WHITEPAPER

From Hardware Concept to Zephyr Bring Up—

The Road to Using a Crossover MCU with Zephyr

Eli Hughes, NXP Pro Support Engineer

Contents

1. Synopsis

The intent of this article is to illustrate a hardware design that uses a “crossover” microcontroller (MCU) and the steps needed to bring-up Zephyr RTOS on the custom hardware. I purposely chose a specialized component that represents a slightly more challenging design task. I thought it would be instructive to show a path going from concept to hardware design through booting the RTOS. I do want to point out that this article will dive deep into both hardware and software as a system; when stakeholders have their feet planted on both sides of the hardware/software fence, a team can often do more with less. In all successful projects, understanding the design choices context is key to success.

Over the course of my career, I have had the privilege to observe the complexity of microcontrollers grow orders of magnitude. It certainly has been a wild ride since I was first experimenting with a 6502 as the number of peripherals, memories, and additional co-processing functions now integrated is mindblowing. Managing complexity in a design, both in the hardware and firmware domains, is paramount to “getting things done”. Simply coming up with a way to manage your build system and the software components can be a full-time job as you start dealing with more complicated devices. With the introduction of “Crossover MCUs” such as the i.MX RT family from NXP, the demarcation between MCU, Digital Signal Processor (DSP) and application processor is no longer clear, and managing system complexity can be an overwhelming task with the amount of resources in a device.

There are applications where one may desire to directly program specialized hardware elements in an MCU to achieve maximum performance and flexibility. However, having tools for the common/general-purpose functions can really assist with managing the overall system complexity, and will pay dividends. I have always used my hometown street layout as a useful analogy when considering software tools. With a small population of 1000 people, a single stop sign in the center of town can be sufficient for managing traffic flow. However, as population increases by orders of magnitude to 10k or 100k citizens, it would be unwise to not have systems in place to manage the traffic.

Likewise, when engineering any complex microcontroller system, an RTOS can be your “traffic lights” and “highways” to ensure the overall system behaves well as you add complexity. Even if you don’t use the RTOS for everything in the system, it can be very helpful with managing general-purpose functions so you can focus on the specialized components. One aspect of the Zephyr RTOS that I find helpful is that you can use as little or as much of the ecosystem as you like. I have found that this aspect gives me a great deal of flexibility when I am actively figuring out how to manage system complexity as I can laser focus on the pieces that are most important.

2. “Crossing Over” with i.MX RT685

To set the stage, I want to spend some time examining an interesting device that represents the quintessential blurring of MCU, DSP and applications processor. I will spend some time pointing out some interesting aspects of this device as there are elements that are new in the microcontroller realm. There is quite a bit of detail here, but I want to make a case that this new class of MCUs demand consideration when planning the management of your software system complexity.

It was October of 2017 when the concept of the “Crossover” MCU was first introduced with the NXP i.MX RT1050. For a microcontroller & DSP enthusiast such as myself, the concept of a high clock rate MCU that could tackle problems previously relegated to application processors was very intriguing. I generally approach problems from the perspective of “simplicity”. For many real-world processing challenges, microcontrollers can be the simplest solution. There are applications however that demand more real-time processing capability on continuous streams of data. Whether it be high channel count audio, or complicated sensor fusion, there are situations where you need more than what a traditional microcontroller can offer. In some cases, one may choose to go down the path of an applications processor, but this approach is not always optimal when low latency, real-time response is a critical requirement.

The typical method to approach this problem was to use a “Digital Signal Processor” (DSP). DSPs generally have an internal processing pipeline highly tuned to real-time sample by sample data processing. DSPs are great solutions, but you often must give up many of the general-purpose features found in a microcontroller. I encounter this scenario quite often and it is not uncommon to pair a general-purpose microcontroller with a dedicated DSP in an application architecture to get the best of both worlds. My first experience with a DSP architecture was with the Motorola 56K series. It was a powerful tool for crunching numbers, but I found that most of my designs with a DSP also required a general-purpose microcontroller for the traditional IO and connectivity.

This is where the i.MX RT685 steps in. The i.MX RT685 crossover MCU is a newer member of the i.MX crossover family focused on real-time number crunching applications such as audio, sensor fusion and machine learning.

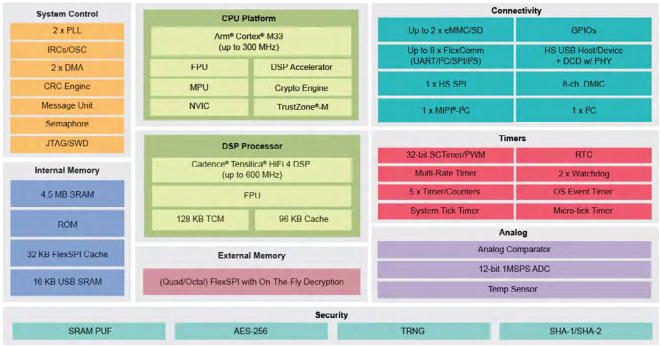


FIGURE 1. THE i.MX RT685.

The i.MX RT685 addresses the audio and sensor fusion challenge by integrating a high-performance 300MHz general purpose microcontroller with a powerful 600MHz DSP processor, a large 4.5MB SRAM bank and a plethora of traditional peripherals & IO.

The i.MX RT685 General Purpose CPU Platform

The CPU platform in the i.MX RT685 is based upon the Arm® Cortex®-M33 core (CM33). By itself, the CM33 is capable of sophisticated audio applications. The CM33 is built on the Armv8-M architecture which includes **Single Instruction on Multiple**

Data (SIMD) as well as **Multiply and Accumulate (MAC)** instructions. There is quite a bit that could be accomplished with CM33 running at 300MHz before even considering the additional DSP core. Last year, I wrote quite a bit about the LPC5500 series MCUs focusing on the LPC55S69 and its ample processing capabilities. The LPC55S69 is also based upon the CM33 core (running at 150MHz vs 300MHz). I think of the i.MX RT685 as a serious upgrade to the LPC55 when you need more of **everything**.

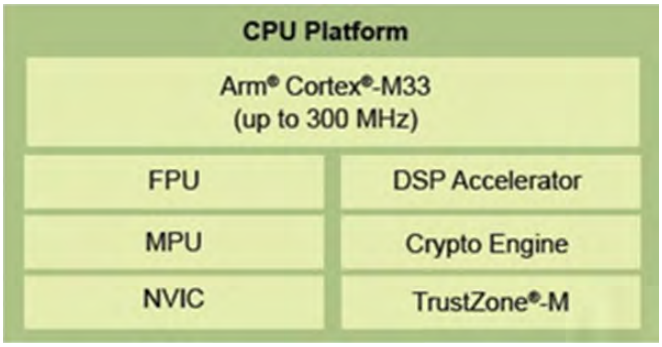


FIGURE 2. THE i.MX RT685 GENERAL PURPOSE CPU PLATFORM.

One feature of the CM33 is a coprocessor interface which can be accessed with special assembly language instructions. There are two "co-processors" attached to the CM33 in the i.MX RT685: The PowerQuad (labeled as the DSP Accelerator) and the CASPER (labeled as the Crypto Engine).

The PowerQuad Co-Processor

The PowerQuad is a dedicated hardware unit that runs in parallel to the CM33 core inside the i.MX RT685. By using the PowerQuad to work in parallel to the CM33, it is possible to implement sophisticated signal processing algorithms while leaving your general purpose CM33 core available to do other tasks such as communication and IO.

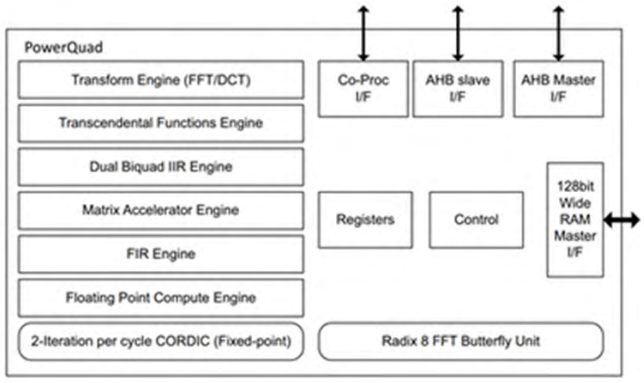


FIGURE 3. THE i.MX RT685 POWERQUAD CO-PROCESSOR.

For the LPC55S69 release, I wrote four articles on the PowerQuad and detailed some of its use cases:

- PowerQuad Part 1 – Industrial IOT, OFDM Communications and Smart Metering
- PowerQuad Part 2 – Digital IIR Filtering
- PowerQuad Part 3 – Fast Fourier Transforms
- PowerQuad Part 4 – Matrix and Vector Processing

The key takeaway here is that before we have even considered the additional DSP core, the i.MX RT685 offers a powerful hardware co-processor that can do very useful operations "out of the box". We can be crunching Fast Fourier Transforms (FFTs) at a high rate before even tapping into any of the resources of the DSP or CM33 cores!

CASPER

CASPER is an accelerator attached to the CM33 coprocessor interface that is optimized for cryptographic computations. At its core, CASPER is a dual multiply-accumulate-shift engine that can operate on large blocks of data. Applications of CASPER include accelerating cryptographic functions such as public key verification (e.g., TLS/SSL) and computing HMAC signatures. Once again, before we have even considered using the Tensilica© HiFi4 DSP in the i.MX RT685, there is

another accelerator in the i.MX RT685 that can off-load complicated operations. Many connected products require multiple cryptographic operations and CASPER is a great way of implementing the functions without taxing your other processing pipelines. There are plenty of examples included in the MCUXpresso SDK for utilizing the CASPER accelerator. This feature makes the i.MX RT685 well suited to IOT applications.

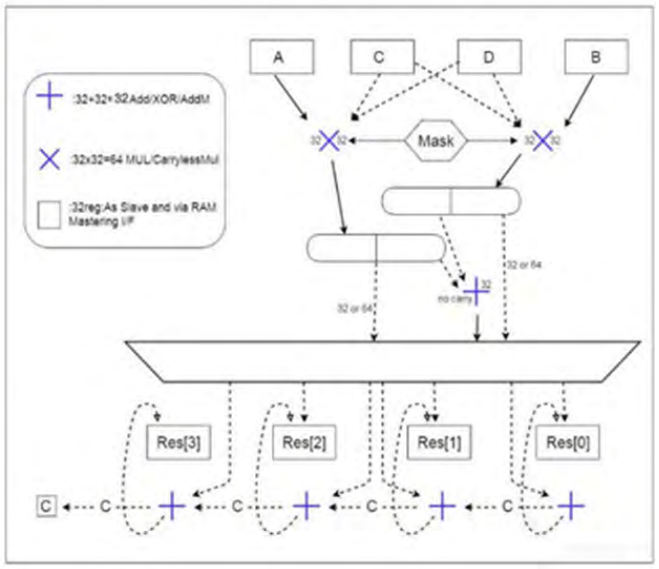


FIGURE 4. THE i.MX RT685 CASPER CRYPTOGRAPHIC ACCELERATOR.

Cadence Tensilica® HiFi 4 DSP

What makes the i.MX RT685 really interesting is the inclusion of a Cadence Tensilica® HiFi4 DSP core. I previously mentioned that it is certainly possible to implement DSP in the general purpose CM33. There are situations however where a dedicated DSP processing pipeline is needed to achieve a required throughput. One of the limitations of the Cortex™-M core is that several cycles can be used just initializing registers before utilizing the SIMD/DSP instructions. In many cases, the SIMD/MAC instructions can execute in a single cycle, but several CPU cycles are required for general purpose registers loaded with input data. Dedicated DSP processors are optimized to allow for continuous processing of single cycle MAC operations using features such as circular indexed memory modes and zero overhead loops. The HiFi4 DSP supports four 32x32-bit MACs and the ability to issue two 64-bit loads per cycle. There is a vector floating point unit providing up to four single-precision IEEE floating point MACs per cycle. All HiFi4 operations can be used as intrinsics in standard C code.

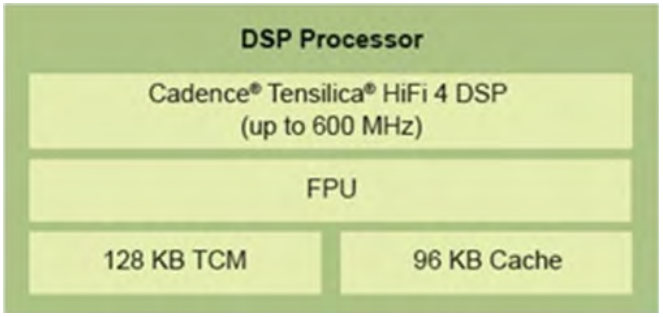


FIGURE 5. CADENCE TENSILICA® HIFI 4 DSP IN THE i.MX RT685.

The HiFi4 Audio DSP was designed specifically for audio and sensor fusion processing pipelines. It is supported with a large 3rd party ecosystem that covers applications such as sensor fusion, real-time audio, noise reduction, sound enhancement and voice processing. There are more than 300 DSP software packages already ported and optimized for the HiFi4 DSP architecture. This means you can get up and running very quickly, and can easily port your own proprietary software, completely in C, while also maintaining or surpassing the performance of assembly on other DSPs. Cadence even offers an optimized version of TensorFlow Lite to enable machine learning and AI applications at the edge.

i.MX RT685 Memory Architecture

Another unique aspect of the i.MX RT685 is its large memory availability and architecture. An important component of an audio processing architecture is the availability of large blocks of memory for time/sample history buffers and fast memories for critical code execution.

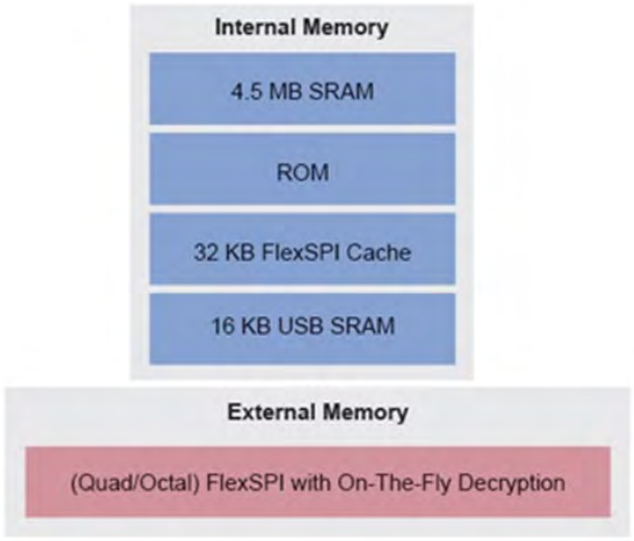


FIGURE 6. i.MX RT685 INTERNAL MEMORY.

The availability of 4.5MB of internal SRAM immediately should catch your attention! I tend to think in terms of real-time audio applications for musical performance. 4.5MB of RAM allows for deep buffers to implement delay-based effects including loopers and large time constant reverbs. A 4.5MB pool of fast SRAM removes the need for slower external SDRAM (which is common in many audio DSP architectures). The 4.5MB is shared between the CM33 and the HiFi4 DSP. The memory is sufficiently partitioned to allow for a large amount of flexibility in the processing architecture. There are **30 partitions** across **9 AHB** ports. This means the processing system can be designed to minimize contention between the CPUs and RAM allowing for maximum throughput.

The HiFi4 DSP has dedicated local Tightly Coupled Memories (TCMs) for data and code. Each TCM is 64 KB accessed by a **128-bit port**. The code and data TCMs can be accessed by the Cortex-M33 and by the DMA controllers through a slave port on the AHB matrix. These connections allow the CM33 to bootstrap the HiFi4 with executable code. In addition to the TCMs, there is a dedicated 4-way data cache of 64 KB with 256 bytes per line and a dedicated 4-way instruction cache of 32 KB with 256 bytes per line. The local HiFi4 memory architecture enables the highest level of processing capability as the DSP engine can access code and data with minimal bottle neck.

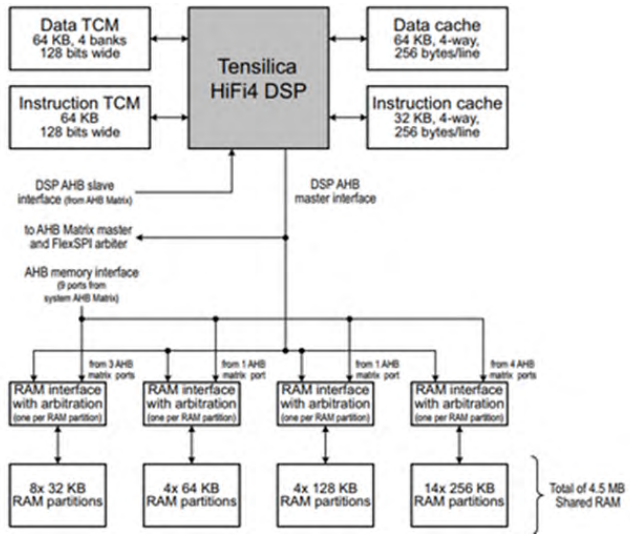


FIGURE 7. HIFI4 LOCAL MEMORY ARCHITECTURE.

Like many of the other i.MX RT crossover parts, the i.MX RT685 is a flash-less component. This allows the i.MX RT685 to be efficiently built on 28nm FD-SOI semiconductor process technology taking advantage of power consumption savings and clock frequency improvements. Code can be stored in low-cost external Quad/Octo SPI NOR Flash memory. Non-time-critical routines can execute in place from external memory while code requiring better performance can execute from internal SRAM. This approach gives architecture maximum flexibility in balancing cost and performance. It was almost 10 years ago when the NXP LPC4357 was introduced with a QSPI XIP flash interface. The XIP external flash approach has proven to be an effective way to lower the total cost of a solution that can provide large amounts of flash for applications as well as offering flexibility when architecting the MCU solution.

Inter-processor Communications and Synchronization

Since there are multiple CPU cores in the i.MX RT685, it is important to have hardware support for Inter-Processor Communication (IPC). The i.MX RT685 includes a Messaging Unit (MU) which provides a hardware-based IPC mechanism. While it is possible to set up shared memory between the CM33 and the Hifi4 DSP, the MU offers a way to efficiently send messages/notifications between the processors with interrupt support. The MU is an important component to allow one CPU to wake up another when using power-down modes.

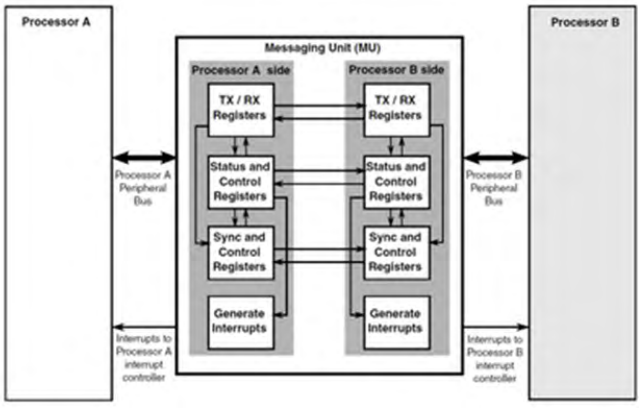


FIGURE 8. THE i.MX RT685 MESSAGING UNIT.

In addition to the MU, the i.MX RT685 includes a hardware enforced semaphore function. Both the CM33 and Hifi4 DSP have access to internal peripherals and memories. It is important to have mechanisms in place to ensure appropriate mutual exclusion. The semaphore unit implements 16 hardware enforced "gates". A processor must write a special sequence to gain access to a hardware gate and to release its lock. The gates are generic in nature and can be used as needed by the software architecture to implement mutual exclusion on both peripherals and memory. Hardware support in the i.MX RT685 for IPC and semaphores make multicore processing simpler to architecture and implement.

i.MX RT685 Audio Peripherals

Processing audio streams is a key feature of the i.MX RT685. A proper audio "Crossover Processor" would not be complete without hardware peripheral support for common audio IO interfaces. The bread and butter of digital audio is the I2S protocol. I2S is the gateway into a large ecosystem of high-quality external data converters, sample rate converters and audio transmitters (AES3/SPDIF). The i.MX RT685 includes eight multi-function "Flexcomm" serial peripherals. Flexcomm peripheral channels are reconfigurable for all the common serial protocols including USART, SPI, and I2S. Each of the Flexcomm interfaces support four I2S channel pairs for a potential of 32 channel pairs available to the system. All the common digital audio modes are supported in the Flexcomm I2S peripheral including Left justified, Right Justified and TDM (Time Division Multiplexing) modes. The Flexcomm I2S peripheral includes an eight entry FIFO to ensure glitch-less audio streams.

For voice applications, the i.MX RT685 contains a flexible Digital Microphone (DMIC) subsystem. DMICs most commonly use **Pulse Density Modulation PDM** to encode audio data. A DMIC has no analog output, data is output digitally synchronous to a clock. The clock is supplied by the DMIC subsystem, and its frequency is at a binary multiple (e.g., 64x) of the audio sample rate. DMICs are an extremely popular replacement to older electret microphone technologies as they are small, can be built on a repeatable semiconductor process and have a direct digital interface. As an example, **Knowles Acoustics** manufactures DMICs for applications including voice and ultrasonic sensing.

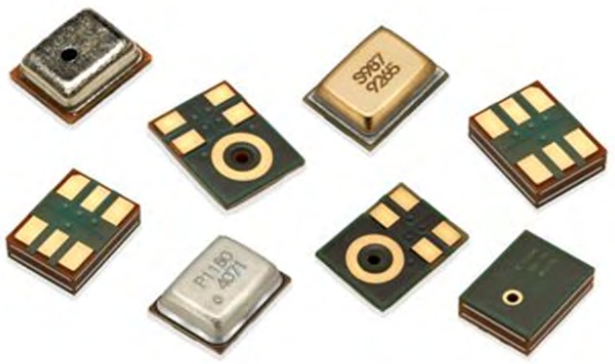


FIGURE 9. KNOWLES ACOUSTIC DIGITAL MICROPHONES.

PDM data streams require a digital filter and decimation to recover the audio waveform. The DMIC subsystem in the i.MX RT685 has the necessary hardware to directly connect and decode PDM data streams. Up to eight microphones are supported with flexibility over decimation and sample rate control. Processing an acoustic array of microphones is now much simpler with the i.MX RT685!

Also included in the DMIC subsystem is a hardware voice activity detector (HWVAD). The HWVAD is a dynamic envelope detector that can be used to trigger /wakeup processing functions when activity is detected in a digital audio stream.

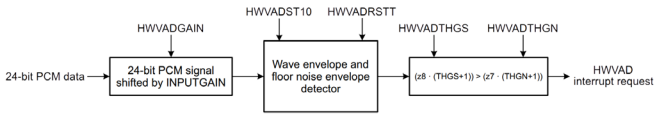


FIGURE 10: i.MX RT685 HARDWARE BASED VOICE ACTIVITY DETECTOR.

One last important component that I want to mention (that is important to audio applications) is a dedicated Phased Locked Loop (PLL). Audio IO protocols require a dedicated master clock that is at some binary multiple of the target sample rate. When using common audio sample rates such as 44.1KHz or 48KHz, the master clock may not easily be derived from an internal MCU clock source. For example, it is common to observe a 12.288MHz or 24.576MHz master lock when working with 48KHz audio streams. The i.MX RT685 includes a dedicated PLL for audio applications.

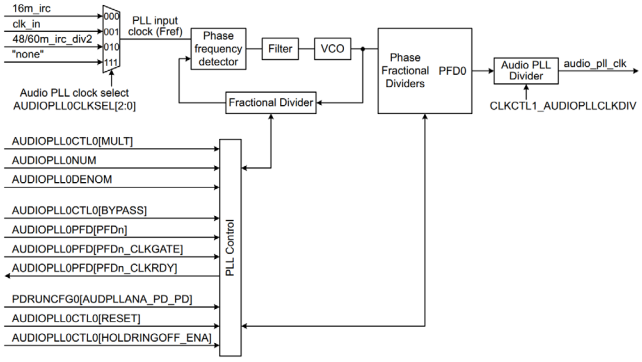


FIGURE 11. i.MX RT685 AUDIO PLL.

General Purpose Connectivity and Timers

With all the dedicated hardware for audio and sensor fusion workloads, there is still a great deal of support in the i.MX RT685 for general purpose connectivity, timing, and analog integration.

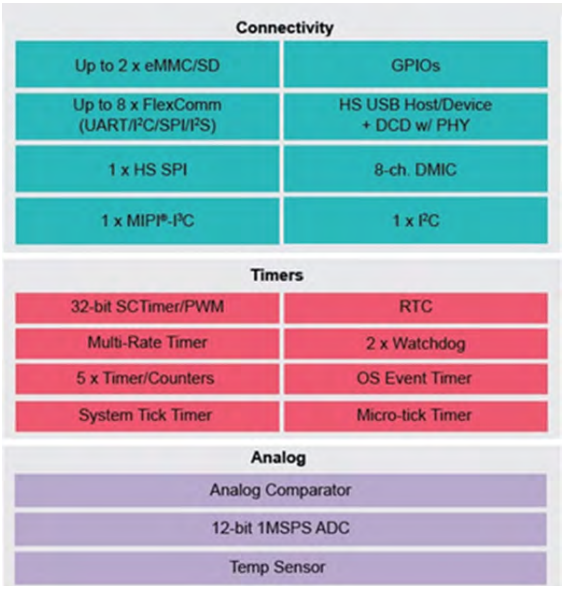


FIGURE 12. i.MX RT685 CONNECTIVITY, TIMERS AND ANALOG INTEGRATION.

The i.MX RT685 has all the standard connectivity support you would expect including two SD/eMMC interfaces and high-speed USB. One aspect of NXP microcontrollers that I love are the plethora of timers! Two of my favorites being the State Configurable Timer (SCT) and the Multi-Rate Timer (MRT). I have previously written about interesting applications such as **ultrasonic pulse pattern generation** with the SCT and **Modbus communication with an RS485 enabled UART and the MRT**. The built-in analog system includes a 1MSPS ADC and analog comparator which can allow the i.MX RT685 to be used into interesting industrial applications that require sensor fusion.

I3C

One last unique feature that I want to point out in the i.MX RT685 is the addition of a MIPI® I3C interface. I3C is a super set of the classic I2C bus. I3C was developed by the MIPI alliance to provide an upgrade to the I2C for mid-speed applications. It is positioned as an alternative to SPI while keeping a simple two wire interface between devices on a PCB. It is widely expected that I3C will be a standard interface on many new sensors and peripheral components. Some notable features:

- Up to 12MHz clock rate. SDA and SCL lines use both open-drain and push-pull modes to increase data rate and allow bi-directional communications
- "In Band Interrupts". A peripheral can interrupt a controller over the bus without extra pins.
- Multi-Controller/Multi-Drop
- Double data rate modes that offer transfer speed on parity with classic SPI
- Hot Joins. Nodes can join the bus at any time. Nodes get notifications when a new device joins the bus.
- Backwards compatibility with I2C.
- Dynamic Addressing
- In-Band Common Command Codes to standardize behaviors.

I3C is looking very cool and offers a unique blend of I2C and SPI capabilities. Be sure to get informed on I3C as you will see a lot more of it in years to come.

I hope this overview of the i.MX RT685 was able to get you interested in the crossover processor concept and its unique capabilities. My background in acoustics, audio and sensor fusion always steer my interests to parts with a diverse mix of capabilities. From my perspective, it is one of the most interesting MCU platforms currently available. I was only able to hit on the highlights and I hope you can find you way to the **i.MX RT685 website to learn more.**

3. The Minimal Configuration i.MX RT685 Hardware Project. Codename “Super-Monkey”

When starting a new design, the most prudent path is to purchase an EVK for a new MCU. Zephyr has support for the i.MX RT685 EVK in the mainline repository. Since the i.MX RT685 is more complicated than most microcontrollers, I did want to show a custom design that deviated from the EVK. This design serves as an example of the steps needed to get Zephyr running on a custom board. The i.MX RT685 requires a bit more care and feeding to get a project up and running and I thought it would be good to have an example available for others to use. The part is offered in a 0.5mm pitch VFBGA176 so I also thought it would be good to show a minimal configuration example that can be built with low-cost PCB technologies.

I find this step to be a very useful exercise as high-end MCU’s can be overwhelming, especially to those coming from a traditional MCU background. The goal here is to develop a simple “minimal configuration” example and build it for a demonstration. There are resources available to helping you design with the i.MX RT685 such as the “Hardware Development Guide for the RT685 Processor” (RT685HDUG) and the MIMXRT685-EVK. These resources can help bootstrap your next design. My personal view is that having several different design perspectives is always beneficial. Between the existing reference material and this article there should be enough information to start your next i.MX RT685 design with confidence.

In 2020, I did similar project codenamed “Mini-Monkey” using the LPC55S69 in its VFBGA98 package. It was an exercise to illustrate a simple project using the 0.5mm VFBGA98 package on a low cost 2-layer PCB process. That hardware project demonstrated the LPC55S69 being able to do things like microphone capture, visualization and animated GIF decoding.

The i.MX RT685 is definitely a “step up” from the LPC55S69 and is well suited to high end audio applications. Between the 300MHz CM33, the PowerQuad coprocessor and the 600MHz tensilica HIFI4 DSP, there is quite a bit of horsepower for your application! It was obvious to me that the i.MX RT685 could be a great fit for all my future real-time DSP audio processing projects. In 2011, I built the “Active Pickguard” demonstrating what could be accomplished with the Kinetis K20 (Cortex-M4) device.

10 years later there have been some serious advances in embedded technologies. Rev 2 of the Active pickguard has been in the back of my mind for a while now and the i.MX RT685 is a potential great fit!

To run some experiments with the i.MX RT685 in-situ, I want to design a small module that bootstraps the i.MX RT685 with everything I need for my high-end audio applications. Since I may go through several iterations, I thought a simple module (like the Mini-Monkey) would be a good start point. To get the project “out the door”, it is important to set clear boundaries at the outset. I do not need the module to bring out all the features of the i.MX RT685. The initial goal is to get the most critical IO to get my audio projects moving. Iteration is very important in the engineering process so I thought I would keep it simple to keep momentum going.

Since the .MX RT685 is a significant step up in performance from the LPC55S69 base Mini-Monkey, I thought the codename “Super-Monkey” would be a good fit. It leaves me with options for future projects (hint hint!). Ultra-Monkey, Mega-Monkey, Nano-Monkey….

My Initial Specs for the i.MX RT685 Super-Monkey Module:

- IO breakout for 8 digital microphones (DMICs)
- One TDM input for up to 8-channel audio in (one flexcomm channel)
- Two separate I2S audio output channels. One will be for outputting audio and the other will be a special monitor (two flexcomm channels)
- I3C Access. This is a new feature I have been wanting to experiment with
- Solder pads to add an optional TFT display.
- SWD debug access
- USB connector for ISP boot and power
- A debug UART
- A handful of GPIO
- +5v power via USB or IO pins
- On board Flash (the i.MX RT685 is a flash-less part)
- Crystal/clock management

i.MX RT685 PCB Layout Consideration: The VFBGA176 Package

The first order of business when I approach a new hardware design is to understand the device package and any implications of the pin geometries. For the Super-Monkey project, I will be designing with the MIMXRT685SFVKB which uses the **VFBGA176** package. The VFBGA176 is a 9mm x 9mm ball grid array (BGA) with 0.5mm pitch. A fine pitch BGA package can be intimidating, especially for those who may only have experience with microcontrollers in QFP packages. One of the reasons I built the **“Mini-Monkey”** was to demonstrate that the LPC55S69 VFBGA98 package option was doable with a 2-Layer process and low-cost design rules. The VFBGA176 was designed to be easy to fanout and is routable with a practical 4-layer process.

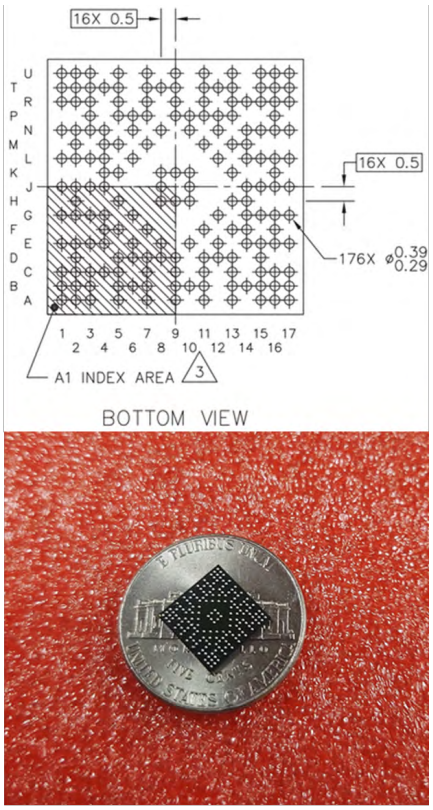


FIGURE 13. THE VFBGA176 PACKAGE.

I enjoy studying unique BGA ball arrangements. This package is a work of art. Notice that there are several regions left unpopulated making it simpler to fan out. In many cases, a 0.5mm BGA requires a filled micro via-in-pad to fanout the IO. There was quite a bit of thought put into the pinout of the i.MX RT685 and it is possible to fan this device out without via-in-pad technology. Many of the balls on the interior are VSS connections. Using the MCUXpresso IDE pin tool, one can highlight pins and get a rough sense of the layout strategy.

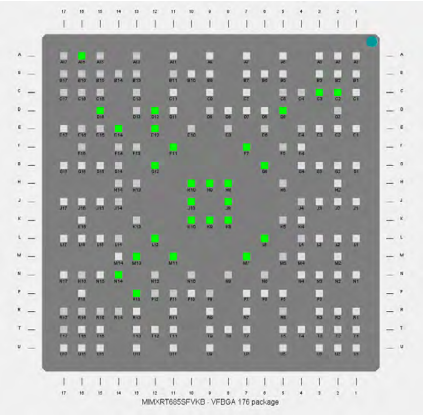


FIGURE 14. SIGNAL HIGHLIGHTING WITH THE MCUXPRESSO IDE PIN TOOL

I spent some time analyzing the Gerber files from i.MX RT685EVK to get a feel for the layout. The fanout of this package is very practical.

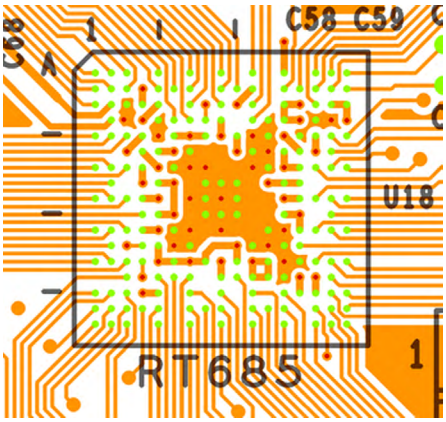


FIGURE 15. i.MX RT685 VFBGA176 FANOUT

Notice that all the signal pads can be fanned out on the top layer. The via connections (red dots) are to connect to the power & return planes on inner layers. The vias are placed in the depopulated ball regions. The IO fanout on the RT685EVK uses 3mil width/ 3mil space geometry. The vias under the device package use 6mil drills. These geometries are a bit tighter than what is typically required by traditional microcontrollers, but it is well within the capabilities of many PCB fab houses.

Since the only vias used in the fanout are for power and returns, there is plenty of space to place decoupling capacitors on the opposite side of the PCB. Figure 16 illustrates the placement of the decoupling caps on the RT685EVK. They are nearby existing vias to direct feed the power planes.

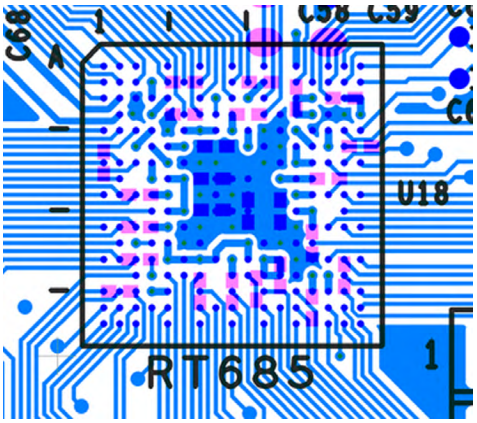


FIGURE 16. RT686 VFBGA176 DECOUPLING CAPACITOR STRATEGY

Many 0.5mm pitch BGAs require via-in-pad technology along with blind/buried stack-ups so the fanout vias do not interfere with the decoupling capacitors on the opposite side of the PCB. In the case of the VFBGA176, we can use lower cost processes as we do not need via-in-pad technology or a blind/buried stackup. Fanout and decoupling capacitor placement is straightforward for the i.MX RT685 VFBGA176 package. I really like to understand package geometry before I start a design as it can directly influence other decisions down the pipeline. The time spent analyzing

the Gerbers on the RT685EVK was well spent as I now have confidence the fanout of the part will not require any exotic PCB process technology.

i.MX RT685 Power

Once I study the packaging for a new MCU, the next order of business is to understand the power supply architecture. MCU power supply design is intertwined with PCB layout strategy, so it is a good idea to understand both early in the design process.

The i.MX RT685 crossover MCU power/performance ratio sits at an intersection of traditional MCUs, dedicated DSPs and application processors. To get significant clock rate improvements over traditional MCUs, the i.MX RT685 was built on [28nm FD-SOI process technology]. Using 28nm process technology for an MCU is a new concept that NXP has helped pioneer.

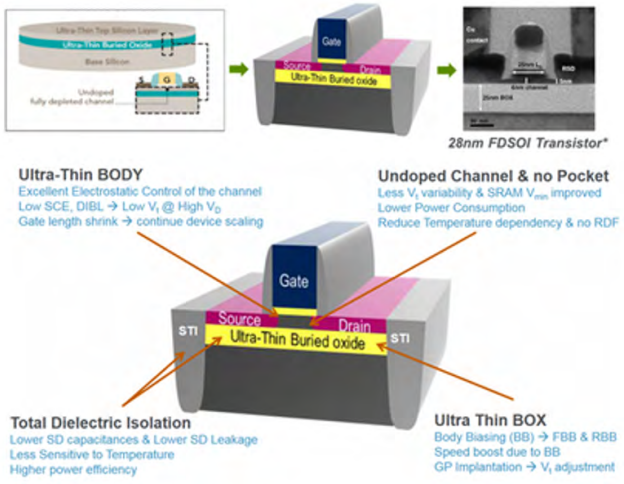


FIGURE 17. 28NM FD-SOI PROCESS TECHNOLOGY.

If you remember your electrical engineering coursework, all the major characteristics of a MOSFET are controlled by geometry. FD-SOI process technology has many advantages, one being that we can bias the body of the MOSFETs dynamically. Body biasing gives us real-time programmatic control over the MOSFET threshold voltage VTH. For MCU applications, we can move VTH up or down allowing one to increase clock frequency/performance or reduce dynamic power consumption.

From the perspective of production yield and a desire for increased CPU clock rates, the geometries of the transistors need to be pushed to their design rule minimums. A byproduct of small transistor geometries is a numerically small value for MOSFET VDS. For 28nm FD-SOI process technology, the nominal MOSFET VDS has been characterized to 1.0v nominal. This means the "core" logic power must be derived from a 1.0v supply.

Why are many MCUs powered at +3.3v?

Even though 28nm FD-SOI is 1.0v nominal, one can imagine the maximum allowed VDS to also scale up with geometry. You might not be aware, but many +3.3v MCUs might still have cores that need +1.2v, 1.8v or 2.5v core power because of the underlying process technology. The MCU will usually incorporate an LDO that the end user may not be aware of. Sometimes this internal LDO is documented in the device datasheet, but it might not always be the case.

As the core voltage requirements are pushed lower, the internal LDO approach can get more inefficient resulting in additional power dissipation in the IC package. The core logic is one of the largest consumers of dynamic power resulting from all the MOSFETS switching at a high clock frequency. As the core logic voltage requirement drops, it is important to have direct access to the core power domain to directly supply power from a switching DC-DC converter. As an example, the NXP LPC55S69 is built on 40nm process technology and uses a 1.2v core. It integrates its own switching DC-DC converter to minimize active power dissipation.

Understanding the i.MX RT685 Power Supply Strategy

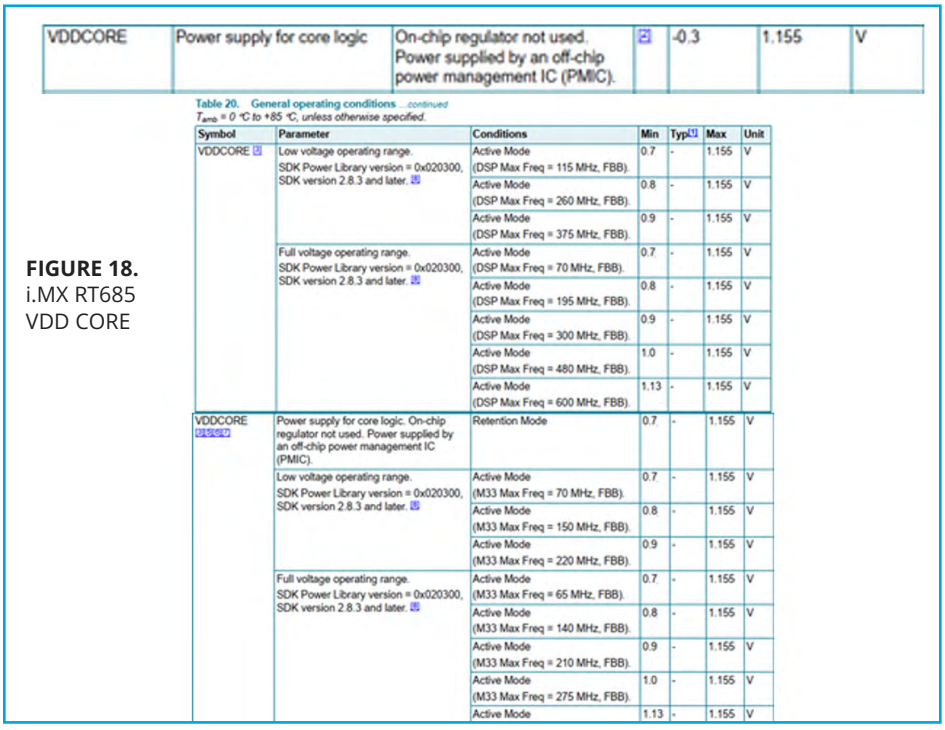
The i.MX RT685 has three general power domains:

- +1.0v Core
- +1.8V for "Always-On" and Analog Functions
- +1.71 to +3.6v capable IO (which can be further divided into 3 distinct groups)

i.MX RT685 Core Power

VDDCORE is used to power the internal CPU(s) and core logic. To maximize clock rate, the CPUs and core logic are constructed from the smallest transistors allowed by the process technology design rules. In the case of the i.MX RT685 and 28nm FD-SOI, the core voltage is nominally 1.0v. Looking at the i.MX RT685 datasheet, the maximum allowed value does not give one much room for error in your power supply!

It is possible to operate the core power at values lower than 1.0v as to save power when using lower clock frequencies. NXP provides a power library which can tune the body bias to achieve low power consumption figures at lower clock rates.



VDDCORE	Power supply for core logic	On-chip regulator not used. Power supplied by an off-chip power management IC (PMIC).	[2]	-0.3	1.155	V

Table 20. General operating conditions ...continued
T_{amb} = 0 °C to +85 °C, unless otherwise specified.

Symbol	Parameter	Conditions	Min	Typ[1]	Max	Unit
VDDCORE [2]	Low voltage operating range. SDK Power Library version = 0x020300, SDK version 2.8.3 and later. [5]	Active Mode (DSP Max Freq = 115 MHz, FBB).	0.7	-	1.155	V
		Active Mode (DSP Max Freq = 260 MHz, FBB).	0.8	-	1.155	V
		Active Mode (DSP Max Freq = 375 MHz, FBB).	0.9	-	1.155	V
	Full voltage operating range. SDK Power Library version = 0x020300, SDK version 2.8.3 and later. [5]	Active Mode (DSP Max Freq = 70 MHz, FBB).	0.7	-	1.155	V
		Active Mode (DSP Max Freq = 195 MHz, FBB).	0.8	-	1.155	V
		Active Mode (DSP Max Freq = 300 MHz, FBB).	0.9	-	1.155	V
		Active Mode (DSP Max Freq = 480 MHz, FBB).	1.0	-	1.155	V
		Active Mode (DSP Max Freq = 600 MHz, FBB).	1.13	-	1.155	V
VDDCORE [2][3][4]	Power supply for core logic. On-chip regulator not used. Power supplied by an off-chip power management IC (PMIC).	Retention Mode	0.7	-	1.155	V
	Low voltage operating range. SDK Power Library version = 0x020300, SDK version 2.8.3 and later. [5]	Active Mode (M33 Max Freq = 70 MHz, FBB).	0.7	-	1.155	V
		Active Mode (M33 Max Freq = 150 MHz, FBB).	0.8	-	1.155	V
		Active Mode (M33 Max Freq = 220 MHz, FBB).	0.9	-	1.155	V
	Full voltage operating range. SDK Power Library version = 0x020300, SDK version 2.8.3 and later. [5]	Active Mode (M33 Max Freq = 65 MHz, FBB).	0.7	-	1.155	V
		Active Mode (M33 Max Freq = 140 MHz, FBB).	0.8	-	1.155	V
		Active Mode (M33 Max Freq = 210 MHz, FBB).	0.9	-	1.155	V
		Active Mode (M33 Max Freq = 275 MHz, FBB).	1.0	-	1.155	V
		Active Mode	1.13	-	1.155	V

FIGURE 18. i.MX RT685 VDD CORE

The take-away here is that by providing a separate core power supply access, the designer has flexibility in how to optimize for a particular use case. It is important to note that the i.MX RT685 includes an LDO that can be powered from a +1.8v rail. Use of the LDO is not required but is available if the efficiency of a switching power supply is not required.

+1.8v "Always-On" and Analog Functions

There are several 1.8V rails in the i.MX RT685 that may be combined if needed. One of these 1.8v rails is the "always-on" power domain. This domain is used for features that must be active in power down states such as the RTC, reset, optional LDO and PMIC control. In addition to the +1.8v "always-on" domain, there are +1.8v rails for other analog functions such as the ADC and comparator. +1.8v power is required for the i.MX RT685 in a minimal configuration scenario.

User IO

Lastly, the IO pins can be powered separately from the +1.8v functions and the core. The IO supply range is +1.71 to +3.6v with +3.3v being the most common. There are other circumstances however when you may want certain IO pins to use +1.8v. For example, many high-speed double data rate quad/octo NOR flash devices operate with a +1.8v power supply. The i.MX RT685 has 3 separate IO banks so you can have a mix of IO voltages. In a minimal configuration scenario, it would be possible to power all the IO from +1.8v reducing the number of DC-DC converters needed in a system. However, it is most likely one will require some IO at +3.3v.

Power Supply for pins

Pin	GPIO pins
VDDIO_0	PIO0_0 to PIO0_13 PIO1_11 to PIO1_29 PIO2_12 to PIO2_23 PIO3_25 to PIO3_31 PIO4_0 to PIO4_10 PIO7_24 to PIO7_31
VDDIO_1	PIO0_14 to PIO0_31 PIO1_0 to PIO1_10 PIO2_24 to PIO2_31 PIO3_0 to PIO3_24 PMIC_I2C_SCL PMIC_I2C_SDA

FIGURE 19. i.MX RT685 POWER DOMAIN PIN ASSIGNMENTS

It is important to understand all the IO power options at the outset of a design. It is easy to make a mistake and end up with a non-functional part design. For my own designs, I take extra steps in the schematic entry stage to ensure success. I like to break up the schematic symbol for a part based upon a particular voltage domain. This makes design reviews much simpler as it is easier to spot a mistake. For example, reset in the i.MX RT685 is in the +1.8v "always-on" domain. Make sure it is clearly marked so you do not pull it to +3.3v!

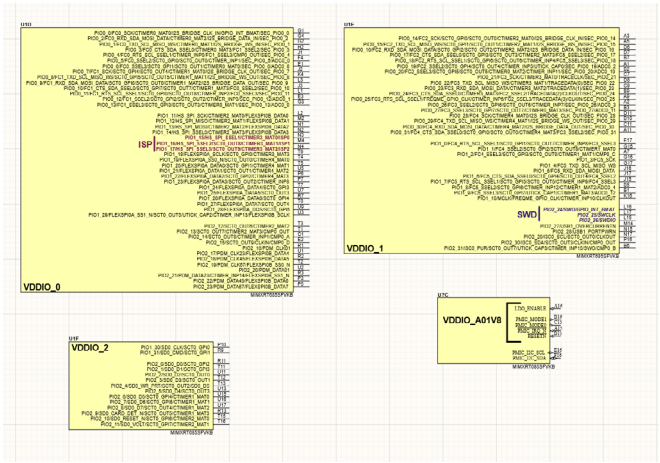


FIGURE 20. i.MX RT685 IO SCHEMATIC BLOCKS

Power Management Options for the i.MX RT685

The power architecture for the i.MX RT685 is more complicated than a traditional single voltage rail MCU because of its flexibility. However, once you understand the architecture, the application design is straightforward. For example, in a minimal configuration that requires +3.3v IO, one could power the i.MX RT685 with +3.3v and +1.8v using the internal LDO for the core.

NXP offers an integrated solution that makes powering the i.MX RT685 much simpler: the PC4390 Power Management Integrated Circuit (PMIC). A PMIC is essentially a handful of DC-DC converters, LDOs and control circuitry integrated into a single package.

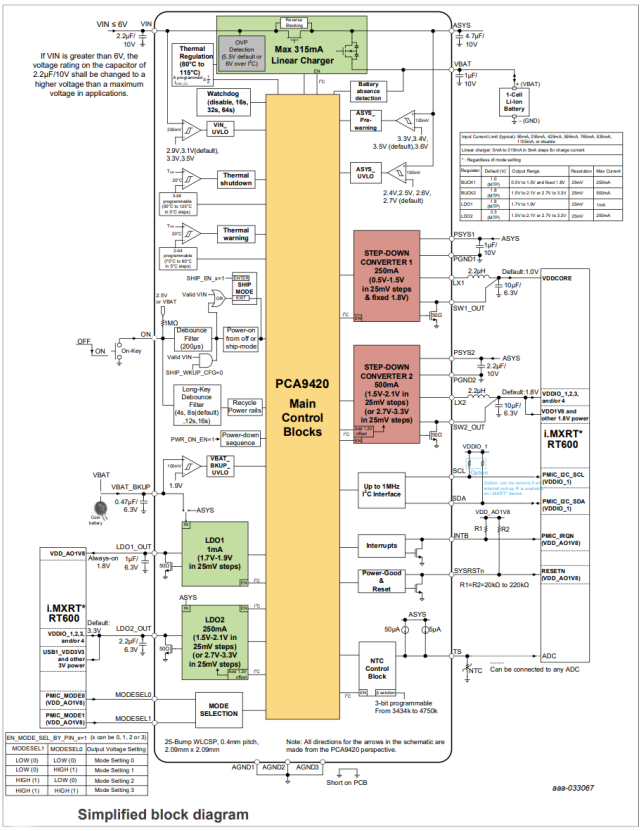


FIGURE 21. THE PCA9420 PMIC

In the case of the PCA9420, it has been specifically designed to power the i.MX RT685. The out of the box configuration will perform all the necessary sequencing to bring up the device correctly. It is packaged in a 24-pin QFN package which is extremely compact given the number of functions it is providing. The PCA9420 also provides reset control and has an I2C interface that allows the supplies to be tuned for the different clocking configurations.

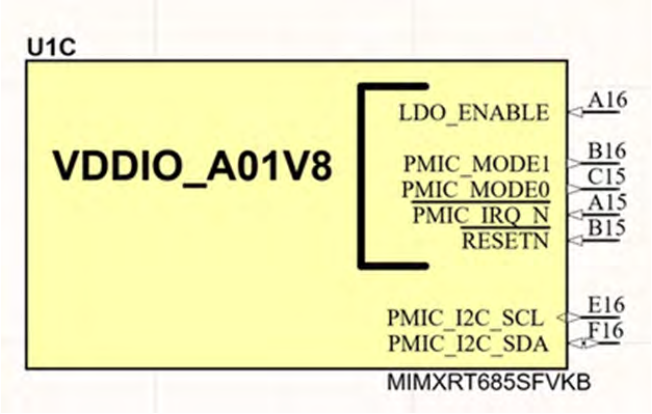


FIGURE 22: i.MX RT685 DEDICATED PMIC INTERFACE

I highly recommend using the PCA9420, especially if this is the first design iteration using the i.MX RT685. A designer is certainly free to use other power management solutions but the PCA9420 is a path to ensure success. If your application requires any power management and sleep functions, then the PCA9420 is an optimum choice. The Super-Monkey design will be using the PCA9420 to simplify the power supply. I plan on using the PCA9420 on the Super-Monkey as the module will be powered with a single +5v rail.

i.MX RT685 Flash Memory Interface

When starting with a new MCU, I almost always examine power architecture and device packaging as the topics are often interwoven. Once I have a good idea of how to properly power a part and understand what the PCB layout will look like, the next order of business is understanding boot-up and debugging.

Like power & package, boot & debug are also often interconnected. It is important to consider both topics simultaneously. The i.MX RT685 series have additional design considerations as they are flash-less MCUs as the designer must add a flash memory external to the part. The flash-less MCU paradigm allows for flexibility in your design but can be challenging if it is your first experience with a flash-less MCU.

For the Super-Monkey project, I am electing to use a simple external memory configuration with common QSPI NOR flash. In my use case, time-critical DSP algorithms will execute from fast internal RAM. Boot functions and non-critical code can live in the external QSPI flash. I feel this is a good tradeoff for cost, storage capacity and performance. QSPI is now well established in the marketplace and NXP was the 1st to offer execute in place (XIP) technology over a quad SPI bus (QSPI or SPIFI). This technology was introduced over 10 years ago with the LPC18xx and LPC43xx microcontrollers. Since then, QSPI XIP has proven to be a very good choice for many applications. It is very common for real-time audio/DSP code to have a reasonably small footprint and it can execute from the fastest internal/tightly coupled memories. Most other system code requirements are well served by XIP from external QSPI (serial communications, USB, system tasks, etc.). It is important to note that the memory controller in the i.MX RT685 has 32KB of cache. A little bit of cache on XIP QSPI memory can go a long way to improve performance.

FlexSPI Flash Controller

Built into the i.MX RT685 is a special flash interface controller called “FlexSPI”. FlexSPI supports access to Single/Dual/Quad/Octal flash interfaces through the internal AHB bus. This means the CPU can access SPI memory as if it were a normal memory mapped flash device. The details of the SPI transactions are handled by the FlexSPI controller. FlexSPI enables a wide array of memories to be connected to the MCU. With

FlexSPI, designers have access to extremely dense flash memories and can use new flash devices as they become available. Designs can be easily scaled to storage requirements as needed. SPI memories are also easy to route on a PCB as there are fewer connections. Chapter 33 of UM11147 details the FlexSPI controller and possible use cases.

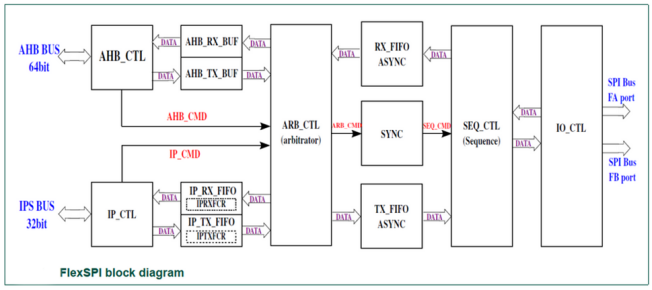


FIGURE 23. NXP FLEXSPI MEMORY CONTROLLER.

The FlexSPI controller has two ports which can be further subdivided into two separate interfaces allowing a maximum of 4 QSPI devices if needed. To support future designs, the controller implements a look up table (LUT) that allows command sequences to be altered. It is also important to note that in addition to flash, there are now SPI based (p)SRAM devices in the marketplace allowing for volatile memory to be added as well.

QSPI on FlexSPI Port A Configuration and Boot

The MIMXRT685-EVK demonstrates use of the FlexSPI peripheral with two devices attached. One being a high-speed octal pSRAM connected to port A and an octal flash connected to port B. Note the "octal" interface. FlexSPI supports up to 8-bit transactions on a Double Data Rate (DDR) interface. Octal devices are still quite new as compared to quad devices but offer a significant performance increase (4x between the increased data path and DDR interface). I also want to note something important about the FlexSPI on the i.MX RT685. There are two ports, but Port A can support higher speed transfer through the use of an additional DQS pin. In the case of the RT685EVK, the pSRAM is wired to the higher speed port. I do have to chuckle a bit when the terms "SPI" and "Octal" are combined. These memory interfaces are now synchronously clocked parallel buses. QSPI was a simple extension to traditional 1 bit SPI but has now grown from the simple "serial" use case. For the Super-Monkey, I want to demonstrate a simplified scenario.

There is one other detail important to the i.MX RT685 with regards to the FlexSPI connections:

If you want to use all 8 channels of the DMIC interface, FlexSPI B will not be available. You must connect to FlexSPI A.

Figure 23 shows how to connect a QSPI flash to the i.MX RT685.

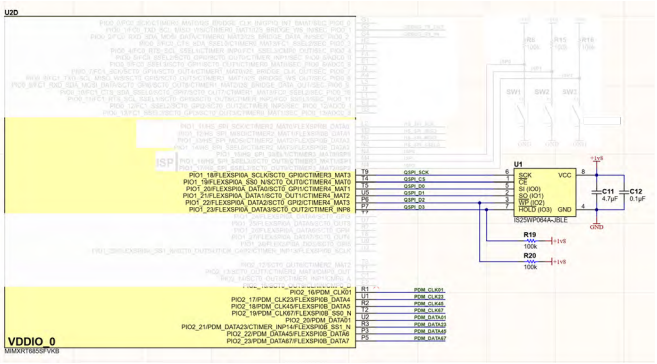


FIGURE 24. QSPI FLASH CONNECTION ON THE i.MX RT685.

As you can see, QSPI memory is simple to connect. I am using a +1.8v flash device. Typically, I arrange my schematic symbols to be group functions by power domain to reduce the chance of error. In this case, VDDIO_0 will be powered by +1.8v from the PMIC. I left the connections to DMICs highlighted so you can see the intersection with FlexSPI B. MCUXpresso IDE includes a pin tool that allows you to plan the device muxing/pinout to reveal any potential conflicts.

With every new hardware design, I create a skeleton MCUXpresso SDK software project that gets included in my hardware design git repository. The skeleton project's sole purpose is to define all of my IO as I am working through the hardware design.

Boot Configuration

Once the flash memory is connected, there are two questions I always ask. How does the CPU know to boot from this external flash and how do I program the flash? The i.MX RT685 series shares a legacy with the LPC family of microcontrollers in that there are a few "ISP" pins which control the boot process. These pins are sampled at power up by routines located in ROM. One of the features in the LPC family of microcontrollers are ROM boot functions/utilities that enable programming over a few different serial interfaces. The i.MX RT685 continues this tradition. Like its LPC predecessors, boot configuration is simple to set up.

Table 991. Boot mode and ISP Downloader modes based on ISP pins

Boot mode	ISP2 pin PIO1_17	ISP1 pin PIO1_16	ISP0 pin PIO1_15	Description
-	low	low	low	Reserved
SDIO0 (SD Card)	low	low	high	Boot from an SD card device connected to SDIO 1 interface. The RT6xx will look for a valid image in the SD card device. If there is no valid image found, the RT6xx will enter the ISP boot mode based on OTP DEFAULT_ISP_MODE bits (6:4, BOOT_CFG [0]) defined in Table 1110 "BOOT_CFG[0] bit fields".
FlexSPI Boot from Port B	low	high	low	Boot from Quad or Octal SPI Flash devices connected to the FlexSPI interface 0 Port B. The RT6xx will look for a valid image in external Quad/Octal SPI Flash device. If there is no valid image found, the RT6xx will enter recovery boot or ISP boot mode.
FlexSPI Boot from Port A	low	high	high	Boot from Quad/Octal SPI Flash devices connected to the FlexSPI interface 0 Port A. The RT6xx will look for a valid image in external Quad/Octal SPI Flash device. If there is no valid image found, the RT6xx will enter recovery boot or ISP boot mode.
SDIO 0 (eMMC)	high	low	low	Boot from an eMMC device connected to SDIO 0 interface. The RT6xx will look for a valid image in the eMMC device. If there is no valid image found, the RT6xx will enter the ISP boot mode based on the value of OTP DEFAULT_ISP_MODE bits (6:4, BOOT_CFG [0]) defined in Table 1110 "BOOT_CFG[0] bit fields".
-	high	low	high	Reserved
Serial ISP (UART, SPI, I2C, USB-HID)	high	high	low	The Serial Interface (UART, SPI, and I2C,USB-HID) is used to program OTP, external Flash, SD or eMMC device.
111	high	high	high	Serial Master boot (SPI Slave, I2C Slave, or UART, USB-HID) is used to download a boot image over the serial interface (SPI Slave, I2C slave or UART, USB-HID).

Table 992 shows the ISP pin assignments and is the default pin assignment used by the ROM code that cannot be changed.

FIGURE 25. SELECTING FLEXSPI PORT A ON THE ISP BOOT PINS.

For the Super-Monkey module, I elected to use three switches to control the boot pins. Note that the ISP lines are in the +1.8v power domain. Setting the switches appropriately will allow FlexSPI port A to be used by the boot ROM.

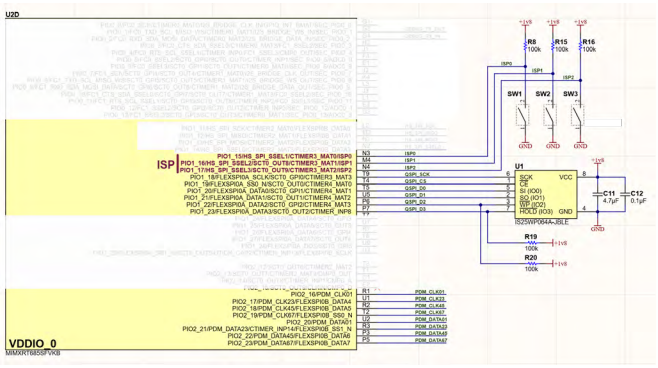


FIGURE 26. i.MX RT685 ISP PIN CONNECTIONS

The i.MX RT685 user manual (Chapter 41) documents how the boot process functions in detail.

Programming and Debug

An important mode of the ROM bootloader in the i.MX RT685 is called "Serial ISP" which enables a PC/host to perform flash operations (and program OTP fuses) via interfaces such as UART or USB. It is important to *always* allow this mode to be available in your designs. It will save a great deal of time and headache down the road.

Table 991. Boot mode and ISP Downloader modes based on ISP pins

Boot mode	ISP2 pin PIO1_17	ISP1 pin PIO1_16	ISP0 pin PIO1_15	Description
-	low	low	low	Reserved
SDIO0 (SD Card)	low	low	high	Boot from an SD card device connected to SDIO 1 interface. The RT6xx will look for a valid image in the SD card device. If there is no valid image found, the RT6xx will enter the ISP boot mode based on OTP DEFAULT_ISP_MODE bits (6:4, BOOT_CFG [0]) defined in Table 1110 "BOOT_CFG[0] bit fields".
FlexSPI Boot from Port B	low	high	low	Boot from Quad or Octal SPI Flash devices connected to the FlexSPI interface 0 Port B. The RT6xx will look for a valid image in external Quad/Octal SPI Flash device. If there is no valid image found, the RT6xx will enter recovery boot or ISP boot mode.
FlexSPI Boot from Port A	low	high	high	Boot from Quad/Octal SPI Flash devices connected to the FlexSPI interface 0 Port A. The RT6xx will look for a valid image in external Quad/Octal SPI Flash device. If there is no valid image found, the RT6xx will enter recovery boot or ISP boot mode.
SDIO 0 (eMMC)	high	low	low	Boot from an eMMC device connected to SDIO 0 interface. The RT6xx will look for a valid image in the eMMC device. If there is no valid image found, the RT6xx will enter the ISP boot mode based on the value of OTP DEFAULT_ISP_MODE bits (6:4, BOOT_CFG [0]) defined in Table 1110 "BOOT_CFG[0] bit fields".
-	high	low	high	Reserved
Serial ISP (UART, SPI, I2C, USB-HID)	high	high	low	The Serial Interface (UART, SPI, and I2C,USB-HID) is used to program OTP, external Flash, SD or eMMC device.
111	high	high	high	Serial Master boot (SPI Slave, I2C Slave, or UART, USB-HID) is used to download a boot image over the serial interface (SPI Slave, I2C slave or UART, USB-HID).

Table 992 shows the ISP pin assignments and is the default pin assignment used by the ROM code that cannot be changed.

FIGURE 27. SERIAL ISP BOOT MODE.

Serial ISP can be your "escape hatch" when you are having issues or need a production programming/ configuration interface. At minimum, you should always provide access to the UART lines on Flexcomm 0 (PIO0_1 / PIO0_2).

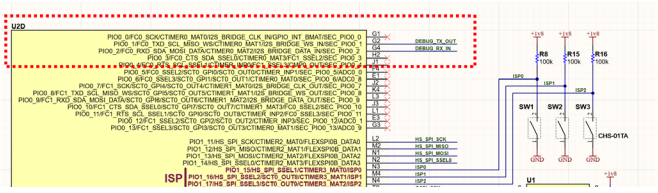


FIGURE 28. SERIAL ISP UART PINS

I cannot stress how important it is to have access to these pins! In addition to the UART, I would also recommend having USB access as well. The ROM bootloader enumerates a USB-HID device for access to the programming/test functions. There are minimal parts required to enable the USB interface. I always add some EMI filtering to the +5v VBUS line as well as some ESD protection to the D+/D- lines.

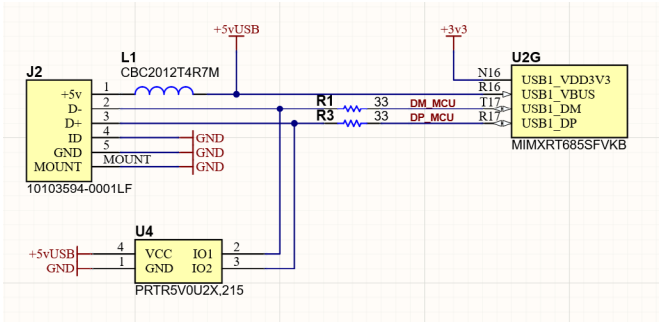


FIGURE 29. i.MX RT685 MINIMAL USB CONNECTIONS.

I do want to point out the minimal clocking configuration required by the ROM bootloader. A 24MHz crystal is required for correct USB operation.

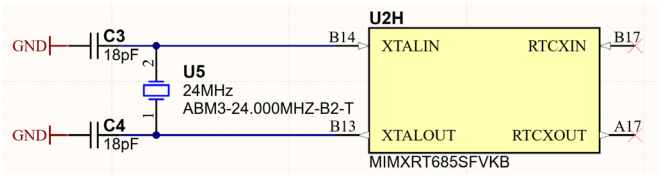


FIGURE 30. i.MX RT685 MINIMAL CRYSTAL/CLOCK CONFIGURATION.

I am not using the RTC and elected to not connect a 32KHz crystal. For the minimal configuration scenario, a 24MHz crystal is the path to quick success.

ISP Serial Boot Support Software

There are a couple of useful software tools to drive the serial ISP boot functions. The first is a command line application "blhost". The i.MX RT685 reference manual has some examples of the different uses of blhost (see chapter 41). blhost can program externally connected flash, discover timing parameters, and program internal OTP fuses. It is a Swiss Army Knife that is very useful for testing, debugging and deploying your design.

"blhost" is one component in a larger suite of tools found in the NXP Secure Provisioning SDK.

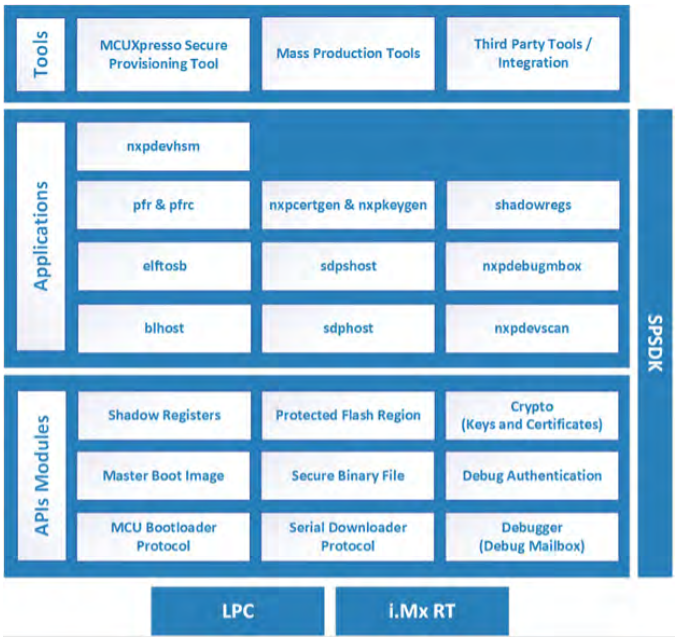


FIGURE 31. BLHOST IN THE CONTEXT OF THE NXP SECURE PROVISIONING SDK.

The NXP Secure Provisioning SDK is open source and can be found at https://github.com/NXPmicro/spsdk.

In addition to blhost, the MCUXpresso Secure Provisioning Tool can also be used to program images over the ISP interface. The MCUXpresso Secure Provisioning Tool provides an easy-to-use GUI for preparing bootable encrypted images and burning OTP fuses over the ISP interface. The GUI is built upon the tools in the NXP Secure Provisioning SDK. I find quite a bit of utility in the GUI tools as having a visual interface to the flash memory and OTP fuses can reduce mistakes. While you can always reprogram flash memory, a mistaken setting in the OTP fuses can be frustrating.

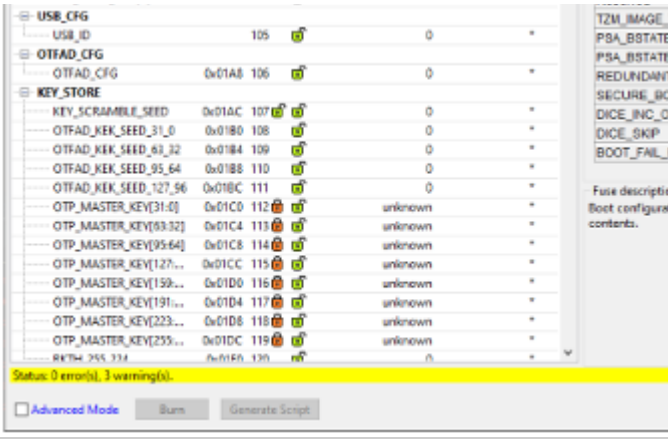


FIGURE 32. MCUXPRESSO SECURE PROVISIONING TOOL - OTP FUSE GUI

The 2nd tool can be found in the open-source community.

An alternative to the NXP supported tools for ISP operations is the "MCUBootUtility" which can be found in the open-source community.

MCUBootUtility is a python-based GUI that wraps blhost to perform all of the most common operations. It supports the i.MX RT685 parts as well as all the other parts in the i.MX RT crossover family.

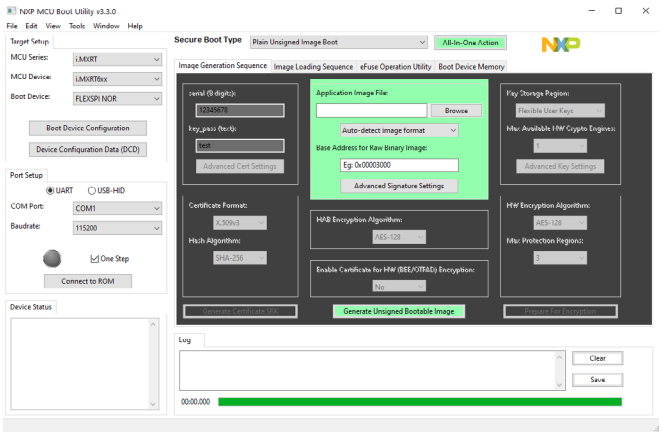


FIGURE 33. OPEN SOURCE MCU BOOT UTILITY

MCUBootUtility and the NXP Secure Provisioning SDK (which contains blhost) are both available as open source. One of the primary differences between the tools is that the Secure Provisioning SDK is officially supported by NXP. MCUBootUtility is supported by its author on GitHub.

i.MX RT685 Debug

The last piece we need to look at is debug access. The i.MX RT685 implements a standard ARM SWD interface. On my 1st design iteration with a new MCU, I always add the standard .050" ARM header for debug. The only connections needed for a minimal configuration are SWDCLK and SWDIO. Keep in mind that SWD trace IO are also available on the i.MX RT685, but I am choosing to use the simplest configuration. The only extra feature I added was Serial Wire Output (SWO). Once in a while, I will turn it on for streaming additional data through the debugger. However, I found that simple SWD is often good enough. I prefer to use Segger's J-Link and their Ozone debugger "but the MCUXpresso IDE can use low-cost debugger" should probably be "but the MCUXpresso IDE can be used with a low cost-debugger such as the MCU-Link. There is quite a bit you can get done with simple 2-wire SWD and the available tools.

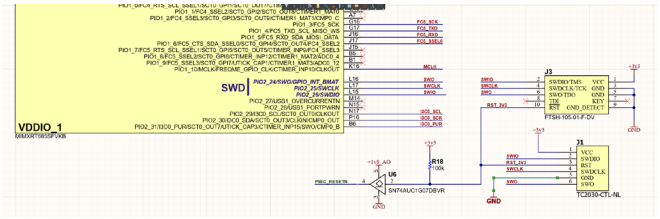


FIGURE 34. i.MX RT685 DEBUG CONNECTIONS

In additional to the standard 0.050" debug header, I always add Tag-Connect pads for debug access. As the design progressed closer to the production build, the standard header can be added to the do not place list and the if debug as needed.

FIGURE 35. TAG-CONNECT SWD CABLE

It should also be noted that the reset signal from the debug header may need a level translator. The reset connection from the PMIC to the i.MX RT685 is in the always on +1.8v power domain. IO Bank 1 in the Super-Monkey will be tied to a +3.3v rail. A simple solution is to use a simple open-drain buffer to connect the reset line into the PMIC/i.MX RT685 reset.

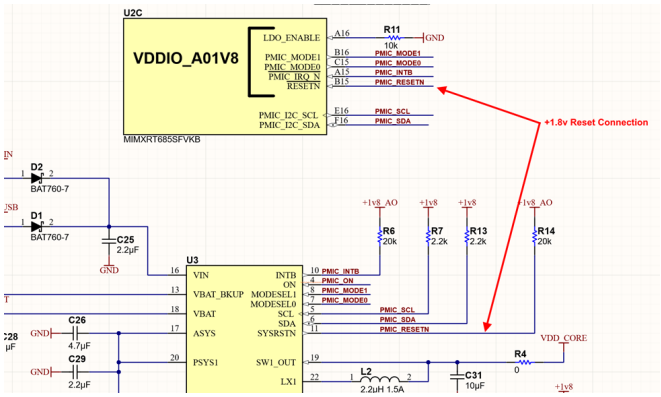


FIGURE 36. PMIC RESET CONNECTION.

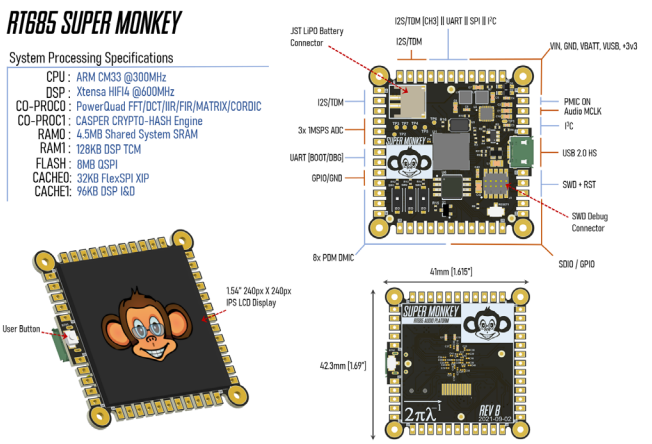


FIGURE 37. THE i.MX RT685 SUPER-MONKEY

Once debug access is enabled, you have another method to program the flash on the FlexSPI port. The MCUXpresso IDE debugger as well as the Segger J-Link support programming of QSPI flashes as they implement programming algorithms for SFDP based flash devices. When you are in an edit/download/debug loop, programming the QSPI via SWD is the quickest way to get code loaded. The Serial ISP function in ROM is also available but it is better suited to loading of finished binaries for production, etc.

Realizing the Super Monkey Hardware

There is quite a bit of IO available on the i.MX RT685, but I chose to constrain my design to the most common functions for real-time audio. My applications generally use professional, "flagship quality" audio codecs for musical instrument signal processing. Using this as a guide, the process of coming up with a minimal IO complement was simplified.

I spent quite a bit of time considering a form factor. Since this was my first attempt at an i.MX RT685 design, I elected to constrain the formfactor to something known. Last year I developed the Mini-Monkey around the NXP LPC55S69. For the "Super-Monkey", I chose to keep this form factor but extend the number of IO. Decision making fatigue is a very real concern for engineers. The "perfect module form factor" trap is easy to fall into. It is tempting to spend a great deal of time optimizing for a general-purpose use case to handle unknown requirements. My experience has shown me that in almost every case, overthinking future possibilities at the outset never pays dividends down the road. We engineer's love to tinker and will end up making changes anyway. I have two separate projects I want to implement with the i.MX RT685, so I decided to focus my efforts exclusively on what I need.

I thought it would be instructive to list in Figure 35 some of the high-level processing capabilities available in the i.MX RT685. It is difficult to overstate the amount of capability packed into the i.MX RT685. Every few years I like to step back and reflect on progress that has been made with MCU integration.

It is often too easy to overlook the sum of science & engineering needed to integrate this amount of technology in a 9mmx9mm package. All of it for a couple lattes at the coffee shop.

Super-Monkey IO

Power Input
+5v from the micro-USB or IO connector. There is also JST "PH" connector for common LiPO batteries. PCA9420 PMIC has a built-in battery charger so the Super-Monkey can be "portable".

Digital Audio IO
The Super-Monkey has two I2S/TDM audio interfaces provided by the flexcomm peripheral. In my applications, I tend to separate the ADC and DAC to achieve the highest SNR. One channel will be used to interface with an 8-channel audio ADC (Cirrus Logic CS5368) and the other will be used with an AES3 Audio Transmitter (Cirrus Logic CS8406). The second channel will drive a professional audio DAC (Cirrus Logic CS4398). For some applications, I prefer the audio master clock to be generated with a high-quality, minimal phase noise oscillator. The Super-Monkey has a pin that can be used to bring in an external master clock.

Digital Microphones
The Super-Monkey exposes all eight DMIC inputs available on the i.MX RT685.

Spare Flexcomm
I reserved pins for one Flexcomm port to be allocated on a per project basis. This will allow me to select I2C, SPI, UART or an extra I2S bus as needed.

Debug UART
There is one dedicated UART exposed via the IO pins. This UART can be used with the ROM bootloader to program flash, fuses, etc.

I3C
I reserved two pins to experiment with the I3C bus. I3C is a new superset of the I2C standard which allows for much faster data rates, multi-master and hot join capability. It lives at an interesting intersection of I2C and SPI, so I wanted a platform to run experiments.

SDIO
A 4-bit SDIO port for adding an SD card. I do not have an immediate requirement for SDIO but at a minimum I get a handful of GPIO that are also multiplexed with SCT (State Configuration Timer) channels. I *really* like the SCT in all the LPC families so I made sure to make it available.

ADC
I exposed three inputs to the i.MX RT685 1MSPS ADC, two of which can be used to form a fully differential input. The intent is to use the ADC channels for potentiometers/user input. I have plans for piezo vibration sensor applications as well.

LCD
I borrowed the 1.54" from the LPC55S69 Mini-Monkey Design. In fact, the module form factor was original designed around this specific display. It looks really sharp, so I decided to include it for some neat audio visualizations. The LCD is wired to a dedicated 50MHz high speed SPI port and can achieve high frame rates. Coupled with the large RAM in the i.MX RT685, the Super-Monkey can perform some interesting graphics applications.

Super Monkey PCB Routing

I intended the Super-Monkey to use a 4-layer PCB stackup. To achieve a 4-layer design, I had to make some compromises as compared to the layout on RT685-EVK. The RT686 in the VFBGA176 package is straightforward to fan out. In fact, the PCB routing on the RT685-EVK has all IO fanned out on the top layer. One caveat is that 3mil trace/space design rules are required The EVK is quite large and there is plenty

of room to bring all the traces on the top layer. The Super-Monkey has more parts placed very close to the MCU package, so I needed to make use of the bottom layer near the package. As an alternative example to the RT685-EVK, I chose to use 5mil design rules coupled with via-in-pad technology to fan out the IO.

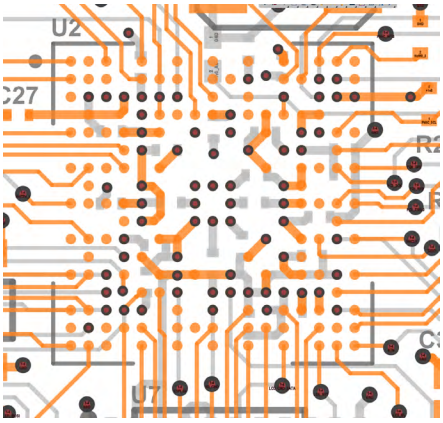


FIGURE 38. SUPER-MONKEY IO FANOUT

The via-in-pad geometry uses a 12-mil pad with 6mil mechanical drill. This geometry is on the edge of where laser microvias would be required. The two inner layers (not shown) are dedicated to return and power (Core, +1.8v and +3.3v). Via-in-pad allowed me to use a more relaxed 5mil trace/space rule which is a lower cost and higher yield option as compared to 3-mil. Since I did not require all the i.MX RT685 IO, keeping to a 4-layer design was straightforward. Via-in-pad does increase cost, but the trade-off vs 3-mil spacing makes it an attractive option to relax the clearance specifications and fanout to other layers. It also allows one place vias on the decoupling pad simplifying placement.

The VFBGA176 package does leave some balls unpopulated making routing and fanout simpler. Most of the core power and ground connectors are on the interior of the package. This alleviates quite a bit of contention with IO. In Figure 36, you can see the 0201 capacitors on the bottom side. I believe with some work, 0402 size capacitors could be used to relax manufacturing tolerances. 0201 capacitors are quite small. Chris Denny of Worthington assembly posted some nice blogs on optimum geometries for high yield when using 0201 packages. He also has a nice article on via-in-pad technology.

It is certainly possible to route the i.MX RT685 VFBGA176 on a 4-layer process and still be able to use quite a bit of the IO. In fact, with a bit more work, I feel I could use about 90% of the IO before resorting to a 6-layer process. While this package is small, it is still possible to use it successfully without breaking the bank with high-cost PCB fabrication processes.

SuperMonkey Board Build

For board fabrication and assembly, I used CircuitHub. CircuitHub handles all aspects of the fabrication process. All you need to do is upload your design files and CircuitHub will acquire the raw PCBs, procure the components, and stuff the boards. The CircuitHub workflow also allows for consigned parts. Given the current situation with the supply chain, this was a welcome feature. I was able to send them some of the parts that were not available through normal channels as I had "banked" a few of the components several months ago.

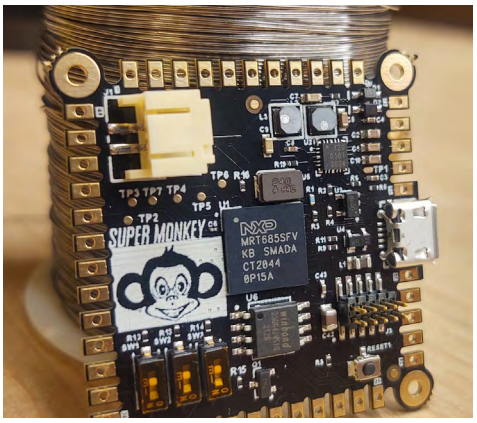


FIGURE 39. THE i.MX RT685 SUPERMONKEY

You can find [source files for the hardware design] on GitHub.

The specs for this board use a small drill requirement (6mil) and a tight annual ring (3mil). This drill geometry was used for filled via-in pad technology on the VFBGA176. I chose this approach for fanning out the VFBGA176 package as it enabled looser specs on trace space/width. The RT685-EVK did not use via-in-pad technology, rather a 3mil trace space/width for escaping some of the signals. I elected to use a filled via-in-pad approach after some discussion with CircuitHub. The result was a fairly simple routing exercise on 4-layers. I do also want to point out that this design utilized 0201 sized decoupling capacitors. The 0201 decoupling capacitors were placed on the bottom side of the board directly under the i.MX RT685 MCU.

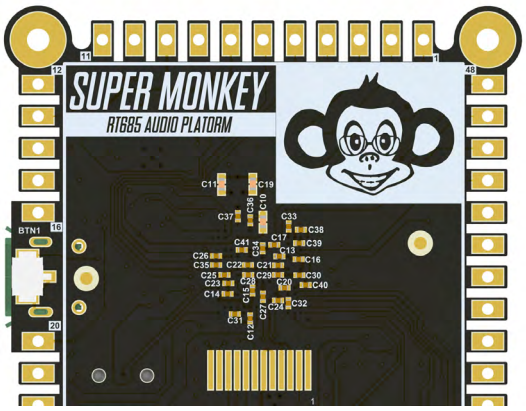


FIGURE 40. 0201 DECOUPLING CAPACITORS

I usually try to use packages that are as large as possible to ease manufacturing design rules but in this case I elected to use an 0201 size capacitor to experiment with the "perfect 0201" land pattern provided by [Chris Denny of Worthington Assembly.] I believe with some work, 0402 size capacitors could be used with the i.MX RT685 to relax manufacturing tolerance. On my next project with the i.MX RT685, I may use a smaller number of 0402 0.22uF decoupling caps in place of 0201 sized 0.1uF decoupling caps.

The simplest approach to providing power to the i.MX RT685 is using the NXP PCA9420BSAZ PMIC. The [PCA9420BSAZ] comes in an interesting 24-pin QFN package that also requires a soldered connection at the corners of the device package.

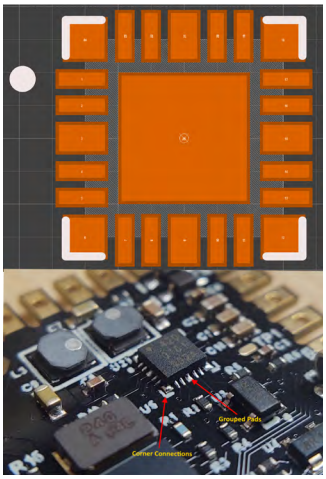


FIGURE 41. THE PCA9420 PMIC FOR THE i.MX RT685

Using a service such as a CircuitHub saved a lot of time and hassle as I did not have to worry about manually placing and reflowing these parts. While the i.MX RT685 in the VFBGA176 is not at the extremes of package technology, I will gladly delegate the assembly work to a 3rd party to ensure success on the first attempt.

Bare Metal Bring Up with QSPI Flash

Before attempting to bring the board up with Zephyr, it is important to make sure everything is working in a bare metal context. The i.MX RT685 is a flashless MCU. An external flash device is required for boot, or you need some other mechanism for loading code in the RAM. The RT685-EVK uses an Octal Flash connected to FlexSPI port B. Octal flash technology is still quite new to the MCU scene, and I really wanted to show an example of using lower cost (and better availability) QSPI Flash. In many cases QSPI will be a suitable

choice as code that needs high performance can be executed from the copious supply of internal SRAM in the i.MX RT685. The i.MX RT685 implements a cache on the FlexSPI interface so execution from external memory is still acceptable for many applications. Another consideration is that if a design requires access to all 8-DMIC channels, FlexSPI Port A must be used for code storage. My intent was to design the SuperMonkey as a reference for using a QSPI device on FlexSPI port A.

The i.MX RT685 Boot Header

At powerup, the i.MX RT685 executes boot code from ROM. The bootloader is configurable and requires the boot source to be specified via three ISP pins. It is important to leave the ability to select other boot sources. In addition to booting from external flash, there is boot mode that uses a USB or a serial connection to program internal fuses and perform production programming operations. I find that indicating the boot options on the schematic is a good design practice to ensure the other utility boot modes do not get overlooked.

From the build tooling perspective, there are a couple changes that are necessary to use QSPI flash with the i.MX RT685. You can use an MCUXpresso SDK example project as a starting point even though it is configured to use octal flash. The i.MX RT685 ROM bootloader will attempt to access an external device on FlexSPI A or B (depending on the ISP [2:0] pins) using a 1-bit IO mode. A special data structure, the boot header, is to be located at a 0x400 byte offset from the start of external flash. This boot header provides additional configuration details to the bootloader about the attached flash device. All the example i.MX RT685 projects in the MCUXPresso SDK contain a boot header suitable for an octal flash device. You can find the boot header in flash_config/flash_config.c and it will need to be modified to support QSPI boot.

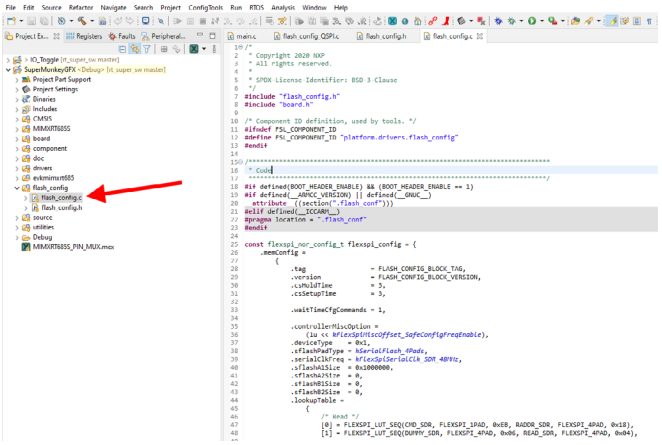


FIGURE 42. BOOT HEADER LOCATION MCUXPRESSO PROJECT

If you dig into the linker file generated by the build tools, it specifies a special section that is placed in the QSPI memory region. Note that the. ld linker files are automatically generated by default by MCUXpresso and placed in the build configuration output folder (for example, “debug”).

```
ENTRY(ResetISR)

SECTIONS
{
     /* Image Vector Table and Boot Data for booting from external flash */
    .boot_hdr : ALIGN(4)
    {
        FILL(0x00)
        __boot_hdr_start__ = ABSOLUTE(.) ;
        . = 0x400 ;
        __flash_conf_hdr_start__ = ABSOLUTE(.) ;
        KEEP(*(.flash_conf))
        __flash_conf_hdr_end__ = ABSOLUTE(.) ;
        . = 0x1000 ;
         __boot_hdr_end__ = ABSOLUTE(.) ;
    } >QSPI_FLASH

    /* MAIN TEXT SECTION */
    .text : ALIGN(4)
    {
        FILL(0xff)
```

FIGURE 43. BOOT HEADER PLACEMENT WITH THE LINKER

Here is an example of a functional QSPI boot header:

```
/*
 * Copyright 2020 NXP
 * All rights reserved.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */
#include "flash_config.h"
#include "board.h"

/* Component ID definition, used by tools. */
#ifndef FSL_COMPONENT_ID
#define FSL_COMPONENT_ID "platform.drivers.flash_config"
#endif

/*******************************************************************************
 * Code
 ******************************************************************************/
#if defined(BOOT_HEADER_ENABLE) && (BOOT_HEADER_ENABLE == 1)
#if defined(__ARMCC_VERSION) || defined(__GNUC__)
__attribute__((section(".flash_conf")))
#elif defined(__ICCARM__)
#pragma location = ".flash_conf"
#endif

const flexspi_nor_config_t flexspi_config = {
    .memConfig =
        {
            .tag              = FLASH_CONFIG_BLOCK_TAG,
            .version          = FLASH_CONFIG_BLOCK_VERSION,
            .csHoldTime       = 3,
            .csSetupTime      = 3,

            .waitTimeCfgCommands = 1,

            .controllerMiscOption =
                (1u << kFlexSpiMiscOffset_SafeConfigFreqEnable),
            .deviceType    = 0x1,
            .sflashPadType = kSerialFlash_4Pads,
            .serialClkFreq = kFlexSpiSerialClk_SDR_48MHz,
            .sflashA1Size  = 0x1000000,
            .sflashA2Size  = 0,
            .sflashB1Size  = 0,
            .sflashB2Size  = 0,
            .lookupTable =
                {
                    /* Read */
                    [0] = FLEXSPI_LUT_SEQ(CMD_SDR, FLEXSPI_1PAD, 0xEB, RADDR_SDR,
FLEXSPI_4PAD, 0x18),
                    [1] = FLEXSPI_LUT_SEQ(DUMMY_SDR, FLEXSPI_4PAD, 0x06, READ_SDR,
FLEXSPI_4PAD, 0x04),

                    /* Read Status */
                    [4 * 1 + 0] = FLEXSPI_LUT_SEQ(CMD_SDR, FLEXSPI_1PAD, 0x05, READ_SDR,
FLEXSPI_1PAD, 0x04),

                    /* Write Enable */
                    [4 * 3 + 0] = FLEXSPI_LUT_SEQ(CMD_SDR, FLEXSPI_1PAD, 0x06, STOP_EXE,
FLEXSPI_1PAD, 0x00),

                },
        },
    .pageSize           = 0x100,
    .sectorSize         = 0x1000,
    .ipcmdSerialClkFreq = 1,
    .blockSize          = 0x10000,
};
#endif /* BOOT_HEADER_ENABLE */
```

Once the boot header is built into your image, the i.MX RT685 becomes “stand-alone” after a power cycle. Note that if your project does not have a boot header, or if it is not valid, the ROM bootloader will jump to USB/serial downloader mode. There are tools, such as:

https://github.com/JayHeng/NXP-MCUBootFlasher

that can also generate a boot header as well as place it in the proper location in flash. I personally like to bake it into the C source code so the project is “bootable” without any additional steps.

MCUXpresso IDE Flash Programming Configuration

Once MCUXpresso IDE is able to build a valid, bootable image we need a method to program it. One approach is to use the serial/USB downloader built into ROM. I prefer to use a debugger tool over the SWD port. There are several options available, but I want to present two that can perform flash programming and debugging over the SWD connection.

MCUXpresso IDE + MCU-Link

The MCUXpresso IDE is capable of programming QSPI flash on the i.MX RT685 with [the low-cost MCU-Link debug adapter.] This is a path for quick success but requires a change to the MCUXpresso IDE project to indicate the type of flash connected and the FlexSPI port.

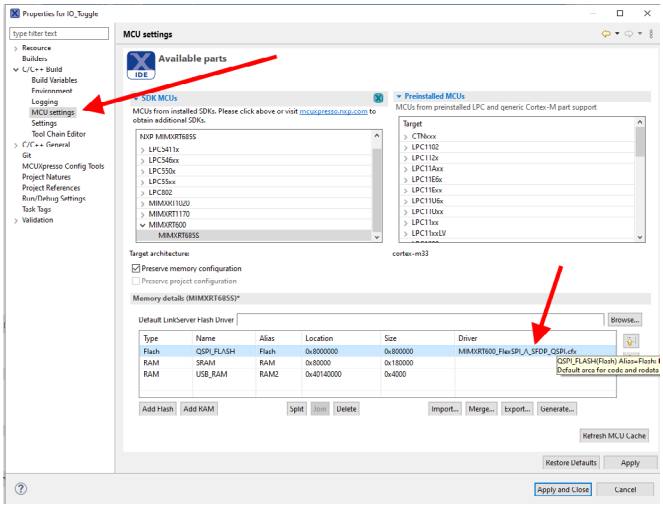


FIGURE 44. QSPI FLASH PROGRAMMING DRIVER IN MCUXPRESSO IDE

Ensure that "MIMXRT685_FlexSPI_A_SFDP_QSPI.cfx" is selected as the "Driver" for the flash region. Once this change is made, you can simply start a debug session and MCUXpresso will be able to program a QSPI flash connected to FlexSPI port A.

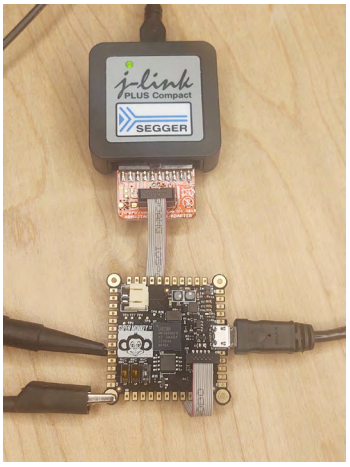


FIGURE 45. PROGRAMMING AND DEBUG WITH AN NXP MCU-LINK PROBE

Programming and Debug with Segger J-Link + Segger Ozone

A tool that has become a favorite in my arsenal is Segger's [Ozone Debugger.] It is a full featured source level graphically debugger designed to work with a Segger J-Link. All you have to do is point to your .elf or .axf file and Ozone handles the rest. I have found Ozone to be a great option as it is very fast, can inspect variables without halting the MCU and works with all of NXP's ARM MCUs. I discovered it back in 2014 while looking for a better tool for dual core debugging on the NXP LPC4357 and have been using it ever since.

When setting up an Ozone project, there are a couple of settings that are required for working with the i.MX RT685. Per [Segger's Wiki Article] for the i.MX RT685, use the MIMXRT685SFAWBR device when your flash device is on FlexSPI port A.

NXP	MIMXRT633S	Cortex-M33	1	64 MB + 64 MB	3.072 MB
NXP	MIMXRT633S_M33	Cortex-M33	1	64 MB + 64 MB	3.072 MB
NXP	MIMXRT685S	Cortex-M33	1	64 MB + 64 MB	4.608 MB
NXP	MIMXRT685S_M33	Cortex-M33	1	64 MB + 64 MB	4.608 MB
NXP	MIMXRT633SFAWBR	Cortex-M33	1	64 MB + 64 MB	3.072 MB
NXP	MIMXRT685SFAWBR	Cortex-M33	1	64 MB + 64 MB	4.608 MB

FIGURE 46: SEGGER i.MX RT685 QSPI ON FLEXSPI PORT A SETUP

Also, per [Segger's Wiki Article] working on with a bootloader, make sure to uncomment AfterTargetDownload() and AfterTargetReset().

```
190  /*********************************************************************
191  *
192  *      AfterTargetDownload
193  *
194  * Function description
195  *   Event handler routine. Optional.
196  *   The default implementation initializes SP and PC to reset values.
197  *
198  *********************************************************************
199  */
200  void AfterTargetDownload (void) {
201
202  Exec.Reset();
203
204  }
205
```

FIGURE 47. AFTER TARGET DOWNLOAD CONFIGURATION

I leave the Exec.Reset() to make sure to reset the CPU after programming so the ROM bootloader can take over. Lastly, make sure TargetReset() is left commented out

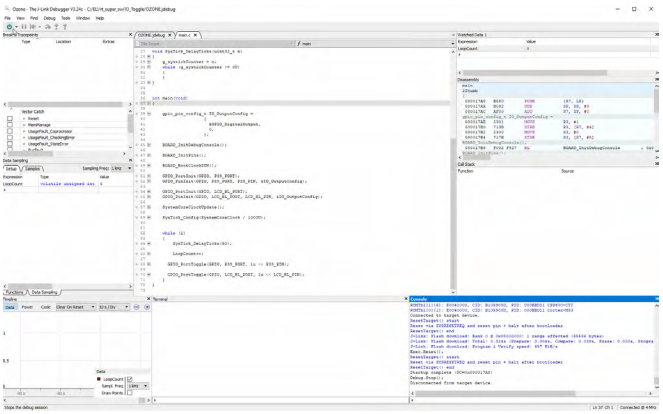
```
/*********************************************************************
*
*       TargetReset
*
* Function description
*   Replaces the default target device reset routine. Optional.
*
* Notes
*   This example demonstrates the usage when
*   debugging an application in RAM on a Cortex-M target device.
*
*********************************************************************
*/
//void TargetReset (void) {
//
//  unsigned int SP;
//  unsigned int PC;
//  unsigned int VectorTableAddr;
//
//  VectorTableAddr = Elf.GetBaseAddr();
//  //
//  // Set up initial stack pointer
//  //
//  if (VectorTableAddr != 0xFFFFFFFF) {
//    SP = Target.ReadU32(VectorTableAddr);
//    Target.SetReg("SP", SP);
//  }
//  //
//  // Set up entry point PC
//  //
//  PC = Elf.GetEntryPointPC();
//
//  if (PC != 0xFFFFFFFF) {
//    Target.SetReg("PC", PC);
//  } else if (VectorTableAddr != 0xFFFFFFFF) {
//    PC = Target.ReadU32(VectorTableAddr + 4);
//    Target.SetReg("PC", PC);
//  } else {
//    Util.Error("Project file error: failed to set entry point PC", 1);
//  }
//}
```

FIGURE 48. TARGETRESET CONFIGURATION

```
/*********************************************************************
*
*       AfterTargetReset
*
* Function description
*   Event handler routine. Optional.
*   The default implementation initializes SP and PC to reset values.
**
*********************************************************************
*/
void AfterTargetReset (void) {
}
```

FIGURE 49. AFTERTARGETRESET CONFIGURATION

These steps are important so that the ROM bootloader functions are not interfered with after the flash is programmed and the target is reset. Once we have a working boot header and the ability to program the board, the first order of business is to toggle an IO pin. My first test program toggled a couple IO every 50mS.



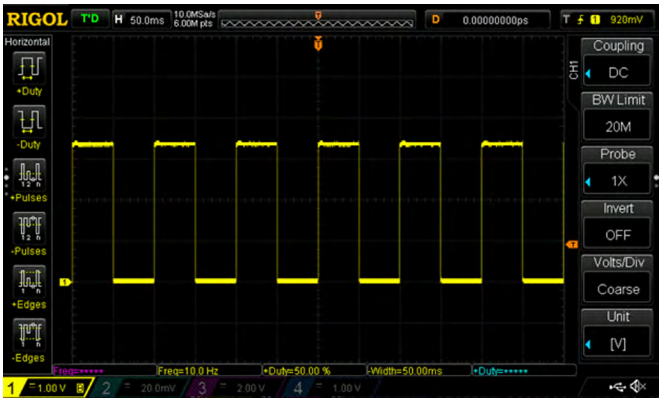


FIGURE 50. PROGRAMMING AND DEBUG WITH A SEGGER J-LINK AND SEGGER OZONE.

At this point, we have successfully brought up the i.MX RT685 with QSPI on FlexSPI port A and successfully have toggled an IO. Now we know that our hardware is good so we can get Zephyr up and running.

4. Zephyr RTOS Bringup on the i.MX RT685 SuperMonkey

I have been both vocal and enthusiastic about the **Zephyr Real Time Operating System** (RTOS) and **recently presented my thoughts** on some of the "why". The i.MX RT685 now has **full platform support** in the latest LTS 2 version of Zephyr. The design goal of the SuperMonkey design was to demonstrate a minimal configuration of an i.MX RT685 using a QSPI flash device. I wanted to demonstrate something a little bit different than what was on the EVK as the flash memory interface is highly configurable. Along those same lines, I want to show what bringing up Zephyr would look like on a custom i.MX RT685 board. The flexibility of its flashless architecture means that there are some additional considerations to get the board up and running.

I really like bare minimum examples as they are the best place to start when learning a new tool. Once one understands the bare minimum configuration, it is easier to pull in more complicated code from other examples. Often the simplest example will show all the structural elements so one can infer how a system works. This makes adding functionality easier to approach and there are just enough breadcrumbs to discover more sophisticated functionality. The example provided here illustrates just that; A minimal example that will bring up enough to show you that the system is working. For the case of a Zephyr application on the i.MX RT685, I want to show that the serial shell is functional. I will be inserting tidbits here and there that are a bit beyond the scope of a board port as they will leave breadcrumbs to some unique Zephyr features and workflow.

A bit about Zephyr boards

When performing application development with Zephyr, the concept of a "board" is central. A powerful feature of Zephyr is to be able to retarget an application to diverse hardware. Moving code between say an LPC55S69 to an i.MX RT685 can be straightforward as use of the built in APIs will allow quite a bit of functionality to be reused with minimal changes. When using Zephyr, you can still write all the custom bare metal, raw register access code you want. You are not limited to the Zephyr API. However, using the common APIs for UART, I2C, etc. will make the portability easier. This will reduce the total surface area of code that will need to change between different boards. The board abstraction is the core component of this process so your application code can run on multiple platforms.

So, what exactly is a Zephyr board? Loosely speaking, it is a folder with

- A device tree overlay that brings in a specific SoC and peripheral configurations
- Kconfig settings for the OS that are important for the board
- .c code to do things like pin muxes, etc.
- Documentation

Basically, everything needed to sit on top of the OS code to get the system up and running in some configuration. An important point to note is that one can use a "built-in" board to do quite a bit of work before even considering a "custom" board definition. The device tree system allows application code override/overlay behaviors to "modify" an existing board. So, you don't always need a dedicated Zephyr "board" when using your own hardware. The good news is that there are a lot of boards to use as reference:

https://docs.zephyrproject.org/latest/boards/index.html

For the case of the i.MX RT685, there is a nuance with external flash setup that requires the use of a custom board. For most MCUs with internal flash, the built-in boards do 99% of what you need to get started.

Getting Setup for Custom Board Development

Setting up a west manifest

In this article I will be weaving in some other Zephyr concepts that I have found useful. There are several workflows for managing Zephyr projects. The RTOS itself is available here:

https://github.com/zephyrproject-rtos/zephyr

When 1st getting started, it is 100% OK to follow the instructions and start hacking on projects in the /samples directory. However, once I know I will be doing some real work on a project, I like to organize my project as a west manifest:

https://docs.zephyrproject.org/latest/guides/west/manifest.html#option-4-sequence

There is quite a bit of documentation around manifests which can be overwhelming. In the simplest case, simply think of it as a way of organizing git repositories for your project with some automation. With a simple manifest, you can specify the repository for your application code and the repository for Zephyr. West can then be used to fetch everything and get it setup for you. There is a bit of setup time at the beginning but pays dividends down the road. I like my application code to exist "out of tree" from the Zephyr repo. The west manifest automates the setup process.

As an example, check out the west.yml from here:

https://github.com/ehughes/rt_super_z/

This is a minimal manifest that will allow west to initialize a folder and fetch the Zephyr source code. As projects get more complicated, you can add other dependencies at different revisions levels, etc. I find this useful for maintaining "internal repositories" that have not been upstreamed into the mainline Zephyr codebase or for anything that I want to keep private while I develop. You could even specify the repository of your own Zephyr fork that is completely under your control. The system is quite flexible. In fact, if you do a bit of digging, you will find a west.yml in the main zephyr repo which is a manifest of all the "stuff" that is pulled in.

The key take-away once the manifest is created, getting your project development environment setup is just two commands (west init, west update) which makes life much easier over the long term. When you have multiple developers and multiple machines. Feel free to use the rt_super_z repository as starting point. This repo has the west manifest and a simple hello world project structure that I will be using for my board port.

Seeding your custom board using the RT685 EVK

It is important to note that board porting is well documented here:

https://docs.zephyrproject.org/latest/guides/porting/board_porting.html

You could use that documentation and start from scratch but a copy/paste from a known working configuration that is “close” is a good strategy. In the Zephyr source tree, you can find the i.MX RT boards under zephyr/boards/arm.

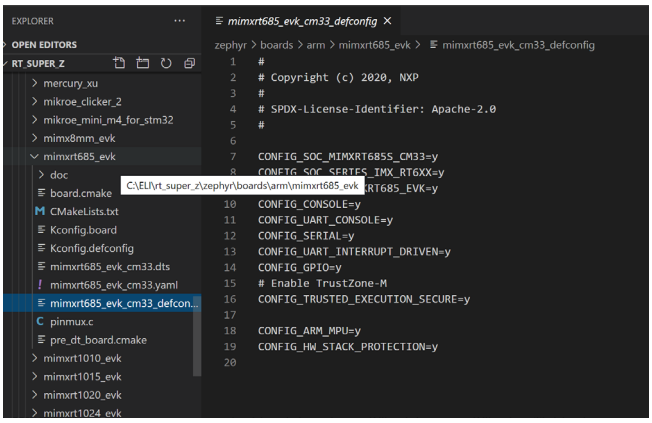


FIGURE 51. ZEPHYR MIMXRT685_EVK BOARD

Note that I use Visual Studio Code as an editor when working with Zephyr. I find it very easy to navigate the source tree when doing driver/board development. The next step I took was to *copy* the mimxrt685_evk folder into my application project under hello_world/boards/arm. For initial development of the board port, the hello world test will house my board direction. I also followed the existing naming conventions to create an “rt_super” board.

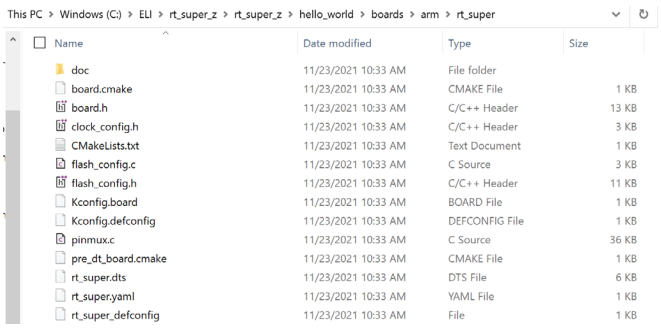


FIGURE 52. CUSTOM RT_SUPER BOARD

It takes just a few minutes to go through the .yaml, defconfig, etc. to get your custom naming in place. I first focused on remapping text that was MIMXRT685_EVK to RT_SUPER. By default, Zephyr will look in its default boards directory when you kick off a build. To change this behavior, simply edit the CMakeLists.txt in your application folder and add this:

```
# Re-direct the directory where the 'boards' directory is found from
# $ZEPHYR_BASE to this directory.
set(BOARD_ROOT ${CMAKE_CURRENT_LIST_DIR})
```

FIGURE 53. SETTING THE BOARD_ROOT

This changes where Zephyr will look for your board when kicking off a fresh build with “west build”. This directory could be anywhere but for now I will keep it local to this test project. I also like to fix the board selection for my project, so I don’t have to specify it on the command line during development of the board package. Add this to your application CMakeLists.txt to fix the board for the application:

```
11    set(BOARD rt_super)
```

FIGURE 54. FIXING THE BOARD FOR A PROJECT

At this point you should be able to kick of a build and Zephyr will use the rt_super board in your local application folder. Keep in mind that the goal here

was to simply copy files and rename items in the .yaml and kconfig files so we have a clean starting point with our “rt_super” board. None of the internal configuration of the hardware has actually changed at this point. We essentially have another board that will target the MIMXRT685-EVK, just under a different name. You could have also kept using the default Zephyr boards folder for development. I chose to put it in my application repository to make it a bit easier to work between my home and office computers

Notable Customizations for the SuperMonkey

QSPI and Boot Header

The i.MX RT685 represents a bit of a trickier case with board porting versus a traditional MCU with built in flash memory. There is a little more work to get Zephyr configured for the particular flash on our custom hardware. If your design kept the same memory configuration as the EVK (OctalSPI on FlexSPI Port B), then additional setup is not necessary. Flashless microcontrollers are both a blessing and a curse in this regard. The flexibility allows for better application customization but requires some more work up front to get your system setup for development

After power up or reset, the i.MX RT685 first executes a bootloader that is built into ROM. The i.MX RT685 ROM bootloader will attempt to access an external device on FlexSPI Port A or B (depending on the state ISP[2:0] pins) using a 1-bit IO mode. A special data structure is expected at a 0x400 offset from the base of the external flash memory. This boot header provides additional configuration details to the bootloader about the attached flash device. The ROM code will reconfigure the interface per the boot header configuration. If the boot header does not exist or is invalid, the ROM will sit and wait for commands over USB or a UART interface.

During a Zephyr build, the boot header needs to get baked in so the generated binary can run properly after reset. Using the mimxrt685_evk board as a guide, we can modify the boot header for our custom board. Learning how to navigate the Zephyr source tree is an important aspect to being successful in developing Zephyr boards and drivers.

Inside of the zephyr/boards/arm/mimxrt685_evk folder is a file Kconfig.board.

```
4  config BOARD_MIMXRT685_EVK
5      bool "NXP MIMXRT685-EVK"
6      depends on SOC_SERIES_IMX_RT6XX
7      select SOC_PART_NUMBER_MIMXRT685SFVKB
8      select NXP_IMX_RT6XX_BOOT_HEADER if !BOOTLOADER_MCUBOOT
9
```

FIGURE 55. BOOTHEADER INCLUSION

Here we are defining a new configuration option for the board that selects a different option NXP_IMX_RT6XX_BOOT_HEADER. The boot header is not actually located in the Zephyr source tree, rather in another NXP repository that gets pulled in when you first setup your Zephyr workspace. Note that this happens automatically when setting up Zephyr initially with west init and west update. One reason I really like using a west manifest as described previously is that all these dependencies come in automagically.

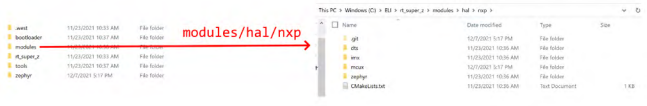


FIGURE 56. NXP HAL LOCATION

In your Zephyr workspace is a modules folder that contains external vendor libraries. If you dig inside, you can find the NXP HAL. Open modules\hal\nxp\mcux\boards\CMakelists.txt to can see how the configuration option NXP_IMX_RT6XX_BOOT_HEADER is used.

```
if (${MCUX_BOARD} MATCHES "evk[bm]imxrt1[0-9][0-9][0-9]")
zephyr_library_sources_ifdef(CONFIG_BOOT_FLEXSPI_NOR ${MCUX_BOARD}/${MCUX_BOARD}_flexspi_nor_config.c)
zephyr_library_sources_ifdef(CONFIG_DEVICE_CONFIGURATION_DATA ${MCUX_BOARD}/dcd.c)
zephyr_include_directories(./${MCUX_BOARD})
elseif (${MCUX_BOARD} MATCHES "evkmimxrt6[0-9][0-9]")
zephyr_library_sources_ifdef(CONFIG_NXP_IMX_RT6XX_BOOT_HEADER ${MCUX_BOARD}/flash_config.c)
endif()
```

FIGURE 57. BOOTHEADER / FLASH CONFIGURATION BUILD

The file flash_config.c contains a data structure with an attribute for the linker to place it in the correct location in flash. For our custom board, we will need to have our own copy of this file with the custom boot header. You can place your own version in the custom board directory.

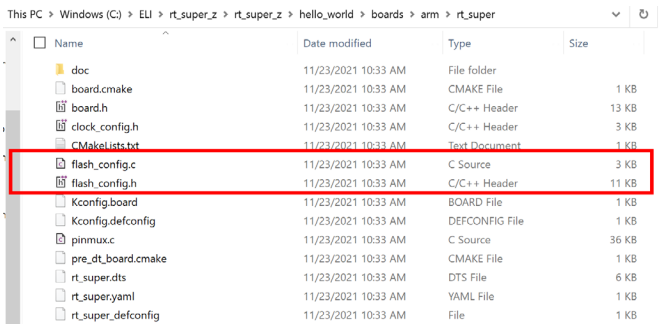


FIGURE 58. CUSTOMIZING FLASH_CONFIG.C

You can read more about the bootheader I used here. To get the new added file included in the build, you can add a line to the CMakeLists.txt in the custom board folder:

```
zephyr_library()

zephyr_library_sources(pinmux.c)

zephyr_compile_definitions_ifdef(CONFIG_NXP_IMX_RT6XX_BOOT_HEADER BOOT_HEADER_ENABLE=1)

zephyr_library_sources(flash_config.c)
```

FIGURE 59. BOARD.CMAKE UPDATE

Here we set BOOT_HEADER_ENABLE as it is used in flash_config.c Note that we are safe to add flash_ config.c here. The version in the NXP HAL directory will not be used as our custom Kconfig.board and Kconfig. defconfig uses alternate naming for the board symbols.

The flash_config.c in the NXL HAL will not be added to the build as the mimxrt685 board symbols will no longer be enabled. At this point we are set up to our custom boot header. This step is probably the most complicated which is the result of the “flashless” nature of the i.MX RT685 and its boot ROM.

Board Device Tree Overlay

RTC Disable

After getting the boot header configured, there were a couple modifications needed in the board device tree overlay. For the most part you can leave this identical to the EVK. Even if there are peripherals you don’t plan on using, you can leave the elements be. There are a couple changes however needed to get the SuperMonkey board to boot. I chose to not populate a 32.768kHz Crystal on my board.

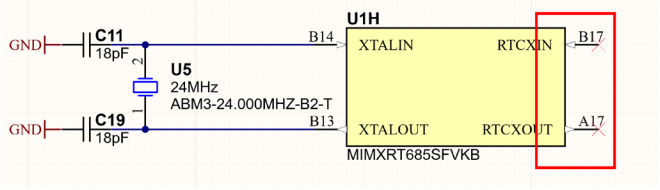


FIGURE 60. NO RTC CRYSTAL ON THE SUPERMONKEY

I found that some of the low level i.MX RT685 SoC init code would try to start it up causing boot issues. To turn this off, I simply had to disable the driver in the board level overlay:

```
/*
 * RT600 EVK board uses OS timer as the kernel timer
 * In case we need to switch to SYSTICK timer, then
 * replace &os_timer with &systick
 */
&os_timer {
    status = "okay";
};

&rtc {
    status = "disabled";
};
```

FIGURE 60. DISABLING THE RTC IN THE DEVICE TREE

Flash Configuration

The SuperMonkey uses a QSPI device for code storage. I had to patch the flexspi node in the board device tree overlay to indicate the new flash driver. As a reference, I actually used the device tree overlay from the i.MXRT1064 EVK board. This board uses the same QSPI flash device as the SuperMonkey. Since the i.MX RT685 and RT1060 share the same FlexSPI IP, we can reuse the drives and device tree configuration. The device tree approach used by Zephyr starts paying dividends quite quickly when one needs to stitch together complicated systems as there is a defined method for IP/driver re-use.

FIGURE 61. DEVICE TREE FLASH NODE

I did patch the flash size and partition sizes to be more appropriate for my smaller capacity QSPI flash.

Board Kconfig.defconfig

There is one last config file that needs to be patched for our particular flash configuration on the SuperMonkey. Kconfig.defconfig inside of the board folder has a bunch of additional Kconfig options added you your project. There is a FLASH_SIZE configuration setting which defaults to a value pulled from the device tree. Using the syntax from the MIMXRT685 EVK, I modified it to pull from our modified device tree with QSPI. The Kconfig and device tree systems are very powerful and allow configuration data to be pulled into the build system and application code.

FIGURE 62. DEVICE TREE FLASH NODE

Keep in mind that at the end of the build process, the Kconfig files and device tree overlays turn into header files for your application to use to determine how the system is configured. Kconfig options can be driven from the device tree.

https://docs.zephyrproject.org/latest/guides/build/kconfig/preprocessor-functions.html

Pin Multiplexing

NXP provided board ports add a file pinmux.c to the build system. The purpose of this code is to provide a function that will set up the pin mux on the device to route desired functions to the physical pins on the IC. Before we take a look at the function that does the work, I wanted to point out a macro located at the bottom of the file.

```
1054   /* priority set to CONFIG_PINMUX_INIT_PRIORITY value */
1055   SYS_INIT(rt_super_pinmux_init, PRE_KERNEL_1, 45);
1056
```

FIGURE 63. PIN MUX PRE-KERNEL SETUP

Zephyr is unique in that it provides macros to statically define functionality in a very “clean” manner. One of those macros is “SYS_INIT”. This macro adds functions to a list that are called during kernel initialization. In this case, rt_super_pinmux_init is called before the kernel is initialized. The advantage of this approach is that one can insert additional behaviors at different points in the boot process without hacking the core kernel code. This also cleans up your “main” routine as you can specify “APPLICATION” in the macro to have functions called right before main(). I find that this approach makes

libraries and modules much easier to develop leaving the application code tidy.

FIGURE 64. KCONFIG VS DEVICE TREE MACROS

In this implementation, you can see there is logic to conditionally include pin mux setup functions. This code provides an example of using both Kconfig macros and device tree access macros. I mentioned before that both KConfig and device tree overlay files ultimately get translated into header files with macros that you can use in your application code. Feel free to use as much or as little functionality as you want in the pin mux initialization code. It is a good idea to get any UART pins setup in this stage as you will be able to see printk, logging and shell output from the boot process. If you want, you can also do pin mux configuration in your application code but keep in mind you have the option to get pins initialized very early in the boot process.

I do want to point out that there are some important changes coming to pin muxing in Zephyr:

https://github.com/zephyrproject-rtos/zephyr/issues/39740

Some platforms, such as the Nordic NRF families, have a great deal of flexibility in pin assignments where any digital function can be routed to any pin. In this case, NRF based device trees overlays can specify IO pins when peripherals nodes are instantiated. This makes modifying boards very simple with the device tree overlay mechanism. There is work in progress on a new pinctrl API that will allow similar behavior (within the limits of the particular chip) such that pin assignments can be performed in the device tree overlay. Keep a lookout for this feature in future releases. For now, you will need to perform pin muxing in the board initialization functions or in your application code.

Board.cmake

This file is used to define "runners" that will allow programming with the west flash. Feel free to look at other board examples to set this up. In my case, I liked to program and debug with Segger Ozone and a J-Link instead of using the command line flash mechanism. You can see my i.MX RT685 setup here:

https://community.nxp.com/t5/Blogs/i-MX-RT685-SuperMonkey-QSPI-Bring-up-with-MCUXpresso-and-Segger/b...

When a build is complete, you can use zephyr.elf in build/zephyr folder of your application. With this approach you can get fully featured source level debugging of your Zephyr application. Ozone even offers the ability to be Zephyr "thread aware".

Final Results

Given the additional steps needed for the i.MX RT685 board port, I wanted to test with a simple project that turns on the shell. This would give me a workable base to expand the board port. My hello_world program turns on the Zephyr shell over a UART and enables a “monkey” command.

FIGURE 65. FINAL RESULTS.

Ascii art sourced from https://textart.sh/topic/monkey

We now have the fundamental elements in place for this board port. I hope you found this information in getting the i.MX RT685, or any other part, setup with a custom board in Zephyr. One you get acclimated to the Zephyr workflow, it isn’t too complicated to build new applications, drivers and boards quickly. The i.MX RT685 requires a few extra steps to get the boot header in place but once you have the RTOS building, you are on your way to making some cool stuff.

This particular has a large amount of internal SRAM (4.5MB), a secondary DSP core (a topic for another day), and cool DSP peripherals (the PowerQuad) so it makes a great playground for Zephyr applications.

[You can find SuperMonkey board port here.]

There you have it! A walkthrough from PCB design through Zephyr RTOS bringup with the i.MX RT685. I hope you found this article helpful in seeing the end to end process.