Bachelor’s Thesis

Information and Communications Technology

2023

Niklas Salminen

Audio synthesis for resource-constrained microcontrollers

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2023 | 62 pages

Niklas Salminen

Audio synthesis for resource-constrained microcontrollers

In the vast world of electronics, the applicability of microcontrollers has been increasing in the last decades. The demand for audio applications such as audio signal processing and synthesis is genuine, but solutions lack thereof. The appeal of audio interfaces is that they provide user experiences that translate into emotional feedback from humans.

The purpose of this thesis was to research and assess software solutions for an existing microcontroller on the market. The thesis analyzes existing software solutions for audio signal processing deemed specifically for audio synthesizer aspirations. The hardware for these solutions was chosen with prerequisite knowledge in the context of digital audio to minimize the risk of failure during the thesis project.

The operating system, peripherals, and drivers were adopted from the beginning, whereas software libraries were explored during the thesis project and would be selected as part of the final solution.

A feasible solution has been used for demonstration purposes. The scope of the thesis was to synthesize a single monophonic note using the target hardware and software with the ported real-time audio software library. The result does not accredit a fully-fledged result that would be considered of production quality.

Keywords: software, constraints, memory, synthesizer, audio

Contents

Equations

Figures

List of abbreviations

1 Introduction **8**

2 Research **10**

2.1 Software licensing 10

2.2 Maintenance and stability 11

2.3 Target hardware and software 12

2.4 Analysis and solutions 13

2.5 Digital signal processing 13

2.5.1 Discrete Fourier Transform 14

2.5.2 Digital filtering 15

2.5.3 Digital audio signal processing 16

2.6 Audio synthesizer 17

2.7 SoundFont 20

2.8 Software porting analysis 24

3 Assessment **28**

3.1 Risk assessment 28

3.2 Memory constraints 28

3.2.1 Dynamic memory 30

3.2.2 Call stack analysis 32

3.3 Performance bottleneck 34

4 Implementation **36**

4.1 Semantic versioning 36

4.2 Dependencies 38

4.3 Configuration 39

4.4 Static analysis 40

4.5 Audio driver 41

4.6 Audio application 47

4.7 Compilation 47

5 Conclusion 55

References 59

Equations

Equation 1. The Discrete Fourier Transform Formula. 15

Figures

Figure 1. Polyphonic and monophobic functions in Fluidsynth. 19
Figure 2. General RIFF file structure. 21
Figure 3. POSIX file operation callbacks for SoundFont. 22
Figure 4. SoundFont inspection with Polyphone software. 24
Figure 5. Second stage of compilation. 26
Figure 6. Third stage of linking an intermediate form binary. 27
Figure 7. Final build stage. 28
Figure 8. Program memory section sizes from FluidSynth. 30
Figure 9. Program memory region sizes from FluidSynth. 31
Figure 10. Program memory layout. 32
Figure 11. Symbols and memory sizes in FluidSynth. 33
Figure 12. Dynamic memory allocation in FluidSynth. 34
Figure 13. Semantic versioning scheme. 38
Figure 14. Available audio drivers and interfaces. 44
Figure 15. License agreement of the audio driver. 45
Figure 16. Type, API and driver dependencies. 46
Figure 17. Preprocessor symbols of driver instances. 46
Figure 18. Audio driver type API constraints. 47
Figure 19. Linker generated cross referenced map file. 51
Figure 20. First build memory statistics. 53
Figure 21. Deconstructed symbols from the ELF file. 53
Figure 22. Audio dithering array. 53
Figure 23. The audio dithering table in source code. 54
Figure 24. Second build memory statistics. 55
Figure 25. Project time management. 57

List of abbreviations

Abbreviation	Meaning
ABI	Application Binary Interface
ALSA	Advanced Linux Sound Architecture
AOSP	Android Open Source Project
API	Application Programming Interface
Arm© Cortex©-M33	32-bit RISC ARM processor core
ARMv8	Architecture used by Arm© Cortex©-M33 processor
C	C Programming Language
C11	C Programming Language Standard roughly adopted in 2011
C90	C Programming Language Standard roughly adopted in 1990
CPU	Central Processing Unit
DAC	Digital-to-Analog Converter
DFT	Discrete Fourier Transform
DMA	Direct Memory Access
DSP	Digital Signal Processing
EasyDMA	Easy-to-use Direct Memory Access
ELF	Executable and Linkable Format
FFT	Fast Fourier Transform
FIR	Finite Impulse Filter
FPU	Floating Point Unit
GCC	GNU Compiler Collection
GNU	Extensive collection of free software
GTK	Free and open-source cross-platform software
HAL	Hardware Access Layer
I2C	Inter-Intergrated Circuit
I2S	Inter-Intergrated Circuit Sound
IEEE	Institute of Electrical and Electronics Engineers
IIR	Infinite Impulse Filter
KB	Kilobyte
LTS	Long Term Support
MB	Megabyte
MIDI	Musical Instrument Digital Interface
MHz	Megahertz

nRF5340	SoC with two Arm© Cortex©-M33 processors
O()	Big O notation
PCM	Pulse-Code Modulation
POSIX	The Portable Operating System Inteface
RIFF	Resource Interchange File Format
SDK	Software Development Kit
SoC	System-on-Chip
SPIM	Serial Peripheral Interface Master
SRAM	Static Random-Access Memory
TWIM	Two-wire Interface Master
WAV	Waveform Audio File Format
WDM	Windows Driver Model
x86	64-bit architecture

1 Introduction

As microcontrollers and their development continue, the yield is more sophisticated, general-purpose, and feature-rich solutions. Audio signal transformation benefits greatly from extensive peripheral support and interfaces. With dedicated digital signal instructions for microcontrollers the revenues of signal processing applications are more in demand.

A microcontroller is essentially a tiny form factor computer manufactured on a single silicon chip. The need for these chips increases as specific purposes appeal mainly because of cost, size, and energy efficiency. In 2021, the global export of microcontroller units grew to 31.2 billion units whereas in 2015 it amounted to 22.06 billion units. [1]

Audio synthesis is the process of generating sound using electronic or digital methods. It involves creating sounds from scratch or manipulating existing sounds to produce new and unique audio signals. Audio synthesis is commonly used in music production, sound design, and other forms of audio processing. The purpose of this thesis is to investigate the use of audio synthesis and actual audio signal transformation from digital audio under memory, performance, and power limitations.

Common applications for existing audio synthesizers include controller keyboards with dedicated computer chips and general-purpose personal computers. However, these devices can be prohibitively expensive, especially for testing. Due to the enormous performance and sophistication of contemporary computer instructions and peripherals, these systems are more than capable of carrying out any work that is asked of them.

The operating system and external libraries used are considered low level software due to the hardware access layer being very close. Low level languages require access to direct access to memory in order to gain performance. The disadvantage of utilizing a low level programming language,

like C, is that it provides very little abstraction from the computer's instruction set, making understanding and implementation more time-consuming.

2 Research

A software library is a collection of non-volatile sources bundled either to work independently or together. A library typically provides a set of functions, objects, or algorithms that can be used to perform common tasks, such as reading and writing files, processing data, and communicating over networks. By using a library, software developers can reduce the amount of code they need to write and can take advantage of tested and reliable components, which can improve the quality and efficiency of their software.

The objective of this thesis was to find a software library explicitly for audio signal processing which would have support dedicated to an audio synthesizer. The criteria for selecting the software would include cost, licensing, maintenance, stability, adoption, and platform support.

2.1 Software licensing

Software licensing is the legal agreement between the software owner and the user that outlines the terms and conditions for using the software. The license agreement typically defines what the user is allowed to do with the software, such as install it on a certain number of computers, use it for a specified period of time, or distribute it to others. The license agreement also outlines any restrictions on the use of the software, such as prohibitions against reverse engineering, modifying the code, or redistributing the software.

There are several types of software licenses, including commercial licenses, open-source licenses, and freeware licenses. Commercial licenses typically require the user to pay a fee to use the software and may include limitations on the use of the software. Open-source licenses, on the other hand, allow users to access and modify the source code of the software and are often free to use. Freeware licenses allow users to use the software for free but may impose

restrictions on the use of the software, such as limitations on commercial use or distribution.

It is important for software users to understand the terms of the software license and to comply with the restrictions outlined in the license agreement. Failure to comply with the terms of a software license can result in legal action, such as fines or lawsuits.

The licensing model would need to permit commercial use, modification, distribution, and private use. The economic impact vector is unknown at this stage so it is advantageous to provide plausible monetary opportunities. Altercations to the library are acknowledged since there is no current implementation that would suffice the needs of this thesis. Distribution should be open but can be more lenient than commercial licensing since it would only hamper third-party users. As the licensing arrangement will demand publishing, private use is required for library usage.

2.2 Maintenance and stability

Software maintenance refers to the process of modifying or updating software after it has been deployed, in order to improve its performance, fix bugs, add new features, or ensure compatibility with new hardware or software platforms. Stability is a critical aspect of software quality, as it ensures that the software can be used effectively by end-users and minimizes the risk of data loss, system downtime, and other negative consequences.

The target software solution maintenance and stability shall be estimated by the amount of pending software bugs, vulnerabilities, and commitment to maintenance primarily measured by the number of pull requests to source code and recent feature and bug fixes.

Adoption and platform support closely favor each other except in cases where multiple frameworks can be used or switched within the operating system. Although the dedicated audio driver can be built upon an audio software sound

architecture which would abstract low-level functionality. A few of such cases are PulseAudio and PipeWire which use Advanced Linux Sound Architecture (ALSA) user-space libraries for higher-level interfacing. This is not relevant to this thesis because there is no user-space audio framework in the Zephyr operating system. As a result, development time is longer because the hardware access is closer to the application than in other operating systems.

2.3 Target hardware and software

For the thesis project, the target microcontroller is nRF5340 SoC which has two Arm© Cortex©-M33 processors dedicated for their purposes as application and the network core. Notable features of the target application core include flexible 128/64 MHz clock frequency for performance-critical applications, 1 MB of Flash and 512 KB of SRAM memory, and support for FPU and DSP instructions. The application processor supports peripherals and interfaces such as I2C, I2S, DMA, and DSP which make this thesis project possible.

The designer and producer of the SoC solution Nordic Semiconductor has a good contribution history for software as well which helps to alleviate the target solution software stack. Since drivers and libraries are already well-tested and in production ready it grants more focus towards the application solution. The nRF Connect SDK provides an environment for scalable application development with drivers, libraries, protocols stacks, and examples. One of the projects to which Nordic Semiconductor contributes is a real-time operating system Zephyr. The operating system has all of the needed support for the research, testing, and implementation of this thesis.

For the purpose and principle of software solutions, it is usually recommended to choose a version to development target environment uses. One of the leading points on why the Zephyr LTS version was chosen was the certainty that the author commission company uses it for their development.

2.4 Analysis of solutions

The research was done as objectively as possible, and the requirements and limitations of the search narrowed it to two software libraries. The options were analyzed by excluding personal coding practices, personal biases, and contributors. The analysis of these solutions had to fulfill software licensing requirements. Stability and maintenance are considered in the estimation of the adoption rate of the software library on different platforms and architectures.

The priority was a software library called FluidSynth. It is introduced as a software synthesizer that has MIDI and SoundFont support. The build system already adopts CMake with the mentioned ability to compile the software library as a shared library. This makes it significantly easier to adopt with an operating system since the Zephyr operating system offers the ability to include and link external libraries during its build stages. The licensing model of the library is GNU Lesser General Public License v2.1 which permissibility fills all cited aspects of licensing required. A second considered option was a software library called Timidity++. It features a software synthesizer, MIDI decoding, SoundFont support, and PCM conversion. The software license is distributed with the terms of GNU General Public v2.0 which allows sufficient permissions to be used for this thesis project.

The fact that the build system were interoperable and configurable with the same language and due to Fluidsynth being already explored and used with Android made the choice of software fitter for the purpose of the this thesis.

2.5 Digital signal processing

Digital signal processing is a field of science and technology in which techniques and applications are applied to transform and operate on signals used. Signal processing has a rich history and its importance is exercised in various fields of engineering such as acoustics, sonar radar, audio generation, data communications, and nuclear science. Out of these audio signal

processing extracts different characteristics from the signal to transform and evaluate it.

Since analog audio is often recorded with different hardware equipment the captured digital signal inevitably has some amount of noise and interference. A signal transmitted over a communications channel is generally perturbed in a variety of ways, including channel distortion, fading, and the insertion of background noise. One of the objectives of the receiver is to compensate for these disturbances. In each case processing of the signal is required. [2]

2.5.1 The Discrete Fourier Transform

The Discrete Fourier Transform (DFT) modifies a fixed sequence of samples of a function into a finite length sequence of proportionate arrangement of samples using the discrete-time Fourier Transform technique which transcribes the function of frequency. The DFT has a frequency domain interpretation of the original signal input. The discrete transform of Fourier transforms is used to achieve Fourier analysis in digital signal processing in that the function of signal that changes over time can be sampled with finite time intervals.

The Discrete Fourier Transform transforms a sequence of N complex numbers x(n) to a different equally spaced sequence of coefficients X(k) (Equation 1). The algorithm is a period function with a sinusoidal period of 2π thus amount of N samples will have can be denoted as 2π/N. Therefore the frequency of the set of samples shall have the form of 2π/N × k where the frequency k = 0, 1, ..., N - 1. The last expression also called the complex sinusoidal part of (Equation 1) is derived from Euler's formula where trigonometric function $e^{i \times x} \cos(x) + i \times \sin(x)$ is transformed into complex exponential function $e^{-i(2\pi/N)nk}$.

$$X(k)=\sum_{n=0}^{N-1} x(n)\,e^{-i(2\pi/N)nk}$$

Equation 1. The Discrete Fourier Transform Formula.

Most commonly used within electronics due to better efficiency is the Fast Fourier Transform algorithm which is used to calculate DFT over a sequence. The algorithm can compute the Fourier Transform in about O(n log n) time instead of the traditional implementation of DFT which has a computation time of O(n2) time. For a large sequence of data such as digital audio, the FFT aggregates considerable gains in processing time and therefore reduces power requirements.

Since the DFT has symmetries it can be taken advantage of to divide the calculation into even and odd number parts that can be done recursively this is also called the divide and conquer algorithm.

2.5.2 Digital filtering

A digital filter is often used to filter a digital signal that is derived from an analog signal through periodic sampling. The specifications for both analog and digital filters are often given in the frequency domain, as, for example, in the case of frequency selective filters such as lowpass, bandpass, and highpass filters. The designing process of various digital filtering algorithms requires that the system has finite-precision arithmetic and discrete time sensibility. [2]

Given the sampling rate, it is straightforward to convert from frequency specifications on an analog filter to frequency specifications on the corresponding digital filter, the analog frequencies being in terms of Hertz and the digital frequencies being in terms of radian frequency of angle. The two

most used digital filtering techniques with the software are finite and infinite impulse linear shift-invariant filters. [2]

Infinite impulse filters achieve extraordinary amplitude response at the expense of non-linear phase shifting. IIR filters have advantages in a variety of frequency selective filters that can be designed using closed-form design formulas. The IIR filter is used in the FluidSynth software library to modulate voices. [2]

For finite impulse filters the closed-oriented design equations do not exist but the filters can have precisely linear phases. Most of the other FIR design methods are iterative procedures requiring exceptionally great computational power due to the complexity of their implementation. In contrast, IIR digital filters can be calculated using tables of analog filter design techniques.

2.5.3 Digital audio signal processing

Digital signal processing is such a wide field of engineering that progressively microcontrollers are gaining more support for dedicated hardware features to calculate Fourier transform in a mathematical unit of the processor to gain performance for such applications. Arm Cortex-M-based microcontrollers allow a low-cost and powerful platform for such a purpose as the target hardware sits under this family of processors the software can harness these features to apply performance-critical algorithms.

Audio frequencies that can be heard by the human ear range from 20Hz to 20KHz. Decoding and encoding audio digitally a method of Pulse Code Modulation is ubiquitous. Pulse Code Modulation (PCM) is a digital audio encoding technique that is widely used for converting analog audio signals to digital signals for storage or transmission. PCM works by sampling the amplitude of an analog audio signal at regular intervals, then quantizing each sample and converting it into a digital code.

Analog signal is sampled at a fixed rate typically in ranges of 20kHz and 96kHz which means the audio signal is measured at that many discrete points per

second. Each sample is quantized, which means that its amplitude is rounded off to the nearest value that can be represented by a specific number of bits. The number of bits used for quantization determines the dynamic range and resolution of the digital audio signal.

The conversion of the digital signal back to an analog signal which humans interpret as sound, the process is reversed. The digital signal is decoded to recover the original quantized samples, which are then reconstructed into an analog signal using a digital-to-analog converter (DAC).

2.6 Audio synthesizer

An audio synthesizer, or simply synthesizer, is an electronic musical instrument that is designed to create, modify, and manipulate sound using various forms of audio synthesis. Synthesizers can produce a wide range of sounds, from traditional instrument emulations to completely unique and abstract tones.

Over the years, synthesizers have played a significant role in many different genres of music, from electronic and dance music to pop, rock, and beyond. Synthesizers have also been used extensively in sound design for film, television, and video games.

Synthesizers typically consist of several modules that work together to create and shape sound. These modules can include oscillators, filters, envelopes, LFOs (low-frequency oscillators), and various other components. By combining and manipulating these modules, users can create complex and varied sounds.

Today, synthesizers are available in a wide range of formats, from hardware units to software plugins and mobile apps. With their versatility and flexibility, synthesizers remain a powerful tool for musicians, producers, and sound designers looking to create new and unique sounds.

The software synthesizer relies on MIDI events to generate, sample, and modulate audio in a predictable and standardized approach. General MIDI protocol implementation has note playing, controller control, channel messages,

and settings configuration. All of these can be found in the FluidSynth software library.

The note-play event handles octave registration from the middle key denoted as 60 which is the C key on a typical piano and to complement that the notes need a scale of volume which is called velocity in the MIDI controller. In combination with the key in octave and velocity the synthesizer has all the required from the user's end info to generate an output. The focused software library has an API for this "fluid_synth_noteon" which needs 4 parameters; an address to the previously constructed synthesizer, the MIDI channel, the MIDI note number from 0 - 127, and the velocity from 0 - 127.

Polyphonic refers to the ability of a musical instrument or device to play multiple notes simultaneously. In musical terms, "polyphonic" refers to a texture of sound in which multiple, independent melodies are played at the same time. In electronic musical instruments and computer music software, "polyphonic" often refers to the number of voices or sound sources that can be played at once. A polyphonic synthesizer, for example, is capable of producing multiple sound sources at once, while a monophonic synthesizer is only capable of producing one sound source at a time. The term is also used to describe multi-timbral synthesizers, which are capable of producing multiple independent sound sources that can be controlled and played back as separate tracks.

In a note-on and note-off command, the notes mustn't have any noticeable break between the interchained notes to achieve the continuous audio stream. This is called Legato in music performance and the term indicates that the musical notes are interconnected to each other so that the transition period is either none or so small that the human ear can not distinguish the disparity of the audio. If there is an intended silence or pause between the notes it is referred to as Staccato which is a design of musical articulation. Legato and Staccato are controlled by a component called "legato detector" in Fluidsynth which acts as an on/off switch similar to a pedal in musical performances. The call stack of MIDI message notes have three scopes; the synthesizer level, the musical articulation level, and the audio generation level (Figure 1).

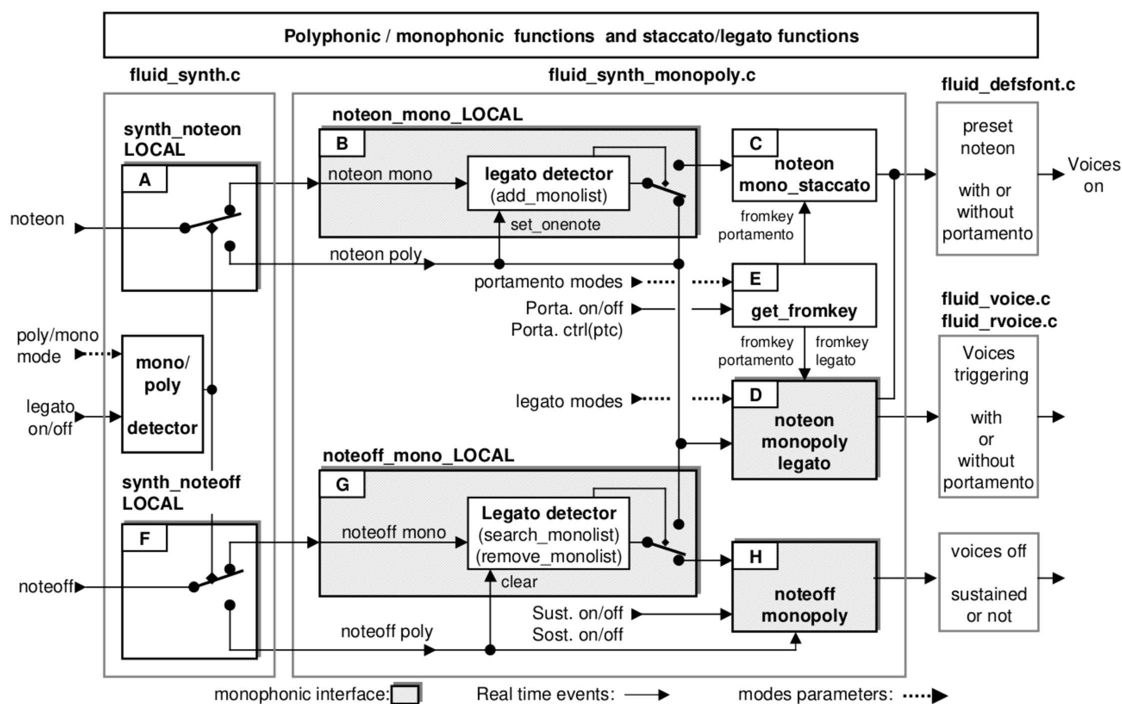

Figure 1. Polyphonic and monophonic functions in FluidSynth.

The application has specific preconfigured settings that will be passed into the synthesizer. These settings used within the context of the library control are used by other modules as well such as the audio driver and SoundFont loader. It is possible to create different settings for diverse audio application purposes however this thesis only distinctively targets one synthesizer instance with a third-party SoundFont.

Synthesizer initialization parses the passed settings and applies them to other modules which may or may not be used by the application since the low-level software does not have any context of high-level software and the library does not offer any preprocessor options to exclude them from the build phase. The synthesizer setups all the default modulators to accommodate further audio processing. Modulators that can create digital effects on the synthesized audio are velocity, attributes, panning, reverb, chorus, pitch bending, and balancing. Internal functions of the synthesizer are voice handling, mixing, and bank selection.

This thesis project focuses on monophonic note playing which is considered the most basic form of testing synthesizer usage. The solution does not aim to use effects such as chorus, reverb, or pitch bending due to the complexity of the software in the cases of testing. For the project showcasing, a channel of 0, a key of 60, and a velocity of 90 have been selected to test the output.

2.7 SoundFont

SoundFont is a file format and technology developed by E-mu Systems for storing and synthesizing digital audio samples, particularly those of musical instruments. The SoundFont file format is a standardized way to store collections of digital audio samples, along with information about how the samples should be played back, such as the pitch, volume, and other characteristics.

In practice, SoundFonts are used with software synthesizers, such as those included in Digital Audio Workstations (DAWs), to create realistic-sounding virtual instruments. A SoundFont file contains a collection of digital audio samples, called "waveforms," that have been recorded or synthesized from actual musical instruments. The SoundFont file also contains information about how the waveforms should be played back, such as how they should be mapped to MIDI notes, how they should be looped, and how they should be modulated.

Soundfont version 2.0 file uses general RIFF (Resource Interchange File Format) which is a multimedia resource file format developed by Microsoft. It provides a common file structure for all of the software contributing to the SoundFont support. Common compatibility is a great prerequisite for the application software because it allows the user to test various SoundFonts as long as they are on the same major version.

The file structure has blocks that allow the data to be partitioned simply into different locations helping further parsing of the file. The basic chunk has for character identification code that tells what type of data is within the chunk, a

32-bit integer that indicates the size of the data chunk, and the actual raw unmodified data chunk.

The RIFF file in SoundFont 2.0 always has the following three chunks. Information-list chunk expressing the optional and required chunks and sub-chunks in the file. The chunk has history metadata, the intended use of the SoundFont, an SDTA-list chunk including a single sub-chunk comprising the raw digital audio samples, and a PDTA-list chunk that has nine sub-chunks describing the modulation and articulation of the digital audio data in the SDTA-list chunk. With the help of the C programming language, these chunks can be expressed in a sequential order (Figure 2).

A RIFF file is constructed from a basic building block called a "chunk." In 'C' syntax, a chunk is defined:

```
typedef DWORD FOURCC;           // Four-character code

typedef struct {
        FOURCC      ckID;            // A chunk ID identifies the type of data within the chunk.
        DWORD       ckSize;          // The size of the chunk data in bytes, excluding any pad byte.
        BYTE        ckDATA[ckSize];        // The actual data plus a pad byte if req'd to word align.
};
```

Figure 2. General RIFF file structure.

In order for the audio synthesizer to manipulate digital audio data it has to reside in the SRAM partition of the microcontroller memory. The Fluidsynth library has prerequisites to allow SoundFont loading from RAM however the implementation has to be implemented by the user. To accomodate this a custom defined file operations API shall be created which will adhere to POSIX file handling API. The file operations callbacks shall be supplied with an API "fluid_sfloader_set_callbacks" from the FluidSynth library. The call requires six parameters; address of the SoundFont loader, file open function, file read function, file seek function, file tell function, and a file close function. These callbacks do not actually read from file in the traditional sense but return various addressing scheming from the raw SoundFont data from SRAM (Figure 3).

```
unsigned long long iter;
void *custom_open(const char *filename) { return (void *)sf2; }

int custom_read(void *buf, long long count, void *handle) {
    if ((count + iter) >= sizeof(sf2)) {
        memcpy(buf, handle + iter, sizeof(sf2) - iter); // Copy remaining
        iter = sizeof(sf2); // Iteration at end
        return sizeof(sf2) - iter; // Remainder of size
    } else {
        memcpy(buf, handle + iter, count);
        iter += count;
        return count;
    }
}

int custom_seek(void *handle, long long offset, int origin) {
    switch (origin) {
    case SEEK_SET:
        iter = offset;
        break;
    case SEEK_CUR:
        iter += offset;
        break;
    case SEEK_END:
        iter = sizeof(sf2);
        break;
    }

    return FLUID_OK;
}

int custom_close(void *handle) {
    return FLUID_OK;
}

long long custom_tell(void *handle) {
    return iter;
}

sfloader = new_fluid_defsfloader(settings);
fluid_sfloader_set_callbacks(sfloader, custom_open, custom_read,
                             custom_seek, custom_tell, custom_close);
```

Figure 3. POSIX file operation callbacks for SoundFont.

Even though SoundFont files provide sample portable interchangeable format they contain raw unmanipulated digital audio data which naturally requires a moderate amount of space. In general, soundfonts are thought to have a small memory footprint, varying in size from a few kilobytes to multiple megabytes depending on the audio fidelity. Despite the fact that they are modest in

comparison to the memory used by x86 computer systems, they are simply too big to be used with the target microcontroller. Fortunately, the portable flexible format can be edited quite easily to strip the SoundFonts of some presets. Another option is to find a tiny SoundFont that can be tested acknowledging the target hardware memory constraints beforehand.

Examination of SoundFonts can be done to figure out the component structure within a font. The font possessess samples, instruments and presets. SoundFont editing and designing can be performed with a popular software called Polyphone. It allows importing sounds, creation of the instrument, audio previewing, recording, bulk editing, transposing, equalizing, filtering, and much more. The software is multi-platform and is available for Windows, Linux, and Mac OS X (Figure 4).

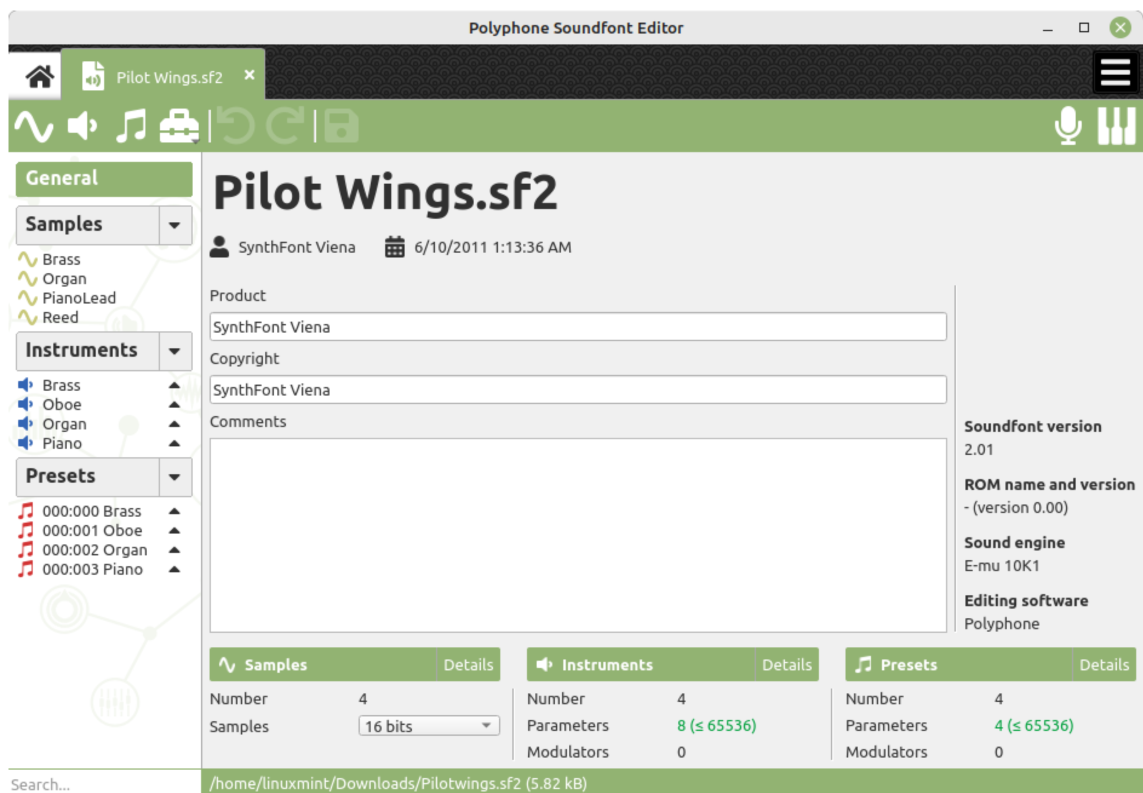

Figure 4. SoundFont inspection with Polyphone software.

To use a SoundFont, one typically loads it into a software synthesizer, which then uses the audio samples and playback information to create a virtual instrument that can be played using a MIDI keyboard or other MIDI input device. SoundFonts can be used to create a wide range of virtual instruments, from simple pianos and guitars to complex orchestral instruments and electronic synthesizers.

2.8 Software porting analysis

Software porting is the process of adapting software to run on a different platform or operating system than the one it was originally developed for. This involves making changes to the source code and resources to ensure compatibility and optimal performance on the new environment. Porting can also refer to the process of moving software from one device to another, such as from a computer to a mobile phone.

With the existing solutions narrowed they have different software requirements which ought to be assessed to estimate the time of porting it to be usable with the Zephyr operating system.

Fluidsynth library uses CMake an open-source building, testing, and packaging software that has multiplatform support. This makes it a good candidate for building a system interchangeable with Zephyr without modifying the GNU makefiles directly. It also saves time in writing the automated build system script by allowing it to be written in CMake language which is arguably easier to approach due to its higher level abstraction compared to GNU Make. Another proposition of language abstraction is the truth that CMake produces GNU Make as an output.

The external software library will be a compiler with Arm Embedded Toolchain which is associated with the selected release of the Zephyr operating system. By the end of compilation, the object files from the target library would be packaged into a static library file which will be referred to as another archive and linked to the other system libraries accordingly (Figure 5). Assembly and

source file dependencies for Arm Cortex M33 are found in the toolchain and compiled into a static archive file. Whereas Zephyr operating system static files are from the operating system directory which is like other prepared dependencies translated into a static library file.

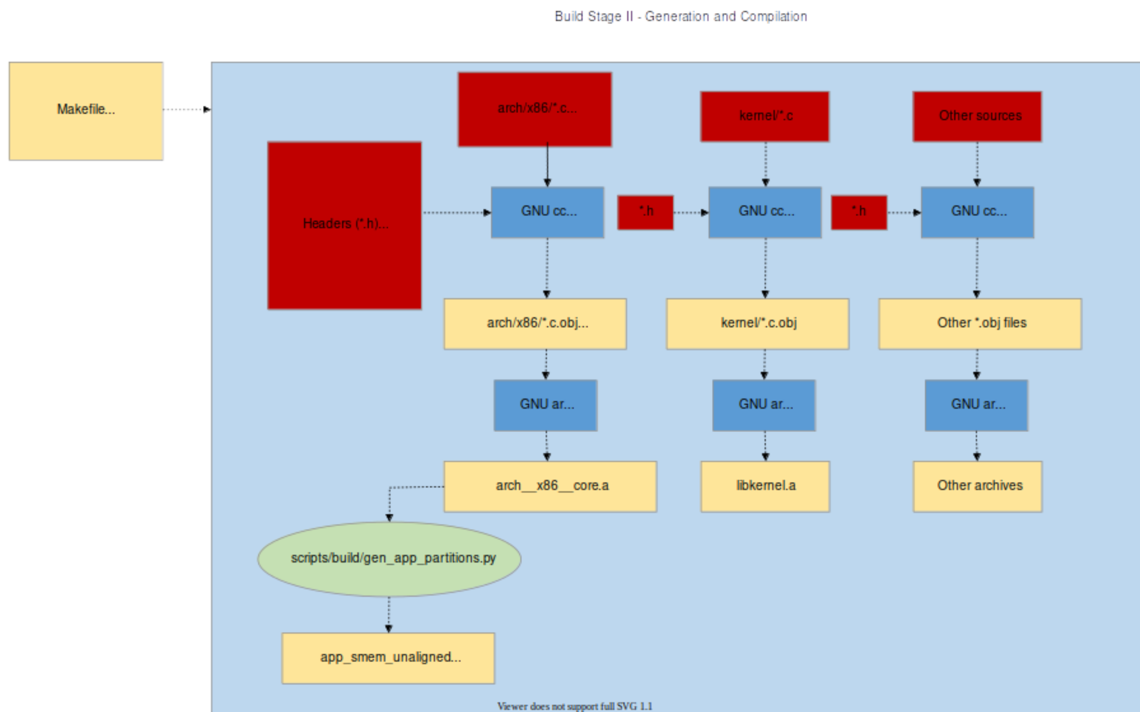

Figure 5. Second stage of compilation.

The fixed-size binary is achieved from the third stage of building (Figure 6). Other archives in this thesis are from the target software library Fluidsynth. Since the binary is unfixed size due to its requirement of post-process steps on a later object handling stage the linker does not need to have a devicetree static data source file in this case.

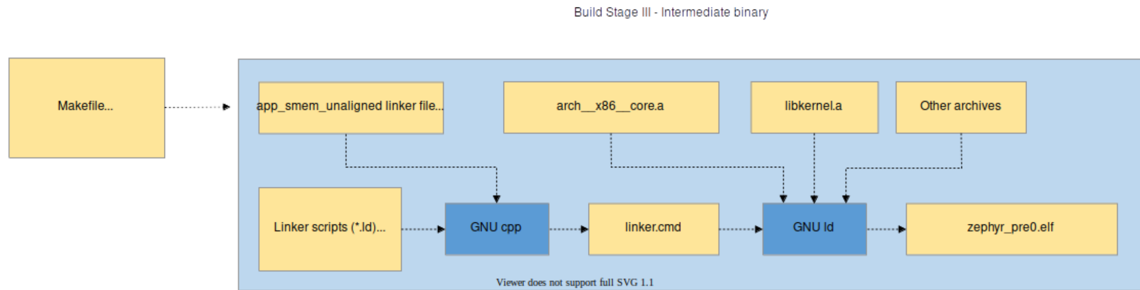

Figure 6. Third stage of linking an intermediate form binary.

The final binary needs to be attained before any statistical analysis can be on the full-sized binary and all its components (Figure 7). The fifth stage of the building phase shall be referred to as the final stage in this case since the file format is sustained for further debugging and testing. The executable and linkable file format in short "elf" is extensible, endian, and address independent used commonly for executables, shared libraries, object files, and core dumps. The format is not bound to any processor or architecture which makes it ubiquitous as a generic executable and object composition amongst embedded processors such as in this case the architecture is ARMv8-M. The device handle object files contain crucial information about the communication busses required to configure the power supplies, audio codec, I2C and SPI busses, I2S peripheral, and DMA handling.

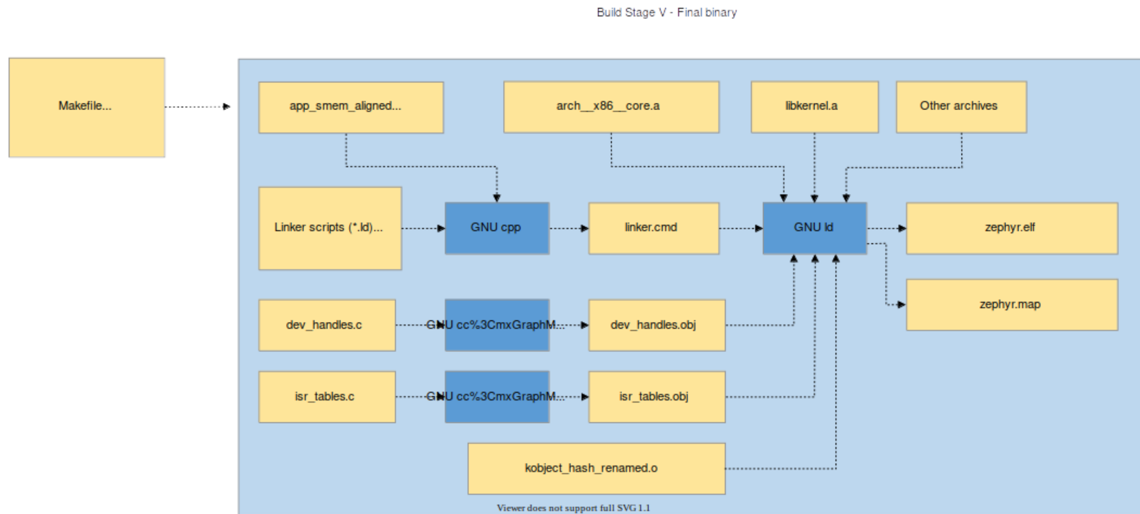

Figure 7. Final build stage.

3 Assessment

Software assessment is the process of evaluating the quality and functionality of a software system or application. This can include evaluating the code, design, architecture, performance, security, and overall user experience. The objective of a software assessment is to identify areas for improvement, potential risks, and to make recommendations for enhancements. The process can be performed by internal or external experts and can involve automated tools and manual testing. The results of a software assessment can be used to inform future development decisions and to guide investment in new technology.

3.1 Risk assessment

Tremendous concerns were conserved amongst the thesis commision company even before software static analysis and compilation and not to mention the runtime testing phase.

Firstly the amount of memory that compiles time and runtime required by the library is hard to estimate without any metrics done by the software writer. Without benchmarks, there is a high risk of honestly not being able to fit software both into the flash and SRAM memory. Therefore a focus was shifted from the target being a microcontroller to an already existing solution temporarily which would provide with some memory consumption statistics.

3.2 Memory constraints

To achieve memory figures, the use of static analysis tools was evident. Essentially all compiled and linked executables have memory sections. These allow the placement of data into different types of memory for instance SRAM and Flash memory. By achieving metrics on both of these memory sections it will be evident to assess applicability level of this thesis.

The text section contains the actual machine instructions which make up the program and the section is further subdivided by the ".initN" and ".finiN" sections. Subpart called ".initN" is used to define the startup code from reset up to the start of the main function. The other section ".finiN" has ownership of the so-called exit code executed after the return from the main function. The data section contains static data which was defined in the code. The ".bss" section contains uninitialized global or static variables. [3]

The compiled program memory analytics for x86 target architecture can be partitioned into non-volatile flash memory and volatile RAM. All of the text section amounts to 22.4 KB, the data section to 2.2 KB, and the bss section merely to 88 bytes (Figure 8). Static memory prerequisites count a total of 24.7 KB.

```
   text    data     bss     dec     hex filename
  22409    2232      88   24729    6099 src/fluidsynth
```

Figure 8. Program memory section sizes from FluidSynth.

As for all of the required drivers, subsystems, and libraries provided by the Zephyr operating system, it is needed to compile an empty application to attain memory statistics for further memory analysis with the external library included. The application enables certain peripherals, drivers, and libraries which will be needed for the implementation stage of the project. These peripherals include the inter-integrated circuit sound interface called I2S, direct memory addressing EasyDMA, floating point unit FPU, two-wire interface master TWIM, watchdog timer WDT, and serial peripheral interface master SPIM. The implementations I2S, TWIM, and SPIM take advantage of DMA for performance and uninterrupted processing for data communication which is another yield leaving more processing power for real-time audio modulation and sampling.

As for the drivers, the software vendor has implemented the serial communication interface I2C, the serial peripheral interface SPI, and the serial sound bus interface I2S. Peripherals and drivers themselves do not depend on

the C standard library but the external library Fluidsynth has system calls that would need to be implemented by hand or use an existing standard library. Zephyr has a version of the embedded C standard library called the Newlib C which contains required system calls and preprocessor definition for the external library.

The Fluidsynth library has dependencies from the Portable Operating System Interface commonly referred to as POSIX. Such calls and preprocessing definitions need to be available during the first stage of the build which is specified by Zephyr as the configuration phase. The external library will need API and filesystem support from the POSIX implementation within Zephyr. In this case, the filesystem support is heavily embedded in Fluidsynth and due to the demand for external filesystem support from the thesis commission party it is enabled in the build. Filesystems would be used to load large files of prerecorded sound data and moved to SRAM during runtime to perform the audio operations with Fluidsynth.

The total amount of previously mentioned software dependencies amounts to 448 KB of SRAM usage and 1MB of Flash memory (Figure 9). The margin for external library remains large considering the SRAM usage is merely 1.26% and the Flash usage is 2.72%.

Memory region	Used Size	Region Size	%age Used
FLASH:	28568 B	1 MB	2.72%
SRAM:	5776 B	448 KB	1.26%
IDT_LIST:	0 GB	2 KB	0.00%

Figure 9. Program memory region sizes from FluidSynth.

3.2.1 Dynamic memory

To elaborate on memory analytics of the compiled software library it does not entirely speak the truth of memory consumption. The C programming language and other compiled languages introduce a consumption of dynamic memory

allocation. The allocation is reserved from the the heap memory which is a region of dynamic memory allocation in computer science. It is used to store data that may change during the course of a program's execution, such as dynamically allocated memory and variables created during runtime. The heap is managed by the operating system and is typically separate from the stack, which stores temporary data, such as function call frames and function parameters. Unlike stack memory, heap memory is not automatically deallocated, so the programmer must manage memory deallocation manually. The heap is also subject to fragmentation, which can reduce the efficiency of memory allocation over time (Figure 10).

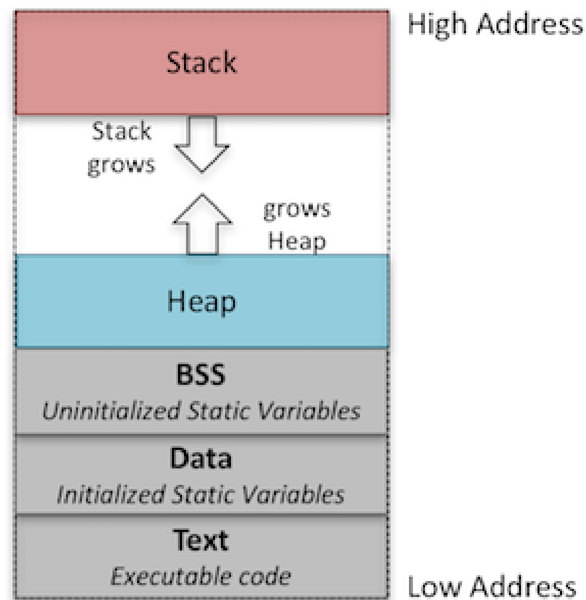

Figure 10. Program memory layout.

Dynamic memory allocation in the programming language C is performed with a group of functions. These functions are "malloc" which allocates the memory, "calloc" which allocates and zeroes the memory, "realloc" which expands previously allocated memory block, and "free" which deallocates previously allocated memory. While dynamic memory management is common with x86

architecture processors it is used less with microcontrollers and microprocessors because of the memory constraints. [4]

Reconstructing ABI identifiers to their source file identifiers is called name demangling. This method enables preliminary memory analysis within the context of the program by exposing symbols and their sizes from the executable and linked file (Figure 11).

```
0000000000033840 0000000000000001 b completed.0
0000000000020480 0000000000000004 R _IO_stdin_used
0000000000033832 0000000000000004 B opterr@GLIBC_2.2.5
0000000000033760 0000000000000004 B optind@GLIBC_2.2.5
0000000000033844 0000000000000004 B option_help
0000000000033836 0000000000000004 B optopt@GLIBC_2.2.5
0000000000033792 0000000000000008 B optarg@GLIBC_2.2.5
0000000000033824 0000000000000008 B stderr@GLIBC_2.2.5
0000000000017056 0000000000000018 t atexit
0000000000000908 0000000000000032 r __abi_tag
0000000000016512 0000000000000036 t print_configure
0000000000014480 0000000000000038 T _start
0000000000016464 0000000000000039 t print_welcome
0000000000026144 0000000000000048 r optchars.1
0000000000014832 0000000000000059 t settings_option_foreach_func
0000000000016384 0000000000000070 t print_usage
0000000000014720 0000000000000105 t show_settings_str_options
0000000000016560 0000000000000485 t print_help
0000000000015872 0000000000000511 t process_o_cmd_line_option
0000000000032800 0000000000000960 d long_options.0
0000000000014896 0000000000000968 t settings_foreach_func
0000000000010464 0000000000004006 t main
```

Figure 11. Symbols and memory sizes in FluidSynth.

3.2.2 Call stack analysis

Call stack analysis is a technique used in computer programming to understand the sequence of function calls in a program at a specific point in time. It provides insight into how a program is executed and can help diagnose errors, such as stack overflows or crashes. The call stack is a data structure that contains information about the current state of a program, including the current

function being executed and the values of its parameters and local variables. When a function is called, a new frame is added to the call stack and when the function returns, the frame is removed. By analyzing the contents of the call stack, developers can track the flow of execution and understand how a program is behaving at a given moment. Call stack analysis is commonly used in debugging, performance optimization, and crash diagnosis.

The FluidSynth software library uses software wrappers that encompass the previously mentioned memory allocation function calls. Software wrapper in software development context wraps around other program components. The software library uses dynamic memory conservatively which makes it more accessible to determine how much it relies on dynamic memory management (Figure 12). Now it is easier to pinpoint every memory allocation call and how much memory it demands.

```
src/midi/fluid_midi.c:184:    buffer = FLUID_MALLOC(buflen);
src/midi/fluid_midi.c:678:            metadata = FLUID_MALLOC(mf->varlen + 1);
src/midi/fluid_midi.c:747:            dyn_buf = FLUID_MALLOC(mf->varlen + 1);
src/midi/fluid_midi.c:809:            tmp = FLUID_MALLOC(size);
src/midi/fluid_midi.c:1473:    track->name = FLUID_MALLOC(len + 1);
src/midi/fluid_midi.c:1886:    fluid_playlist_item *pi = FLUID_MALLOC(sizeof(fluid_playlist_item));
src/midi/fluid_midi.c:1917:    fluid_playlist_item *pi = FLUID_MALLOC(sizeof(fluid_playlist_item));
src/midi/fluid_midi.c:1918:    void *buf_copy = FLUID_MALLOC(len);
src/drivers/fluid_winmidi.c:248:    new_dev_name = FLUID_MALLOC(size + 2 + FLUID_STRLEN(dev_name));
src/drivers/fluid_winmidi.c:554:    dev = FLUID_MALLOC(i);
src/drivers/fluid_coreaudio.c:93:        AudioBufferList *bufList = FLUID_MALLOC(size);
src/drivers/fluid_sndmgr.c:113:        doubleBuffer = (SndDoubleBufferPtr) FLUID_MALLOC(sizeof(SndDoubleBuffer)
src/drivers/fluid_portaudio.c:107:        *name_ptr = FLUID_MALLOC(size);
src/drivers/fluid_coremidi.c:126:    dev = FLUID_MALLOC(sizeof(fluid_coremidi_driver_t));
src/drivers/fluid_oss.c:173:    dev->buffer = FLUID_MALLOC(dev->buffer_byte_size);
src/drivers/fluid_oss.c:409:    dev->buffer = FLUID_MALLOC(dev->buffer_byte_size);
src/drivers/fluid_alsa.c:780:        pfd = FLUID_MALLOC(sizeof(struct pollfd) * count);
src/drivers/fluid_alsa.c:781:        dev->pfd = FLUID_MALLOC(sizeof(struct pollfd) * count);
src/drivers/fluid_alsa.c:1121:        id = FLUID_MALLOC(32);
src/drivers/fluid_alsa.c:1157:        pfd = FLUID_MALLOC(sizeof(struct pollfd) * count);
src/drivers/fluid_alsa.c:1158:        dev->pfd = FLUID_MALLOC(sizeof(struct pollfd) * count);
src/rvoice/fluid_rvoice_mixer.c:605:    newptr = FLUID_REALLOC(buffers->finished_voices, value * sizeof(fluid_rvoice_t
 *));
src/rvoice/fluid_rvoice_mixer.c:631:    newptr = FLUID_REALLOC(handler->rvoices, value * sizeof(fluid_rvoice_t *));
src/sfloader/fluid_sffile.c:724:            if(!(item.fcc = FLUID_MALLOC(chunk.size + sizeof(uint32_t) + 1)))
src/utils/fluid_ringbuffer.c:56:    queue->array = FLUID_MALLOC(elementsize * count);
src/utils/fluid_list.c:37:    list = (fluid_list_t *) FLUID_MALLOC(sizeof(fluid_list_t));
src/utils/fluidsynth_priv.h:187:#define FLUID_MALLOC(_n)             fluid_alloc(_n)
src/utils/fluidsynth_priv.h:188:#define FLUID_REALLOC(_p,_n)         realloc(_p,_n)
src/utils/fluid_settings.c:1837:    str = FLUID_MALLOC(len);
src/utils/fluid_sys.c:242:    wpath = FLUID_MALLOC(length * sizeof(wchar_t));
src/utils/fluid_sys.c:259:    wmode = FLUID_MALLOC(length * sizeof(wchar_t));
src/synth/fluid_synth.c:3625:        fluid_voice_t **new_voices = FLUID_REALLOC(synth->voice,
src/synth/fluid_synth.c:7909:        scores->important_channels = FLUID_REALLOC(scores->important_channels,
```

Figure 12. Dynamic memory allocation in FluidSynth.

Since profiling is not necessary and performance or runtime analytics is not relevant at this stage. The modest approach is to add a counter and a call to the

output stream that yields the counter. Without using program runtime features such as MIDI files or SoundFont loading, the dynamic memory allocation required by the executable amounts to 3.54 MB. Assuredly we have accurate memory metrics from the compiled program. By adding the dynamic and static memory requirements the total amount used by the program is roughly 3.56 MB.

It seems that some amendment has to be done in furtherance of actually respecting the memory constraints of the target microcontroller. In this case, the memory of target nRF5340 SoC has SRAM the size of 512 KB, and Flash the size of 1024 KB. Static memory demanded by the library that is 22.4 KB quite suitably can reside in SRAM. In pursuance of assessing the applicability of software for the given target, it is obvious that the existing software relies heavily on dynamic memory allocation. To be able to exert the practicality of the software dynamic memory usage needs to be rethought.

Microcontrollers do not implement memory management units for hardware simply because it increases power consumption the one key factors why microcontrollers gain an advantage over more sophisticated processors.

The software's exceptional dynamic memory allocation begs the question of where the bulk came from. whether memory is distributed fairly evenly among software components or whether it prioritizes large amounts of memory. The latter is superior in terms of applicability and software development time.

3.3 Performance bottleneck

A performance bottleneck is a factor that limits the overall performance of a system. In computing, a performance bottleneck occurs when a particular component of a system, such as the CPU, memory, disk, or network, becomes a limiting factor for the system as a whole. This can result in slow performance, long wait times, and reduced scalability. Performance bottlenecks can be caused by hardware limitations, software design issues, or a combination of both. To identify and resolve performance bottlenecks, systems administrators

and developers use performance monitoring tools and techniques to analyze system behavior and identify the source of the problem. Once a bottleneck is identified, steps can be taken to resolve it, such as adding hardware, optimizing software, or re-architecting the system.

Performance requirements were almost non-existent for the external library. Simply as a result of the market not demanding extensive audio signal processing in their products or the target processor being sufficiently equipped with high clock frequency and considerable peripheral support. The market corner for niche audio synthesizer applications creates a more generous budget and economic price for such products as dedicated real-time audio synthesizer keyboards. As the market prices of the products are quite bounds to the demand and cost model it makes an insignificant difference whether the product houses silicon which costs ten times as much as a typical microcontroller in the market during the publishing year of this thesis.

The port of the library for the Android Operating System Project (AOSP) and the notion of embedded system appliance exercises further prospects that the performance could be ample for the target hardware of the project.

4 Implementation

Software implementation refers to the process of executing the design and development of a software system. This includes the deployment of the software, setting up the infrastructure, and configuring the system to meet the specified requirements. During implementation, the software is integrated into the existing technology environment, and any necessary modifications or customizations are made to ensure that the system operates effectively and efficiently. The implementation phase is a crucial step in the software development process, as it involves testing and validation of the software to ensure that it meets the desired performance, security, and functionality requirements.

A successful software implementation requires careful planning, coordination between development and operations teams, and close monitoring and testing to identify and resolve any issues that may arise. The central theme of the implementation is solution functionality. It is implicitly hinted that the produced software is pragmatic, constructive, and practical for testing purposes.

The software changes to the library are tracked with Git, a popular distributed version control system that allows developers to track changes to their code over time, collaborate with others, and manage different versions or branches of the same codebase.

4.1 Semantic versioning

Much like the underlying operating system it is crucial to adopt a major version of the audio synthesizer and sampling library FluidSynth. To determine which version to pick for development it is key to pick a version that has the most stable and maintained features and drivers. This is quite tricky because for the project to be successful and to be adopted for over a decade one could argue that its accomplishments would arise from its maintained and stable codebase. For software version control different methods of revision and version tracking is

used. It is very common to be able to see the same notation convention among programmers and scientists which is semantic versioning. The approach marks software for its major, minor, and patch numbers delimited by a period (Figure 13). Many projects also tend to relate a version prefix with the letter 'v' before the semantic versioning handle just preface that they mean a version change.

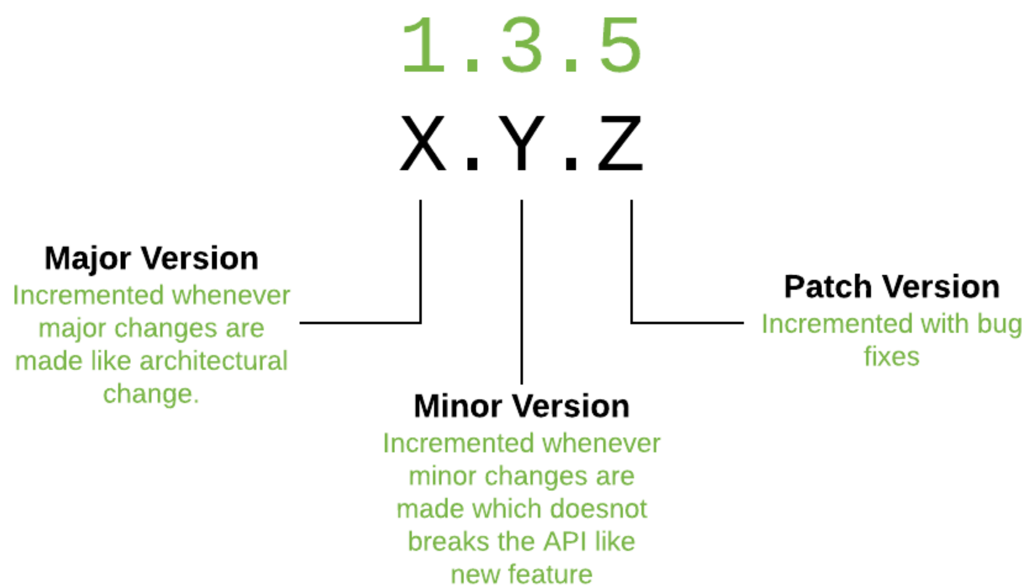

Figure 13. Semantic versioning scheme.

The time between major versions of the audio software library is around a decade which is quite long for an update however period for minor versions is roughly 18 months. Since the project does not introduce the concept of long-term support for its minor versions it is arguable significance whether to pick the latest minor version or the latest minor version with patches. Another developer may pick the other solely for its convention of being the first minor version upgrade and assess whether the subsequent patches would be a useful or decisive factor in choosing the version.

4.2 Dependencies

A software dependency is a relationship between two software components, where one component requires the presence or use of another component in order to function properly. This can refer to software libraries, frameworks, or other components that are required by a program in order to run. Dependencies can introduce complexity into a software system, as they can introduce potential compatibility issues, security vulnerabilities, and difficulties with maintenance and upgrades.

To manage dependencies effectively, software developers often use dependency management tools, such as package managers, that help to automate the process of resolving and updating dependencies. These tools can help to ensure that the correct version of a dependency is used, and that all dependencies are kept up-to-date, which can improve the overall stability and security of a software system.

The toolchain, which consists of the linker and compiler, would first need to comply with C90 standard. Since the build is done with the Zephyr operating system it has an environment called Zephyr SDK which contains all necessary toolchains for building on the ARM 32-bit architecture ARMv8. The targeted Zephyr LTS version comes with its preconfigured and selected tools which are used for this thesis project. Toolchain encompasses GNU Binutils a collection of binary utility tools from their main ones needed without question is "ld" the GNU linker and "as" the GNU assembler. For object handling the use of GNU "objcopy" and GNU "objdump" programs are also crucial to have since Zephyr relies on object copying during its archive construction process. Remember that in Zephyr the building of external libraries is bundled into archives which require archive handling tools such as GNU "ar" which is a utility tool for modifying, creating, and extracting from archives, and GNU "ranlib" used for generating an index to the contents of the bundled archive.

To compile object files a C90-compliant C compiler is needed as noted by the Fluidsynth building from source guidelines. Since Zephyr does not target

restricted versions of the C standard library the codebase does not rely on any of the latest features of the standard library such as C11 which offers a possibility of a specification of C standard compliance to a more desirable version with Fluidsynth such as C99 or C90. Zephyr does not recommend using a toolchain with a compiler with compliance older than C11 however the features which rely on the C11 properties and implementation can be turned off simply by not configuring them in the build.

4.3 Configuration

Build configuration refers to the process of defining and managing the settings and options used to compile and build a software project. Build configuration includes specifying the source code files that should be included in the build, selecting compiler options, specifying the location of libraries and dependencies, and setting other options related to the build process. Build configuration is a critical aspect of software development, as it affects the behavior and performance of the resulting software.

It is important to properly manage build configurations to ensure that builds are repeatable, that builds are consistent across different platforms and environments, and that builds can be easily reproduced if necessary. To manage build configurations, software developers often use build tools, such as make, ant, or gradle, that automate the build process and help to manage build configurations and dependencies. Build tools can also help to streamline the build process, making it faster and more efficient, and can provide detailed build reports to help identify and resolve build problems.

The build shall be configured to be comprised of previously approved drivers, peripherals, and libraries. They consist of software vendor implementation drivers to the lower abstraction peripheral supported subsystems and drivers such as sound interface, serial communication interface, and C standard libraries.

Zephyr offers the build through the usage of its metacommand line software called West. The tool provides higher abstraction from CMake and GNU making commands with more readable and temporarily configurable settings. For the use of this thesis project the evaluation of how the static library is compiled and with what settings it uses is crucial. As the concern is a too high abstraction it has been decided that build configuration and commands will be done with only CMake and GNU make as they cater to more information about the building process which will be affluent at the compilation process.

4.4 Static analysis

Static analysis is a method of evaluating software source code or compiled binaries to identify potential bugs, security vulnerabilities, and other code-level issues. It is performed without executing the code, and instead uses automated tools to analyze the code structure and identify patterns that indicate potential problems. Static analysis can help to improve the quality of software by detecting issues early in the development process, before the code is released or deployed. Some common areas that static analysis can address include security vulnerabilities, performance issues, coding standards violations, and memory management problems. Static analysis is widely used in software development, and is often integrated into the build and deployment process to ensure that code quality is maintained throughout the software lifecycle.

The external library is dependent on some API calls from POSIX and GTK software library which can be assessed by program examination. The Fluidsynth library contains calls and type definitions from these libraries which can not be disabled by any means with a simple preprocessor flag thus it is evident that the external library needs to be modified for the use case.

One of the methods to replace API calls is called a preprocessor directive replacement that defines identifiers as so-called macros which discipline the compiler to replace instances of identifiers found in the implementation source code files. For this instruction identifier theory to work the external library

Fluidsynth requires object and function macros that evaluate either to nothing at compile time or to some system calls which are similar to the ones from the dependency libraries. One could argue that porting a GTK software library would be a more sophisticated method of including everything that the Fluidsynth library demands but the counterpoint to this is code size. Since the GTK already implements some of the same features found in the kernel libraries of Zephyr it is wasteful to define them twice if they inherently perform the same tasks alike.

4.5 Audio driver

Driver development is the process of creating software that interfaces with hardware devices to control and manage their operation. Drivers are essential components of computer systems, as they provide a bridge between the hardware and the operating system or application software. They allow the operating system to interact with hardware devices, such as network cards, printers, and storage devices, and to manage their input and output.

Driver development requires knowledge of both hardware and software, as well as a deep understanding of operating system internals. Driver development can be challenging, as it requires dealing with low-level hardware details, understanding and managing system resources, and dealing with platform-specific issues. To ensure compatibility and stability, driver developers must also be familiar with industry standards, such as the Windows Driver Model (WDM) and the Linux Kernel-Module API.

Device driver is a software layer that lies between the applications and the actual device. This privileged role of the driver allows the driver programmer to choose exactly how the device should appear: different drivers can offer different capabilities, even for the same device. The actual driver design should be a balance between many different considerations. For instance, a single device may be used concurrently by different programs, and the driver programmer has complete freedom to determine how to handle concurrency. [5]

In driver development, the HAL (hardware access layer) provides a set of functions and structures that allow the driver to interact with the hardware in a standardized way, regardless of the specific hardware implementation. This allows the driver to be easily ported to different hardware platforms without needing to change the driver code.

To use the HAL in driver development, you typically need to include the appropriate header files in your driver source code and link your driver with the HAL libraries. The access layer functions and structures can then be used in your driver code to access the hardware components of the system in a consistent and standardized way.

Some common functions provided by the HAL include functions for accessing memory-mapped I/O registers, functions for configuring interrupt handling, and functions for controlling power management and device initialization. By using these functions provided by the HAL, the driver can interact with the hardware in a way that is independent of the specific hardware implementation, making the driver more portable and easier to maintain. In the context of the audio driver, the hardware access layer is accessed indirectly through lower-level peripheral drivers.

Fluidsynth offers quite a plenty of options to use different drivers based on the target machine, architecture, or user preference (Figure 14). From user-space libraries to non-real-time audio synthesis dumping to a file, the library does not meet the demands of the project. Thus a new audio driver has to be implemented preferably by using an existing solution as a template.

```
▾ src/
  ▸ bindings/
  ▾ drivers/
      fluid_adriver.c
      fluid_adriver.h
      fluid_alsa.c
      fluid_aufile.c
      fluid_coreaudio.c
      fluid_coremidi.c
      fluid_dart.c
      fluid_dsound.c
      fluid_jack.c
      fluid_mdriver.c
      fluid_mdriver.h
      fluid_midishare.c
      fluid_oboe.cpp
      fluid_opensles.c
      fluid_oss.c
      fluid_pipewire.c
      fluid_portaudio.c
      fluid_pulse.c
      fluid_sdl2.c
      fluid_sndmgr.c
      fluid_wasapi.c
      fluid_waveout.c
      fluid_winmidi.c
```

Figure 14. Available audio drivers and interfaces.

Emphasis on the project firmware design is high practicability and ease of use without too much diversion from the Zephyr or Fluidsynth library guidelines. Here main functionalities that the audio driver will perform are I2S peripheral configuration, audio codec configuration, MIDI event capture, and I2S streaming. The custom Zephyr shall be called "fluid_zephyr.c" since it is constrained to be used with Zephyr dependencies such as device tree, I2S API, Audio codec API, threading, and type definition support.

The driver shall follow the common notation of explicitly providing a section at the start of the file specifying the license agreement that is the LGPL-2.1 model (Figure 15).

```
/*
 * FluidSynth - A Software Synthesizer
 *
 * Copyright (C) 2022 Niklas Salminen
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public License
 * as published by the Free Software Foundation; either version 2.1 of
 * the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free
 * Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
 * 02110-1301, USA
 */
```

Figure 15. License agreement of the audio driver.

After that much like other audio drivers, the Zephyr custom driver shall include Fluidsynth audio driver and midi driver bindings which come from "fluid_adriver.h", "fluid_mdriver.h" and "fluid_midi.h". The driver shall use Fluidsynth settings conventions to configure and fetch information about configured synthesizer component at runtime thus it will require headers "fluid_settings.h" and "fluid_synth.h". For the memory copy from C standard library the header "string.h" will be included as well. As for the Zephyr API the audio codec has its interface file and I2S driver functions reside under drivers because in this case, the audio driver needs the implementation of the I2S driver which comes from Nordic Semiconductor, not the I2S peripheral implementation (Figure 16).

```
#include <string.h>

#include "fluid_adriver.h"
#include "fluid_mdriver.h"
#include "fluid_midi.h"
#include "fluid_settings.h"
#include "fluid_synth.h"

#if ZEPHYR_SUPPORT

#include <audio/codec.h>
#include <drivers/i2s.h>
```

Figure 16. Type, API and driver dependencies.

It is almost always preferred to catch errors earlier in firmware than in runtime which is why programmers want the compiler to warn them about all errors and warnings beforehand. For this purpose, the Zephyr provides static memory allocation and memory addressing linking for the drivers with the help of the target hardware device tree. By harnessing the power of statically checking if drivers are enabled we can notify the end user in this case it will be a application programmer that the Zephyr Fluidsynth audio driver will be used but will not use the full extent of the driver (Figure 17).

```
/* Check for audio codec existence in devicetree */
#define ZEPHYR_AUDIO_CODEC_EXISTS DT_NODE_EXISTS(DT_CHOSEN(zephyr_audio))
#if !ZEPHYR_AUDIO_CODEC_EXISTS
#warning "Fluid zephyr audio driver does not use any audio codec"
#endif
const struct device *audio_codec;

/* Check for I2S peripheral existence in devicetree */
#define ZEPHYR_I2S_EXISTS DT_NODE_EXISTS(DT_NODELABEL(i2s0))
#if !ZEPHYR_I2S_EXISTS
#warning "Fluid zephyr audio driver does not use I2S"
#endif
const struct device *i2s;
```

Figure 17. Preprocessor symbols of driver instances.

The Fluidsynth audio driver API provides the following type definition to constrain user driver implementation (Figure 18). The predefined interfaces

shall be fulfilled by the Zephyr custom audio driver which will provide the end user with functionality to create, configure, and free driver context.

```
struct _fluid_audriver_definition_t
{
    const char *name;
    fluid_audio_driver_t *(*new)(fluid_settings_t *settings, fluid_synth_t *synth);
    fluid_audio_driver_t *(*new2)(fluid_settings_t *settings,
                                  fluid_audio_func_t func,
                                  void *data);
    void (*free)(fluid_audio_driver_t *driver);
    void (*settings)(fluid_settings_t *settings);
};
```

Figure 18. Audio driver type API constraints.

The creation and configuration of the driver shall take place in type constraint "new" that shall be responsible for the configuration of the I2S driver and peripheral, allocation of memory for I2S streams, the configuration of audio codec, and creating performance critical thread for audio synthesizing and I2S driver triggering. The function call will also be responsible for fetching settings such as the count of channels for stereo or mono audio, period of frames, period size of the frame, sample rate, and sample format. All of the frame settings are used to calculate the amount of memory needed for I2S driver streaming. Since the application usage is very heavy memory constrained every kilobyte of memory is useful for this.

The dedicated thread shall be referred to as a real-time audio synthesizing and streaming thread. It will intercept the MIDI event handler layer to gain knowledge if any MIDI event has happened such as pitch bending or note-on. This is crucial since it will provide a way to block the execution of the thread in other stop the constant audio streaming to save power and not perform tasks if one is not given to it.

4.6 Audio application

The application will use the custom implementation of the audio driver and audio synthesizer already found in Fluidsynth. The application will first configure the synthesizer with settings such as stereo audio, sample rate specific to the target hardware, and audio codec. In the case of using Soundfont support, it is enabled by default in the Fluidsynth builds and has a default configuration driver setup which shall be used to load Soundfont from SRAM address to the SoundFont module. To take advantage of this notation the SoundFont loader component requires the implementation of user file operation callbacks. This will simply return the address of the SoundFont raw data exported from a file to the embedded C source file.

4.7 Compilation

Compilation is the process of converting human-readable source code into machine-executable code. Compilation is typically performed by a compiler, which is a specialized software tool that translates source code written in a high-level programming language into machine code that can be executed by a computer's processor. During the compilation process, the compiler checks the source code for errors, such as syntax errors and type mismatches, and generates error messages if any are found. Once the source code is error-free, the compiler optimizes the code for performance and produces an object file that contains machine-executable code. The object files for multiple source code files are then combined, along with libraries and other dependencies, to create a final executable program.

Compilation is an important step in the software development process, as it allows developers to create programs that run efficiently and can be deployed on different platforms and operating systems. Moreover, kernel code can be optimized for a specific processor in a CPU family to get the best from the target

platform: unlike applications that are often distributed in binary format, a custom compilation of the kernel can be optimized for a specific computer set. [5]

Upon the pre-processing phase of the compilation, it is axiomatic for the compiler to know where to find the expansion of preprocessor directive macros and included header files, removal of comments from the source code, and conditional compilation. Once all the preprocessor directive function calls have been replaced with either Zephyr system calls or nullified to remove them from the source code altogether the inclusion of header files for the external library need to be modified to a degree since the POSIX API does not come preconfigured for the Arm Embedded Toolchain as conventionally for an x86 toolchain it would have. Instead, it comes from the Zephyr source code libraries to wrap the functions to the API and system calls found in Zephyr.

For an x86 compiler such as GCC version 11.0 a user would need to pass a command line argument "-pthread" to enable POSIX threading API support and the compiler simply knows to include necessary headers and link the source files to the application assembly. This is not an option for the embedded toolchain software which is compliant with Zephyr SDK therefore the inclusion of header files needs to be done manually. As for the source files they are required to be enabled by the Kconfig Language in the precompilation configuration phase.

The positive side of using Kconfig language is its extensible scripting support. Zephyr generates a vast amount of build metadata such as absolute pathing to the compiler, object copying tool, object dumping tool, assembler, archiving handling tool, name demangler, debugger, and compilation flags passed to the compiler which includes absolute pathing to the include files. By enabling the Kconfig definition "CONFIG_MAKEFILE_EXPORTS" the Zephyr CMake extensions will output a build metadata file with the include paths to the POSIX API headers that are required for preprocessing stage of the external library Fluidsynth.

For the application build script, it shall load and search for a package called Zephyr operating system from the local machine with a provided absolute path from the environment variable ZEPHYR_BASE. This is recommended way for application developers to find the Zephyr directory where the kernel and all of the other previously mentioned modules reside. The type of application shall be freestanding since the repository is located outside the Zephyr directory and workspace. The minimum recommended version for applications and tests developed for Zephyr is 3.20.0 which is above the version of the Fluidsynth library. Thus it sets precedent for the minimum version for our CMake to build configuration.

The build script within Zephyr has defined the target architecture which shall determine the instructions it emits when generating the assembly code. However, since the application will be built inspired by porting guide from the Fluidsynth documentation instructions the target architecture can be forced with the "add_compile_options" command with a parameter "-mcpu=m33" which is a shorthand notation from Arm Cortex M33 processor.

At this stage optimization focuses on code size because of the memory constraints of the target hardware thus the parameter "-Os" will be passed to the compile options command. It is also important that the optimizations enable inlining since the functions that use the specifier are already well-tested and do not contain any bugs. Inlining however does increase code size but perform better at runtime due to the avoidance of function call that produces overhead from the usage of a stack and allocating data for the return value of the function.

The application will undoubtedly require some runtime debugging such as performance profiling and bug fixing from the modification of the Fluidsynth library. For the compiler and linker to produce and retain information for debugging stage a parameter "-g" shall be enabled for the compiler. The debugging information contains symbols and source-level naming conventions which will be helpful when analyzing stack frames at runtime.

To further analyze the dependency graph from translation units and libraries we need cross-referenced map file across Fluidsynth and Zephyr sources therefore the GCC linker needs a parameter "-Wl,--cref" to the build options to produce it. The main purpose of the common cross-referenced map file is to gain knowledge about the memory locations and particular symbol and variable addressing which will reveal how close the application is running out of memory. Secondly, the cross-reference map file is particularly useful for detailed information about the memory sections such as ".code" and ".data" (Figure 19).

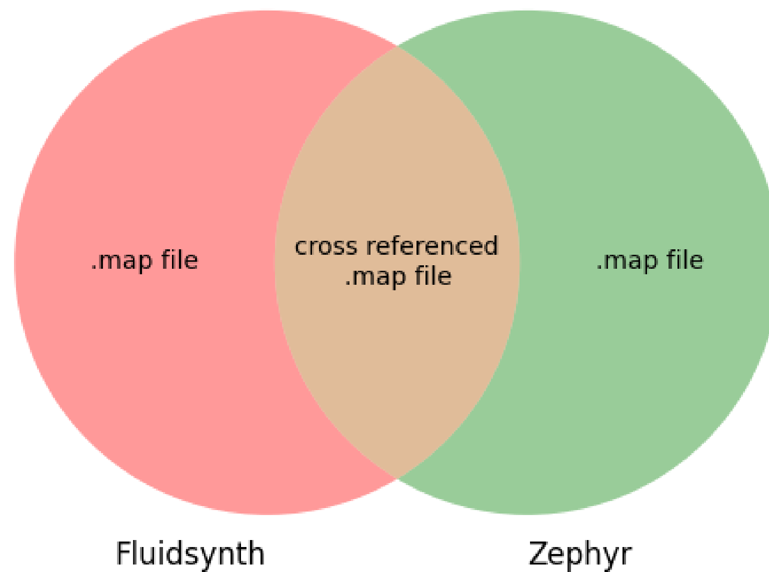

Figure 19. Linker generated cross referenced map file.

By default, the compiler toolchain assumes that the compilation will soft-floating point calculations. Since the target hardware has a dedicated floating-point unit thus the software can take advantage of the fast floating-point instructions for this purpose. Soft floating point instructions are done in software that makes them significantly slower to perform. Before Zephyr had to configure that for the build scripts it has to be enabled in the compile options with the parameter "-

mfloat-abi=hard". The format of floating points shall be specified by making them conform to IEEE format with the option "-mfp16-format=ieee". The application binary interface has to support virtual floating point registers because the target hardware must enable these to save and restore floating points during context switching. To coerce the virtual floating point register support a parameter "-mfpu=fpv5-sp-d16" shall be passed to the compiler.

User-space drivers and libraries are turned on on a need-to-have basis and the top-level build script shall disable all of these because they are not needed. A common way for CMake to find POSIX threading API compliance support is to call find_package with the "Threads" parameter however this is exactly what the build script cannot do since the Zephyr already includes the relevant functions in its libraries. So to pass the preprocessor stage the header files will be given as library paths and source files shall be enabled by Kconfig language with properties "CONFIG_POSIX_API=y" and "CONFIG_POSIX_FS=y".

CMake takes privileged support from pkg-config which is software used for inquiring about packages on the local system and providing paths to them. The general-purpose GLib and the portable threading API library GThread are queried by the build script. Since it is not in the scope of this library to cross-compile the whole of these two libraries the package module checking for them shall be disabled.

To add the application file "main.c" and the custom Zephyr audio driver to the build target the relative file paths shall be appended as a parameter to the "target_sources" in the top-level application build script. Zephyr freestanding test application will be compiled for the same SoC without audio codec to gain more insight and first debugging build for use without audio codec. The purpose of this test build is to catch major errors from the audio driver and bugs as a byproduct of the library modifications. The test builds without Soundfont data fill up the SRAM by 99.23 percent (Figure 20). It is common for memory constraint analysis to at least leave 10 percent of memory as a free capacity for good conscience if the program will have some tasks to perform.

```
Memory region         Used Size  Region Size  %age Used
           FLASH:      196088 B         1 MB     18.70%
            SRAM:      455224 B       448 KB     99.23%
        IDT_LIST:           0 GB         2 KB      0.00%
Generating files from zephyr.elf for board: nrf5340dk_nrf5340_cpuapp
```

Figure 20. First build memory statistics.

A prompt investigation on the size of symbols with the help of the name demangler that comes embedded in the toolchain binary utilities reveals that there is a single symbol that takes up 384 KB called "rand_table" (Figure 21). It is also visible from the name demangler output that the I2S memory streaming buffer "buf" takes up 9.2 KB of space.

```
00002400 b buf  /home/linuxmint/fluidsynth/src/drivers/fluid_zephyr.c:65
00002580 r fluid_ct2hz_tab      /home/linuxmint/fluidsynth/build/fluid_conv_tables.inc.h:3
00002d08 r fluid_cb2amp_tab     /home/linuxmint/fluidsynth/build/fluid_conv_tables.inc.h:1206
00003800 r sinc_table7  /home/linuxmint/fluidsynth/build/fluid_rvoice_dsp_tables.inc.h:2059
00004000 B z_main_stack /home/linuxmint/zp/zephyr/kernel/init.c:44
00008000 B kheap__system_heap   /home/linuxmint/zp/zephyr/kernel/mempool.c:61
0005dc00 b rand_table   /home/linuxmint/fluidsynth/src/synth/fluid_synth.c:4525
```

Figure 21. Deconstructed symbols from the ELF file.

Closer look at the largest symbol found in the ELF file reveals that the huge statically allocated array is used to audio dithering (Figure 22). It is a process for adding random noise to the audio signal to suppress and fix audio distortion issues caused by audio quantization. These unwanted distortions include artifacts and unintended noise from splitting frames from spots that were not intended.

```
#define DITHER_SIZE 48000
#define DITHER_CHANNELS 2

static float rand_table[DITHER_CHANNELS][DITHER_SIZE];
```

Figure 22. Audio dithering array.

The scope of the this thesis is to provide a practical and goal-oriented solution thus the dithering table has to be heavily truncated or get be generated at runtime to eliminate its memory requirements and to free usable memory for Soundfonts data (Figure 23). The implementation of audio dithering uses full-range noise with correction on entropy generation as it calculates the delta value from the previous dithering amplitude. The cost of this audio dithering table which is preconfigured to be used with 48kHz sample rate and stereo audio as the dithering table indicates is not very feasible keeping in mind the memory constraints.

```
src/synth/fluid_synth.c-4540-            d = rand() / (float)RAND_MAX - 0.5f;
src/synth/fluid_synth.c:4541:            rand_table[c][i] = d - dp;
src/synth/fluid_synth.c-4542-            dp = d;
--
src/synth/fluid_synth.c-4544-
src/synth/fluid_synth.c:4545:        rand_table[c][DITHER_SIZE - 1] = 0 - dp;
src/synth/fluid_synth.c-4546-    }
--
src/synth/fluid_synth.c-4753-                *chan_out[c] = round_clip_to_i16(left_in[in_idx] * 32766.0f +
src/synth/fluid_synth.c:4754:                                                 rand_table[0][di]);
src/synth/fluid_synth.c-4755-
--
src/synth/fluid_synth.c-4757-                *chan_out[c+1] = round_clip_to_i16(right_in[in_idx] * 32766.0f +
src/synth/fluid_synth.c:4758:                                                   rand_table[1][di]);
src/synth/fluid_synth.c-4759-                /* advance output pointers */
--
src/synth/fluid_synth.c-4818-    {
src/synth/fluid_synth.c:4819:        left_out[j] = round_clip_to_i16(lin[i] * 32766.0f + rand_table[0][di]);
src/synth/fluid_synth.c:4820:        right_out[k] = round_clip_to_i16(rin[i] * 32766.0f + rand_table[1][di]);
```

Figure 23. The audio ditheting table in source code.

Excluding the dithering table and its usage will grant the opportunity to compile without more available space in SRAM and assess the impact of dithering on the audio synthesizer. By removing the dithering table the SRAM available memory decreased by 83.7 percent leaving much of the desired space to the raw audio data (Figure 24).

```
Memory region         Used Size  Region Size  %age Used
           FLASH:      195412 B         1 MB     18.64%
            SRAM:       71224 B       448 KB     15.53%
        IDT_LIST:          0 GB         2 KB      0.00%
Generating files from zephyr.elf for board: nrf5340dk_nrf5340_cpuapp
```

Figure 24. Second build memory statistics.

5 Conclusion

The main objective of the thesis was to produce a single monophonic note using the target hardware and software and a ported Fluidsynth real-time software audio library. The thesis assessed existing software libraries, weighed risks during and before the project, and implemented a software driver to access the hardware layer.

The subjective assessment of the results was carried out to better understand the possibilities and potential uses of audio signal processing software, not to diminish the significance of the thesis project. More audio processing applications are now possible thanks to the software and hardware used to synthesize the digital audio. Future versions of similar applications will not face as severe performance and memory limitations, and they will also be more affordable thanks to inevitable technological advancements.

The research time took one fourth of the time in the thesis project (Figure 25). The software licensing models had been met with the intention of modifications, redistribution, and publication. Software maintenance and stability were roughly estimated by the software libraries exercises such as commits, pull requests, platform coverage, and unsolved issues.

The author of this thesis had prerequisite experience with the target hardware microcontroller and its specifications as well as the software toolchains and operating system used which massively alleviated the research and implementation time.

Although a thorough understanding of digital signal processing could have been avoided, it was perceived by the author that it is essential in audio software to understand the interrelationship between analog and digital signaling in the context of audio signal processing. Once a software solution was found, the audio synthesizer was tested to understand its internal and basic functionality at the synthesizer level, but not at the articulation or generation levels. A SoundFont used for testing purposes that would fit in the memory of the target

microcontroller was discovered after the author had downloaded a number of sound font from the internet.

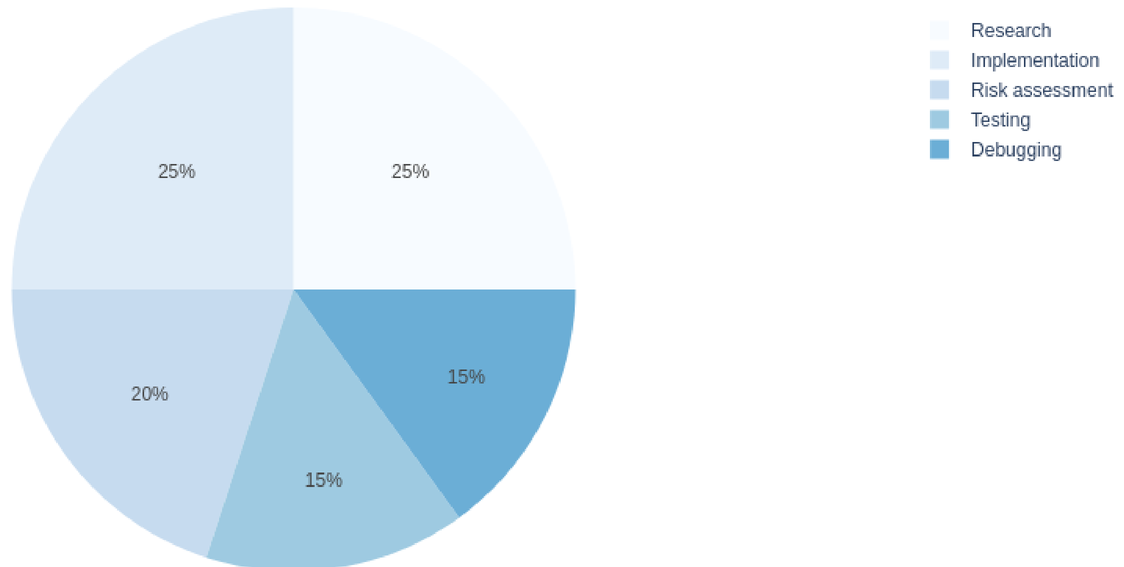

Figure 25. Project time management

The conceivably most challenging part of the research was determining how the software solution could be ported to work together with the target software operating system. Assertably, there were indeed multiple methods to integrate and accommodate the software in the building process. The author deemed that the audio software library should be treated as an external library in contrast to the operating system and included as a static library which seemed to be most practical and it also contained the most documentation.

The risks of the thesis project were well described from the beginning which helped to keep expectations of the outcome reasistic for both the author and the thesis commisioner. It was very important for both parties to designate resources where the engineers would need them. The most valuable resource in this theis project was time and existing software which alleviated much of the groundwork needed to create application software. Even though the software library was not fully qualified for the target software components' interoperation

and hardware, the shortcomings were counterbalanced with caution and years of low-level software experience.

The most crucial piece of software implemented was the audio driver which needed to be compatible with the operating system and software library to yield any digital audio from the audio processing library to the target hardware.

The implementation phase which required the most practical low-level system software knowledge and time took approximately one fourth of the whole project (Figure 25). The software dependencies such as toolchains and standard libraries were smooth to cover and accommodate. However, during the building phase, the audio software library had dependencies on the operating system thus it was obligatory to compile and link them after drivers and peripherals were included.

The adoption of the Kconfig language from Linux Foundation projects such as the Linux kernel allowed an easier predetermined and static build configuration. Static analysis tools and compile time warnings stem some dependencies which were fulfilled by external libraries on supported platforms. However, the author found it necessary to strip these dependencies to achieve less code footprint mainly concerning the available memory on the target hardware.

The software audio driver, the component bridging the software drivers, and the audio software had to be created from the principles of another audio driver template found in the audio software library. In order to showcase the audio driver and synthesizer, a software application had to be implemented aswell.

The compilation phase revealed many of the discrepancies between the two independently built configurations initially. Many of the discrepancies were resolved and regulated from the side of the external library since it was easier to modify them from this side rather than from that of the operating system.

Software testing and debugging were highly pragmatic and similar work in themselves; therefore, they were left out of the implementation chapter. The estimated debugging and testing time amounts to roughly one third of the time

in the project (Figure 25) since statically compiled software needs to be programmed every time a change is made in the source code, which leads to mundane testing.

References

[1] www.statista.com (n.d.). Topic: Microcontrollers. [online] Statista. Available at: https://www.statista.com/topics/9939/microcontrollers/ [Accessed 20 Feb. 2023].

[2] Oppenheim, A.V. and Schafer, R.W. (1975). Digital Signal Processing. Ane Books Pvt Ltd.

[3] www.gnu.org. (n.d.). Memory Sections. [online] Available at: https://www.gnu.org/savannah-checkouts/non-gnu/avr-libc/user-manual/mem_sections.html/.

[4] Cppreference.com. (2019). *cppreference.com*. [online] Available at: https://en.cppreference.com/w/.

[5] Corbet, J., Rubini, A., and Kroah-Hartman, G. (2005). Linux device drivers. Beijing ; Sebastopol, Ca: O'reilly.

[6] Jean J., Lizhe, T., (2019). Image Processing Basics. In: *Digital Signal Processing (Third Edition)*. [online] Available at: https://www.sciencedirect.com/topics/engineering/pulse-code-modulation [Accessed 17 Jan. 2023].

[7] timidity.sourceforge.net. (n.d.). TiMidity++. [online] Available at: https://timidity.sourceforge.net/ [Accessed 19 Feb. 2023].

[8] Administrator (2016). Home - Polyphone Soundfont Editor. [online] www.polyphone-soundfonts.com. Available at: https://www.polyphone-soundfonts.com/ [Accessed 19 Feb. 2023].

[9] cmake.org. (n.d.). Documentation | CMake. [online] Available at: https://cmake.org/documentation/ [Accessed 19 Feb. 2023].

[10] arvindpdmn (2020). Semantic Versioning. [online] Devopedia. Available at: https://devopedia.org/semantic-versioning.

[11] Cppreference.com. (2019). cppreference.com. [online] Available at: https://en.cppreference.com/w/.

[12] www.kernel.org. (n.d.). Kconfig Language — The Linux Kernel documentation. [online] Available at: https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html [Accessed 19 Feb. 2023].

[13] docs.zephyrproject.org. (n.d.). Zephyr Project Documentation — Zephyr Project Documentation. [online] Available at: https://docs.zephyrproject.org [Accessed 15 Dec. 2022].

[14] GitHub. (n.d.). Release fluidsynth 2.3.0 · FluidSynth/fluidsynth. [online] Available at: https://github.com/FluidSynth/fluidsynth/releases/tag/v2.3.0 [Accessed 23 Oct. 2022].

[13] opensource.org. (n.d.). GNU General Public License | Open Source Initiative. [online] Available at: https://opensource.org/licenses/gpl-license.

[14] www.nordicsemi.com. (n.d.). nRF5340 - Nordic Semiconductor. [online] Available at: https://www.nordicsemi.com/Products/nRF5340.

[15] infocenter.nordicsemi.com. (n.d.). Nordic Semiconductor Infocenter. [online] Available at: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fstruct_nrf53%2Fstruct%2Fnrf5340.html [Accessed 19 Feb. 2023].

[16] elinux.org. (n.d.). Executable and Linkable Format (ELF) - eLinux.org. [online] Available at: https://elinux.org/Executable_and_Linkable_Format_(ELF) [Accessed 16 Dec. 2022].

[17] William Kage - Official Website. (n.d.). William Kage - SNES Soundfonts. [online] Available at: https://www.williamkage.com/snes_soundfonts/ [Accessed 22 Dec. 2022].

[18] www.fluidsynth.org. (n.d.). FluidSynth | Software synthesizer based on the SoundFont 2 specifications. [online] Available at: https://www.fluidsynth.org/.

[19] www.synthfont.com. (n.d.). SynthFont, Viena, VSTSynthFont, SyFonOne, MIDI-to-WAV. [online] Available at: http://www.synthfont.com/ [Accessed 19 Feb. 2023].

[20] www.midi.org. (n.d.). The MIDI Association - Home. [online] Available at: https://www.midi.org/specifications.

[21] Techopedia.com. (n.d.). What is Porting? - Definition from Techopedia. [online] Available at: https://www.techopedia.com/definition/8925/porting.

[22] www.arm.com. (n.d.). Resources for Arm products and technologies – Arm®. [online] Arm | The Architecture for the Digital World. Available at: https://www.arm.com/resources [Accessed 19 Feb. 2023].

[23] www.gnu.org. (n.d.). gnu.org. [online] Available at: https://www.gnu.org/software/binutils/.

[24] clang-analyzer.llvm.org. (n.d.). Clang Static Analyzer. [online] Available at: https://clang-analyzer.llvm.org/.

[25] Owasp.org. (2013). Static Code Analysis | OWASP. [online] Available at: https://owasp.org/www-community/controls/Static_Code_Analysis.

[26] www.gnu.org. (n.d.). The GNU C Reference Manual. [online] Available at: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html.

[27] GitLab. (n.d.). *Niklas Salminen / fluidsynth · GitLab*. [online] Available at: https://gitlab.com/xcb3djok/fluidsynth [Accessed 5 Apr. 2023].