**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF COMPUTER SYSTEMS**
**ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ**


# ADAPTATION OF ZEPHYR RTOS AND SOUND OPEN FIRMWARE ON A HIFI4 DSP OF THE NXP I.MX RT685 MCU

**ADAPTACE RTOS ZEPHYR A SOUND OPEN FIRMWARE NA HIFI4 DSP ČIPU NXP I.MX RT685**

**BACHELOR'S THESIS**
**BAKALÁŘSKÁ PRÁCE**


**AUTHOR** **VÍT STANIČEK**
**AUTOR PRÁCE**

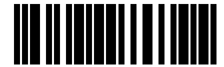
**SUPERVISOR** **Ing. VÁCLAV ŠIMEK**
**VEDOUCÍ PRÁCE**



**BRNO 2024**

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

# Bachelor's Thesis Assignment



156783

| | |
|---|---|
| Institut: | Department of Computer Systems (DCSY) |
| Student: | **Staniček Vít** |
| Programme: | Information Technology |
| Title: | **Adaptation of Zephyr RTOS and Sound Open Firmware on a HiFi4 DSP of the NXP i.MX RT685 MCU** |
| Category: | Operating Systems |
| Academic year: | 2023/24 |

Assignment:

1. Analyze and describe the NXP i.MX RT685 microcontroller, along with its subsystems. Propose suitable use cases for this device.
2. Familiarize yourself with the Zephyr RTOS and its distinguished features.
3. Familiarize yourself with the Sound Open Firmware project, describe its purpose and features. Compare it against the Maestro and Xtensa Audio Framework software layers.
4. Port the Zephyr RTOS to the Cadence HiFi4 DSP core on the mentioned microcontroller and test basic functions.
5. Port the Sound Open Firmware project and implement a suitable inter-core communication mechanism.
6. Create an application demonstrating audio processing flow using implemented solutions from points 4) and 5).
7. Evaluate the results and propose possible ways to continue this project.

Literature:

- According to the instructions of the supervisor.

Requirements for the semestral defence:
Fulfillment of points 1 to 4 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

| | |
|---|---|
| Supervisor: | **Šimek Václav, Ing.** |
| Head of Department: | Sekanina Lukáš, prof. Ing., Ph.D. |
| Beginning of work: | 1.11.2023 |
| Submission deadline: | 9.5.2024 |
| Approval date: | 30.10.2023 |

## Abstract

This thesis analyzes the specifics of the i.MX RT685 microcontroller made by NXP Semiconductors, the Cadence Tensilica HiFi 4 DSP processing core implemented in it and describes an effort of porting the Zephyr RTOS along with Sound Open Firmware to that platform. An application demonstrating an audio processing pipeline on those technologies is implemented in this work as well. The thesis is motivated by the aim to both efficiently implement digital audio processing techniques and streamline their use on the said device.



## Abstrakt

Tato práce se zabývá analýzou mikrokontroléru i.MX RT685 vyráběného firmou NXP Semiconductors, v něm realizovaného Tensilica HiFi 4 DSP jádra a popisuje úsilí adaptace RTOS Zephyr společně s vrstvou Sound Open Firmware na tuto platformu. Aplikace demonstrující zpracování digitálních zvukových signálů nad těmito technologiemi je v této práci implementována rovněž. Práce je motivována záměrem jak efektivně realizovat techniky pro digitální zpracování zvukových signálů na této platformě, tak sjednotit jejich použití.



## Keywords

microcontroller, embeddded system, digital signal processing, operating system

## Klíčová slova

mikrokontrolér, vestavený systém, signál, digitální zpracování signálů, operační systém


## Reference


STANIČEK, Vít. *Adaptation of Zephyr RTOS and Sound Open Firmware on a HiFi4 DSP of the NXP i.MX RT685 MCU*. Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Václav Šimek

# Rozšířený abstrakt

Jedním z přístupů k realizaci řídídích úloh současných mikroelektronických zařízení či rovnou k realizaci celé jejich funkce je tvorba tzv. vestavěného systému, tedy systému, který je řízen integrovaným obvodem obsahujícím jedno či více mikroprocesorových jader. Takový přístup přesouvá značnou část charakteru zařízení z hardwarové domény do domény softwarové. Tento charakter lze v průběhu celého životního cyklu zařízení měnit, narozdíl od návrhových rozhodnutí učiněných v hardwaru, která lze měnit jen nahrazením celého zařízení či relevantní fyzické součásti za kus vyrobeného dle aktualizovaného návrhu. Na vývoj softwaru pro taková zařízení lze rovněž aplikovat přístupy softwarového inženýrství používané v jiných odvětvích informačních technologií (návrh, dekompozice, projektové řízení, statická analýza, ...), které systematizují tvorbu tohoto specializovaného programového vybavení a umožňují tvořit produkty, které lépe plní zadání, jsou otestované a spolehlivější.

Mezi takové obvody patří mikrokontroléry. Ty se obzvlášť hodí pro realizaci vestavěných systémů se značnými návrhovými omezeními (rozměry, prodleva při uvedení do chodu, spotřeba elektrické energie, ...), protože nejsou tak sofistikované jako běžné mikroprocesory (menší kapacita pamětí, pracovní frekvence, ...) a zároveň integrují vše potřebné v jednom pouzdře - vstupně-výstupní periferie, komunikační rozhraní, regulátory napětí, apod. V současné době již nejde o součástky určené výhradně na jednoduché řídící aplikace sestávající se z obsluhy jednoduchých součástí (LED diod, tlačítek, segmentových displejů, jednoduché mechatroniky apod.), moderní mikrokontroléry poskytují celou řadu komplexních periferií a výpočetních jader a zároveň zachovávají řečené atributy. Tyto bloky existující vedle hlavního procesorového jádra umožňují konstrukci zařízení se zajímavými funkcemi, vysokou propustností, pokročilou konektivitou s vnějším prostředím a o to více sníženou spotřebou elektrické energie.

Jedním z těchto výpočetních jader je DSP jádro Tensilica Xtensa HiFi 4 od firmy Cadence Design Systems. Toto jádro představuje blok procesoru uzpůsobeného pro zpracování digitálních signálů - je optimalizováno na úlohy jakými jsou spektrální analýza, filtrování, detekce příznaků, ztrátová komprese, dekódování komprimovaných proudů apod. Představuje to prostředek pro efektivnější implementaci řečených úloh než procesorová jádra uzpůsobená na obecná použití (Arm Cortex-M, i8051, AVR, PIC, MSP430, ...), čímž jejich využití může pomoci při plnění řečených návrhových omezení.

Řečené jádro Tensilica Xtensa HiFi 4 se, přinejmenším v případě součástek od firmy NXP Semiconductors, nevyskytuje jako jediné procesorové jádro ale v kontextu těchto součástek funguje jako dodatečný, specializovaný procesor. Tímto se otevírají možnosti rozdělení odpovědností - řečené DSP lze výlučně naprogramovat pro zpracování digitálních signálů a zbytek funkce zařízení (komunikace s okolím, uživatelské rozhraní, řízení napájení, ...) lze ponechat na hlavním aplikačním procesoru. Dynamicky konfigurovaným modulárním softwarovým vybavením pro řečený DSP blok lze vytvořit prostředí s funkčními bloky, jejichž konfiguraci a topologii lze z pohledu aplikačního procesoru dynamicky určovat. Toto by zcela abstrahovalo problémy spojené se zpracováním digitálních signálů a v zásadě předepisuje obecný, opětovně využitelný rámec (framework) pro tyto úlohy.

Programování těchto jader však vyžaduje dodatečné úsilí. Přinejmenším je potřeba odpovědnosti řešené realizovaným systémem vhodně rozdělit, zajistit synchronizaci a abstrahované komunikační kanály. Řešení úloh citlivých na časování implikuje nutnost zvláštní péče při realizaci a využití synchronizačních mechanismů. Programování DSP jader, vzhledem k jejich specializovanosti, rovněž vyžaduje jim uzpůsobené nástroje (překladač, linker) a dodatečné kroky optimalizace kódu, aby bylo možné využít speciální instrukce, které zpravidla tato jádra poskytují.

Motivací této práce je zjednodušit realizaci aplikací využívajících toto DSP jádro pro zpracování zvukových signálů a abstrahovat jej do opětovně využitelné podoby. Tímto problémem se zabývá v kontextu mikrokontroléru i.MX RT685 vyráběného firmou NXP Semiconductors.

# Adaptation of Zephyr RTOS and Sound Open Firmware on a HiFi4 DSP of the NXP i.MX RT685 MCU

## Declaration

I hereby declare that this Bachelor's thesis was done as an original work by the author under the supervision of Ing. Václav Šimek while consulting colleagues working at NXP Semiconductors. I have listed all the literary sources, publications and other sources, which were used in this thesis.

. . . . . . . . . . . . . . . . . . . . . . .
Vít Staniček
May 7, 2024

## Acknowledgments

First and foremost, I would like to sincerely thank my colleagues at NXP Semiconductors who endured my endless stream of beginner questions and provided me with valuable insight, which has helped immensely both in developing my professional skills and in writing this thesis. This thesis couldn't be written without the technical insight of Daniel Baluta and Iuliana Prodan. The list also includes my direct colleagues - Ing. Tomáš Bařák, Ing. Petr Lukáš, Ing. David Jurajda, Ing. Stanislav Pobořil, Ing. Ivo Solanský, Ing. Pavel Kovář and Ing. Samuel Mudrík, all of them helped me tremendously during solving other tasks at this company and even with getting myself familiar with programming microcontrollers.

I would also like to thank Ing. Václav Šimek who supervised this thesis and fellow students, who helped me with distracting myself from mundane duties throughout my time at this university. At last, a very special thanks goes to my parents and my grandparents, who raised me with love, care and were with me every step of the way throughout my life.

# Contents


**1 Introduction** **4**

**2 Commonalities of microcontrollers** **5**
2.1 Definition and brief history . . . 5
2.2 Common interfaces . . . 5
2.3 Programmer's model . . . 9
2.4 DMA . . . 11

**3 The i.MX RT685 microcontroller** **12**
3.1 Overview . . . 12
3.2 Memory architecture, interrupt handling . . . 12
3.3 DMA . . . 13
3.4 Inter-core communication . . . 14
3.5 Evaluation boards . . . 15
3.6 Possible use cases . . . 17

**4 The Zephyr RTOS for Embedded Domain** **18**
4.1 Overview . . . 18
4.2 Configuration . . . 19
4.3 Development toolset . . . 19
4.4 Driver model . . . 20
4.5 Device trees . . . 21
4.6 Audio driver APIs . . . 23
4.7 Comparison with other systems . . . 24

**5 The HiFi 4 DSP core** **26**
5.1 Overview . . . 26
5.2 Instruction set . . . 26
5.3 ABI . . . 27
5.4 Available tools . . . 28

**6 The Sound Open Firmware layer** **29**
6.1 Overview . . . 29
6.2 Programmer's model . . . 29
6.3 Architecture . . . 30
6.4 Comparison with similar layers . . . 31

**7 Porting the Zephyr RTOS** **33**

7.1 First steps . . . 33
7.2 Project structure . . . 34
7.3 DSP code loading and starting . . . 34
7.4 Enabling basic drivers . . . 35
7.5 Interrupt handling . . . 35
7.6 Testing and debugging . . . 35
7.7 GCC toolchain porting . . . 36
7.8 Audio input and output . . . 37
7.9 Basic IPC . . . 38
7.10 Memory layout . . . 39

**8 Porting the Sound Open Firmware layer** **40**
8.1 First steps . . . 40
8.2 IPC . . . 41
8.3 Audio data exchange between domains . . . 42
8.4 Audio input and output . . . 43

**9 Test application** **44**
9.1 Overview . . . 44
9.2 Memory usage . . . 45

**10 Conclusion** **46**

**Bibliography** **48**

**A Optical media contents** **50**

**B The Zephyr RTOS port** **51**

**C Sound Open Firmware port** **53**

# List of Figures

2.1 Timing diagram of an 8-N-1 transmission . . . . . . . . . . . . . . . . . . . . 6
2.2 Timing diagram of a simple I$^2$C transaction . . . . . . . . . . . . . . . . . . . . 7
2.3 Timing diagram of a simple SPI transaction . . . . . . . . . . . . . . . . . . . . 8
2.4 Timing diagram of a classic I$^2$S frame bearing 16-bit stereo data . . . . . . 9

3.1 Photo of the MIMXRT685-EVK evaluation board . . . . . . . . . . . . . . . . 15
3.2 Photo of the MIMXRT685-AUD-EVK evaluation board . . . . . . . . . . . 16

4.1 Interactive configuration tools . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

# Chapter 1

# Introduction

Microcontrollers, highly integrated computers suited for use in embedded use cases, now form the basis of today's microelectronics. As time progressed, they evolved from simple microprocessor cores with primitive I/O peripherals made for simple controlling applications (utilising LED diodes, buttons, segmented display devices, simple mechatronics) to complex devices containing sophisticated peripherals and acceleration blocks. These blocks can help implement more sophisticated devices while, at the same time, meeting design constraints typical for compact embedded systems, such as physical dimensions, power consumption, cold start latency and, last but not least, cost.

One of those blocks is the Xtensa Tensilica HiFi 4 DSP from Cadence Design Systems. This IP core implements a microprocessor specialised towards the processing of digital signals - it's optimised for tasks like spectrum analysis, filtering, feature detection, lossy compression, decoding of compressed signals, etc. This specialisation makes it more efficient at those workloads than microprocessor cores designed towards more general use cases (ARM Cortex-M, i8051, AVR, PIC, MSP430, ...), which can help to further meet said design constraints. Chapter 5 discusses the specifics of that core and its instruction set extensible by the silicon vendor.

Software development for cores such as the HiFi 4 DSP, however, presents extra challenges. At the very least, because of their position in devices that contain them, essentially forming an AMP system, it is necessary to implement needed synchronisation primitives and inter-core communication mechanisms. That can present an additional burden when dealing with time-sensitive tasks. Also, cores like the HiFi 4 DSP block are best utilised with a specialised compiler, which can emit code containing special instructions specific to the HiFi 4 DSP itself and its instance in a particular device.

This thesis is oriented towards audio applications of the NXP's i.MX RT685 microcontroller, which instantiates said HiFi 4 DSP core as a coprocessor. Specifics of that microcontroller are discussed in Chapter 3. Some of those said burdens associated with utilising DSPs as an audio processing engine can be mitigated by developing an audio processing framework, which presents the DSP as an environment with discrete processing blocks, which configuration and topology can be changed dynamically. The Sound Open Firmware (SOF) framework is exactly that. Its porting effort is described in Chapter 6. SOF requires an RTOS to run and one of the supported RTOSes is Zephyr, which is discussed in Chapter 4 and ported in Chapter 7.

# Chapter 2

# Commonalities of microcontrollers

## 2.1 Definition and brief history

A microcontroller can be defined as a highly integrated computer system suited towards single-purpose (embedded) applications, which exists on a single integrated circuit, requiring a minimal set of external electronic components to properly operate. This fact is in direct contrast with microprocessors, often times used as well in embedded applications to achieve similar goals. Microprocessors can't work standalone - they require external parts to fully function. These include the RAM (for runtime volatile storage), ROM or another type of permanent memory element (for storing its program and configuration), input-output hardware, clock sources, etc.

Because they are generally small and compact, their historical use was to replace electronically controlled systems built with discrete logic gate chips, with the added benefit of being able to change the „control logic“ (program) in the future, thus easily altering the function of a constructed device. This is possible with microcontrollers that feature an erasable type of program memory (EPROM, EEPROM or flash) and, with the advent of current microcontrollers with integrated communication peripherals, is possible „in the field“ via computer networks (OTA - Over the Air).

## 2.2 Common interfaces

There are a number of low-speed, low-complexity serial interfaces that can be found on contemporary microcontrollers. These usually serve the purpose of interfacing with various external components, such as power management ICs, mixed-signal processing ICs (codecs, DACs, ADCs, ...), sensors, input and output peripherals, wireless communication devices and alike.

### 2.2.1 UART

The simplest of them all is the UART interface. This interface provides an asynchronous, generally point-to-point link oriented at transferring values of an arbitrary length (usually 8 bits) in length with optional parity and flow control. The usual manifestation of it in microcontrollers is as the Rx/Tx pair of signals at the raw TTL voltage level, providing a full-duplex serial link without flow control suitable for communication at shorter distances, however, devices operating with conformance to standards like RS-232, RS-422 and RS-485 also fall under the definition of a UART peripheral.

A driven UART line (Tx exposed by the master device, order of signals is swapped at the other end) is held high when idle - no transmission of data is in progress. When a transmission is triggered, the line is held low for the period of a single bit, transmitting a start bit. Then follow the desired data bits, LSB first, an optional parity bit and then stop bits, transmitted as high. A UART line has to be configured beforehand, specifying the baud rate (frequency of bits on the line), number of stop bits and the type of the parity bit. Usual baud rates include 9600, 19200, 57600 and 115200 baud. The timing diagram of a typical 8-N-1 (8 data bits, no parity, 1 stop bit) transmission is shown in Figure 2.1.

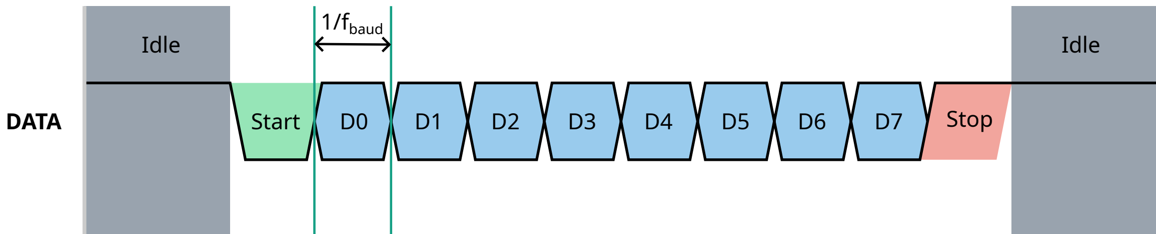



Figure 2.1: Timing diagram of an 8-N-1 transmission

This scheme allows the asynchronous transmission of data - without providing a clock signal, against which the bits are referenced. UART receivers usually run at a higher clock rate than the configured baud rate and sample each bit either in the middle of its period or multiple times during it, performing a vote on its final value. As a receiver is effectively resynchronised at the start of each transmission by the start bit, any clock skew is eliminated.

This interface can be used wherever point-to-point serial communication is necessary, such as for exchanging data between microcontrollers or to provide diagnostic (debug) communication, a usual use case for this interface. This interface has the advantage of being easy to implement and use, however, communication parameters have to be manually configured and is best used only as a point-to-point link because there's no arbitration, although implementing a simplex link routed to a number of slave devices is possible.

### 2.2.2 I²C

$I^2C$ is also an interface that is commonly used in embedded systems. It's a synchronous bus exposing two signals - SDA (shared, bidirectional data line) and SCL (clock), that allows the communication of a number of slave devices with a master device. Usually, the master device is singular, but it's possible to have multiple masters on the same bus with arbitration. Each slave device is addressed by a 7-bit address, although the standard also describes a voluntary extension that expands the address to 10 bits.

Commonly used modes of $I^2C$ have both lines configured as open drain - both lines are pulled high with pull-up resistors (that are either integrated into the master device or are placed externally) and are „grounded“ by driving transistors of each participant. The SCL line is primarily driven by the master and all participants take turns in driving the SDA line. This way, if any device drives any of the lines low, it will be measured as low by all devices. Bus arbitration happens by comparing the current state of signals to the values the device expects. For example, a slave can pull the SCL line low when it acknowledges a request but can't respond yet (data is not yet available), a technique known as clock stretching.

A simple transaction is illustrated in Figure 2.2. When the bus is in the idle state, both signals are not driven, thus high. Upon starting the transaction, SDA is pulled low and SCL is held high for the duration of a single bit. Then SCL goes low and on its rising edge, the 7-bit address is signalled. Then a read-write flag follows, along with an acknowledge bit. That bit is driven low by the slave device if it acknowledges the transaction. If no slave acknowledged the transaction, it is terminated in a fashion described below. If it was acknowledged, 8 data bits follow, MSB first. If a write transaction was signalled, the master signals those bits and, conversely, slave signals them for a read transaction. Acknowledge bit, driven by the receiving side (master for read, slave for write), then follows. The transaction is terminated (stopped) by driving that bit high and the master signalising a stop condition - SCL is left high, SDA is driven low for the period of one bit and then driven high. The transaction can continue with multi-byte values by driving the acknowledge bit low and continuing with the next byte up until another acknowledge bit.

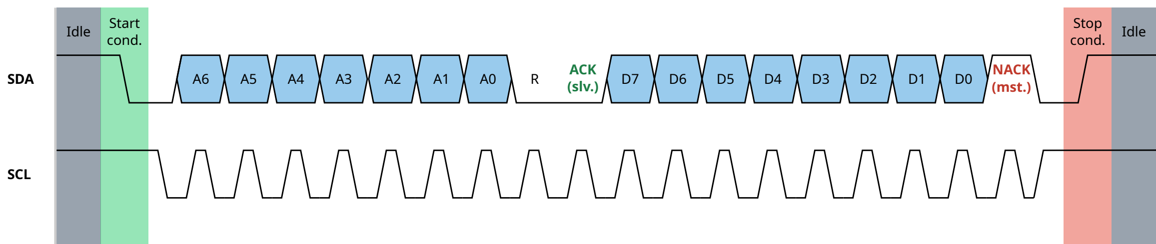



Figure 2.2: Timing diagram of a simple I$^2$C transaction

The bus is defined on a number of signalling modes:

- Standard (Sm) - 100 kbit/s
- Fast (Fm) - 400 kbit/s
- Fast plus (Fm+) - 1 Mbit/s
- High-speed (Hs) - 1.7 Mbit/s and 3.4 Mbit/s

The typical use case for this bus is interfacing with various low-speed devices, such as GPIO expanders, sensors, ADCs/DACs, digital potentiometers, programmable gain amplifiers, audio codecs, EEPROMs, touch panel controllers and others. Its advantages include the fact that it's a multidrop bus, allowing the connection of multiple slaves and masters. Ease of use is also one of its properties - this bus requires just a couple of pull-up resistors and its low speed doesn't pose any extra requirements for signal routing (length matching, microstrips, ...). However, it's not that easy to emulate in software (bitbang), is inherently half-duplex and its speed, depending on its application, can also be a disadvantage. Low speeds are implied by the fact that the lines are configured as open drain - signal rise times (transitions from low to high) tend to be high because of the parasitic properties of the used conductors.

### 2.2.3 SPI

SPI is a serial, synchronous, full-duplex bus implemented on 4 signals - MISO (Master In, Slave Out), MOSI (Master Out, Slave In), CLK (clock) and CS (Chip Select). Slave devices should use tri-state output drivers for the MISO signal, controlled by the CS signal,

thus, when the slave is inactive (particular CS is high), the signal should be in the high-impedance (practically disconnected) state. Failure to ensure this can result in short circuits and hardware destruction.

A transaction is begun by the master selecting a device by pulling a particular CS line to low and by simultaneously generating the CLK signal and outputting data on the MOSI signal in reference to the clock. Selected slave can respond by outputting data on the MISO line in the same fashion. SPI can be considered as a simple interface between the shift registers of each participant - there aren't any facilities for data framing, flow control and others. A simple transaction is illustrated in Figure 2.3.

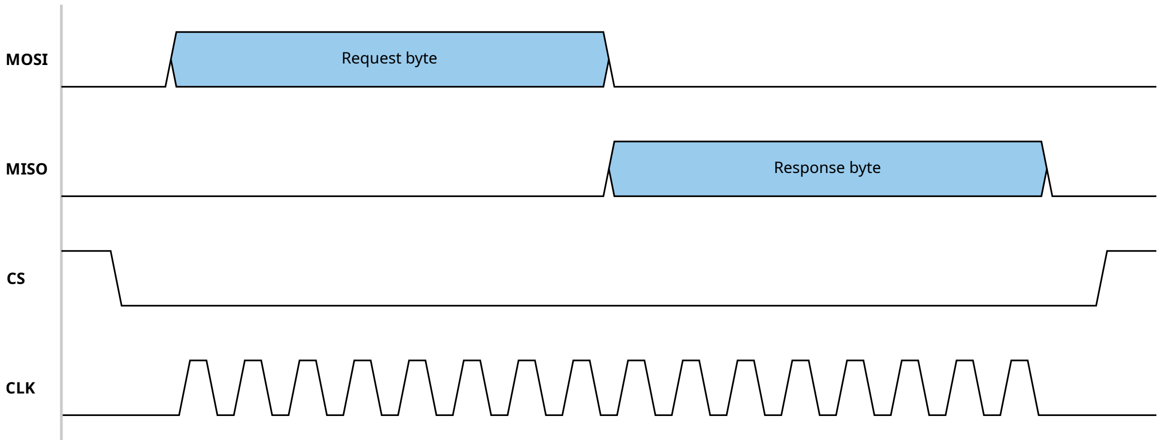



Figure 2.3: Timing diagram of a simple SPI transaction

The SPI interface has 4 different modes (0 to 3), implying when data is transmitted and when sampled (combinations of rising/falling edge of the CLK signal and CS).

SPI can be used for higher speed peripherals, such as LCD controllers, faster ADCs, DACs, communication adaptors (IEEE 802.3, IEEE 802.15.4, USB, ...), flash memories (including SD cards) and others. Its primary advantage is simplicity - data is transmitted and sampled against a clock without any framing, transaction control or flow control. This implies that the communication can be easily bitbanged. As the clock frequency is not defined, the possibility of using higher rates is entirely dependent on the capabilities of each participant and parasitic properties of the utilised wiring. On the other hand, exactly those properties are a limiting factor that can prevent such higher speeds (in the magnitude of tens Mbit/s) and the fact that an SPI interface consists of 4 signals, 3 of which are shared (CLK, MISO, MOSI) and 1 of which is a simplex signal unique to each device (CS) can make this interface somewhat cumbersome to utilise with more slaves.

### 2.2.4 I²S

The I²S interface is a serial, point-to-point, simplex, synchronous interface designed for transmitting audio data over short distances between ICs. The interface consists of 3 signals - SD (serial data), SCK (clock) and WS (word select). All of those 3 signals are, in the case of audio playback configurations, usually generated by the transmitter device producing the audio data, however, it's possible for the receiving device to dictate timing by it generating the SCK and WS signals instead. [19]

A transmission of an I²S frame is demonstrated in Figure 2.4. The audio samples are simply transmitted MSB first on the SD line and clocked against the SCK signal - bits

are sampled on each rising edge of that clock. The WS signal is used for channel selection - unused for mono audio, low for the left channel and high for the right channel when transmitting stereo audio data. The WS signal is shifted against the SD line by 1 bit into the past - upon changing the value of the WS signal, there's still a single bit of a current sample to be transmitted before moving to transmit the MSB of the following channel. [19]

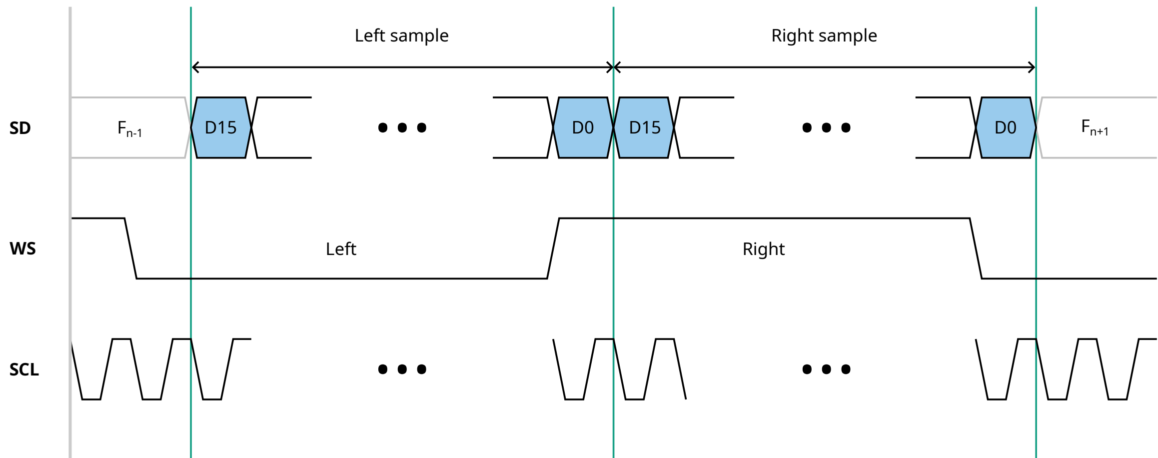



Figure 2.4: Timing diagram of a classic I$^2$S frame bearing 16-bit stereo data

The variant described above is sometimes referred to as classic I$^2$S in the field. There are other variants of I$^2$S interface, however. For example, the left-justified mode eliminates the 1-bit shift shift when changing the WS signal, upon it changing, the MSB of the following sample is available on the rising edge of the SCK signal. For configurations with more than 2 channels of audio, the TDM mode is used. This mode pulls the WS signal up with the falling edge of SCK when the LSB of a last sample is being transmitted and pulls it low at the next falling edge, keeping it low for the remainder of the frame until the last LSB is being transmitted. Samples then follow a predefined order arbitrarily specified by the devices. [5]

The advantages of this interface is that it's easy to implement, use and that it's a de facto standard for exchanging audio data between various ICs and systems - there is a staggering number of I$^2$S-compatible parts from competing vendors. I$^2$S was strictly defined by then Philips Semiconductors (now NXP Semiconductors), however, the other variants aren't ratified anywhere.

## 2.3 Programmer's model

Contemporary microcontrollers can be frequently seen with RISC-style processor cores (ARM, RISC-V, Xtensa, ...), peripherals that are exclusively memory-mapped and optional memory protection. Because of their scale and ultimately target application, the programming environment differs from, for example, desktop computers running contemporary operating systems. These differences can be:

- limited resources - computing power, the capacity of available memory devices
- no sophisticated memory management features (virtual memory, paging, lazy allocation, file views, ...)

- exclusively static linking of software libraries
- linking against particular memory segments, no relocations
- direct access to hardware peripherals
- need for more careful sequencing of actions and events - greater desire for deterministic behaviour
- specialised toolchains and development environments

Virtual memory, as usually implemented in application SoCs or personal computers, is generally not present in microcontrollers and isn't even desirable for applications which microcontrollers can fulfil. Page faults can bring non-deterministic behaviour into the application and modify the timing of events, a crucial aspect of real-time applications. Mapping of a whole 4 GiB virtual address space can be considered as hard to achieve in constrained environments, as an application-accessible TLB requires extra hardware, implying increased power consumption. A tree of page tables, as seen in the Intel IA-32 instruction set, would also require extra hardware to facilitate address translation and also implies requirements for application memory to store such trees - a resource that is already scarce.

The desire to link object files against a set of predefined memory addresses and not to have any dynamic libraries is implied by the fact that the address space of microcontrollers is usually statically defined in the hardware, thus all code has to be linked with those statically defined addresses in mind. As microcontrollers don't provide any „runtime environment“ to speak of, which could handle dynamic linking and relocations, execution of application code is commenced by directly transferring execution (jumping) to the application's entry point. All addresses to all symbols (static variables and subroutines) thus have to be resolved beforehand, at the time of compiling and linking the code. Self-modifying code which could relocate itself is difficult or even impossible to implement - application code can be stored in an EPROM/EEPROM, which can't be directly changed, as well as flash memories, which require special procedures to be reprogrammed. However, it can be possible to relocate certain symbols outside the ROM/flash to RAM to speed up execution of frequently used code, the code just has to be copied before used. Microcontrollers featuring Harvard-style processor cores (AVR, PIC), by definition, forbid direct write accesses to instruction memory.

Requirements for specialised toolchains and environments are mandated by the lack of a runtime environment integral to the target platform itself and, as Chapter 5 discusses, those toolchains have to properly handle specialised devices that are fundamentally different to regular desktop computers, such as processor architectures that are reconfigurable by the hardware vendor. Special debuggers have to be used for microcontrollers as analysing their state and influencing their behaviour means accessing a low-level interface exposed by them. Such debuggers have to have facilities for statefully interfacing with the system's flash memory, to allow program upload. „Creature comfort“ features, such as fault state analysis, peripheral state analysis, data logging, execution tracing, time profiling, simulated breakpoints and others are also highly desirable, if not necessary for the development of embedded applications.

## 2.4 DMA

DMA is a technique for transferring blocks of data where peripherals can directly access system memory independently. This technique is useful for efficiently implementing transfers of data from various sources to various destinations - instead of the processor core performing them by itself, a DMA-capable peripheral or a DMA controller can be programmed to perform the transfer independently and notify the processor upon finishing such transfer by raising an interrupt request. As an effect, the act of copying data is offloaded off the processor and the processor time spent by the application is reduced. That saved processor time can be then spent by other parts of it. Data is usually transferred using DMA between blocks of memory or between a block of memory and an input/output peripheral.

Current DMA controllers can have rich sequencing capabilities. For example, transfers can be started (triggered) upon a peripheral asserting a special trigger signal, or by external signals. In addition, programmed transfers can repeat (loop) or link to other programmed transfers, leaving the programmer with the option of completely offloading the act of desirably sequencing and timing the transfers from the processor and it not participating in the data flow itself. Configuring transfers this way may have the benefit of being able to implement continuous, uninterrupted flows of data. This is an important aspect of use cases where interruptions of such flows are highly undesirable, such as audio playback and capture.

The use of ping-pong transfers is a manifestation of this. Such a configuration implements a pair of buffers, where one of the buffers is used for storing data about to be transferred by a DMA controller and the other one holds data which is actively being transferred by it. Upon completing a DMA transfer, the two buffers are switched and a new DMA transfer is programmed and triggered. If the timing is correct and the data to be transferred is present before the buffers are switched, the result is an uninterrupted sequence of DMA transfers and thus a flow of data. Such a technique can also can be extended to a ring of buffers.

Various addressing modes, such as modulo or scatter-gather addressing, can also be found and used. Modulo addressing is useful for transferring data to and from ring buffers and scatter-gather is about linking various blocks of memory and treating them as a single block, both for transferring to and from. A scatter-gather addressing mode can be useful for receiving, modifying and retransmitting data, for example for routing computer network traffic, where different portions of a processed packet can be handled and modified separately and even concurrently, if the hardware is constructed in such a way that permits this.

# Chapter 3

# The i.MX RT685 microcontroller

## 3.1 Overview

The i.MX RT685 is a 32-bit microcontroller made by NXP Semiconductors. It is based around the Cortex-M33 processor core running at 300 MHz and has 4.5 MiB of SRAM available in total for application use. Atypically, the program flash memory is not built in - the device utilises an external SPI quad-port or octa-port flash memory chip. It does so by utilising the FlexSPI peripheral, which can provide a transparent memory-mapped interface for external memory devices, even SRAMs. 8 Flexcomm peripherals are provided, which function can be switched between providing UART, $I^2C$, $I^2S$ or SPI interfaces. 2 eMMC/SD/SDIO interfaces are available, as well as a reconfigurable clock distribution system and power domains controllable by software. The chip offers 2 USB 2.0 interfaces, which can both implement a device and a host. Debugging and tracing is possible using the exposed SWD interface, commonly used with SEGGER's J-Link and NXP's LPC-LinkII debuggers. Common algorithms used in cryptography (such as AES-256 and SHA-2) are accelerated with an extra hardware peripheral - the CAMMU block.

This device is marketed by NXP as a crossover microcontroller, meaning some of the features provided are not typical for a classic microcontroller (high-speed SDIO interfaces, ability to transparently utilise external memory, high-resolution LCD interfaces, ...). While NXP's portfolio is not unique in this regard - one of the competing products is the highly advanced ESP32 series of microcontrollers from Espressif Systems, this fact shows that the line between application SoCs (MPUs) and microcontrollers gets blurrier with each development.

One of the highlighted features of this device is the inclusion of a Cadence HiFi 4 DSP processor block, around which this thesis revolves. Its specifics are mainly discussed in Chapter 5 and Chapter 7.

## 3.2 Memory architecture, interrupt handling

Like comparable devices, the i.MX RT685 employs a flat, non-virtualised memory model with memory-mapped input/output (MMIO) peripherals. The discussed 4.5 MiB of built-in SRAM is directly accessible both by the main Cortex-M33 core and the DSP, optionally facilitating shared memory. That main SRAM block is also accessible by all DMA-enabled peripherals, so that various DMA transfers can occur.

The Cortex-M33 core is attached to peripherals using a multi-layer AHB matrix. Higher speed peripherals (Flexcomm, FlexSPI, SDIO, USB, ...) are attached to the matrix directly. Lower speed peripherals (timers, system control, reset control, ...) are attached using an AHB to ABP bridge. The FlexSPI peripheral directly maps external storage devices (such as quad-SPI or octal-SPI flash devices) to a block of system memory, both visible by the Cortex-M33 and the DSP core. It also provides transparent encryption and decryption of memory contents and data caches.

Although virtual memory is not implemented on this device (and isn't generally desirable in microcontrollers), region-based memory protection is available. Arm's MPU built into the Cortex-M33 core facilitates this. It can divide the whole address space into 8 segments, access permissions (read, write) to which can be controlled. [2]

The HiFi 4 DSP has direct access to SRAM TCMs assigned directly to it - these are the DTCM (data memory) and the ITCM (instruction memory), both sized at 64 KiB. Although the DSP can use any piece of the main SRAM, both for data and code (instructions), the default vector table is expected in the ITCM, thus it's mandatory when using the DSP in the target application.

Interrupts of the main Cortex-M33 core are controlled and handled using the standard ARM NVIC controller. It is instantiated with 52 vectored interrupts with 8 priority levels and hardware priority masking.

Interrupts on the DSP are handled using an interrupt controller integral to the Xtensa architecture. It offers 32 interrupts in total, 27 of which are external, wired to peripherals. Priority allocation is static - implied by the interrupt number. The external interrupt lines are multiplexed using an interrupt multiplexer implemented in the INPUTMUX peripheral of the microcontroller, which attaches the external interrupt lines to selected peripherals. There are 34 total sources (Flexcomm, GPIO, timers, DMA, ...) which can be attached to those lines through the multiplexer. [18]

## 3.3 DMA

The microcontroller has 2 DMA controllers, `DMA0` and `DMA1`. Their capabilities are identical, however, it's recommended to utilise both controllers if both domains (Cortex-M33 and HiFi 4 DSP) are intended to program DMA transfers and/or secure and non-secure domains are implemented. [18] The first case applies to the use case, which this thesis discusses, thus the Cortex-M33 core uses `DMA0` for peripherals driven in its domain and the DSP uses `DMA1`.

Each of the DMA controllers has 33 channels, which can be triggered by various events. By default those channels are triggered by software (by supplying and the validating channel configuration), but channels can also be triggered by hardware events from various peripherals on the device (timers, Flexcomm, ...). The selection of a peripheral trigger source is done by selecting a channel and enabling a peripheral trigger in the controller's registers - trigger sources are defined in hardware and there are channels without a peripheral trigger source. In addition, DMA transfers can be triggered by a second set of trigger sources that are selected using the INPUTMUX peripheral. Both DMA peripherals provide trigger outputs which can be routed back as selectable trigger inputs using the same peripheral, enabling chained operation, which can be used to implement more involved DMA transfers (such as ping-pong transfers).

## 3.4 Inter-core communication

As this microcontroller has 2 separate cores, extra hardware helping implement inter-core communication mechanisms should be expected. And that is the case - the i.MX RT685 microcontroller has two peripherals which help meet that goal.

One of those peripherals is the MU (Message Unit). This peripheral implements a simple mailbox mechanism - one core can write (post) a value and the other one can retrieve it. Upon writing a value, an IRQ is generated to notify the other core. The MU works with 32-bit values and has 4 channels - 4 registers which can be written and 4 registers which can be read. It's divided into 2 ports, each one mapped at a different base address and separately utilised by respective cores. Each port has separate read and write registers reflecting their counterparts in the opposing port. IRQs can also be generated without the need of posting a value - MU can facilitate generating plain IPIs. The interrupts can be masked to abandon notifications using IRQs, letting the receiving cores poll the peripheral instead, as the MU has status registers. [18]

The remaining peripheral is the SEMA42 peripheral, which provides 16 gates - hardware-assisted state machines exposed as registers, helping with the implementation of simple spinlocks. Each register holds a 4-bit value, with 0 signifying an unlocked state and all other values signifying a locked state by a particular bus master (processor core). An attempt to lock a gate is made by writing a bus master index (0 for the Cortex M33 core, 1 for the DSP) incremented by one, then reading the particular register back and checking if the read value is the same as the one written. Unlocking is made by writing a zero to that particular register. Only the core which locked the gate can unlock it - the index of a particular bus master accessing a given register is checked against its value. This eliminates the need for atomic instructions and a coherent cache across all cores, otherwise needed if spinlocks were to be implemented with just shared memory locations.

## 3.5 Evaluation boards

The i.MX RT685 microcontroller is integrated on two evaluation boards made and distributed by NXP.

The MIMXRT685-EVK (fig. 3.1) integrates the microcontroller with the PCA9420 PMIC featuring 2 programmable LDOs and 2 programmable buck converters, the Macronix MX25UM51345GXDI00 octal SPI flash memory, apmemory APS6408L-OBM-BA PSRAM, an accelerometer, a digital microphone with a PDM interface, the Wolfson WM8960 stereo audio codec, dual class D power amplifiers and an SD card slot. [13]

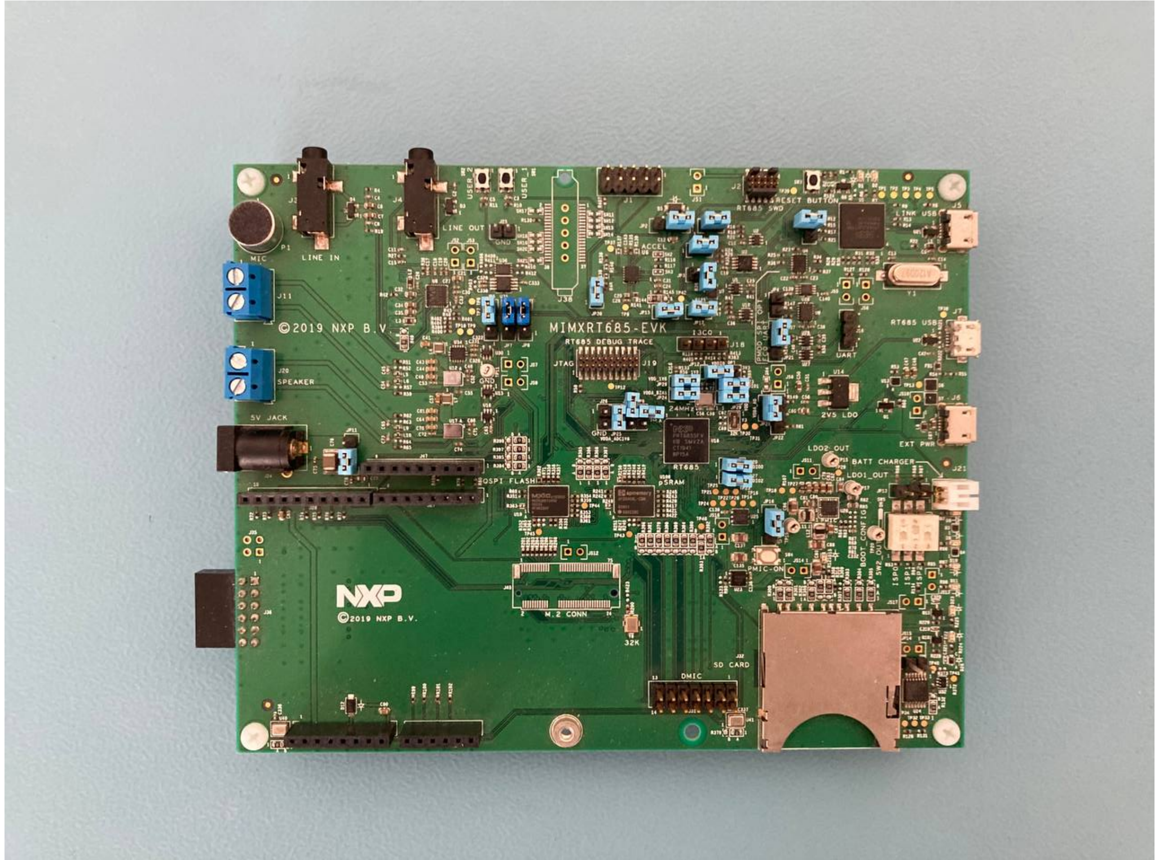



Figure 3.1: Photo of the MIMXRT685-EVK evaluation board

The MIMXRT685-AUD-EVK (fig. 3.2) differs slightly in the form factor, the flash, interfaces and supplemental ICs. The power audio amplifiers are absent and the flash chip has been substituted with the MX25U51245GXDI00 from the same manufacturer. [12] Also the audio codec has been replaced - the board contains the CS42448-DQZ from Cirrus Logic, which allows to transmit (output) up to 8 and receive (capture) up to 6 channels of digital audio over a standard I$^2$S interface or its modifications. [5] An M.2 E-key slot is also exposed for connecting SDIO devices, such as IEEE 802.11 (Wi-Fi), IEEE 802.15.1 (Bluetooth) or IEEE 802.15.4 (used for 6LoWPAN, Zigbee, ...) radios.

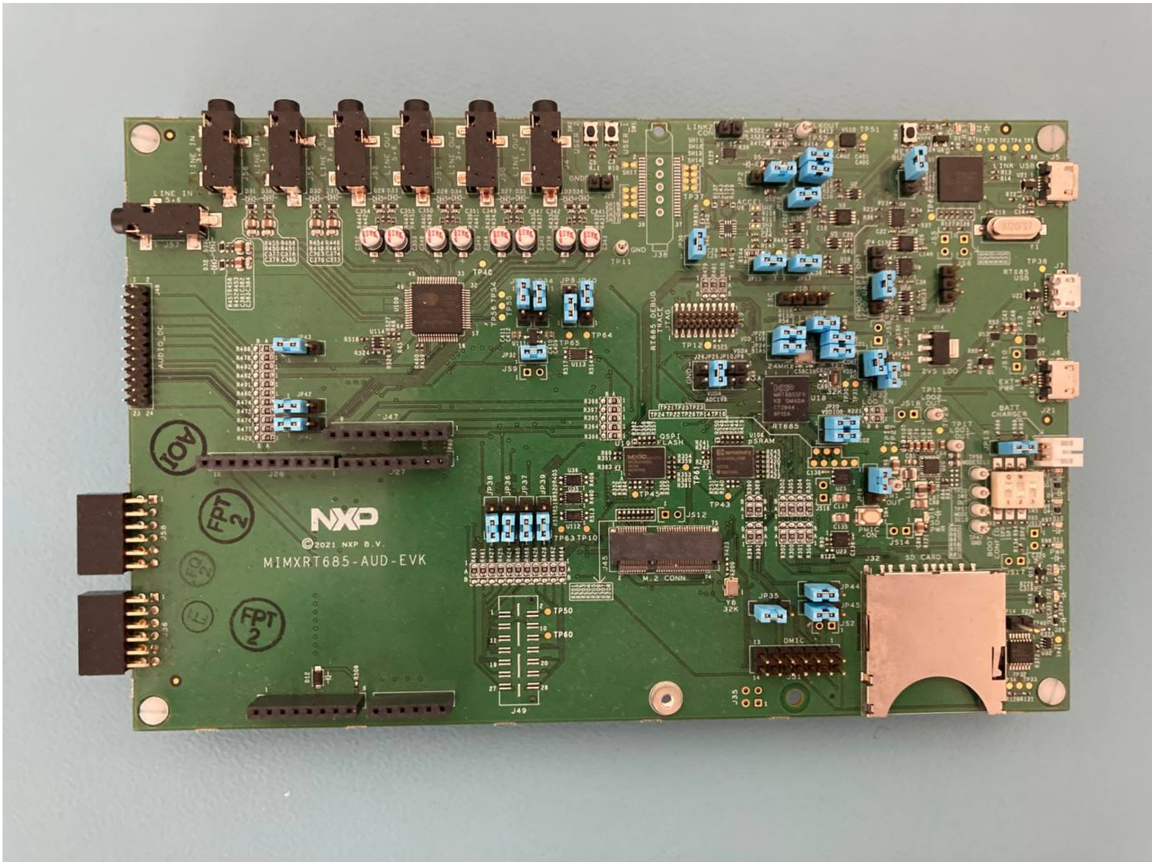



Figure 3.2: Photo of the MIMXRT685-AUD-EVK evaluation board

The difference of flash memories (and their configured interfaces) makes the boards object-code incompatible.

## 3.6 Possible use cases

Subjectively, the discussed microcontroller can be rightfully used in these applications:

- network audio streamer
- Bluetooth A2DP compatible speaker device
- smartwatch
- IoT sensor or control device
- voice assistant client device

These use cases were implied from the connectivity, processing and low power capabilities of the discussed microcontroller. One distinguished application of a related device, the i.MX RT595, is the use in Garmin and Xiaomi smartwatches. [8, 17] Fact sheet for both of these devices ([15]) lists more possible use cases for them.

# Chapter 4

# The Zephyr RTOS for Embedded Domain

## 4.1 Overview

Zephyr is an RTOS suited for use in resource-constrained embedded devices. It's written in C with platform-specific routines written in assembly and supporting tools in Python. Zephyr is built using the CMake build system. The kernel implements preemptive multi-tasking, synchronisation facilities, a unified device and driver model, memory management (focused on just implementing heaps) and power management. Additional components such as file systems, logging support, communication stacks and POSIX support are also provided by Zephyr. Currently, these platforms are supported:

- ARCv2, ARCv3
- Arm Cortex-M, Cortex-A and Cortex-R
- x86, x86-64
- MIPS Release 1
- NIOS II Gen2
- RISC-V
- SPARC V8
- Cadence Tensilica Xtensa family cores

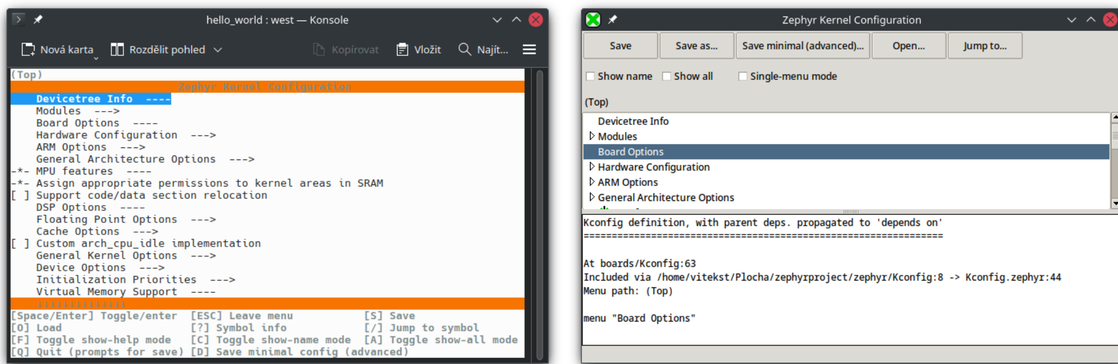
Zephyr originated in Wind River Systems, where it was eventually released and licensed under the Apache 2.0 license. Currently, Zephyr is an open-source project managed by The Linux Foundation and development is steered by a committee formed with industry representatives. This committee includes members from, among others, NXP, Nordic Semiconductor, Meta, Linaro, Intel, Synopsys and Wind River.

Zephyr can be considered as a unikernel operating system - instead of implementing a completely abstracted runtime environment for applications, in reality, it functions more like a software framework for developing embedded applications. It runs in a single address space, thus in the same address space as the application code, and all Zephyr routines are

statically linked, resulting in a single image where system services are invoked directly. All abstractions provided by it are implemented just to provide unified, conventional structures and facilities for writing applications.

## 4.2 Configuration

Zephyr RTOS and the components associated with it use the Kconfig mechanism for the configuration of compile-time features and variables. [22] The result of the Kconfig mechanism is a C header file containing all of the set properties, which all have the `CONFIG_` prefix. Properties available for configuration are described hierarchically in Kconfig files with, optionally, implications, automatic selections and constraints. Based on this description, the resulting configuration is validated against this database at build time, which can rule out inconsistent or unsupported set of settings. The configuration can be fragmented - there are separate Kconfig files for SoCs, boards, application projects themselves and the resulting configuration can be overridden (overlaid) on the application-board level. Interactive configuration tools, such as `menuconfig` (fig. 4.1a) and `guiconfig` (fig. 4.1b) are also provided. They are available as CMake targets of each properly instantiated project.




(a) The `menuconfig` tool

(b) The `guiconfig` tool

Figure 4.1: Interactive configuration tools

## 4.3 Development toolset

Zephyr offers West - a command-line tool written in Python for managing a set of multiple Git repositories and executing implemented commands on those repositories. [22] Through the West manifest, located in each workspace and each repository in that workspace, it manages revisions of each repositories and provides a way to change (checkout) those revisions in bulk. This makes tasks like branching (and keeping dependent repositories at a specific revision) and updating repositories to a newer version easier. This works subjectively well especially because the Zephyr RTOS itself is structured into multiple repositories. Among those is the base repository, HAL repositories and additional components like communication components, debugging extensions, file systems and others.

Zephyr also uses West as a way to build applications (by invoking the correct CMake targets), flash them and start their debugging sessions. Zephyr does this by having custom commands defined in the included repositories, which can be parameterised by configuration

supplied at the board layer. For example, it's possible to directly start a GDB server for the proper part, for which the debug session is being started.

By default, Zephyr is built with the Zephyr SDK - a collection of prebuilt toolchains, QEMU variants and other tools. Each of these toolchains consists of a version of GCC set up for cross compiling, C and C++ runtime libraries and a version of GDB. User can select which toolchains to install and just has to execute a setup script - no other steps (like including the toolchain paths in environment variables) are necessary. The toolchains are built with crosstool-NG, an environment for building toolchains, which enables building toolchains based on source tree overlays and a Kconfig-style configuration. This makes the adaptation of the toolchains for new platforms and subsequent maintenance less difficult than having a separate source tree for each target.

## 4.4 Driver model

Zephyr encompasses all structures associated with a particular device and its driver in the `device` structure. Major items of that structure are name, initial driver configuration (information such as base addresses, IRQ numbers, custom configuration, etc.), a structure with pointers to API functions which the driver implements and its internal state. [22]

When booting, Zephyr calls registered initialisation functions of all devices defined in the system. Initialisation function then initialises the driven hardware device and can make calls to other drivers or subsystems to facilitate initialisation (such as clock control or interrupt handling). Priority with which are those functions called can be adjusted, as Zephyr implements 5 different initialisation levels, which differ in the execution context and the range of services available [22]:

- `EARLY` - executed immediately after and invoking Zephyr's C entry point (`z_cstart`), no kernel services are available
- `PRE_KERNEL_1` - suitable for initialising devices with no dependencies outside the microcontroller, no kernel services are available except interrupt attach functions
- `PRE_KERNEL_2` - intended for initialising devices which depend on devices initialised in previous levels, availability of kernel services is the same as in `PRE_KERNEL_1`
- `POST_KERNEL` - used for devices depending on kernel services for initialisation, possible use case is initialisation of devices existing on dedicated buses or communication channels (UART, SPI, I$^2$C, ...)
- `APPLICATION` - dedicated for user components, such as parts of communication stacks

Initialisation functions are mainly used for drivers, but don't have to be associated with drivers. Using the `SYS_INIT` and `SYS_INIT_NAMED` macros, it's possible to define an arbitrary initialisation function at any level. Initialisation functions can fail (return a negative `errno`). If such condition occurs, the device is considered as not ready. By convention, any code that uses a device through the Zephyr's driver model is supposed to check device's readiness with the `device_is_ready` function.

A driver can expose an API in the form of a pointer to an arbitrary structure containing pointers to functions implementing the desired API. Those can be directly consumed by the application or wrapped by a subsystem layer, which conceptually implements syscalls.

Those subsystem layers can range from simple wrapper functions invoking functions referenced in the API structure to subsystems with extra handling logic. By convention, all of those functions consume a pointer to the device's `device` structure as a first parameter.

Devices (driver instances) can be defined only statically in Zephyr - dynamic device discovery and creation is not supported. Not considering DTs, this is done using the `DEVICE_DEFINE` macro, which statically allocates the `device` structure, fills it out with supplied data and registers an initialisation function, if one was given.

Zephyr also supports power management. Drivers can optionally provide a `pm_device` structure (defined by the `PM_DEVICE_DEFINE` or `PM_DEVICE_DT_DEFINE` macros) which exposes a callback function used for controlling the power state of a particular device. Following actions are supported:

- `PM_ACTION_SUSPEND` - causes the device to enter a low-power state
- `PM_ACTION_RESUME` - causes the device to exit a low-power state
- `PM_ACTION_TURN_ON` - powers up the device completely
- `PM_ACTION_TURN_OFF` - powers down the device completely

Structures associated with devices instantiated under the Zephyr driver model are compiled into special sections (`device_area`, `device_states`), which implies the possibility of moving those sections into different memory segments (thus different memory devices). This can help speed up the execution of driver routines and, by extension, the whole application, if the driver routines present a hot spot.

## 4.5 Device trees

Together with a unified driver model described above, the Zephyr RTOS implements device trees.

A device tree (example in Listing 4.1) is a hierarchical structure that describes specifics of different pieces of hardware (base addresses, interrupt vectors, variants / quirks, parameters, ...), relationships between such devices and references to compatible drivers, which then control the described devices. [22] The overall concept is the same as device trees in Linux and, in fact, source representations of both have the same syntax and similar conventions.

All definitions in a device tree are structured into nodes. Each node has a name (1) and can have a unit address (2), label (3), properties (4) and child nodes (5). Properties are defined by their name (6) and value (7). Value can be a string (8), a reference to another node (a phandle, 9), a tuple of 32-bit integers (cells value, 10), an array of bytes or a list consisting of items of any said types. A root node, `/` (11), is always defined in a valid resulting device tree. Nodes can be later located by their absolute path (for example, `/soc/peripheral/clkctl@21000`) or referenced by their label (3), which must be unique in the whole tree.

There are properties which have a fixed, defined meaning (such as `ranges`, `compatible`, `status` and others) and properties which are specific to each node, or more precisely a driver which is instantiated from a given node.

```
/ (11) {
    soc (1) {
        peripheral (3) : peripheral@40000000 (2) {
            ranges = <0x0 0x40000000 0x10000000>; (4)
            #address-cells = <1>;
            #size-cells = <1>;

            clkctl1: clkctl@21000 { (5)
                reg = <0x21000 0x1000> (10) ;
                compatible (6) = "nxp,lpc-syscon"(7) ;

                #clock-cells = <1>;

                status = "okay";
            };

            i3c0: i3c@36000 {
                reg = <0x36000 0x1000>;
                compatible = "nxp,mcux-i3c" (8) ;

                interrupts = <49 0>;
                clocks = <&clkctl1 (9) 0x600>;

                status = "okay";

                redundantproperty = <1>;

                audio_codec: wm8904@1a (12) {
                    reg = <0x1a 0 0>;
                    compatible = "wolfson,wm8904";

                    status = "okay";
                };
            };
        };
    };
};
```

Listing 4.1: A simplified device tree example

Among those is the `reg` property specifies the base register of the device being described - usually its address and length. If the base addresses are translated (using the `ranges` property in the parent node), then this address is relative (non-translated). The contents of the `reg` value are dictated by special properties in the parent node - `#address-cells` and `#size-cells`. This can cover both definitions for devices directly mapped to memory (5) and for devices on a special bus (12, such as $I^2C$, where devices are accessed only by a device address).

Similarily, there are bindings for common resources, such as GPIO pins, clock signals and interrupts. The structure of a binding is dictated by a cell count property (`#clock-cells`, `#gpio-cells`, ...) of a node providing that resource. These are then interpreted by the driver providing that resource and can be used to further describe that binding (pin direction, desired clock frequency, interrupt priority, ...).

Device trees can be merged - overlaid on top of each other. An example of an overlay is in Listing 4.2. Device tree overlays mainly contain changed properties of nodes, new values of existing nodes or new nodes, but by prefixing node and property names with delete prefixes (`/delete-property/` and `/delete-node/` respectively), information can be removed.

The ability to overlay device trees enables developers of different pieces of a resulting application to structure device definitions as different, abstracted device tree fragments (part, board, application, ...), promoting abstraction, problem isolation and application migratability. If an application is developed solely against the Zephyr's unified driver model, migrating it between hardware platforms, theoretically, can be just a matter of compiling the application with a suitable toolchain and a different device tree.

```
i3c0 {
    /delete-property/ redundantproperty;
    status = "okay";
};
```

Listing 4.2: An example of an overlay modifying the `i3c` node

Contrary to Linux, Zephyr does not directly use compiled device trees (DTBs - device tree blobs), but translates the structure into a C header file and object files with associated binary contents. This structure is then statically traversed using C preprocessor macros, from which drivers are instantiated at compile time and from which all required values are acquired.

Compiler errors produced from this process can be cryptic and thus device tree issues can be quite difficult to grasp and rectify, but this technique guarantees allocations and relationships between drivers that are known at compile time and also eliminates runtime costs of traversing device trees and instantiating drivers. This also makes the final sizes of structures associated with drivers known statically, making predictions about the developed application's memory requirements easier by that amount.

## 4.6 Audio driver APIs

Zephyr currently offers 2 APIs for audio input and output peripherals. These are the `i2s` and the `dai` APIs.

The `i2s` API is the one that is subjectively more straightforward to use. It presents a completely abstracted audio device with a defined direction. Like the name suggests, it's centred around peripherals implementing the $I^2S$ bus and its variants. Its configuration accepts a heap (`k_mem_slab`), which is used as a queue for storing audio data about to be consumed (transmitted) or audio data that was produced (received) over the interface. The application then uses the `i2s_write`, `i2s_buf_write`, `i2s_read` or `i2s_buf_read` functions respectively to provide or gather that data. If the peripheral is configured as an $I^2S$ transmitter, the queue must be prefilled before starting the transmission (`i2s_trigger`), as queue overflows and underflows result in transitions into an error state. The act of sequencing transmissions is completely abstracted by the driver. If DMA transfers are employed, the driver is responsible for interfacing with a driver of an assigned DMA peripheral to facilitate those transfers. This API is thoroughly demonstrated in Zephyr examples (`samples/drivers/i2s/output`, `samples/drivers/i2s/echo`).

On the other hand, drivers implementing the `dai` API do not abstract the sequencing of transmissions and don't even provide a way to directly write or read data from the

peripheral's internal queue. Instead, the internal queue is described in a properties structure exposed by the driver (`dai_get_properties`). The API then offers just a way to trigger a continuous transmission (`dai_trigger`) and the responsibility of reading or writing to the peripheral's queue is shifted to the application. This API isn't demonstrated anywhere in Zephyr, but it's relevant for Sound Open Firmware, as is discussed in Section 8.4.

## 4.7 Comparison with other systems

RTOSes like FreeRTOS, NuttX and XTOS strive to achieve goals similar to Zephyr.

FreeRTOS is a simple system implementing just preemptive multitasking, synchronisation primitives and heap-based memory management. It is very small, but doesn't solve some of the more complex problems that Zephyr does solve (unified driver model, communication stacks, file systems, ...). FreeRTOS can be already found in existing SDKs made by microcontroller vendors (such as the ESP-IDF from Espressif Systems, MCUXpresso SDK from NXP and STM32 Cube from ST Microelectronics). Each SDK thus offers differing drivers with differing interfaces, preventing application portability and accentuating differences between those SDKs and thus microcontroller platforms. CMSIS standards and interfaces try to rectify this, but they abstract just a handful of hardware and software components. Some of those SDKs are also tied to development tools created or supported by the vendor, making them difficult or impossible to use with different IDEs and toolchains, if a situation requires such a choice. Some of those included drivers also may not be fully utilising services provided by the RTOS kernel (synchronisation primitives, for example), as they also need to run in bare-metal applications. One of those examples is the `sdmmc` driver in the MCUXpresso SDK - it uses active waiting for waiting on peripheral conditions in both bare-metal and RTOS applications. Such implementations make use of available processor time inefficiently.

NuttX is an RTOS that strives to create a POSIX-compatible operating environment for target applications, thus it maps all hardware interactions as Unix-style file accesses. Subjectively, not every device can be well abstracted with a file, devices requiring complex operations that go beyond simple reads and writes from a byte (for character devices) or a block (for block devices) stream have to be implemented by either `ioctl()` calls or by dynamically creating extraneous files representing each possible point for data input or output. Compared to just consuming an API made of function calls that are not a product of an abstraction, both of these methods can significantly impact the readability and simplicity of the target application, aspects which are both desirable especially in constrained environments like microcontrollers. Zephyr also offers a POSIX layer, but it's not integral to the operating system and thus not mandatory for applications to utilise.

Another relevant aspect of NuttX is how drivers are implemented and instantiated. NuttX does not instantiate drivers using a device tree, drivers are instantiated using code in the board layer and configured using Kconfig variables.

XTOS is an RTOS developed by Cadence targeted only at Xtensa platforms. It can support all of the extensions of an Xtensa instance directly - unlike Zephyr, context switches include switching the contents of registers used in SIMD instructions. However, the level of features provided by XTOS itself is even lower than the one of FreeRTOS - it provides only basic synchronisation primitives, manual context switching and heap management appears to be implemented by a tightly coupled C runtime library. As can be implied, XTOS does not provide the same level of functions as Zephyr - apart from the ones listed above,

facilities like a comprehensive device and driver model, communication stacks, logging, file systems and others are absent.

# Chapter 5

# The HiFi 4 DSP core

## 5.1 Overview

The Tensilica Xtensa HiFi 4 DSP is an extension to the base Xtensa processor architecture bringing specialised facilities (instructions, registers, ...) tailored to tasks involving digital signal processing (DSP) work. Despite the name itself referencing only that extension, this thesis refers to the Xtensa processor with the HiFi 4 extension implemented as it is implemented in the i.MX RT685 microcontroller as the *HiFi 4 DSP* and just *the DSP*, to be inline with existing documentation and associated texts produced by NXP Semiconductors.

Initial IP was developed by Tensilica, Inc., which was subsequently bought out by Cadence Design Systems, Inc. in 2013. One of the distinguished features of that IP is that the resulting processor core can be reconfigured based on the silicon vendor's requirements. Parameters like size of caches, MMU presence, ECC or parity support, exception handling specifics and others can be modified. In addition, the whole instruction set can be modified - both by deciding on the inclusion of predefined instruction set extensions and by developing custom extensions to the instruction set written in the proprietary TIE language. Based on those parameters and extensions, a hardware description is generated for synthesis and simulation, as well as a support package for software development along with a cycle-accurate simulation system. These tools integrate tightly with the Xtensa Xplorer - an Eclipse based IDE for software development for Xtensa instances. Deep runtime analyses can be done with this environment, for example processor pipeline usage can be studied.

## 5.2 Instruction set

The Xtensa base instruction set is a RISC-style ISA intended for general computing tasks. This is apparent in the Espressif's ESP32 series of microcontrollers, where Xtensa LX6 and LX7 cores are used as primary ones. [7] Despite the base instruction set being presented as RISC-like, the base Xtensa ISA employs conditional register move instructions (`MOVxxx`) and implicit bitwise shifts when adding (`ADDMI`, `ADDXn`). Among others, the following optional facilities extending the minimal core instruction set are available: [4, 10]

- variable instruction length (shorter instruction lengths)
- zero-overhead loops (loops of a known number of iterations without branching)
- 16-bit MAC instructions

- integer arithmetic instructions (`CLAMPS`, `MAX`, `MIN`, `SEXT`, ...)
- VLIW facility, enabling the explicit issue of multiple instructions in parallel, both base instructions and defined by the vendor (TIE)

The HiFi 4 extension exists as a coprocessor to the base Xtensa LX processor, that can be included during configuration of the Xtensa IP. It exploits the aforementioned VLIW (FLIX) facility by enabling the programmer to program and issue certain instructions in parallel. Some of the instructions offered by the HiFi 4 DSP extension include: [3]

- circular buffer (modulo) loads and stores
- multiply and accumulate
- fixed-point arithmetic (round, truncate, saturate, multiply, divide, ...)
- floating-point arithmetic
- codebook loads and stores

In the case of the i.MX RT685 microcontroller, the HiFi 4 DSP coprocessor has been instantiated on an LX6 processor.

## 5.3 ABI

One of the configuration options enabled for the discussed instance of the Xtensa processor IP is the Windowed Register Option. This replaces the 16-entry general-purpose register file (AR) with a register file that has 64 registers instead. This option then imposes a register window onto this file, making 16 registers of this file available at any time and mapping them to original names of the AR file. The window is rotated on each subroutine call and return using the `CALLn` (PC-relative call) and `CALLXn` (call target, address in register). When a write access of a register belonging to the caller's window is made, a window overflow exception is raised and serviced, handler of which is responsible for saving the accessed value and rotating the window manually. Conversely, on subroutine return, the window underflow exception will be generated for restoring the overwritten values, if such accesses were made. [4] This enables high-performance subroutine invocations without the need for explicitly programming register store instructions for saving callee-saved registers and then restoring them upon a return from an invoked subroutine.

The Windowed Register Option works in conjunction with the Xtensa windowed ABI. First 8 registers of the available window are reserved for data usually found on the beginning of the stack frames - return address, subroutine arguments, stack frame pointer and return values. The rest of the registers are intended for the application and for subroutine calls. The fixed window variant rotates the window by 8 registers, the variable window variant can, depending on the needs of the programmer, rotate the register window by 4, 8 or 12 registers. Call parameters are provided before the call and the window rotation - first argument given to the subroutine is located in the `a2` register and, if using the fixed variant, should be provided in `a10` in the callee's window.

The Xtensa architecture also employs the CALL0 ABI, which doesn't work with any register windows and, instead, uses the default 16-register window as it is, with the scratch registers being callee-saved.

## 5.4 Available tools

Cadence offers a C/C++ retargetable compiler based around LLVM - `xt-clang`, together with a complete GNU toolchain (linker, assembler, debugger and adapted GNU binutils). That toolchain is entirely proprietary - source code is not released to the public and usage licenses are provided at Cadence's discretion. [1] Older versions of toolchain packages also included `xt-xcc` - Cadence's optimised GCC variant, however, current versions don't include that particular compiler anymore.

Standard GCC supports the Xtensa ISA as well and can be adapted to work with the different Xtensa profiles and extensions (such as the Fusion F1 DSP, HiFi 4 DSP and even vendor-specific TIE extensions). Support packages for the toolchain and Xtensa Xplorer available from Cadence contain, despite the access and license to those packages being restrictive, MIT-licensed source files that describe that particular core and can be just copied into the GCC source tree. This is further discussed in Section 7.7.

No study on compiler characteristics comparing `xt-clang` and GCC was found or done. Thus it is generally believed by NXP employees with whom this thesis was consulted, that the Xtensa toolchain is superior in the terms of size and execution performance of the compiler output. An internally discussed case with a prominent customer also discovered that the GCC's maths library for Xtensa is not complete and lacks some functions, making it unusable for DSP work and favouring `xt-clang`.

[1] http://tensilicatools.com

# Chapter 6

# The Sound Open Firmware layer

## 6.1 Overview

Sound Open Firmware (SOF) is a community developed audio framework covered by The Linux Foundation. It's a modular, pipeline-based audio framework centred around abstracting DSPs as a flexible audio processing domain controlled using messages exchanged over an IPC channel. The initial use case of this project was its use on Cadence HiFi DSP instances in desktop computer audio solutions manufactured by Intel, but it was quickly adapted for other targets, such as the i.MX series of application SoCs featuring the same DSPs or even ARM Cortex-A cores segregated by a hypervisor running in conjunction with Linux.

This project is closely tied, but not tightly coupled by the ALSA subsystem in Linux. The ALSA subsystem is capable of offloading tasks like volume control, stream mixing, equalisation, multiplexing and others. This is done by utilising an ASoC platform driver, that abstracts the audio processing domain as a whole and interprets topology configuration files generated beforehand, that describe the desired arrangement and configuration of defined processing blocks. [1] In case of SOF, the platform driver associated with it converts these topology blobs to a sequence of IPC messages, which then the SOF instance running on the audio domain interprets. As this is the only coupling between the host and the DSP and is inherently loose, making SOF functional with different hosts doesn't imply the need to replicate any of the facilities or mechanisms implemented in ALSA.

## 6.2 Programmer's model

From a programmer's perspective, the root object around which all work with SOF revolves is the *pipeline*. *Components*, processing blocks with a desired function, are created in the context of it. They're linked to each other using *buffers*, which have a defined size. Each component has a list of buffers attached to the *source* (receiving) and *sink* (producing) side of it, and between both of these lists the core function of a component (`comp_ops.copy`) takes place. Buffers in a pipeline are practically queues (ring buffers), which keep track of how much data was produced and consumed by adjacent components. If not enough data was produced (samples can't be dequeued) or the buffer is full (samples can't be enqueued), the copy function of a component attached to such buffer should fail. Each pipeline has a component considered as a *host* component, which serves as the entry point of the pipeline - a pipeline is triggered (started or stopped) by performing a trigger operation

on its host component. That component can be a true host component (`SOF_COMP_HOST`), which is responsible for exchanging audio data with the host, but any other accordingly implemented component can be used, such as the *tone* component (`SOF_COMP_TONE`), which function is to generate a sine wave.

Depending on the configuration, pipelines can be traversed in two ways. Playback pipelines (downstream direction) are traversed in the downstream direction, starting with the source (first) component and by following buffers in the sink buffers list, with the copy function being invoked before recursing. Capture pipelines (upstream direction) start with the sink (last) component and follow the source buffers, with the copy function being invoked after recursing. This scheme ensures that the copy function of a source component is always being invoked first.

Each instantiated object (pipeline, buffer or a component) is configured with a numerical ID, by which it is referenced in all operations. A pipeline is also configured with its scheduling priority, execution period (timer period at which the pipeline is traversed copy functions of its components are invoked), computing core index on which the pipeline will run, underrun or overrun reporting maximum limit and other parameters. The traversal can be scheduled upon a DMA interrupt, but this possibility is in the process of being deprecated, in favour of scheduling the traversal using a timer.

A component is instantiated on the basis of its ID, type index, pipeline ID and configuration specific to the particular component type. Some component types are identified not by a defined numerical index, but by an UUID. The instantiation is then carried out by formatting a component create message that references a special, virtual component type index (`SOF_COMP_ADAPTER`) and by extending the message with a UUID. A component creation message is then dispatched accordingly based on the specified UUID.

To establish a working pipeline which can be triggered to start, the pipeline object is created first. Then all other associated objects (components and buffers) are created. These are then connected together and a complete operation is performed on the pipeline. The host component is then configured with stream parameters (such as the sample rate, sample format, channel count, ...), which are then automatically distributed among other components in the pipeline. At last, the pipeline, as described above, can be started by triggering its host component, which starts repeatedly scheduling the function responsible for accordingly traversing the pipeline and invoking copy functions of all components. Conversely, the pipeline is stopped by triggering the host component to stop.

## 6.3 Architecture

SOF currently can be compiled against two different operating systems - Cadence's XTOS for Xtensa-family processors and the Zephyr RTOS. Because of this, one of the basic sections of SOF is an abstraction layer responsible for abstracting basic services (heap allocation, task management and scheduling, synchronisation, interrupt control, ...) of the underlying operating system.

As SOF was initially developed only against XTOS, which doesn't have any sort of a comprehensive device and driver model, SOF has its own driver model. Integral drivers (such as interrupt controllers, an IPC channel or timers) are implemented as plain functions written against common headers, which are then linked in by CMake configuration and the linker. Devices, of which multiple counts are expected, are described using device structures, which house attributes, private data and a function table, pointing to API functions implemented by a driver of each particular device. Lists of such device structures

are then provided by platform sources, which exist for each target that is supported by SOF. Among those device types are DMA controllers and audio interfaces.

In case of the Zephyr RTOS, a subjectively preferred operating system for new platforms, SOF drivers, which SOF is able to directly utilise, wrap around common Zephyr driver APIs. If SOF is built against Zephyr and Zephyr-native SOF drivers are enabled, the discussed device structures contain an additional field, `z_dev`, which points to the Zephyr's `device` structure. The platform sources then check for particularly named nodes in the utilised Zephyr device tree and statically populate the lists with those nodes. This enables SOF to utilise drivers that are already a part of the Zephyr RTOS.

Other than, of course, preemptive multitasking in the underlying RTOS, SOF has its own task scheduling system, that schedules and cooperatively executes different functions that are being scheduled. These include the tasks carrying out IPC operations (the handling of IPC messages is deferred outside the interrupt context), traversals of actual audio pipelines and other tasks which are needed to be executed periodically. The scheduler works as part of a *domain* object, which executes a selected scheduler as a consequence of a domain event. Every task is then checked if their execution is due (`domain_is_pending`) and marked as pending if so. SOF offers two domains - a timer domain and the DMA domain. Timer domain uses the system timer to run the scheduler and a task is determined if its execution is due based on the current timer value. The DMA domain differs in that it executes its handler function (to which the scheduler is attached) on DMA transfer completion interrupts. SOF also offers two schedulers - a simple scheduler based on a linked-list as a queue and a scheduler implementing the Earliest Deadline First (EDF) algorithm. The EDF scheduler is used in SOF only for IPC work.

## 6.4 Comparison with similar layers

Maestro is an openly licensed audio framework developed by NXP Semiconductors targeted towards this company's ARM microcontrollers. As of the time of writing this thesis, a publicly released version is implemented as a software library fully functional on top of FreeRTOS with a Zephyr port in progress. Its conceptual model is similar - it offers an environment where different audio processing blocks (sources, codecs, transforms, sinks, ...) can be connected together as a processing pipeline. However, unlike SOF, Maestro is closely tied to the developed application itself, as it doesn't utilise any RPC or IPC frameworks for exchanging control messages and audio data - merely a simple command queue is used as it is provided by the operating system and abstracted away with an API layer. It can be concluded that its use in AMP systems would require extra adaptation work, now making it impossible to use on high-end microcontroller devices with multiple cores. The possibilities of pipeline construction are greater in SOF, as pipelines can be linked together and the size of buffers associated with each processing block can be explicitly adjusted in the pipeline definition, allowing the use of, for example, crossovers. This use case proved problematic in Maestro. In addition, none of the Xtensa variants is among the set of supported targets, thus prohibiting its use on the HiFi 4 DSP and special instructions tailored for DSP work. However, judging from general observations, Maestro does not have such heavy memory requirements as SOF.

XAF is a proprietary audio framework developed by Cadence solely for Xtensa platforms. Like SOF and Maestro, it also offers a pipeline-based processing environment. Unlike Maestro, XAF completely exposes its function over an API accessible in the same domain in which it runs. Example projects in MCUXpresso SDK utilising XAF are running XAF on

the secondary DSP core of the given target device and use the RPMsg-Lite RPC component to exchange control messages and audio data. That access layer, though, is application-specific, as it's part of the application itself and deals with messages for audio data exchange and pipeline control (create, play, stop, ...) - the application does not view the abstracted DSP side as a complete audio processing framework.

# Chapter 7

# Porting the Zephyr RTOS

## 7.1 First steps

First steps were to properly instantiate the SoC and board layers in the Zephyr source tree. When the work on this porting task began (August 2023), it was discovered that a Google employee has submitted a pull request on Zephyr's upstream GitHub repository, which did that for the similarly situated Fusion F1 DSP in the i.MX RT595. That device is a close relative to the i.MX RT685 in the terms of hardware structure, overall characteristics and possible use cases. [20] The pull request provided a linker file, Kconfig definitions, default Kconfig configuration fragments, YAML board description, and modified layers for other Xtensa-enabled parts made by NXP to properly separate hardware differences. These changes don't result in a fully functional Zephyr port - no peripherals were instantiated, handling of interrupts from peripherals was not implemented and no code loading and run control facilities for the DSP core were implemented. With this port as it is, it is only possible to run a Zephyr application on the RT595 through a debugger and peripheral usage is up to the application, completely overriding the Zephyr's driver model. However, this pull request helped greatly in understanding Zephyr's structure and developing the port for the RT685. Contents of the pull request were studied, copied, names referencing the particular microcontroller were changed and, in the end, a new device tree was written.

The `soc/xtensa/nxp_adsp/rt6xx/include/_soc_inthandlers.h` file is the only file that was machine-generated. This was done using the `xtensa_intgen.py` script located in `arch/xtensa/core` of the Zephyr source tree. It generates the `_soc_inthandlers.h` file from the result of preprocessing `arch/xtensa/core/xtensa_intgen.tmpl` using a C preprocessor. That template file includes `core-isa.h` from the Xtensa core support package provided by Cadence. This file implements root interrupt handlers for all external interrupts that can be raised - these handlers infer the particular interrupt number based on the interrupt's controller level and mask and then transfer execution to a service routine registered for that particular IRQ.

## 7.2 Project structure

Because the microcontroller used in the considered scenario is essentially an AMP system, any project developed in that scenario must be structured to separately build images for the ARM core and the DSP. Given the position of the HiFi 4 DSP in the microcontroller, where the ARM core starts first and its intended responsibility is to initialise and start the DSP, such a project must be structured in a way where the code associated with the DSP must be located in a subproject and linked as a dependency for the primary project targeted towards the ARM core.

That was done using the CMake `ExternalProject_Add` call in the `CMakeLists.txt` of the primary project. To facilitate code loading, an extra build artefact was added to the subproject - binary images of data and text memory regions for the DSPs. These are created using the `xt-objcopy` tool, which copies them from the resulting `zephyr.elf` ELF image. Such files are then included as content in the `adspimgs.S` assembly source file of the parent project and exported as symbols.

## 7.3 DSP code loading and starting

The general flow of initialisation is to configure necessary blocks in the microcontroller's clock system, enable power to the DSP, enable its clock signals, copy program sections to associated memory devices and start the DSP.

Clock configuration was copied from the `dsp_xaf_usb_demo_cm33` example project located in the MCUXpresso SDK for the i.MX RT685. Power control and clock enablement is indirectly handled by the `fsl_dsp.c` driver, which the MCUXpresso SDK instance in the NXP Zephyr HAL provides and which is used directly in the port. Starting and stopping the DSP is done by directly accessing the `DSPSTALL` register in the `SYSCTL0` peripheral. Its C definition is also provided by the MCUXpresso SDK.

The part that required the deepest investigation in the process of starting the DSP is loading the code. First attempts to copy the DSP code simply utilised the `memcpy` function implemented by the picolib runtime library, which was invoked to copy the objects from wherever they were placed (in the particular instance where development took place the microcontroller's flash memory) to the DSP's TCMs. While the invocation of that function succeeded, in case of the ITCM the memory remained unchanged. An attempt to examine the memory's contents using the J-Link GDB server attached to the ARM core failed with the message that the particular section was not readable. A byte write invoked from the debugger, likewise, did not do anything. When a byte read from the application was attempted, a HardFault exception was raised. I discovered that it is possible to access the ITCM only with the granularity of 32 bits, given how the DSP's PIF is implemented and interfaced to the rest of the microcontroller, thus accesses of smaller sizes are illegal. A tailored memory copy routine was implemented that copies the code by 32 bits. This property was found experimentally - it was not mentioned in any documentation that NXP provides for the microcontroller.

The `drivers/misc/nxp_rtxxx_adsp_ctl/nxp_rtxxx_adsp_ctl.c` file located in the Zephyr tree ultimately integrates all of the said routines and implements a driver responsible for DSP initialisation, code copying, starting and stopping.

## 7.4 Enabling basic drivers

The GPIO and Flexcomm UART were chosen to be enabled at this stage. That way, blinking an LED and transmitting textual messages to a serial terminal becomes possible, easily assessing the system's state.

Because all peripherals on the APB bridges 0 and 1 are accessible from the DSP's address space, getting those peripherals to work didn't require any additional effort, despite their drivers being originally developed for the ARM core of the microcontroller. Only instantiating them properly in the DT, resolving unmet dependencies and appropriately modifying the port's Kconfig configuration fragments was required.

Those unmet dependencies were related to the NXP's Zephyr HAL, as its encompassing `mcux/CMakeLists.txt` did not include all the necessary include paths and source files related to used drivers and definitions in the MCUXpresso SDK, on which this HAL is based. Simple conditional inclusion sufficed.

## 7.5 Interrupt handling

As was discussed in Chapter 3.2, the microcontroller provides an additional multiplexer that, among other signals, routes interrupt lines from the device's peripherals to the HiFi 4 DSP core. There are 27 external interrupt lines exposed by the core and 38 available sources. The priority of each interrupt is selected by its number - different interrupt numbers can have different priorities. [18]

This presents a challenge while instantiating drivers of peripherals emitting interrupt requests, as not all interrupts can be mapped into the space of interrupts created by the DSP's interrupt controller. While prioritising peripherals most likely to be used by the DSP and employing a fixed allocation scheme is possible, it reduces a number of possible use cases of the DSP. Thus it was decided that the allocation of interrupts is best left to the developer - based on the project requirements, the developer can manually select needed interrupts, allocate them based on the priority desired and overlay the board's DT with the required definitions.

There was an effort of implementing an interrupt controller driver that would configure both the interrupt controller integral to the DSP and the interrupt multiplexer mentioned above. That collides with the way how Zephyr manages interrupts - it is possible to specify an interrupt controller DT node with required interrupts nodes bearing meaning specific to that particular interrupt controller driver, but it is not possible to statically traverse the DT for interrupt attributes related to that particular interrupt controller DT node. Because of that, a simpler and more expressive approach was chosen - a separate driver instantiated by a separate DT node responsible for just the multiplexer, or rather the INPUTMUX peripheral as a whole. That driver is implemented in `drivers/misc/nxp_inputmux/nxp_inputmux.c` of the Zephyr tree and provides a way to describe interrupt allocations in the DT, among other signal assignments by which the INPUTMUX peripheral can be controlled.

## 7.6 Testing and debugging

Testing was done by adapting, compiling, manually building the `hello_world`, `blinky`, `synchronization` and `philosophers` projects, running them on the board and observing

their effects. That way, basic functional tests of the instantiated drivers mentioned in Section 7.4 could be (and were) carried out. The mentioned adaptation step consisted of copying the project's code and definitions into a test project, where it existed as a subproject, build result of which is included in the image running on the ARM core.

Since SEGGER J-Link suite version V7.70, direct debugging of the DSP core is supported. [21] As a result of this, debugging was done using the J-Link GDB server both for the ARM core and the DSP. The `arm-none-eabi-gdb` and `xt-gdb` tools were used as well respectively. It is possible to debug code that is not first loaded by the ARM core by starting a debugging session with a desired ELF file and issuing the `load` GDB command. The clock and power subsystems have to be initialised first, otherwise just attaching to the DSP with the GDB server suffices.

`xt-ocd` is a GDB server providing an alternative way of debugging the DSP core, possessing the ability of debugging the CM33 core and the DSP simultaneously. This server can use the J-Link probe as a backend, thus not implying any extraneous hardware requirements for simultaneous debugging. This is very useful, for example, for debugging synchronisation issues, IPC problems and AMP systems generally. The steps are described in [16].

## 7.7 GCC toolchain porting

A GCC port (as provided by the Zephyr SDK described in Section 4.3) is required for a complete port, as the Cadence's toolchain containing `xt-clang`, however subjectively superior, is restrictively licensed and not even considered for testing work and CI (GitHub Actions) pipelines by the Zephyr committee.

Creating that port is, supposedly, a matter of acquiring the Zephyr SDK repository, compiling `crosstool-ng` located in it, creating an overlay directory containing files describing the Xtensa ISA profile and its specifics for `gcc`, `gdb`, `nanolib` and `picolib`, creating a configuration file and compiling the toolchain. For creating the overlay, steps described in [11] were followed.

The first attempt of porting the GCC toolchain was made with the sources in the `RI2021.8` package. This did not result in a functional toolchain, as the register map which is used by GDB was empty. (`gdb/xtensa-regmap.c` in the overlay package)

The second attempt was made with a newer Xtensa support package for the platform - `RI23.11`. Every tool in the toolchain did build, but the compiler itself produced an output that would crash during Zephyr boot. Furthermore, GDB was able to load an executable, but wasn't able to connect to the J-Link GDB server. Traffic inspection with Wire shark [1] revealed that the cause is an error response to a `g` packet, which serves as a *load all general registers* command issued by the client. [9] The server responded with an error code implying that it doesn't recognise that particular packet type. Manually disabling the logic for issuing that command in the GDB sources made the GDB client able to connect to the GDB server, but presented the user with register values of the debug target that were nonsensical and not in check with the target's true state. However, run control worked. It was later discovered during discussions about merging the work on GitHub that 2 extra parameters for GDB's build configuration step (`--enable-xtensa-use-target-regnum` and `--disable-xtensa-remote-g-packet`) were required to rectify both problems. Restoring

[1] https://www.wireshark.org/

GDB's sources to the original state and applying those parameters to the crosstool-NG overlay configuration produced a working GDB client for the target platform.

Despite the compiler producing insufficient output, it's still able to lexically, syntactically and semantically analyse input code. Consequently, together with the rest of the toolchain, it's still being able to function in CI workflows, where changed code is checked for build errors.

## 7.8 Audio input and output

The Flexcomm peripherals configured as I$^2$S interfaces handle audio data input and output on the RT685. As of this time, Zephyr has a subsystem in place for I$^2$S peripherals and even contains a driver (`i2s_mcux_flexcomm`) for the RT685's Flexcomm peripheral in the I$^2$S mode, but no codec configuration subsystem is implemented in Zephyr, thus leaving the problem of configuring codecs entirely up to the application developer. This presents a challenge as the codec integrated on the MIMXRT685-EVK board, the Wolfson WM8960, requires configuration to make any audio playback or capture possible and does not „just work" upon applying sufficient voltage to its power rail.

Zephyr demonstrates audio capture and playback using sample projects located in `samples/drivers/i2s/output` and `samples/drivers/i2s/echo` of the Zephyr source tree. Code on an experimental branch authored by Hake Huang implements a yet not merged codec configuration subsystem along with a driver for the WM8960 chip and strives to enable audio I/O on the MIMXRT595-EVK, the MIMXRT685-EVK's ancestor, using the same codec and the same Flexcomm peripheral, both driven from a Cortex-M33 core of each device. However, both boards were tested with code located on this branch, resulting in no audio signals being produced. Debugging revealed that the I$^2$S driver transitions to an erroneous state upon receiving an interrupt signalising an underflowed internal FIFO queue condition while driving a Flexcomm peripheral configured for playback (audio transmission).

After receiving a patch enabling playback on the i.MX RT595's Cortex-M33 domain, both of the samples were adapted to the i.MX RT685's Cortex-M33 domain. This involved the creation of overlay device trees, adding special nodes for the pin multiplexer of the device, setting up signal sharing for the Flexcomm peripherals used in the I$^2$S mode (in the `echo` example, one is used for transmission and the other one is used for capture - both peripherals share the `MCLK` signal) and modifying the sample rate - clock divider constraints from the codec side forbid using 44.1 kHz as the sample frequency. The MCLK signal generated by the Flexcomm I$^2$S peripheral is the same regardless of bitstream parameters and the final clock signal is generated by a divider in the codec, with a limited set of possible values. [6]

The application of described modifications and fixes led to a situation where capture and playback was fully functional, however, trying to adapt both of the examples to the DSP domain in a similar fashion resulted in the same failure as described above. The only major difference is that, as described in Chapter 3.3, a recommendation of using the 2nd DMA controller (`DMA1`) was followed, as also suggested in [14].

Further debugging of the `output` sample project revealed that the likely culprit exists between the DMA controller and the DSP. The DMA controller is used for transferring audio data between any of the portions of available system memory and the FIFO queue register exposed by the Flexcomm peripheral. The mentioned driver wraps the usage of that controller together with driving the Flexcomm peripheral itself. Upon receiving the

error interrupt (signalising the discussed FIFO underflow) from the Flexcomm peripheral, exposed registers of the employed DMA controller in the DSP domain were examined. The DMA controller completed the transfer (invalidated the DMA channel configuration register with an invalid flag) with no error flags observed as being set. It also likely raised an interrupt as a result of completing that transfer (flags signalising a yet not acknowledged interrupt were set), however, that interrupt was never serviced by the DSP, despite the INPUTMUX peripheral being correctly configured, the interrupt enabled and its service routing registered. As a likely result, no subsequent DMA transfer was programmed, leaving the Flexcomm's FIFO queue with no fresh data. As the Flexcomm peripheral progressed with the job of transmitting audio data, it drained the queue without it being replenished, subsequently causing the FIFO underflow condition.

The discussed condition was caused by the driver not enabling the peripheral's interrupt at all - the call responsible for enabling the configured interrupt in DT was omitted. Amending the driver initialisation routine enabled the servicing of IRQs generated by the DMA peripheral and, ultimately, enabled audio capture and playback.

## 7.9 Basic IPC

As was discussed in Chapter 3.4, the i.MX RT685 has the MU and SEMA42 peripherals assisting with inter-core communication.

The effort of enabling IPC began with building a testing application implementing a simple situation - the CM33 core posts a value over a chosen MU channel, the DSP receives it and prints it to the debug console. The `mbox_nxp_imx_mu.c` driver implementing the Zephyr `mbox` driver API was used for this. The test application was written and necessary changes in the SoC's and board's DT were made, as this driver wasn't yet instantiated in those trees. The first attempt to run the application resulted in the target's CM33 core deeply locking up - the J-Link GDB server reported the inability to stop program execution, run control became impossible and all registers were read with the hexadecimal value `0xDEADBEEF`. Investigation revealed that the driver did not clear the MU's reset signal emitted by the SYSCON peripheral in its initialisation routine. Upon adding adding code for clearing it, the application became functional as intended.

## 7.10 Memory layout

The linker file used for the target platform is parameterised from nodes in the device tree. This allows the application developer to change locations of defined memory segments easily without modifying the linker file or any of the files specific to the SoC or board layer.

At first, the ITCM and DTCM memory blocks directly attached to the DSP were used for storing its software. This worked right away in all yet executed scenarios (direct upload by debugger, boot from the CM33 core and boot from the CM33 core starting with a debugger attached), but these memories don't have enough capacity to hold more sophisticated applications, such as the ultimate goal of this thesis - an instance of Sound Open Firmware.

Ultimately, text and data segments were moved to the microcontroller's main SRAM, a 512 KiB block was reserved and divided into 256 KiB for code and data each by modifying the DT. The DSP expects a startup code (reset handler) in the ITCM, so this became the only item occupying it. This worked when directly loading the segments using the GDB `load` command and when the CM33 core booted from the „cold" state, but the DSP ended up attempting executing instructions at nonsensical locations when the CM33 core, with the function of booting the DSP, was started from a debugging session - the microcontroller was reset and the program loaded with the `load` command. This situation was remedied by resetting the microcontroller again after issuing the said command and writing the debugged image to the flash. The considered hypothesis is that the microcontroller has a builtin ROM that ultimately performs hardware initialisation either by itself or by executing DCD commands located in the stored image. Hardware that is uninitialised by bypassing the ROM and directly setting the CM33 core's PC register to the image entry point, which the J-Link GDB server does, can be the cause of the discussed illegal states.

# Chapter 8

# Porting the Sound Open Firmware layer

## 8.1 First steps

Porting Sound Open Firmware began with properly instantiating the platform layer for the target (`src/platform/imxrt685`), which provides a CMake file including all necessary platform-specific files, list of peripherals to be used directly by SOF and other code and definitions needed to abstract SOF from a given platform. Definitions for a HiFi4 DSP core on the i.MX 8 series of application SoCs (`src/platform/imx8`) were copied and modified to, for this phase, not include any devices in the device lists. After the compilation script (`scripts/xtensa-build-zephyr.py`) was modified to fit the build environment and the new platform, it was discovered that all of all of the memory segments were overflowing once the build system invoked the `xt-ld` linker. The application required about 8 MiB of text and data memory in total because of heap sizes, on which audio buffers are allocated. This was sufficiently reduced (`src/platform/imxrt685/include/platform/lib/memory.h`) and all optional features of SOF (components, algorithms, Zephyr facilities, ...) were excluded through the Kconfig mechanism. The resulting image almost fills up the entirety of the HiFi 4 DSP's ITCM, despite it being an application without any useful functions.

The next step involved constructing a minimal example of making SOF internals work - a basic pipeline with a null source, volume control component and a null sink. While this example does not interact with any real hardware and does not implement any user-perceivable function, it's a testing ground that doesn't involve any drivers for external peripherals and potential problems while driving them. Code that instantiated a static pipeline by directly invoking RPC handlers was discovered, studied and backported to the working copy. A single pipeline with a tone source, volume control and a final buffer was hard coded - SOF does not have a null (dummy) sink component. Debugging of that situation revealed a mismatch of sizes of IPC messages - a check of an IPC message size field was present and that field was not filled out by the macros in the committed code.

After SOF was instantiated without any optional facilities, those features were gradually reenabled. Of course, the capacity of the DSP's ITCM was hit, so the previously constructed target device tree was reconfigured to use a portion of the main system SRAM both for code and data. The ITCM is then used only for the startup code. Those features include DMA support, timer domain, host components and other components needed to construct a basic pipeline. Zephyr wrapper drivers adhering to the SOF driver model were used, as

they enable the use of already existing Zephyr drivers in SOF without the need to adapt them in any way. The result was a pipeline that produced a single period of a 32-bit little-endian 48 kHz sine wave into a buffer. In addition, the use of Zephyr-native drivers (`CONFIG_ZEPHYR_NATIVE_DRIVERS`) was enabled, as it's advantageous to utilise already implemented Zephyr drivers for peripherals in the targeted platform.

## 8.2 IPC

Making IPC work in the SOF port required developing a tailored IPC transport layer, whose function is to transport messages between the two processing cores. This layer uses the `mbox` Zephyr API (implemented by the `mbox_nxp_imx_mu` driver) and shared memory. It's synchronous (a message can be transferred in the same direction only after it has been acknowledged by the recipient) and full-duplex (2 of the 4 MU channels are utilised - one for transmitting and the other one for receiving messages from one core's point of view).

When a message is to be sent, the message is copied into a memory location reserved as shared memory for IPC and its address is posted into the mailbox. The mailbox then rises an interrupt request aimed at the recipient core, which proceeds with servicing that request and, consequently, handling that message. Once the message is handled, the mailbox is then triggered in the signalling mode (a plain IPI is raised) towards the sender, signifying that the message was handled and concluding the transaction.

On top of this transport layer, a simple host layer was built for the purposes of interfacing with SOF from the CM33 domain. This layer just provides means to easily construct and send messages to the DSP core. It also ensures that message send calls are synchronous (blocking), as it blocks execution until sent messages are acknowledged.

This layer was tested by transmitting test messages from both sides. Upon the SOF start procedure finishing, a `SOF_IPC_FW_READY` message is sent to the CM33 domain. Reception of this message is waited upon before sending any more messages to the DSP. The CM33 host can send a `SOF_IPC_GLB_TEST | SOF_IPC_TEST_IPC_FLOOD` message, which will be acted upon by just acknowledging it in the fashion described above.

At first, it was determined that no address translation was necessary, as [18] says, that the main SRAM is accessible from the DSP at the same address at which it's mapped to the CM33 core, omitting memory cache. The data written from the CM33 core, however, wasn't visible upon delivering the address through the MU and reading it. The mechanism started to transmit messages successfully after translating addresses, counter-intuitively, from the CM33 core to the DSP with cached accesses. This was found out experimentally and the subjectively probable cause is inaccurate documentation. If this interpretation is correct, this is the second manifestation of NXP's documentation shortcomings.

During implementation and debugging efforts concerning this section, the debugging setup employing the J-Link GDB server was no longer sufficient, as it can't debug the two cores, the CM33 core and the DSP, simultaneously. Instead of using that, the CM33 core was debugged with the J-Link GDB server as usual, but the DSP was debugged with the `xt-ocd` server, as was briefly described in Section 7.6. It became the only option to examine the state and behaviour of both domains that became sequentially bound to each other by IPC, as the act of detaching from the CM33 core at a desired point and attaching to the DSP caused the DSP to fetch illegal instructions upon performing instruction steps, driving the DSP domain into an illegal state and ultimately crashing it.

## 8.3 Audio data exchange between domains

The attempt of enabling audio data exchange started with creating the `host` component in the pipeline, which function is to, depending on its configuration, provide audio data to or from the host domain. As the Zephyr-native SOF drivers were enabled, a Zephyr variant of the host component (`host-zephyr.c`) is used. This variant uses the `dma` API for facilitating data transfers between domains. The host buffer description is provided as the `SOF_IPC_GLB_STREAM | SOF_IPC_STREAM_PCM_PARAMS` IPC message, body of which is the `sof_ipc_pcm_params` struct. All components classified as host components, though, are configured using this message type.

As this component was inserted into the pipeline, it logged that no DMA controllers were available. This was fixed by conditionally adding a new item populated from the device tree to a DMA controller list located in SOF platform sources specific to Zephyr. As the component was written with systems utilising virtual memory (page tables) in mind, a new function that transforms the host buffer description into an array of scatter-gather DMA transfer descriptors was written (`ipc_process_flat_host_buffer`). Because the i.MX RT685 is a microcontroller with a flat memory model with no concept of virtual memory and paging, this function generates just a single descriptor, that contains the host buffer in its entirety.

In addition, the `dma_mcux_lpc` DMA driver, which is relevant for this platform, had to be modified. As the device's memory is flat, there's no concept of host and local memory, but SOF operates with those concepts. The support for `HOST_TO_MEMORY` and `MEMORY_TO_HOST` DMA channel directions (`dma_channel_direction` enumeration type) was added as an alias to the `MEMORY_TO_MEMORY` direction. Also because the driver's internal data structure (`dma_mcux_lpc_dma_data` struct) did not contain a valid header for this structure (`dma_context` struct), the functions for requesting and releasing channels provided by Zephyr (`dma_request_channel`, `dma_release_channel`) were failing, so that header was added and properly initialised.

A host buffer was provided in the demonstration application and the function was tested with a capture pipeline consisting of the `tone` component and the `host` component in order, with necessary buffers also in place. The `host` component was now invoking the `dma` API and the selected driver (`dma_mcux_lpc` in this case), but no useful DMA activity was observed. As the pipeline was traversed, few of the first bytes in the host buffer were overwritten with zero bytes. Other traversals failed to trigger any other DMA transfers, either if using one-shot transfers (`host_copy_one_shot`) or transfers with the descriptor array being continually modified (`host_copy_normal`). At first, the data memory cache was suspected, but was ruled out by using addresses that map to the main SRAM without a cache. The effort wasn't continued and was abandoned here.

## 8.4 Audio input and output

As was discussed in Section 4.6, Zephyr offers 2 APIs for audio interfaces - the `i2s` and `dai` APIs. The latter one, `dai`, is the one consumed by a Zephyr variant of the `dai` component (`dai-zephyr.c`). Because of this, the driver responsible for driving Flexcomm peripherals in the I$^2$S mode (`i2s_mcux_flexcomm.c`) can't be used directly. As no implementation work was done on this matter, following ways are considered for establishing audio inputs and outputs to and from SOF pipelines:

- bridge the two APIs with an adaptor driver - this would impose extra latency and processor time requirements, as the audio data would have to be copied one more time, in addition the `dai` API doesn't offer a way to keep and transmit information about the positions in buffers, ring buffer overruns and underruns could cause irregularities in audio captures or transmissions (missing or overwritten data, repeated portions)

- write and test a new driver for the Flexcomm peripheral in the I$^2$S mode conforming to the `dai` API - this could raise concerns about code duplication and, because of its sheer scale, is far from trivial

- write and test a new SOF component that enables audio input and output over peripherals driven by drivers conforming to the `i2s` API - code duplication would also be an issue and the act of instantiating such a component would deviate from the established set of IPC messages - such a component would have to be referenced by an UUID, as extending the set of default component types would present a greater deviation from the set

- extend the existing `dai-zephyr.c` component to enable the use of `i2s` drivers - this is subjectively considered as the path of least resistance, while it would also present a deviation from the already established set of IPC messages, the deviation could be just an addition of an extraneous member of the `sof_ipc_dai_type` struct, which, observing by its already existing members, is extended regularly by adding support for new platforms

# Chapter 9

# Test application

## 9.1 Overview

As part of efforts described in Chapter 8, a test application (`host_rt685`) was created, both as an ultimate goal of this thesis and as a development vehicle for SOF. This application uses the i.MX RT685 as an AMP system, where SOF runs in the DSP domain and is controlled over an implemented IPC channel described in section 8.2 and a minimal host layer. It includes SOF as a build dependency in the same fashion as described in Section 7.2 and uses the `nxp_rtxxx_adsp_ctl` driver for bootstrapping the DSP described in Section 7.3.

This host layer consists of an IPC transport and a set of routines responsible for formatting messages. In essence, the IPC transport is a reflection of the IPC layer implemented as part of the SOF port. It's responsible for transferring messages and acknowledgements between domains and ensures that the communication is synchronous from the host side - it uses a Zephyr semaphore to block execution until a message acknowledge condition is observed. The formatting routines provide a programmer-friendly way to construct IPC messages and do ensure, within reason, that the constructed IPC messages are valid. As described in Section 8.2, it uses the device's MU peripheral to signal IPIs and 32-bit integers between the two domains - such integers are used as message addresses. Because of the difference between cached and non-cached accesses, it takes care of address translation as well.

The test application then uses this host layer as a way to instantiate a pipeline containing a tone component with a host component connected to its sink side with a buffer. It is then possible to read out that buffer using an attached debugger and observe a generated sine wave.

## 9.2 Memory usage

Memory usage of the `host_rt685` application, both static and runtime, was studied. Table 9.1 shows the usage of different segments that is statically known for the DSP domain, Table 9.2 shows the same for the whole image and Table 9.3 shows the runtime heap usage in the DSP domain. The total heap capacity is statically allocated as part of the data segment located in the main SRAM. Debug build of the application was used.

| Init code (ITCM) | Data (SRAM) | Text (SRAM) |
|---|---|---|
| 273 bytes | $\sim$ 210 KiB | $\sim$ 140 KiB |

Table 9.1: Static memory usage for the DSP domain

| Flash | SRAM |
|---|---|
| $\sim$ 386 KiB | $\sim$ 8 KiB |

Table 9.2: Static memory usage of the whole application

| Idle (IPC initialised) | Pipeline created | Heap capacity |
|---|---|---|
| 1644 bytes | 6288 bytes | 16 KiB |

Table 9.3: Runtime heap usage in the DSP domain

The size and location of the memory segments used by both domains is statically determined by the device tree of each domain, so its modification should be performed in order to change how the device's main SRAM is split between the domains. SOF's heap size can be modified in `src/platform/imxrt685/include/platform/lib/memory.h`.

# Chapter 10

# Conclusion

The Zephyr RTOS was successfully ported to the target platform and the function of all of its integral facilities (context switching, synchronisation, memory management, ...) was tested, along with drivers for basic peripherals (UART, GPIO) and peripherals responsible for audio playback and capture (I3C, Flexcomm in the $I^2S$ mode, DMA). Facilities for using the i.MX RT685 microcontroller as an AMP system (control driver for the DSP, CMake definitions for project linking) were created. Porting of the GCC toolchain to the target was also attempted, albeit with only partial success.

The SOF layer has been ported as well and its core facilities (pipeline creation, management, execution, timing, ...) proved functional. An IPC channel, used for exchanging control messages between the CM33 domain and the DSP, was implemented along with a simple host layer and is able to control the SOF instance as far as tasks revolving around pipeline creation and run control go. A test application (appendix C), was written with the use of this layer.

The test application is able to generate a single period of a sine wave into a pipeline buffer. This product had to be adjusted from the original idea of an audio player, that processes audio data in the DSP domain with SOF, which was initially considered together with this thesis' supervisor. The obstacles to the said original idea are the inability to exchange audio data with the host (Section 8.3) and the difference of audio APIs used in various parts of the resulting application (Section 8.4).

This deviation was caused by underestimating the practical difficulty of the assignment - Zephyr itself and the associated components are under active development. As support for NXP devices seems to be primarily focused on ARM cores and the support of HiFi 4 DSP cores was attempted only on i.MX application SoCs so far in conjunction with the SOF driver model, the effort of porting Zephyr by itself and its drivers to the target platform created edge cases that were, subjectively, far from trivial to investigate and solve. The fact that the documentation for the i.MX RT685 is lacking in discussed respects also contributed to the said deviation. Consequently, less time remained for matters concerning SOF itself. Its documentation is also lacking - the work became dependent on colloquially communicated knowledge from peers and reverse engineering efforts.

The completion of the SOF port is the natural direction in which this thesis can be continued. One could also develop applications with components not used in this thesis or try to implement their own with the accelerating facilities of the DSP core, such as yet not integrated audio codecs. As of the time of writing this thesis, all the work performed here is in the process of upstreaming - merging with originating code repositories. It will remain open source.

# Bibliography


[1] ALSA project. *ALSA topology* online. 2024. Available at: https://www.alsa-project.org/wiki/ALSA_topology. [cit. 3 May 2024].

[2] Arm Ltd. *Arm Cortex-M33 Devices Generic User Guide* online. Available at: https://developer.arm.com/documentation/100235/latest/. [cit. 25 Jan 2024].

[3] Cadence Design Systems Inc. *HiFi 4 DSP User's Guide*. 2021. Proprietary - available at author's discretion.

[4] Cadence Design Systems Inc. *Xtensa Instruction Set Architecture (ISA) Summary* online. 2022. Available at: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/ip/tensilica-ip/isa-summary.pdf.

[5] Cirrus Logic Inc. *CS42448 Product Data Sheet* online. 2017. Available at: https://statics.cirrus.com/pubs/proDatasheet/CS42448_F5.pdf. [cit. 18 Jan 2024].

[6] Cirrus Logic Inc. *WM8904 Product Datasheet* online. 2018. Available at: https://statics.cirrus.com/pubs/proDatasheet/WM8904_Rev4.1.pdf. [cit. 22 Apr 2024].

[7] Espressif Systems. *ESP32-S3 Series Datasheet* online. 2023. Available at: https://www.espressif.com/sites/default/files/documentation/esp32-s3_datasheet_en.pdf. [cit. 20 Nov 2023].

[8] Fidan'in, F. Garmin Fenix 7X Solar Teardown (Non Destructive). *F Tipi Blog* online, 2022. Available at: http://www.f-blog.info/garmin-fenix-7x-solar-teardown-non-destructive/. [cit. 18 Jan 2024].

[9] GDB developers. *GDB: The GNU Project Debugger* online. Available at: https://sourceware.org/gdb/current/. [cit. 17 Jan 2024].

[10] Leibson, S. *Designing SOCs with Configured Cores: Unleashing the Tensilica Xtensa and Diamond Cores*. Morgan Kaufmann Publishers, 2006. Electronics & Electrical. ISBN 9780123724984. Available at: https://books.google.cz/books?id=h79BlQEACAAJ.

[11] Linux/Xtensa Wiki. *Toolchain Overlay File* online. 2017. Available at: https://wiki.linux-xtensa.org/index.php?title=Toolchain_Overlay_File. [cit. 20 Nov 2023].

[12] NXP Semiconductors B.V. *i.MX RT600 Audio Evaluation Kit* online. Available at: https://www.nxp.com/design/design-center/development-boards/i-mx-evaluation-

and-development-boards/i-mx-rt600-audio-evaluation-kit:MIMXRT685-AUD-EVK. [cit. 18 Jan 2024].

[13] NXP Semiconductors B.V. *i.MX RT600 Evaluation Kit* online. Available at: https://www.nxp.com/design/design-center/development-boards/i-mx-evaluation-and-development-boards/i-mx-rt600-evaluation-kit:MIMXRT685-EVK. [cit. 18 Jan 2024].

[14] NXP Semiconductors B.V. *AN12749: I2S(Inter-IC Sound Bus) Transmit and Receive on RT600 HiFi4* online. 2020. Available at: https://www.nxp.com/docs/en/application-note/AN12749.pdf. [cit. 18 Mar 2024].

[15] NXP Semiconductors B.V. *I.MX RT500 and i.MX RT600 Crossover MCUs - Fact Sheet* online. 2020. Available at: https://www.nxp.com/docs/en/fact-sheet/IMXRT500RT600FS.pdf. [cit. 26 Jan 2024].

[16] NXP Semiconductors B.V. *Getting started with Xplorer for EVK-MIMXRT595* online. 2021. Available at: https://www.nxp.com/docs/en/supporting-information/GSXEVKMIMXRT595.pdf. [cit. 20 Nov 2023].

[17] NXP Semiconductors B.V. NXP i.MX RT MCU Technology Powers Our Smartwatch Future. *Smarter World Blog* online, 2022. Available at: https://www.nxp.com/company/blog/nxp-i-mx-rt-mcu-technology-powers-our-smartwatch-future:BL-NXP-IMX-RT-MCU-TECHNOLOGY. [cit. 26 Jan 2024].

[18] NXP Semiconductors B.V. *UM11147 - RT6xx User manual* online. Sep 2022. Proprietary - available at author's discretion.

[19] NXP Semiconductors B.V. *UM11732 - I²S bus specification* online. 2022. Available at: https://www.nxp.com/docs/en/user-manual/UM11732.pdf. [cit. 8 Apr 2024].

[20] NXP Semiconductors B.V. *i.MX RT500 Low-Power Crossover MCU Reference Manual.* Jan 2023. [cit. 18 Jan 2024]. Proprietary - available at author's discretion.

[21] SEGGER Microcontroller GmbH. *Release notes for the J-Link / Flasher Software and Documentation Package* online. Available at: https://www.segger.com/downloads/jlink/ReleaseNotes_JLink.html. [cit. 19 Nov 2023].

[22] Zephyr Project members and contributors. *Zephyr Project Documentation* online. Available at: https://docs.zephyrproject.org/latest/index.html. [cit. 5 May 2024].

# Appendix A

# Optical media contents

The included optical disc bears the following contents:

```
/
├── thesis/
│   ├── xstani07-bthesis.pdf .......... This thesis text rendered as a PDF file.
│   └── xstani07-bthesis-src.zip ..... LaTeX sources of this thesis.
└── code/
    ├── zephyr.7z ...................... Archived West workspace Zephyr RTOS with
    │   │                                 dependencies. (appendix B)
    │   ├── zephyr/ ..................... Ported Zephyr RTOS. (chapter 7)
    │   ├── sdk-ng/ ..................... Zephyr SDK with the attempted toolchain
    │   │                                 port. (section 7.7)
    │   ├── apps
    │   │   ├── test_rt685_amp_blink .... AMP test application - „Hello World“ with
    │   │   │                             a blinking LED and button control with a
    │   │   │                             GPIO interrupt.
    │   │   ├── test_rt685_amp_mbox ..... AMP test application demonstrating the use
    │   │   │                             of the MU peripheral.
    │   │   ├── test_rt685_amp_output ... Demonstrates audio playback on the DSP do-
    │   │   │                             main.
    │   │   └── test_rt685_amp_echo ..... Demonstrates audio playback and capture.
    │   └── modules/
    │       └── hal/
    │           ├── nxp/ ................ Modified NXP HAL.
    │           └── xtensa/ ............. Modified Xtensa HAL.
    └── sof.7z ......................... Archived West workspace containing SOF
        │                                 with dependencies. (appendix C)
        ├── sof/ ........................ SOF itself. (chapter 8)
        └── apps/
            └── host_rt685 .............. Test application with the host layer.
                                          (chapter 9)
```

# Appendix B

# The Zephyr RTOS port

The Zephyr RTOS port is recorded on the included optical disc as a compressed West workspace with Git repositories checked out at necessary revisions - `code/zephyr.zip`. The `zephyr` (`/zephyr`), `hal_nxp` (`/modules/hal/nxp/`), `hal_xtensa` (`/modules/hal/xtensa`) and `sdk-ng` (`/sdk-ng`) repositories are of interest. The repositories contain all of the done changes up to the point of making audio playback work (section 7.8), retargeting program segments (section 7.10) and trying to adapt the GCC toolchain. The different changes are separated into commits to track progress. Revisions relevant to this thesis are:

- `zephyr`: `vitstanicek-nxp/rt685support-i2s`
- `hal_nxp`: `vitstanicek-nxp/rt685support-provisional`
- `hal_xtensa`: `4f3293cbb79b9d210c0fe0a4b238417043c5438b` (work upstreamed)
- `sdk_ng`: `vitstanicek-nxp/rt685support`

The code is also available online on personal repositories hosted on GitHub, however, as further development of all codebases is expected, it may change outside the scope of this thesis as it was turned in and published. [1]

Zephyr prerequisites (Zephyr SDK, Python, West, CMake, Ninja, device tree compiler, ...) need to be installed on the target machine to build and debug this code, as well as the SEGGER J-Link suite and the Cadence Xtensa toolchain with a support package for the i.MX RT685's HiFi 4 DSP instance. This thesis worked with the `RI-2021.8-win32` toolchain and the `nxp_rt600_RI2021_8_newlib` core, but it's likely that newer versions of both will also provide a satisfactory result. Using a GNU/Linux-based environment is highly recommended, but it's possible to build this instance on Windows under the MSYS2 environment. Using WSL2 on Windows also should be possible, but wasn't tested.

[1] https://github.com/VitekST/zephyr, https://github.com/VitekST/hal_nxp, https://github.com/VitekST/sdk_ng

To build applications directly for the DSP domain, specify these environment variables and use the `west build . -b nxp_adsp_rt685` command:

- `ZEPHYR_TOOLCHAIN_VARIANT=xt-clang`
- `XTENSA_CORE=<xtensa_core_name>`
- `XTENSA_TOOLCHAIN_PATH=<path_to_XtDevTools/install/tools/>`
- `TOOLCHAIN_VER=<toolchain_version>`

These (`<project_dir>/build/zephyr/zephyr.elf`) then can be launched using the `xt-gdb` debugger from the Xtensa toolchain and either the J-Link GDB server or the `xt-ocd` server. However, the DSP domain needs to be initialised first by the CM33 core (section 7.3), so it's possible to run and debug those examples only with code utilising the `nxp_rtxxx_adsp_ctl` driver in place, preferably not sequentially bound. Complete platform reset is desirable before the CM33 core booting the DSP and launching them. These example projects were tested this way:

- `samples/hello_world`
- `samples/philosophers`
- `samples/basic/blinky`

The `/apps` directory of the `zephyr.zip` archive also contains the `test_rt685_hybrid` application, which provides a minimal example of an AMP system - an example where an image for the DSP domain is built as an dependency for the CM33 domain and the CM33 core boots the DSP. To build and test, omit the `ZEPHYR_TOOLCHAIN_VARIANT` environment variable from the list above and build like described.

# Appendix C

# Sound Open Firmware port

Likewise, a compressed West workspace containing everything related to Sound Open Firmware is included on the disc - as `code/sof.zip`. Repositories which are of interest include the repositories discussed in appendix B and the `sof` repository (`/sof`) at the branch `vitstanicek-nxp/rt685support`.

Standalone SOF image can be built with the `sof/scripts/xtensa-build-zephyr.py` script. The toolchain version and core name are hardcoded in it, it just needs to be invoked with the Xtensa toolchain root path set as an environment variable and with the target name passed as an argument. Use this command to do so:

```
XTENSA_TOOLS_ROOT=<path_to_XtDevTools> python
    sof/scripts/xtensa-build-zephyr.py -p imxrt685
```

The `apps/host_rt685/` directory contains an AMP project that targets the CM33 domain and builds the SOF as a dependency for booting the DSP with it. It also contains the host IPC layer for controlling SOF from the CM33 domain, which it demonstrates with a simple pipeline, which is then briefly triggered and stopped. Build it in the same fashion as the `test_rt685_hybrid` example described in appendix B.