forced an all-or-nothing approach, introducing bloat. The only exception was the OTM8009 LCD display start-up configuration as it could be used on its own and starting over would have introduced a lot of unnecessary work. Only a few modifications were done to be in line with the display datasheet timings and to improve brightness. Thus, drivers for the touch-screen controllers and the audio codec were remade, allowing the usage of many hardware functionalities not present in the BSP for the codec and improving gesture tracking for the touchscreen controller.

The included middlewares were also not taken from the STM32Cube package either because newer versions were available, or because the additional layers of abstraction for portability, for example in the case of CMSIS RTOS, were deemed unnecessary since portability was not a priority for this project and the wrapper functions added needless complexity. While it required additional modifications from the remaining ST’s libraries, cutting it down made the project more navigable and was seen as an advantage.

1.4 Middlewares

Two middlewares were added to the projects, FreeRTOS and ChaN's FatFs.

FreeRTOS is a real-time operating system designed for resources-constrained systems, as it is the case with a microcontroller. It provides a task scheduler, synchronization primitives and mechanisms for inter-task communication [8, pp. 2-3]. The FreeRTOS kernel allows the application to be split in different tasks, each with their own priority of execution, with the scheduler managing the task switching. From the application point of view, it allows to build more modular and clearer code, as well as guaranteeing a better response time to various external events [8, pp. 4-5].

Among the mechanisms for inter-task communication offered by FreeRTOS and used in this project are mutexes, binary semaphores and queues. Mutexes are used to guard shared resources, like an area of memory or a peripheral, and must be taken before access then released after usage. It allows a second task trying to access the shared resource to either wait and go to sleep until the resource is released or jump over the access and continue execution, if the situation allows. Thus, it prevents some race conditions from occurring and prevent errors caused by concurrent access to a peripheral. Binary semaphores operate in a similar way. A task can block itself completely or guard a specific block of code while waiting for a certain semaphore to be given either by another task or by an interrupt service routine [8, pp. 192-193]. They were used mainly to make task sleep while waiting for a DMA transfer to be completed or to process the music player routine every 20 milliseconds tick, using a timer. Lastly, queues are used to send data between tasks. Configured to hold a certain number of items of a specific size, they allow tasks to copy items on them, peek to see if the queue is empty or not and retrieve items [8, pp. 103-106]. In this case it is used, for example, to send the position data processed from the touchscreen to the main task and to send the control buttons events from the main task to the player task.

The second middleware used, FatFs is a filesystem layer module providing a file and directory access and management API for storage media formatted with FAT or exFAT. The media access interface was not part of the module and had to be implemented. Used to transfer music files from a card to internal memory, the read-only features were the only ones needed. Since the SD card interface is initialized on device start up, the disk_initialize and disk_status functions needed by fatFS were only set to return true. The same for the ioctl interface, as none of its commands were used through the application. The only function that had to be implemented is the disk_read function, which copy a set number of sectors from the SD card into a pre-allocated buffer.

2 Program overview

The program is separated in four different tasks running concurrently. Two major ones, the UI and the player tasks, and two others that serve a more supporting role, the touchscreen and the display tasks.

The UI task is responsible for most of the user interface processing. It dispatches the user touch input to the right GUI element and process its effect, update the state of the graphical user interface and draw the user interface elements in the framebuffer.

The second task, the player task, receives both commands and music data from the main task, when a control (play, pause, fast forward) is selected from the user interface. It is set to run every 20 milliseconds to parse one tick of the music file and update the state of the sound hardware. If auto-play is enabled, it also fetches the next music file and signals the main task to update the current file information.

The third task is the touch event task. It is woken up when the touch-screen controller triggers an interrupt, receive the touch information from the controller, format it and send it to the main task through a queue for processing.

Finally, the display task manages the display update and the swapping of the two frame buffers. It transfers the front buffer content to the display panel's graphic RAM then monitor the composition completion of the back buffer by the main task before exchanging them.

This subdivision allows to break down the complexity of the program flow in multiple units, guarantee a faster response time to the most crucial event and reduce the processor idle time by interleaving actions which have very little sequential dependency. As each of the tasks regularly enter an idle state, either waiting for an external event or for a DMA transfer to complete, CPU time could be kept under 20%

3 User interface implementation

3.1 Display

To create a frame to be sent to the display, the first step is to write each pixel value into a contiguous area of memory, the frame buffer. To reach an optimal refresh rate, both a front buffer, the one being displayed on screen, and a back buffer, the one being written to are needed. Once the front buffer content has been transmitted to the display and the back buffer has finished being composed by the UI engine the buffers are swapped and the operations are repeated. In this case, using 8 bits for each of the red, green, blue and alpha channel (32 bits per pixel) and an 800x480 pixels resolution, close to 3 megabytes are needed for the 2 buffers. This is exceeding by far the microcontroller RAM, so the buffers must be stored in external memory.

The SDRAM chip connected to the FMC was used for that purpose. Once the I/O pins were configured for external the address, data and control buses on the microcontroller, the FMC were configured for the properties of the SDRAM and the SDRAM has been initialized through commands sent via the FMC, the external memory could be accessed transparently as any internal memory or peripheral (in this case, occupying the address space starting at 0xC000 0000).

Then, to be able to render the content of the frame buffer on the LCD screen, components on the MCU and the display itself had to be configured as well. On the MCU side, the LTDC was used as a specialized DMA controller to transmit the front buffer data to the LCD panel own Graphics RAM (GRAM) over the DSI Host physical link.

The LTDC was configured with a single layer without operating any blending in this project as this task was covered during the composition of the back buffer. Since a 32-bit alpha, red, green and blue pixel format were used in the frame buffers to keep the pixels memory-aligned, one of its tasks was to convert it to the 24-bit red, green, blue format used in the display GRAM during the memory transfers. The DSI host for its part receives the data from the LTDC and transmit it to the LCD over the physical link. Configured in adapted command mode, it can refresh the display GRAM completely or in parts through memory write commands. Configuration, control and interrupt management of the host interface were done through a set of registers in the DSI wrapper.

The main issue when updating the display GRAM in a single operation was the presence of a diagonal tearing effect. Since the frame buffer is generated and updated on the panel GRAM in landscape mode while the panel LCD refreshes itself in portrait mode, a tear would appear diagonally where the two concurrent refreshes clash together, as pictured in figure 4 below.

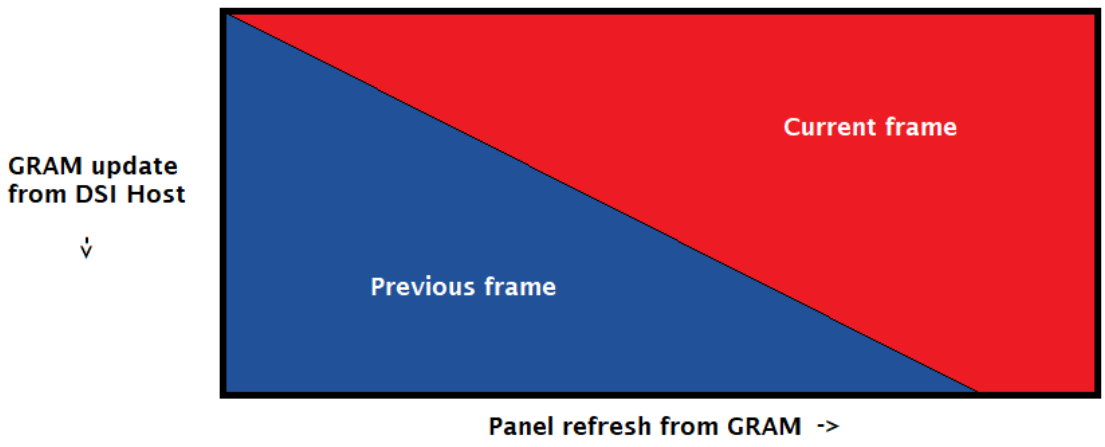


Figure 4. Tearing effect

Thus, the display task was conceived to serve two purposes: allow the transfer of the front buffer to the display while the back buffer is generated by the UI task and prevent the tearing effect. To remedy this, a method to refresh the screen in two parts was devised. Configuring the layer on the LTDC to retrieve a 400x480 window from an 800x480 image allowed to transfer only half the screen at a time without having to bother with the rows offset. An interrupt was also set on the display to trigger at the 400th line.

Once a frame is ready to be sent, the LTDC layer window is configured with the parameters for the left part of the screen and a column address set command (CASET) is sent to the LCD display to select that portion of GRAM to be updated. Then a semaphore is given, and the task go to sleep. When this semaphore is available to be taken, the transfer is initiated the next time the tearing effect interrupt routine is triggered. The task then resumes when the transfer is completed, and the right half is updated in the same way at the vertical blanking interval signal. The display task then waits until the back buffer is drawn, swaps both buffers and starts over. This process, its interactions with the interrupt service routines (ISR) and the display are illustrated in the figure on the next page.

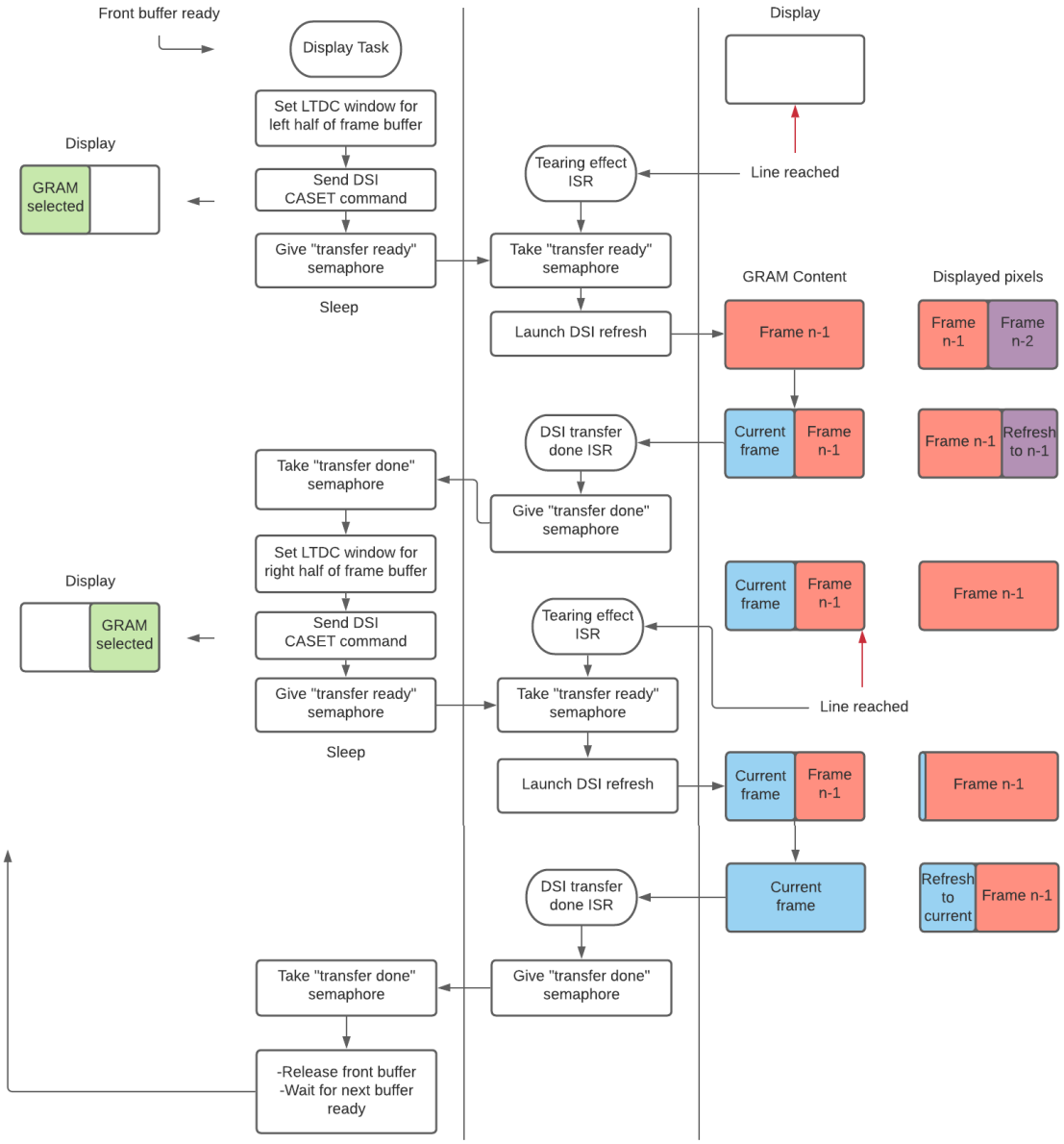


Figure 5. Flowchart of the display task.

Using this method the task itself spend most of the time sleeping as it only configures the peripherals and delegates the workload to the LTDC DMA. It also keeps the work in the interrupt service routine minimal, since with the configurations already done it only has to write a bit in the DSI wrapper to enable the LTDC to launch the transfer.

3.2 Touch Screen

The user touch input is received through what is believed to be a FocalTech FT6206 self-capacitive touch panel controller. That model number is mentioned in the BSP comments, but no detail is given in the board manufacturer documentation. Although the display board user manual mention support for two-points touches and gestures [9, p. 5], their respective registers as listed in FocalTech datasheet did not return valid data when probed, even after attempts to adjust the threshold and offset range values by writing on the appropriate controller's registers. As this issue was reproduced with the manufacturer code examples using the BSP, there has been no further attempts to solve the problem. It could be a different part number, or it may need to be enabled in the undocumented factory mode mentioned in the datasheet. Code in the BSP, for instance, invoke that factory mode to perform the touch panel auto-calibration, although that function did not have any perceivable effect.

Despite the unusable features, its mode of operation is straightforward. Once a touch is registered by the user, the touch panel controller triggers an interrupt to signal that new data is ready to be read. Through I2C, the MCU set the address of the first register to be read on the touch panel controller then initiate a read transfer. In this case, 5 registers contain relevant touch data, 0x03 to 0x06, as detailed in table 1.

Table 1. FT6206 touch data registers [10, p. 8]

Address	Name	Bit							
		7	6	5	4	3	2	1	0
0x03	P1_XH	Event Flag				Touch X position [11:8]			
0x04	P1_XL	Touch X position [7:0]							
0x05	P1_YH	Touch ID				Touch Y position [11:8]			
0x06	P1_YL	Touch Y position [7:0]							

In addition to the touch coordinate, an event flag which can have three possible values is provided. “Press down” when the user first makes contact with the screen, “lift up” when contact is lost, and “contact” when a movement is registered in between.

To respond to the touch-screen controller interrupt event and process the data, the touch-screen task was made. Figure 6 illustrates the process from the interrupt event until the data is written into the queue.

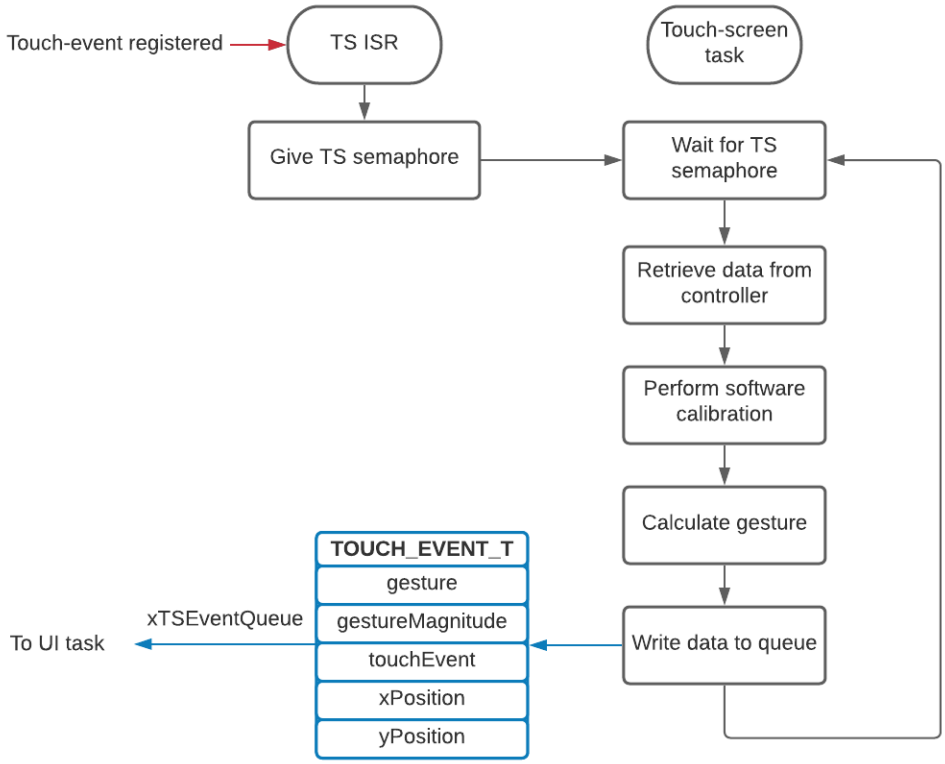


Figure 6. Touch-screen task.

After acquiring the input, since a non-linear drift was observed along the Y axis and the auto-calibration was not effective, the first operation was to apply software calibration. As the drift was only noticeable on the second half of the screen but was increasing as the values got larger, a few different methods and

values were tried and, in the end, the one in the next figure was retained.

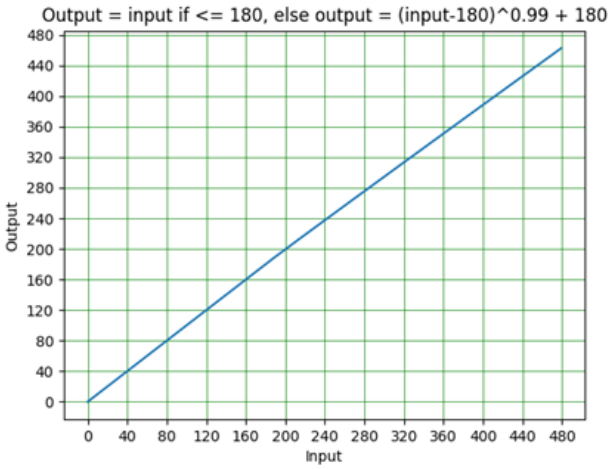


Figure 7. Y axis adjustment

Starting at the 180th line, it corrects the drift gradually up to a value around twenty at the maximum value, which is close to what was observed. While not perfect, this adjustment did improve the response on the lower end of the screen, lowering the odds of selecting the wrong item on the file browser.

Next, gesture data were absent from the controller itself and were needed for some of the GUI elements, so basic ones were added via software: swipe up, down, left, right. When a contact event is registered, it is compared to the previous recorded coordinates and if it has changed by a minimal threshold on one axis while staying under the drift limit on the other axis, a swipe gesture in the direction of the change is registered.

Once the data has been processed, a structure containing the gesture type, if any, the gesture magnitude, the event flag and both coordinates are copied into a queue, waiting to be read by the GUI engine and the task go back to sleep until another interrupt is triggered by the touch screen controller.

3.3 GUI engine

3.3.1 Overview

The GUI engine is responsible for presenting information and a set of controls to the user, direct the user touch input to the right control and modify the application behavior accordingly. In order to achieve this, an object-oriented approach was taken. A set of screen elements were designed using C++ classes, all with common functionalities but with different internal behaviors, configurable via parameters to offer customization and facilitate reuse.

In order to store these elements and allow the application to switch between different sets to be displayed, in the case of this project between the main music player window, the file browser and the volume control window, another set of classes has been created. Derived from an abstract class, “BaseScreen”, their purpose is to hold the code necessary to instantiate the diverse elements, push them in a vector, and redirect the touch-screen input data to the correct element. The layouts of these three windows are represented in figure 8.

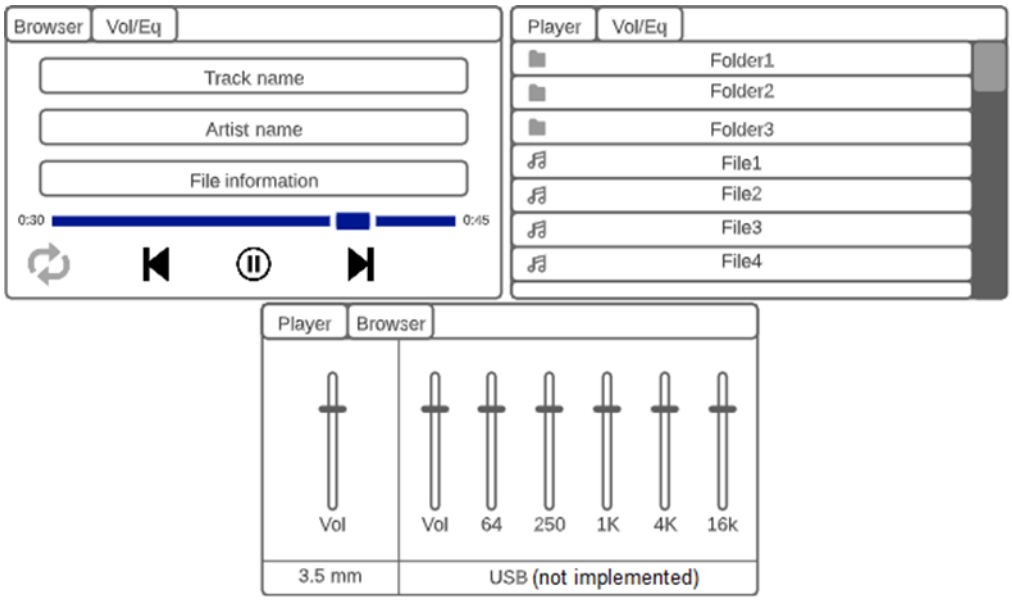


Figure 8. Layouts of the three windows used in the application.

As seen in the figure, switching between the different windows is done through buttons placed on the upper left part of the screen. On the player window are three text boxes with automatic horizontal scrolling, a responsive scroll bar to adjust the position within the current music track, timing display and a set of controls to enable auto-play, navigate to other tracks or pause. The browser window for its part has an indefinite number of buttons, generated through the directory content retrieved on the SD card and navigable through up-down gestures or through a vertical scrollbar to the right. Finally, the control window has a vertical slider to adjust the volume, and other controls were planned for further development.

Thus, to make the application these components had to be implemented:

- A generic button, able to be represented by an image and/or text.
- A horizontal scrolling text area, able to print a moving section of a string.
- A track slider, updated through the current track elapsed time and able to update it in return.
- List elements for the file browser, able to print themselves in part or in full to be cropped as they are scrolled in and out from the bottom and top parts, as well as support manual scrolling in response to swipe right and swipe left in the case their text string does not fit in their given area.
- A vertical scroll bar to set the index of the file browser.
- A vertical slider, for controls such as volume.
- Static text area, which has no effect when touched but is displayed at its coordinates.

To avoid having to create a new derived class for every different response to the user touch input, multipurpose elements like sliders and buttons were designed so they could be assigned function pointers which are called in response to an event. More details on the implementation will follow in the next sub-sections.

3.3.2 Screen element classes

The common functionalities are defined in an abstract class, ScreenElement, from which all the components forming the GUI are derived. In that manner, the different elements can be stored in a common container and accessed by the client code irrespectively of their implementation. Furthermore, it allows the derived classes to define their own behaviours, constructors and optionally additional methods. Each of these elements store their position and size in private variables, and the following six methods must be defined for every class:

- draw
- isInside
- press
- contact
- liftoff

The draw method is invoked for each element of the current screen on every frame. It is then the responsibility of the element to draw itself on the frame buffer using a specific image, text string and/or font assigned to it. The isInside method for its part is invoked on each element of the current screen every time a touch event is registered. A true or false value is returned depending on if the touch event occurred in the object area. If true, the program can then execute the press, contact or liftoff method of that object based on the touch event flag.

A font class was made independently so the elements containing text can be assigned different fonts. It contains the font properties, and a map paring characters with their graphical data location. Elements containing text can then draw strings on the screen by iterating through each character, retrieving its location from the font then calculating the framebuffer offsets for the DMA2D transfer.

The diagram on figure 9 shows the various classes created for the UI and the relationship between them.

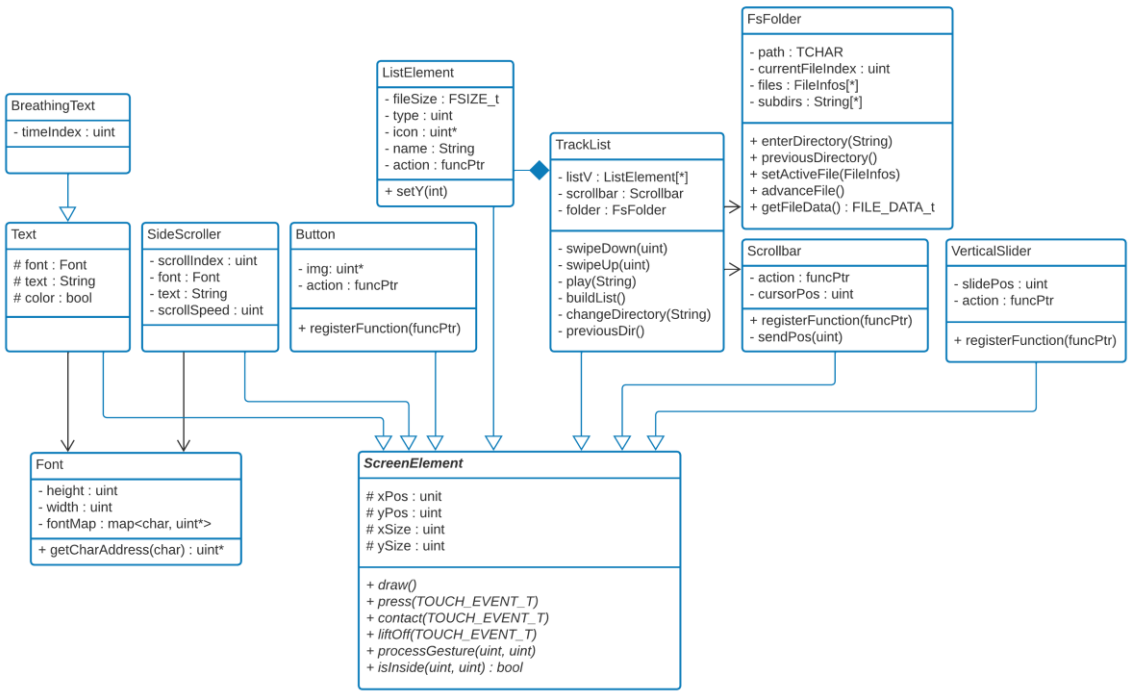


Figure 9. UI elements class diagram.

3.3.3 Screen classes

The screens were not created in a generic manner like the screen elements; thus, a new class is derived from the BaseScreen abstract class for every window used in the program. It allowed, though, to confine the code for their elements creation in their constructor and define containers and methods distinct to a single window. The abstract class enforce only the definition of two methods: processTouch and drawScreen. The first one finds the element associated to the coordinate and pass it the touchEvent data, while the drawScreen calls the draw method of every visible element.

Finally, the code controlling the GUI is running in its own FreeRTOS task (MainTask), the main loop being in the MainEngine class which store a pointer to the currently selected screen. For each frame that is be generated, the program logic is as follows: first retrieve the structure as described in 4.2 from the touch screen queue, if any, then pass it to the processTouch method of the current screen.

The current screen's processTouch method then go through every element in its vector, calling the isInside method with the given coordinates until an element returns true, signaling the event happened within its perimeters. That element's action is then processed, and execution goes back to the main loop. Then, once the back buffer is available the background is drawn on it, and finally the draw method of every element is executed.

The layout of the classes involved in this process is illustrated in the figure below.

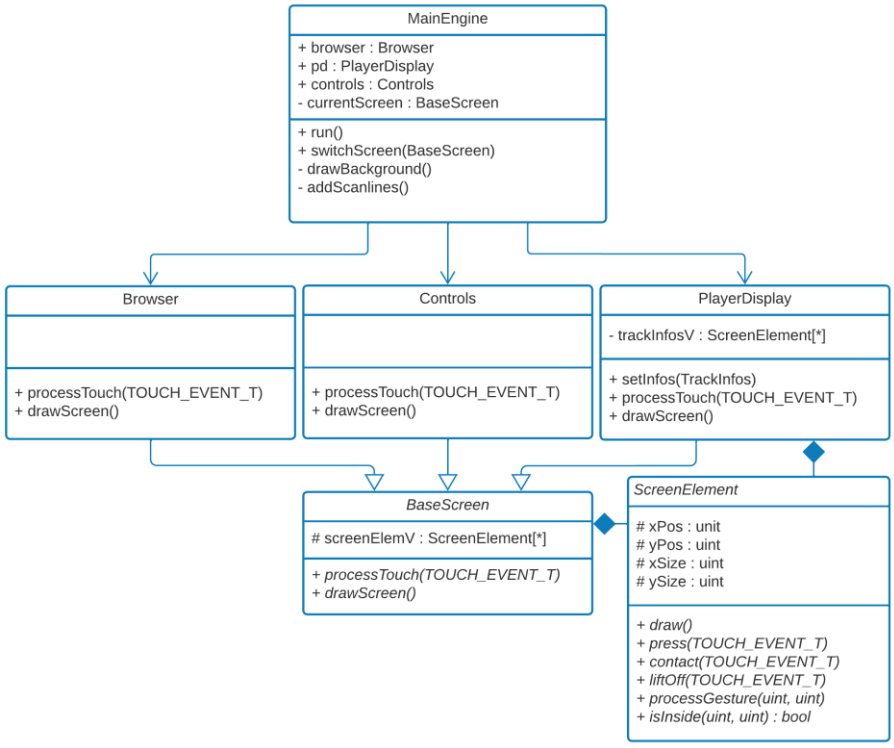


Figure 10. Screen class diagram.

4 Sound generation hardware

4.1 Programmable sound generators

The audio is generated using two simple integrated circuits, variants of the AY-3-8910 from General Instrument. An AY8930 was used in the first slot as it is an enhanced version with more bits of precision and could be used for further experiments, while having a backward-compatibility mode. For the second slot an AY-3-8913 was selected for its form factor, lacking the superfluous I/O port present on other chip of the family thus reducing the pin count from 40 to 24.

Capable of producing square waves and noise of variable frequency and amplitude, these devices require no outside intervention beyond refreshing their register array, pictured in the following figure.

Register \ Bit		B7	B6	B5	B4	B3	B2	B1	B0
R0	Frequency of channel A	8 bit fine tone adjustment							
R1						4 bit rough tone adjustment			
R2	Frequency of channel B	8 bit fine tone adjustment							
R3						4 bit rough tone adjustment			
R4	Frequency of channel C	8 bit fine tone adjustment							
R5						4 bit rough tone adjustment			
R6	Frequency of noise				5 bit noise frequency				
R7	I/O port and mixer Settings	I/O		Noise			Tone		
		IOB	IOA	C	B	A	C	B	A
R8	Level of channel A				M	L3	L2	L1	L0
R9	Level of channel B				M	L3	L2	L1	L0
RA	Level of channel C				M	L3	L2	L1	L0
RB	Frequency of envelope	8 bit fine adjustment							
RC		8 bit rough adjustment							
RD	Shape of envelope					CONT	ATT	ALT	HOLD
RE	Data of I/O port A	8 bit data							
RF	Data of I/O port B	8 bit data							

Figure 11. AY-3-891x family register array [11, p. 5]

Each of the three channels can produce simultaneously a square wave and pseudo-random noise by dividing the master clock with their frequency register (R0 to R6). The output is then mixed or muted according to the mixer register (R7) bits. Afterward the amplitude is set either to a fixed level following the L3-

L0 value of R8-RA if their mode bit (M) is clear, otherwise following the variable envelope shape defined in RD. The resulting value then go through a digital to analog converter and the signal is available on an open-emitter output pin.

The interface to update the values of these registers consist of an 8-bit data bus, two address input pin and a three-pin control bus. As these signals were designed to maximize compatibilities with existing microprocessors busses, only one of the address input pins was kept for the chip select signal and the other was tied to 5 volts using a pull-up resistor. Same goes for the control bus as two of them were enough for the three control signals: bus inactive, write to PSG and latch address.

Lacking a proper address bus, the target register address is sent through the data bus with the latch address control signal, effectively requiring two write for every register update: first to write the register address, second to write the register data. Thus, the register update operations below were created following the datasheet timing diagrams in figure 12.

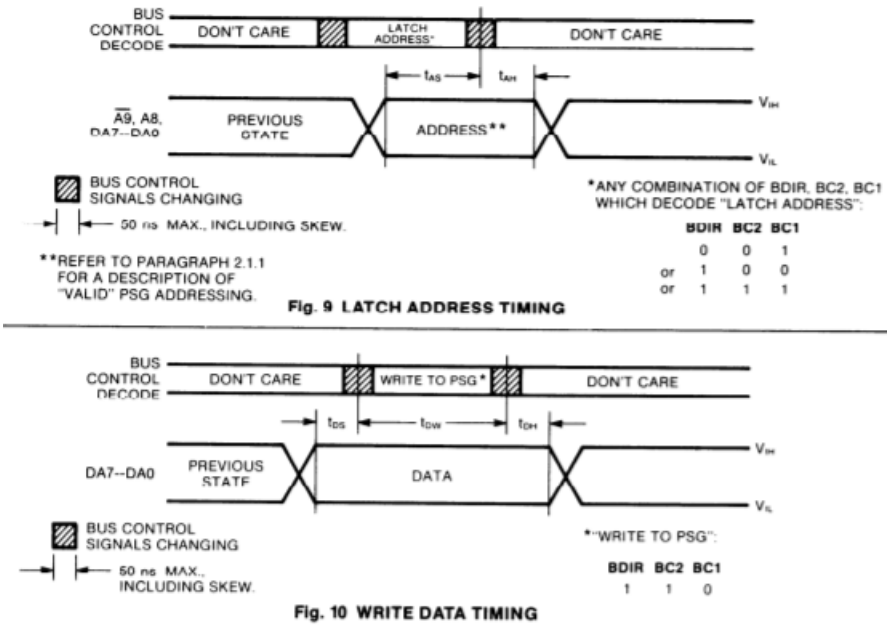


Figure 12. AY-3-891x timing diagrams [11, pp. 10-11]

BC2 being pulled-up to 5 volts, the required operations are then:

- Enable the chip by setting the address line to 5 volts.
- Output the address on the data bus.
- Set BDIR and BC1 to 1, sending the latch address signal.
- Set BDIR and BC1 to 0 to set the bus back to inactive, effectively selecting the register address present on the data bus at that time.
- Write the register data on the data bus.
- Set BDIR to 1 and BC1 to 0 to initiate write to PSG, then set the control bus back to inactive.

Thus, 14 pins in total were required to drive both PSGs. Eight for the 8-bit data bus, two for the control bus (BDIR and BC1), one other for each of the ICs chip select, one for the chip reset signal and finally one for the master clock. To simplify the wiring and free the CPU during the registers update operation while maintaining proper timing an I2C I/O expander was used to drive the chips via a serial protocol, thus reducing the pin count to 4, excluding power and ground.

A Microchip MCP23017 16-bit I/O expander was elected for this usage. Having two 8-bit ports, the data bus could be put on its own port while the control bus and both chip-select be wired on the other one. A special mode [12, p.13] could also be enabled to write to the I/O expander while alternating between its own register pair, in this case between the port A and B data. Conveniently, it allowed the construction of a large array containing the 14 registers addresses and data interleaved with the control bus signals which, sent to the I/O expander in one large DMA transfer, could update the state of a sound chip in a single transaction. This way, the CPU can work on other tasks once the transfer is initiated and the sound chip setup, hold and access specified in the datasheet could be matched by tuning the MCU I2C controller clock speed.

An intermittent glitch was detected and investigated with the second PSG, the AY-3-8913, which forced later to rethink the update mechanism for this one chip. At irregular interval a clicking sound could be heard, while the problem did not manifest with any other compatible PSG. To rule out any software cause, a sweep signal was programmed on both PSGs at the same time, putting their

chip select on the same line. The output of each PSG channel A was captured independently, as well as BDIR and BC1 in attempt to correlate the issue with any bus transaction. The results are shown in the next figure.

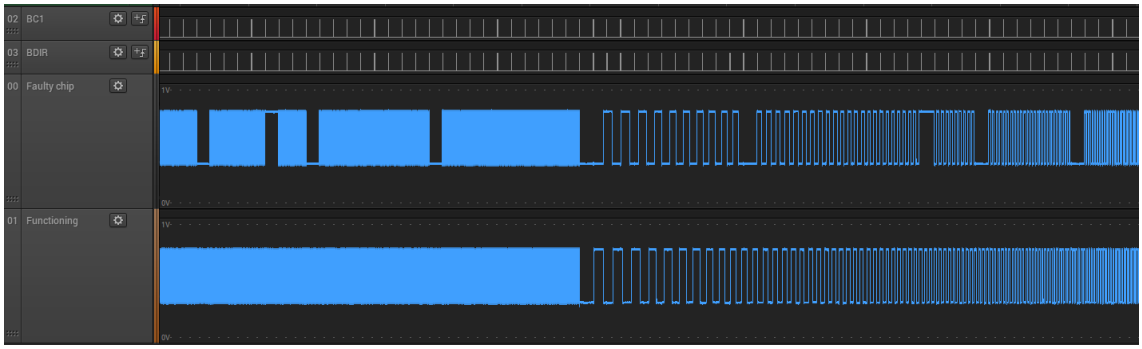


Figure 13. Sweep signal captured on AY-3-8913 (top) and AY-8930 (bottom)

Interruptions in the square wave first hinted a faulty chip, but ten others AY-3-8913 with a total of four different date codes and coming from at least two different factories were later tested with similar results. On some captures a correlation between the write interval (20 milliseconds) and the length of the interruption was observed, as shown on the next figure.

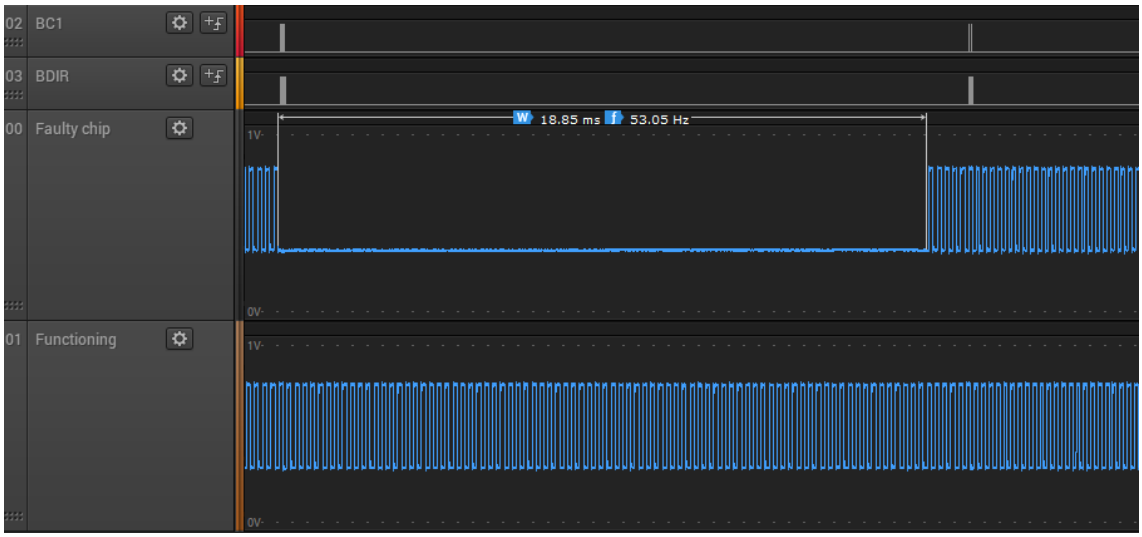


Figure 14. AY-3-8913 interruption duration

But changing the chip clock, thus the output frequency, showed that the interruption length was proportional with the clock frequency. It was, in all case,

correlated with a period value change. The PSG internal counter was then investigated and while the user manual specify it is using a down counter [13, p.18], the following findings tells otherwise.

First, it is specified that the frequency of the tone generator is obtained by counting down the master clock by 16, then by counting down the 12-bit value of the tone period register pair. Thus, the tone frequency can be calculated with the following equation.

$$fT = \frac{fClock}{16TP}$$

It would mean, with a 1 MHz clock as used on this test bench, that the lowest possible frequency, with a tone period of 0xFFF, would be 15.26 Hz, or 65.53 milliseconds per cycle.

$$\frac{1000000MHz}{16 * 0xFFF} = 15.26Hz$$

$$\frac{1cycle}{15.25\,cycle/s * 1000\,ms/s} = 65.53ms$$

That would be 32.76 milliseconds between each transition (half-period) for a complete run of the tone counter. Reproducing the glitch by reducing the half-period from 6 milliseconds to 4 milliseconds while the counter was in between allowed the observation presented in figure 13.

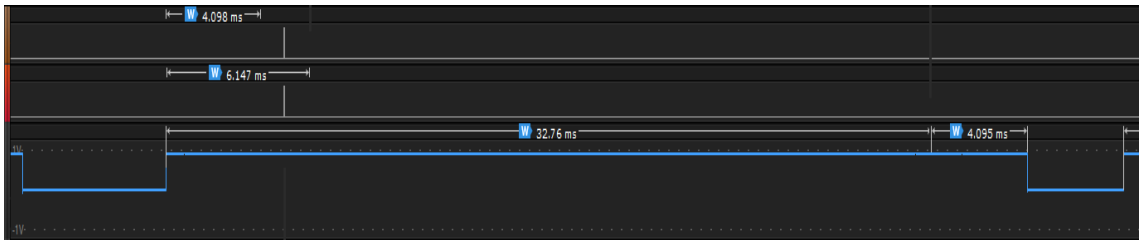


Figure 15. Tone counter

The pulses on the two top rows represent the moment the tone half-period value changed from 6.1 to 4 milliseconds. As the counter is manifestly counting up, it does not reach zero at the end of the 6.1 milliseconds period nor reset with the new value but rather keeps counting up, overflows at 0xFFF and counts back up to the new value instead. Two conclusions could be reached then: an up counter is comparing with the value of the tone period registers directly and a design flaw is making an « equal » instead of a «greater than or equal » comparison between the tone counter and the tone period registers value. From the date codes on the tested ICs, unless they have been remarked, it is suspected they were produced for a decade without addressing the issue and it might have gone undetected until now, as no report of this flaw could be found. Their only reported commercial usages for this model were arcade machines and a third-party sound card for the Apple II, the Mockingboard. Used for rudimentary sound effects and simple melodies it is plausible that without being pushed to their limits with rapid changes this clicking sound went undetected or was much less bothersome.

To work around this issue a solution was devised using the microcontroller. Three timers, one for each channel, were set up with a frequency of the PSGs clock divided by 16 to synchronize them with the tone counters. Instead of updating the registers in the same DMA transfer as the other ones, when an update is necessary an interrupt on the associated timer is enabled. Inside the interrupt routine the transfer is initiated to update the associated tone period register pair while the tone counter is close to zero, reducing the possibilities of overflow when switching to a higher frequency. Then the timer auto-reload value is updated to match the new tone period, keeping the timers in near synchronization with their respective sound chip’s channels.

4.2 Audio expansion board

An audio expansion board was manufactured to control the PSGs and capture the audio output signal from them. The schematic and printed circuit board (PCB) layout design was done using the KiCad software, resulting in what is seen in figure 16.

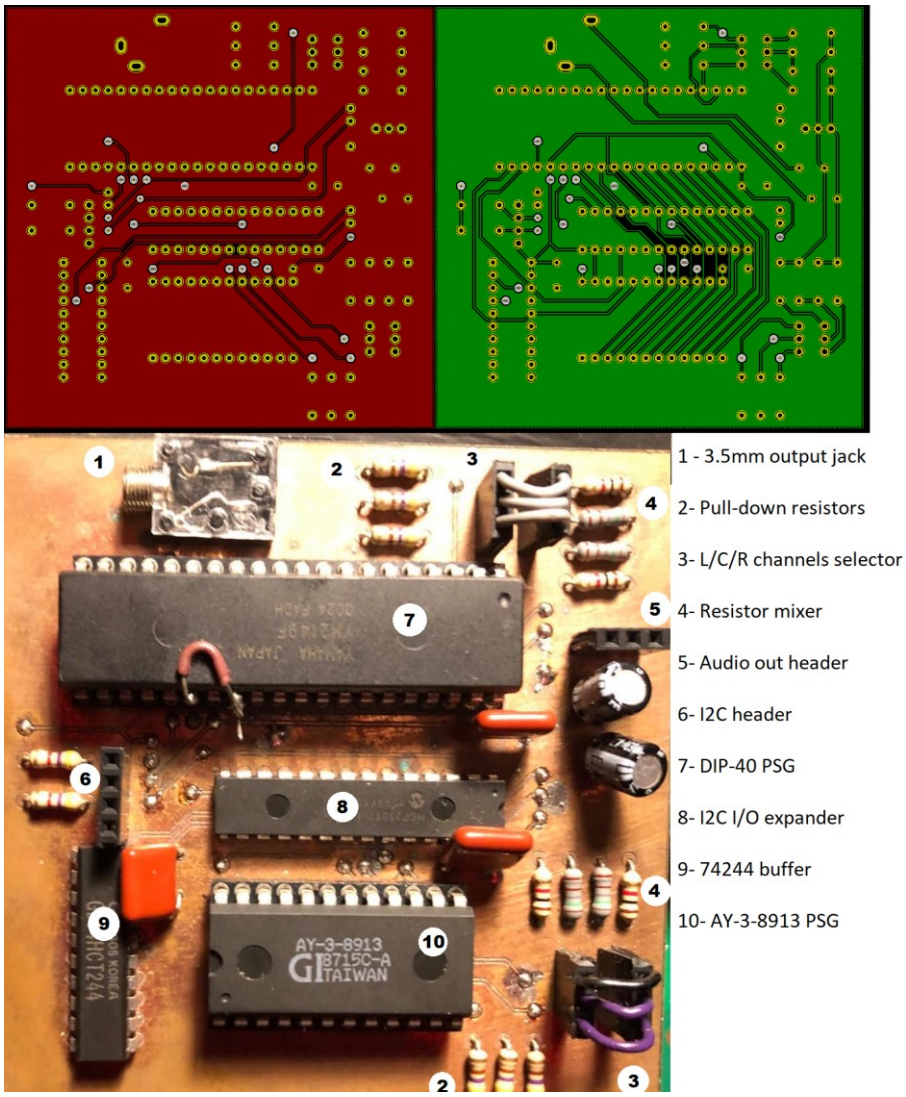


Figure 16. Back and front view of the PCB design (top) and populated board (bottom).

One thing that had to be taken in consideration here is the logic level shifting between the 3.3 volts I/O of the development board and the 5 volts logic of the driven sound generators. According to the specifications of the latter they should have been able to be driven with 3.3 volts on the input pins, requiring 2.2 volts to register a high level [11, p.9] while the I/O expander specify 2.6 volts (supply voltage – 0.7 volt) minimum on the high-level output [12, p.4]. But in practice it has shown otherwise, and malfunctions were observed. Therefore, logic level shifting was operated in two ways: with a 74HCT244 buffer for the sound generator clock and reset signals, and on the I/O expander itself.

The I2C bus was pulled-up to 3.3 volts with two resistors on the board and a header exposing the serial data line, serial clock line, ground and 3.3 volts power was installed for debugging purpose and allowing further expansions. While the I2C lines were operated at 3.3 volts the I/O expander was powered at 5 volts. Since I2C works with open-drain configuration and the devices are pulling the line low, not high, there is no risk of damaging the microcontroller by interfacing it with a 5 volts device. And although this configuration is completely out of the specifications as the minimum input high-level voltage on the I2C lines for the I/O expanders is written as supply voltage x 0.8, thus requiring 4 volts while 3.3 is given, extensive testing produced no faults and proved to be much more reliable than following the datasheets.

For the sound generators audio output, a simple resistors circuit was made. First, each channel is pulled down to ground as they are open-emitter output. Second, a pair of headers are used as jumpers to select which of the A, B and C channels are assigned to the right, middle and left channel. Finally, both left and right output goes to their respective channels through 1 kilo-ohms resistors while the middle is split between the two with 1.6 kilo-ohms resistors. With this configuration the signal peaks at 1 volt at maximum volume. The audio output is then available at two points on the board. The first point is through a header exposing left, right and ground for debugging purposes. Then for the proper audio output the two channels first pass through a capacitor to remove any direct-current bias and the signal is then routed to a stereo 3.5-millimetre jack.

5 Audio Codec configuration

Analog to digital conversion, audio processing and digital to analog conversion was done on the audio codec present on the development board, the WM8994 by Cirrus Logic. This way, both volume control and noise reduction features could be added to the project and the codec's headphone output driver could be

used instead of the line level output of the sound generators, allowing to interface with a wider variety of headphones and passive speakers.

The codec has two interfaces with the microcontroller. The configuration registers are modified via I2C, and an audio interface is present to control the sample rate and transfer data to and from the microcontroller. A class was therefore written to initialize the codec's audio path registers, set up the audio interface on both the codec and the microcontroller peripheral and provide a method to tune the volume of the headphone driver.

The signal first goes through the analog input path, as pictured in figure 14.

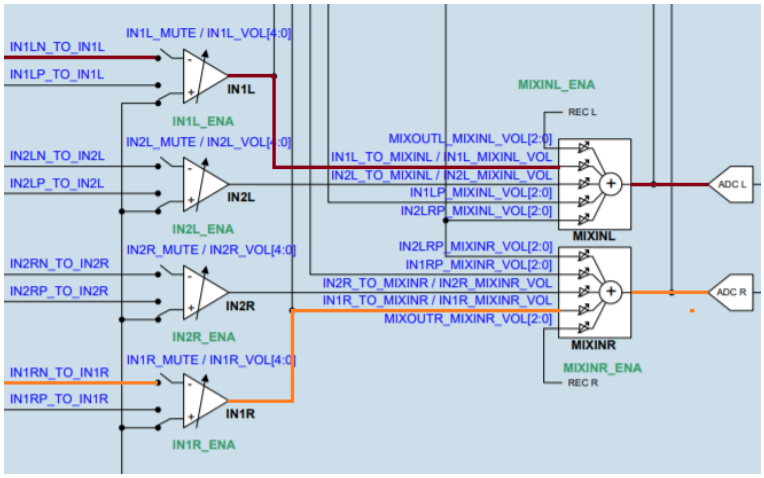


Figure 17. WM8994 Analog input path. [14, p. 44]

On both left and right channel, the signal is routed through a programmable gain amplifier (PGA) and a mixer before reaching the analog-to-digital converter. The ADC range being 0 to 1.8 volts [6, p.42], a gain of 3 decibel was applied on the 0 to 1 volt input signal through the PGA. The amplified signals of 0 to 1.43 volts are then sampled at 44.1 kilohertz on the 16-bit ADCs. Since noise was introduced in the system through the IC2 bus shared with the touch-screen controller (touch input could be heard through the codec output), a gain that would make use of the whole ADC dynamic range was counter-productive. These settings along with the noise reduction operated in the following part were seen as the best compromise.

The digital samples then had to be sent out to the audio interface so they could go through the dynamic range control block. Compression is operated on the digital signal to get rid of the noise introduced by the touchscreen as well as negate any humming that could be heard when the player was idle. A hard knee was configured in the compressor to reduce any low-level noise (-52.5 decibel relative to full scale) furthermore. The resulting samples then are transmitted to the microcontroller through the serial audio interface and are looped back as well to the codec's audio interface input. On the codec side, they are converted back to analog signals again, are attenuated or amplified based on the volume slider set inside the UI and are finally available on the headphone jack of the development board, following the path illustrated in figure 18.

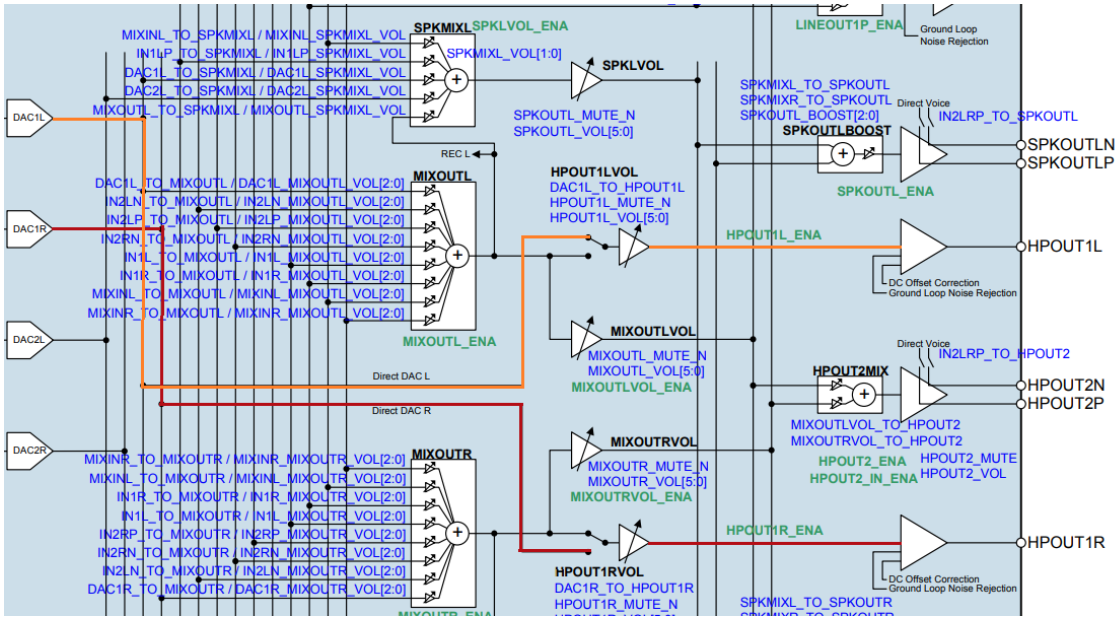


Figure 18. WM8994 analog output path. [15, p.112]

6 PT3 player

To gain access to an extensive library of ready-made tracks to be played on the device an interpreter for the PT3 tracker format was built. Originally made for the ZX Spectrum home computer which used the same sound generator but not dependent on any microprocessor architecture made it a good candidate. Through different reusable blocks of data which can act as base values or

modifiers for the sound generator register contents, relatively complex music can be made while keeping file sizes minimal. These building blocks are specifically patterns, samples and ornaments.

The main building blocks are the patterns, which specify the base tone, noise and envelope value at certain points in time. Some special commands such as slides, porta-mento and tremolo can also be added. The following figure containing a screen capture of Vortex Tracker, a PT3 editor for Windows illustrate this.

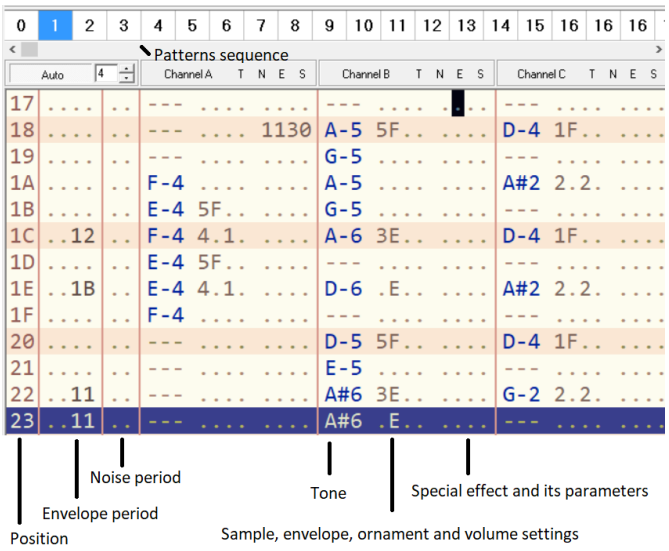


Figure 19. Vortex Tracker pattern window.

The samples, for their part, operates on finer time ticks and will give an effect akin to an instrument timbre, adding small shifts in the tone value, enabling the tone, envelope and noise of the channel as well as adding a volume multiplier. Figure 20 illustrates such a sample data.

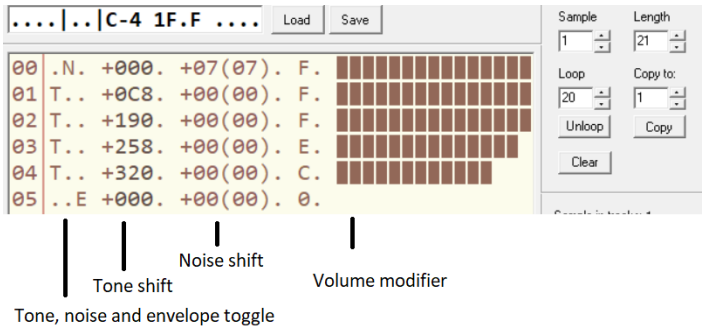


Figure 20. Vortex Tracker sample window.

The third building block is the ornament, which are half-tones modifiers. They are used, for example, to simulate bends or perform fast arpeggios sequences.

The organization of the data in these files was deciphered with the help of a file format description automatically translated from Russian [15]. It allowed to find the location of the file's properties, offsets of the patterns, ornaments and samples data as well as their general organization. Behavior on some edge cases, such as overflows when adding modifiers to a base value were not in the specifications so in these cases trial and error was used until the result was faithful to the original player behavior. Figure 21 on the next page gives an overview of the data organization.

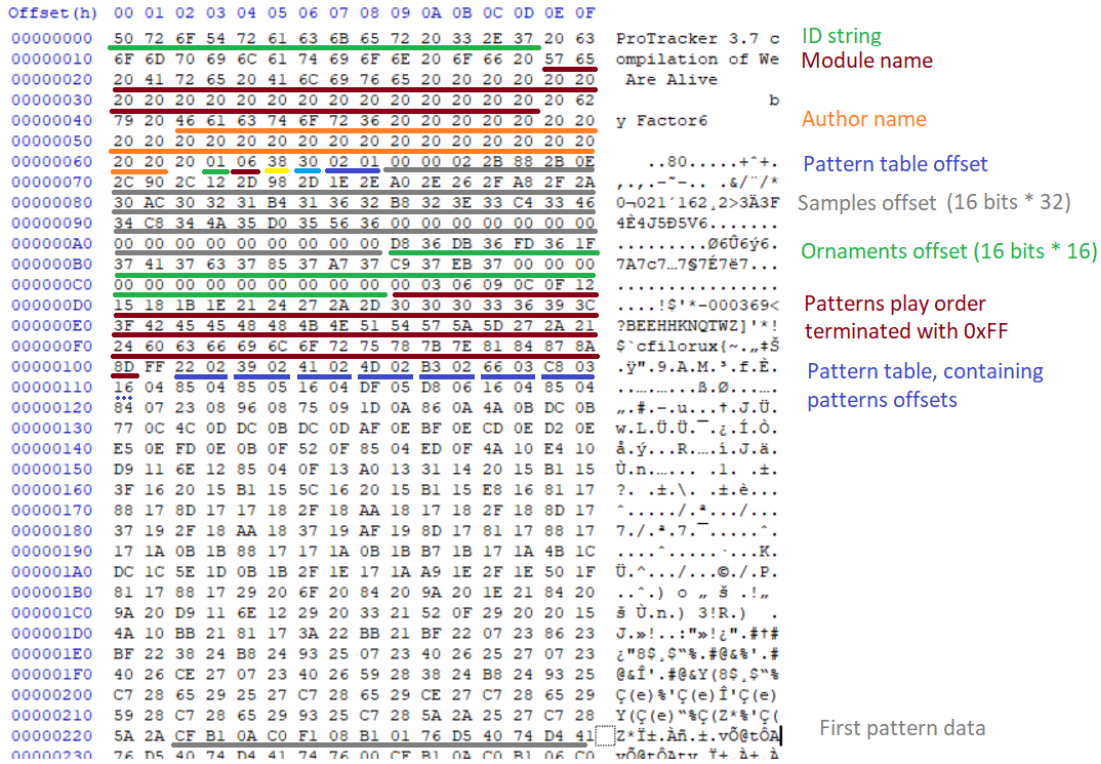


Figure 21. Annotated hex-editor view of a PT3 file.

To navigate the file in memory while playing, an approach using pointers to the patterns, samples and ornaments tables with the addition of a current position offset was taken. The complexity was further broken down using an object-oriented approach, with objects for each of the three channels of the sound

generator and their current ornament and sample. Thus, their respective operations on the data and their current internal states could be kept independent. Their attributes and methods can be seen on the figure below.

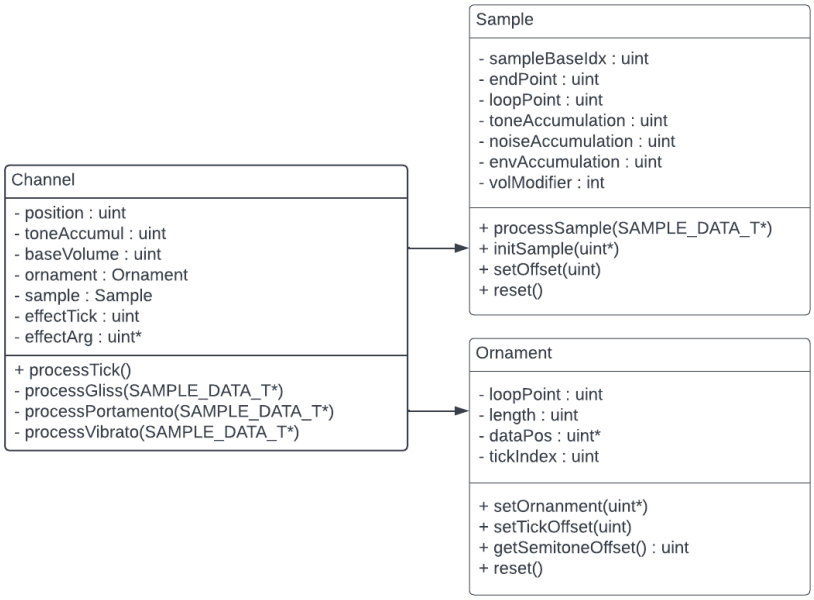


Figure 22. Diagram of the Channel, Sample and Ornament classes.

7 Conclusion

This project has demonstrated that a microcontroller-based device could handle the processing of multiple complex tasks at once, given the right support peripherals and an appropriate usage of direct memory access. While successful and operational in the end, unforeseen problems and the necessity to add additional features to work around them greatly increased the complexity of both the project and the software source code itself. As many different software and hardware components were meant to work together, the biggest challenge was to keep their respective software component separated as much as possible while still being able to interface with one another. Both FreeRTOS and the object-oriented approach of C++ helped to reach this end, but inevitably some entangling came along the way.

While developing a UI engine from the beginning has its charm as a novelty, the amount of work needed to reach a refined application programming interface akin to the available commercial frameworks makes in unviable for a production environment. Although the UI developed for this project was kept simple to use, the methods to send commands on inputs to other parts of the program was poorly thought of and was responsible for most of the convolutions.

In the end, the outcome was a usable product with possibility for extensions but the most important part was the experience gained from studying microcontrollers internals, performing electronics debugging, planning an object-oriented, medium-scale software solution to designing a printed circuit board.

References

1 Arm Limited. Arm Cortex-M7 Processor Technical Reference Manual Rev F; 2018.
URL: https://developer.arm.com/documentation/ddi0489/

2 Parris, Neil. Extended System Coherency: Part 1 – Cache Coherency Fundamentals. Arm Limited; 2013.
URL: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/extended-system-coherency---part-1---cache-coherency-fundamentals

3 STMicroelectronics. DS11532: STM32F765xx STM32F767xx STM32F768Ax STM32F769xx datasheet Rev 7; 2021.
URL: https://www.st.com/resource/en/datasheet/stm32f769ni.pdf

4 STMicroelectronics. AN4667: STm32F7 Series system architecture and performance, Rev 4; 2017.
URL: https://www.st.com/resource/en/application_note/dm00169764-stm32f7-series-system-architecture-and-performance-stmicroelectronics.pdf

5 STMicroelectronics. RM0410: STM32F76xxx and STM32F77xxx advanced Arm®-based 32-bit MCUs reference manual, Rev 4; 2018.
URL: https://www.st.com/resource/en/reference_manual/dm00224583-stm32f76xxx-and-stm32f77xxx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf

6 STMicroelectronics. UM2033: Discovery kit with STM32f769NI MCU user manual, Rev 3; 2018.
URL: https://www.st.com/resource/en/user_manual/dm00276557-discovery-kit-with-stm32f769ni-mcu-stmicroelectronics.pdf

7 STMicroelectronics. UM1891: Getting started with STM32CubeF7 MCU Package for STM32F7 Series, Rev 9; 2019.
URL: https://www.st.com/resource/en/user_manual/dm00180213-getting-started-with-stm32cubef7-mcu-package-for-stm32f7-series-stmicroelectronics.pdf

8 Barry, Richard. Mastering the FreeRTOS™ Real Time Kernel: A Hands-On Tutorial Guide. Real Time Engineers Ltd; 2016.
URL: https://www.freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

9 STMicroelectronics. UM2104: 4-inch WVGA TFT LCD board with MIPI® DSI interface and capacitive touch screen user manual, Rev 1; 2016.
URL: https://www.st.com/resource/en/user_manual/dm00321394--4inch-wvga-tft-lcd-board-with-mipi-dsi-interface-and-capacitive-touch-screen-stmicroelectronics.pdf

10 FocalTech. FT6x06 Self-Capacitive Touch Panel Controller datasheet, Ver 0.4; 2013.
URL: https://www.displayfuture.com/Display/datasheet/controller/FT6206.pdf

11 Yamaha. YM2149 Software-Controlled Sound Generator (SSG) datasheet; 1987.
URL: http://www.ym2149.com/ym2149.pdf

12 Microchip Technology Inc. MCP23017/MCP23S17: 16-Bit I/O Expander with Serial Interface datasheet; 2016.
URL: https://ww1.microchip.com/downloads/en/devicedoc/20001952c.pdf

13 General Instrument. AY-3-8910/8912 Programmable Sound Generator Data Manual; 1979.
URL: http://dunfield.classiccmp.org/r/ay38910.pdf

14 Cirrus Logic. WM8994: Multi-channel Audio Hub CODEC for Smartphones datasheet, Rev. 4.6; 2018.
URL: https://statics.cirrus.com/pubs/proDatasheet/WM8994_Rev4.6.pdf

15 ACNews. Pro Tracker v3.x Module Format, Russian; 2002.
URL: http://zxpress.ru/article.php?id=9313