Joel Taro Mörlin

Progressing the exploration of portable impulse response using the Sony Spresense

Master's thesis in Electronic Systems Design and innovation
Supervisor: Guillaume Dutilleux
June 2023

Joel Taro Mörlin

Progressing the exploration of portable impulse response using the Sony Spresense

Master’s thesis in Electronic Systems Design and innovation
Supervisor: Guillaume Dutilleux
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Acknowledgements

I would like to thank Guillaume Dutilleux for continued guidance and support throughout this semester. Thanks to Peter Svensson for introducing me to this wonderful topic and providing guidance throughout the prepatory work. Both Guillaume and Peter have been wonderful throughout this entire year and i wish them luck with future endeavours.

Thanks to my Peers for the continued support i have received throughout these 2 years. Special thanks go to my wonderful roommates and fellow members of the PR-committee for providing motivation through this entire process. Finally i would like to thank my family for continued love and support through the last five yeas.

Abstract

As technology progresses at a rapid rate the size requirements for large computations diminish. This allows previously extensive operations such as audio signal processing to become portable, expanding efficiency and possibilities within the field. This report presents a continuation of the exploratory work performed on the Sony Spresense with the aim of establishing a solid foundation for future work and documenting the potential of the platform. Findings include a system still under construction five years after initial release and libraries restrictive to such an extent that they limit system potential. The consequences of a microcontrollers failure to establish a community is discussed together with the overall documentation and developer experience of the Spresense. A complete serial interface is constructed as a foundation for future work on the SDK platform together with a system for portable generation of exponential impulse responses. The presented work shows a platform immensely capable of portable audio signal processing that is ultimately let down by its own developers and failure to appeal to consumers.

Sammendrag

Mens teknologien utvikler seg i raskt tempo, reduseres arealkravene for store beregninger. Dette gjør det mulig for en tidligere omfattende operasjon, som for eksempel signalbehandling, å bli bærbar. Resultatet er en utvidning i effektiviteten og mulighetene innen feltet. Denne oppgaven presenterer en fortsettelse av det tideligere utforskende arbeidet på Sony Spresensen, med mål om å etablere et solid fundament for fremtidig arbeid og dokumentere plattformens potensial. Funnene inkluderer et system som fortsatt er under konstruksjon fem år etter utgivelse, og biblioteker som begrenser systemets potensial. Konsekvensene av at en mikrokontroller ikke klarer å etablere entusiasme hos brukerne blir diskutert, sammen med dokumentasjonen og utvikleropplevelsen som en helhet. Et komplett serielt grensesnitt blir konstruert som et fundament for fremtidig arbei, sammen med et system for bærbar generering av eksponentielle impulsresponser. Det presenterte arbeidet viser en plattform som er utrolig kapabelt til bærbar lydsignalbehandling men som til slutt blir skuffet av sine egne utviklere og manglende entusiasme hos kundene.

Contents

Chapter 1

Introduction

1.1 Commercial microcontrollers

The proliferation of low-cost open-source commercially available microcontrollers has resulted in a revolution within embedded systems. Popularized by general purpose microcontrollers such as Arduino during the early 2010s the access ceiling to embedded programming has diminished down to a commercial level. Distributed with an open source nature the modification potential of these products have been immensely popular among consumers as it allows free creation and adaption of their own tools. This freedom has encouraged the creation and growth of communities dedicated to furthering software developments allowing the platform to reach its full potential.

Popularity growing has led to a lucrative business with several different competitors eager to capture a market share. This competitive market combined with the technology's potential has led manufacturers to create embedded products targeted towards specialized tasks. Allowing a product to surpass competitors within specific application concepts centralizes the competition and allows the product to capture a specific market through a direct approach. An example of such a product is the Sony Spresense, a microprocessor solution based Sony's powerful CDX5602 chipset. Designed for portability, Sony has put special focus on audio processing and sensor fusion performance which makes the Spresense an excellent portable audio analysis platform.

1.2 Background

Two previous projects have explored the potential of using the Sony Spresense for auditory purposes at NTNU. Running simultaneously during spring 22' the work covered auditory recordings of insects. Designed to benchmark Spresense capabilities, the work clearly illustrated the earliest reported issues while proving the raw power of the technology. Both projects ran the Spresense in a multi analysis configuration with the system performing tasks in parallel. This was aimed at exploring the ability to exploit the multi core CPU. Although both papers [1] [2] attempted differing parallel interfacing the same issue was encountered. Final conclusions where that the main issue lied with the Arduino library support for recording and subsequently storing audio and video in a parallel environment. Both papers further concluded that programming the Spresense using the SDK environment would fix this issue. However, it remained unproven as neither paper decided to further pursue this theory due to time constraints.

Preparatory work for this thesis was performed as to familiarize with the Spresense. Main focus was put on exploring the three different development environments. As previous work had been done exclusively in Arduino the projects focus was to verify the claimed restrictions and analyze if similar limitations where present in other environments. Testing was performed through attempted implementation of an audio playback example combined with an OLED display. Doubt had initially been placed upon the conclusions drawn by previous work. Although of higher abstraction than the SDK environment, the Arduino should not be restricted in a differing manner as both environments utilize the same libraries. Analysis of the SDK documentation confirmed that a number of libraries including the Audio where restricted from use in sub cores. This prohibits all use of audio tools in multi processing confirming initial suspicion.

Further testing concluded that the python environment was unsuited for all future work. Official support for the python environment is minimal with most necessary libraries not currently available. This includes the audio libraries making implementation of any impulse applications difficult. Python being interpreted and thus inefficient for embedded applications[3] further compromises its potential and allows the environment to be deemed unsuited for future work.

Time restrictions forced the project to skip full exploration of the SDK preventing full conclusions to be drawn. However the SDK documentation alluded to an environment better suited for embedded applications. Thus; although confirmed that the SDK environment would not resolve the restrictions faced by previous work on the Arduino it was decided that the SDK should be the main environment moving forward.

1.3 Intent

Previous work has proved the Spresense as a flawed but capable platform for acoustic applications. Being in early development constricts collective knowledge to official documentation allowing system restrictions to be undocumented. Although some records has been conducted in regards to previous work at NTNU, these have primarily focused on aspects related to acoustics. As such the embedded perspective of the Spresense has not been explored. The SDK is of peculiar interest as all previous projects has been conducted utilizing the Arduino. Exhibiting enhanced resemblance to traditional embedded environments and being in closer proximity to the compiler grants the SDK a theoretical increased potential. Although not a definitive solution as stated by past contributions, study of the SDK environment is essential for accessing the complete potential of the Spresense. This thesis will focus on continuing the exploration of the Spresense to further investigate its potential in auditory work at NTNU. Focus will be put upon the embedded aspect of application construction with the aim of uncovering and resolving uncertainties not adequately covered by official documentation. Issues encountered by previous efforts are of extra interest and will be explored with the aim of further documentation or rectification if possible. The SDK environment is currently undocumented by NTNU causing assumption mistakes in regards to system capabilities. Absence of knowledge about system limits makes spec construction difficult potentially risking project failure if requirements are impossible. This thesis aims to aid future work by performing a complete analysis of the SDK environment with intent of providing vital guidance. Additionally, the attempted construction of a portable impulse response measurement system will provide a foundation for future work on the platform. The Spresense is a new embedded platform under current development by Sony, it is therefore crucial that it is explored and documented properly if to be used by NTNU. Through mapping limitation and features this thesis aims at providing a foundation towards all future work hindering the uncertainty plaguing current work on the platform.

Chapter 2

The Spresense

Introduced in July 2018 the Spresense is a semi recent addition to the microcontroller market. Designed towards hardware performance and low power consumption it presented great potential for portable IoT solutions. The Spresense is Sony's first endeavour into microcontrollers and presents an unique environment and potential compared to competitive platforms.

2.1 Software

Traditional microcontrollers utilize a low level programming language compiled and flashed to an embedded OS designed to store and execute a single executable at a time. Commercial introduction combined with technological advancements has resulted in several variations on the traditional formula. Arduino being an early contributor still utilizes the traditional microcontroller standard but relies on their own Arduino programming language. Unique to the Arduino development environment this language is not directly able to be compiled thus requiring a interpretation before compilation 2.1.2. This allows the Arduino language to be of simpler nature, lowering the knowledge barrier and decreasing construction time. Rising interest in general purpose microcontrollers combined with technological advancements has allowed a constant increase in power and cost effectiveness. Less restrictions by hardware allows the implementation of more intricate operating systems with current microcontrollers attaining stronger resemblance to computers than embedded platforms. Recent examples include the Arduino Portenta X8 running an embedded adaptation of Linux [4] and the Sony Spresense utilizing Nuttx.

2.1.1 Nuttx

Developed by the Apache software foundation Nuttx is a embedded scalable RTOS designed for the purpose of microcontroller use [5]. Driven by a nonprofit organization Nuttx is designed towards an open source nature allowing tailoring to specific products needs. Nuttx supports several development environments with Linux and GNU make being the most natural development environment [6]. As compilation is based on GNU make, Nuttx development not recommended on Windows platforms. GNU compilation relies on file indexing, a process the Windows NTFS journaling system is not optimized for [7] resulting in poor compilation performance. Although possible to utilize Windows, the performance loss is of such an extent that Spresense documentation warns users and recommends a switch to Linux. Newer windows editions include WSL allowing the operation of a Linux distribution without reliance on a virtual machine. Granting the ability to run a windows distribution without compilation performance loss.

The Spresense utilizes a modified rendition of Nuttx suited towards the CDX5602 microcontroller. Structurally similar to the regular OS, modifications are minor and include the additions of Spresense specific libraries and features. Being indistinguishable from distributed Nuttx during application construction the modifications are observable during system configuration. Nuttx is designed towards flexible development, as such nearly all features are configurable allowing exclusion and inclusion dependent on hardware and application needs. Configuration utilizes the Linux Kconfig system allowing easy alteration through the "menuconfig" frontend. Spresense specific features and libraries likewise need to be modifiable through the Kconfig, as such the menuconfig frontend has been modified to suit this requirement. Nuttx default configuration include no non vital features requiring application construction to begin with a complete configuration build. Allowing the prevention of bloat but necessitating the manual inclusion of all necessary application features. To prevent manual configuration each time a new application is loaded Nuttx supports a storage system where builds can be saved and accessed through a naming index.

2.1.2 Development Environments

Supporting three differing development environments allow the Spresense to present a competitive platform in several markets. Each environment yielding differing potential and accessibility makes the Spresense lucrative to multiple developer classes allowing a larger consumer base.

Arduino

Arduino is an open source programming language designed for the Arduino development software. Based upon C++ it bares a similar structure with the addition of predefined functions and methods. Although considered an independent programming language Arduino .ino files are a set of C/C++ functions called from the code. Official documentation utilizes the name "sketch" as the files are "sketches" of program interpretation. When compiled the Arduino sketch is interpreted by the Integrated Development Environment (IDE) which performs minor pre-processing to convert the sketch into a C++ program. The programs dependencies are located and passed on to a AVR-GCC compiler that will compile the code into a single machine readable intel .hex file that is passed to the board via the bootloader. The AVR-GCC compiler used is architecture specific and will vary from what board is chosen and connected to the IDE. However, caused by the Spresense not utilizing an AVR chip for the CPU the environment instead relies on a gcc cross compiler for compilation.

Main advantages of development through Arduino IDE are vast amounts of resources and ease of use. The IDE itself is "plug and play" in nature with the environment automating the installation and flashing process. As such, the setup only requires selection of the correct board model in the IDE. Resulting in a minimal idea to prototyping time, beneficial for projects with tight time budgets. Moreover the IDE provides a set of preinstalled development tools that require no set-up or additional installs. The most significant of these additions is the Serial Monitor and Serial plotter. Serial communication is beneficial in microcontroller development as it allows live surveillance of program running order through writing and reading the serial bus. Although serial writes significantly diminish program execution they are vital in allowing live monitoring of program running order. Serial communication is not possible in native windows and require additional tools such as putty. This is a non issue in Linux based systems as they allow SSH connections natively. However this is not applicable to the Arduino IDE as the necessary Serial communication software is bundled with the IDE further adding to the ease of use and aforementioned "plug and play" nature.

Python

Python is an open source high level object oriented programming language that trough a emphasis on simplicity and code readability has become one of the most popular programming languages for both beginners and professionals [8]. With an extensive standard library Python ensures a short and simple idea to prototyping time with minimal required focus on package management. Being open source in nature combined with its popularity has also led to communities forming which in turn develop their own libraries and packages [9]. The result has been one of the best supported programming languages to date with a substantial amount of official and community made packages. This can be observed in the word of digital signal processing where the python library dsp-py have been developed to mimic the functionality of industry titans like Mathworks Matlab [10]. With an simple prebuilt package manager these additional packages can be installed in seconds making it possible for some sectors to use python as their only programming language.

Being a powerful and user friendly programming language great effort have been invested to port Python to as many platforms as possible. Microcontrollers are no exception and there are currently two Python renditions optimized to run on microcontrollers. MicroPython was originally released in 2014 and is a software implementation of python largely written in C. Featuring a reduced library subset allowing it to run on systems with 256k of code space and 16k of ram MicroPython is optimized to run in constrained environments making it suitable for microcontrollers [11]. CircuitPython is a open source derivative of Micropython released in 2017. Development is supported by Adafruit industries and the language is aimed at simplifying MicroPhyton to make embedded programming as accessible as possible [12]. The backing from Adafriut allows Circuitpython great hardware support in addition to legitimacy.

Python distinguishes itself from other embedded languages by being interpreted. An interpreted language is not compiled but is instead aided by an interpreter that compiles and executes the code during runtime. The main advantage of interpreted languages are their flexibility, as the code is compiled on a line by line basis during runtime changes to the program without recompilation is possible. Furthermore, the languages are platform independent as compilation and execution are reliant on the interpreter and not hardware specific compiled machine code. Disadvantages of interpreted languages are speed and hardware control. Interpreted languages are slower than their pre-complied counterparts as the compilation is performed during runtime.

SDK

A Software Development Kit is the complete development package for a given system. An IDE and SDK perform the same tasks, however the IDE gives an interfaced access to the SDK toolset making development through an IDE easier than purely through an SDK. Sony's official development kit for the Spresense is called Spresense SDK and is based on the open source NuttX RTOS [13]. Development using Spresense SDK can be performed through a Command Line Interface or Sony's "Spresense VSCode IDE" extension for VSCode. The CLI in combination with a text editor such as Emacs or VIM is the preferred choice for some experienced programmers as it is more efficient for large systems However, an IDE comes with the benefit of understanding the code written by the programmer. This allows the IDE to detect and suggest fixes to compilation errors, warn the user if syntax is broken and automatically suggest code during programming. This is however dependent on the quality of the IDE and will vary depending on the language and platform used.As the compiler used is the same for both CLI and IDE the choice of what platform to use is decided by the developers own preference. Nonetheless, the beginner friendly nature of the IDE makes is the recommended choice for inexperienced and intermediate programmers.

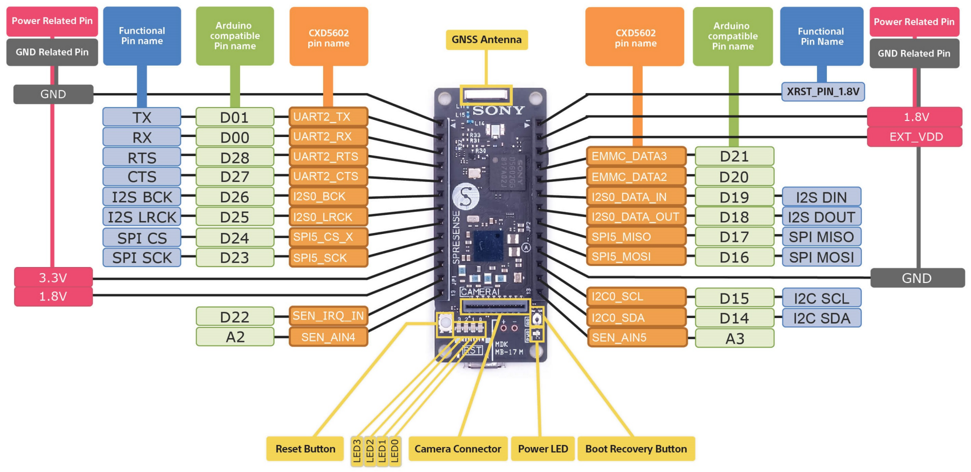
The Spresense SDK toolset relies on the GNU Arm Toolchain for compilation and thus allows programming in C and C++. These are compiled languages and considered the current norm for embedded programming. Unlike interpreted languages discussed in section 2.1.2 compiled languages are compiled into hardware specific machine code before compilation. Compiled languages are much faster than interpreted and preferred for microcontollers as the platform puts a higher emphasis on performance and efficiency. Both C and C++ also allow for direct memory allocation through the use of tools like pointers. Memory allocation is vital in embedded programming as the tight tolerances require complete memory control to maximize performance.

Stemming from the same family of languages C and C++ share similar aspects. However, although all C++ compilers support C linkage the age and thus development time of the two languages have caused them to become distinctive. Being object oriented C++ allows for higher expressive power and easier code recycling. This expressive power comes at the cost of complexity making C++ harder to master with an increased difficult of grasping total program size. With memory optimization crucial for embedded programming this volatility makes C++ unfit for certain systems requiring extreme optimization. Being introduced in the 1970s C is a simple and strict language. This allows less expressive power but a quicker learning curve and a better grasp of total programming size. With its age C is per writing a common industry standard allowing for superior documentation and support.

2.2 Hardware

The Sony Spresense is an ARM Cortex -M4F 32-bit RISC based microprocessor specialized for auditory and video processing purposes. The Spresense achieves this performance through a 6 core rendition of the M4F processor running at 156 MHz. Although running at a lower frequency than the competition the multi core arrangement allows the Spresense excellent throughput for its price range. The nearest competitor, the Arduino Portenta H7 combines a single core Arm M7 running at 480MHz with a single core Arm M4 running at 240Mhz [4]. Despite yielding impressive computational power and outperforming the Spresense for single core operations the core count of the Spresense allows greater throughput. This allows the Spresense to greater exploit the parallel nature of digital signal processing.

Figure 2.1: The Spresese mainboard portlist[14]

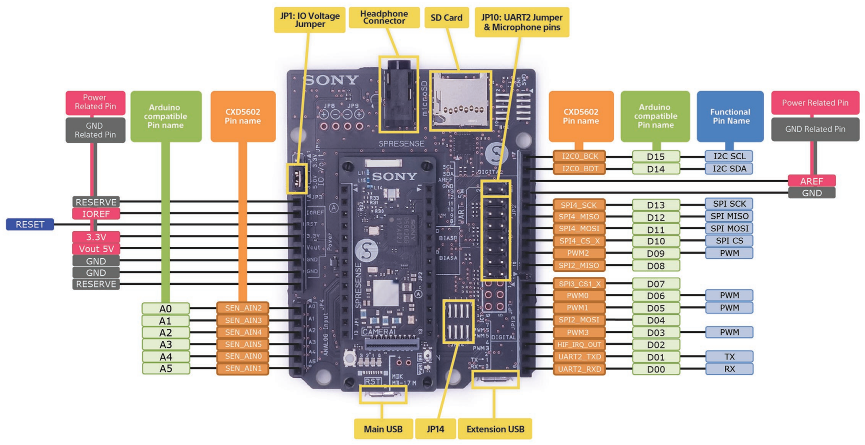


The Spresense contain a large set of unique features on a small footprint. Incorporating a standardised set of GPIO protocols with the inclusion of the audio oriented I2S allows the Spresense to achieve standard microcontroller flexibility. Facilitating the focus on media the Spresense features a Camera Serial Interface which in conjunction to the aforementioned I2S GPIO allows full audio and video support [14]. Satellite navigation is supported through the GNSS antenna and 2 ADC channels allow analog interfacing. The small scale of the Spresense combined with the large set of features to result in limitations. As such support for external interfacing has been included through a B-2-B connector allowing expansion of the Spresense feature set at the cost of size.

2.2.1 Extensions

Considerate of its small size the Spresense was designed towards the expansion of features through extensions. Sony currently offers two differing expansion boards, LTE and a general port extension. Both interface through the B-2-B connector and add designated microphone pins, a headphone connector and support for a micro SD card. Similar in construction the two boards differ in application. Implied by name the LTE extension board allow communication via LTE-M network through a SIM. At the time of writing the connection is only supported by 4 LTE providers making the extension highly specialized. Less specialized, the "general" extension board increases the GPIO count and adds the ability to tweak pin voltage through jumper adjustment.

Figure 2.2: The Spresese extension portlist[14]



Offering an significant increase in features, the extension board emerges as the the default and recommended configuration. Extended storage through SD-card support allows the Spresense to operate with large file mediums including audio and video. This is further encouraged through the inclusion of four designated microphone pins an the inclusion of a 3.5mm audio jack for playback. The reduced functionality induced by the small footprint of the main board limits application possibilities. Although designed towards media application the restrictive design of the Spresense results in wasted resources if not paired with the extension board.

2.3 Support

Sony provides documentation for all three development environments and official extensions available to the Spresense through their "Developer World" website. Designed to act as a central hub for development the website in theory provides everything required for creation on Sony products. Structure is conformed to the product and will thus vary depending on type and needs. The Spresense development site is partitioned into 7 sections, each containing respective subsections. The most vital are the sections dedicated to the three design environments available to the Spresense. Containing details regarding installation and setup as well as API guides and explanations to each example application the section is essential in granting knowledge of the development environment. Other sections detail general introductory information, details and explanations about hardware and links to support and release notes. Sony also hosts their own developer forum allowing users to discuss development of the hardware and software featured on the "Developer World" site. The forum is moderated and supported by official Sony representatives allowing questions and discussions to be answered by professionals. Official support allows the forum a responsibility for providing feedback allowing user input to contribute to development. Although an excellent resource the forum lacks a search function restricting users to manual searching the entire forum for information. Individual forum posts are further not treated as pages, as such they are not included in search engine indexes making them unavailable for direct linking. At the time of writing forum searches are thus limited to manually scrolling and reading topic titles leading to poor User Experience.

Chapter 3

Application construction

Impulse measurement using a single microcontroller requires a collection of multiple individual programs to work in harmony. In order to successfully gather impulse responses each part of the system will have to work flawlessly as run time errors or simple timing mistakes will jeopardize recorded data. Thorough testing and construction of the system is therefore imperil as this will minimize the risks of program errors. The completed system has shown potential for real world applications, thus great emphasis will be put on program design. Modality and organization will allow for a greater user experience whilst also quickening understanding and modification of program code. The latter is of significant importance as if the project should fail to achieve the spec or the program is to be reworked the effort will be minimized. This chapter will present the construction and hierarchy of the complete system as described by the spec.

3.1 Serial Communication

Discussed in 2 the Nuttx Operating System lacks native two way serial communication support. The ability to communicate directly with the hardware during run time is not crucial as the Nuttx operating system allows flashing of multiple programs simultaneously. This allows debugging by splitting code into multiple programs and calling them directly from the OS. However, communication through serial is preferred as this allows direct program control during run time through buffer comparisons with a state machine. Initial focus will therefore be on the construction of a two way serial communication system. This will provide the basis for all future testing and control of the system. The function of this design is therefore crucial and ensuring correct operation necessary before moving on to other features.

Official Sony documentation for the SDK environment list no resources for the implementation of a serial communication system. However, with the SDK environment being based on a more prevalent operating system several external resources for Nuttx can be used. The serial protocol implementation is based on the Nuttx serial guide[15] provided by micro-ROS. Originally developed for microcontroller based robotics the code and guidance provides a sufficient baseline and is fully compatible with the SDK-rendition of the OS.

When activated the serial communication will open the UART with read and write permission along with initializing a while loop. The while loop performs active polling on the UART and stores transmitted characters in a buffer. As a result of Nuttx limitations in the UART read interface, only one character can be extracted per cycle. Behaviour will continue until a return character is detected. Indicating that the user is finished typing the arrival of the return will trigger a UART write-back of the buffer displaying the message sent and wiping redundant data. Although forming the basis of the UART communication, the code lacks a functional interface to directly control program running order. This can be achieved by implementing a buffer comparison in the write back stage allowing certain commands to trigger specific system functions. Calling a function during a while loop will temporarily break it until the function is returned creating a command based state machine. Finally, as the GNU based compiler does not support strings the Serial State Machine "*SSM*" is aided by a "String Compare" function. This function simplifies the comparison of char arrays operated by the serial protocol.

Code listing 3.1: Simplified example showing the Serial State Machine

```c
// char to string conversion //
bool stringCompare(char *a, char *b){
  int sizea = sizeof(a);
  for(int i = 0; i <= sizea-1; i++){
    if(a[i] != b[i]){
      return false;
    }
  }
  return true;
}

//UART communications //
bool uart_comms(){
  printf("initializing␣uart\n");
  int fd;
  char buffer;
  char buffer_aux[256] = {};
  int ret;
  int i = 0;
  //command buffers
  char quit_buf[4] = {’q’,’u’,’i’,’t’};
  char test_buf[4] = {’t’,’e’,’s’,’t’};

  fd = open("/dev/ttyS0", O_RDWR); //Open the uart with RW permission
  if (fd < 0) {
    printf("Error␣UART");
  }

  while (1) {
    ret = read(fd, &buffer, sizeof(buffer));//It return only a char
    if (ret > 0) {
      buffer_aux[i] = buffer;//Saving in the auxilary buffer
      i++;
      printf("%c", buffer);

      //user presses enter
      if (buffer == ’\r’)  {
        ret = write(fd, buffer_aux, sizeof(char) * i); //writeback
        printf("\n");

        //just a test of the uart
        if(stringCompare(test_buf, buffer_aux)){
          printf("well..␣the␣uart␣works\n\n");
        }

        //termination of the program
        if (stringCompare(quit_buf, buffer_aux)) {
          printf("quitting␣the␣program...␣\n");
          return false;
        }

        if (ret > 0) {i = 0;}
      }
    }
  }
}
```

3.2 Sine wave generation

Early specification listed interest in performing exponential sine wave generation on chip. Although several external tools existed and the process of transferring files to the Spresense was observed to be fluid. The practicality of having portable impulse generation was too large to ignore. Should a tool be developed that allows the hardware to generate impulse noise the Spresense would provide a fully independent platform only requiring external computation for post-processing. As such the specification listed a tool set for generating exponential sine waves and storing them on the Spresense SD card in the .Wav format.

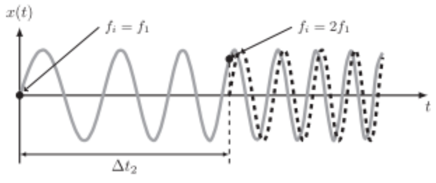
Table 3.1: Header structure for Wav file

Offset (bytes)	Field Size (bytes)	Field Name	Sample Value
0	4	Chunk ID	"RIFF"
4	4	Chunk Size	File size - 8 bytes
8	4	Format	"WAVE"
12	4	Subchunk1 ID	"fmt"
16	4	Subchunk1 Size	16
20	2	Audio Format	1 *PCM*
22	2	Num Channels	2
24	4	Sample Rate	44100
28	4	Byte Rate	176400
32	2	Block Align	4
34	2	Bits Per Sample	16
36	4	Subchunk2 ID	"data"
40	4	Subchunk2 Size	total file size
44	–	Data	sample data

Before data can be written a .Wav file a correct header needs to be constructed. This header file contains information about the file such as the waveform data format, sampling rate and channel count. Pictured in appendix **??** the header is responsible for giving information regarding interpretation of the bit stream contained in the file. Correct header construction is therefore crucial as errors will lead to file corruption. Header and file construction is based upon [16] which utilizes a struct based system to establish and write header data. This method exploits a manner in the function "fwrite" which retains the original format of data written. Allowing struct arrangement to emulate the header structure and be directly written to a waw file. When called a set of essential parameters will be passed to a function "generate_waw_file" and an empty file opened in write binary mode. The dynamic header values are calculated from the function parameters and are combined with statics into a struct which is then written to the file.

Header generation complete, the file can be filled with data. Samples are generated from a sine oscillator function which calculates and translates the samples into the correct format. The algorithm used in the oscillator is based on the synchronized swept sine method revised in [17]. This method prevents period drift in exponential sine sweep which if not adjusted can lead to incorrect phase estimation for High Harmonic frequency responses. An example of a non synchronized linear sine-sweep is pictured in 3.1

Figure 3.1: Non synchronized swept-sine signal[17]



The sine wave generator is heavily reliant on the "cmath" library as this provides sinusoidal and logarithmic functions. Sample format is dependent on the "bits per sample" defined by the wav header and are required to conform to the defined bit length. By default the samples are represented with a int16_t variable and cannot be changed without altering program code, locking the header value to 16. Finally samples are written individually to the wav file with stereo files having two writes per sample. All code responsible for the generation of impulses have been localized within a header "wawgen.h". This header will contain all necessary functions and structures needed to generate a full impulse.

3.3 Audio Player

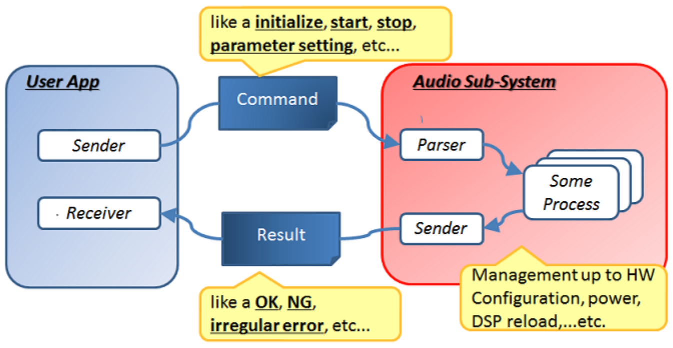
Playback is fundamental in impulse measurement as it is responsible for producing the impulse noise. The SDK environment provides a prebuilt audio subsystem with audio player functionality. Divided into a layer structure, the subsystem allows access to three partitions:

- Audio manager (High Level API)
- Object Layer (Object Level API)
- Component layer (Low Level API)

3.3.1 Audio manager

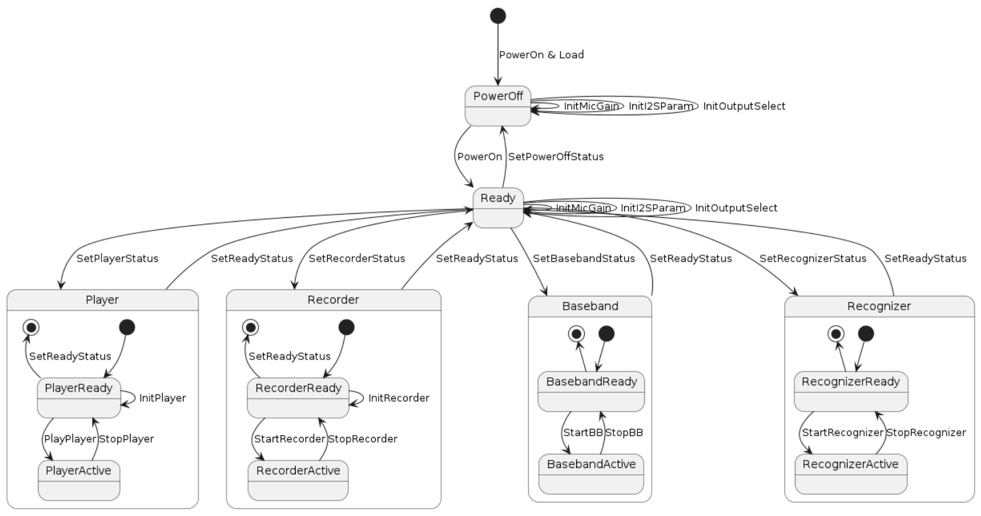
The audio manager is the high level access point to the audio subsystem and utilises a message passing interface to control system behavior. Pictured in 3.2 Commands are sent to the subsystem via "AS_SendAudioCommand" which carries the "AudioCommand" format. Each command passed invokes a response from the subsystem, this response carries the "AudioResult" format and can be collected with "AS_RecieveAudioResult". The command system is synchronous and will block all incoming commands until the result is returned, paring of send and receive commands is therefore recommended.

Figure 3.2: The command process of the High Level API[13]


like a initialize, start, stop, parameter setting, etc...
User App
Sender
Receiver
Command
Result
Audio Sub-System
Parser
Some Process
Sender
like a OK, NG, irregular error, etc...
Management up to HW Configuration, power, DSP reload,...etc.

Command format is a data structure which starts with a 4 byte command header. This header contains the size of the command, a command code identifying the command to be executed and a sub code specifying command target. Following the command header is a set of command parameters, these are dependent on the command code and not size limited allowing multiple inclusions per command. The return format retains a similar structure to the command with a header and return parameters.

Figure 3.3: The complete state machine for high level API control[13]

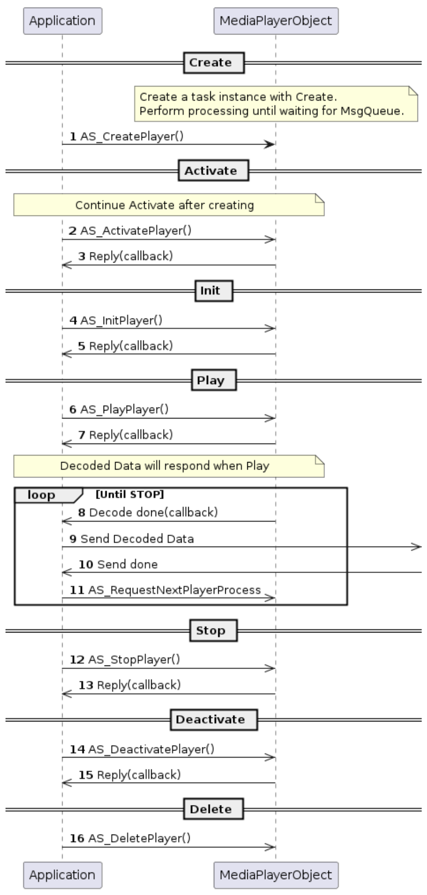


System behaviour is controlled by a state machine pictured in 3.3. State transitions are instigated by received commands which in turn transmits a handshake upon completed transition. The message system allow simple and intuitive interaction with the Audio subsystem. However, the lack of access to individual program functions combined with the handshake protocol limit application potential. Additionally the state machine makes the High level API unsuitable in constructing a impulse response system as it prohibits running audio recordings and playbacks synchronously.

3.3.2 Object Layer

Providing simpler functions, the object level API allows construction of more flexible applications than the high level API. Based on a "Media Player Object" controlling the playback, object level API operates by passing commands between the Object Player and the application. This interaction is done through a set of predefined functions passing parameters which through a more direct approach offers larger application possibilities. Function structure is similar to the High Level API with each command requiring a specific predefined parameter set following a target id. The target ID is required as the Object Level API allows the initialization of two separate "Media Player Objects" allowing dual playback. Out of scope as post processing is done off board the Object layer API features support for audio mixing through the "Output Mixer Object". The Mixer is initialized in the same manner as the "Media Player Object" and shares a similar command based behaviour model.

Figure 3.4: Example running of the MediaPlayerObject[13]



Example use case is figured in 3.3 and shows the handshake based running order of the Object level API. Operation begins with creation, activation and initialization of a player object which is done through respective functions. Following a completed starting process the player is controlled via a simple "start - stop - skip" interface and playback will continue until all audio files are performed or the stop call is received. The Object Level API allows direct access to the audio playback interface and thus foregoes the limitations of the state machine running order enforced by the High Level API. This will in theory allow parallel operation of the recording and playback objects, making an impulse response application possible.

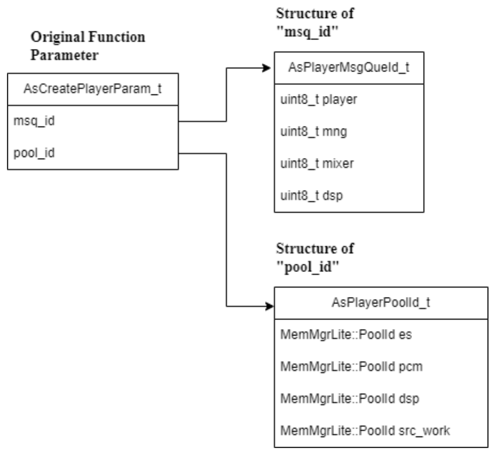
3.3.3 Component Layer

Warned by [13] the low level API is still under development and thus no official documentation is provided. With the limited time budget it was therefore decided that exploration of the Low Level API was to be dropped.

3.3.4 Final Player Function

Limits to the High Level API induced by the state machine architecture discussed in 3.3.1 and the Low Level API still being developed 3.3.3 Forces the final player application to be implemented via the Object level API. The control commands provided by the audio player libraries allow for easy control of the Media Player Object. However, the struct based parameters utilized by the control functions follow a hierarchical structure and require complex initialisation. Picured in 3.5 is the structure of "AsCreatePlayerParam_t" passed by "AS_CreatePlayerMulti" responsible for creating the Media Player Object. The struct is simple and organized but the "pool_id" substruct contain scope resolution operators to the "MemMgrLite" class. Being a part of the Memory Manager this class is responsible for allocating memory pools and returning identification. This requires a new set of functions and parameters that adhere to the memory manager libraries further complicating the parameter initialization.

Figure 3.5: Structure of the "AsCreatePlayerParam_t" parameter passed by "AS_CreatePlayerMulti" [18]



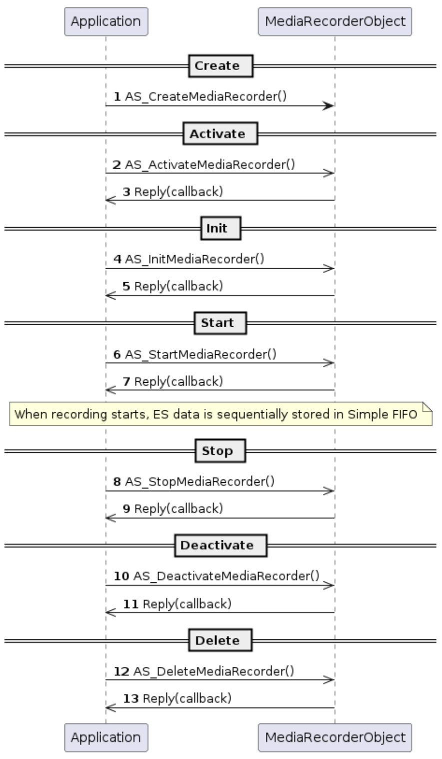
This snowballing effect is best visualized by the *Audio Player Objif* example code. Written by the same developers responsible for the Audio libraries the example consists of 1400 lines of code. The example can be split into two parts where the first 1000 lines define functions necessary in creating memory pools and correct function parameters. The final 400 lines describe application initialization, running order and error handling. This size and complexity makes construction of a custom program tailored to spec time consuming and difficult. However, as example code already exists it can be modified and converted to a header allowing direct access to all structures and functions defined in the example application.

Final audio playback implementation utilizes a modified copy of the audio player objif example. Modifications include porting to a header file and reconstruction of the main entry point to a standard function. Transformation of the entry point is required as the header extension identifies a non-executable library file. The result is an executable main file "audio_Player_Call" which creates and initializes all required parameters and objects required by the audio player library. Followed by the playback of audio files and eventual closing of the media player object upon completion. Although being a copy of example code the audio player provides simple playback of sound files and player control fulfilling the spec requirements.

3.4 Audio Recorder

Crucial in impulse response measurement the audio input is responsible for accumulating reactions triggered by the audio output. Specification described an audio recorder capable of 4 channel analog input and user controlled gain. Initially expected to be located in a separate subsystem the audio recorder library operates on the same layer structure as the audio player. Providing API access to three differing control partitions of different abstraction levels. Discussed in 3.3 restrictions in operation and incomplete work constrains choice to the Object level API.

Figure 3.6: Example running of the MediaRecorderObject[13]



Derived from the same subsystem and thus inheriting its features, the Audio Recorder utilizes an equal "media object" structure to the audio player. Example running order is portrayed in 3.6 and contains equal elements to the media player object 3.4. This similarity allows operation to be plagued by equal complexity issues to the Audio playback. However, this resemblance grants the possibility of implementing a similar solution to that implemented in3.3.4.

Final recorder implementation utilizes a modified header implementation of the "audio recorder objif" example. The main entry point has been converted to a function allowing calling of the example running order. Including complex system initialisation and a full recording cycle the example in this configuration provides sufficient recording functionality. Reasoning behind such an implementation has previously been covered in 3.3.2. Bearing similar structure with a length of 1200 lines the recorder is far too complex to construct from scratch given the time budget. As the example provides all necessary features a custom implementation is thus not required

Chapter 4

Results

Development on any new embedded platform is guaranteed to include a diverse set of challenges. The spresense is not an unfamiliar platform as previous work had been performed on the hardware 1.2. This work was preparatory and conducted using the Aurduino IDE which allows for quick development through a simplified interface. However, Arduino operates in an unconventional manner compared to traditional embedded environments making it unsuited for larger applications. For larger development the SDK environment is preferred as this more closely resembles traditional embedded development environments. This chapter will present the experience constructing an impulse application using SDK including the resulting performance of its individual parts.

4.1 Serial

Originally designed for the Nuttx based Robotic control system Micro-ROS 3.1 the serial interface has performed very well. Initially conceived as non essential, the serial communication has provided an invaluable flexibility to application structure and debugging. The command based state machine forms the backbone of system running order and allows quick command list modification via if statement. Debugging and verification can thus be done by formulating multiple variations of the afflicted application section and creating unique command calls for each. Similarly complex program parts can be split up and called individually allowing clear examination of running order and issues. Both techniques was actively used in the debugging and construction of the sine oscillator significantly speeding up verification. However, the construction of the serial communication protocol in addition to compiler restrictions hinder full functionality. Mentioned in 3.1 the serial buffer is filled by using a busy-wait protocol on the UART. Achieved by constantly reading the UART buffer and checking for new characters busy-waiting ensures that no passed parameter is lost. This continuous read is resource heavy and in the case of the Spresense blocks some operations while in effect. As the original serial protocol provides no direct write-back and support for buffer modification the User Experience was suboptimal. Direct write-back would allow the user to monitor their own inputs similar to common terminal or text editors. This would allow review of the UART buffer before sending improving UX, and was intended to coincide with buffer modification connected to the backspace allowing full buffer control.

Code listing 4.1: Attempts at implementing direct printing and buffer modification to the serial protocol

```
printf("%c", buffer);   //print buffer contents

if (buffer == 127) {    //char is backspace
  buffer_aux[i] = NULL;
  buffer_aux[i-1] = NULL;
  i--;
}
```

Attempted improvements to the serial protocols are figured in 4.1 and would be implemented between the auxiliary buffer saving and return check. Having been verified in isolated environments these implementations are hindered by the busy waiting polling of the UART. This issue can be resolved by rewriting the serial protocol to be less intrusive such as via interrupts. However, as the serial monitor is fully functional the benefit of rewriting a protocol for convenience is not logical. As this would revert resources from more crucial implementations, wasting an already strained time budget. Likewise the issues regarding lack of support for strings as variables was ignored as tools where implemented to make the issue acceptable. The root of the issue is unknown but is likely caused by absence of support from the compiler or Nuttx.

The inclusion of a serial monitor has greatly benefited the application performance and efficiency. Although not perfect the command based structure allows for flexible application construction and verification. Had the Serial communication non been implemented early the project, time span would be increased significantly and many features would not have been implemented.

4.2 sine wave generation

Initially believed to be supported as the Spresense is advertised towards acoustic and media processing. The hardware generation of impulse response signals was conceived as an easy specification. However, no official documentation hints towards such an toolset and after becoming intimately familiar with the Spresense throughout development, it can be concluded that such a library does not exist. The lack of official toolsets complicate module construction as a Wav file will have to be generated from scratch. Although complicated, several external resources for waveform generators exists. Created by a course team on The Univeristy of Notre Dame. The "wavfile" library [16] describes a digital waveform toolset that allows simple generation and storage of digital sound. Fulfilling all spec requirements and being based in C original intention was to create an exponential sine oscillator extension and utilize the prebuilt library. However, as the library was incompatible with the Spresense a custom extension with inspiration from the "wawfile" library was created.

Initial errors where largely related to formatting of the Wav file as header and sample data would cause file corruptions if not defined correctly. Issues where caused by two 4 byte segments defining total file and data size. Located in the header and thus written before the samples these caused initial issues as file size is difficult to define before completion. The waveform toolset solved this issue with a double pass that utilized "fseek" to find the end pointer allowing easy calculation of file and data size. Theoretically effective the direct implementation resulted in file corruption caused by improper header parameters. Instead a more direct approach was chosen, as the header size is 44 bytes and sample size can be calculated from existing parameters. File size can be found by calculating total data size and adding 44 bytes.

Additional Issues where caused by the procedure of writing samples to the file. Samples need to be written in a binary 16-bit format and was initially passed via "fprintf". Designed towards formatted printing "fprint" will ignore protocols defined during file opening and will default towards the passed parameters format. This will cause corruption for non binary variables but is fixed by utilizing "fwrite" as a substitue. Generation speed of the wav file is a non critical issue but can cause issues for larger processes. During verification it was noted that the system performance for sine wave generation was below expectations. Testing concluded that a 1 second signal with a SR of 48000 would average at 15 seconds. Initially unknown if this issue was caused by writing or processing it was theorized that the individual writing protocol caused a bottleneck. Later verification confirmed the sine oscillator as the issue. This would require a rewrite of the calculation protocols such as the implementation of Look Up Tables. However, as the bottleneck is non critical and discovery was done late in development an implementation was not attempted.

Development time of the exponential impulse generator was longer than initially suspected. Despite the presence of a library capable of exact spec several modifications had to be implemented and tested. Expected as the Spresense is an embedded platform the total time committed was far larger than initially anticipated. Lengthy development time has nevertheless resulted in a robust system that fulfils the goals set by the spec. The library is however plagued by slow performance stemming from poor calculation protocols. Requiring an extensive restructure not allowed by the current time budget this implementation is strongly recommended for future developments.

4.3 Audio Player

Initial evaluations of audio support for the spresence was conceived as impressive. Documentation presented an intuitive library, containing multiple API options of differing complexity offering versatile application construction. Supporting multiple API abstraction levels the Object Level was deemed most suitable. This decision was made following documentation studies concluding with the Object level as the least restrictive option. Despite early impressions the audio libraries place certain restrictions on running order and operation, limiting total utility. This discovery was surprising as the Spresense is heavily advertised towards audio applications. The absence of certain features, namely the low level API is excusable as the platform is still in a developmental phase. However, several core design decisions restrict application flow to such an extent that their inclusions are questionable.

Specification describes an audio output capable of playing a choice of differing audio tracks with low latency. Despite being uncomplicated the audio library core structure embroils this task to an unnecessary extent. Official documentation alludes to an alternative setup allowing playback of individual sound files. This setup procedure is neither provided in example code or official documentation and is thus non-replicable. As instructions are not provided operation is forced to the playlist centered default provided in examples. Working upon an predefined CSV list containing track databases this mode is targeted towards construction of a conventional audio player. Dynamic playback of single tracks during running order therefore requires a program rewrite of the playlist. An unnecessary complication of a already large system.

Final implementation utilized a modified version of the existing object layer example code. Discussed in 3.3 discovery of the intricate object setup caused this implementation. Although initially conceived as substandard, continued research of the core design revealed a functional audio player given its environment. Initial appearance of the object layer API showed promise. As initial work consisted of grasping the convoluted system structure, little attention was put towards the control API. It was assumed that the API would allow a limited interaction similar to a standard audio control interface *(play, pause, skip)*. Further studies revealed the absence of any control interface. Player operation would therefore be reliant on the playlist stopping only if the stop command was issued or the playlist was completed. Justifying the existence of a playlist based operation protocol, this affirms initial suspicions that the audio library in its current state is suited for simple playback applications. This may change upon the completion and introduction of the Low level API or an update of the existing media object structure. However, if the current core design is not changed these implementations will fail to grant significant improvements as the current structure is significantly flawed. Although unnecessarily complicated, the audio player works and will provide a viable platform of impulse playback.

4.4 Audio Recorder

Caused by the extended development time of audio playback features, work on audio input was commenced late in the project lifespan. Initial concerns where raised by potential results similar to the complex playback subsystem. Although at this point familiar with the Spresense libraries and file structure the temporal budget did not allow an implementation of equal complexity to the audio player. Relief was thus shown upon the discovery that audio recording was located within the same audio subsystem as the player. This would however denote that the recorder was plagued by the same inhibiting restrictions as the playback.

Discussed in 4.3 the audio subsystem central structure inhibit design flexibility. The Media Player Object was plagued by a forced reliance on playlist based scheduling and a limited command set. Under similar context, the Recorder Object is less affected by subsystem restrictions as its required workload is smaller. Recorder task load typically only require resources for starting and stopping. As both are present in the object command set the construction of additional support systems are not required. This is in stark contrast to the Playback Object which required several support implementations in order to fulfil expected operation.

Final implementation can thus be constructed using a modified copy of the "Audio Recorder Objif" example. This implementation is performed in the same manner as the Aduio Playback with a header file conversion and redefinition of the entry point into a standard function. The example provides complete functionality and requires no additional custom tools. Although a significant improvement compared to the Player Object the Recorder still suffers from significant set-up time. Discussed in 4.3 this is caused by the core design of the audio subsystem and will not be resolved unless a object level redesign is performed. The completion and release of the low level environment has the potential to relieve some core issues but as of writing no information about the development or structure is available.

Chapter 5

Discussion

5.1 Documentation

The development experience for the Spresense SDK environment has been unsatisfactory in several aspects. Expected support tools such as documentation and forums have delivered subpar resources forcing a trial and error approach to setup and programming. This lack of guidance has been one crucial factor in the projects failure to achieve spec as budgeted time was wasted on forced system deconstruction. Absence of community documentation further complicates this issue. At the time of writing few documented Spresense SDK projects or forums transcending the Developer World website exist symbolizing a larger issue of failure to reach consumer appeal.

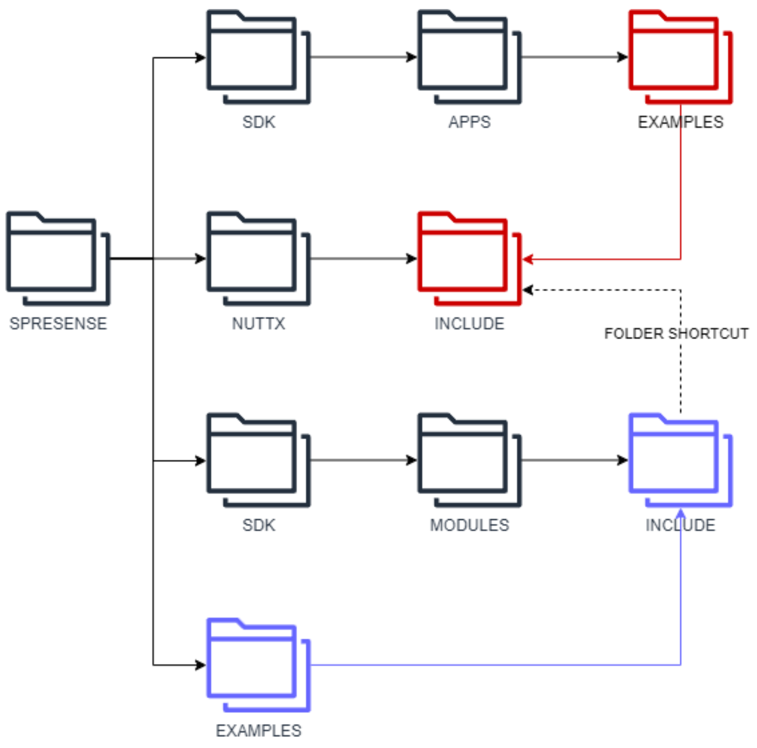
Early project setbacks where attributed in failure to embrace the developer world forum. Fully moderated, it provides a valuable resource through direct contact and guidance with Spresense developers. Restrictive in its lack of search functionality and search engine indexing, the connection to Sony staff negates the shortcomings of official documentation. Further, the project has through involvement in the forum directly influenced Spresense development by notifying issues plaguing the Audio Player. Updates to official documentation regarding SDK on WSL have also been observed following a forum inquiry over issues plaguing the platform. Inability to establish a dedicated user base is further illustrated with low forum contributions and a predominance of Arduino related topics.

As of the time of writing the Spresense reaches its 5th year on the commercial market. Failure to achieve mainstream appeal is apparent through minuscule community activity and absence of projects exploiting full hardware potential. Documentation is subpar but as observed receives updates at slow but regular intervals. Considering its persistent low popularity and slow development the risk of abandonment is realistic and necessitates evaluation for future projects.

5.2 System

The modified Nuttx operating system has performed beyond all initial expectations. Flexibility and intuitive design has allowed effective prototyping and construction throughout the entire development process. Modified in a manner that retains OS core structure allows Nuttx specific resources to be used, simplifying comprehension and easing construction process. Extensive configuration allows the inclusion and exclusion of all non essential features, permitting builds optimized towards certain hardware or applications. However, the default configuration excludes all non essentials requiring reconfiguration for each new application flashed 2.1.1. Nuttx attempts to negate this by including an index based storage system for configurations. This systems failure was observed several times throughout early development and was not relied on further. Nuttx documentation hints towards configuration files included with applications, such a system would streamline configuration but a limited time budget hindered further exploration.

Figure 5.1: The example folder structure with colour indexed library connections, the shortcut fix is portrayed through the dotted connection



Success of the SDK developer environment can be attributed to the inclusion of Nuttx. The modifications performed by Sony are of such a poor standard that it impairs the OS and caused several substantial setbacks in the early phase of the project. Nuttx is directly connected to the underlying file structure of the environment. As such poor file management directly affects the Nuttx user experience. Initial issues where faced upon the construction of a custom application. Documentation clearly describes the process of adding an custom application which is required to be located within the example folder. However, documentation fails to declare the fact that there are two example folders located within the hierarchy 5.1. Command line based application selection which in documentation is used to select and flash the examples only allows access to the lowest example folder. This results in new users creating an custom application in the topmost folder resulting in Nuttx failing to locate the file. Additionally the differing example folders are granted access to separate libraries. This issue was faces by the project when attempting to port the Audio Player to the lower example folder and was solved by locating the library files available to each example location and creating folder shortcuts linking them together. As the location of both libraries were not referenced and unclear, the paths had to be reverse engineered using error messages and directory scavenging. The success of this solution is particularly strange as it displays that separation of examples is not based upon dependencies which indicates it as unnecessary.

Discussed in 2.1.1 Nuttx compilation is optimized towards Linux based operating systems. Although initial attempts where conducted on Windows, performance was of such poor nature that a switch was required to achieve effective development. Later tests verified the performance difference as tenfold, resulting in 5-10 minute compilations for larger applications. Initially WSL was chosen as it allows the creation of a Linux subsystem within Windows, voiding the need for dual boot and virtual machines. Documentation had referenced support for WSL but described uncertainties as the system had not been verified. Implementation was attempted using Win10 WSL2 but failed during the setup, later verification by Spresense forum moderation verified that Win11 WSL2 functions properly whilst Win10 fails due to unknown errors. This issue has been rectified in an update to the documentation. As attempts with WSL2 failed, Ubuntu running in an VirtualBox VM was decided as an suitable alternative. Although the projects second option, operation through a desktop allow easier maneuvering and thus better grasp of the convoluted file structure discussed previously. Design and operation through a dual boot or virtual machine is thus deemed as the preferred option for new developers to the SDK environment.

5.3 Features

Spresense SDK documentation presents a feature rich environment with library support for all advertised applications. The reliance on quality audio libraries was crucial for project success as the budgeted time did not allow for development of custom libraries. Initial study of the object oriented API was promising and it was believed that spec would be achieved utilizing the media object libraries. However as discussed in 4.3 and 4.4 the core hierarchy of the libraries are structured in a manner that disincentives modification. The fundamental issue is the function control parameters reliance on resources defined by separate libraries. Creating dependencies not declared by documentation and forcing a staggering initialization process spanning multiple libraries. The consequences of this hierarchy is best observed through the playback and recording examples respective 1000 lines of initialization code. Several improvements could be implemented to alleviate the dependency issues. The inclusion of setup functions for the respective libraries would reduce dependencies and lessen code size improving comprehension. However, improvements would only mediate issues as a complete reconstruction is require should the dependency issues be rectified.

Control functions and operation of the Audio player clearly demonstrate a failure by developers to recognize its utilization case. Current design limit playback to a playlist based system, lacking the capability to select and play individual files. The decision to force playlist based operation is perplexing as this pattern combined with the absence of a traditional playback interface force an execution pattern reminiscent of a tape deck. Consequently, the resulting library is optimized towards the construction of a inferior audio player and fiercely complicate the development of other applications. Although unique to the audio player library the failure to create a feature embracing the possibilities granted by hardware is a recurring theme and one of the core issues faced by the current development of the Spresense.

5.4 Application

Final application construction does not accomplish the requirements set by the spec. Although disappointing, the resulting information gathered through attempted construction has greatly improved the knowledge about the SDK environment. The constructed application provides an excellent foundation for future developments but improvements can still be made. Current running order revolves around the serial state machine which currently does not allow backspace. Several attempts where made to implement buffer manipulation using similar logic to the return trigger. The efforts where unsuccessful as the active polling blocks operation, thus requiring a complete rewrite of the serial protocol to be less intrusive. The sine wave generator required extensive development time as failure to properly configure the header would cause file corruption. This lengthy development has resulted in a very robust system. But as verification exceeded its time budget the exponential sine wave generator has not been verified to the same standard. Although theoretically functional the lack of proper testing disallows identifying the generator as fully functional. Should the exponential sine wave generator be deemed as defective the repair process is simplified through a modular system structure. As discussed in 3.3 the playback and recording is constructed through modification of the example applications. Initially a consequence of poor design, the limitations of the framework result in the examples fully exploiting library potential. As such the consequences of utilizing modified audio examples are limited. Excluding the use of pre-existing examples, modular application construction was primary when designing the application. Modular design allows the reuse of all project assets for future developments and was largely motivated by the inferior versatility of the Audio framework provided by the Spresense. The final application provides a solid foundation for future efforts both in program code and documentation of Spresense shortcomings.

Chapter 6

Conclusion

This thesis has further explored use of the Sony Spresense as an impulse response measurement platform. Through attempted construction utilizing an unknown environment further knowledge about the Spresense has been obtained with the goal of aiding subsequent projects on the platform. This newfound understanding of how the Spresense performs under all environments will allow future work to comprehend the current issues plaguing the system and prevent overestimation of system capabilities.

Official development of the Spresense is still incomplete 5 years after its introduction to the commercial market. The completed frameworks, namely the audio libraries are of poor design and disallow the extraction of full hardware potential. As such many completed Spresense libraries will require redesigns should the issues be alleviated, furthering its status as incomplete.

The failure to establish a dedicated community is evident through the lack of forum activity and documented projects on the platform. Symbolizing a commercial failure, this hurts the Spresense as community created resources allow for quicker innovations through shared libraries and guides diminishing the overall potential of the microcontroller platform.

A foundation for future works on the platform has been constructed. Designed as a modular system this foundation will provide a valuable resource for subsequent attempts. Several modules of the system are also generic allowing porting to other microcontrollers should future works move away from the Spresense.

For conclusion, the Spresense demonstrates great potential but suffers due to poor framework and failure to establish a dedicated community. Should the platform continue development under the acoustics department several critical issues require addressing. Largely related to the current Spresense SDK construction the issues are only rectified by a rewrite of the entire framework. As such the move away from the Spresense to a different platform should be considered.

Bibliography

[1] A. C. Worsøe, 'Insect flight sound acquisition system with eight microphones,' *Master's thesis in Electronics Systems Design and Innovation NTNU*, vol. 1, pp. 1–79, 2022.

[2] K. Øverli, 'Acquisition system with recording of audio and images for hymenoptera,' *Master's thesis in Electronics Systems Design and Innovation NTNU*, vol. 1, pp. 1–85, 2022.

[3] R. Avila. 'Embedded software programming languages: Pros, cons, and comparisons of popular languages.' (2022), [Online]. Available: `https://www.qt.io/embedded-development-talk/embedded-software-programming-languages-pros-cons-and-comparisons-of-popular-languages` (visited on 20/05/2023).

[4] Arduino. 'Arduino portenta h7.' (2022), [Online]. Available: `https://docs.arduino.cc/hardware/portenta-h7` (visited on 19/12/2022).

[5] T. A. S. Foundation. 'About apache nuttx.' (2020), [Online]. Available: `https://nuttx.apache.org/docs/latest/introduction/about.html` (visited on 02/06/2023).

[6] T. A. S. Foundation. 'Development environments.' (2020), [Online]. Available: `https://nuttx.apache.org/docs/latest/introduction/development_environments.html` (visited on 23/05/2023).

[7] S. Groot. 'Major performance (i/o?) issue in /mnt/* and in (home.' (2018), [Online]. Available: `https://github.com/Microsoft/WSL/issues/873#issuecomment-425272829` (visited on 02/06/2023).

[8] P. software foundation. 'Python for beginners.' (2022), [Online]. Available: `https://www.python.org/about/gettingstarted/` (visited on 19/12/2022).

[9] P. software foundation. 'The python package index.' (2022), [Online]. Available: `https://pypi.org/` (visited on 19/12/2022).

[10] D. Anh. 'Dsp-py-lib.' (2020), [Online]. Available: `https://github.com/dinhanhx/DSP-py-lib` (visited on 02/06/2023).

[11] D. George. 'Micropython.' (2022), [Online]. Available: `https://micropython.org/` (visited on 19/12/2022).

[12] CricuitPython. 'Micropython.' (2022), [Online]. Available: `https://circuitpython.org/` (visited on 19/12/2022).

[13] S. Corp. 'Spresense sdk developer guide.' (2023), [Online]. Available: `https://developer.sony.com/develop/spresense/docs/sdk_developer_guide_en.html#_audio_subsystem` (visited on 04/05/2023).

[14] Sony. 'Introduction - spresense hardware.' (2023), [Online]. Available: `https://developer.sony.com/develop/spresense/docs/introduction_en.html#_spresense_hardware` (visited on 27/05/2023).

[15] jfm92. 'How to use serial comunication in your app.' (2018), [Online]. Available: `https://github.com/micro-ROS/NuttX/issues/10` (visited on 02/05/2023).

[16] dthain. 'Wavfile: A simple sound library.' (2013), [Online]. Available: `https://www3.nd.edu/~dthain/courses/cse20211/fall2013/wavfile/` (visited on 03/05/2023).

[17] S. Novak Lotton, 'Synchronized swept-sine: Theory, application and implementation,' *Journal of the Audio Engineering Society*, vol. 63, no. 10, p. 786, 2015.

[18] C. A. S. Team. 'Audio player api.h file reference.' (2023), [Online]. Available: `https://developer.sony.com/develop/spresense/developer-tools/api-reference/api-references-spresense-sdk/audio__player__api_8h.html` (visited on 09/05/2023).

Appendix A

Spec

Topic	Functionality	Priority
Audio output	On-board generation of exponential sine sweeps[1]	mandatory
Audio output	Generated sine sweeps must start and stop at zero crossing [1]	mandatory
Audio output	Parameters for on-board generation: fmin, fmax, duration (s), repetitions, delay between repetitions (s) [1]	mandatory
Audio output	Playback of audio files prepared on a host computer	mandatory
Audio output	Choice between several files for playback	optional
Audio output	Single channel	mandatory
Audio output	Dual channel	optional
Audio output	Playback only mod[2]	optional
Audio input	Channel count of 4	mandatory
Audio input	Analog inputs	mandatory
Audio input	User controllable input gain (if allowed)	mandatory
Audio input	Toggelable microphone power	optional
Audio input	Record only mode[2]	optional
FDxS	Scheduling is done through standalone config file	mandatory
FDxS	Support for scheduling relating to sunrise to sunset	optional
FDxS	Use of GNSS to get latitude/longitude	optional
FDxS	Support for continuous recording	mandatory
FDxS	Support for recording according to a duty cycle	mandatory
FDxS	Way to turn a configuration file to a timeline for verification	optional
Synchronization	GNSS time stamp	optional
Recorded audio	Format: WAV	mandatory
Recorded audio	Recording time(start) and date	mandatory
Recorded audio	Location is user-defined	mandatory
Recorded audio	Location is based on GNSS signal	optional
Hardware	Power source: Li-ion 18650 batteries in parallel	optional
Hardware	Preamplifier: 40dB gain on/off	optional
Hardware	High-pass filter: 20 Hz on/off	optional
Hardware	Sensor supply: constant current 4 mA on/off	optional
Hardware	Waterproof casing	optional
Hardware	Internal humidity sensor	optional
Synchronization	Functionality	optional