Adapting the Communication Process for a CMOS-Based E-nose System on the FRDM-MCXN947 Board

BSc Thesis

J. K. Pang
W. H. Y. Rong

Technische Universiteit Delft

Adapting the Communication Process for a CMOS-Based E-nose System on the FRDM-MCXN947 Board

BSc Thesis

by

Jin-tjat Kevin Pang
Wilson Hao Yian Rong

Student:	J. K. Pang	5684595
	W. H. Y. Rong	5686873
Project duration:	April 21, 2025 – June 27, 2025	
Supervisors:	Prof. dr. ir. F. P. Widdershoven,	TU Delft, NXP Semiconductors
	MSc. T. Shen	TU Delft
Thesis committee:	Prof. dr. ir. F. P. Widdershoven,	TU Delft, NXP Semiconductors
	MSc. T. Shen	TU Delft
	Dr. ir. C. J. M. Verhoeven	TU Delft

TU Delft

Abstract

This report outlines the design and implementation of a subsystem within the automated e-nose prototype, which is based on a CMOS Pixelated Capacitive Sensor (PCS) array. The objective of the e-nose prototype is to detect the presence and identify the type of volatile organic compounds (VOCs). A potential use case is to accommodate greenhouses with these e-noses to detect infestations in plants.

The subsystem presented in this report is tasked with programming the FRDM-MCXN947 development board to enable and improve communication and control between the PCS array and the microcontroller unit (MCU). In addition, the development board is also explored to investigate the feasibility of replacing currently used external modules with on-board peripherals. To achieve this, the CTIMER and SCTIMER are analyzed and implemented with the MCUXpresso IDE. The results showed that both of these peripherals have difficulties in simultaneously generating clock signals of different frequencies required for the communication protocol between the sensor and the MCU. However, the SCTIMER is capable of generating a differential clock, currently produced by the AD9552 external clock, though with lower signal quality.

Preface

This thesis was written as part of the Bachelor's program in Electrical Engineering at TU Delft. It reports on the software development conducted within the project "Building a Fully Automated E-nose Prototype Based on a CMOS Pixelated Capacitive Sensor Array." The intended audience of this thesis includes fellow students participating in the Electrical Engineering Bachelor. Therefore, it is assumed that the reader possesses the knowledge acquired from all courses within the program.

This project is divided into three subgroups, each responsible for a specific component of the fully automated e-nose prototype. To gain a complete understanding of the system's overall development, the reader is encouraged to consult the theses written by the other two subgroups [1][2].

We would like to express our gratitude to Prof. Dr. Ir. F. P. Widdershoven and Mr. T. Shen for proposing such an intriguing project. Their guidance over the past eight weeks has been both educational and encouraging. Working on this project has given us valuable insights into how academic research and engineering practice are integrated within an educational setting.

We also thank Delft University of Technology, especially the staff at the Tellegen Hall, for providing the necessary materials and a safe, supportive working environment. Finally, we want to thank our fellow students Yousri Machtane, Pieter Olyslaegers, Chantal Chen and Feifei Lin for their collaboration and for making this project an enjoyable and rewarding experience.

J. K. Pang
W. H. Y. Rong
Delft, June 2025

Contents

1 Introduction

Humans and animals rely heavily on their sense of smell for flavor perception, hazard detection, and social communication, yet replicating this sensory capability in machines remains a complex and ongoing challenge. Various electronic nose (e-nose) systems have been developed using different techniques such as gas chromatography and mass spectrometry [3]. However, the type of e-nose system that forms the basis for this project, utilizes complementary metal oxide semiconductor (CMOS) technology in combination with microcapacitors arranged in a 2-D array, referred to as “Pixelated Capacitive Sensor (PCS)” arrays [4]. This existing e-nose system uses the LPC1769 microcontroller [5] and several external components like amplifiers, ADCs and a differential clock generator.

While the hardware of the e-nose is crucial for olfactory capabilities, the functionality of the system heavily depends on the embedded software that controls various processes. This thesis focuses on the development and implementation of the software layer, specifically the software running on the microcontroller (MCU), which controls sensor readout, data acquisition, data transfer, preprocessing, and communication with external systems.

Objectives

Before diving into the embedded software and MCU, a global understanding of the e-nose system is required. In Figure 1.1 a strongly simplified block diagram of the e-nose system is shown.

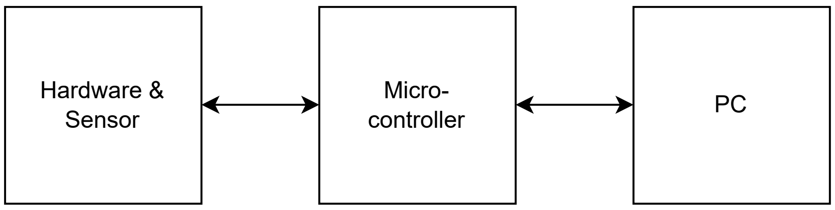


Figure 1.1: Simplified overview of the e-nose system

The *Hardware & Sensor* block consists of the hardware to facilitate signal acquisition from the PCS array to the MCU. The PC block represents the computer to which data is sent from the MCU, furthermore, the PC allows for configuring the MCU. This thesis concerns the *Microcontroller* block. The centerpiece of this block is the FRDM-MCXN947 board, detailed information about this MCU board is provided in [6]–[8]. This board houses numerous peripherals, such as direct memory access (DMA) modules, Digital-to-Analog Converters (DAC), Analog-to-Digital Converters (ADC), a neural processing unit (NPU), and timer modules. This extensive choice of peripherals along with the presence of an NPU are the largest motivators for utilizing the FRDM-MCXN947 board. The objectives of the software group are:

- to program the FRDM-MCXN947 board to facilitate and improve communication and control between the *Hardware & Sensor* block and the *Microcontroller* block in Figure 1.1.

- to explore the FRDM-MCXN947 board in search of peripherals that are able to replace external components ”in the existing sensor board”.

Lastly, the NPU allows for machine learning (ML) purposes in the future. The ultimate goal is to have on-board machine learning, however a good starting point is to have a *TinyML* model trained on the PC and then mapped onto the FRDM-MCXN947. This *TinyML* algorithm will then run in inference mode on the MCU. That said, use of the NPU and *TinyML* algorithms is beyond the scope of this thesis.

State of the Art

Microcontrollers are compact integrated circuits designed to perform and control specific operations in embedded systems. They integrate a processor core (CPU), memory, and programmable input/output peripherals on a single chip, enabling efficient control of electronic devices [9]. The evolution of MCUs has seen a transition from simple 4-bit processors to complex 32-bit and 64-bit systems, accommodating the growing demands of modern applications [10]. However, these improvements in processing power are not unbounded; they are constrained by physical limitations (e.g., Moore’s Law) and economic constraints [11]. As MCUs are increasingly tasked with handling real-time signal processing, controlling peripherals, and moving data, relying only on the processor core becomes inefficient and may lead to performance bottlenecks. Consequently, specialized hardware and parallel processing techniques to offload the CPU have become the focus for continuation of performance scaling [12].

One such specialized hardware module to offload the CPU is a direct memory access (DMA) controller [13]. The main purpose of the DMA is to regulate the exchange of a large volume of data between the memory and peripherals at high speed without CPU intervention [14]. In particular, the decoupled DMA described in [15] and [16] are interesting for this project because of the split between the CPU data and the I/O data. A programmable logic unit (PLU) may also be used to offload the CPU and may even increase throughput [17].

Another technique to offload the CPU and improve performance is multicore processing [18]. Multicore processing may be implemented for handling the signals between the three blocks in Figure 1.1, as multicore capabilities can enhance performance in multitasking environments, where a number of foreground applications, such as virus protection, wireless management, compression, encryption, and synchronization, run concurrently with background processes [19].

Finally, in the scope of this project, the aforementioned techniques, PLU, and DMA module can be implemented on the FRDM-MCXN947, as this board has two cores and also DMA modules [7]. However, this board is relatively new, having been released in 2024 [20], thus very few projects have been conducted yet on this board. Nonetheless, [21] provides example code for setting up the PLU on the FRDM-MCXN947. Furthermore, [22] contains numerous examples of code for implementing the DMA and [23] shows projects from the NXP Software Development Kit (SDK).

Thesis Outline

After this chapter, background information will be provided in chapter 2, which outlines the current software and the differential clock implementation of the existing e-nose system. Following that, the project’s program of requirements will be presented in chapter 3. Tasks outlined in chapter 4 are defined based on these requirements. In chapter 5, the FRDM-MCXN947 will be examined to identify suitable peripherals for completing the tasks. The implementation and results with these peripherals are shown in chapter 6 and discussed in chapter 7. Finally, the thesis will conclude with a summary of this thesis and recommendations for future work in chapter 8.

2 Background Information

This chapter presents the essential background information and foundational concepts necessary to understand the steps outlined in the subsequent chapters. By establishing this context, the reader is better prepared to follow the upcoming tasks, methodology, analyses, and conclusions. The current software implementation of the e-nose system is discussed in detail in section 2.1. Furthermore, the currently used external clock is outlined in section 2.2. These subsections provide a foundation for understanding the remainder of this thesis.

2.1. Current Software Implementation

The current software is programmed on a board based on an LPC1769 microcontroller of NXP. It is an ARM Cortex-M3-based MCU for embedded applications featuring a high level of integration and low power consumption.
The software configures the MCU to manage communication between the MCU and the computer, as well as between the MCU and the PCS arrays (*Hardware & Sensor* block), as shown in Figure 1.1. The communication between the MCU and the computer uses a standard USB protocol, which requires no additional implementation. Hence, the focus will be on the software implementation for the connection between the *Hardware & Sensor* and *Microcontroller* blocks.
The MCU is connected to 3 PCS arrays via 3 ADCs, allowing the MCU to read the analog output signals of the PCS arrays. Furthermore, the MCU also has the ability to write to the PCS arrays to select a specific sensor readout line and sensor input clock. The MCU handles read and write operations in a systematic manner with precise timing. Figure 2.1 shows the communication process for a single capacitive sense pixel transfer from the PCS array to the MCU via one ADC, while simultaneously sending the sensor configuration bits.

```
-------------------------------------------------------------------------------------------------------------------------------------
             +--+                                                                                                        +--+
CSS_CS       |  |                                                                                                        |  |
            -+  +--------------------------------------------------------------------------------------------------------+  +---
SET          1     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     1
clock       0| 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35| 0| 1|...
CLR             1     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     1

            ----------+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +-----------+
ADC_SCK               | 1|  | 2|  | 3|  | 4|  | 5|  | 6|  | 7|  | 8|  | 9|  |10|  |11|  |12|  |13|  |14|  |15|  |16|           |
                      +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+  +--+           +-
SET          0     0     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     0
clock       0| 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35| 0| 1|.....
CLR             0     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     0     0

                       ____  ____  ____  ____  ____  ____  ____  ____  ____  ____  ____  ____  ____  ____  ____               ____
                      /    \/    \/    \/    \/    \/    \/    \/    \/    \/    \/    \/    \/    \/    \/    \             /    \
SDATA        ---------|  0 ||  0 ||  0 || B11|| B10|| B9 || B8 || B7 || B6 || B5 || B4 || B3 || B2 || B1 || B0 |-------------|  0 |
                      \____/\____/\____/\____/\____/\____/\____/\____/\____/\____/\____/\____/\____/\____/\____/             \____/
DMA2         z     z     z     z     z     S     S     S     S     S     S     S     S     S     S     S     S     z     z     z
clock       0| 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35| 0| 1|...

         -------------------+  +--------+  +--------+  +-----------------------------------------------------------------------------
CSS_SCK                     | 1|        | 2|        | 3|
                            +--+        +--+        +--+
SET          0     0     0     1     0     1     0     1     0     0     0     0     0     0     0     0     0     0     0
clock       0| 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35| 0| 1|...
CLR             0     0     1     0     1     0     1     0     0     0     0     0     0     0     0     0     0     0     0

                          _______     _______     _______
                      ////       \////       \////       \
CSS_MOSI -------------||||   B2  |||||   B1  |||||   B0  |---------------------------------------------------------------------------
                      \\\\_______/\\\\_______/\\\\_______/
SET          0     0     B2    0     B1    0     B0    0     0     0     0     0     0     0     0     0     0     0     0
clock       0| 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35| 0| 1|...
CLR             0    /B2    0    /B1    0    /B0    0     1     0     0     0     0     0     0     0     0     0     0     0
```

Figure 2.1: Communication process between PCS array and MCU

Below, the signals and registers from Figure 2.1 are described:

- `CSS_CS`: Chip select signal outputted by the MCU. This signal initiates the data transfer between the ADC and the MCU.
- `SET`: Register to set general-purpose input/output (GPIO) pin high.
- `clock`: The internal clock on which the MCU runs. The clock cycles are numbered in Figure 2.1.
- `CLR`: Register to set general-purpose input/output (GPIO) pin low.
- `ADC_SCK`: The clock signal for the ADCs. It is generated by the MCU. The ADC clock signal controls the timing of the the successive approximation analog-to-digital conversion process and the serial output of the generated bits.
- `SDATA`: These are the successive bits generated by the successive approximation register (SAR) ADC. The MSB is generated first (B11), then the MSB-1 bit (B10) etc. The leading 3 zero bits are the sample-and-hold phase of the ADC. During this phase the ADC samples the analog input signal.
- `DMA2`: Refers to the peripheral: the Direct Memory Access controller, channel 2. It transfers the ADC data from the `SDATA` line to memory in sync with the ADC clock. For the first 3 bits, the `DMA2` is in high impedance/inactive mode (z), as the first 3 bits are always invalid because of the ADC setup time. After the first 3 bits, `DMA2` enters active mode (S).
- `CSS_SCK`: The serial clock signal from the MCU that provides the clock signal for synchronizing the transmission of pixel configuration bits (B2, B1, B0) from the MCU to the PCS array.
- `CSS_MOSI`: The pixel configuration bits generated by the MCU. These bits go via a flipflop chain with 3 flip flops per pixel to the PCS array, which selects the readout line and the clock source for the PCS array.

All output signals from the MCU that are necessary for the communication process shown in Figure 2.1 are generated by writing to the `SET` and `CLR` registers on multiple GPIO pins. As mentioned above, the read and write operations are precisely timed. Figure 2.1 shows that the output and input signals follow a specific sequence and exhibit defined logic levels at designated clock cycles. Therefore, controlling the timing of the write operations in the `SET` and `CLR` registers is crucial for the communication process between the PCS array and the MCU.

The current software implementation that controls this process is called ”scatter-gather DMA”. This approach utilizes the DMA to transfer the set and clear instructions directly from memory to the GPIO pins' `SET` and `CLR` registers. Additionally, it uses the DMA to transfer input data (`SDATA` in Figure 2.1) from the GPIO pin to the memory. These operations are stored in the memory as an array with data transfer descriptors. These descriptors contain information about the source, destination, size, and next descriptor address, enabling autonomous execution of a sequence of memory-to-peripheral or peripheral-to-memory transfer operations.
To control the timing of these operations, Counter Timer (CTIMER) modules are used. These modules contain match registers, which can hold specific predefined values. When the timer value matches the value in the match register, an interrupt will be generated, which initiates a DMA transfer. Hence, the match values essentially determine the timing of the operations stored in the memory and the speed at which the scatter-gather DMA operates.

2.2. Current External Clock

The current implementation relies on external modules. Since this thesis focuses specifically on replacing the external clock module, this section provides background information on that module.

Apart from the signals shown in section 2.1, a differential clock signal is generated. This differential clock signal is used in the *Hardware & Sensor* block in Figure 1.1. The differential clock is a pair of signals that have the same frequency, duty cycle, and are complementary to each other. They are used for charging and discharging the capacitors in the PCS array. In Figure 2.2 the waveform of a differential clock is shown. The external clock module that is currently used to generate this differential clock signal is the AD9552 Oscillator Frequency Upconverter [24].

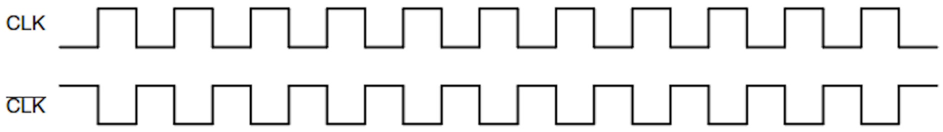


Figure 2.2: Differential Clock

Within the scope of this thesis, the reader is expected to be familiar with the structure and behavior of a differential clock signal, as well as some specifications of the AD9552 module. For further details, the reader is referred to [1].

The following specifications of the AD9552 are relevant:

- The rise/fall time (20% to 80%) in Low Voltage Positive Emitter-Coupled Logic (LVPECL) mode is typically 255 ps and at most 305 ps. In CMOS mode, it is typically 500 ps and at most 745 ps.
- The jitter in the frequency range of 4 MHz to 80 MHz is 0.11 ps rms.

These specifications will serve as a reference for assessing the differential clock signal produced in chapter 6, ultimately eliminating the need for the AD9552 module.

In summary, the current implementation uses the scatter-gather DMA together with the CTIMERs to perform the communication process shown in Figure 2.1 without CPU involvement. Furthermore, the external clock module that is to be replaced is the AD9552, for which the rise/fall time and jitter specifications are known.

3 Program of Requirements

This chapter outlines the project requirements, which are divided into mandatory and trade-off requirements. The mandatory requirements serve as constraints which should not be violated under any circumstances. These requirements are proposed by the project supervisors or stem from the current implementation described in section 2.1. The trade-off requirements describe the objectives for the new implementation, derived from the limitations of the current implementation from section 2.1. The new implementation aims to fulfill these objectives.

3.1. Mandatory Requirements

Functional Requirements

- The software must be programmed on the FRDM-MCXN947 development board from NXP.
- The MCUXpresso IDE must be used to develop the software for the FRDM-MCXN947 development board from NXP.
- The software must replicate the process of the current implementation shown in Figure 2.1 on the FRDM-MCXN947 development board from NXP.
- The software must perform the communication process, shown in Figure 2.1, without CPU involvement.
- The input data from the MCU must ultimately be transferred to the main computer.

System Requirements

- The MCU must communicate with 3 ADCs simultaneously.
- The system must generate 1 `CSS_CS` chip select signals.
- The system must generate 1 `ADC_SCK` clock signals.
- The system must have 3 inputs for 3 `SDATA` input data signals from 3 ADCs. These inputs must be from the same GPIO port.
- The system must generate 1 `CSS_SCK` clock signal.
- The `CSS_SCK` active clock edge should overlap with the data-valid period of the `CSS_MOSI` signal.
- The `CSS_SCK` signal must stay high for 1 pulse width of the `ADC_SCK` signal.
- The system must be capable of transmitting 3 configuration bits.
- The replacements for the external clock must meet or exceed the original specifications.

3.2. Trade-off Requirements

- The new software implementation should be able to react on interventions during runtime.
- The new software implementation should utilize the memory on the FRDM-MCXN947 development board more efficiently than the current scatter-gather DMA approach.
- The use of an external clock module should be eliminated.

4 Tasks

This chapter summarizes the tasks that need to be completed in order to achieve the objectives of this project. These tasks should take the mandatory requirements into account while aiming to fulfill the trade-off requirements.

4.1. Signal Generation

The proposed tasks specified in this section focus on modifying the current implementation to make it responsive to interventions while running. Additionally, these tasks aim to reduce memory usage compared to the current scatter-gather DMA approach described in section 2.1. Therefore, the main task is to search for peripherals on the FRDM-MCXN947 development board that can generate the `CSS_CS`, `ADC_SCK`, `CSS_SCK` and `CSS_MOSI` without using an array with data transfer descriptors stored in memory. To determine whether a peripheral is capable of generating these signals, the following implementation steps are carried out:

1. Generate a single clock signal at a fixed frequency.
2. Generate two synchronized clock signals with equal fixed frequency.
3. Generate two synchronized clock signals with different fixed frequencies.
4. Generate the `ADC_SCK` signal in combination with the `CSS_CS`, `CSS_SCK` or `CSS_MOSI` signal identical to Figure 2.1.

The proposed steps mainly require the generation of two signals only.

4.2. External Clock

The proposed tasks specified in this section focus on the replacement of the external clock mentioned in section 2.2. The following tasks will be executed:

1. Search for peripherals on the FRDM-MCXN947 board that can generate the waveform shown in Figure 2.2.
2. Investigate the feasibility of a differential clock with a variable frequency output.
3. Investigate the feasibility of a differential clock with an input-controlled variable frequency output. ’Input-controlled’ means that the differential clock is able to change frequencies based on an input signal.
4. Assess the generated differential clock signal by comparing its performance to the specifications mentioned in section 2.2.

The subsequent chapters will discuss the approaches and considerations for completing these tasks. The results will then be presented and thoroughly discussed.

5 Peripheral Analysis

This chapter analyzes all the peripherals on the FRDM-MCXN947 board that have been considered for the tasks stated in chapter 4. The analyses are conducted with the mandatory requirements from section 3.1 in mind and aim to meet the trade-off requirements from section 3.2. The chapter begins by examining peripherals that could automate signal generation, followed by an investigation into potential replacements for the external clock. The goal of this chapter is to provide a basis for chapter 6, where the implementation using these peripherals is outlined along with their results.

5.1. Signal Generation

5.1.1. CTIMER

The CTIMER is designed to count cycles from either an internal clock or an external clock that drives the peripheral. At specified counter values stored in match registers, the CTIMER is capable of performing various actions, such as generating an interrupt. There are five instances of the CTIMER peripheral on the FRDM-MCXN947 board, each with 5 outputs. Therefore, a single CTIMER contains a sufficient number of outputs for the signals that must be generated as stated in section 3.1. Furthermore, the CTIMER includes features that can automate signal generation. The key features are:

- The Pulse Width Modulation (PWM) mode. Read Section 48.2 of [6] for a detailed description of the PWM mode.
- DMA linkage, DMA request can be generated when the counter value matches with the match values stored in match registers. Read section 48.3.7 of [6] for a detailed description of DMA requests via the CTIMER module.

The PWM mode allows for PWM outputs from the CTIMER. By configuring the PWM signal with a 50% duty cycle, a clock signal appropriate for the communication process illustrated in Figure 2.1 can be generated. Additionally, the CTIMER's ability to trigger DMA requests on match events allows data transfers to occur at precise time instants. These functionalities operate without requiring CPU intervention or memory usage. The features of the CTIMER appear to align with the functional requirements stated in section 3.1. Hence, the CTIMER peripheral on the FRDM-MCXN947 serves as a suitable option to make the communication process autonomous and more memory efficient.

5.1.2. SCTIMER

The State Controlled Timer or SCTIMER (SCT) is a powerful and flexible timer module capable of creating complex PWM waveforms with minimal or no CPU intervention [6]. It functions similarly to the CTIMER described in subsection 5.1.1, except that its outputs may be determined by states. This means that the SCT can implement state machines, whereas the CTIMER cannot. On the FRDM-MCXN947 board there is one SCT module with 10 outputs. This means that this module also has enough outputs for the signals mentioned in section 3.1. Furthermore, as with most timers, the SCT has a selection of match registers, the ability to generate interrupts, act on events and send DMA requests. The most important features are:

- Ability to use counters with match registers to toggle outputs and create time-proportioned PWM signals (PWM waveforms can change based on current state) [6].
- Ability for selected events to limit, halt, start, or stop a counter or change direction [6].
- Event creation for DMA requests [6].
- Ability for events to trigger state changes, output transitions, timer captures, interrupts, and DMA transactions [6].
- Ability to transition into another state as a result of an event [6].

The aforementioned features are useful for making the software implementation non-autonomous, as mentioned in section 3.2. Furthermore, the ability to work with DMA requests and to create events to trigger DMA transactions allows for a CPU-less process. Thus, complying with the functional requirements mentioned in section 3.1. Hence, the SCT is deemed suitable for implementing the communication process from section 2.1. This suitability in combination with the ability to implement state machines made the SCT module an interesting peripheral to investigate further and to execute the tasks stated in chapter 4 with.

5.2. Replacement of External Clock

From subsection 5.1.1 and subsection 5.1.2, it is known that these peripherals can produce PWM signals without CPU overhead. Both of these may be suitable to produce the differential clock shown in Figure 2.2. However, the SCTIMER is considered more preferable because of its extended functionalities compared to the CTIMER, particularly its support for state machines. This feature can be leveraged to generate a differential clock with a variable frequency output as stated in task 2 in section 4.2. Additionally, the SCTIMER features eight input channels, allowing state transitions triggered by external signals, which can be utilized to implement task 3 in section 4.2. Therefore, the SCTIMER is being evaluated as a potential replacement for the AD9552 module.

6 Implementation and Results

This chapter contains the implementation and results of the peripherals mentioned in chapter 5 with the aim of completing the tasks from chapter 4. The code used to configure the peripherals can be found in Appendix B. The outline of this chapter is as follows: section 6.1 introduces the testing setup used to obtain the results presented in this chapter, while section 6.2 and section 6.3 discuss the implementation and results of the relevant peripherals. In section 6.2 and section 6.3, the implementations are proposed and the results are shown immediately after. This structure makes it clear which implementation was used to obtain each result. Furthermore, this chapter only contains observations and in chapter 7 these observations are explained.

6.1. Testing Setup

The testing setup shown in Figure 6.1 was used to obtain all the waveforms shown in this chapter.

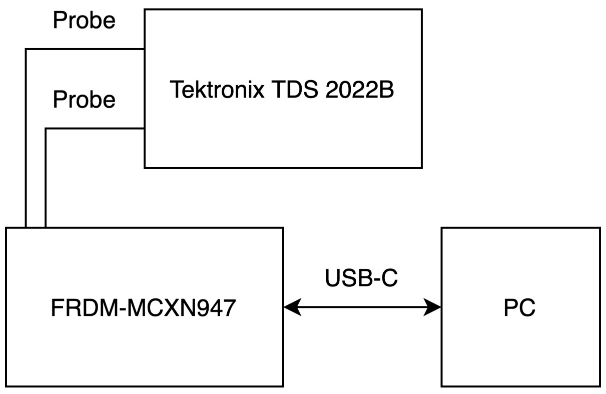


Figure 6.1: Testing setup

The tools used in this testing setup are:

- Oscilloscope: Tektronix TDS 2022B [25].
- Two BNC oscilloscope probes (10x).
- Tinned copper wires.
- USB Type-C cable.

The connections in the testing setup are as follows: a USB Type-C cable is used to connect the computer to the MCU, allowing the MCU to be programmed with the developed software. The MCU's output signals are available through its GPIO pins, which are extended using tinned copper wires to connect BNC oscilloscope probes to the oscilloscope. This testing setup enables visual inspection of the clock signals. The signals are displayed as voltage (y-axis) versus time in seconds (x-axis). Images of the testing setup can be found in section A.1. All accomplished tasks from chapter 4 have been validated using this measurement setup.

6.2. Signal Generation

6.2.1. CTIMER

The CTIMER peripheral on the MCU is configured using the code shown in section B.1, where a detailed explanation of the code implementation is also provided.

For task 1, the code in subsection B.1.1 was developed to generate a single clock signal. The result is shown in Figure 6.2.

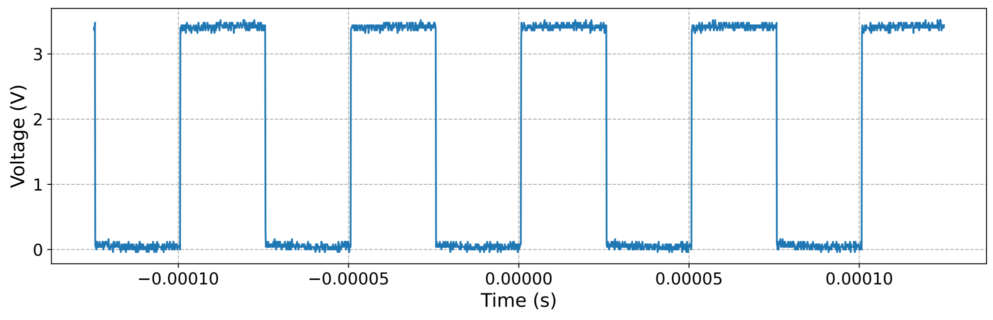


Figure 6.2: CTIMER single signal

The blue waveform from Figure 6.2 shows the single clock signal. The clock signal appears to have a consistent period and pulse width.

For task 2, two implementations have been developed to generate two synchronized clock signals. The first implementation uses a single CTIMER instance as shown in subsection B.1.2, while the second implementation uses two CTIMER instances as shown in subsection B.1.3. The result of the first implementation is shown in Figure 6.3.

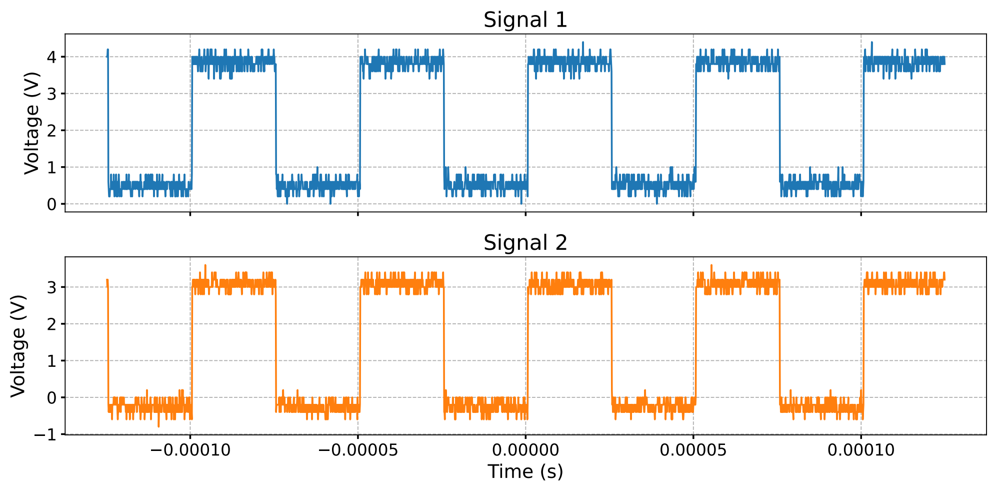


Figure 6.3: One CTIMER with equal frequency outputs

The blue and orange waveforms from Figure 6.3 represent the clock signals from a single CTIMER instance. The clock signals appears to be synchronized with consistent periods and pulse widths.

The result of the second implementation is shown in Figure 6.4.

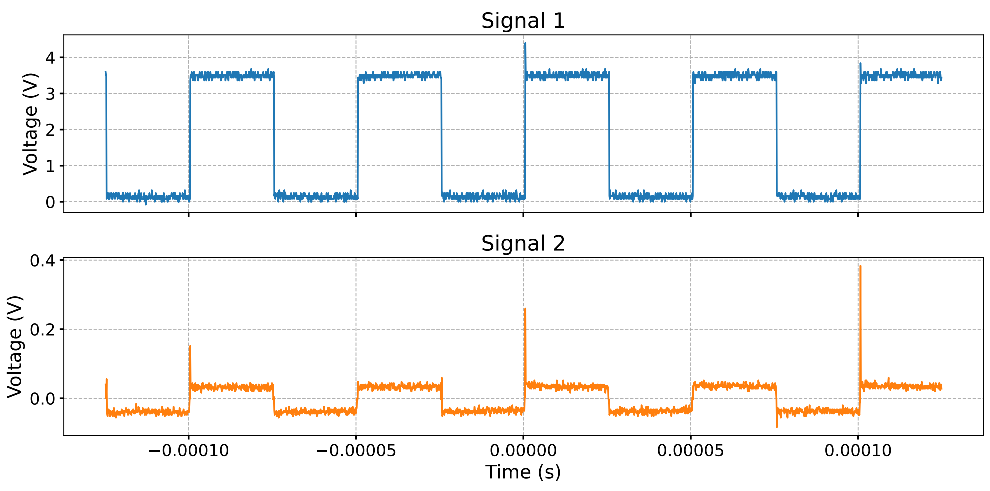


Figure 6.4: Two CTIMERs with equal frequency outputs

The blue and orange waveforms from Figure 6.4 show the clock signals generated from two CTIMER instances. The clock signal at the top appears to be stable with consistent periods and pulse widths. The clock signal at the bottom appears noisy and has low voltage levels. Moreover, the square waveform is distorted and lacks consistency.

For task 3, two implementations have been developed, similarly to task 2. The first implementation uses a single CTIMER and drives different output frequencies to separate channels as shown in subsection B.1.4. The second implementation uses two CTIMERS to output two clock signals of different frequency as shown in subsection B.1.5. The result of the first implementation is shown in Figure 6.5.

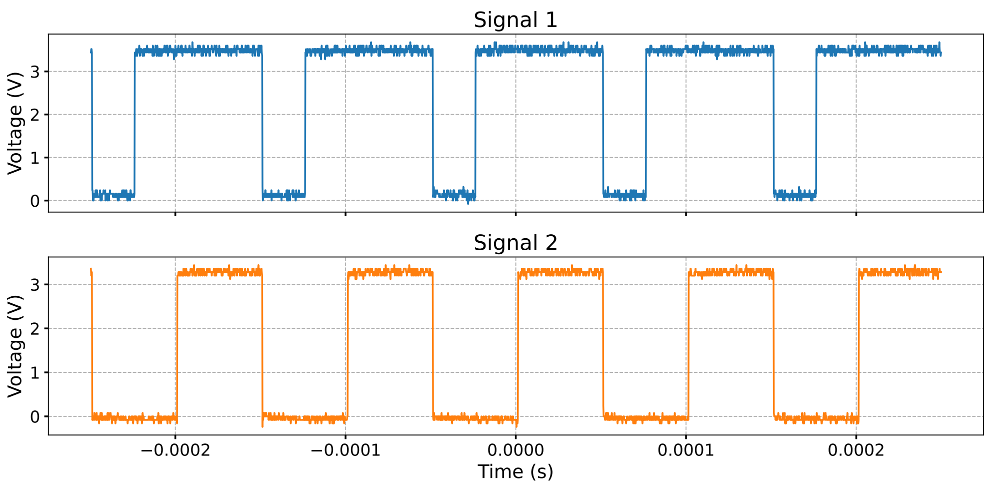


Figure 6.5: One CTIMER with different frequency outputs

The blue and orange waveforms from Figure 6.5 show the clock signals generated from a single CTIMER instance. The clock signal at the top has a duty cycle greater than 50% and the rising edges of the clock signals are not synchronized.

The result of the second implementation is shown in Figure 6.6.

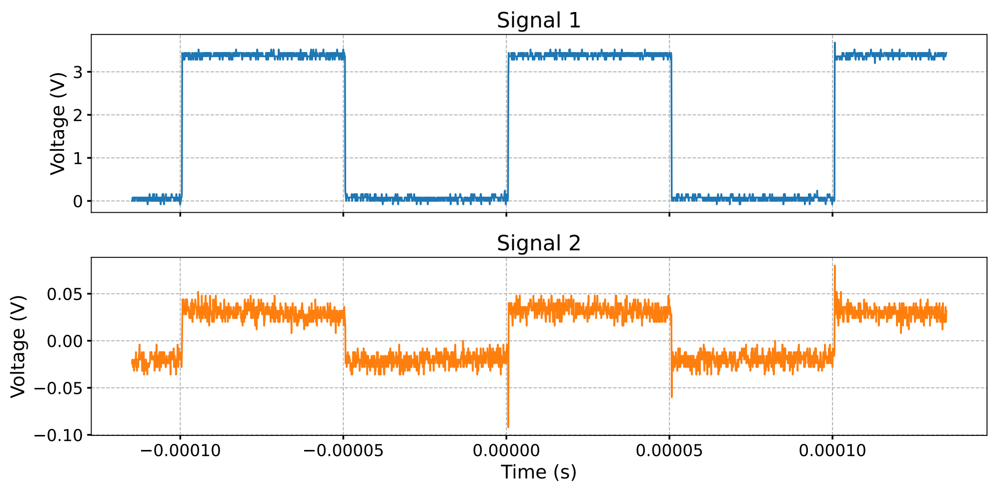


Figure 6.6: Two CTIMERs with different frequency outputs

The blue and orange waveforms from Figure 6.6 display the clock signals generated from two different CTIMER instances. The clock signal at the bottom also appears to be noisy and has low voltage levels. Furthermore, the output frequency does not match its configuration in subsection B.1.5 and the square waveform is inconsistent. The waveforms from both Figure 6.5 and Figure 6.6 deviate from the expected waveforms. Specifically, it was expected to observe two synchronized clock signals with different frequencies.

6.2.2. SCTIMER

For the implementation of the SCT module, the tasks from chapter 4 have been further divided into the paragraphs **Without States** and **With States**. This is because, for tasks 1 and 2 in section 4.1, no state machines are needed, while task 3 is implemented both with and without states. On the other hand, task 4 is only implemented with states. All code related to the SCT can be found in section B.2, where the code is also explained thoroughly.

Without States

For task 1, the code in subsection B.2.1 was developed for generating a single clock signal. The result is shown in Figure 6.7.

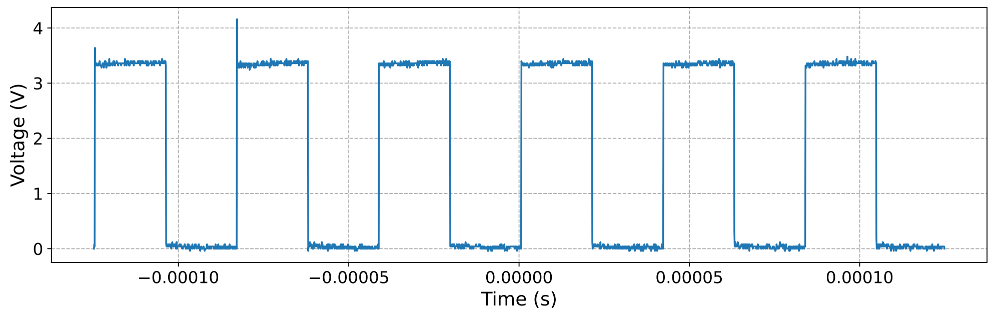


Figure 6.7: SCT single signal

The blue waveform from Figure 6.7 shows the clock signal from the SCTIMER peripheral. The clock signal appears to have a consistent period and regular pulse width.

For task 2, the code in subsection B.2.2 was developed for generating two clock signals of the same frequency. The result is shown Figure 6.8.

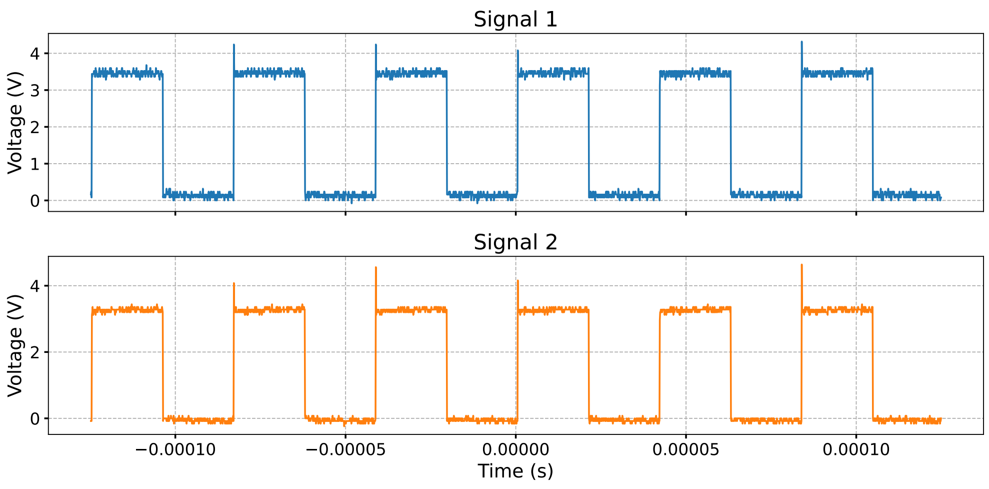


Figure 6.8: SCT with equal frequency outputs

The blue and orange waveforms from Figure 6.8 show the two generated clock signals from the SCTIMER. The clock signals appear stable and well synchronized with consistent periods and pulse widths. Furthermore, at some rising edges overshoot is observed.

For task 3, the code in subsection B.2.3 was developed for generating two synchronized clock signals with different frequencies. The result is shown in Figure 6.9.

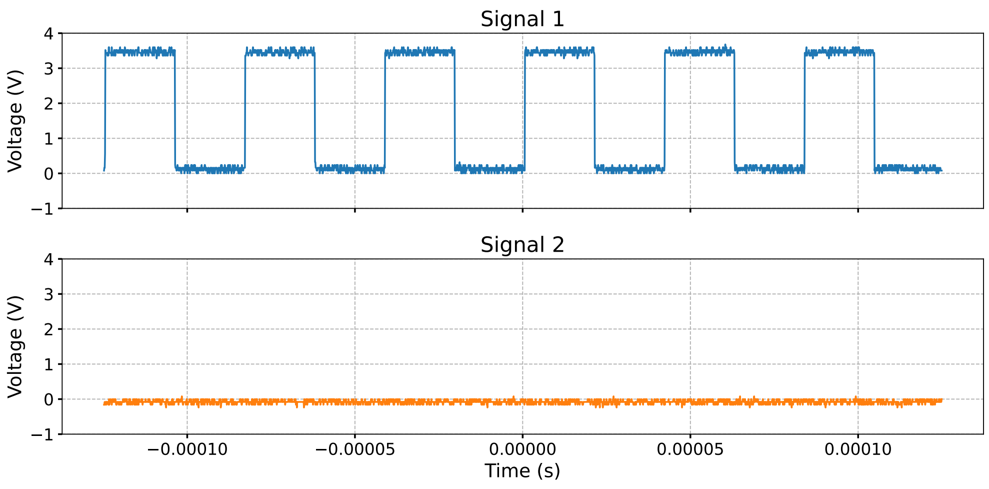


Figure 6.9: SCT with different frequency outputs

The blue and orange waveforms from Figure 6.9 show the output of the SCT when configured for different output frequencies. It can be observed that only the blue waveform has the expected output. The orange waveform on the other hand, has no clock signal.

With States

As mentioned before, task 3 was also implemented with states. For the implementation with states, the finite state diagram (FSD) in Figure 6.10 was designed. In Figure 6.10, 'RE' stands for Rising Edge and `sct_out4` and `sct_out5` are the output signals. This FSD works as follows: `sct_out4` is the signal on which the state transitions are triggered. This signal is a periodic clock signal. After two rising edges of the `sct_out4` signal, `sct_out5` turns high. The `sct_out5` signal goes back to low after two rising edges of the `sct_out4`.

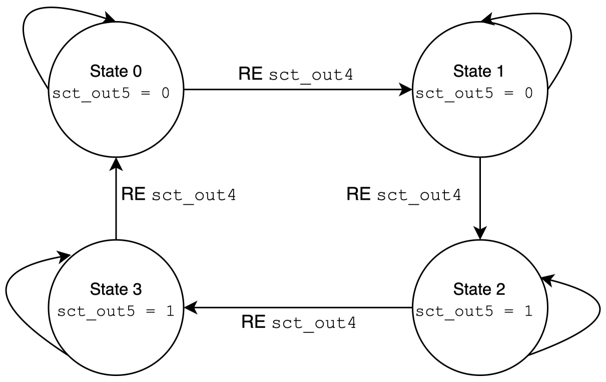


Figure 6.10: FSD for `sct_out4` and `sct_out5`

The implementation shown in Figure 6.10 thus generates two signals with different frequencies, namely the frequency of `sct_out4` is four times that of `sct_out5`. This relation stems from the fact that the signal `sct_out4` was used as the state transition trigger signal/reference signal.

The FSD from Figure 6.10 was implemented in the code shown in subsection B.2.4. The result is shown in Figure 6.11.

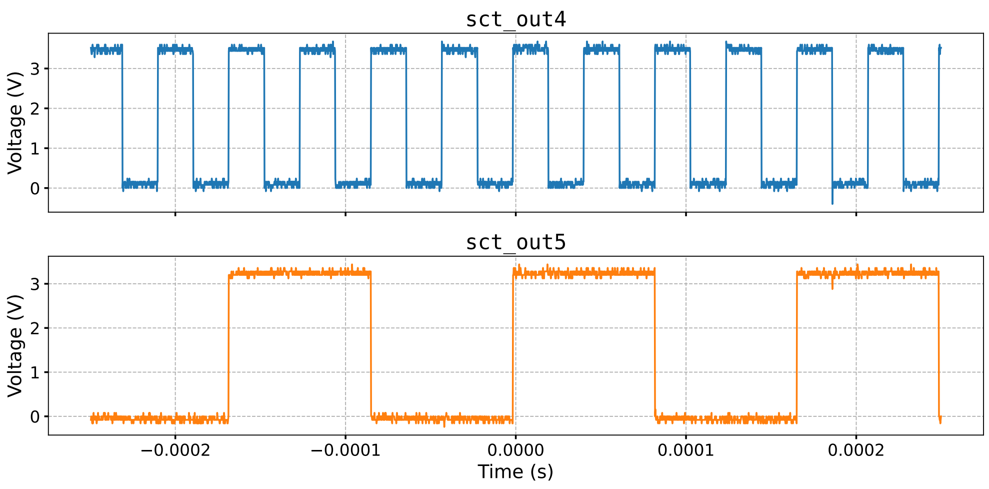


Figure 6.11: `sct_out4` and `sct_out5` obtained with Figure 6.10

The blue and orange waveforms from Figure 6.11 represent the signals `sct_out4` and `sct_out5`. It can also be observed that `sct_out5` has a period that is four times that of `sct_out4`, which was expected. Furthermore, both signals appear to be synchronized and stable.

For task 4, the signals `ADC_SCK` and `CSS_SCK` were chosen for implementation. This decision was based on the similarity between the `CSS_SCK` and the `CSS_MOSI` signal, as well as the simplicity of the `CSS_CS` signal shown in Figure 2.1. Simplicity here refers to the fact that the `CSS_CS` signal remains constantly low during the bit transfer. Therefore, the FSD was designed based on `ADC_SCK` and `CSS_SCK`. The FSD is shown in Figure 6.12, where ”FE” stands for falling edge.

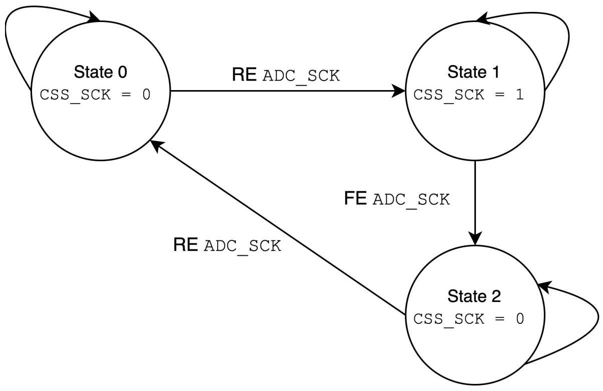


Figure 6.12: FSD for ADC_SCK and CSS_SCK

This FSD works as follows: ADC_SCK is the signal on which the state transitions are triggered. This signal is a periodic clock signal. After one rising edge of the ADC_SCK signal, CSS_SCK turns high. Then at the rising edge of the ADC_SCK signal, the CSS_SCK signal turns low. It stays low for two periods of ADC_SCK, after which the process starts over. This effectively creates the behavior of the CSS_SCK signal as shown in Figure 2.1.

The FSD from Figure 6.12 was thus implemented, however in order to implement this FSD, the CSS_SCK should be able to react on the falling edge of ADC_SCK. When programming this functionality as shown in line 131 in subsection B.2.5, triggering on the falling edge of ADC_SCK did not result in any output. Therefore, it was decided to still implement this FSD with rising edge triggering to verify the design and feasibility of the FSD. This implementation in code is also found in subsection B.2.5. The resulting output is shown in Figure 6.13.

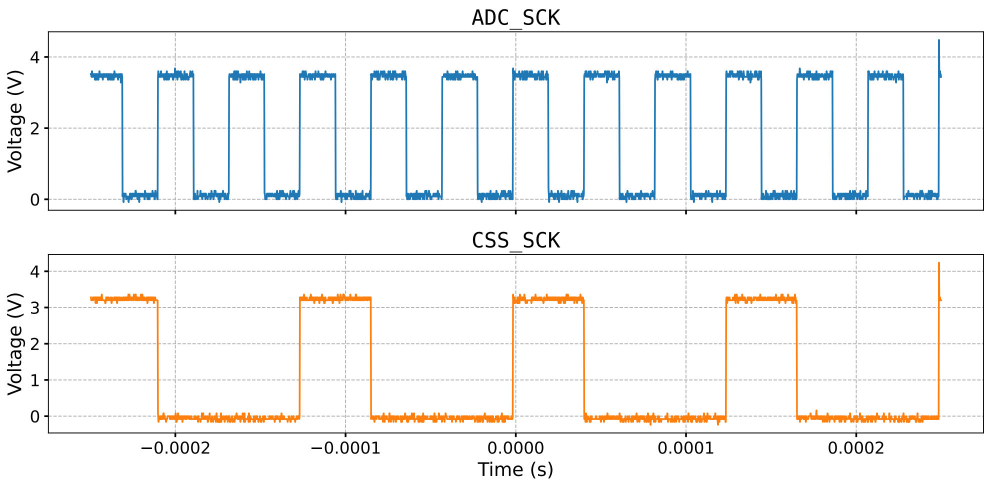


Figure 6.13: ADC_SCK and CSS_SCK realized with Figure 6.12

The blue and orange waveforms from Figure 6.13 display the clock signals ADC_SCK and CSS_SCK. The clock signals appear to be stable and well synchronized with consistent periods and pulse widths. As expected, the Rising edge of the CSS_SCK aligns with the rising edge of the ADC_SCK rather than with the falling edge of the ADC_SCK.

6.3. Differential Clock

The implemented code for the differential clock with SCTIMER is shown in section B.3, where a detailed explanation of the code is also given. Two implementations of the differential clock have been successfully achieved: one with a manually adjustable frequency output shown in subsection B.3.1 and the other with an input-controlled frequency output shown in subsection B.3.2. Both implementations generate the same differential clock signal but are based on different underlying logic. The differential clock signal is wired to the pins on port 2 of the FRDM-MCXN947 board. These pins belong to the fast I/O pins. Their resulting differential clock signal is shown in Figure 6.14.

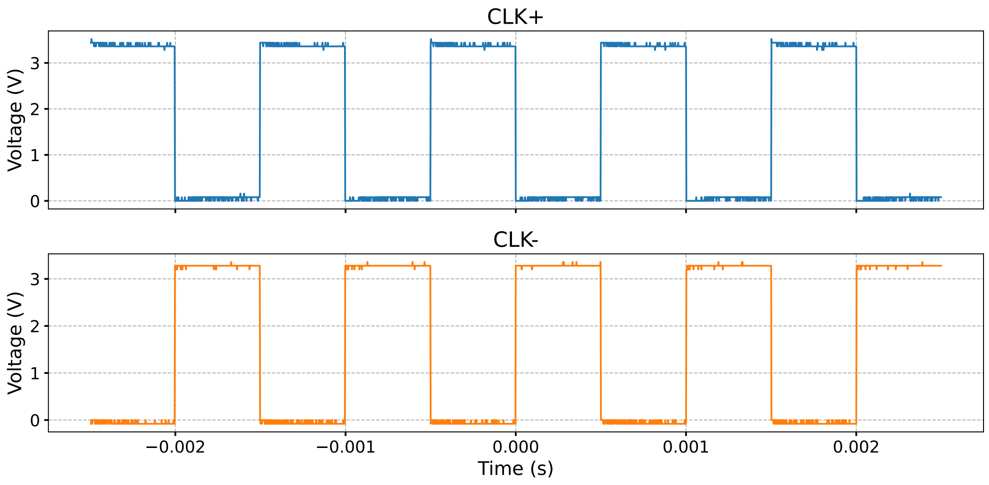


Figure 6.14: Differential clock generation with the SCT

The blue and orange waveforms in Figure 6.14 display the differential clock signals generated by the SCTIMER. The differential clock appears to be stable and quite synchronized with consistent periods and a 50% duty cycle.

To further examine the performance of the generated differential clock, it was decided to zoom in on the rising and falling edges of Figure 6.14 to inspect the rise and fall times. The close-up figures are shown in Figure 6.15 and Figure 6.16. In these figures, red dashed lines have also been plotted at the 20% and 80% values of the clock signal to determine the fall/rise times more easily.

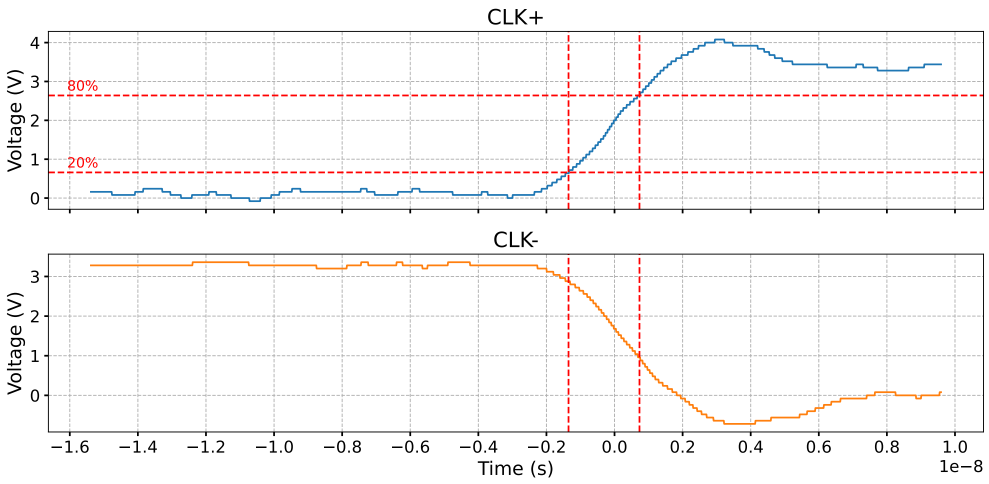


Figure 6.15: Close-up rising edge of the differential clock

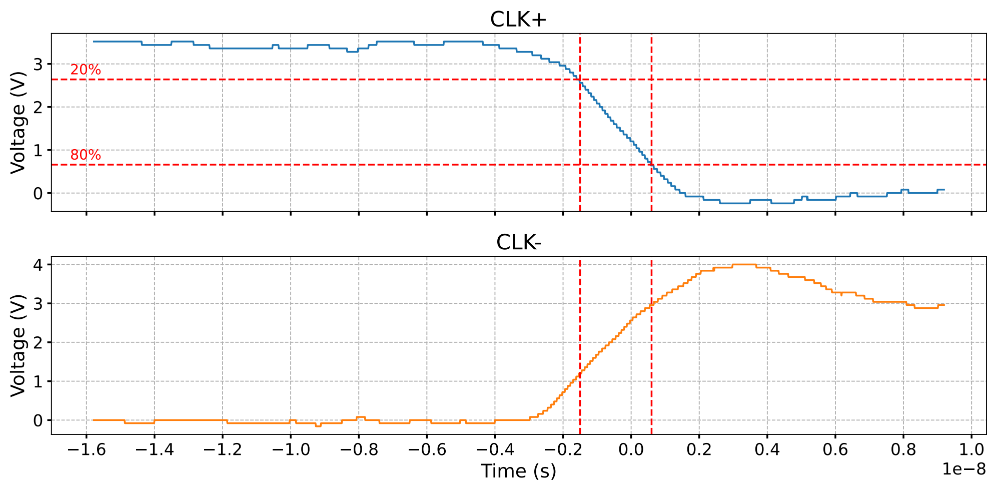


Figure 6.16: Close-up falling edge of the differential clock

From the blue waveform in Figure 6.15, it can be observed that the rise time is around 2 ns and that there is some overshoot after which the signal settles around 3.3 V. As for the blue waveform in Figure 6.16, it can be derived that the fall time is also around 2 ns. This is in line with the general switching specifications of the fast I/O pins as shown in Table 28 in [7].Overshoot is also present at the falling edge, however, this overshoot is less than the overshoot present at the rising edge.

7 Discussion

This chapter discusses the obtained results from chapter 6. The chapter has been split into two parts. In section 7.1 the results from section 6.2 are discussed and in section 7.2 the results from section 6.3 are discussed.

7.1. Signal Generation

The CTIMER and SCT peripherals are tested to investigate their ability to perform tasks 1-4 listed in section 4.1. Accomplishing these tasks are important steps in achieving the trade-off requirement: ”The new software implementation should make the current implementation non-autonomous i.e. it should be able to act on external input signals during runtime”, while maintaining the current functionality of the communication process explained in section 2.1. This section will discuss and evaluate the results in section 6.2.

Figure 6.2 and Figure 6.7 show that the waveforms are clear PWM signals with the intended 50% duty cycle needed for clock generation. The result confirms that both peripherals can successfully generate a single continuous clock signal, thus fulfilling task 1.

For task 2, the CTIMERs were used in two different ways: 1) using a single CTIMER instance and 2) using multiple CTIMER instances. Figure 6.3 shows the result of generating two synchronized clock signals with a single CTIMER instance. The waveforms are synchronized as the rising and falling edges of both clocks are aligned. Furthermore, the output waveforms are consistent square waves, meeting the expected behavior. The result of the second implementation depicted in Figure 6.4 reveals that the clock signals generated by two separate CTIMER instances demonstrate irregular behavior. The clock signal at the bottom appears to be noisy and has low voltage levels close to zero, while a voltage of 3.3 volts is expected for logic highs. Therefore, the output shown in Figure 6.4 does not match the configuration programmed in subsection B.1.3.
This unexpected behavior is suspected to be caused by measuring the wrong GPIO pin. Further research has confirmed this suspicion. In Appendix C it can be seen that the correct clock signal is directed to a different GPIO pin than the one configured. In this appendix this finding is described in more detail. After reviewing the chip design files of the FRDM-MCXN947 and the data sheet [7], it can be concluded that there is a high possibility that the board contains some wiring errors. This would explain for the resulting bottom clock signal in Figure 6.4.
As opposed to the CTIMERs, which were present in multiple, there was only one SCT on the FRDM-MCXN947 board, which meant that it was only possible to perform task 2 using that single module. The result for task 2 of the SCT is shown in Figure 6.8. It can be seen that the results closely resemble that of the implementation with a single CTIMER. Thus, both the CTIMER and SCT are able to perform task 2.

For task 3, the CTIMER was used in the same way as in task 2, namely using two implementations, but with different output frequencies. The result of generating two clock signals with unmatched

frequencies from a single CTIMER instance is depicted in Figure 6.5. This figure displays behavior that was not expected. The top clock signal shows a duty cycle greater than 50%, despite being programmed to 50% as shown in subsection B.1.4. This is because of the fact that each CTIMER instance only contains a single counter shared by all the outputs. The counter resets every time it matches a value in a match register. This poses a problem for generating multiple base frequencies simultaneously, because the counter can only count for one specific period at a time before resetting. As a result, only the output with the shortest period will be timed correctly. In fact, this is the case in Figure 6.5. In subsection B.1.4, the bottom clock signal is programmed to operate at 20 kHz, while the top one is set to 10 kHz. Therefore, the top clock signal exhibits an incorrect waveform.
The result of generating two clock signals with different frequencies from two separate CTIMER instances is shown in Figure 6.6. The result is similar to the implementation with multiple CTIMERs for task 2. It can be observed that the bottom clock signal also appears noisy and has low voltage levels resembling the bottom clock signal in Figure 6.4. Using the same explanation for the result of the implementation with multiple CTIMERs for task 2, the result for the implementation with multiple CTIMERs for task 3 can be explained.

The SCT is implemented with and without the state feature to create different frequency outputs. The stateless implementation encounters the same problem as utilizing a single CTIMER for multiple frequency outputs. From Figure 6.9, it can be observed that the bottom does not show any signal. In subsection B.2.3, the bottom signal is programmed to output a clock signal at 12 kHz and the top one at 24 kHz. The discrepancy between the configuration and the actual output can also be explained by the use of a single counter within the SCT peripheral.
The result of the SCT implementation with states is shown in Figure 6.11. It can be observed that both clock signals have clear square waveforms with 50% duty cycle. The rising edges of the bottom clock signal are well synchronized with those of the top clock signal.
In conclusion, the implementation with states of the SCT is a viable solution to output clock signals at different frequencies, thereby fulfilling task 3.

For task 4, the SCT in combination with states was implemented to generate the `ADC_SCK` and `CSS_SCK` signals. Although the resulting `CSS_SCK` waveform from Figure 6.13 differ from the desired `CSS_SCK`. This difference lies in the fact that the SCT did not output any waveform when configured to react on the rising and falling edges. It is suspected that the nonfunctioning falling edge triggering is caused by incorrect or incomplete initialization of the SCT. However, this result did confirm the feasibility of the proposed FSD design. It is clear that, with this design, the `CSS_SCK` is able to achieve the waveform from Figure 2.1. Namely, a periodic clock signal with highs that are a pulse width of the `ADC_SCK` and lows that are multiple periods of the `ADC_SCK`.

7.2. Differential Clock

This section discusses the results obtained in section 6.3. In section 6.3 differential clocks with a manually adjustable frequency and an input-controlled frequency were generated with the SCT module on the FRDM-MCXN947 board. The resulting differential clock was shown in Figure 6.14.

Taking into account the program of requirements from chapter 3 and the current external clock module from section 2.2, then it can be said that both the manually adjustable and the input-controlled frequency differential clocks generated a differential clock waveform similar to the waveform shown in Figure 2.2. This means that task 1 from section 4.2 is fulfilled by the existence of the SCT. Furthermore, since both the manually adjustable and input-controlled frequency differential clocks are able to produce more than one frequency, it is thus feasible to generate a differential clock with a variable frequency output. Thereby, task 2 from section 4.2 is fulfilled. In fact, not only task 2, but also task 3 from section 4.2 was fulfilled by the use of manually adjustable and input-controlled frequency differential clocks. However, the input-controlled frequency differential clock has one flaw, which is that the implementation utilizes the CPU since it uses the ”while(1)” loop to constantly check for the inputs as shown in subsection B.3.2.

To fulfill task 4 from section 4.2, the performance and quality of the generated differential clock signal was compared to that of the currently used external clock module AD9552 mentioned in section 2.2. This was done by evaluating the fall/rise times and the jitter of both the SCT and the AD9552. From

Figure 6.15 and Figure 6.16 it can be deduced that fall/rise time of the SCT is 2 ns. Compared to the fall/rise time of the AD9552, which ranges from 255 ps to 745 ps as shown in section 2.2, the fall/rise time of the SCT is 2.68 to 7.84 times larger. Though, the measured fall/rise times might be larger because of the capacitive loading by the probes used in the testing setup.
The same comparison was made for the jitter. However, the jitters for both modules were not measured, but taken from the data sheets [7] and [24]. According to [7] the jitter of the SCT is typically 200 ps rms, whereas the jitter of the AD9552 is 0.11 ps rms as mentioned in section 2.2. This means that the jitter of the SCT is approximately 1818 times larger than that of the AD9552.
From the comparison of fall/rise times and jitter of both the modules it can be concluded that the SCT performs worse than the AD9552. Though, further research is needed to assess whether this decrease in quality is significant enough to discard the SCT for generating the differential clock. Furthermore, the SCT is an on-board peripheral as opposed to the external AD9552 module. This also comes into the equation when making the choice between both.

8 Conclusion

This thesis extended the current implementation of the e-nose system by introducing a new MCU board: the FRDM-MCXN947. Building on the existing implementation, a set of tasks was defined to evaluate the capabilities of this new board. To support these tasks, a peripheral analysis was conducted, resulting in the selection of the CTIMER and the SCTIMER as the most suitable peripherals.
Based on the results from task 1 through 3, it can be concluded that the CTIMER peripheral is capable of generating single clock signals and two clock signals with equal frequency when using a single CTIMER instance. However, the use of multiple CTIMER peripherals resulted in unexpected waveform inconsistencies. This is likely caused by an architectural error in the MCU, which resulted in the correct clock signal being wired to a different output pin than the one specified in the code. Furthermore, a single CTIMER failed to generate clock signals of different frequencies due to the constraint of a single shared counter. Thus the CTIMER on its own is not suitable for generating the `CSS_CS`, `ADC_SCK`, and `CSS_SCK` stated in the program of requirements, as this would require the peripheral to output multiple frequencies at once.
As for the SCT, this peripheral is also able to generate single clock signals as well as two clock signals with equal frequency. Although the stateless implementation of task 3, where two signals of different frequencies was the desired output, suffered from the same limitations as the CTIMER because of a single shared counter. The implementation with states via a finite state machine solved this issue. Furthermore, the state-based implementation proved to be successful for generating the `ADC_SCK` and `CSS_SCK`. Though, the falling edge-triggering mechanism did not work as expected, the finite state machine design was feasible and flexible enough to replicate the intended behavior. Ultimately, the unresponsive falling edge-triggering mechanism disqualified the SCT since it failed to generate a `CSS_SCK` that stays high for one pulse width of the `ADC_SCK` signal as mentioned in the program of requirements.

The SCT was also investigated as a potential replacement for the external clock AD9552 that served as a differential clock. The results showed that the SCT produced a correct waveform and had the desired frequency control. Though, the input-controlled version relied on the CPU, which is not in line with the functional requirements mentioned in chapter 3. On top of that, the signal fall/rise times and jitter of the differential clock generated with the SCT is worse than that of the AD9552. This means that not all system requirements are met and thus it can be concluded that the SCT may not be suitable for replacing the AD9552. Despite this, the SCT offers better integration benefits as an on-board peripheral. Its suitability is ultimately dependent on how well the signal quality should be for the application.

8.1. Future Work

This thesis reviewed the CTIMER and SCT thoroughly to accomplish the tasks in chapter 4 and ultimately their suitability to perform the communication process in Figure 2.1. The SCTIMER implementation using states has demonstrated promising results in generating the clock signals needed for the communication process. Further development is recommended to address the falling edge-triggering mechanism.

An alternative peripheral of interest is the enhanced DMA (eDMA) on the FRDM-MCXN947. The eDMA is a more extensive version of the current used DMA on the LPC1769 board and is capable of performing the scatter gather DMA, but uses a different application. The initial steps to perform a single transfer using the eDMA have been taken; see section B.4. For future work, it is advisable to build upon the code implementation in section B.4 and resolve the issue to develop a new scatter-gather DMA approach on the FRDM-MCXN947 board.
Lastly, it would be advised to do further research on the impact of fall/rise times and jitter specifications of the differential clock on PCS array chip. The impact of the limits set by these specifications are yet to be discovered.

A
Figures

A.1. Testing Setup

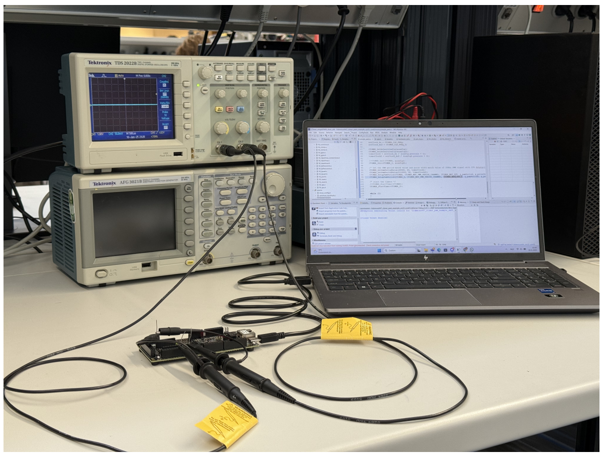


Figure A.1: Testing setup front view

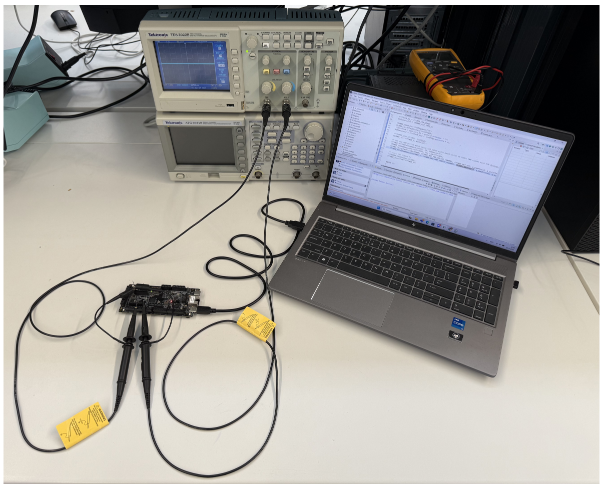


Figure A.2: Testing setup top view

B Code

The code in this appendix is written in C using the MCUXpresso IDE. All of the code originates from source files, therefore the code will not function on its own. To run this code, the peripheral drivers from MCUXpresso must be imported into the project.

B.1. CTIMER

B.1.1. CTIMER Single Signal

This code shows how the CTIMER is programmed to execute task 1 in section 4.1. The CTIMER3 instance is used to generate a single clock signal and output it through Match Output 0. The macros for these are defined in definitions (lines 16-25) alongside the macro for the internal clock frequency. The internal clock frequency is used in the function ”CTIMER_GetPwmPeriodValue” (lines 42-51), which calculates for the match values of the pulse width and the period length. The main function first declares the temporary variables (lines 58-60), followed by the initialization of the FRDM-MCXN947 development board (line 64) and the CTIMER3 peripheral with default configurations (line 70). Consequently, the PWM is configured (line 77) and the CTIMER3 module is activated (line 79).

```
1  /*
2   * Authors: Kevin Pang, Wilson Rong
3   * Date: June 2025
4   */
5  
6  /*******************************************************************************
7   * Includes
8   ******************************************************************************/
9  
10 #include ”fsl_debug_console.h”
11 #include ”board.h”
12 #include ”app.h”
13 #include ”fsl_ctimer.h”
14 
15 /*******************************************************************************
16  * Definitions
17  ******************************************************************************/
18 #ifndef CTIMER_MAT_PWM_PERIOD_CHANNEL
19 #define CTIMER_MAT_PWM_PERIOD_CHANNEL kCTIMER_Match_3
20 #endif
21 
22 /*${macro:start}*/
23 #define CTIMER          CTIMER3         /* Timer 3 */
24 #define CTIMER_MAT_OUT  kCTIMER_Match_0 /* Match output 0 */
25 #define CTIMER_CLK_FREQ CLOCK_GetCTimerClkFreq(2U) /*return Frequency of CTimer
       functional Clock*/
26 
```

```c
/*${macro:end}*/

/*******************************************************************************
 * Prototypes
 ******************************************************************************/

/*******************************************************************************
 * Variables
 ******************************************************************************/
volatile uint32_t g_pwmPeriod   = 0U;
volatile uint32_t g_pulsePeriod = 0U;

/*******************************************************************************
 * Code
 ******************************************************************************/
status_t CTIMER_GetPwmPeriodValue(uint32_t pwmFreqHz, uint8_t dutyCyclePercent,
    uint32_t timerClock_Hz)
{
    /* Calculate PWM period match value */
    g_pwmPeriod = (timerClock_Hz / pwmFreqHz) - 1U;

    /* Calculate pulse width match value */
    g_pulsePeriod = (g_pwmPeriod + 1U) * (100 - dutyCyclePercent) / 100;

    return kStatus_Success;
}

/*!
 * @brief Main function
 */
int main(void)
{
    ctimer_config_t config;
    uint32_t srcClock_Hz;
    uint32_t timerClock;


    /* Init hardware*/
    BOARD_InitHardware();

    /* CTimer0 counter uses the AHB clock, some CTimer1 modules use the Aysnc
        clock */
    srcClock_Hz = CTIMER_CLK_FREQ;


    CTIMER_GetDefaultConfig(&config);

    timerClock = srcClock_Hz / (config.prescale + 1)
    CTIMER_Init(CTIMER, &config);


    /* Get the PWM period match value and pulse width match value of 20Khz PWM
        signal with 50% dutycycle */
    CTIMER_GetPwmPeriodValue(20000, 50, timerClock);
    CTIMER_SetupPwmPeriod(CTIMER, CTIMER_MAT_PWM_PERIOD_CHANNEL, CTIMER_MAT_OUT,
        g_pwmPeriod, g_pulsePeriod, false);
    CTIMER_StartTimer(CTIMER);


    while (1)
    {
```

```
84      }
85  }
```

B.1.2. One CTIMER with Equal Frequency Outputs

This code is an extended version of subsection B.1.1. It programs a single CTIMER to execute task 2 in section 4.1. The CTIMER3 instance is used to generate two synchronized clock signals with the same frequencies. These will be output through Match Output 0 and Match Output 1. This code is extended with an extra macro for match output 1 (line 25) and a second PWM wave (line 82) is added.

```
1   /*
2    * Authors: Kevin Pang, Wilson Rong
3    * Date: June 2025
4    */
5
6   /*******************************************************************************
7    * Includes
8    ******************************************************************************/
9
10  #include "fsl_debug_console.h"
11  #include "board.h"
12  #include "app.h"
13  #include "fsl_ctimer.h"
14
15  /*******************************************************************************
16   * Definitions
17   ******************************************************************************/
18  #ifndef CTIMER_MAT_PWM_PERIOD_CHANNEL
19  #define CTIMER_MAT_PWM_PERIOD_CHANNEL kCTIMER_Match_3
20  #endif
21
22  /*${macro:start}*/
23  #define CTIMER          CTIMER3         /* Timer 3 */
24  #define CTIMER_MAT_OUT  kCTIMER_Match_0 /* Match output 0 */
25  #define CTIMER_MAT_OUT_2  kCTIMER_Match_1 /* Match output 1 */
26  #define CTIMER_CLK_FREQ CLOCK_GetCTimerClkFreq(2U) /*return Frequency of CTimer
        functional Clock*/
27  /*${macro:end}*/
28
29  /*******************************************************************************
30   * Prototypes
31   ******************************************************************************/
32
33  /*******************************************************************************
34   * Variables
35   ******************************************************************************/
36
37
38  volatile uint32_t g_pwmPeriod   = 0U;
39  volatile uint32_t g_pulsePeriod = 0U;
40
41  /*******************************************************************************
42   * Code
43   ******************************************************************************/
44  status_t CTIMER_GetPwmPeriodValue(uint32_t pwmFreqHz, uint8_t dutyCyclePercent,
        uint32_t timerClock_Hz)
45  {
46      /* Calculate PWM period match value */
47      g_pwmPeriod = (timerClock_Hz / pwmFreqHz) - 1U;
48
49      /* Calculate pulse width match value */
```

```c
    g_pulsePeriod = (g_pwmPeriod + 1U) * (100 - dutyCyclePercent) / 100;

    return kStatus_Success;
}


/*!
 * @brief Main function
 */
int main(void)
{
    ctimer_config_t config;
    uint32_t srcClock_Hz;
    uint32_t timerClock;



    /* Init hardware*/
    BOARD_InitHardware();

    /* CTimer0 counter uses the AHB clock, some CTimer1 modules use the Aysnc clock */
    srcClock_Hz = CTIMER_CLK_FREQ;

    CTIMER_GetDefaultConfig(&config);
    timerClock = srcClock_Hz / (config.prescale + 1);


    CTIMER_Init(CTIMER, &config);

    /* Get the PWM period match value and pulse width match value of 20Khz PWM signal with 20% dutycycle */
    CTIMER_GetPwmPeriodValue(20000, 50, timerClock);
    CTIMER_SetupPwmPeriod(CTIMER, CTIMER_MAT_PWM_PERIOD_CHANNEL, CTIMER_MAT_OUT, g_pwmPeriod, g_pulsePeriod, false);
    CTIMER_SetupPwmPeriod(CTIMER_2, CTIMER_MAT_PWM_PERIOD_CHANNEL, CTIMER_MAT_OUT_2, g_pwmPeriod, g_pulsePeriod, false);

    /* start the timers */
    CTIMER_StartTimer(CTIMER);



    while (1)
    {
    }
}
```

B.1.3. Two CTIMERs with Equal Frequency Outputs

This code is an extended version of subsection B.1.2. It programs two CTIMERs to execute task 2 in section 4.1. The CTIMER2 and CTIMER3 instances are used to generate two synchronized clock signals with the same frequencies. These will be output through match output 0 of both CTIMERs. In the definitions, extra macros for CTIMER2 and match output 0 (lines 23 and 24) are added. Furthermore, ”CTIMER_GetPwmPeriodValue” function is duplicated (lines 60-69) to store the match values for two PWM waves. The main function now initializes two CTIMERs, CTIMER2 and CTIMER3 (lines 95 and 96), and configures two PWM waves (lines 99 and 100) in those CTIMERs.

```c
1  /*
2   * Authors: Kevin Pang, Wilson Rong
3   * Date: June 2025
```

```
 */

/*******************************************************************************
 * Includes
 ******************************************************************************/

#include "fsl_debug_console.h"
#include "board.h"
#include "app.h"
#include "fsl_ctimer.h"

/*******************************************************************************
 * Definitions
 ******************************************************************************/
#ifndef CTIMER_MAT_PWM_PERIOD_CHANNEL
#define CTIMER_MAT_PWM_PERIOD_CHANNEL kCTIMER_Match_3
#endif

/*${macro:start}*/
#define CTIMER          CTIMER2         /* Timer 0 */
#define CTIMER_MAT_OUT  kCTIMER_Match_0 /* Match output 0 */


#define CTIMER_2          CTIMER3         /* Timer 0 */
#define CTIMER_MAT_OUT_2  kCTIMER_Match_0 /* Match output 0 */
#define CTIMER_CLK_FREQ CLOCK_GetCTimerClkFreq(2U) /*return Frequency of CTimer
    functional Clock*/
/*${macro:end}*/

/*******************************************************************************
 * Prototypes
 ******************************************************************************/

/*******************************************************************************
 * Variables
 ******************************************************************************/
volatile uint32_t g_pwmPeriod   = 0U;
volatile uint32_t g_pulsePeriod = 0U;

volatile uint32_t g_pwmPeriod2   = 0U;
volatile uint32_t g_pulsePeriod2 = 0U;


/*******************************************************************************
 * Code
 ******************************************************************************/
status_t CTIMER_GetPwmPeriodValue(uint32_t pwmFreqHz, uint8_t dutyCyclePercent,
    uint32_t timerClock_Hz)
{
    /* Calculate PWM period match value */
    g_pwmPeriod = (timerClock_Hz / pwmFreqHz) - 1U;

    /* Calculate pulse width match value */
    g_pulsePeriod = (g_pwmPeriod + 1U) * (100 - dutyCyclePercent) / 100;

    return kStatus_Success;
}

status_t CTIMER_GetPwmPeriodValue2(uint32_t pwmFreqHz, uint8_t dutyCyclePercent,
    uint32_t timerClock_Hz)
{
    /* Calculate PWM period match value */
    g_pwmPeriod2 = (timerClock_Hz / pwmFreqHz) - 1U;

    /* Calculate pulse width match value */
    g_pulsePeriod2 = (g_pwmPeriod2 + 1U) * (100 - dutyCyclePercent) / 100;

    return kStatus_Success;
}

/*!
 * @brief Main function
 */
int main(void)
{
    ctimer_config_t config;
    ctimer_config_t config2;
    uint32_t srcClock_Hz;
    uint32_t timerClock;



    /* Init hardware*/
    BOARD_InitHardware();

    /* CTimer0 counter uses the AHB clock, some CTimer1 modules use the Aysnc
       clock */
    srcClock_Hz = CTIMER_CLK_FREQ;
    srcClock_Hz2 = CTIMER_CLK_FREQ_2;

    CTIMER_GetDefaultConfig(&config);
    CTIMER_GetDefaultConfig(&config2);
    timerClock = srcClock_Hz / (config.prescale + 1);


    CTIMER_Init(CTIMER, &config);
    CTIMER_Init(CTIMER_2, &config2);

    /* Get the PWM period match value and pulse width match value of 20Khz PWM
       signal with 20% dutycycle */
    CTIMER_GetPwmPeriodValue(20000, 50, timerClock);
    CTIMER_GetPwmPeriodValue2(20000, 50, timerClock);
    CTIMER_SetupPwmPeriod(CTIMER, CTIMER_MAT_PWM_PERIOD_CHANNEL, CTIMER_MAT_OUT,
       g_pwmPeriod, g_pulsePeriod, false);
    CTIMER_SetupPwmPeriod(CTIMER_2, CTIMER_MAT_PWM_PERIOD_CHANNEL,
       CTIMER_MAT_OUT_2, g_pwmPeriod2, g_pulsePeriod2, false);

    /* start the timers */
    CTIMER_StartTimer(CTIMER);
    CTIMER_StartTimer(CTIMER_2);


    while (1)
    {
    }
}
```

B.1.4. One CTIMER with Different Frequency Outputs

This code is largely the same as the code in subsection B.1.2 . It programs a single CTIMER to execute task 3 in section 4.1. The CTIMER3 instance is used to generate two synchronized clock signals with different frequencies. These will be output through match output 0 and match output 1.

The function ”CTIMER_GetPwmPeriodValue” is duplicated (lines 58-67) to store the match values for two PWM waves of different frequencies. Furthermore, two PWM waves are configured (lines 95-96) with unequal frequency in the main function.

```
1  /*
2   * Authors: Kevin Pang, Wilson Rong
3   * Date: June 2025
4   */
5
6  /*******************************************************************************
7   * Includes
8   ******************************************************************************/
9
10 #include "fsl_debug_console.h"
11 #include "board.h"
12 #include "app.h"
13 #include "fsl_ctimer.h"
14
15 /*******************************************************************************
16  * Definitions
17  ******************************************************************************/
18 #ifndef CTIMER_MAT_PWM_PERIOD_CHANNEL
19 #define CTIMER_MAT_PWM_PERIOD_CHANNEL kCTIMER_Match_3
20 #endif
21
22 /*${macro:start}*/
23 #define CTIMER          CTIMER3         /* Timer 3 */
24 #define CTIMER_MAT_OUT  kCTIMER_Match_0 /* Match output 0 */
25 #define CTIMER_MAT_OUT_2  kCTIMER_Match_1 /* Match output 1 */
26 #define CTIMER_CLK_FREQ CLOCK_GetCTimerClkFreq(2U) /*return Frequency of CTimer
       functional Clock*/
27
28 /*${macro:end}*/
29
30 /*******************************************************************************
31  * Prototypes
32  ******************************************************************************/
33
34 /*******************************************************************************
35  * Variables
36  ******************************************************************************/
37 volatile uint32_t g_pwmPeriod   = 0U;
38 volatile uint32_t g_pulsePeriod = 0U;
39
40 volatile uint32_t g_pwmPeriod2   = 0U;
41 volatile uint32_t g_pulsePeriod2 = 0U;
42
43
44 /*******************************************************************************
45  * Code
46  ******************************************************************************/
47 status_t CTIMER_GetPwmPeriodValue(uint32_t pwmFreqHz, uint8_t dutyCyclePercent,
       uint32_t timerClock_Hz)
48 {
49     /* Calculate PWM period match value */
50     g_pwmPeriod = (timerClock_Hz / pwmFreqHz) - 1U;
51
52     /* Calculate pulse width match value */
53     g_pulsePeriod = (g_pwmPeriod + 1U) * (100 - dutyCyclePercent) / 100;
54
55     return kStatus_Success;
```

```c
}

status_t CTIMER_GetPwmPeriodValue2(uint32_t pwmFreqHz, uint8_t dutyCyclePercent,
    uint32_t timerClock_Hz)
{
    /* Calculate PWM period match value */
    g_pwmPeriod2 = (timerClock_Hz / pwmFreqHz) - 1U;

    /* Calculate pulse width match value */
    g_pulsePeriod2 = (g_pwmPeriod2 + 1U) * (100 - dutyCyclePercent) / 100;

    return kStatus_Success;
}

/*!
 * @brief Main function
 */
int main(void)
{
    ctimer_config_t config;
    uint32_t srcClock_Hz;
    uint32_t timerClock;


    /* Init hardware*/
    BOARD_InitHardware();

    /* CTimer0 counter uses the AHB clock, some CTimer1 modules use the Aysnc
        clock */
    srcClock_Hz = CTIMER_CLK_FREQ;


    CTIMER_GetDefaultConfig(&config);

    timerClock = srcClock_Hz / (config.prescale + 1);


    CTIMER_Init(CTIMER, &config);


    /* Get the PWM period match value and pulse width match value of 20Khz PWM
        signal with 20% dutycycle */
    CTIMER_GetPwmPeriodValue(10000, 50, timerClock);
    CTIMER_GetPwmPeriodValue2(20000, 50, timerClock);
    CTIMER_SetupPwmPeriod(CTIMER, CTIMER_MAT_PWM_PERIOD_CHANNEL, CTIMER_MAT_OUT,
        g_pwmPeriod, g_pulsePeriod, false);
    CTIMER_SetupPwmPeriod(CTIMER, CTIMER_MAT_PWM_PERIOD_CHANNEL, CTIMER_MAT_OUT_2,
         g_pwmPeriod2, g_pulsePeriod2, false);

    /* start the timers */
    CTIMER_StartTimer(CTIMER);



    while (1)
    {
    }
}
```

B.1.5. Two CTIMERs with Different Frequency Outputs

This code is nearly identical to the one in subsection B.1.3. It programs two CTIMERs to execute task 3 in section 4.1. The CTIMER2 and CTIMER3 instances are used to generate two synchronized clock signals with different frequencies. These will be output through match output 0 of both CTIMERs. Compared to the code in subsection B.1.3, the PWMs are now configured to have two unequal frequency outputs (lines 99 and 100).

```
1  /*
2   * Authors: Kevin Pang, Wilson Rong
3   * Date: June 2025
4   */
5
6  /*******************************************************************************
7   * Includes
8   ******************************************************************************/
9
10 #include "fsl_debug_console.h"
11 #include "board.h"
12 #include "app.h"
13 #include "fsl_ctimer.h"
14
15 /*******************************************************************************
16  * Definitions
17  ******************************************************************************/
18 #ifndef CTIMER_MAT_PWM_PERIOD_CHANNEL
19 #define CTIMER_MAT_PWM_PERIOD_CHANNEL kCTIMER_Match_3
20 #endif
21
22 /*${macro:start}*/
23 #define CTIMER          CTIMER2          /* Timer 2 */
24 #define CTIMER_MAT_OUT  kCTIMER_Match_0 /* Match output 0 */
25
26
27 #define CTIMER_2          CTIMER3          /* Timer 3*/
28 #define CTIMER_MAT_OUT_2  kCTIMER_Match_0 /* Match output 0 */
29 #define CTIMER_CLK_FREQ   CLOCK_GetCTimerClkFreq(2U) /*return Frequency of CTimer
       functional Clock*/
30 /*${macro:end}*/
31
32 /*******************************************************************************
33  * Prototypes
34  ******************************************************************************/
35
36 /*******************************************************************************
37  * Variables
38  ******************************************************************************/
39 volatile uint32_t g_pwmPeriod   = 0U;
40 volatile uint32_t g_pulsePeriod = 0U;
41
42 volatile uint32_t g_pwmPeriod2   = 0U;
43 volatile uint32_t g_pulsePeriod2 = 0U;
44
45
46 /*******************************************************************************
47  * Code
48  ******************************************************************************/
49 status_t CTIMER_GetPwmPeriodValue(uint32_t pwmFreqHz, uint8_t dutyCyclePercent,
       uint32_t timerClock_Hz)
50 {
51     /* Calculate PWM period match value */
52     g_pwmPeriod = (timerClock_Hz / pwmFreqHz) - 1U;
```

```

    /* Calculate pulse width match value */
    g_pulsePeriod = (g_pwmPeriod + 1U) * (100 - dutyCyclePercent) / 100;

    return kStatus_Success;
}

status_t CTIMER_GetPwmPeriodValue2(uint32_t pwmFreqHz, uint8_t dutyCyclePercent,
    uint32_t timerClock_Hz)
{
    /* Calculate PWM period match value */
    g_pwmPeriod2 = (timerClock_Hz / pwmFreqHz) - 1U;

    /* Calculate pulse width match value */
    g_pulsePeriod2 = (g_pwmPeriod2 + 1U) * (100 - dutyCyclePercent) / 100;

    return kStatus_Success;
}

/*!
 * @brief Main function
 */
int main(void)
{
    ctimer_config_t config;
    ctimer_config_t config2;
    uint32_t srcClock_Hz;
    uint32_t timerClock;



    /* Init hardware*/
    BOARD_InitHardware();

    /* CTimer0 counter uses the AHB clock, some CTimer1 modules use the Aysnc
        clock */
    srcClock_Hz = CTIMER_CLK_FREQ;


    CTIMER_GetDefaultConfig(&config);
    CTIMER_GetDefaultConfig(&config2);
    timerClock = srcClock_Hz / (config.prescale + 1);


    CTIMER_Init(CTIMER, &config);
    CTIMER_Init(CTIMER_2, &config2);

    /* Get the PWM period match value and pulse width match value of 20Khz PWM
        signal with 50% dutycycle */
    CTIMER_GetPwmPeriodValue(10000, 50, timerClock);
    CTIMER_GetPwmPeriodValue2(20000, 50, timerClock);
    CTIMER_SetupPwmPeriod(CTIMER, CTIMER_MAT_PWM_PERIOD_CHANNEL, CTIMER_MAT_OUT,
        g_pwmPeriod, g_pulsePeriod, false);
    CTIMER_SetupPwmPeriod(CTIMER_2, CTIMER_MAT_PWM_PERIOD_CHANNEL,
        CTIMER_MAT_OUT_2, g_pwmPeriod2, g_pulsePeriod2, false);

    /* start the timers */
    CTIMER_StartTimer(CTIMER);
    CTIMER_StartTimer(CTIMER_2);

```

```
109     while (1)
110     {
111     }
112 }
```

B.2. SCTIMER

B.2.1. SCT Single Signal

This code shows how the SCTIMER is programmed to execute task 1 in section 4.1. The SCTIMER is used to generate a single clock signal and output it through SCTIMER output 4. In definitions (lines 12-15), the macros for the internal clock frequency and output 4 are defined. The main function first declares the temporary variables (lines 29-32), followed by the initialization of the FRDM-MCXN947 development board (line 35) and the SCTIMER with default configurations (line 40). After that, the PWM output, active level and duty cycle are set (lines 49-51). The if statement in line 54 configures the SCTIMER to produce the single clock signal. Finally, the SCTIMER is activated by starting the timer (line 61).

```
 1 /*
 2  * Authors: Kevin Pang, Wilson Rong
 3  * Date: June 2025
 4  */
 5 
 6 #include "fsl_debug_console.h"
 7 #include "board.h"
 8 #include "app.h"
 9 #include "fsl_sctimer.h"
10 
11 /*******************************************************************************
12  * Definitions
13  ******************************************************************************/
14 #define SCTIMER_CLK_FREQ        CLOCK_GetFreq(kCLOCK_BusClk)
15 #define FIRST_SCTIMER_OUT               kSCTIMER_Out_4
16 
17 /*******************************************************************************
18  * Prototypes
19  ******************************************************************************/
20 
21 /*******************************************************************************
22  * Code
23  ******************************************************************************/
24 /*!
25  * @brief Main function
26  */
27 int main(void)
28 {
29     sctimer_config_t sctimerInfo;
30     sctimer_pwm_signal_param_t pwmParam;
31     uint32_t eventFirstNumberOutput;
32     uint32_t sctimerClock;
33 
34     /* Board pin, clock, debug console init */
35     BOARD_InitHardware();
36 
37     sctimerClock = SCTIMER_CLK_FREQ;
38 
39     /* Default configuration operates the counter in 32-bit mode */
40     SCTIMER_GetDefaultConfig(&sctimerInfo);
41 
42     /* Initialize SCTimer module */
43     SCTIMER_Init(SCT0, &sctimerInfo);
```

```

    /////////////////* Schedule events in current state; State 0 *//////////////

    /* Schedule events for generating a 24KHz PWM with 50% duty cycle from first Out in the current state */
    /* Configure PWM params with frequency 24kHZ from first output */
    pwmParam.output           = FIRST_SCTIMER_OUT;
    pwmParam.level            = kSCTIMER_HighTrue;
    pwmParam.dutyCyclePercent = 50;


    if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm, 24000U, sctimerClock, &eventFirstNumberOutput) ==
        kStatus_Fail)
    {
        return -1;
    }

    /* Start the 32-bit unify timer */
    SCTIMER_StartTimer(SCT0, kSCTIMER_Counter_U);

    while (1)
    {
    }
}
```

B.2.2. SCT with Equal Frequency Outputs

This code is an extended version of the code in subsection B.2.1. It programs the SCTIMER to execute task 2 in section 4.1. The SCTIMER is used to generate two synchronized clock signals with the same frequencies. These will be output through SCTIMER outputs 4 and 5. Therefore, an extra macro is added in definitions for output 5 (line 16). Additionally, a second PWM is configured (line 69) with the same active level, duty cycle and frequency as the first PWM (line 57) in the main function.

```
 1  /*
 2   * Authors: Kevin Pang, Wilson Rong
 3   * Date: June 2025
 4   */
 5
 6  #include "fsl_debug_console.h"
 7  #include "board.h"
 8  #include "app.h"
 9  #include "fsl_sctimer.h"
10
11  /*******************************************************************************
12   * Definitions
13   ******************************************************************************/
14  #define SCTIMER_CLK_FREQ        CLOCK_GetFreq(kCLOCK_BusClk)
15  #define FIRST_SCTIMER_OUT  kSCTIMER_Out_4
16  #define SECOND_SCTIMER_OUT kSCTIMER_Out_5
17  /*******************************************************************************
18   * Prototypes
19   ******************************************************************************/
20
21  /*******************************************************************************
22   * Code
23   ******************************************************************************/
24  /*!
25   * @brief Main function
26   */
27  int main(void)
```

```
{
    sctimer_config_t sctimerInfo;
    sctimer_pwm_signal_param_t pwmParam;
    uint32_t stateNumber;
    uint32_t eventFirstNumberOutput, eventSecondNumberOutput;
    uint32_t sctimerClock;

    /* Board pin, clock, debug console init */
    BOARD_InitHardware();

    sctimerClock = SCTIMER_CLK_FREQ;

    /* Default configuration operates the counter in 32-bit mode */
    SCTIMER_GetDefaultConfig(&sctimerInfo);

    /* Initialize SCTimer module */
    SCTIMER_Init(SCT0, &sctimerInfo);

    stateNumber = SCTIMER_GetCurrentState(SCT0);

    ////////////////* Schedule events in current state; State 0
        *////////////////////

    /* Schedule events for generating a 24KHz PWM with 50% duty cycle from first
        Out in the current state */
    /* Configure PWM params with frequency 24kHZ from first output */
    pwmParam.output            = FIRST_SCTIMER_OUT;
    pwmParam.level             = kSCTIMER_HighTrue;
    pwmParam.dutyCyclePercent = 50;


    if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm, 24000U,
        sctimerClock, &eventFirstNumberOutput) ==
        kStatus_Fail)
    {
        return -1;
    }


    /* Configure PWM params with frequency 24kHZ from second output */
    pwmParam.output            = SECOND_SCTIMER_OUT;
    pwmParam.level             = kSCTIMER_HighTrue;
    pwmParam.dutyCyclePercent = 50;

    if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm, 24000U,
        sctimerClock, &eventSecondNumberOutput) ==
            kStatus_Fail)
    {
        return -1;
    }

    /* Start the 32-bit unify timer */
    SCTIMER_StartTimer(SCT0, kSCTIMER_Counter_U);

    while (1)
    {
    }
}
```

B.2.3. SCT with Different Frequency Outputs

This code is nearly identical to the one in subsection B.2.2. It programs the SCTIMER to execute task 3 in section 4.1. The SCTIMER is used to generate two synchronized clock signals with different frequencies. These will be output through SCTIMER outputs 4 and 5. The PWMs configured (lines 57 and 69) in the main function are now set to two different frequencies.

```
1   /*
2    * Authors: Kevin Pang, Wilson Rong
3    * Date: June 2025
4    */
5
6   #include "fsl_debug_console.h"
7   #include "board.h"
8   #include "app.h"
9   #include "fsl_sctimer.h"
10
11  /*******************************************************************************
12   * Definitions
13   ******************************************************************************/
14  #define SCTIMER_CLK_FREQ        CLOCK_GetFreq(kCLOCK_BusClk)
15  #define FIRST_SCTIMER_OUT  kSCTIMER_Out_4
16  #define SECOND_SCTIMER_OUT kSCTIMER_Out_5
17  /*******************************************************************************
18   * Prototypes
19   ******************************************************************************/
20
21  /*******************************************************************************
22   * Code
23   ******************************************************************************/
24  /*!
25   * @brief Main function
26   */
27  int main(void)
28  {
29      sctimer_config_t sctimerInfo;
30      sctimer_pwm_signal_param_t pwmParam;
31      uint32_t stateNumber;
32      uint32_t eventFirstNumberOutput, eventSecondNumberOutput;
33      uint32_t sctimerClock;
34
35      /* Board pin, clock, debug console init */
36      BOARD_InitHardware();
37
38      sctimerClock = SCTIMER_CLK_FREQ;
39
40      /* Default configuration operates the counter in 32-bit mode */
41      SCTIMER_GetDefaultConfig(&sctimerInfo);
42
43      /* Initialize SCTimer module */
44      SCTIMER_Init(SCT0, &sctimerInfo);
45
46      stateNumber = SCTIMER_GetCurrentState(SCT0);
47
48      /////////////////* Schedule events in current state; State 0
            *////////////////////
49
50      /* Schedule events for generating a 24KHz PWM with 50% duty cycle from first
            Out in the current state */
51      /* Configure PWM params with frequency 24kHZ from first output */
52      pwmParam.output            = FIRST_SCTIMER_OUT;
53      pwmParam.level             = kSCTIMER_HighTrue;
```

```
    pwmParam.dutyCyclePercent = 50;


    if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm, 24000U,
        sctimerClock, &eventFirstNumberOutput) ==
        kStatus_Fail)
    {
        return -1;
    }


    /* Configure PWM params with frequency 12kHZ from second output */
    pwmParam.output            = SECOND_SCTIMER_OUT;
    pwmParam.level             = kSCTIMER_HighTrue;
    pwmParam.dutyCyclePercent = 50;

    if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm, 12000U,
        sctimerClock, &eventSecondNumberOutput) ==
            kStatus_Fail)
    {
        return -1;
    }

    /* Start the 32-bit unify timer */
    SCTIMER_StartTimer(SCT0, kSCTIMER_Counter_U);

    while (1)
    {
    }
}
```

B.2.4. SCT with Different Frequency Outputs Generated with States

This code shows how the SCTIMER is programmed to execute task 3 in section 4.1. The code implementation is in accordance with the finite state diagram in Figure 6.10. The SCTIMER utilizes the state switching feature to generate two synchronized clock signals with different frequencies. These will be output through SCTIMER outputs 4 and 5. Therefore, the macros in the definitions (lines 16-18) are the same as in the previous SCTIMER code. The main function has 4 defined states. The first state configures two PWM waves. The first one is produced on line 67, which is a 24 kHz clock. The second is configured on line 76 as a 24 kHz PWM wave with a 0% duty cycle, effectively resulting in a constant 0 volts output. Accordingly, on line 86 a state switch condition is set to the rising edge of the first PWM wave. When this condition is met, the program will continue in ”second low state” executing the code from line 101-128.
The second state is used primarily to extend the PWM waveforms from the first state by one additional period. Lines 111, 113, 120 and 122 allow for the continuation of the PWM waves configured in the first state. The same state switch condition as state 0 is set on line 128. Satisfying the condition will transition the program to the third state.
In the third state, the second PWM wave is reconfigured on line 150 to a 24 khz PWM wave with a 100% duty cycle which is essentially a constant 3.3-volt output. Furthermore, the first PWM waveform is re-enabled on lines 161 and 163 to ensure its continuation in the second state. The same state transition condition is applied on line 169 and allows the program to move to the fourth state when the condition is met.
Lastly, the fourth state is used to extend the PWM waveforms from the third state by one additional period. Lines 189, 191, 198 and 200 re-enable them. The final state switch condition is also the same as the previous, but it will redirect the program back the first state. This enables the four states to operate continuously.

```
1  /*
2   * Authors: Kevin Pang, Wilson Rong
```

```
 * Date: June 2025
 */

#include "fsl_debug_console.h"
#include "board.h"
#include "app.h"
#include "fsl_sctimer.h"

/*******************************************************************************
 * Definitions
 ******************************************************************************/
/*${macro:start}*/

#define SCTIMER_CLK_FREQ        CLOCK_GetFreq(kCLOCK_BusClk)
#define FIRST_SCTIMER_OUT  kSCTIMER_Out_4
#define SECOND_SCTIMER_OUT kSCTIMER_Out_5

/*${macro:end}*/
/*******************************************************************************
 * Prototypes
 ******************************************************************************/

/*******************************************************************************
 * Code
 ******************************************************************************/
/*!
 * @brief Main function
 */
int main(void)
{
    sctimer_config_t sctimerInfo;
    sctimer_pwm_signal_param_t pwmParam;
    uint32_t stateNumber;
    uint32_t eventFirstNumberOutput, eventSecondNumberOutput, eventNumberInput;
    uint32_t sctimerClock;

    /*these variables used to calculate Match value of period of First PWM on OUT4
         */
    uint32_t period = 0;
    uint32_t pwmFreq_Hz;
    uint32_t period2= 0;
    uint32_t pwmFreq_Hz2;
    uint32_t sctClock;


    /* Board pin, clock, debug console init */
    BOARD_InitHardware();

    sctimerClock = SCTIMER_CLK_FREQ;

    sctClock = sctimerClock / (((SCT0->CTRL & SCT_CTRL_PRE_L_MASK) >>
        SCT_CTRL_PRE_L_SHIFT) + 1U);

    /* Default configuration operates the counter in 32-bit mode */
    SCTIMER_GetDefaultConfig(&sctimerInfo);

    /* Initialize SCTimer module */
    SCTIMER_Init(SCT0, &sctimerInfo);

    stateNumber = SCTIMER_GetCurrentState(SCT0);
////////////////////////////////////////////////////////////////////////////////
```

```c
/* First STATE */
/////////////////////////////////////////////////////////////////////////////////
    /* schedule events for STATE 0 */
    /* Configure PWM params with frequency 24kHZ from first output */
    pwmParam.output           = FIRST_SCTIMER_OUT;
    pwmParam.level            = kSCTIMER_HighTrue;
    pwmParam.dutyCyclePercent = 50;

    /* Schedule events in current state; State 0 */
    /* Schedule events for generating a 24KHz PWM with 50% duty cycle from first
        Out in the current state */
    if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm, 24000U,
        sctimerClock, &eventFirstNumberOutput) ==
        kStatus_Fail)
    {
        return -1;
    }

    /* Schedule events for generating a 24KHz PWM with 0% duty cycle from second
        Out in this new state */
    pwmParam.output           = SECOND_SCTIMER_OUT;
    pwmParam.dutyCyclePercent = 0;
    if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm, 24000U,
        sctimerClock, &eventSecondNumberOutput) ==
        kStatus_Fail)
    {
        return -1;
    }


    pwmFreq_Hz = 24000U; /*input parameter of SCTIMER_SetupPwm 4th */
    period = (sctClock / pwmFreq_Hz) - 1U;

    if (SCTIMER_CreateAndScheduleEvent(SCT0, kSCTIMER_OutputRiseEvent, period,
        kSCTIMER_Out_4, kSCTIMER_Counter_U,
                                       &eventNumberInput) == kStatus_Fail)
    {
        return -1;
    }



    /* Transition to next state when a rising edge is detected on input 1 */
    SCTIMER_SetupNextStateActionwithLdMethod(SCT0, stateNumber + 1,
        eventNumberInput, true);

    /* Go to next state; State 1 */
    SCTIMER_IncreaseState(SCT0);

/////////////////////////////////////////////////////////////////////////////////
/* SECOND STATE */
/////////////////////////////////////////////////////////////////////////////////

    /* Schedule events in State 1 */

        /* Re-enable PWM coming out from Out 4 by scheduling the PWM events in
            this new state */
        /* To get a PWM, the SCTIMER_SetupPwm() function creates 2 events; 1 for
            the pulse period and
         * and 1 for the pulse, we need to schedule both events in this new state
         */
        /* Schedule the period event for the PWM */
        SCTIMER_ScheduleEvent(SCT0, eventFirstNumberOutput);
        /* Schedule the pulse event for the PWM */
        SCTIMER_ScheduleEvent(SCT0, eventFirstNumberOutput + 1);

        /* Re-enable PWM coming out from Out 5 by scheduling the PWM events in
            this new state */
        /* To get a PWM, the SCTIMER_SetupPwm() function creates 2 events; 1 for
            the pulse period and
         * and 1 for the pulse, we need to schedule both events in this new state
         */
        /* Schedule the period event for the PWM */
        SCTIMER_ScheduleEvent(SCT0, eventSecondNumberOutput);
        /* Schedule the pulse event for the PWM */
        SCTIMER_ScheduleEvent(SCT0, eventSecondNumberOutput + 1);

        pwmFreq_Hz2 = 240000U; /*input parameter of SCTIMER_SetupPwm 4th */
        period2 = 2 * ((sctClock / pwmFreq_Hz2) - 1U);

        /*look for rising edge of first PWM*/
        if (SCTIMER_CreateAndScheduleEvent(SCT0, kSCTIMER_OutputRiseEvent, period2
            , kSCTIMER_Out_4, kSCTIMER_Counter_U, &eventNumberInput) ==
            kStatus_Fail)
        {
                return -1;
        }

        /* Transition to next state when a rising edge is detected on input 1 */
        SCTIMER_SetupNextStateActionwithLdMethod(SCT0, stateNumber + 2,
            eventNumberInput, true);

        /* Go to next state; State 2 */
        SCTIMER_IncreaseState(SCT0);



/////////////////////////////////////////////////////////////////////////////////
/* THIRD STATE */
/////////////////////////////////////////////////////////////////////////////////

    /* Schedule events in State 2 */
    /* Schedule events for generating a 24KHz PWM with 100% duty cycle from second
         Out in this new state */
    pwmParam.output            = SECOND_SCTIMER_OUT;
    pwmParam.level             = kSCTIMER_HighTrue;
    pwmParam.dutyCyclePercent = 100;
    if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm, 24000U,
        sctimerClock, &eventSecondNumberOutput) ==
        kStatus_Fail)
    {
        return -1;
    }

    /* Re-enable PWM coming out from Out 4 by scheduling the PWM events in this
        new state */
    /* To get a PWM, the SCTIMER_SetupPwm() function creates 2 events; 1 for the
        pulse period and
     * and 1 for the pulse, we need to schedule both events in this new state
     */
    /* Schedule the period event for the PWM */
    SCTIMER_ScheduleEvent(SCT0, eventFirstNumberOutput);
    /* Schedule the pulse event for the PWM */
    SCTIMER_ScheduleEvent(SCT0, eventFirstNumberOutput + 1);

    pwmFreq_Hz2 = 240000U; /*input parameter of SCTIMER_SetupPwm 4th */
    period2 = 2 * ((sctClock / pwmFreq_Hz2) - 1U);

    /*look for rising edge of first PWM*/
    if (SCTIMER_CreateAndScheduleEvent(SCT0, kSCTIMER_OutputRiseEvent, period2,
        kSCTIMER_Out_4, kSCTIMER_Counter_U, &eventNumberInput) == kStatus_Fail)
    {
        return -1;
    }

    /* Transition to next state when a rising edge is detected on input 1 */
    SCTIMER_SetupNextStateActionwithLdMethod(SCT0, stateNumber + 3,
        eventNumberInput, true);

    /* Go to next state; State 3 */
    SCTIMER_IncreaseState(SCT0);
/////////////////////////////////////////////////////////////////////////////////
/* FOURTH STATE */
/////////////////////////////////////////////////////////////////////////////////
    /* Schedule events in State 3 */

    /* Re-enable PWM coming out from Out 4 by scheduling the PWM events in this
        new state */
    /* To get a PWM, the SCTIMER_SetupPwm() function creates 2 events; 1 for the
        pulse period and
     * and 1 for the pulse, we need to schedule both events in this new state
     */
    /* Schedule the period event for the PWM */
    SCTIMER_ScheduleEvent(SCT0, eventFirstNumberOutput);
    /* Schedule the pulse event for the PWM */
    SCTIMER_ScheduleEvent(SCT0, eventFirstNumberOutput + 1);

    /* Re-enable PWM coming out from Out 5 by scheduling the PWM events in this
        new state */
    /* To get a PWM, the SCTIMER_SetupPwm() function creates 2 events; 1 for the
        pulse period and
     * and 1 for the pulse, we need to schedule both events in this new state
     */
    /* Schedule the period event for the PWM */
    SCTIMER_ScheduleEvent(SCT0, eventSecondNumberOutput);
    /* Schedule the pulse event for the PWM */
    SCTIMER_ScheduleEvent(SCT0, eventSecondNumberOutput + 1);

    pwmFreq_Hz2 = 24000U; /*input parameter of SCTIMER_SetupPwm 4th */
    period2 = 2 * ((sctClock / pwmFreq_Hz2) - 1U);

    /*look for rising edge of first PWM*/
    if (SCTIMER_CreateAndScheduleEvent(SCT0, kSCTIMER_OutputRiseEvent, period2,
        kSCTIMER_Out_4, kSCTIMER_Counter_U,
                                        &eventNumberInput) == kStatus_Fail)
    {
        return -1;
    }

    /* Transition back to State 0 when a rising edge is detected on input 1 */
    /* State 0 has only 1 PWM active, there will be no PWM from Out 2 */
    SCTIMER_SetupNextStateActionwithLdMethod(SCT0, stateNumber, eventNumberInput,
        true);
```

```c

    /* Start the 32-bit unify timer */
    SCTIMER_StartTimer(SCT0, kSCTIMER_Counter_U);

    while (1)
    {
    }
}
```

B.2.5. `ADC_SCK` and `CSS_SCK` generation with SCT

This code is a modified version of the code in subsection B.2.4. The code implementation is in accordance with the finite state diagram in Figure 6.12. The SCTIMER utilizes the state switching feature to generate the `ADC_SCK` and `CSS_SCK` clock similar as in Figure 2.1. However, this code implementation configures an active-high clock for `ADC_SCK`, instead of the active-low clock shown in Figure 2.1. The clock signals are output through SCTIMER outputs 4 and 5. Therefore, the macros in definitions (lines 16-18) are identical to the previous SCTIMER code. The main function consists of three states. The first state configures the `ADC_SCK` on line 71 and will be re-enabled in the other states to run continuously. On line 80, the `CSS_SCK` is programmed to stay at 0 volts for a single period of the `ADC_SCK` until the rising edge of the `ADC_SCK` is detected. This condition is defined on line 90.
In the second state, `CSS_SCK` is set high to 3.3 volts on line 112. Ideally, the transition from the second state to the third state should occur at the first falling edge of `ADC_SCK`, ensuring that `CSS_SCK` is pulled down to 0 volts at the correct moment. However, this state transition condition did not function as intended. For review purposes, the condition on line 131 is configured to trigger on the rising edge of `ADC_SCK`.
The third state mirrors the first state. On line 153, `CSS_SCK` is pulled low. The state transition condition, defined on line 174, is set to trigger on the rising edge of `ADC_SCK`. Once this condition is met, the program transitions back to the first state, enabling the three states to operate continuously.

```c
1  /*
2   * Authors: Kevin Pang, Wilson Rong
3   * Date: June 2025
4   */
5
6  #include "fsl_debug_console.h"
7  #include "board.h"
8  #include "app.h"
9  #include "fsl_sctimer.h"
10
11 /*******************************************************************************
12  * Definitions
13  ******************************************************************************/
14 /*${macro:start}*/
15
16 #define SCTIMER_CLK_FREQ        CLOCK_GetFreq(kCLOCK_BusClk)
17 #define FIRST_SCTIMER_OUT  kSCTIMER_Out_4
18 #define SECOND_SCTIMER_OUT kSCTIMER_Out_5
19 /*${macro:end}*/
20 /*******************************************************************************
21  * Prototypes
22  ******************************************************************************/
23
24 /*******************************************************************************
25  * Code
26  ******************************************************************************/
27 /*!
28  * @brief Main function
29  */
30 int main(void)
31 {
    sctimer_config_t sctimerInfo;
    sctimer_pwm_signal_param_t pwmParam;
    uint32_t stateNumber;
    uint32_t eventFirstNumberOutput, eventSecondNumberOutput, eventNumberInput;
    uint32_t sctimerClock;

    /*these variables used to calculate Match value of period of First PWM on OUT4
         */
    uint32_t period = 0;
    uint32_t pwmFreq_Hz;
    uint32_t period2= 0;
    uint32_t pwmFreq_Hz2;
    uint32_t sctClock;


    /* Board pin, clock, debug console init */
    BOARD_InitHardware();

    sctimerClock = SCTIMER_CLK_FREQ;

    sctClock = sctimerClock / (((SCT0->CTRL & SCT_CTRL_PRE_L_MASK) >>
        SCT_CTRL_PRE_L_SHIFT) + 1U);

    /* Default configuration operates the counter in 32-bit mode */
    SCTIMER_GetDefaultConfig(&sctimerInfo);

    /* Initialize SCTimer module */
    SCTIMER_Init(SCT0, &sctimerInfo);

    stateNumber = SCTIMER_GetCurrentState(SCT0);
////////////////////////////////////////////////////////////////////////////////
/* FIRST STATE */
////////////////////////////////////////////////////////////////////////////////
    /* schedule events for STATE 0 */
    /* Configure PWM params with frequency 24kHZ from first output */
    pwmParam.output            = FIRST_SCTIMER_OUT;
    pwmParam.level             = kSCTIMER_HighTrue;
    pwmParam.dutyCyclePercent = 50;

    /* Schedule events in current state; State 0 */
    /* Schedule events for generating a 24KHz PWM with 50% duty cycle from first
        Out in the current state */
    if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm, 24000U,
        sctimerClock, &eventFirstNumberOutput) ==
        kStatus_Fail)
    {
        return -1;
    }

    /* Schedule events for generating a 24KHz PWM with 0% duty cycle from second
        Out in this new state */
    pwmParam.output            = SECOND_SCTIMER_OUT;
    pwmParam.dutyCyclePercent = 0;
    if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm, 24000U,
        sctimerClock, &eventSecondNumberOutput) ==
        kStatus_Fail)
    {
        return -1;
    }


    pwmFreq_Hz = 24000U; /*input parameter of SCTIMER_SetupPwm 4th */
    period = (sctClock / pwmFreq_Hz) - 1U;

    if (SCTIMER_CreateAndScheduleEvent(SCT0, kSCTIMER_OutputRiseEvent, period,
        kSCTIMER_Out_4, kSCTIMER_Counter_U,
                                        &eventNumberInput) == kStatus_Fail)
    {
        return -1;
    }



    /* Transition to next state when a rising edge is detected on input 1 */
    SCTIMER_SetupNextStateActionwithLdMethod(SCT0, stateNumber + 1,
        eventNumberInput, true);

    /* Go to next state; State 1 */
    SCTIMER_IncreaseState(SCT0);

////////////////////////////////////////////////////////////////////////////////
/* SECOND STATE */
////////////////////////////////////////////////////////////////////////////////
/* Schedule events in State 1 */

        /* Schedule events for generating a 24KHz PWM with 100% duty cycle from
            second Out in this new state */
        pwmParam.output           = SECOND_SCTIMER_OUT;
        pwmParam.dutyCyclePercent = 100;
        if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm, 24000U,
            sctimerClock, &eventSecondNumberOutput) ==
                kStatus_Fail)
        {
                return -1;
        }

        /* Re-enable PWM coming out from Out 4 by scheduling the PWM events in
            this new state */
        /* To get a PWM, the SCTIMER_SetupPwm() function creates 2 events; 1 for
            the pulse period and
         * and 1 for the pulse, we need to schedule both events in this new state
         */
        /* Schedule the period event for the PWM */
        SCTIMER_ScheduleEvent(SCT0, eventFirstNumberOutput);
        /* Schedule the pulse event for the PWM */
        SCTIMER_ScheduleEvent(SCT0, eventFirstNumberOutput + 1);

        pwmFreq_Hz2 = 240000U; /*input parameter of SCTIMER_SetupPwm 4th */
        period2 = 2 * ((sctClock / pwmFreq_Hz2) - 1U);

        /*look for rising edge of first PWM*/
        if (SCTIMER_CreateAndScheduleEvent(SCT0, kSCTIMER_OutputRiseEvent, period,
             kSCTIMER_Out_4, kSCTIMER_Counter_U,&eventNumberInput) == kStatus_Fail)
        {
                return -1;
        }

        /* Transition to next state when a rising edge is detected on input 1 */
        SCTIMER_SetupNextStateActionwithLdMethod(SCT0, stateNumber + 2,
            eventNumberInput, true);

        /* Go to next state; State 2 */
        SCTIMER_IncreaseState(SCT0);



////////////////////////////////////////////////////////////////////////////////
/* THIRD STATE */
////////////////////////////////////////////////////////////////////////////////

    /* Schedule events in State 2 */
    /* Schedule events for generating a 24KHz PWM with 0% duty cycle from second
        Out in this new state */
    pwmParam.output            = SECOND_SCTIMER_OUT;
    pwmParam.level             = kSCTIMER_HighTrue;
    pwmParam.dutyCyclePercent = 0;
    if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm, 24000U,
        sctimerClock, &eventSecondNumberOutput) ==
        kStatus_Fail)
    {
        return -1;
    }

    /* Re-enable PWM coming out from Out 4 by scheduling the PWM events in this
        new state */
    /* To get a PWM, the SCTIMER_SetupPwm() function creates 2 events; 1 for the
        pulse period and
     * and 1 for the pulse, we need to schedule both events in this new state
     */
    /* Schedule the period event for the PWM */
    SCTIMER_ScheduleEvent(SCT0, eventFirstNumberOutput);
    /* Schedule the pulse event for the PWM */
    SCTIMER_ScheduleEvent(SCT0, eventFirstNumberOutput + 1);



    pwmFreq_Hz2 = 240000U; /*input parameter of SCTIMER_SetupPwm 4th */
    period2 = 2 * ((sctClock / pwmFreq_Hz2) - 1U);

    /*look for rising edge of first PWM*/
    if (SCTIMER_CreateAndScheduleEvent(SCT0, kSCTIMER_OutputRiseEvent, period,
        kSCTIMER_Out_4, kSCTIMER_Counter_U,
                                       &eventNumberInput) == kStatus_Fail)
    {
        return -1;
    }

    /* Transition to next state when a rising edge is detected on input 1 */
    SCTIMER_SetupNextStateActionwithLdMethod(SCT0, stateNumber, eventNumberInput,
        true);

    /* Start the 32-bit unify timer */
    SCTIMER_StartTimer(SCT0, kSCTIMER_Counter_U);

    while (1)
    {
    }
}
```

B.3. Differential Clock

B.3.1. Manually Adjustable Differential Clock with SCT

This code shows how the manually adjustable differential clock is implemented to execute task 2 in section 4.2. The SCTIMER is utilized to generate a differential clock signal with a fixed, predefined frequency. The two clock signals are output through SCTIMER output 4 and 5. In definitions (lines 14-16), the macros for these outputs and the internal clock frequency are defined. The main function first declares the temporary variables (lines 33-36), followed by the initialization of the FRDM-MCXN947 development board (line 41) and the SCTIMER with default configurations (line 51). The function ”ConfigDifferentialClock” (line 54) sets up the differential clock, with its input parameter specifying the desired frequency. The clocks are configured (lines 62-77) to operate at the same frequency but with opposite active levels. The ”Start” function starts the timer of the SCTIMER, initiating the operation of the differential clock. These functions are called in the ”while(1)” loop to continuously generate the differential clock.

```
1  /*
2   * Authors: Kevin Pang, Wilson Rong
3   * Date: June 2025
4   */
5  
6  #include ”fsl_debug_console.h”
7  #include ”board.h”
8  #include ”app.h”
9  #include ”fsl_sctimer.h”
10 
11 /*******************************************************************************
12  * Definitions
13  ******************************************************************************/
14 #define SCTIMER_CLK_FREQ        CLOCK_GetFreq(kCLOCK_BusClk)
15 #define FIRST_SCTIMER_OUT  kSCTIMER_Out_4
16 #define SECOND_SCTIMER_OUT kSCTIMER_Out_5
17 /*******************************************************************************
18  * Define the PORT/PINS to read the command from *
19  ******************************************************************************/
20 
21 /*******************************************************************************
22  * Prototypes
23  ******************************************************************************/
24 
25 /*******************************************************************************
26  * Code
27  ******************************************************************************/
28 /*!
29  * @brief Main function
30  */
31 int main(void)
32 {
33     sctimer_config_t sctimerInfo;
34     sctimer_pwm_signal_param_t pwmParam;
35     uint32_t eventFirstNumberOutput, eventSecondNumberOutput;
36     uint32_t sctimerClock;
37 
38 
39 
40     /* Board pin, clock, debug console init */
41     BOARD_InitHardware();
42 
43 
44     sctimerClock = SCTIMER_CLK_FREQ;
45 
46 
```

```
    /* Default configuration operates the counter in 32-bit mode */
    SCTIMER_GetDefaultConfig(&sctimerInfo);

    /* Initialize SCTimer module */
    SCTIMER_Init(SCT0, &sctimerInfo);

    /* function to initialize and configure the differential clock */
    void ConfigDifferentialClock(uint8_t frequency)
    {

        // Disable/Reset SCTimer before reconfiguring
           SCTIMER_StopTimer(SCT0, kSCTIMER_Counter_U);


        /* Configure PWM params for first clock output */
           pwmParam.output            = FIRST_SCTIMER_OUT;
           pwmParam.level             = kSCTIMER_HighTrue;
           pwmParam.dutyCyclePercent = 50;


           if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm,
               frequency, sctimerClock, &eventFirstNumberOutput) ==
               kStatus_Fail)
           {

           }

         /* Configure PWM params for second clock output */
           pwmParam.output            = SECOND_SCTIMER_OUT;
           pwmParam.level             = kSCTIMER_LowTrue;
           pwmParam.dutyCyclePercent = 50;
           if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm,
               frequency, sctimerClock, &eventSecondNumberOutput) ==
               kStatus_Fail)
           {

           }
    }

    /* function to kick start the SCTIME for differential clock output */
    void Start(void){
                SCTIMER_StartTimer(SCT0, kSCTIMER_Counter_U);
    }

    while (1)
    {
                ConfigDifferentialClock(24000);
                Start();


    }
}
```

B.3.2. Input-based Adjustable Differential Clock with SCT

This code is an extended version of the code in subsection B.3.1. It programs the SCTIMER to execute task 3 in section 4.2.The SCTIMER is used to generate a differential clock, with its frequency determined by the inputs of three GPIO pins. The two clock signals are output through SCTIMER output 4 and 5. In definitions (lines 14-16), the macros for these outputs and the internal clock frequency are defined. Additionally, the macros defined on lines 24–31 specify which pin corresponds to each

input bit for clarity. The main function first declares the temporary variables (lines 44-46), followed by the initialization of the FRDM-MCXN947 development board (line 50) and the SCTIMER with default configurations (line 58). The array ”frequencyTable” (line 61) stores some values for the frequency of the differential clock. The function ”UpdateDifferentialClock” (line 72) is similar to the ”ConfigDifferentialClock” function in subsection B.3.1, but modified to read a 3-bit input parameter. The 3-bit input is constructed in the function ”Read3BitCommandFromGPIO” (line 117). The logic level at port 1 pin 2 defines the most significant bit (MSB), port 1 pin 13 defines the center bit and port 1 pin 12 defines the least significant bit (LSB). The function ”StartDifferentialClock” (line 126) starts the timer of the SCTIMER, initiating the operation of the differential clock.
All these functions are used in the ”while(1)” (line 133) loop to provide a continuous input controlled differential clock. First, the logic levels of the input pins are polled. If a change in logic level in the pins are detected, then the frequency of the differential clock will be updated accordingly to one of the values in the ”frequencyTable” array.

```
1  /*
2   * Authors: Kevin Pang, Wilson Rong
3   * Date: June 2025
4   */
5
6  #include ”fsl_debug_console.h”
7  #include ”board.h”
8  #include ”app.h”
9  #include ”fsl_sctimer.h”
10
11 /*******************************************************************************
12  * Definitions
13  ******************************************************************************/
14 /*${macro:start}*/
15
16 #define SCTIMER_CLK_FREQ        CLOCK_GetFreq(kCLOCK_BusClk)
17 #define FIRST_SCTIMER_OUT  kSCTIMER_Out_4
18 #define SECOND_SCTIMER_OUT kSCTIMER_Out_5
19 /*${macro:end}*/
20
21 /*******************************************************************************
22  * Define the PORT/PINS to read the command from *
23  ******************************************************************************/
24 #define CMD_BIT0_PORT    1U
25 #define CMD_BIT0_PIN     12U   // e.g. P1_12
26
27 #define CMD_BIT1_PORT    1U
28 #define CMD_BIT1_PIN     13U   // e.g. P1_13
29
30 #define CMD_BIT2_PORT    1U
31 #define CMD_BIT2_PIN     2U   // e.g. P1_2
32 /*******************************************************************************
33  * Prototypes
34  ******************************************************************************/
35
36 /*******************************************************************************
37  * Code
38  ******************************************************************************/
39 /*!
40  * @brief Main function
41  */
42 int main(void)
43 {
44     sctimer_config_t sctimerInfo;
45     uint32_t eventFirstNumberOutput, eventSecondNumberOutput;
46     uint32_t sctimerClock;
```

```


    /* Board pin, clock, debug console init */
    BOARD_InitHardware();

    sctimerClock = SCTIMER_CLK_FREQ;

    /* Default configuration operates the counter in 32-bit mode */
    SCTIMER_GetDefaultConfig(&sctimerInfo);

    /* Initialize SCTimer module */
    SCTIMER_Init(SCT0, &sctimerInfo);

    /* An array to store some frequency values for the differential clock */
    const uint32_t frequencyTable[8] = {
        24000, // 1 kHz
        2000000, // 2 MHz
        4000000, // 4 MHz
        8000000, // 8 MHz
        10000000, // 10 MHz
        12000000, // 12 MHz
        16000000, // 16 MHz
        20000000  // 20 MHz
    };
    /* function to initialize and configure the differential clock */
    void UpdateDifferentialClock(uint8_t freqIndex)
    {
        if (freqIndex >= 8) return; // invalid input

           uint32_t frequency = frequencyTable[freqIndex];

        // Disable/Reset SCTimer before reconfiguring
           SCTIMER_StopTimer(SCT0, kSCTIMER_Counter_U);

           /* Default reconfiguration operates the counter in 32-bit mode */
           SCTIMER_GetDefaultConfig(&sctimerInfo);

           /* Reinitialize SCTimer module */
           SCTIMER_Init(SCT0, &sctimerInfo);

           sctimer_pwm_signal_param_t pwmParam;
       /* Configure PWM params for first clock output */
           pwmParam.output            = FIRST_SCTIMER_OUT;
           pwmParam.level             = kSCTIMER_HighTrue;
           pwmParam.dutyCyclePercent = 50;

           if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm,
               frequency, sctimerClock, &eventFirstNumberOutput) ==
               kStatus_Fail)
           {

           }
       /* Configure PWM params for second clock output */
           pwmParam.output            = SECOND_SCTIMER_OUT;
           pwmParam.level             = kSCTIMER_LowTrue;
           pwmParam.dutyCyclePercent = 50;
           if (SCTIMER_SetupPwm(SCT0, &pwmParam, kSCTIMER_EdgeAlignedPwm,
               frequency, sctimerClock, &eventSecondNumberOutput) ==
               kStatus_Fail)
           {

           }
    }

    /* function to read logic levels of GPIO pins P1_2, P1_13 and P1_12*/
    /* The 3 bit input is used to select the frequency in the frequency table array */
    /* The 3 bit input is read in the sketch shown below */
    /*
     * MSB ----------> lSB
     * Bit2 | Bit1| Bit0 |
     * P1_2 |P1_13| P1_12|
     */
    uint8_t Read3BitCommandFromGPIO(void)
    {
        uint8_t bit0 = GPIO_PinRead(GPIO1, CMD_BIT0_PIN) ? 1U : 0U;
        uint8_t bit1 = GPIO_PinRead(GPIO1, CMD_BIT1_PIN) ? 1U : 0U;
        uint8_t bit2 = GPIO_PinRead(GPIO1, CMD_BIT2_PIN) ? 1U : 0U;
        return (bit2 << 2) | (bit1 << 1) | bit0;
    }

    /* function to kick start the SCTIME for differential clock output */
    void StartDifferentialClock(void) {

        SCTIMER_StartTimer(SCT0, kSCTIMER_Counter_U);
    }



    while (1)
    {

        uint8_t cmd = Read3BitCommandFromGPIO();
            if (cmd != lastCommand)
            {
                UpdateDifferentialClock(cmd);
                StartDifferentialClock();
                lastCommand = cmd;
            }

    }
}
```

B.4. Enhanced DMA

This unfinished code was developed to perform a single memory to GPIO pin transfer. The code would serve as an initial step for programming the scatter-gather DMA 2.1 on the FRDM-MCXN947 development board. Unfortunately, the code is non-functional because the program remains stuck in the first while loop (line 201), continuously waiting for the transfer done flag.
The code starts by defining a few macros. Lines 17-19 define some macros for some values. The macros on lines 21-75 are used for direct register access to configure the security and privilege modes of the DMA. A detailed explanation about these modes can be found in section 23.1.4 of [6]. Lines 89 and 91 define lists in some memory space. Afterward, on lines 96-100, the TCD pointer is initialized. The function ”EDMA_Callback” acts as a callback for the EDMA transfer; it sets a flag when a transfer is completed. The main function is composed of several tasks. First, the DMA is configured in secure and privileged mode on lines 136-143. Second, GPIO2 pin 0 is configured as an output with a default output logic set to low on lines 145-150. Lastly, the EDMA is initialized and configured for a DMA transfer from memory to GPIO2 PTOR register on lines 157-197. These steps originate from some example projects from the NXP Software Development Kit (SDK). It is recommended to read ”DMA: Direct Memory Access Controller Driver” in [23] to understand the functions and procedures.

```c
/*
 * Authors: Kevin Pang, Wilson Rong
 * Date: June 2025
 */

#include "board.h"
#include "app.h"
#include "fsl_debug_console.h"
#include "fsl_edma.h"
#include "pin_mux.h"
#include "fsl_inputmux_connections.h"
#include <stdlib.h>

/*******************************************************************************
 * Definitions
 ******************************************************************************/
#define BUFF_LENGTH      8U
#define HALF_BUFF_LENGTH (BUFF_LENGTH / 2U)
#define TCD_QUEUE_SIZE   2U

#define DMA_MP_CSR_GMRC_MASK                     (0x80U)
#define DMA_MP_CSR_GMRC_SHIFT                    (7U)
/*! GMRC - Global Master ID Replication Control
 *  0b0..Master ID replication disabled for all channels
 *  0b1..Master ID replication available and controlled by each channel's CHn_SBR[
     EMI] setting
 */
#define DMA_MP_CSR_GMRC(x)                       (((uint32_t)(((uint32_t)(x)) <<
    DMA_MP_CSR_GMRC_SHIFT)) & DMA_MP_CSR_GMRC_MASK)


#define AHBSC_MASTER_SEC_LEVEL_EDMA0_MASK        (0xC0U)
#define AHBSC_MASTER_SEC_LEVEL_EDMA0_SHIFT       (6U)
/*! eDMA0 - eDMA0
 *  0b00..Non-secure and non-privileged Master
 *  0b01..Non-secure and privileged Master
 *  0b10..Secure and non-privileged Master
 *  0b11..Secure and privileged Master
 */
#define AHBSC_MASTER_SEC_LEVEL_EDMA0(x)          (((uint32_t)(((uint32_t)(x)) <<
    AHBSC_MASTER_SEC_LEVEL_EDMA0_SHIFT)) & AHBSC_MASTER_SEC_LEVEL_EDMA0_MASK)



#define AHBSC_MASTER_SEC_ANTI_POL_REG_EDMA0_MASK (0xC0U)
#define AHBSC_MASTER_SEC_ANTI_POL_REG_EDMA0_SHIFT (6U)
/*! eDMA0 - eDMA0
 *  0b00..Secure and privileged Master
 *  0b01..Secure and non-privileged Master
 *  0b10..Non-secure and privileged Master
 *  0b11..Non-secure and non-privileged Master
 */
#define AHBSC_MASTER_SEC_ANTI_POL_REG_EDMA0(x)   (((uint32_t)(((uint32_t)(x)) <<
    AHBSC_MASTER_SEC_ANTI_POL_REG_EDMA0_SHIFT)) &
    AHBSC_MASTER_SEC_ANTI_POL_REG_EDMA0_MASK)

#define DMA_CH_SBR_PAL_MASK                      (0x8000U)
#define DMA_CH_SBR_PAL_SHIFT                     (15U)
/*! PAL - Privileged Access Level
 *  0b0..User protection level for DMA transfers
 *  0b1..Privileged protection level for DMA transfers
 */
#define DMA_CH_SBR_PAL(x)                        (((uint32_t)(((uint32_t)(x)) <<
    DMA_CH_SBR_PAL_SHIFT)) & DMA_CH_SBR_PAL_MASK)

#define DMA_CH_SBR_SEC_MASK                      (0x4000U)
#define DMA_CH_SBR_SEC_SHIFT                     (14U)
/*! SEC - Security Level
 *  0b0..Nonsecure protection level for DMA transfers
 *  0b1..Secure protection level for DMA transfers
 */
#define DMA_CH_SBR_SEC(x)                        (((uint32_t)(((uint32_t)(x)) <<
    DMA_CH_SBR_SEC_SHIFT)) & DMA_CH_SBR_SEC_MASK)


#define DMA_CH_SBR_EMI_MASK                      (0x10000U)
#define DMA_CH_SBR_EMI_SHIFT                     (16U)
/*! EMI - Enable Master ID Replication
 *  0b0..Master ID replication is disabled
 *  0b1..Master ID replication is enabled
 */
#define DMA_CH_SBR_EMI(x)                        (((uint32_t)(((uint32_t)(x)) <<
    DMA_CH_SBR_EMI_SHIFT)) & DMA_CH_SBR_EMI_MASK)
/*******************************************************************************
 * Prototypes
 ******************************************************************************/

/*******************************************************************************
 * Variables
 ******************************************************************************/


edma_handle_t g_EDMA_Handle;
volatile bool g_Transfer_Done = false;

/* defining memory space in secure memory element */
__attribute__((at(0x30000000))) uint32_t myData[4] = {0x01};
/* example SDK memory space definition for the source */
AT_NONCACHEABLE_SECTION_INIT(uint32_t srcAddr[1U])   = {0x00000001U};
/* example SDK memory space definition for the destination */
//AT_NONCACHEABLE_SECTION_INIT(uint32_t destAddr[BUFF_LENGTH]) = {0x00U, 0x00U, 0
    x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U};

/* Allocate TCD memory poll */
#if defined(DEMO_QUICKACCESS_SECTION_CACHEABLE) &&
    DEMO_QUICKACCESS_SECTION_CACHEABLE
AT_NONCACHEABLE_SECTION_ALIGN(edma_tcd_t tcdMemoryPoolPtr[TCD_QUEUE_SIZE], sizeof(
    edma_tcd_t));
#else
AT_QUICKACCESS_SECTION_DATA_ALIGN(edma_tcd_t tcdMemoryPoolPtr[TCD_QUEUE_SIZE],
    sizeof(edma_tcd_t));
#endif

/*******************************************************************************
 * Code
 ******************************************************************************/

/* User callback function for EDMA transfer. */
void EDMA_Callback(edma_handle_t *handle, void *param, bool transferDone, uint32_t
     tcds)
{
```

```c
    if (transferDone)
    {
        g_Transfer_Done = true;
    }
}

/* POINTERS Pointing to the address of the PTOR register of GPIO2 */
#define setadd GPIO2->PTOR
uint32_t *setaddr= &setadd;
uint32_t *P2_0_PTOR = &((GPIO_Type *)GPIO2)->PTOR;

/*!
 * @brief Main function
 */
int main(void)
{
    edma_transfer_config_t transferConfig;
    edma_config_t userConfig;
    edma_channel_config_t channelconfig;
    edma_tcd_t tcd;
    uint32_t *PTOR_P2_0 = &((GPIO_Type *)GPIO2)->PTOR; /*address of PTOR register for GPIO2 pin0*/

    BOARD_InitHardware();

    /******** steps accordingly to section 23.1.4 reference manual security considerations eDMA***/
    /****all steps are done specifically for DMA0 channel 0****/

    AHBSC_MASTER_SEC_LEVEL_EDMA0(0b01);
    AHBSC_MASTER_SEC_ANTI_POL_REG_EDMA0(0b10);

    DMA_MP_CSR_GMRC(0b1);

    DMA_CH_SBR_EMI(0b1);
    DMA_CH_SBR_PAL(0b0);
    DMA_CH_SBR_SEC(0b0);
    /************************** CONFIGURE PIN P2_0*****************************/
    gpio_pin_config_t P2_0_config = {
        .pinDirection = kGPIO_DigitalOutput,
        .outputLogic = 0U
    };

    GPIO_PinInit(BOARD_INITPINS_P2_0_GPIO, BOARD_INITPINS_P2_0_PIN, &P2_0_config);

    /**********************************************************************/

    /**********************************************************************/
    /* EDMA initializations */
    /*  initializations steps 1-3 of reference manual */
    EDMA_GetDefaultConfig(&userConfig);
    /* here internal clock enabled of the DMA module en initilizations of steps 1-3 are configured to the registers */
    EDMA_Init(EXAMPLE_DMA_BASEADDR, &userConfig);
    EDMA_InitChannel(EXAMPLE_DMA_BASEADDR, EXAMPLE_DMA_CHANNEL, &channelconfig);
    EDMA_SetChannelMux(EXAMPLE_DMA_BASEADDR, EXAMPLE_DMA_CHANNEL, kDma0RequestMuxGpio2PinEventRequest0);

    EDMA_CreateHandle(&g_EDMA_Handle, EXAMPLE_DMA_BASEADDR, EXAMPLE_DMA_CHANNEL);
    EDMA_SetCallback(&g_EDMA_Handle, EDMA_Callback, NULL);
    EDMA_ResetChannel(g_EDMA_Handle.base, g_EDMA_Handle.channel);

    /* Configure and submit transfer structure using TCD functions shown in fsl.edma.c */
    EDMA_InstallTCDMemory(&g_EDMA_Handle, tcdMemoryPoolPtr, TCD_QUEUE_SIZE);
    EDMA_PrepareTransferTCD(&g_EDMA_Handle, tcdMemoryPoolPtr, srcAddr, sizeof(uint32_t), 0, PTOR_P2_0, sizeof(uint32_t), 0, 4U, 4U, NULL);
    EDMA_InstallTCD(EXAMPLE_DMA_BASEADDR, EXAMPLE_DMA_CHANNEL, tcdMemoryPoolPtr);
    EDMA_SubmitTransferTCD(&g_EDMA_Handle, tcdMemoryPoolPtr);
    EDMA_StartTransfer(&g_EDMA_Handle);

/******************************************************************************/
    /* Configure and submit transfer structure from SDK: memory to peripheral*/

//    EDMA_InstallTCDMemory(&g_EDMA_Handle, tcdMemoryPoolPtr, TCD_QUEUE_SIZE);
//    /* Configure and submit transfer structure 1 */
//    EDMA_PrepareTransfer(&transferConfig, myData, 4, PTOR_P2_0, 4, 4, 4, kEDMA_MemoryToPeripheral);
//    EDMA_SubmitTransfer(&g_EDMA_Handle, &transferConfig);
//    EDMA_StartTransfer(&g_EDMA_Handle);

/******************************************************************************/

/******************************************************************************/
    /* Configure and submit transfer structure 2 from example sdk */

//    EDMA_PrepareTransfer(&transferConfig, &srcAddr[4], sizeof(srcAddr[0]), &destAddr[4], sizeof(destAddr[0]),
//                         sizeof(srcAddr[0]) * HALF_BUFF_LENGTH, sizeof(srcAddr[0]) * HALF_BUFF_LENGTH,
//                         kEDMA_MemoryToPeripheral);
//    EDMA_SubmitTransfer(&g_EDMA_Handle, &transferConfig);

/******************************************************************************/


    /* Trigger transfer start */
    EDMA_TriggerChannelStart(EXAMPLE_DMA_BASEADDR, EXAMPLE_DMA_CHANNEL);


    /* Wait for the first TCD finished */
    while (g_Transfer_Done != true)
    {
    }


    while (1)
    {
    }
}
```

C Extended Research CTIMER

This appendix entails the further research on the possible cause of the distorted second signal when configuring two CTIMERs. As mentioned in chapter 7, it is considered very strange that multiple CTIMERs were not able to run simultaneously. Another attempt was made in which every GPIO pin on the board was checked. Indeed, the second (orange) signal from Figure 6.4 and Figure 6.6 was not outputted at the configured pin P1_10 as shown in Configuration Tool of the MCUXpresso IDE in Figure C.2 (the red rectangle). However, this signal was wired to pin P1_16. This is a very remarkable finding, since output signals from the CTIMER instances cannot be multiplexed to pin P1_16 according to the data sheet, see Figure C.4.

A photo of the position of the probes when measuring, is shown in Figure C.5. In this figure, the red rectangles highlight the pin numbers P1_16, P1_18 and P1_10. The resulting waveforms are shown in Figure C.1.

Thus, it can be concluded that the CTIMERs are capable of being configured simultaneously. Furthermore, there is an inconsistency between the data sheet, MCUXpresso, and the FRDM-MCXN947 board.

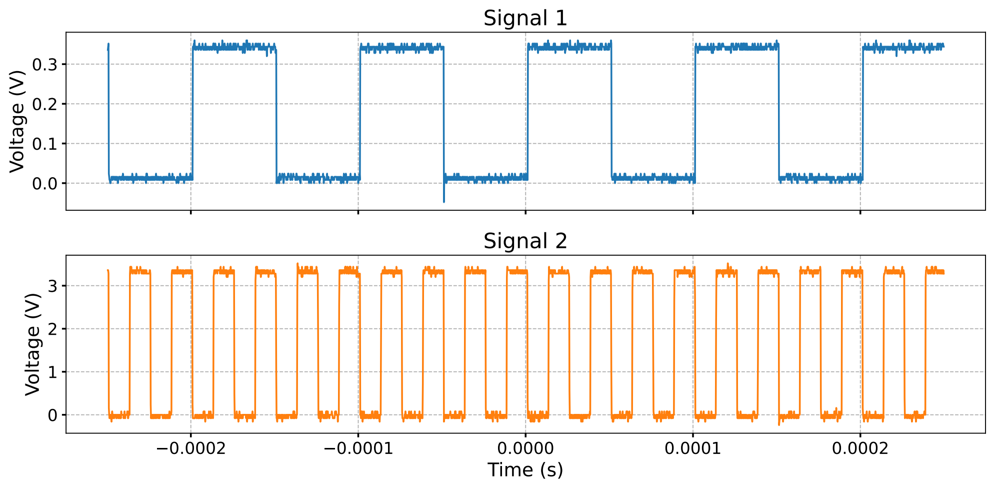
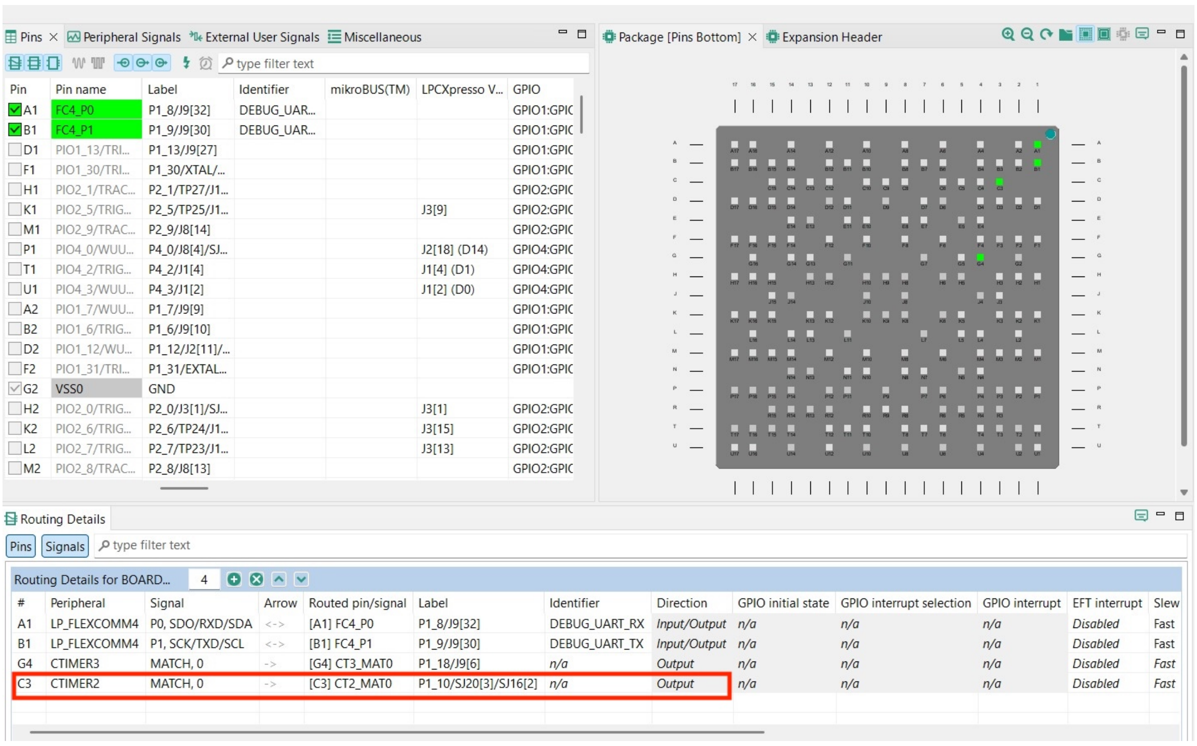


Figure C.1: P1_18 (blue) and P1_16 (orange) output

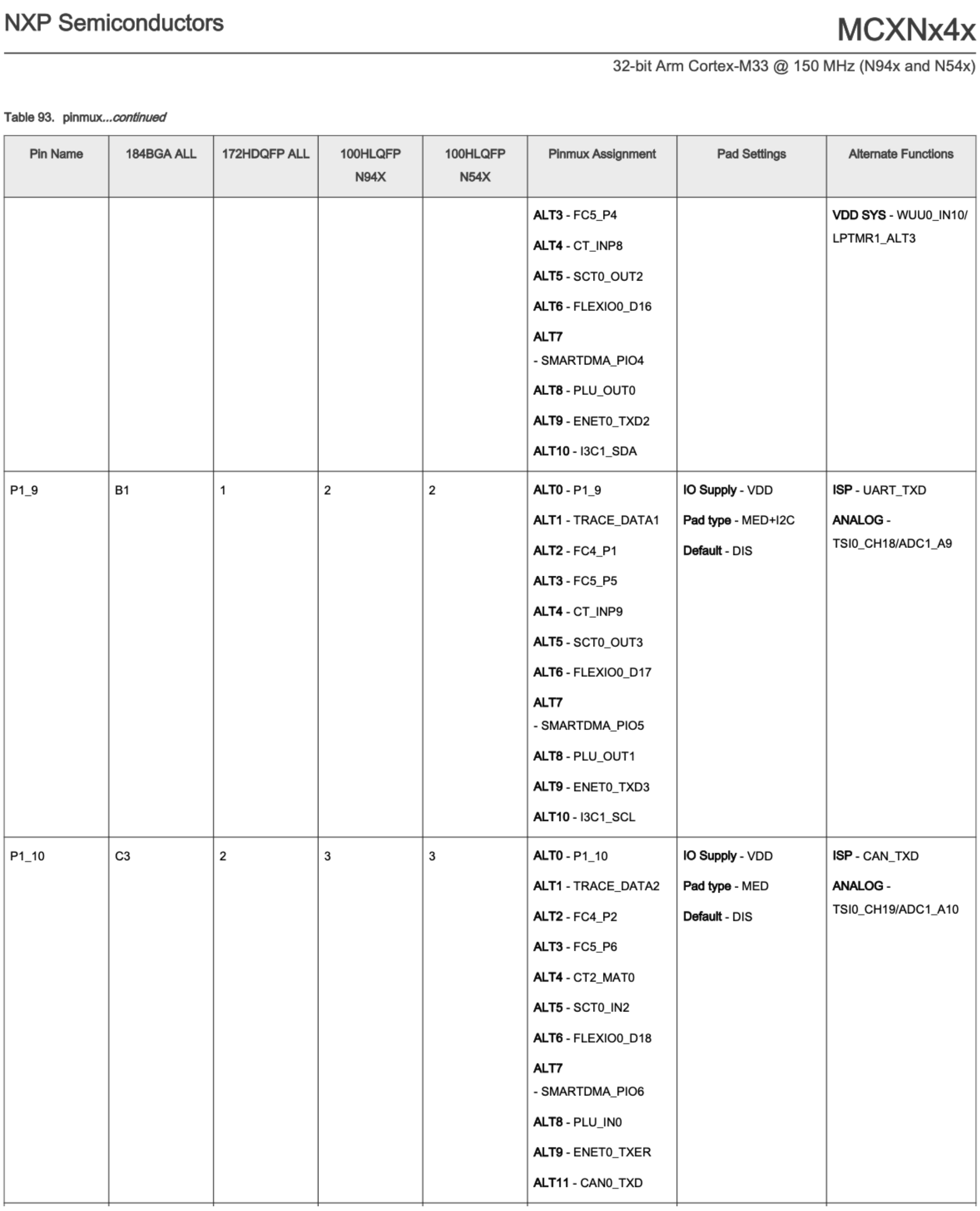


Pin	Pin name	Label	Identifier	mikroBUS(TM)	LPCXpresso V...	GPIO
✓ A1	FC4_P0	P1_8/J9[32]	DEBUG_UAR...			GPIO1:GPIC
✓ B1	FC4_P1	P1_9/J9[30]	DEBUG_UAR...			GPIO1:GPIC
D1	PIO1_13/TRI...	P1_13/J9[27]				GPIO1:GPIC
F1	PIO1_30/TRI...	P1_30/XTAL/...				GPIO1:GPIC
H1	PIO2_1/TRAC...	P2_1/TP27/J1...				GPIO2:GPIC
K1	PIO2_5/TRIG...	P2_5/TP25/J1...			J3[9]	GPIO2:GPIC
M1	PIO2_9/TRAC...	P2_9/J8[14]				GPIO2:GPIC
P1	PIO4_0/WUU...	P4_0/J8[4]/SJ...			J2[18] (D14)	GPIO4:GPIC
T1	PIO4_2/TRIG...	P4_2/J1[4]			J1[4] (D1)	GPIO4:GPIC
U1	PIO4_3/WUU...	P4_3/J1[2]			J1[2] (D0)	GPIO4:GPIC
A2	PIO1_7/WUU...	P1_7/J9[9]				GPIO1:GPIC
B2	PIO1_6/TRIG...	P1_6/J9[10]				GPIO1:GPIC
D2	PIO1_12/WU...	P1_12/J2[11]/...				GPIO1:GPIC
F2	PIO1_31/TRI...	P1_31/EXTAL...				GPIO1:GPIC
✓ G2	VSS0	GND				
H2	PIO2_0/TRIG...	P2_0/J3[1]/SJ...			J3[1]	GPIO2:GPIC
K2	PIO2_6/TRIG...	P2_6/TP24/J1...			J3[15]	GPIO2:GPIC
L2	PIO2_7/TRIG...	P2_7/TP23/J1...			J3[13]	GPIO2:GPIC
M2	PIO2_8/TRAC...	P2_8/J8[13]				GPIO2:GPIC

Routing Details for BOARD... 4

#	Peripheral	Signal	Arrow	Routed pin/signal	Label	Identifier	Direction	GPIO initial state	GPIO interrupt selection	GPIO interrupt	EFT interrupt	Slew
A1	LP_FLEXCOMM4	P0, SDO/RXD/SDA	<->	[A1] FC4_P0	P1_8/J9[32]	DEBUG_UART_RX	*Input/Output*	*n/a*	*n/a*	*n/a*	*Disabled*	Fast
B1	LP_FLEXCOMM4	P1, SCK/TXD/SCL	<->	[B1] FC4_P1	P1_9/J9[30]	DEBUG_UART_TX	*Input/Output*	*n/a*	*n/a*	*n/a*	*Disabled*	Fast
G4	CTIMER3	MATCH, 0	->	[G4] CT3_MAT0	P1_18/J9[6]	*n/a*	*Output*	*n/a*	*n/a*	*n/a*	*Disabled*	*Fast*
C3	CTIMER2	MATCH, 0	->	[C3] CT2_MAT0	P1_10/SJ20[3]/SJ16[2]	*n/a*	*Output*	*n/a*	*n/a*	*n/a*	*Disabled*	*Fast*

Figure C.2: Pin configuration tools in MCUXpresso IDE



NXP Semiconductors

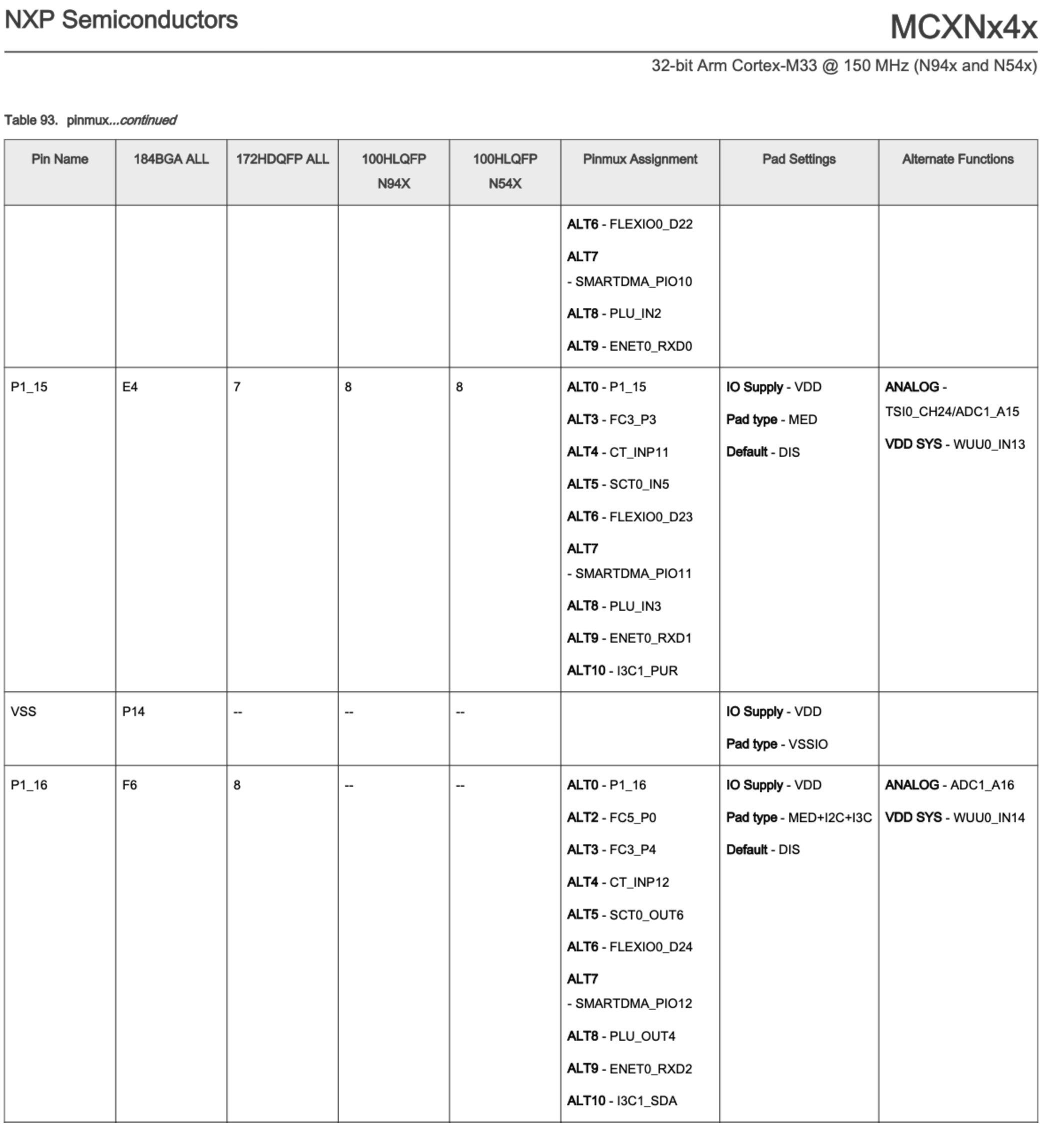
MCXNx4x

32-bit Arm Cortex-M33 @ 150 MHz (N94x and N54x)

Table 93. pinmux...*continued*

Pin Name	184BGA ALL	172HDQFP ALL	100HLQFP N94X	100HLQFP N54X	Pinmux Assignment	Pad Settings	Alternate Functions
					ALT3 - FC5_P4 **ALT4** - CT_INP8 **ALT5** - SCT0_OUT2 **ALT6** - FLEXIO0_D16 **ALT7** - SMARTDMA_PIO4 **ALT8** - PLU_OUT0 **ALT9** - ENET0_TXD2 **ALT10** - I3C1_SDA		**VDD SYS** - WUU0_IN10/ LPTMR1_ALT3
P1_9	B1	1	2	2	**ALT0** - P1_9 **ALT1** - TRACE_DATA1 **ALT2** - FC4_P1 **ALT3** - FC5_P5 **ALT4** - CT_INP9 **ALT5** - SCT0_OUT3 **ALT6** - FLEXIO0_D17 **ALT7** - SMARTDMA_PIO5 **ALT8** - PLU_OUT1 **ALT9** - ENET0_TXD3 **ALT10** - I3C1_SCL	**IO Supply** - VDD **Pad type** - MED+I2C **Default** - DIS	**ISP** - UART_TXD **ANALOG** - TSI0_CH18/ADC1_A9
P1_10	C3	2	3	3	**ALT0** - P1_10 **ALT1** - TRACE_DATA2 **ALT2** - FC4_P2 **ALT3** - FC5_P6 **ALT4** - CT2_MAT0 **ALT5** - SCT0_IN2 **ALT6** - FLEXIO0_D18 **ALT7** - SMARTDMA_PIO6 **ALT8** - PLU_IN0 **ALT9** - ENET0_TXER **ALT11** - CAN0_TXD	**IO Supply** - VDD **Pad type** - MED **Default** - DIS	**ISP** - CAN_TXD **ANALOG** - TSI0_CH19/ADC1_A10

Figure C.3: Table 93 on page 100 in [7]



NXP Semiconductors

MCXNx4x

32-bit Arm Cortex-M33 @ 150 MHz (N94x and N54x)

Table 93. pinmux...*continued*

Pin Name	184BGA ALL	172HDQFP ALL	100HLQFP N94X	100HLQFP N54X	Pinmux Assignment	Pad Settings	Alternate Functions
					ALT6 - FLEXIO0_D22 **ALT7** - SMARTDMA_PIO10 **ALT8** - PLU_IN2 **ALT9** - ENET0_RXD0		
P1_15	E4	7	8	8	**ALT0** - P1_15 **ALT3** - FC3_P3 **ALT4** - CT_INP11 **ALT5** - SCT0_IN5 **ALT6** - FLEXIO0_D23 **ALT7** - SMARTDMA_PIO11 **ALT8** - PLU_IN3 **ALT9** - ENET0_RXD1 **ALT10** - I3C1_PUR	**IO Supply** - VDD **Pad type** - MED **Default** - DIS	**ANALOG** - TSI0_CH24/ADC1_A15 **VDD SYS** - WUU0_IN13
VSS	P14	--	--	--		**IO Supply** - VDD **Pad type** - VSSIO	
P1_16	F6	8	--	--	**ALT0** - P1_16 **ALT2** - FC5_P0 **ALT3** - FC3_P4 **ALT4** - CT_INP12 **ALT5** - SCT0_OUT6 **ALT6** - FLEXIO0_D24 **ALT7** - SMARTDMA_PIO12 **ALT8** - PLU_OUT4 **ALT9** - ENET0_RXD2 **ALT10** - I3C1_SDA	**IO Supply** - VDD **Pad type** - MED+I2C+I3C **Default** - DIS	**ANALOG** - ADC1_A16 **VDD SYS** - WUU0_IN14

Figure C.4: Table 93 on page 102 in [7]

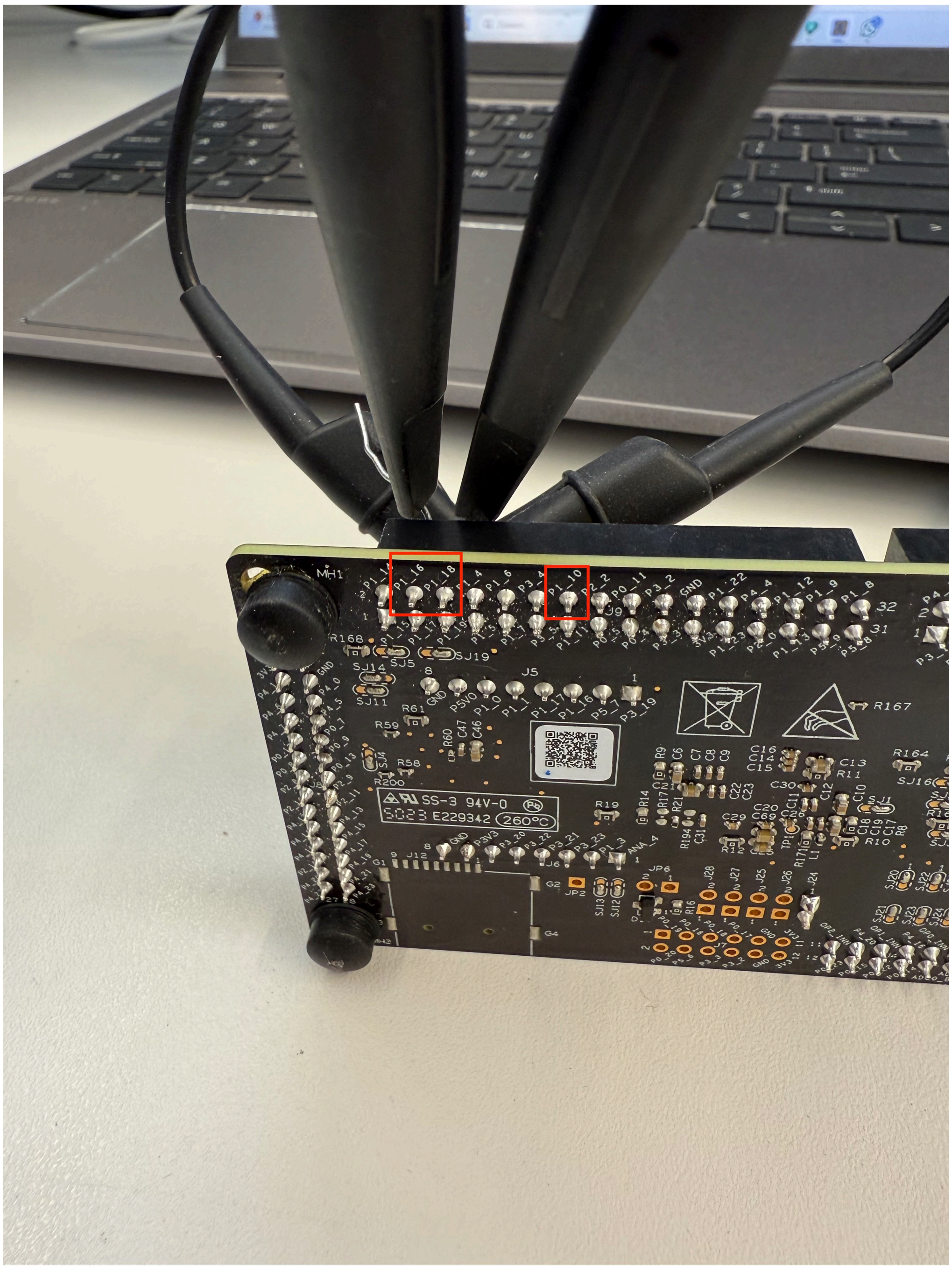


Figure C.5: Probe attachment on pins P1_16 and P1_18

Bibliography

[1] Y. Machtane and P. Olyslaegers, “Mixed signal pcb design and analog readout circuitry for a pixelated capacitive sensor e-nose array,” Bachelor’s thesis, Delft University of Technology, Jun. 2025.

[2] C. R. Chen and F. Lin, “Design and integration of a socket, chamber, and automated control system for an e-nose based on cmos pixelated capacitive sensor array,” Bachelor’s thesis, Delft University of Technology, Jun. 2025.

[3] F. Röck, N. Barsan, and U. Weimar, “Electronic nose: Current status and future trends,” *Chemical reviews*, vol. 108, pp. 705–25, Mar. 2008. DOI: `10.1021/cr068121q`.

[4] F. Widdershoven, “Pixelated capacitive sensors (pcs) for embedded multi-sensing,” in *Imaging Sensors, Power Management, PLLs and Frequency Synthesizers: Advances in Analog Circuit Design 2023*, K. A. A. Makinwa, A. Baschirotto, and B. Nauta, Eds. Cham: Springer Nature Switzerland, 2025, pp. 23–35, ISBN: 978-3-031-71559-4. DOI: `10.1007/978-3-031-71559-4_2`. [Online]. Available: `https://doi.org/10.1007/978-3-031-71559-4_2`.

[5] NXP Semiconductors, *Lpc1769/68/67/66/65/64/63 product data sheet*, Rev. 9.10, 32-bit ARM Cortex-M3 microcontroller; up to 512 kB flash and 64 kB SRAM with Ethernet, USB 2.0 Host/Device/OTG, CAN, Sep. 2020. [Online]. Available: `https://www.nxp.com/docs/en/data-sheet/LPC1769_68_67_66_65_64_63.pdf`.

[6] NXP Semiconductors, *Mcx nx4x reference manual*, Document number: MCXNX4RM, NXP Semiconductors, 2024. [Online]. Available: `https://www.nxp.com/webapp/sps/download/preDownload.jsp?render=true`.

[7] NXP Semiconductors, *Mcx nx4x microcontrollers: 32-bit arm cortex-m33 @ 150 mhz (n94x and n54x)*, Rev. 7, Product Data Sheet, NXP Semiconductors, Jan. 2025. [Online]. Available: `https://www.nxp.com/docs/en/data-sheet/MCXNx4xDS.pdf`.

[8] NXP Semiconductors, *Frdm-mcxn947 board user manual*, Rev. 2.0, Document number: UM12018, NXP Semiconductors, Aug. 2024. [Online]. Available: `https://www.nxp.com/webapp/sps/download/preDownload.jsp?render=true`.

[9] D. Yadav, *Microcontroller: features and applications*. New Age International, 2004.

[10] K. R. Raghunathan, “History of microcontrollers: First 50 years,” *IEEE Micro*, vol. 41, no. 6, pp. 97–104, 2021. DOI: `10.1109/MM.2021.3114754`.

[11] I. L. Markov, “Limits on fundamental limits to computation,” *Nature*, vol. 512, no. 7513, pp. 147–154, 2014.

[12] J. Nider and A. Fedorova, “The last cpu,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 1–8.

[13] C. Arun, A. Gopinath, A. Hanumanthaiah, and R. Murugan, “Implementation of direct memory access for parallel processing,” in *2020 4th International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, 2020, pp. 390–394. DOI: `10.1109/ICECA49313.2020.9297651`.

[14] A. Ahmed, A. Aljumah, and M. Ahmad, “Design and implementation of a direct memory access controller for embedded applications,” *International Journal of Technology*, vol. 10, p. 309, Apr. 2019. DOI: `10.14716/ijtech.v10i2.795`.

[15] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, “Decoupled direct memory access: Isolating cpu and io traffic by leveraging a dual-data-port dram,” in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 174–187. DOI: `10.1109/PACT.2015.51`.

[16] D. Tang, Y. Bao, W. Hu, and M. Chen, “Dma cache: Using on-chip storage to architecturally separate i/o data from cpu data for improving i/o performance,” in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12. DOI: `10.1109/HPCA.2010.5416638`.

[17] T. Murayama, H. Yamada, T. Nakamura, Y. Shigei, and Y. Yoshioka, “Performance evaluation of a computer using programmable logic units,” *Systems and Computers in Japan*, vol. 18, no. 8, pp. 57–66, 1987. DOI: `https://doi.org/10.1002/scj.4690180806`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/scj.4690180806`. [Online]. Available: `https://onlinelibrary.wiley.com/doi/abs/10.1002/scj.4690180806`.

[18] B. Venu, “Multi-core processors - an overview,” *CoRR*, vol. abs/1110.3535, 2011. arXiv: `1110.3535`. [Online]. Available: `http://arxiv.org/abs/1110.3535`.

[19] P. Gepner and M. Kowalik, “Multi-core processors: New way to achieve high system performance,” in *International Symposium on Parallel Computing in Electrical Engineering (PARELEC’06)*, 2006, pp. 9–13. DOI: `10.1109/PARELEC.2006.54`.

[20] NXP Semiconductors, *FRDM-MCXN947 Development Board*, Accessed: June 5, 2025, 2024. [Online]. Available: `https://www.nxp.com/design/design-center/development-boards-and-designs/FRDM-MCXN947`.

[21] NXP Semiconductors, *AN-MCXN-PLU-SetupAndUsage*, `https://github.com/nxp-appcodehub/an-mcxn-plu-setupandusage`, Accessed: June 5, 2025, 2024.

[22] Arm Keil, *Frdm-mcxn947 board projects - dma*, Accessed: 2025-06-04, 2025. [Online]. Available: `https://www.keil.arm.com/boards/nxp-frdm-mcxn947-634a158/projects/?q=DMA`.

[23] NXP Semiconductors, *Mcuxpresso sdk api reference manual*, Revision 0, NXP Semiconductors, 2025. [Online]. Available: `https://mcuxpresso.nxp.com/api_doc/dev/96/modules.html`.

[24] Analog Devices, Inc., *AD9552: Oscillator Frequency Upconverter*, Rev. E, 2012. [Online]. Available: `https://www.analog.com/en/products/ad9552.html`.

[25] Tektronix, Inc., *TDS1000B Series • TDS2000B Series: Digital Storage Oscilloscopes*, 2006. [Online]. Available: `https://www.alldatasheet.com/datasheet-pdf/view/512404/ETC1/TDS2022B.html`.