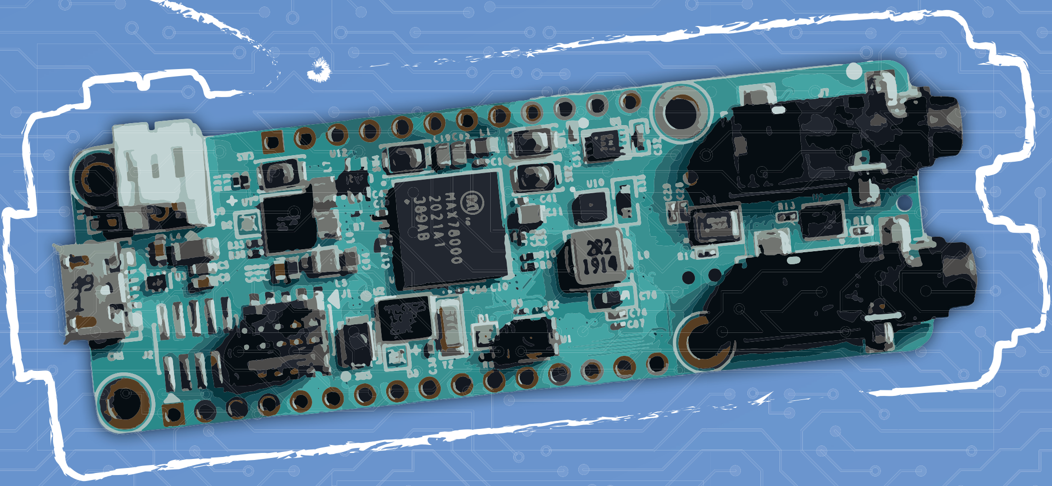
H0W2 Volume 2

# Get Started with the MAX78000FTHR Development Board

Build your own AI microcontroller applications from scratch



Dogan Ibrahim

maxim integrated

elektor knows how

# How2: Get Started with the MAX78000FTHR Development Board

**Dogan Ibrahim**

elektor
design > share > sell

an Elektor Publication

● This is an Elektor Publication. Elektor is the media brand of
Elektor International Media B.V.
78 York Street
London W1H 1DP, UK
Phone: (+44) (0)20 7692 8344
© Elektor International Media BV 2021
First published in the United Kingdom 2021

● All rights reserved. No part of this book may be reproduced in any material form, including photocopying, or storing in any medium by electronic means and whether or not transiently or incidentally to some other use of this publication, without the written permission of the copyright holder except in accordance with the provisions of the Copyright, Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London, England W1P 9HE. Applications for the copyright holder's written permission to reproduce any part of this publication should be addressed to the publishers. The publishers have used their best efforts in ensuring the correctness of the information contained in this book. They do not assume, and hereby disclaim, any liability to any party for any loss or damage caused by errors or omissions in this book, whether such errors or omissions result from negligence, accident or any other cause.

● British Library Cataloguing in Publication Data
Catalogue record for this book is available from the British Library

● ISBN: 978-3-89576-447-9

● EISBN: 978-3-89576-448-6

Prepress production: DMC ¦ dave@daverid.com
Printed in the Netherlands by Ipskamp

elektor
design > share > sell

Elektor is part of EIM, the world's leading source of essential technical information and electronics products for pro engineers, electronics designers, and the companies seeking to engage them. Each day, our international team develops and delivers high-quality content - via a variety of media channels (e.g., magazines, video, digital media, and social media) in several languages - relating to electronics design and DIY electronics. **www.elektor.com**

*To my wife Nadire, my daughter Alev, and my son Ahmet, for their love and wisdom.*

## • Declaration

The author and publisher have used their best efforts in ensuring the correctness of the information contained in this book. They do not assume, or hereby disclaim, any liability to any party for any loss or damage caused by errors or omissions in this book, whether such errors or omissions result from negligence, accident, or any other cause.

The author expresses his sincere thanks to **Maxim Integrated** for permitting to include various tables, figures, and program codes in this book.

Additionally, the author would like to thank Mr. Ole Dreessen (Principal MTS, Field Applications of Maxim Integrated, Munich) and his team for reviewing the manuscript.

# ● Preface

A microcontroller is a single-chip microprocessor system that contains data and program memory, serial and parallel I/O, timers, external and internal interrupts, all integrated into a single chip that can be purchased for as little as $2.00. About 40% of microcontroller applications are in office automation, such as PCs, laser printers, fax machines, intelligent telephones, and so forth. About one-third of microcontrollers are found in consumer electronic goods. Products like CD and DVD players, hi-fi equipment, video games, washing machines, cookers, and so on fall into this category. The communications, automotive, and military markets share the rest of the application areas.

The MAX78000FTHR (from Maxim Integrated) is a small development board based on a MAX78000 microcontroller unit (MCU). This MCU is targeted at artificial intelligence (AI) applications running at the edge of the technology. AI applications require large amounts of processing power and memory. This is why the MAX78000 combines an Arm Cortex-M4 processor with a floating-point unit (FPU), convolutional neural network (CNN) accelerator, and a RISC-V core into a single device. The MAX78000 microcontroller also has the important feature that it is designed for ultra-low power consumption, thus making it ideal in many AI-based portable applications.

The MAX78000 microcontroller uses a Convolutional Neural Network (CNN) based approach. A CNN is a special kind of neural network where convolution is used at its heart. Convolution is a kind of matched filtering and is often used in signal and image processing. Since CNNs are particularly used in these applications, the MAX78000FTHR development board integrates an onboard camera and microphone interfaced with I2S for streaming audio applications.

The board is recognised as a USB device when connected to a computer. It comes preloaded with an audio keyword spotting (KWS) demo, where the flashing rate of an onboard LED can be changed by speech commands. When it detects the word "Go" the demo enters number recognition mode in which it will blink an LED the number of times as commanded by the speaker. In other words, after saying "Four", the LED will blink four times. "Stop" returns to normal mode.

The MAX78000FTHR development board includes the following peripheral devices on-board:

- Tiny VGA camera
- Digital microphone
- microSD card slot
- 1 MB RAM
- SWD debugger/programmer over USB
- LiPo battery charger
- RGB LEDs and user pushbuttons for projects

An Eclipse-based SDK is provided by Maxim that can be used to develop programs for the MAX78000FTHR. The development environment contains the Eclipse IDE, MinGW, GCC toolchains for Arm and RISC-V processors, OpenOCD, and a few other utilities. Additionally, libraries and many example programs are provided to help the starters develop projects.

This book is project-based and its main aim has been to teach the basic features of the MAX78000FTHR development board and show how it can be used in various projects. Many fully tested projects are given in the book, where each project is described fully and in detail, and the complete program listings are given for each project. Readers should be able to use the projects as they are, or modify them to suit their own needs. The following sub-headings are used while describing each project:

- Description of the project
- Aim of the project
- Background (if applicable)
- Block diagram
- Circuit diagram
- Program listing
- Suggestions for future work (if applicable)

Knowledge of C will be useful to readers. Also, familiarity with at least one microcontroller development board (preferably with an Arm processor) will be an advantage. Knowledge of assembly language programming is not required because all projects in the book are based on using C. Some knowledge of the theory of neural networks will make it easier for the readers to follow the chapter on Convolutional Neural Networks.

This book is written for students, practising engineers, and for hobbyists interested in developing artificial interface applications using the MAX78000FTHR development board. An attempt has been made to include as many projects as possible, limited only by the size of the book.

Dogan Ibrahim
London, 2021

# ● Table of Contents

# Chapter 1 ● The MAX78000FTHR Development Board

## 1.1 ● Overview

The MAX78000FTHR is an advanced ultra-low-power microcontroller development board, created to help implement artificial intelligence (AI) solutions. Based on the Arm Cortex-M4F processor, the MAX78000 includes an integrated Convolution Neural Network (CNN) accelerator. In this chapter, we will be looking at the basic hardware details of this board.

## 1.2 ● Basic features

The MAX78000FTHR development board (see Figure 1.1 for top view, and Figure 1.2 for bottom view) has the following basic features:

- Arm Cortex-M4 100MHz processor with FPU and RISC-V core
- 32-bit RISC-V coprocessor
- 512KB Flash memory
- 128KB SRAM
- 16KB Cache
- Convolution Neural Network Accelerator (CNN)
- 12-bit camera interface (parallel)
- Wearable PMIC (MAX20303)
- On-board DAPLink Debug and programming
- Micro-USB connector
- Micro SD card holder
- SPI, I2C, LPUART, I2S interfaces
- LPTIMER and analog comparators
- Breadboard compatible pins (dual-row header)
- Digital microphone (Knowledge Acoustics SPH0645LM4H-B)
- 3 x RGB indicator LEDs
- 5 x pushbuttons
- CMOS VGA image sensor (Omnivision OVM7692-RYAA)
- Stereo audio CODEC (Maxim MAX9867)
- Virtual UART console
- 10-pin Cortex Debug Header for RISC-V Coprocessor
- Li-Ion battery charger (battery not included)

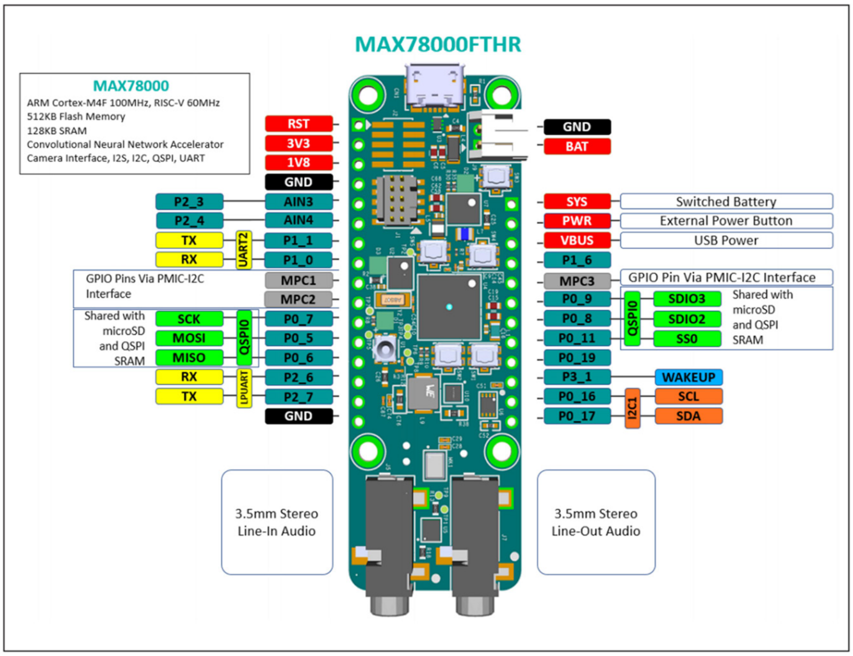
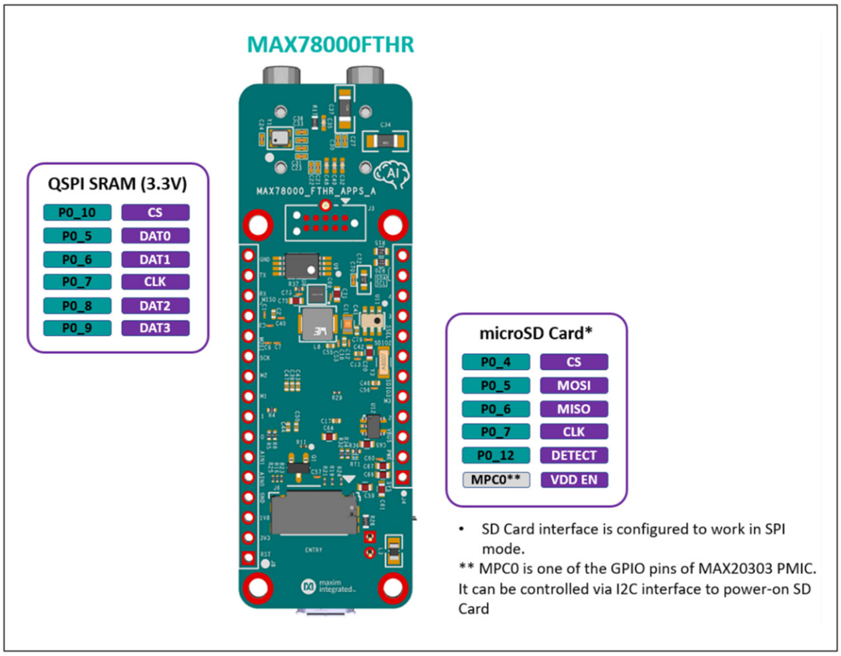

MAX78000FTHR
MAX78000
ARM Cortex-M4F 100MHz, RISC-V 60MHz
512KB Flash Memory
128KB SRAM
Convolutional Neural Network Accelerator
Camera Interface, I2S, I2C, QSPI, UART
RST
3V3
1V8
GND
P2_3 AIN3
P2_4 AIN4
TX P1_1
RX P1_0
UART2
GPIO Pins Via PMIC-I2C Interface
MPC1
MPC2
Shared with microSD and QSPI SRAM
SCK P0_7
MOSI P0_5
MISO P0_6
QSPI0
RX P2_6
TX P2_7
LPUART
GND
GND
BAT
SYS Switched Battery
PWR External Power Button
VBUS USB Power
P1_6
MPC3 GPIO Pin Via PMIC-I2C Interface
P0_9 SDIO3
P0_8 SDIO2
P0_11 SS0
QSPI0
Shared with microSD and QSPI SRAM
P0_19
P3_1 WAKEUP
P0_16 SCL
P0_17 SDA
I2C1
3.5mm Stereo Line-In Audio
3.5mm Stereo Line-Out Audio

Figure 1.1 Top view of the MAX78000FTHR board



- SD Card interface is configured to work in SPI mode.

** MPC0 is one of the GPIO pins of MAX20303 PMIC. It can be controlled via I2C interface to power-on SD Card

Figure 1.2 Bottom view of the MAX78000FTHR board

## 1.3 ● Pushbuttons and LEDs

There are 5 pushbuttons (named **SW1** to **SW5**) and 3 RGB LEDs (named **D1** to **D3**) on the board as shown in Figure 1.3.

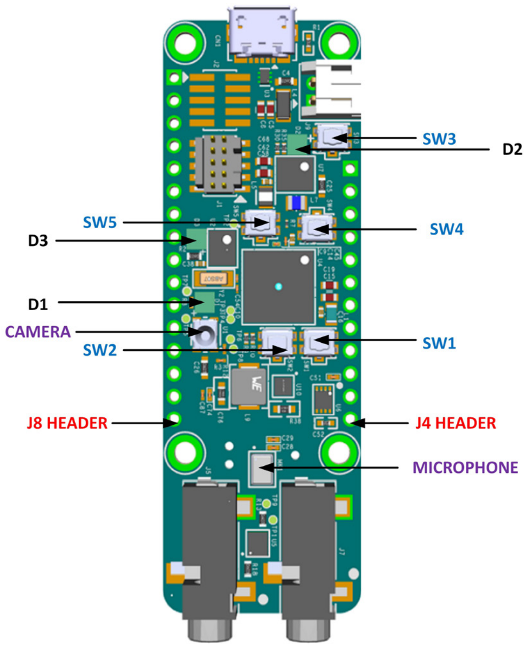


Figure 1.3 Pushbuttons and LEDs on the board

The functions of the pushbuttons are:

**SW1**: User-programmable function button connected to the MAX78000 Port 0_2 through a debouncer IC.

**SW2**: User-programmable function button connected to the MAX78000 Port 1_7 through a debouncer IC.

**SW3**: PMIC Power Button. When the board is in a powered-on state, pressing this button for 12 seconds performs a hard power-down. When the board is in a powered-off state, pressing this button powers on the board. This button can also be read by the MAX78000 firmware, PMIC_PFN2 signal connected to Port 3_1 is a buffered input of the button status. When the button is pressed, this signal goes to a logic-low state.

**SW4**: Resets the MAX78000 through the RSTN input of the MAX78000.

**SW5**: DAPLink adapter button. Keep this button pressed while applying power to the board to put the MAX32625 DAPLink adapter onboard to MAINTENANCE mode for DAPLink firmware updates.

The functions of the LEDs are:

**D1**: Connected to the MAX78000 GPIO ports. This LED can be controlled by user firmware.

**Port 2_0 : Red**
**Port 2_1 : Green**
**Port 2_2 : Blue**

**D2**: Connected to MAX20303 PMIC LEDx outputs. These LEDs can be controlled through I2C commands. They also can be configured as charge status indicators by issuing I2C commands.

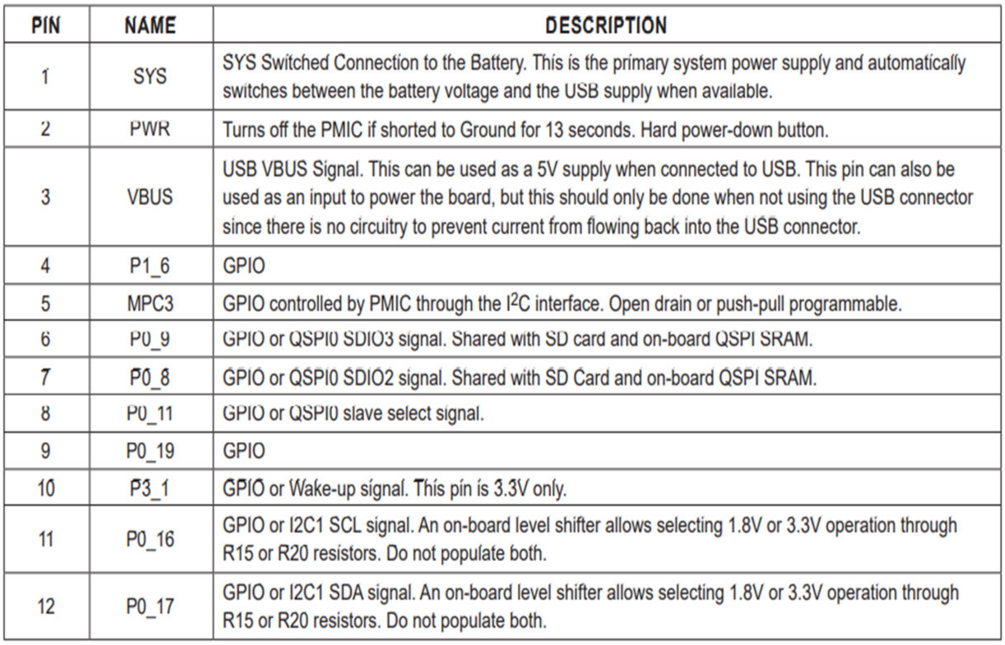
**D3**: DAPLink adapter MAX32625 status LED. Controlled by the DAPLink adapter and cannot be used as a user LED.

## 1.4 • GPIO Pinout

There are two headers at either side of the board where GPIO signals are terminated. As shown in Figure 1.3, **Header J4** has 12 pins and **Header J8** has 16 pins. The board includes the following GPIO port pins:

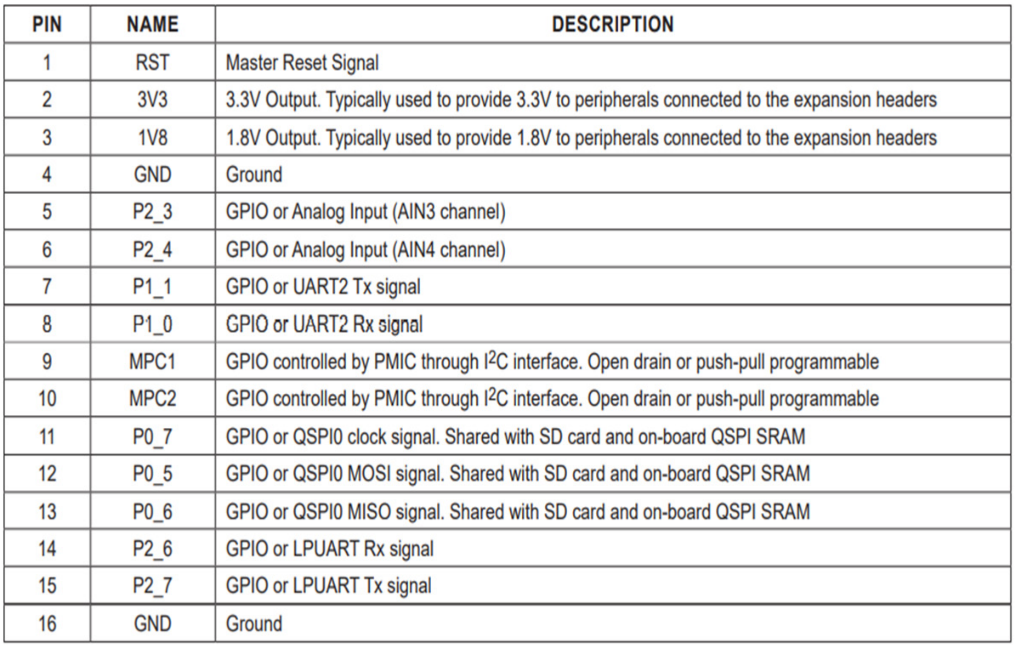
**PORT0**: P0_5, P0_6, P0_7, P0_8, P0_9, P0_11, P0_16, P0_17, P0_19
**PORT1**: P1_0, P1_1
**PORT2**: P2_3, P2_4, P2_6, P2_7
**PORT3**: P3_1

Tables 1.1 and 1.2 show the J4 and J8 pin names and their descriptions. Notice some pins are shared. For example, P1_0 and p1_1 are shared with the UART RX and TX pins respectively. Similarly, P2_3 and P2_4 are shared with analog inputs AIN3 and AIN4 respectively.



| PIN | NAME | DESCRIPTION |
|---|---|---|
| 1 | SYS | SYS Switched Connection to the Battery. This is the primary system power supply and automatically switches between the battery voltage and the USB supply when available. |
| 2 | PWR | Turns off the PMIC if shorted to Ground for 13 seconds. Hard power-down button. |
| 3 | VBUS | USB VBUS Signal. This can be used as a 5V supply when connected to USB. This pin can also be used as an input to power the board, but this should only be done when not using the USB connector since there is no circuitry to prevent current from flowing back into the USB connector. |
| 4 | P1_6 | GPIO |
| 5 | MPC3 | GPIO controlled by PMIC through the $I^2C$ interface. Open drain or push-pull programmable. |
| 6 | P0_9 | GPIO or QSPI0 SDIO3 signal. Shared with SD card and on-board QSPI SRAM. |
| 7 | P0_8 | GPIO or QSPI0 SDIO2 signal. Shared with SD Card and on-board QSPI SRAM. |
| 8 | P0_11 | GPIO or QSPI0 slave select signal. |
| 9 | P0_19 | GPIO |
| 10 | P3_1 | GPIO or Wake-up signal. This pin is 3.3V only. |
| 11 | P0_16 | GPIO or I2C1 SCL signal. An on-board level shifter allows selecting 1.8V or 3.3V operation through R15 or R20 resistors. Do not populate both. |
| 12 | P0_17 | GPIO or I2C1 SDA signal. An on-board level shifter allows selecting 1.8V or 3.3V operation through R15 or R20 resistors. Do not populate both. |

Table 1.1 Header J4 pins



| PIN | NAME | DESCRIPTION |
|---|---|---|
| 1 | RST | Master Reset Signal |
| 2 | 3V3 | 3.3V Output. Typically used to provide 3.3V to peripherals connected to the expansion headers |
| 3 | 1V8 | 1.8V Output. Typically used to provide 1.8V to peripherals connected to the expansion headers |
| 4 | GND | Ground |
| 5 | P2_3 | GPIO or Analog Input (AIN3 channel) |
| 6 | P2_4 | GPIO or Analog Input (AIN4 channel) |
| 7 | P1_1 | GPIO or UART2 Tx signal |
| 8 | P1_0 | GPIO or UART2 Rx signal |
| 9 | MPC1 | GPIO controlled by PMIC through $I^2C$ interface. Open drain or push-pull programmable |
| 10 | MPC2 | GPIO controlled by PMIC through $I^2C$ interface. Open drain or push-pull programmable |
| 11 | P0_7 | GPIO or QSPI0 clock signal. Shared with SD card and on-board QSPI SRAM |
| 12 | P0_5 | GPIO or QSPI0 MOSI signal. Shared with SD card and on-board QSPI SRAM |
| 13 | P0_6 | GPIO or QSPI0 MISO signal. Shared with SD card and on-board QSPI SRAM |
| 14 | P2_6 | GPIO or LPUART Rx signal |
| 15 | P2_7 | GPIO or LPUART Tx signal |
| 16 | GND | Ground |

Table 1.2 Header J8 pins

## 1.5 ● The FTHR board component interface signals

Figure 1.4 shows the MAX78000FTHR Application Platform. The figure shows the application interface signals, buttons, and LEDs for:

- RISC-V JTAG
- Camera module
- PMIC
- Digital microphone
- Audio CODEC
- UART
- RGB LED
- Buttons

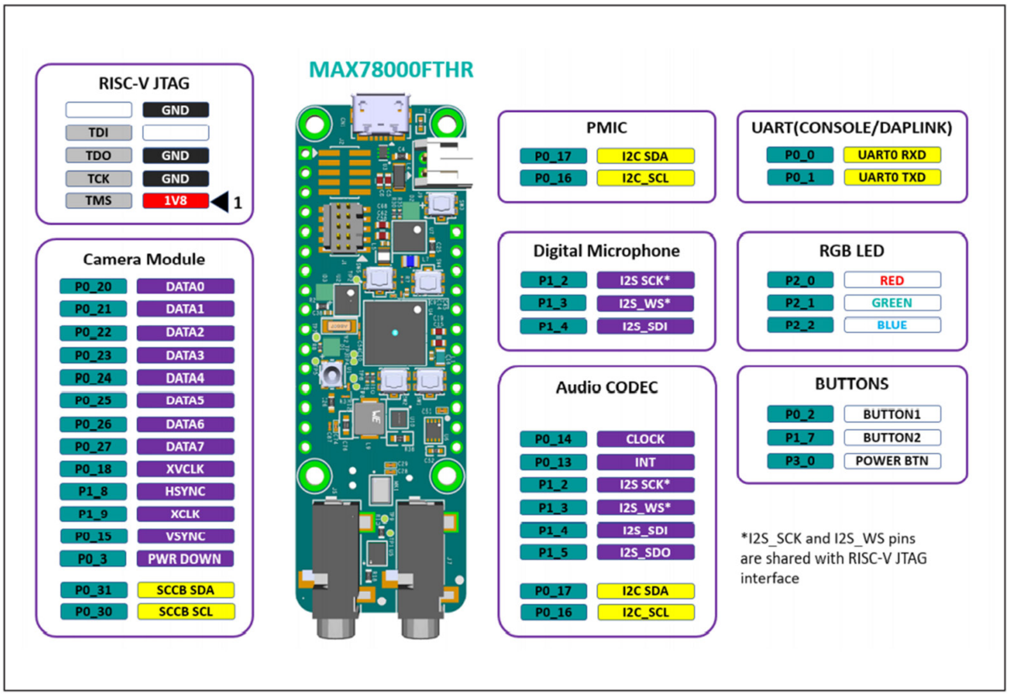


Figure 1.4 The application platform

## 1.6 ● The startup and the demo application

The MAX78000FTHR is pre-programmed with the **Audio Keyword Spotting** demo application. This demo application is useful for checking if the board is working correctly.

The steps to startup the MAX78000FTHR and the demo application are:

- Connect the MAX78000FTHR to the USB port of your PC using a micro-USB cable
- RGB LED (**D2**) will turn ON green to indicate that the pre-programmed demo application **Audio Keyword Spotting** is running
- The on-board microphone (see Figure 1.3) starts listening for the keyword GO
- When the keyword **GO** is detected, RGB LED (**D2**) turns yellow
- In this mode, when one of nine keywords is detected, the RGB LED (**D1**) blinks blue one to nine times based on the number detected by the convolutional neural network. For example, speak the word **FOUR** and  the blue LED blinks 4 times
- The STOP command exits number keyword detection, and the RGB LED (**D2**) turns on green again, and RGB LED (**D1**) turns off

## 1.7 ● The voltage regulator/battery charger

The MAX20303 PMIC chip is used to power the MAX78000FTHR board and also to charge a Li-Ion battery (not included). The MAX20303 has an internal MOSFET that connects the battery to system output. The smart power selector unit inside the PMIC seamlessly controls power to the board when a battery is present and when the board is connected to a USB power source (for more information, see document: *MAX78000FTHR Application Platform, Rev. 11/20, Maxim Integrated Products Inc.* at link: https://datasheets.maximintegrated.com/en/ds/MAX78000FTHR.pdf).

## 1.8 ● DAP-link/SWD debug interface

The onboard MAX32625 microcontroller is pre-programmed with DAPLink firmware. It allows the debugging and programming of the MAX78000 Arm core through a USB interface. A standard 10-pin DAP-link/SWD interface (header J2) is provided as shown in Figure 1.5 with the pin configuration shown in Figure 1.6. DAPLink adapter button SW5 must be kept pressed while applying power to the board to put the MAX32625 DAPLink adapter onboard to MAINTENANCE mode for DAPLink firmware updates (see link: https://os.mbed.com/teams/MaximIntegrated/wiki/MAX32625PICO-Firmware-Updates). The DAPLink adapter status LED D3 is controlled by the DAPLink adapter. The board comes pre-installed with a bootloader enabling driverless drag-n-drop updates. This bootloader can be used to update or restore the DAPLink firmware on the MAX32625, or to load your custom application on the board. To activate the bootloader, simply hold the SW5 button down while applying power to the board. When the bootloader detects the button press at power on, it will connect to the PC as a drive named "MAINTENANCE". Simply Drag-n-Drop the desired image onto the MAINTENANCE drive to load new firmware into the board.

We can use a virtual com port and send data from the development board to a terminal emulation program (e.g. Putty) using e.g. printf statements. This can be very helpful during debugging a program.

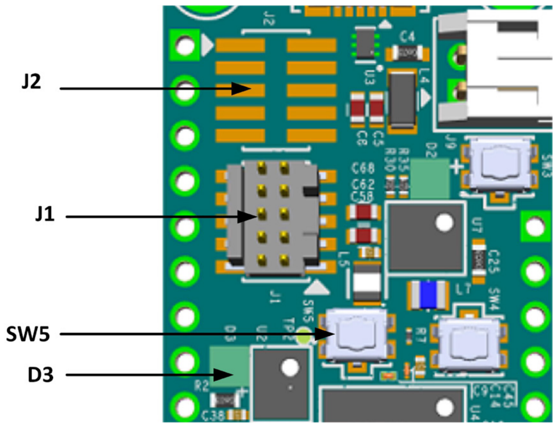


Figure 1.5 Headers J1 and J2

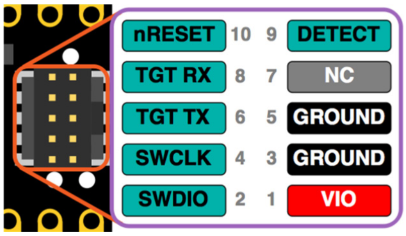


Figure 1.6 Header J2 pin configuration

A standard 10-pin JTAG header J1 (Figure 1.5) allows debugging and programming the RISC-V core of the MAX78000.

# Chapter 2 ● The MAX78000 Microcontroller

## 2.1 ● Overview

The MAX78000 is an advanced ultra-low-power microcontroller, developed to help implement artificial intelligence (AI), as well as standard embedded microcontroller solutions. Among the Arm Cortex-M4F processor, the MAX78000 includes an integrated Convolutional Neural Network (CNN) accelerator. In this chapter, we will be looking at some of the basic hardware details of this microcontroller (much of this Chapter has been taken from the **MAX78000 User Guide: MAX78000 User Guide, UGXXXX; Rev 0.5; 08/31/2020**)

## 2.2 ● Basic features of the MAX78000

One of the problems with developing Artificial Intelligence (AI) based products is the requirement for extreme computational power. This can only be achieved by having a fast processor in addition to using a processor dedicated to AI. The MAX78000 is a new AI microcontroller built to enable neural network applications to be executed with ultra-low-power and ease of use.

The MAX78000 is an advanced microcontroller featuring an Arm Cortex-M4 with FPU CPU for efficient system control and a Convolutional Neural Network (CNN) accelerator. The CNN has a weight storage SRAM memory of 442 KB, and can support 1-, 2-, 4-, and 8-bit weights, enabling AI network updates to be made on the fly. Additionally, the CNN engine has 512 KB of data memory, and because of its highly flexible architecture, it allows networks to be trained in conventional toolsets (e.g. PyTorch and TensorFlow) and converted for execution on the MAX78000 using Maxim tools.

In addition to the memory in the CNN engine, the MAX78000 has large 512 KB flash core memory and 128 KB SRAM. The microcontroller supports high-speed communication interfaces, such as I2S, I2C, SPI, UART, camera interface, and so on.

The MAX78000 microcontroller can be used in the following applications where high processing speeds are required:

- Audio processing (e.g. sound classification, noise cancellation, multi-keyword recognition, etc)
- Facial recognition
- Time-series data processing (heart rate signal analysis, health signal analysis, multi-sensor analysis, etc)

The MAX78000 microcontroller is available in 81-pin and 130-pin packages. Figure 2.1 shows the simplified internal structure of the MAX78000 microcontroller. On the top right of the figure is the CNN engine communicating over the internal multi-layer bus matrix. RISC-V and the clock circuitry are located on the top left of the figure. Below the CNN engine we can see the peripheral control devices which consist of the following modules:

- I2S master/slave
- 3 x I2C
- 3 x UART and 1 x LPUART
- 2 x SPI
- 1-wire master
- 4 x pulse train engines
- 4 x 32-bit timers and wake-up timer
- 2 x 32-bit LPTIMERS
- 1 x parallel camera interface

In the middle left part of the figure we can see the various memory modules, including the cache and boot memories. The bottom left part of the figure is where voltage regulation, dynamic voltage scaling, and power control are handled. Finally, at the bottom part of the figure, we can see the DMA, security modules, ADC, and comparator modules.

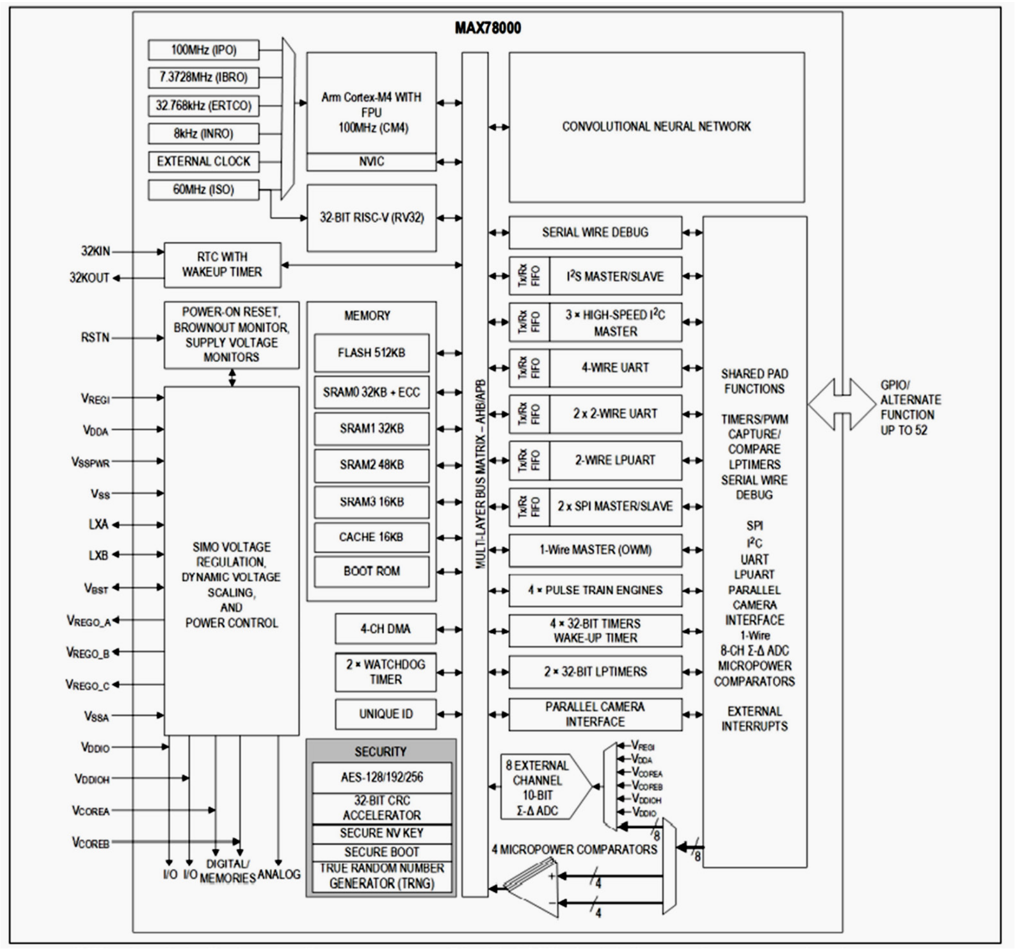


Figure 2.1 Internal structure of the MAX78000 microcontroller

### 2.2.1 ● The Convolutional Neural Network Accelerator (CNN)

The CNN engine is at the heart of the MAX78000 and it consists of 64 parallel processors with 512 KB of SRAM. Each of the processors includes a pooling unit and a convolutional engine with dedicated weight memory, where 4 processors share one 32KB data memory. These are further organised into groups of 16 processors that share common controls. A group of 16 processors operates as a slave to another group or independently. Data is read from SRAM associated with each processor and written to any data memory located within the accelerator. Any given processor has the visibility of its dedicated weight memory and to the data memory instance it shares with the three others.

Further information on the CNN engine features can be obtained from the document: **MAX78000 Ultra-Low-Power Arm Cortex-M4 Processor with FPU-Based Microcontroller with Convolutional Neural Network Accelerator**, located on the website:

https://www.maximintegrated.com/en/products/microcontrollers/MAX78000.html

### 2.2.2 ● The memory

The internal flash memory providing non-volatile storage of programs and data is 512 KB. Internal SRAM is 128 KB that provides the retention of application data in all power modes except POWER DOWN. The SRAM is divided into 4 banks (see Figure 2.1): SRAM0 and SRAM1 are both 32 KB, SRAM2 is 48 KB and SRAM3 is 16 KB.

### 2.2.3 ● Comparators

The eight AIN[7:0] inputs can be configured as four pairs and deployed as four independent comparators with the following features:

- Comparison events can trigger interrupts
- Events can wake the CM4 from SLEEP, LOW POWER, MICRO POWER, STANDBY, or BACKUP operating modes
- Can be active in all power modes

### 2.2.4 ● Clocking

The following can be selected as the clock sources (Figure 2.2):

- Internal primary oscillator (IPO) at a nominal frequency of 100MHz
- Internal secondary oscillator (ISO) at a nominal frequency of 60MHz
- Configurable internal nano-ring oscillator (INRO) at 8kHz, 16kHz, or 30kHz
- External RTC oscillator at 32.768kHz (ERTCO) (external crystal required)
- Internal baud rate oscillator at 7.3728MHz (IBRO)
- External square-wave clock up to 80MHz

There are multiple external clock inputs:

- LPTMR0 and LPTMR1 can be clocked from unique external sources
- I2S can be clocked from its own external source

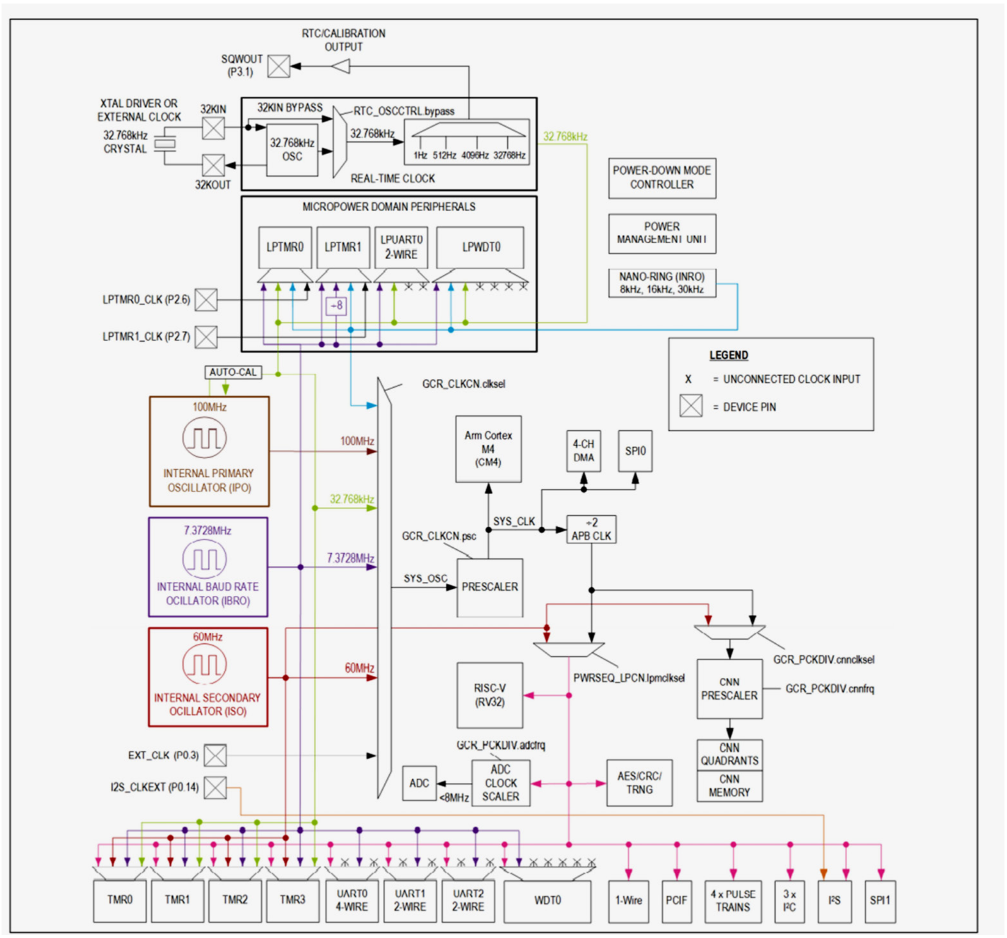


Figure 2.2 Clock sources

### 2.2.5 ● General-purpose input-output (GPIO) and special function pins

The MAX78000 microcontroller provides up to 52 GPIO pins. Most general-purpose I/O (GPIO) pins share both a firmware-controlled I/O function and one or more alternate functions, associated with peripheral modules. Pins can be individually enabled for GPIO or peripheral special function use.

Configuring a pin as a special function usually supersedes its use as a firmware-controlled I/O. Although this multiplexing between peripheral and GPIO functions is usually static, it can also be done dynamically. In GPIO mode, pins are logically divided into ports of 32 pins. Each pin of a port has an interrupt function that can be independently enabled and configured as a level or edge-sensitive interrupt. All GPIOs of a given port share the same interrupt vector.

When configured as GPIO, the following features are provided, which can be independently enabled or disabled:

- Configurable as input, output, bidirectional, or high impedance
- Optional internal pull-up resistor or internal pull-down resistor when configured as input
- Exit from low-power modes on rising or falling edge
- Selectable standard- or high-drive modes

### 2.2.6 ● Parallel Camera Interface (PCIF)

The Parallel Camera Interface (PCIF) is a peripheral designed to read data from camera sensors. The PCIF is a low voltage interface suited for CMOS image sensors. It provides up to 12-bits of parallel access capability with single capture and continuous mode operation.

### 2.2.7 ● Analog-to-Digital Converter (ADC)

The ADC resolution is 10-bits with an 8MHz maximum clock rate, providing an integrated reference generator and single-ended input multiplexer. The multiplexer selects an input channel from one of the eight external analog input signals (AIN0–AIN7) or the internal power supply inputs.

There are two sources for ADC reference voltage:

- Internal 1.22V bandgap
- VSSA analog supply

An optional feature allows samples captured by the ADC to be automatically compared against user-programmable high and low limits. Up to four-channel limit pairs can be configured in this way. The comparison allows the ADC to trigger an interrupt (and potentially wake the CPU from a power mode) when a captured sample goes outside the pre-programmed limit range. Since this comparison is performed directly by the sample limit monitors, it can be performed even while the CPU is in SLEEP, LOW POWER, or MICRO POWER mode. The eight AIN[7:0] inputs of the ADC can be configured as four pairs and deployed as four independent comparators.

In addition to the 8 inputs, the ADC can measure several other voltages (see the MAX78000 datasheet for further information)

### 2.2.8 ● Power management (PMU)

The power management unit (PMU) provides high-performance operation while minimising power consumption. It exercises intelligent, precise control of power distribution to the CPUs and peripheral circuitry.

The following modes of operation are available (see Table 2.1):

**ACTIVE Mode**

This is the normal operational mode, all digital and analog peripherals are available on demand. Peripherals not in use can be disabled by dynamic clocking.

**SLEEP Mode**

This mode consumes less power but wakes faster because the clocks can optionally be enabled.

**LOW POWER Mode (LPM)**

This mode is suitable for running the RISC-V processor to collect and move data from enabled peripherals.

**MICRO POWER Mode (μPM)**

This mode is used for extremely low power consumption while using a minimal set of peripherals to provide wake-up capability.

**STANDBY Mode**

This mode is used to maintain the system operation while keeping time with the RTC.

**BACKUP Mode**

This mode is used to maintain the system RAM.

**POWER DOWN Mode (PDM)**

This mode is used during product level distribution and storage. All oscillators are powered down and there is no data retention in this mode, however values in the flash are preserved.

| | CM4 | RV32 | CNN | Periph. | GPIO | DMA | Clocks |
|---|---|---|---|---|---|---|---|
| **Active** | ON | ON | ON | ON | ON | ON | All ON |
| **Sleep** | Sleep | Sleep | Optional | ON | ON | ON | All ON |
| **Low power** | Retained | Retained | ON | ON | ON | Retained | IPO: OFF |
| **Micro power** | Retained | Retained | Memory: retained<br>Quadrants: OFF | Retained | Retained | Retained | IPO: OFF<br>ISO: OFF |
| **Standby** | Retained | Retained | Memory: retained<br>Quadrants: OFF | Retained | Retained | Retained | IPO: OFF<br>ISO: OFF |
| **Backup** | OFF | OFF | Memory: retained<br>Quadrants: OFF | OFF | OFF | OFF | IPO: OFF<br>ISO: OFF<br>IBRO:<br>OFF<br>INRO:<br>OFF |
| **Power down** | OFF | OFF | OFF | OFF | OFF | OFF | OFF |

Table 2.1 Power management modes comparison

In all operating modes other than ACTIVE, wakeup sources are required to re-enter ACTIVE operation.

### 2.2.9 ● Real-time clock (RTC)

The Real-Time Clock (RTC) is a 32-bit binary timer that keeps the time of day up to 136 years. It provides time-of-day and sub-second alarm functionality in the form of system interrupts. The 32-bit seconds register can count up to approximately 136 years and be translated to calendar format by application software. The RTC provides a time-of-day alarm that can be programmed to any future value between 1 second and 12 days. When configured for long intervals, the time-of-day alarm can be used as a power-saving timer, allowing the device to remain in an extremely low-power mode, but still awaken periodically to perform assigned tasks. A second independent 32-bit 1/4096 sub-second alarm can be programmed with a tick resolution of 244µs. Both can be configured as recurring alarms. When enabled, either alarm can cause an interrupt or wake the device from most low-power modes. The time base is generated by a 32.768kHz crystal or an external clock source.

### 2.2.10 ● Programmable timers

#### 32-Bit Timer/Counter/PWM (TMR, LPTMR)

General-purpose, 32-bit timers provide timing, capture/compare, or generation of pulse-width modulated (PWM) signals with minimal software interaction.

The timer provides the following features:

- 32-bit up/down auto-reload
- Programmable prescaler
- PWM output generation
- Capture, compare, and capture/compare capability
- External pin multiplexed with GPIO for timer input, clock gating, or capture

- Timer output pin
- TMR0–TMR3 can be configured as 2 × 16-bit general-purpose timers
- Timer interrupt

The MAX78000 provides six 32-bit timers (TMR0, TMR1, TMR2, TMR3, LPTMR0, and LPTMR1). LPTMR0 and LPTMR1 are capable of operation in the SLEEP, LOW POWER, and MICRO POWER modes.

I/O functionality is supported for all of the timers. Note that the function of a port can be multiplexed with other functions on the GPIO pins, so it might not be possible to use all of the ports depending on the device configuration. See Table 3 for individual timer features

### 2.2.11 ● Watchdog timer (WDT)

The WDT on the MAX78000 is a 32-bit, free-running counter with a configurable Prescaler. When enabled, the WDT must be periodically reset by the application software. Failure to reset the WDT within the user-configurable timeout period indicates that the application software is not operating correctly and results in a WDT timeout. A WDT timeout can trigger an interrupt, system reset, or both.

### 2.2.12 ● Pulse train engine (PTE)

Multiple, independent pulse train generators can provide either a square-wave or a repeating pattern from 2 to 32 bits in length. Any single pulse train generator or any desired group of pulse train generators can be synchronized at the bit level, allowing for multibit patterns. Each pulse train generator is independently configurable.

### 2.2.13 ● I2C Interface (I2C)

The I2C interface is a bidirectional, two-wire serial bus that provides a medium-speed communications network. It can operate as a one-to-one, one-to-many, or many-to-many communications medium.

### 2.2.14 ● I2S Interface (I2S)

The I2S interface is a bidirectional, four-wire serial bus that provides serial communications for codecs and audio amplifiers compliant with the I2S Bus Specification. Stereophonic (2 channel) and monophonic (left or right channel option) both master and slave formats are supported.

### 2.2.15 ● Serial Peripheral Interface (SPI)

SPI is a highly configurable, flexible, and efficient synchronous interface among multiple SPI devices on a single bus. The bus uses a single clock signal and multiple data signals, and one or more slave select lines to address only the intended target device. The SPI operates independently and requires minimal processor overhead.

### 2.2.16 ● UART (UART, LPUART)

The universal asynchronous receiver-transmitter (UART, LPUART) interface supports full-duplex asynchronous communication with optional hardware flow control (HFC) modes to prevent data overruns. If HFC mode is enabled on a given port, the system uses two extra pins to implement the industry-standard request to send (RTS) and clear to send (CTS) flow control signalling. Each instance is individually programmable.

### 2.2.17 ● 1-Wire Master (OWM)

Maxim's 1-Wire bus consists of one signal that carries data and also supplies power to the slave devices and a ground return. The bus master communicates serially with one or more slave devices through the bidirectional, multi-drop 1-Wire bus. The single-contact serial interface is ideal for communication networks requiring minimal interconnection.

### 2.2.18 ● DMA Controller

The standard DMA controller allows high-speed automatic one-way data transfer between two entities. These entities can be either memories or peripherals. The transfers are done without using CPU resources. Interrupts can be enabled for each DMA channel.

### 2.2.19 ● Security (AES)

The dedicated hardware-based AES engine supports the following algorithms:

- AES-128
- AES-192
- AES-256

The AES keys are automatically generated by the engine and stored in dedicated flash to protect against tampering. Key generation and storage are transparent to the user.

### 2.2.20 ● True Random Number Generator (TRNG) Non-Deterministic Random Bit Generator (NDRBG)

The device provides a non-deterministic entropy source that can be used to generate cryptographic seeds or strong encryption keys as part of an overall framework for a secure customer application. Software can use random numbers to trigger asynchronous events that add complexity to program execution to thwart replay attacks or key search methodologies.

### 2.2.21 ● CRC

A Cyclic Redundancy Check (CRC) hardware module provides fast calculations and data integrity checks by application software. It supports a user-defined programmable polynomial up to 32-bits.

### 2.2.22 ● Bootloader

The bootloader allows loading and verification of program memory through a UART or SWD interface.

### 2.2.23 ● Device Resets

Four device resets are available:

- Peripheral
- Soft
- System
- Power-On

All peripherals are reset with the *Peripheral Reset*, the CPU retains its state. The GPIO, watchdog timers, General Control Registers (GCR), including the clock configuration, are unaffected.

*Soft Reset* is similar to *Peripheral Reset*, but it also resets the GPIO to its power-on reset state.

*System Reset* is similar to Soft Reset except it also resets all GCR, resetting the clocks and the CPU to their default state.

Everything is reset to their default states with the *Power-On Reset*.

### 2.2.24 ● Interrupts and Exceptions

Interrupts and exceptions are managed by the Arm Cortex-M4 with FPU Nested Vector Interrupt Controller (NVIC) or the RV32 interrupt controller. 8 programmable priority levels are provided with nested exception and interrupt support, and interrupt masking.

# Chapter 3 • Beginning with the MAX78000FTHR Development Board

## 3.1 • Overview

In this chapter, we will learn how to install software on a PC so we can start writing programs for the MAX78000FTHR microcontroller development board.

## 3.2 • Installing the Eclipse MaximSDK Software

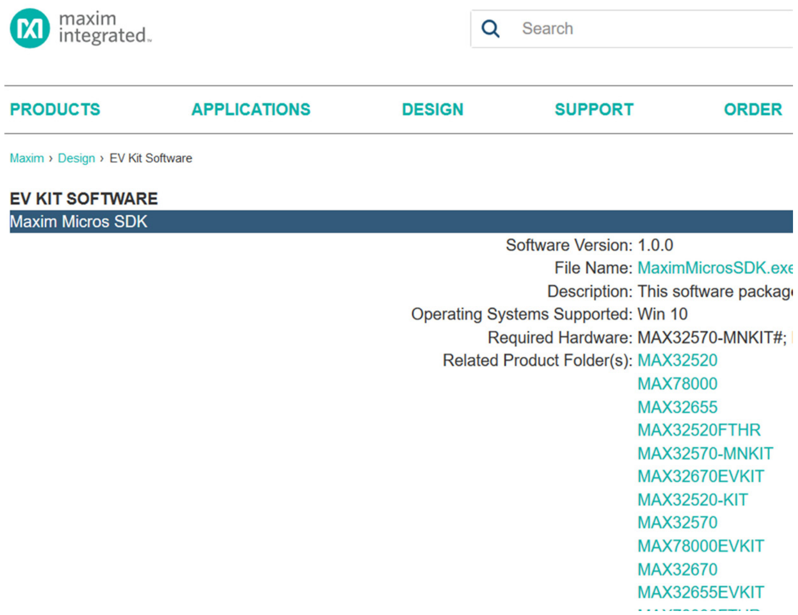
The quickest and most recommendable way to develop programs for the MAX78000FTHR is to install **Eclipse MaximSDK** on your PC (Windows 10).

The steps are:

- Open the following link on your web browser:

https://www.maximintegrated.com/en/design/software-description.html/swpart=SFW0010820A

- Click on **MaximMicrosSDK.exe** (Figure 3.1)



EV KIT SOFTWARE

Maxim Micros SDK

| | |
|---|---|
| Software Version: | 1.0.0 |
| File Name: | MaximMicrosSDK.exe |
| Description: | This software packag |
| Operating Systems Supported: | Win 10 |
| Required Hardware: | MAX32570-MNKIT#; |
| Related Product Folder(s): | MAX32520 |
| | MAX78000 |
| | MAX32655 |
| | MAX32520FTHR |
| | MAX32570-MNKIT |
| | MAX32670EVKIT |
| | MAX32520-KIT |
| | MAX32570 |
| | MAX78000EVKIT |
| | MAX32670 |
| | MAX32655EVKIT |

Figure 3.1 Click on MAX78000FTHR

- Click the **Download** button to download the **Maxim Micros SDK** (you will find some other useful files in this link, such as the MAX78000FTHR schematic, bill of materials, PCB, etc). See Figure 3.2

Figure 3.2 Click on Download

- You will have to register at this point with Maxim before you can download the file. Follow the required registration details. You will get an email to validate your registration
- The program will be copied to your **Download** folder. Double click to install the program, after accepting the license conditions. The installation will take some time.
- After the installation is complete, the program **Eclipse MaximSDK** should be under **Maxim Integrated SDK**. You should always start the SDK using the command **C:\MaximSDK\Tools\Eclipse\cdt\eclipse.bat**. This is because the **.bat** sets environment variables and also calls the SDK updater to update documentation provided with the SDK, bug fixes, and potentially newer examples.

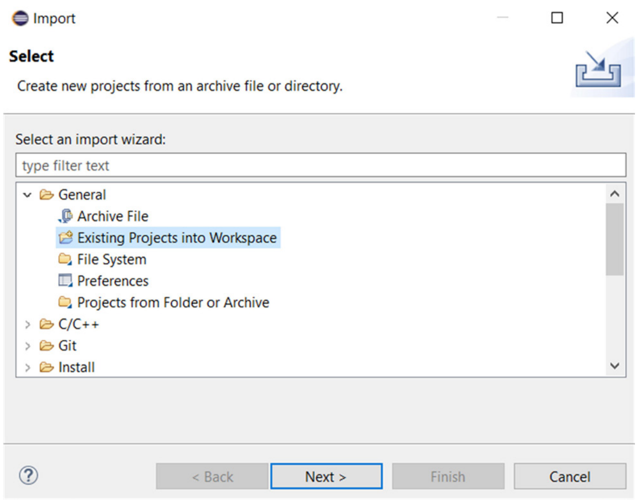
## 3.3 ● Using the Eclipse MaximSDK – example MAX78000FTHR program

The SDK contains many example projects for all integrated peripherals, such as I2C, GPIO, UART, etc. Looking at these example projects could be very helpful while developing new programs. In this section, we will look at a simple example program (**Hello_World**) which flashes the onboard LED every 0.5 seconds and also displays count on a terminal through its UART output.

Loading all of the example programs (except the CNN based programs):

The steps to load all the example programs from the installation folders are as follows:

- Start Eclipse MaximSDK and enter a new Workspace name
- Click **File -> Import -> General -> Existing Projects -> Next** (Figure 3.3)


Import

Select

Create new projects from an archive file or directory.

Select an import wizard:

type filter text

Figure 3.3 Click New and the Maxim Microcontrollers

- Browse to **C:\MaximSDK\Examples\MAX78000** if you didn't change your install path.
- Make sure to check **Copy projects into workspace**
- Click **Finish**

You should now see all the example projects loaded into your workspace (Figure 3.4)

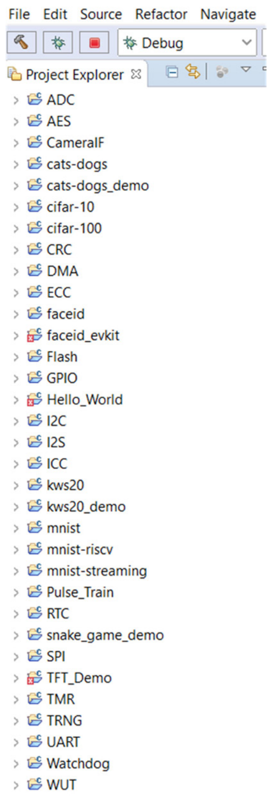
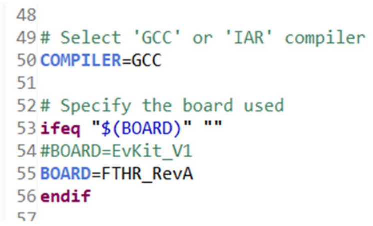


Figure 3.4 All example projects loaded into workspace

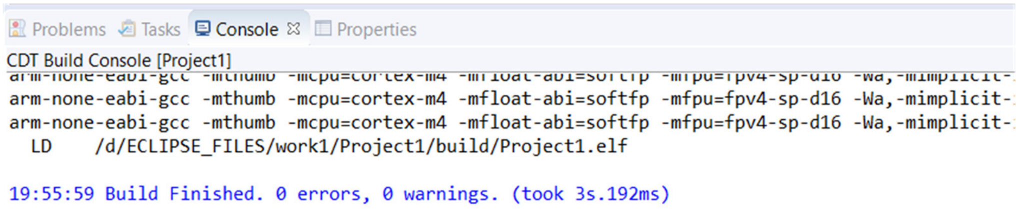
- Now, let us select, compile and run the program named **Hello_World**. Expand this program and then double click on file **Makefile**. Change the line shown in Figure 3.5 so that the program is configured for the MAX78000FTHR board.



```

# Select 'GCC' or 'IAR' compiler
COMPILER=GCC

# Specify the board used
ifeq "$(BOARD)" ""
#BOARD=EvKit_V1
BOARD=FTHR_RevA
endif
```

Figure 3.5 Edit file Makefile

- Double click on **main.c** to list the program
- Click on **Project -> Build Project** to build the project. Check the console display at the bottom part of the screen to make sure that there are no build errors (see Figure 3.6).



```
arm-none-eabi-gcc -mthumb -mcpu=cortex-m4 -mfloat-abi=softfp -mfpu=fpv4-sp-d16 -Wa,-mimplicit-
arm-none-eabi-gcc -mthumb -mcpu=cortex-m4 -mfloat-abi=softfp -mfpu=fpv4-sp-d16 -Wa,-mimplicit-
arm-none-eabi-gcc -mthumb -mcpu=cortex-m4 -mfloat-abi=softfp -mfpu=fpv4-sp-d16 -Wa,-mimplicit-
  LD    /d/ECLIPSE_FILES/work1/Project1/build/Project1.elf

19:55:59 Build Finished. 0 errors, 0 warnings. (took 3s.192ms)
```

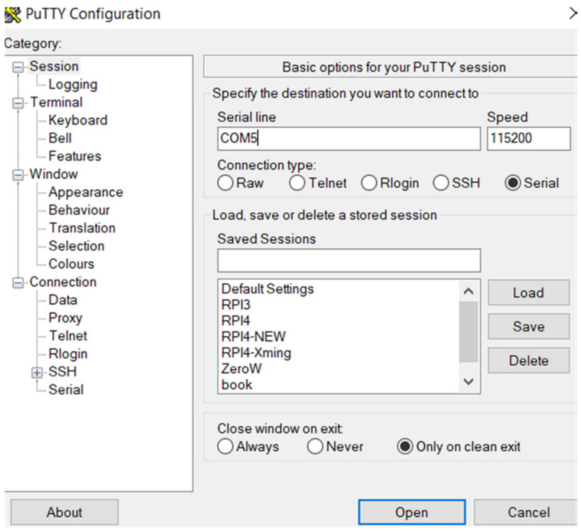
Figure 3.6 Make sure there are no build errors

This example was built in debug mode. At the beginning of the program, the header files of the functions used in the library are included. The program displays the message **Hello World!** on a terminal and then flashes the LED every 500ms. Additionally, a counter is incremented and displayed on a terminal emulation program, e.g. using e.g. Putty or Terraterm or HyperTerm. In this project, we will run the program in debug mode and observe the LED flashing and the counter updating on the terminal.

**Running in debug mode**

The steps to run the program in debug mode are:

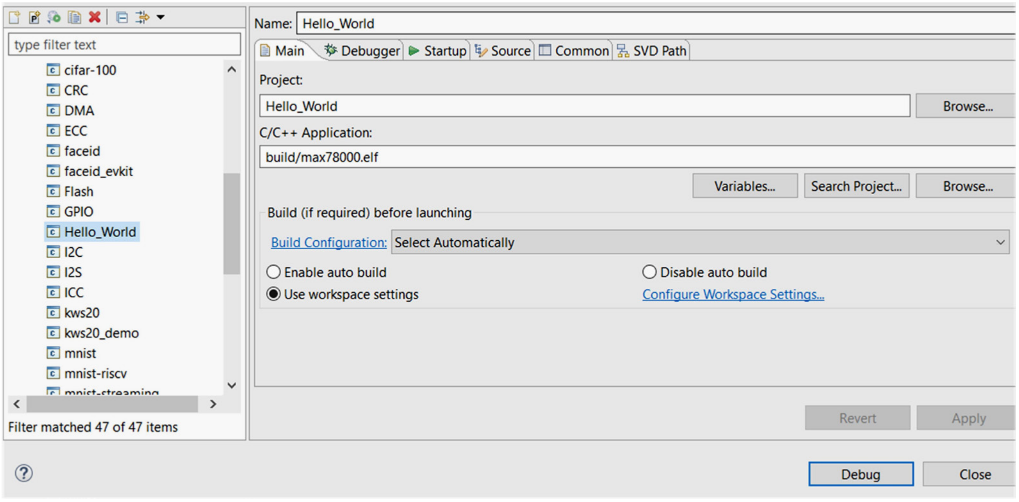
- Connect the MAX78000FTHR development board to your PC using a micro USB cable
- Open **Device Manager** on your PC and note the COM port name assigned to MAX78000FTHR (e.g. COM5)
- Run a terminal emulation program on your PC. For example, run Putty and set the **Connection type** to Serial, **Serial line** to COM5, and **Speed** to 115200 (see Figure 3.7)
- Click **Open** on Putty



Serial line: COM5
Speed: 115200
Connection type: Raw Telnet Rlogin SSH [✓ Serial]
Close window on exit: Always Never [✓ Only on clean exit]

Figure 3.7 Configure Putty

- Click **Run**, followed by **Debug Configurations** on your Eclipse
- Click **GDB OpenOCD Debugging** and then click **Hello_World** (Figure 3.8)



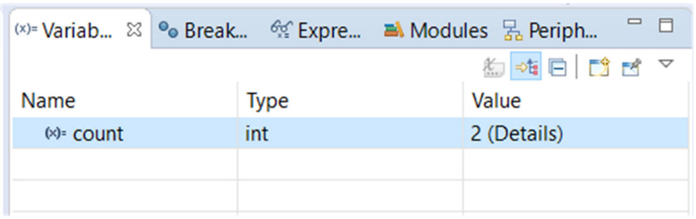
Name: Hello_World
Project: Hello_World
C/C++ Application: build/max78000.elf
Build (if required) before launching
Build Configuration: Select Automatically
Enable auto build
[✓ Use workspace settings]
Disable auto build
Configure Workspace Settings...

Figure 3.8 Debug Configuration

- Click **Debug** to start the debugging session
- You should now see the cursor positioned at the first executable statement of the program. The green LED should be flashing on your hardware to indicate that we are in the debug session
- Perhaps the easiest thing to do now is single-step through the program statements. Press Function key **F6** to move between the statements of the program. You should see the cursor moving to the next statements. The LED should turn ON just after the

**LED_On** statement is executed. Keep pressing **F6** for a few iterations of the loop to make sure that the program is stepping through the statements correctly.

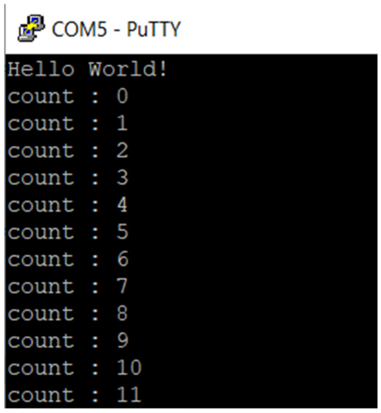
- We can examine the variables as we single-step through the program. For example, the value of variable **count** is displayed on the top right side of the screen (debug watch window) and this value will increment every time we go round the loop (Figure 3.9)



| Name | Type | Value |
|---|---|---|
| count | int | 2 (Details) |

Figure 3.9 Examine the value of count

- Make sure the **Variables** tab is selected at the debug watch window. Write click on variable **count** and you should be able to change the number base (decimal, hexadecimal, binary, and octal), display as array, etc.
- We can set breakpoints in the program and then run the program up to the breakpoint and examine the variables. For example, to set a breakpoint at the **printf** statement, place the cursor on the blue line at the left of line number 69 and right-click the mouse. Then, select **Add Breakpoint**. You should see a small green mark on the left-hand side of the statement to indicate the presence of the breakpoint. Now press keys **Ctrl+R**. You should see the program running and then stopping at the point where the breakpoint is placed. At the same time, the value of count will be incremented.
- To remove the breakpoint, place the cursor on the breakpoint marker and right-click the mouse. Select **Disable Breakpoint**. Alternatively, click **Run**, followed by **Remove All Breakpoints** to remove all breakpoints.
- All other breakpoint instructions are available by clicking **Run**.
- After testing a program, we may want to run it continuously. This can be done by clicking **Run**, followed by **Resume**, or simply by pressing Function key **F8**. You should now see the LED flashing continuously every 0.5 seconds and the count incrementing on the terminal emulation screen (Figure 3.10).



```
COM5 - PuTTY
Hello World!
count : 0
count : 1
count : 2
count : 3
count : 4
count : 5
count : 6
count : 7
count : 8
count : 9
count : 10
count : 11
```

Figure 3.10 Counting on the display

- To terminate the program, click **Run**, followed by **Terminate**, or simply **Ctrl+F2**.
- To return to normal programming mode, click the button **C/C++** on the top right corner of the screen (Figure 3.11)

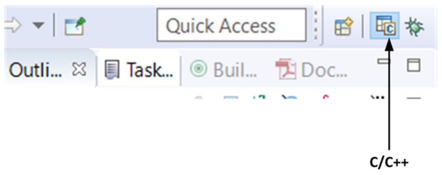


Figure 3.11 Click C/C++

You can also run the program continuously by restarting the MAX78000FTHR.

Notice the console of the MAX78000FTHR is routed to UART0 (this can be a header), connected to the MAX32625 debugger. You can display the output of the virtual com port with putty (blue flashing LED on the FTHR board indicate UART traffic). Nevertheless; this behaviour can be changed in file board.h in C:\MaximSDK\Libraries\Boards\MAX78000\FTHR_RevA\Include by changing #define CONSOLE_UART 0. If you want to have the console on pins 7 and 8 change it to 2.

**Examining the program**

The program listing is shown in Figure 3.12. Although the program is very simple, it is worthwhile to look at the operation of this program.

```
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "led.h"
#include "board.h"
#include "mxc_delay.h"

/***** Definitions *****/

/***** Globals *****/

/***** Functions *****/

// *****************************************************************************
int main(void)
{
    int count = 0;

    printf("Hello World!\n");

    while (1) {
        LED_On(LED1);
        MXC_Delay(500000);
        LED_Off(LED1);
        MXC_Delay(500000);
        printf("count : %d\n", count++);
    }
}
```

Figure 3.12 Program listing

<**stdio.h**> and <**stdint.h**> are the standard C header files.

**board.h** defines the configurations of the components on the board. For example, the UART port and speed, I2C ports, on-board LED states (LED_ON, LED_OFF), on-board LED ports (LED1 = 0, LED2 = 1), console initialisation, microphone power control, and camera power control.

**led.h** defines the on-board LED states, LED initialisation, and LED toggle

**mxc_device.h** defines the target device

**mxc_delay.h** defines the following delay functions:

| | |
|---|---|
| MXC_Delay(n) | n microseconds delay (blocking) |
| MXC_DelayAsync(n) | n microseconds delay (non-blocking) |
| MXC_DelayCheck() | returns the status of a non-blocking delay request |
| MXC_DelayAbort() | stop a delay started earlier |
| MXC_DelayHandler() | process the delay interrupt |

Additionally, the following Macros are defined by **mxc_delay.h**:

MXC_Delay_SEC()
MXC_Delay_MSEC()
MXC_Delay_US()

For example, we can use the function call **MXC_Delay(SEC(1))** to introduce a 1-second delay.

## <u>Loading a single example program</u>

There are several ways a single example program can be loaded from the SDK installation folders. The steps given below will load the example program **Hello_World**:

- Start Eclipse MaximSDK and enter a new Workspace name
- Click **File -> New**, followed by **Maxim Microcontrollers** (Figure 3.13)

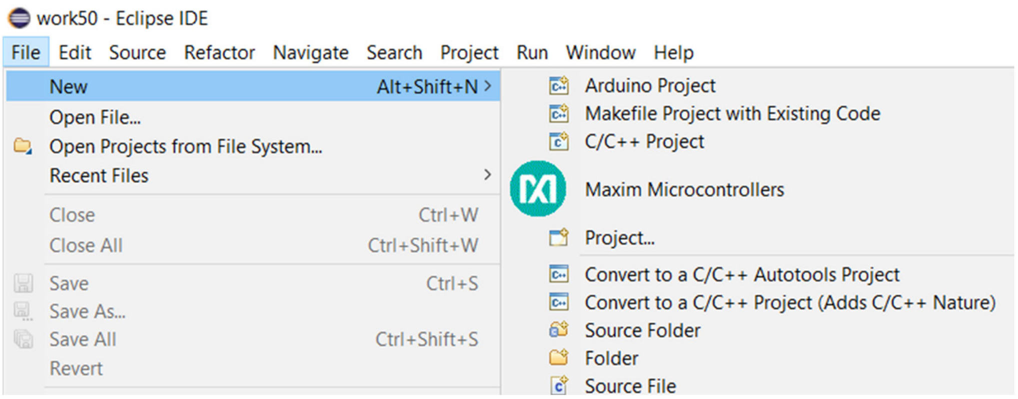
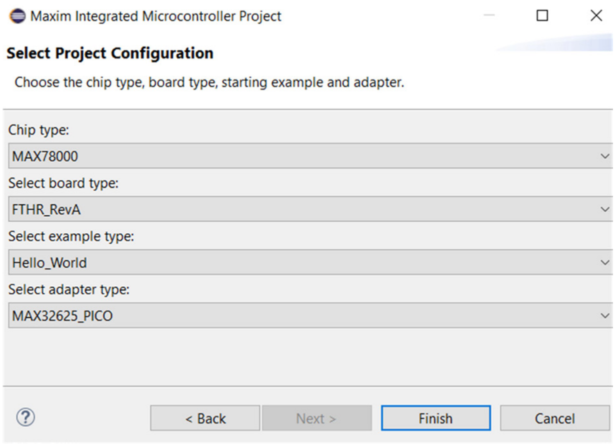


Figure 3.13 Click New and the Maxim Microcontrollers

- Enter a name for your project, e.g. **Project1**, and click **Next**
- Fill in the **Select Project Configuration** form as follows (see Figure 3.14). Notice Hello_World is the example program we will be using:

| | |
|---|---|
| **Chip type:** | MAX78000 |
| **Select board type:** | FTHR_RevA |
| **Select example type:** | Hello_World |
| **Select adapter type:** | MAX32625_PICO |


Maxim Integrated Microcontroller Project

Select Project Configuration

Choose the chip type, board type, starting example and adapter.

Chip type: MAX78000
Select board type: FTHR_RevA
Select example type: Hello_World
Select adapter type: MAX32625_PICO

Figure 3.14 Fill in the project configuration

- Click **Finish**. Eclipse will open your project so that you may edit, compile and debug. Alter the **Makefile** and change the board to **FTHR_RevA** as described earlier. Open main.c by double-clicking on it. The Eclipse **Project Explorer** window is shown in Figure 3.15. Figure 3.16 shows part of the program listing (the Maxim Copyright text at the beginning of the program has been omitted to save space. This text must however be included in all example programs as provided by Maxim).

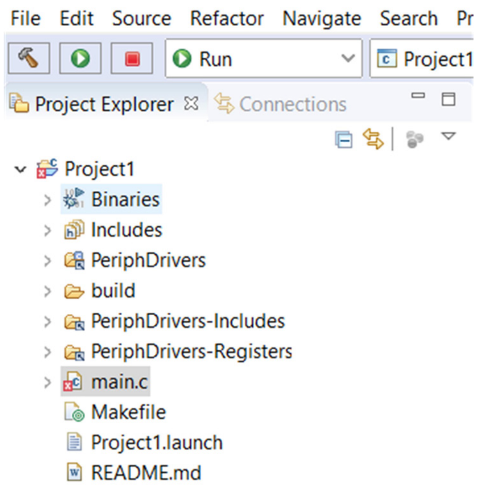
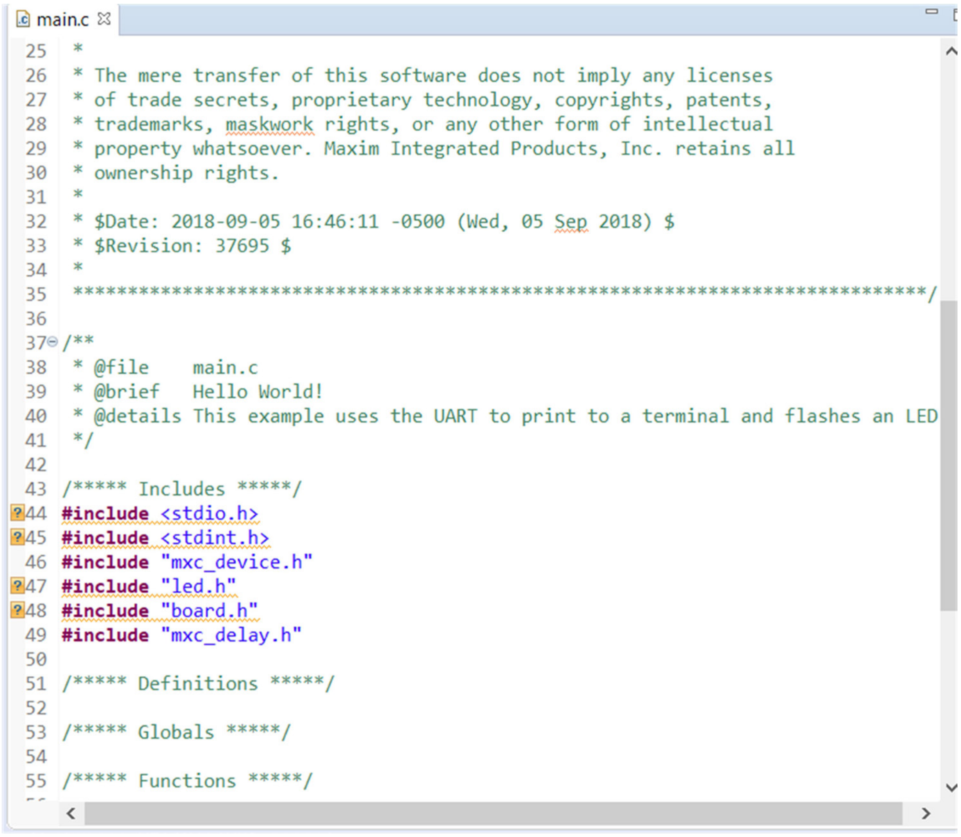


Figure 3.15 Eclipse Project Explorer window


main.c

```
 *
 * The mere transfer of this software does not imply any licenses
 * of trade secrets, proprietary technology, copyrights, patents,
 * trademarks, maskwork rights, or any other form of intellectual
 * property whatsoever. Maxim Integrated Products, Inc. retains all
 * ownership rights.
 *
 * $Date: 2018-09-05 16:46:11 -0500 (Wed, 05 Sep 2018) $
 * $Revision: 37695 $
 *
 ******************************************************************************/

/**
 * @file    main.c
 * @brief   Hello World!
 * @details This example uses the UART to print to a terminal and flashes an LED
 */

/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "led.h"
#include "board.h"
#include "mxc_delay.h"

/***** Definitions *****/

/***** Globals *****/

/***** Functions *****/
```

Figure 3.16 Part of the program listing

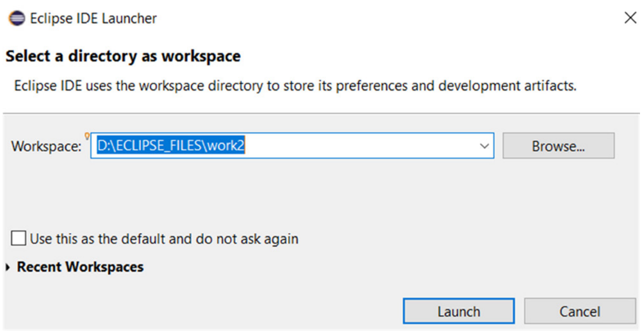
- Build the program by clicking **Project**, followed by **Build All** (or **Ctrl+B**).
- Click **Run**, followed by **Debug Configurations**. Click **GDB OpenOCD Debugging** and click **Project1**. The program should start in debug mode after a short while.
- Click **Run -> Resume** to run the program without any breakpoints

## 3.4 ● Project 1 – Creating a C program – display message

**Description**: In this section, we will create a program that displays the message **!!!Hello World!!!** on a PC screen. This program is independent of the MAX78000FTHR and is not loaded to its memory.

The steps are:

- Start Eclipse MaximSDK
- Give a name to your Workspace (e.g. **Work2**) as shown in Figure 3.17


Eclipse IDE Launcher

**Select a directory as workspace**

Eclipse IDE uses the workspace directory to store its preferences and development artifacts.

Workspace: D:\ECLIPSE_FILES\work2

Figure 3.17 Give a name to your workspace

- Click **New** followed by **Projects** under **Maxim Microcontrollers** (Figure 3.18)

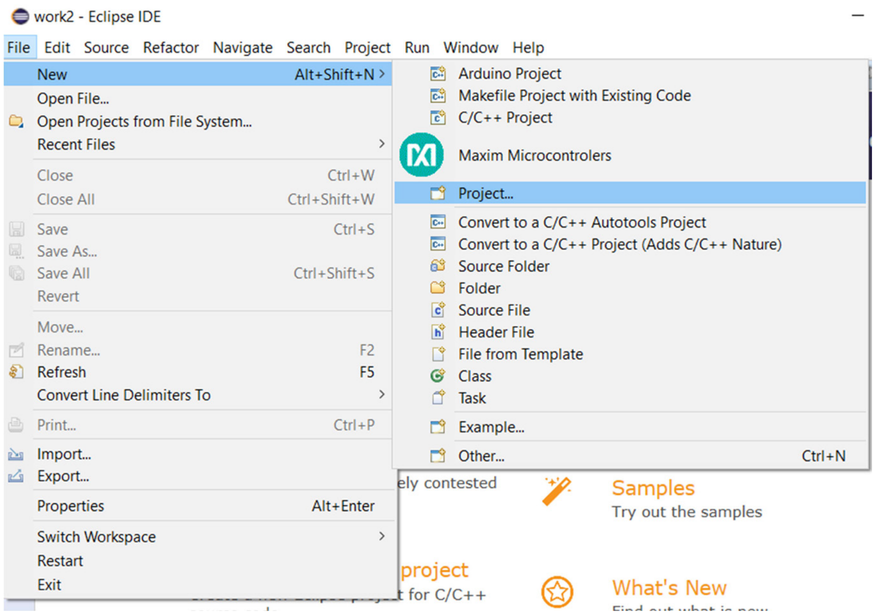
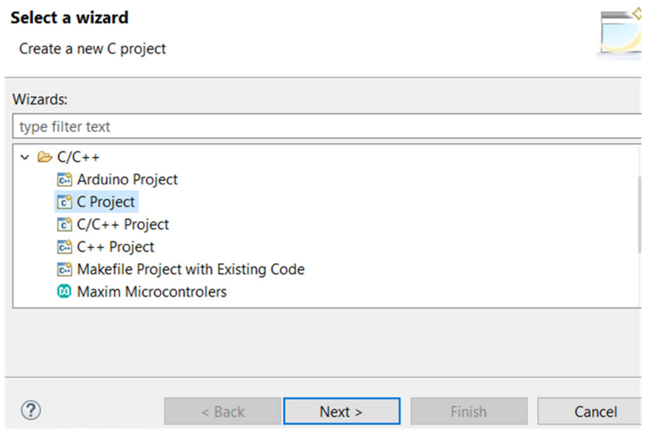


Figure 3.18 Select Projects

- Click to expand **C/C++** and then press **C Project** (Figure 3.19). Click **Next** to continue



**Select a wizard**

Create a new C project

Wizards:

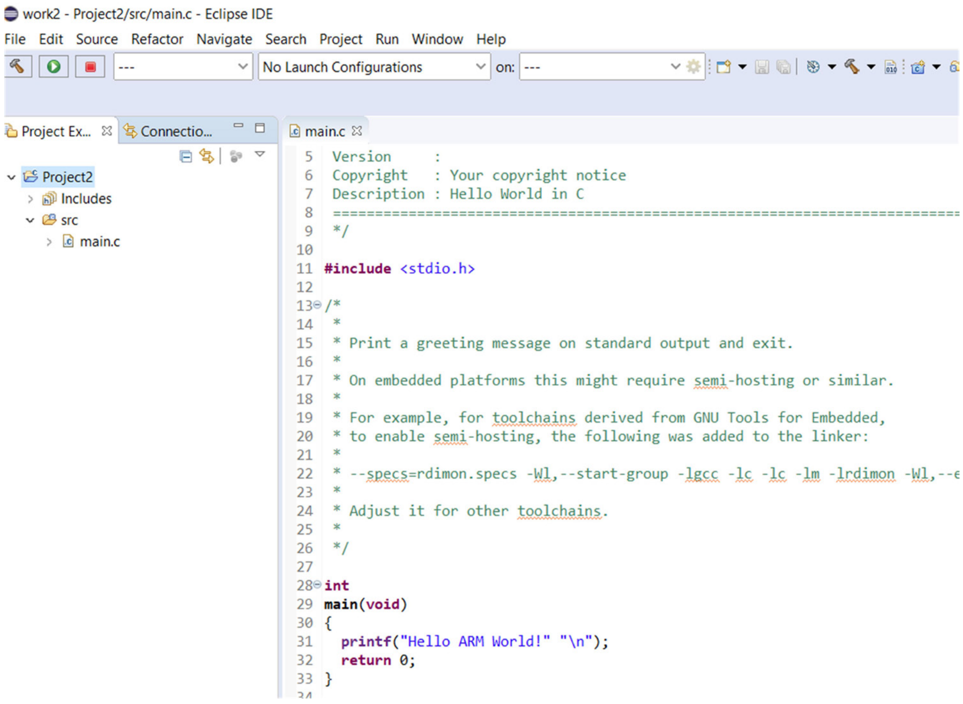
type filter text

- C/C++
  - Arduino Project
  - C Project
  - C/C++ Project
  - C++ Project
  - Makefile Project with Existing Code
  - Maxim Microcontrolers

< Back | Next > | Finish | Cancel

Figure 3.19 Click C Project

- Select **Hello World ANSI C Project** under **Project type**, and select **MINGW GCC** as the **Toolchain** give a name to your project (e.g. **Project2**) and click **Next**
- Give a name to the author field and click **Next**
- Check both **Debug** and **Release**
- Click **Next** and **Finish**
- You should now see your workspace with the program as shown in Figure 3.20



work2 - Project2/src/main.c - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
No Launch Configurations
on:
Project Ex...
Connectio...
main.c
Project2
Includes
src
main.c

```
 Version     :
 Copyright   : Your copyright notice
 Description : Hello World in C
 ============================================================================
 */

#include <stdio.h>

/*
 *
 * Print a greeting message on standard output and exit.
 *
 * On embedded platforms this might require semi-hosting or similar.
 *
 * For example, for toolchains derived from GNU Tools for Embedded,
 * to enable semi-hosting, the following was added to the linker:
 *
 * --specs=rdimon.specs -Wl,--start-group -lgcc -lc -lc -lm -lrdimon -Wl,--
 *
 * Adjust it for other toolchains.
 *
 */

int
main(void)
{
  printf("Hello ARM World!" "\n");
  return 0;
}
```

Figure 3.20 Workspace with the program

- Click **Project -> Build configurations -> Set Active** and select **Release**
- Click **Project**, followed by **Build All** (or simply click Ctrl+B) to build the project. Make sure there are no error messages are displayed at the bottom part (Console) of the screen
- Click **Run**, followed by **Run configurations**
- Double click **C/C++ Application**, and click **Run**
- You should see the message **!!!Hello World!!!** Displayed on the screen

**Description:** In this project, we will turn ON/OFF the onboard RGB LED in sequence as follows: First the Red colour will be ON, then after a second the Red will be OFF and the Green colour will be ON. Then the Green will be OFF and the Blue colour will be ON. Finally, all colours will be OFF. This sequence will be repeated until stopped by the user.

The steps to access this simple example are the same as in Section 3.3 and are repeated below:

- Start the Eclipse MaximSDK and click New, followed by Maxim Microcontrollers
- Enter a name for your project, e.g. RGB, and click Next
- Fill in the Select Project Configuration form. Notice Hello_World is the example program we will be using:

| | |
|---|---|
| **Chip type**: | MAX78000 |
| **Select board type**: | FTHR_RevA |
| **Select example type**: | Hello_World |
| **Select adapter type**: | MAX32625_PICO |

- Alter the **Makefile** and change the board to **FTHR_RevA** as described earlier. Open main.c by double-clicking. Delete the main program code and enter the code shown in Figure 3.21. In this program, function **LED_RGB** is called to turn ON/OFF a specified colour of the onboard RGB LED. For example, calling **LED_RGB(1, 0, 0)** turns ON the Red LED and so on.

```
/*******************************************************************************
                         ON-BOARD RGB LED CONTROL
                         ========================

This program flashes the on-booard RGB LED colours

Author: Dogan Ibrahim
File: LED
Date: March 2021
 ******************************************************************************/
#include "mxc_device.h"
#include "led.h"
#include "board.h"
#include "mxc_delay.h"
```

```
//
// Turn ON/OFF the on-board RGB LED colours
//
void LED_RGB(uint8_t r, uint8_t g, uint8_t b)
{
        if (r == 0)
                LED_Off(LED_RED);
        else
                LED_On(LED_RED);

        if (g == 0)
                LED_Off(LED_GREEN);
        else
                LED_On(LED_GREEN);

        if (b == 0)
                LED_Off(LED_BLUE);
        else
                LED_On(LED_BLUE);
}

int main(void)
{
        while(1)
        {
                LED_RGB(1,0,0);                             // RED ON
                MXC_Delay(SEC(1));                          // Wait 1 second
                LED_RGB(0,1,0);                             // GREEN ON
                MXC_Delay(SEC(1));                          // Wait 1 second
                LED_RGB(0,0,1);                             // BLUE ON
                MXC_Delay(SEC(1));                          // Wait 1 second
                LED_RGB(0,0,0);                             // All OFF
                MXC_Delay(SEC(1));                          // Wait 1 second
        }
}
```

Figure 3.21 Program listing

Build the program by clicking **Project**, followed by **Build All** (or **Ctrl+B**) to build the program. Click **Run**, followed by **Debug Configurations**. Click **GDB OpenOCD Debugging** and click **LED**. The program should start to run after a short while.

# Chapter 4 • Simple MAX78000FTHR Hardware Projects

## 4.1 ● Overview

In this chapter, we will develop simple embedded projects to show how the MAX78000FTHR board can be used in projects. These projects aim to make readers more familiar with the various features of the MAX78000FTHR development board.

The following sub-headings are given (where appropriate) for each project:

- Title
- Description
- Aim of the project
- Block diagram
- Circuit diagram
- Project construction
- Program listing and a full description
- Suggestions for more work

It is recommended that the readers start from the first project since some of the later projects may depend on the hardware/software of the earlier projects.

## 4.2 ● Project 1 – External flashing LED (+1.8V output port voltage)

**Description:** In this project, an external LED is connected to one of the GPIO ports of the MAX78000. The project flashes the LED every 250ms.

**Aim:** This project aims to show how an external LED can be connected and controlled from a program.

**Block diagram:** Figure 4.1 shows the block diagram of the project

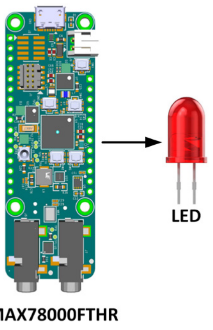


Figure 4.1 Block diagram of the project

**Circuit diagram**: There are two basic ways that we can connect an LED to a microcontroller output port: in source mode, and sink mode.

Figure 4.2 shows the connection in source mode. Here, the cathode of the LED is connected to ground. The anode is connected to the output port through a current limiting resistor. In this configuration, the LED is turned ON when logic 1 is applied to the output port pin.

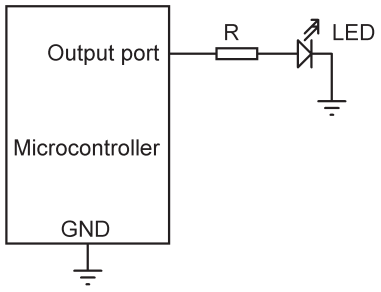


Figure 4.2 Connecting in current source mode

The value of the current limiting resistor can be calculated as follows: The voltage drop across a red LED is about 2V. The current through the LED depends on the LED specifications and can vary from a few milliamperes to up to 16mA. In this project, we will be using small LEDs with 1mA operating current. The required resistor value is then given by:

R = (Voh – 2) / 1mA

Where Voh is the output HIGH voltage of the microcontroller, and R is in KOhms.

**The output HIGH voltage of the MAX78000 microcontroller in default mode is only +1.8V** and therefore the LED cannot be connected in source mode since the voltage is less than the required LED operating voltage (*we will see in a later project how to change the port output voltage to +3.3V*).

Figure 4.3 shows the LED connected in current sink mode. In this configuration, the anode of the LED is connected to the supply voltage. The LED is turned ON when the port output pin is at logic 0. In this configuration, the value of the current limiting resistor is calculated as:

R = (Vs - Vol – 2) / 1mA

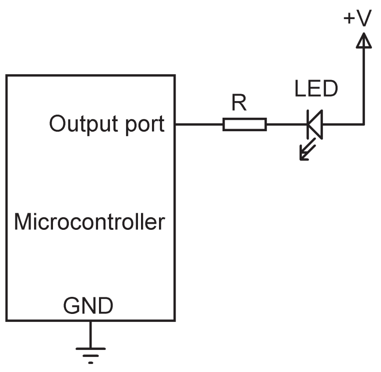


Figure 4.3 Connecting in current sink mode

Where Vs is the supply voltage, Vol is the output low voltage of the port pin and R is in KOhms. **The typical Vol of the MAX78000 microcontroller is 0.2V**. Using a supply voltage of Vs = 3.3V, the required resistor value is:

**R = (3.3 – 0.2 – 2) / 1mA**

Which gives R = 1100 Ohm. We will choose 1K for a slightly higher current than 1mA which will give a slightly higher brightness.

Figure 4.4 shows the circuit diagram of the project where the LED is connected in current sink mode. Pin 7 of Port 2 (P2_7) is used in this project. Power is supplied from +3.3V (pin 2) of the MAX78000FTHR.

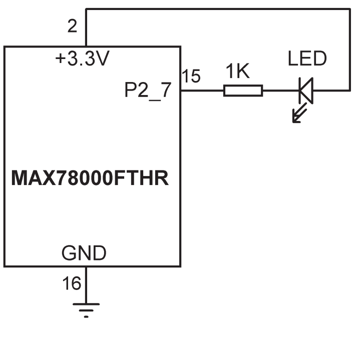


Figure 4.4 Circuit diagram of the project

**Construction**: Figure 4.5 shows the project constructed on a breadboard and connections made using jumper wires.

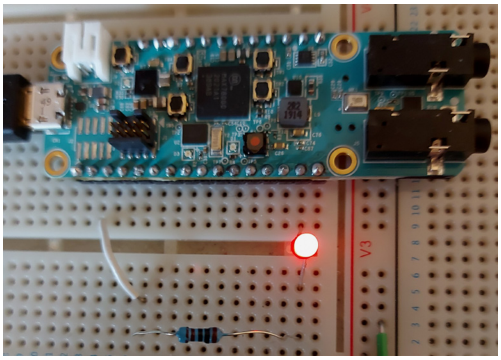


Figure 4.5 Construction of the project

**Program listing**: The program listing (**FlashLED**) is shown in Figure 4.6. The program was built in workspace **FlashLED**. At the beginning of the program, the header files of the modules used are included.

```
/*-----------------------------------------------------------------
                        FLASH EXTERNAL LED

This program flashes an external LED connecte to port pin P2_7
every 250ms

Author: Dogan Ibrahim
Date  : March 2021
Work  : FlashLED
------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"

/***** Main program *****/
int main(void)
{
        mxc_gpio_cfg_t gpio_out;

/* Setup output pin P2_7 */
    gpio_out.port = MXC_GPIO2;                         // Port 2
    gpio_out.mask = MXC_GPIO_PIN_7;                    // Pin 7
    gpio_out.pad = MXC_GPIO_PAD_NONE;                  // None
    gpio_out.func = MXC_GPIO_FUNC_OUT;                 // Output
    MXC_GPIO_Config(&gpio_out);

    while (1)                                          // Do forever
    {
        MXC_GPIO_OutSet(gpio_out.port, gpio_out.mask);  // LED OFF
        MXC_Delay(250000);                              // Wait 250ms
        MXC_GPIO_OutClr(gpio_out.port, gpio_out.mask);  // LED ON
        MXC_Delay(250000);                              // Wait 250ms
    }
}
```

Figure 4.6 Program listing

Port pin P2_7 (Port 2 pin 7) is configured as an output port using the structure mxc_gpio_cfg_t. The remainder of the program runs in an endless loop. Inside this loop, the LED is turned ON and OFF every 250000 microseconds (i.e. 250 ms). The port pin is set to logic 1 (i.e. LED OFF) with the function call:

```
MXC_GPIO_OutSet(gpio_out.port, gpio_out.mask);          // LED OFF
```

And to logic 0 (i.e. LED ON) with the function call:

```
MXC_GPIO_OutClr(gpio_out.port, gpio_out.mask);          // LED ON
```

## 4.3 • Project 2 – Alternately flashing LEDs (+1.8V output port voltage)

**Description**: In this project, two LEDs are connected to four GPIO ports. The LEDs are turned ON/OFF alternately

**Aim**: This project aims to show how more than one LED can be connected to the MAX78000FTHR.

**Block diagram**: The block diagram of the project is shown in Figure 4.7.

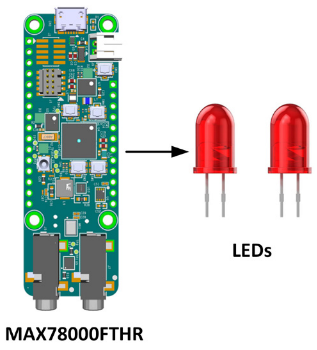


Figure 4.7 Block diagram of the project

**Circuit diagram**: The circuit diagram of the project is shown in Figure 4.8. The two LEDs are connected to the port pins P2_6 and P2_7 through 1K current limiting resistors in current sink mode.

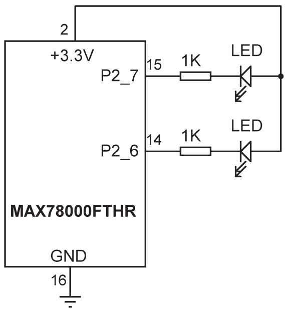


Figure 4.8 Circuit diagram of the project

**Construction**: Figure 4.9 shows the project constructed on a breadboard and connections made using jumper wires.

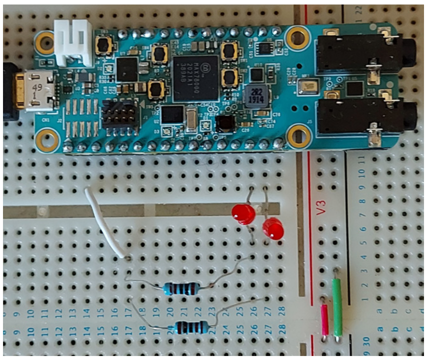


Figure 4.9 Construction of the project

**Program listing**: The program listing (**Flash2LED**) is shown in Figure 4.10. At the beginning of the program, the header files of the modules used are included. Pins P2_6 and P2_7 are then configured as outputs. The remainder of the program runs in a loop where the two LEDs are flashed alternately.

```
/*------------------------------------------------------------------
                FLASH EXTERNAL LED

This program flashes an external LED connected to port pin P2_7
every 250ms

Author: Dogan Ibrahim
Date  : March 2021
Work  : Flash2LED
--------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"

/***** Main program *****/
int main(void)
{
    mxc_gpio_cfg_t gpio_out7;
    mxc_gpio_cfg_t gpio_out6;

/* Setup output pin P2_7 */
    gpio_out7.port = MXC_GPIO2;                             // Port 2
    gpio_out7.mask = MXC_GPIO_PIN_7;                        // Pin 7
    gpio_out7.pad = MXC_GPIO_PAD_NONE;                      // None
    gpio_out7.func = MXC_GPIO_FUNC_OUT;                     // Output
    MXC_GPIO_Config(&gpio_out7);

/* Setup output pin P2_6 */
    gpio_out6.port = MXC_GPIO2;                             // Port 2
    gpio_out6.mask = MXC_GPIO_PIN_6;                        // Pin 6
    gpio_out6.pad = MXC_GPIO_PAD_NONE;                      // None
    gpio_out6.func = MXC_GPIO_FUNC_OUT;                     // Output
    MXC_GPIO_Config(&gpio_out6);

    while (1)                                               // Do forever
    {
        MXC_GPIO_OutSet(gpio_out6.port, gpio_out6.mask);    // LED at P2_6 OFF
        MXC_GPIO_OutClr(gpio_out7.port, gpio_out7.mask);    // LED at P2_7 ON
        MXC_Delay(250000);                                  // Wait 250ms
        MXC_GPIO_OutSet(gpio_out7.port, gpio_out7.mask);    // LED at P2_7 OFF
        MXC_GPIO_OutClr(gpio_out6.port, gpio_out6.mask);    // LED at P2_6 ON
        MXC_Delay(250000);                                  // Wait 250ms
    }
}
```

Figure 4.10 Program listing

## 4.4 ● Project 3 – Alternately flashing LEDs (+3.3V output port voltage)

**Description**: This project is similar to the previous one where two LEDs are used and flashed alternately. In this program, the port output voltage is set to +3.3V and the LEDs are connected in current source mode, i.e. an LED is turned ON when logic 1 is applied to it.

**Aim**: This project aims to show how the port pins can be set to +3.3V (rather than the default +1.8V).

**Block diagram**: The block diagram of the project is the same as in Figure 4.7.

**Circuit diagram**: Figure 4.11 shows the circuit diagram of the project. The two LEDs are connected to the MAX78000FTHR in current source mode through 1K current limiting resistors.

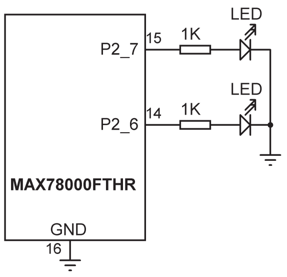


Figure 4.11 Circuit diagram of the project

**Program listing**: The program listing (**Flash3LED**) is shown in Figure 4.12. At the beginning of the program, the header files of the modules used are included. Pins P2_6 and P2_7 are then configured as outputs. Notice the two pins are logically Ored in the same structure. Port output voltage is set to +3.3V by the following function call:

```
gpio.out.vssel = MXC_GPIO_VSSEL_VDDIOH;           // Set port to +3.3V
```

To set the output port to +1.8V use the following option:

```
MXC_GPIO_VSSEL_VDDIO
```

The remainder of the program runs in a loop where the two LEDs are alternately flashed

```
/*-------------------------------------------------------------------
                FLASH EXTERNAL LED

This program flashes alternately 2 external LED connected to port
pins P2_6 and P2_7 every 250ms.

In this version, the LEDs are connected in current sourcing mode
and the port output is set to +3.3V

Author: Dogan Ibrahim
Date  : March 2021
Work  : Flash3LED
--------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"

/***** Main program *****/
int main(void)
{
        mxc_gpio_cfg_t gpio_out;

/* Setup output pins P2_6 and P2_7 */
    gpio_out.port = MXC_GPIO2;                                  // Port 2
    gpio_out.mask = MXC_GPIO_PIN_6 | MXC_GPIO_PIN_7;            // Pins 6, 7
    gpio_out.pad = MXC_GPIO_PAD_NONE;                           // None
    gpio_out.func = MXC_GPIO_FUNC_OUT;                          // Output
    gpio_out.vssel = MXC_GPIO_VSSEL_VDDIOH;                     // Set port to +3.3V
    MXC_GPIO_Config(&gpio_out);

    while (1)                                                   // Do forever
    {
        MXC_GPIO_OutSet(gpio_out.port, MXC_GPIO_PIN_6);         // LED at P2_6 ON
        MXC_GPIO_OutClr(gpio_out.port, MXC_GPIO_PIN_7);         // LED at P2_7 OFF
        MXC_Delay(250000);                                      // Wait 250ms
        MXC_GPIO_OutSet(gpio_out.port, MXC_GPIO_PIN_7);         // LED at P2_7 ON
        MXC_GPIO_OutClr(gpio_out.port, MXC_GPIO_PIN_6);         // LED at P2_6 OFF
        MXC_Delay(250000);                                      // Wait 250ms
    }
}
```

Figure 4.12 Program listing

From now onwards, we will be setting the output port logic HIGH voltages to +3.3V in the remaining projects where the GPIO ports are used.

The absolute maximum current ratings of the MAX78000FTHR GPIO ports are as follows (these limits must not be exceeded, otherwise the chip may be damaged):

| | |
|---|---|
| Output sink current by any GPIO pin: | 25 mA |
| Output source current by any GPIO pin: | -25 mA |
| Combined pins (sink with LOW output): | 100 mA |
| Combined pins (sink with HIGH outputs): | 100 mA |

## 4.5 ● Project 4 – Rotating LEDs – same port pins

**Description**: In this project, four LEDs are connected to four pins of the same GPIO port. The LEDs are turned ON/OFF in a rotating manner where only one LED is ON at any given time. i.e., the required LED pattern is as shown in Figure 4.13.

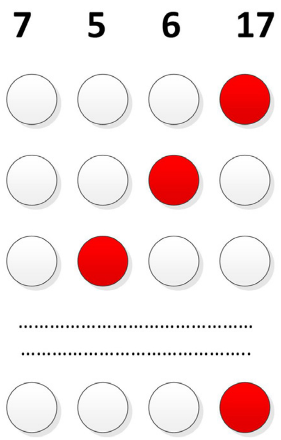


Figure 4.13 Rotating LEDs

**Aim**: This project aims to show how an array and a for loop can be used in a program to control multiple devices (LEDs) connected to the GPIO ports.

**Block diagram**: The block diagram of the project is shown in Figure 4.14.

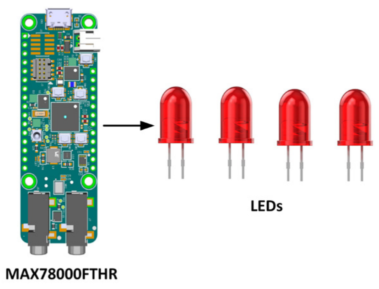


Figure 4.14 Block diagram of the project

**Circuit diagram**: The circuit diagram of the project is shown in Figure 4.15. The LEDs are connected to the port pins P0_7, P0_5, P0_6, and P0_17 through 1K current limiting resistors. The LEDs are connected to Port 0 pins. All the LEDs are connected in current source mode.

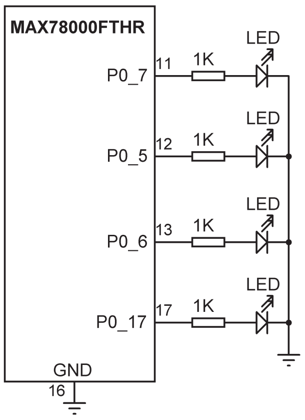


Figure 4.15 Circuit diagram of the project

**Program listing**: Figure 4.16 shows the program listing (**RotateLED**). At the beginning of the program, the LEDs are assigned to Port 0 pins:

```
#define led0 MXC_GPIO_PIN_7
#define led1 MXC_GPIO_PIN_5
#define led2 MXC_GPIO_PIN_6
#define led3 MXC_GPIO_PIN_17
```

An array called LEDS is created which stores the LED assignments:

```
int LEDS[] = {led0, led1, led2, led3};
```

The LEDs are configured as outputs. The remaining parts of the program run in a **while** loop. Inside this loop, a **for** loop is formed which iterates 4 times and turns the LEDs ON and OFF to give the rotating effect. Notice in this program all the used port pins are from the same port (Port 0).

```
/*------------------------------------------------------------------
                ROTATE LEDs

In this program 4 LEDs are connected to PORT 0. The program turns
the LEDs ON/OFF in a rotating manner as descibed in the text
Author: Dogan Ibrahim
Date  : March 2021
Work  : RotateLED
-------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"

#define led0 MXC_GPIO_PIN_7
#define led1 MXC_GPIO_PIN_5
#define led2 MXC_GPIO_PIN_6
#define led3 MXC_GPIO_PIN_17
int LEDS[] = {led0, led1, led2, led3};

/***** Main program *****/
int main(void)
{
        int j;
        mxc_gpio_cfg_t gpio_out;

/* Setup output pins P0_7,P0_5,P0_6,P0_17 */
    gpio_out.port = MXC_GPIO0;                              // Port 2
    gpio_out.mask = led0 | led1 | led2 | led3;              // Pins 6, 7
    gpio_out.pad = MXC_GPIO_PAD_NONE;                       // None
    gpio_out.func = MXC_GPIO_FUNC_OUT;                      // Output
    gpio_out.vssel = MXC_GPIO_VSSEL_VDDIOH;                 // Set port to +3.3V
    MXC_GPIO_Config(&gpio_out);

    while (1)                                               // Do forever
    {
        for(j = 0; j < 4; j++)                              // Do for 4 LEDs
        {
                MXC_GPIO_OutSet(gpio_out.port, LEDS[j]);    // LED[j] ON
                MXC_Delay(250000);                          // Wait 250ms
                MXC_GPIO_OutClr(gpio_out.port, LEDS[j]);    // LED[j] OFF
                MXC_Delay(250000);                          // Wait 250ms
        }
    }
}
```

Figure 4.16 Program listing

## 4.6 • Project 5 – Rotating LEDs – different port pins

**Description**: This project is similar to the previous one, but the LEDs are not all connected to the same port. The LEDs are turned ON/OFF in a rotating manner as in the previous project where only one LED is ON at any given time. i.e. the required LED pattern is as shown in Figure 4.13.

**Block diagram**: The block diagram of the project is as in Figure 4.14.

**Circuit diagram**: The circuit diagram of the project is shown in Figure 4.17. The LEDs are connected to the port pins P0_5, P0_6, P2_6, and P2_7 through 1K current limiting resistors in current source mode.

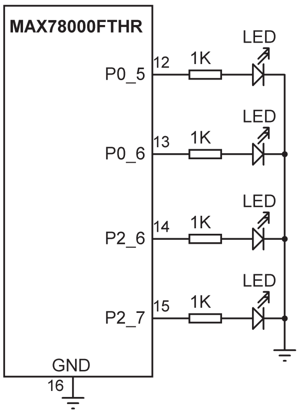


Figure 4.17 Circuit diagram of the project

Program listing: Figure 4.18 shows the program listing (Rotate2LED). An array called PORTS is created to store the port numbers. Corresponding pin numbers are stored in array LEDS:

```
#define led0 MXC_GPIO_PIN_7
#define led1 MXC_GPIO_PIN_5
#define led2 MXC_GPIO_PIN_6
#define led3 MXC_GPIO_PIN_17

int LEDS[] = {led0, led1, led2, led3};      // 0,1 on P0, 2,3 on P2
int PORTS[] = {0, 0, 2, 2};                 // Ports 0, 0, 2, 2
```

Port 0 and Port 2 are then configured. The remaining parts of the program run in a **while** loop. Inside this loop, a **for** loop is formed which iterates 4 times. The port number is checked and if it is 0, **gpio0_out** is used, otherwise **gpio2_out**.

```
/*----------------------------------------------------------------------
                        ROTATE LEDs

In this program 4 LEDs are connected to PORT 0 and Port 2. The program
turns the LEDs ON/OFF in a rotating manner as descibed in the text.

In this version of the program different ports are used

Author: Dogan Ibrahim
Date  : March 2021
Work  : Rotate2LED
-----------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"

#define led0 MXC_GPIO_PIN_5                     // P0_5
#define led1 MXC_GPIO_PIN_6                     // P0_6
#define led2 MXC_GPIO_PIN_6                     // P2_6
#define led3 MXC_GPIO_PIN_7                     // P2_7
int LEDS[] = {led0, led1, led2, led3};          // 0,1 on P0, 2,3 on P2
int PORTS[] = {0, 0, 2, 2};                     // Ports 0,0,2,2

/***** Main program *****/
int main(void)
{
        int j;
        mxc_gpio_cfg_t gpio0_out;
        mxc_gpio_cfg_t gpio2_out;

/* Setup output pins P0_5,P0_6 */
    gpio0_out.port = MXC_GPIO0;                 // Port 0
    gpio0_out.mask = led0 | led1;               // Pins 5, 6
    gpio0_out.pad = MXC_GPIO_PAD_NONE;          // None
    gpio0_out.func = MXC_GPIO_FUNC_OUT;         // Output
    gpio0_out.vssel = MXC_GPIO_VSSEL_VDDIOH;    // Set port to +3.3V
    MXC_GPIO_Config(&gpio0_out);

/* Setup output pins P2_6,P2_7 */
    gpio2_out.port = MXC_GPIO2;                 // Port 2
    gpio2_out.mask = led2 | led3;               // Pins 6, 7
    gpio2_out.pad = MXC_GPIO_PAD_NONE;          // None
    gpio2_out.func = MXC_GPIO_FUNC_OUT;         // Output
    gpio2_out.vssel = MXC_GPIO_VSSEL_VDDIOH;    // Set port to +3.3V
    MXC_GPIO_Config(&gpio2_out);

    while (1)                                                   // Do forever
    {
        for(j = 0; j < 4; j++)                                  // Do for 4 LEDs
        {
                if(PORTS[j] == 0)                                       // If PORT 0
                        MXC_GPIO_OutSet(gpio0_out.port, LEDS[j]);       // LED[j] ON
                else if(PORTS[j] == 2)                                  // If Port 2
                        MXC_GPIO_OutSet(gpio2_out.port, LEDS[j]);
                MXC_Delay(250000);                              // Wait 250ms

                if(PORTS[j] == 0)                                       // If Port 0
                        MXC_GPIO_OutClr(gpio0_out.port, LEDS[j]);       // LED[j] ON
                else if(PORTS[j] == 2)                                  // If Port 2
                        MXC_GPIO_OutClr(gpio2_out.port, LEDS[j]);
                MXC_Delay(250000);                              // Wait 250ms
        }
    }
```

Figure 4.18 Program listing

## 4.7 ● Project 6 – Binary up counter with LEDs

**Description**: In this project 4 LEDs are connected to Port 0 pins P0_7, P0_5, P0_6, P0_17 as in Project 4. The program counts up from 0 to 15 continuously and turns ON/OFF the appropriate LEDs to show the count. A one second delay is inserted between each count. Figure 4.19 shows the count with the LEDs.

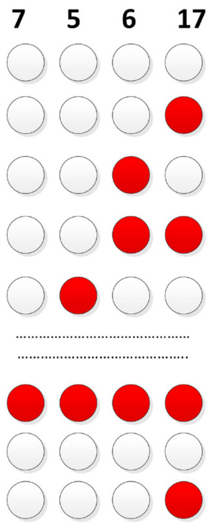


Figure 4.19 Counting up in binary

**Aim**: This project aims to show how several port pins can be grouped together and then accessed as a group. Here, 4 pins are grouped and accessed as a single nibble. P0_7 is assumed to be the LSB bit and P0_17 is assumed to be the MSB bit.

The block diagram and circuit diagram of the project are as in Figure 4.14 and Figure 4.15 respectively.

**Construction**: The project is built on a breadboard as shown in Figure 4.20.

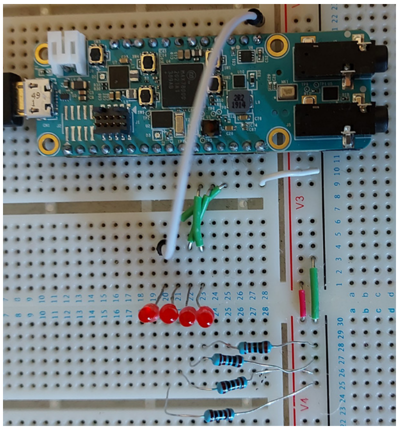


Figure 4.20 Project built on a breadboard

**Program listing**: Figure 4.21 shows the program listing (**BinCount**). At the beginning of the program, the connections to the LEDs are defined. P0_17 is configured as the LSB and P0_7 the MSB. Function Display has two arguments: No and L. No is the number to be displayed on the LEDs, and L is the LED count (4 in this example). This function turns ON/OFF the LED corresponding to No. For example, if No = 1 then the LED at P0_17 is turned ON, and so on. The main program runs in a loop where variable Count is incremented by one and function Display is called to display the number. A one second delay is used between each output.

```
/*----------------------------------------------------------------------
                           LED BINARY COUNTER

In this program 4 LEDs are connected to PORT 0 pins. The program
counts up every second and displays the count on the LEDs

Author: Dogan Ibrahim
Date  : March 2021
Work  : BinCount
-----------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
```

```
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include "math.h"

#define led0 MXC_GPIO_PIN_7                                 // P0_7 (MSB)
#define led1 MXC_GPIO_PIN_5                                 // P0_5
#define led2 MXC_GPIO_PIN_6                                 // P0_6
#define led3 MXC_GPIO_PIN_17                                // P0_17 (LSB)
int LEDS[] = {led0, led1, led2, led3};

mxc_gpio_cfg_t gpio_out;



//
// Group the port pins together. L is the number of bits (4 here)
// and No is the data to be displayed
//
void Display(int No, int L)
{
        int j, m, i;
        m = L - 1;
        for(i = 0; i < L; i++)
        {
                j = pow(2, m);
                if((No & j) != 0)
                        MXC_GPIO_OutSet(gpio_out.port, LEDS[i]);
                else
                        MXC_GPIO_OutClr(gpio_out.port, LEDS[i]);
                m--;
        }
}



/***** Main program *****/
int main(void)
{
        int Count = 0;

        /* Setup output pins P0_7,P0_5,P0_6,P0_17 */
        gpio_out.port = MXC_GPIO0;
// Port 0
        gpio_out.mask = led0 | led1 | led2 | led3;      // Pins 7,5,6,17
        gpio_out.pad = MXC_GPIO_PAD_NONE;               // None
        gpio_out.func = MXC_GPIO_FUNC_OUT;              // Output
        gpio_out.vssel = MXC_GPIO_VSSEL_VDDIOH;         // Set port to +3.3V
        MXC_GPIO_Config(&gpio_out);


    while (1)                                               // Do forever
    {
        Display(Count, 4);                                  // Display Count
        if(Count == 15)                                     // If 15
            Count = 0;                                      // Reset to 0
        else                                                // If not 15
            Count++;                                        // Increment Count
        MXC_Delay(1000000);                                 // Wait 1 second
    }
}
```

Figure 4.21 Program listing

**Suggestion for more work**: As an exercise, try to extend the LED count to 8.

## 4.8 ● Project 7 – Random flashing LEDs

**Description**: In this project, 4 LEDs are connected to the MAX78000FTHR development board as in the previous project. The program flashes the LEDs randomly so that they look like flashing Christmas lights.

**Aim**: This project aims to show how random numbers can be generated.

The block diagram and circuit diagram of the project are as in Figure 4.14 and Figure 4.15 respectively.

**Program listing**: Figure 4.22 shows the program listing (**RandomLED**). The program is very similar to Figure 4.21, except that here an integer random number is generated using the built-in function rand. This function generates a random number between 0 and 32767 and modulo 16 is used to configure the number to be between 0 and 15. The process is repeated every 250ms.

```
/*----------------------------------------------------------------------
                RANDOM FLASHING LEDs

In this program 4 LEDs are connected to PORT 0 pins. The program
turns ON/OFF the LEDs randomly every 250ms

Author: Dogan Ibrahim
Date  : March 2021
Work  : RandomLED
----------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
```

```
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include "math.h"

#define led0 MXC_GPIO_PIN_7                                  // P0_7 (MSB)
#define led1 MXC_GPIO_PIN_5                                  // P0_5
#define led2 MXC_GPIO_PIN_6                                  // P0_6
#define led3 MXC_GPIO_PIN_17                                 // P0_17 (LSB)
int LEDS[] = {led0, led1, led2, led3};
int rand();

mxc_gpio_cfg_t gpio_out;

//
// Group the port pins together. L is the number of bits (4 here)
// and No is the data to be displayed
//
void Display(int No, int L)
{
        int j, m, i;
        m = L - 1;
        for(i = 0; i < L; i++)
        {
                j = pow(2, m);
                if((No & j) != 0)
                        MXC_GPIO_OutSet(gpio_out.port, LEDS[i]);
                else
                        MXC_GPIO_OutClr(gpio_out.port, LEDS[i]);
                m--;
        }
}


/***** Main program *****/
int main(void)
{
        int r;

        /* Setup output pins P0_7,P0_5,P0_6,P0_17 */
        gpio_out.port = MXC_GPIO0;                           // Port 0
        gpio_out.mask = led0 | led1 | led2 | led3;           // Pins 7,5,6,17
        gpio_out.pad = MXC_GPIO_PAD_NONE;                    // None
        gpio_out.func = MXC_GPIO_FUNC_OUT;                   // Output
        gpio_out.vssel = MXC_GPIO_VSSEL_VDDIOH;              // Set port to +3.3V
        MXC_GPIO_Config(&gpio_out);
```

```
    while (1)                                                       // Do forever
    {
        r = rand() % 16;                                            // Random number
0-15
        Display(r, 4);                                              // Display
        MXC_Delay(250000);                                          // Wait 250ms
    }
}
```

Figure 4.22 Program listing

## 4.9 • Project 8 – Push button and LED – using an on-board button

**Description**: In this project, an onboard button is used to toggle an external LED.

**Aim**: This project aims to show how the state of a button can be read in a program.

**Circuit diagram**: Figure 4.23 shows the circuit diagram of the project. An LED is connected to port pin P0_7 through a 1K current limiting resistor. There are two user-programmable buttons on the MAX78000FTHR development board. SW1 is connected to P0_2 and SW2 is connected to P1_7. A pin of the buttons is connected to GND, while the other pins are connected to a MAX6817EUT+ type dual contact debouncing chip. The output states of the buttons are normally floating, but they can be pulled up in software so that their output becomes normally logic 1 and go to logic 0 when pressed. In this project, the onboard button at pin P0_2 (SW1) is used.

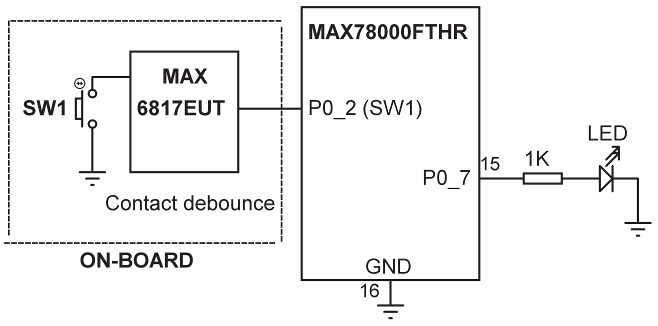


Figure 4.23 Circuit diagram of the project

**Program listing**: Figure 4.24 shows the program listing (**ButtonLED**). At the beginning of the program, input and output pins are configured. Input pin P0_2 is pulled up using option MXC_GPIO_PAD_PULL_UP. The program loop runs in a while loop. Inside this loop, the state of the button is checked. If the button is pressed, the program waits until it is released. As soon as the button is released the state of the LED is toggled (i.e. if it is ON it turns OFF, and if it is OFF it turns ON).

```
/*----------------------------------------------------------------------
                PUSHBUTTON TO TOGGLE AN LED

In this program an external LED is connected to P0_7. The LED state
is toggled by pressing the on-board button SW1

Author: Dogan Ibrahim
Date  : March 2021
Work  : ButtonLED
-----------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"

#define LED MXC_GPIO_PIN_7                                   // LED at P0_7
#define Button MXC_GPIO_PIN_2                                // Button at P0_2

mxc_gpio_cfg_t gpio_out;
mxc_gpio_cfg_t gpio_in;

/***** Main program *****/
int main(void)
{
        /* Setup output pins P0_7 */
        gpio_out.port = MXC_GPIO0;                           // Port 0
        gpio_out.mask = LED;                                 // Pins 7
        gpio_out.pad = MXC_GPIO_PAD_NONE;                    // None
        gpio_out.func = MXC_GPIO_FUNC_OUT;                   // Output
        gpio_out.vssel = MXC_GPIO_VSSEL_VDDIOH;              // Set port to +3.3V
        MXC_GPIO_Config(&gpio_out);

        /* Setup input pin at P0_2 (SW1) */
        gpio_in.port = MXC_GPIO0;
        gpio_in.mask = Button;
        gpio_in.pad = MXC_GPIO_PAD_PULL_UP;
        gpio_in.func = MXC_GPIO_FUNC_IN;
        MXC_GPIO_Config(&gpio_in);

//
// Detect when button SW1 is pressed. Wait until the button is released and then
// toggle the LED
//
    while (1)                                                // Do forever
    {
        if(MXC_GPIO_InGet(gpio_in.port, Button) == 0)
        {
                while(MXC_GPIO_InGet(gpio_in.port, Button) == 0);
                MXC_GPIO_OutToggle(gpio_out.port, LED);
        }
    }
}
```

Figure 4.24 Program listing

An output port pin is toggled using the **MXC_GPIO_OutToggle** function as in the following example:

```
        MXC_GPIO_OutToggle(gpio_out.port, LED);
```

**pad** supports the following options:

| | |
|---|---|
| MXC_GPIO_PAD_NONE | /* No pull-up or pull-down */ |
| MXC_GPIO_PAD_PULL_UP | /* Set pad to strong pull-up */ |
| MXC_GPIO_PAD_PULL_DOWN | /* Set pad to strong pull-down */ |
| MXC_GPIO_PAD_WEAK_PULL_UP | /* Set pad to weak pull-up */ |
| MXC_GPIO_PAD_WEAK_PULL_DOWN | /* Set pad to weak pull-down */ |

**func** supports the following options:

| | |
|---|---|
| MXC_GPIO_FUNC_IN | /* GPIO Input */ |
| MXC_GPIO_FUNC_OUT | /* GPIO Output */ |
| MXC_GPIO_FUNC_ALT1 | /* Alternate Function Selection */ |
| MXC_GPIO_FUNC_ALT2 | /* Alternate Function Selection */ |
| MXC_GPIO_FUNC_ALT3 | /* Alternate Function Selection */ |
| MXC_GPIO_FUNC_ALT4 | /* Alternate Function Selection */ |

## 4.10 ● Project 9 – Two pushbuttons and two LEDs – using external buttons

**Description**: In this project, two external buttons (Button1 and Button2) and LEDs (LED1 and LED2) are used. Pressing Button1 toggles LED1, and pressing Button2 toggles LED2.

**Aim**: This project aims to show how external buttons can be connected to the MAX78000FTHR development board.

**Block diagram**: Figure 4.25 shows the block diagram of the project.

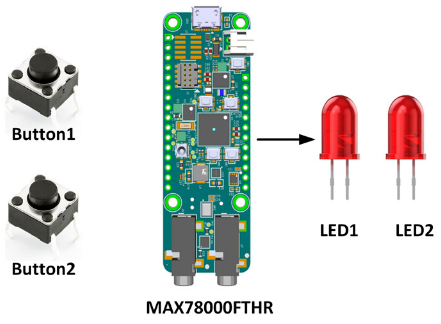


Figure 4.25 Block diagram of the project

**Circuit diagram**: In general, buttons can be connected in two different ways: normally HIGH, and LOW. Figure 4.26 shows normally HIGH mode where the output state of the button is at logic 1 and goes to logic 0 when the button is pressed.

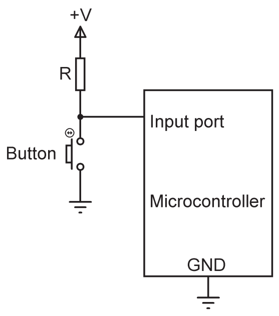


Figure 4.26 Normally HIGH button

Figure 4.27 shows the normally LOW mode where the output state of the button is at logic 0 and goes to logic 1 when the button is pressed.

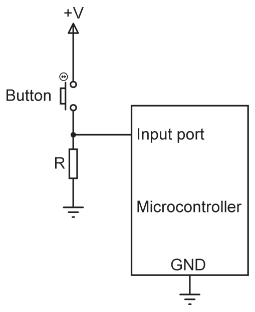


Figure 4.27 Normally LOW button

In this project, switch states are pulled HIGH by software and there is no need to use resistors. Figure 4.28 shows the circuit diagram of the project. P2_6 and P2_7 are used as the **Button1** and **Button2** buttons respectively. LED1 and LED2 are connected to P0_6 and P0_5 respectively.

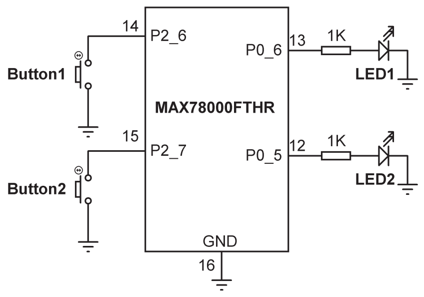


Figure 4.28 Circuit diagram of the project

**Program listing**: Figure 4.29 shows the program listing (**ButtonsLED**). At the beginning of the program, the input and output pins are configured. Input pins P2_6 and P2_7 of PORT 2 are pulled up. The program loop runs in a while loop. Inside this loop, the state of the two buttons is checked. If Button1 is pressed, LED1 is toggled. Similarly, if Button2 is pressed, LED2 is toggled.

```
/*----------------------------------------------------------------------
                USING TWO BUTTONS AND TWO LEDs

In this program an two external buttons named Buton1 and Button2 are
connected to the MAX78000FTHR. Also, two LEDs, LED1 and LED2 are
connected. Pressing Button1 toggles LED1, pressing Button2 toggles LED2

Author: Dogan Ibrahim
Date  : March 2021
Work  : ButtonsLED
----------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"

#define LED1 MXC_GPIO_PIN_6                                  // LED1 at P0_6
#define LED2 MXC_GPIO_PIN_5                                  // LED2 at P0_5
#define Button1 MXC_GPIO_PIN_6                               // Button1 at P2_6
#define Button2 MXC_GPIO_PIN_7                               // Button2 at P2_7

mxc_gpio_cfg_t gpio_out;
mxc_gpio_cfg_t gpio_in;

/***** Main program *****/
int main(void)
{
        /* Setup output pins P0_6 */
        gpio_out.port = MXC_GPIO0;                      // Port 0
        gpio_out.mask = LED1 | LED2;                    // Pin 6
        gpio_out.pad = MXC_GPIO_PAD_NONE;               // None
        gpio_out.func = MXC_GPIO_FUNC_OUT;              // Output
        gpio_out.vssel = MXC_GPIO_VSSEL_VDDIOH;         // Set port to +3.3V
        MXC_GPIO_Config(&gpio_out);

        /* Setup input pins at P2_6 and P2_7 */
        gpio_in.port = MXC_GPIO2;
        gpio_in.mask = Button1 | Button2;
        gpio_in.pad = MXC_GPIO_PAD_PULL_UP;
        gpio_in.func = MXC_GPIO_FUNC_IN;
        MXC_GPIO_Config(&gpio_in);

//
// Detect when buttons are pressed and change the LED flashing rate
//
    while (1)                                           // Do forever
    {
        if(MXC_GPIO_InGet(gpio_in.port, Button1) == 0)
        {
                while(MXC_GPIO_InGet(gpio_in.port, Button1) == 0);
                MXC_GPIO_OutToggle(gpio_out.port, LED1);
        }
        else if(MXC_GPIO_InGet(gpio_in.port, Button2) == 0)
        {
                while(MXC_GPIO_InGet(gpio_in.port, Button2) == 0);
                MXC_GPIO_OutToggle(gpio_out.port, LED2);
        }
    }
}
```

Figure 4.29 Program listing

It may be necessary to use contact debouncing circuits or chips (e.g. MAX6817EUT+) at the outputs of the buttons to eliminate the contact bouncing problems associated with mechanical switches.

Figure 4.30 shows the project built on a breadboard.

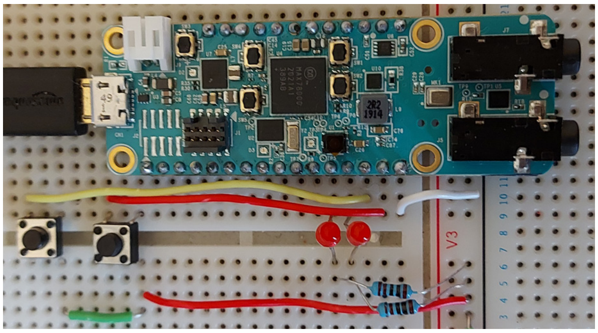


Figure 4.30 Project built on a breadboard

## 4.11 • Project 10 – Using an external button – external interrupts

**Description**: In this project, an external button and LED are connected to the project. Pressing the button generates an interrupt. The state of the LED is toggled inside the interrupt service routine.

**Aim**: This project aims to show how external interrupts can be generated and serviced.

**Circuit diagram**: Figure 4.31 shows the circuit diagram of the project. The button is connected to P2_6, and the LED is connected to P0_6.

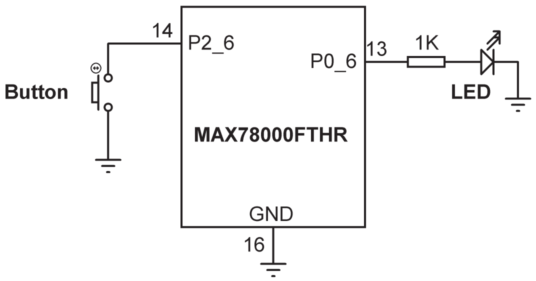


Figure 4.31 Circuit diagram of the project

**Program listing**: Figure 4.32 shows the program listing (**LEDInt**). At the beginning of the program LED and Button connections are defined. Inside the main program, P0_6 is configured as an output, and P2_6 as input. Pin P2_6 is then configured as an external interrupt pin such that interrupts will be recognised on the FALLING edge of P2_6. The call-back function is called **gpio_ISR** and no data is passed to the interrupt service routine. Interrupts are enabled and the main program with in a loop until an interrupt is generated. When the button is pressed an external interrupt is generated and the program jumps to function **gpio_ISR**. Inside this function, the state of the LED is toggled.

The following functions are called to configure and enable external interrupts.

```
        MXC_GPIO_RegisterCallback(&gpio_interrupt, gpio_ISR, 0);
        MXC_GPIO_IntConfig(&gpio_interrupt, MXC_GPIO_INT_FALLING);
        MXC_GPIO_EnableInt(gpio_interrupt.port, Button);
        NVIC_EnableIRQ(MXC_GPIO_GET_IRQ(MXC_GPIO_GET_IDX(MXC_GPIO2)));

/*---------------------------------------------------------------------
                EXTERNAL INTERRUPTS

In this program an external button is connected to MAX78000FTHR. Also,
an external LED is connected. Pressing the button generates an interrupt
where inside the ISR the state of the LED is toggled

Author: Dogan Ibrahim
Date  : March 2021
Work  : LEDInt
----------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"

#define LED MXC_GPIO_PIN_6                                      // LED at P0_6
#define Button MXC_GPIO_PIN_6                                   // Button at P2_6

mxc_gpio_cfg_t gpio_out;
mxc_gpio_cfg_t gpio_interrupt;

//
// Interrupt service routine. Toggle the state of the LED
//
void gpio_ISR(void* dummy)
{
        MXC_GPIO_OutToggle(gpio_out.port, LED);
}



/***** Main program *****/
int main(void)
{
        /* Setup output pin P0_6 (LED) */
        gpio_out.port = MXC_GPIO0;                      // Port 0
        gpio_out.mask = LED;                            // Pin 6
        gpio_out.pad = MXC_GPIO_PAD_NONE;               // None
        gpio_out.func = MXC_GPIO_FUNC_OUT;              // Output
        gpio_out.vssel = MXC_GPIO_VSSEL_VDDIOH;         // Set port to +3.3V
        MXC_GPIO_Config(&gpio_out);

        /* Setup external interrupt on pin P2_6 (Button) */
        gpio_interrupt.port = MXC_GPIO2;
        gpio_interrupt.mask = Button;
        gpio_interrupt.pad = MXC_GPIO_PAD_PULL_UP;
        gpio_interrupt.func = MXC_GPIO_FUNC_IN;
        MXC_GPIO_Config(&gpio_interrupt);

//
// Interrupts will be configured at FALLING edge of P2_6
//
        MXC_GPIO_RegisterCallback(&gpio_interrupt, gpio_ISR,0);
        MXC_GPIO_IntConfig(&gpio_interrupt, MXC_GPIO_INT_FALLING);
        MXC_GPIO_EnableInt(gpio_interrupt.port, Button);
        NVIC_EnableIRQ(MXC_GPIO_GET_IRQ(MXC_GPIO_GET_IDX(MXC_GPIO2)));

//
// The main programs does nothing. It waits for interrupts to occur
//
    while (1)
    {
    }
}
```

Figure 4.32 Program listing

The following interrupt polarity modes are supported:

```
MXC_GPIO_INT_FALLING          /* Interrupt triggers on falling edge */
MXC_GPIO_INT_HIGH             /* Interrupt triggers when level is high*/
MXC_GPIO_INT_RISING           /* Interrupt triggers on rising edge */
MXC_GPIO_INT_LOW              /* Interrupt triggers when level is low */
MXC_GPIO_INT_BOTH             /* Interrupt triggers on either edge */
```

Interrupts can be disabled by the function call:

```
        MXC_GPIO_DisableInt(mxc_gpio_port, mask);
```

## 4.12 ● Using LCDs

In microcontroller-based systems, we usually want to interact with the system for example to enter a parameter, change the value of a parameter, or to display the output of a measured variable. Data is usually entered into a system using a switch, small keypad, or full-blown keyboard. Data is usually displayed using an indicator such as one or more LEDs, 7-segment, or LCD type displays. LCDs have the advantage that they can display alphanumeric as well as graphical data. Some LCDs have 40 or more character lengths with the capability to display data in several lines. Other LCDs can be used to display graphical images (Graphical LCDs, or simply GLCDs), such as animation. Some displays are in single or multi-colour, while others incorporate backlighting so they can be viewed in dimly lit conditions.

LCDs can be connected to a microcontroller either in parallel or through the I2C interface. Parallel LCDs (e.g. Hitachi HD44780) are connected using more than one data line and several control lines and data is transferred in parallel. It is common to use either 4 or 8 data lines and two or more control lines. Using a 4-wire connection saves I/O pins, but is slower since the data is transferred in two stages. I2C based LCDs on the other hand are connected to a microcontroller using only 2 wires, the data, and the clock. I2C based LCDs are in general much easier to use and require less wiring, but cost more than parallel. In this chapter, we will learn to use both parallel and I2C based LCDs in projects.

The programming of LCDs is a complex task, requiring a good understanding of the internal operations of the LCD controllers, including knowledge of their exact timing requirements. Fortunately, several libraries can be used to simplify the use of both parallel and serial LCDs.

**HD44780 LCD module**

Although there are several types of LCDs, the HD44780 is currently one of the most popular modules used in industry and hobbyists (Figure 4.33). This module is an alphanumeric monochrome display and comes in different sizes. Modules with 16 columns are popular in most small applications, but other modules with 8, 20, 24, 32, or 40 columns are also available. Although most LCDs have two lines (or rows) as standard, it is possible to purchase models with 1 or 4. LCDs are available with standard 14-pin connectors, although 16-pin modules are also available, providing terminals for backlighting. Table 4.1 gives the pin configuration and functions of a 16-pin LCD module. A summary of the pin functions is given below:

| Pin no | Name | Function |
|---|---|---|
| 1 | VSS | Ground |
| 2 | VDD | + ve supply |
| 3 | VEE | Contrast |
| 4 | RS | Register select |
| 5 | R/W | Read/write |
| 6 | E | Enable |
| 7 | D0 | Daat bit 0 |
| 8 | D1 | Data bit 1 |
| 9 | D2 | Data bit 2 |
| 10 | D3 | Data bit 3 |
| 11 | D4 | Data bit 4 |
| 12 | D5 | Data bit 5 |
| 13 | D6 | Data bit 6 |
| 14 | D7 | Data bit 7 |
| 15 | A | Backlight anode (+) |
| 16 | K | Backlight cathode (GND) |

Figure 4.33 HD44780 compatible parallel LCD

$V_{SS}$ (pin 1) and $V_{DD}$ (pin 2) are the ground and power supply pins. The power supply should be +5V.

$V_{EE}$ is pin 3 and this is the contrast control pin used to adjust the contrast of the display. The arm of a 10K potentiometer is normally connected to this pin and the other two terminals of the potentiometer are connected to the ground and power supply pins. The contrast of the display is adjusted by rotating the potentiometer arm.

Pin 4 is the Register Select (RS) and when this pin is LOW, data transferred to the display is treated as commands. When RS is HIGH, character data can be transferred to and from the display.

Pin 5 is the Read/Write (R/W) line. This pin is pulled LOW to write commands or character data to the LCD module. When the pin is HIGH, character data or status information can be read from the module. The pin is normally connected permanently LOW so commands or character data can be sent to the LCD module.

Enable (E) is pin 6 which is used to initiate the transfer of commands or data between the LCD module and the microcontroller. When writing to the display, data is transferred only on the HIGH to LOW transition of this pin. When reading from the display, data becomes available after the LOW to HIGH transition of the enable pin and this data remains valid as long as the enable pin is at logic HIGH.

Pins 7 to 14 are the eight data bus lines (D0 to D7). Data can be transferred between the microcontroller and the LCD module using either a single 8-bit byte or two 4-bit nibbles. In the latter case, only the upper four data lines (D4 to D7) are used. 4-bit mode has the advantage that four fewer I/O lines are required to communicate with the LCD. 4-bit mode is on the other hand slower since the data is transferred in two stages. In this book, we will use the 4-bit interface only.

Pins 15 and 16 are for background brightness control. To enable the background brightness, a 220 Ohm resistor should be connected from pin 15 to +5V supply, and pin 16 should be connected to ground.

In 4-bit mode, the following pins of the LCD are used. The R/W line is permanently connected to ground. This mode uses 6 GPIO port pins of the microcontroller:

$V_{SS}$, $V_{DD}$, $V_{EE}$, E, R/S, D4, D5, D6, D7

In the next section, we will create an LCD library of functions that can be used to send data and text to standard HD44780 type character LCDs.

## 4.13 ● Project 11 – LCD seconds counter

**Description**: In this project, we will develop several functions that can be used to send data and text to 16 x 2 character LCDs. The program in this project counts up every second and displays the count on the LCD.

**Aim**: This project aims to develop a library of functions that can be used to control LCDs. These functions can be used in projects to send text and numbers to LCDs.

**Block diagram**: Figure 4.34 shows the block diagram of the project.

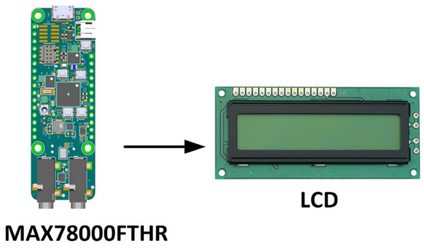


Figure 4.34 Block diagram of the project

Circuit diagram: The circuit diagram of the project is shown in Figure 4.35. The LCD is connected to the MAX78000FTHR using 4 data wires (D4 – D7) and 2 control wires (E and R/S). The connections between the LCD and development board are as follows:

| LCD pin | MAX78000FTHR pin |
|---|---|
| E | P0_6 |
| R/S | P0_17 |
| D4 | P0_8 |
| D5 | P0_11 |
| D6 | P0_19 |
| D7 | P0_16 |

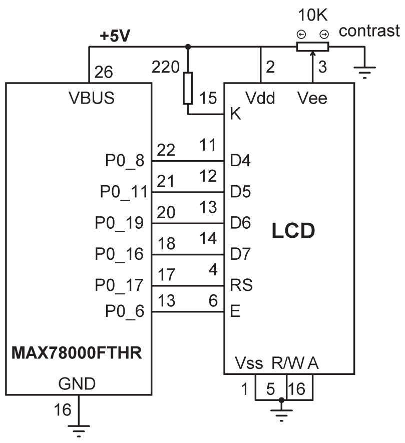


Figure 4.35 Circuit diagram of the project

The LCD requires +5V for its operation. This is obtained from the VBUS pin of the MAX78000FTHR. The contrast of the LCD is controlled using a 10K potentiometer.

**Program listing**: Figure 4.36 shows the functions listing (Program: **LCD**). The connections between the LCD and MAX78000FTHR are defined at the beginning and can be changed if desired. The remainder of the functions should not be changed for proper control of the LCD. These functions implement the initialisation and control of the LCD. Variable count is initialised to 0. It is then incremented every second and displayed on the second row of the LCD. The text **Counter** is displayed on the first row.

The following LCD control functions are available:

**lcd_init**: this is the LCD initialisation function and must be called first before any other functions are called

**lcd_clear**: clears the LCD

**lcd_home**: homes the cursor (top-left position)

**lcd_cursor_blink**: enables blinking cursor

**lcd_cursor_on**: enables visible cursor

**lcd_cursor_off**: disables visible cursor

**lcd_puts(s)**: displays string s

**lcd_putch(c)**: displays character c

**lcd_goto(col, row)**: positions the cursor at the specified column and position. (0, 0) is the left corner of the LCD. The first row is row 0. The second is row 1 and so on.

```
/*----------------------------------------------------------------------
                USING LCDs - LCD SECONDS COUNTER

In this program an LCD is connected to MAX78000FTHR. The program counts
up by one every second and displays the count on the LCD

Author: Dogan Ibrahim
Date  : March 2021
Work  : LCD
-----------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include "string.h"
#include "stdlib.h"

#define D4 MXC_GPIO_PIN_8                                   // D4 pin
#define D5 MXC_GPIO_PIN_11                                  // D5 pin
#define D6 MXC_GPIO_PIN_19                                  // D6 pin
#define D7 MXC_GPIO_PIN_16                                  // D7 pin
#define RS MXC_GPIO_PIN_17                                  // RS pin
#define E MXC_GPIO_PIN_6                                    // E pin
int DataPin[] = {D4, D5, D6, D7};

mxc_gpio_cfg_t gpio_out;

//======================================================================
// START OF LCD FUNCTIONS
//
void lcd_strobe()
{
        MXC_GPIO_OutSet(gpio_out.port, E);
        MXC_Delay(1000);
        MXC_GPIO_OutClr(gpio_out.port, E);
        MXC_Delay(1000);
}

void lcd_write(char c, int mode)
{
        int i,b;
        char d;

    d = c;
    d = d >> 4;
    for(i = 0; i < 4; i++)
    {
        b = d & 1;
        if(b == 0)
                MXC_GPIO_OutClr(gpio_out.port, DataPin[i]);
        else
                MXC_GPIO_OutSet(gpio_out.port, DataPin[i]);
        d = d >> 1;
    }
    if(mode == 1)
        MXC_GPIO_OutSet(gpio_out.port, RS);
    else
        MXC_GPIO_OutClr(gpio_out.port, RS);
    lcd_strobe();

    d = c;
    for(i = 0; i < 4; i++)
    {
        b = d & 1;
        if(b == 0)
                MXC_GPIO_OutClr(gpio_out.port, DataPin[i]);
        else
                MXC_GPIO_OutSet(gpio_out.port, DataPin[i]);
        d = d >> 1;
    }
    if(mode == 1)
        MXC_GPIO_OutSet(gpio_out.port, RS);
     else
        MXC_GPIO_OutClr(gpio_out.port, RS);
     lcd_strobe();
     MXC_Delay(1000);
     MXC_GPIO_OutSet(gpio_out.port, RS);
}

void lcd_clear()
```

```
{
    lcd_write(0x01, 0);
    MXC_Delay(10000);
}

void lcd_home()
{
    lcd_write(0x02, 0);
    MXC_Delay(5000);
}

void lcd_cursor_blink()
{
    lcd_write(0x0D, 0);
    MXC_Delay(1000);
}

void lcd_cursor_on()
{
    lcd_write(0x0E, 0);
    MXC_Delay(1000);
}

void lcd_cursor_off()
{
    lcd_write(0x0C, 0);
    MXC_Delay(1000);
}

void lcd_putch(char c)
{
    lcd_write(c, 1);
}

void lcd_puts(char *s)
{
        int i, l;
    l = strlen(s);
    for(i = 0; i < l; i++)
    {
       lcd_putch(*s);
       s++;
    }
}

void lcd_goto(int col, int row)
```

```
{
        int c, address;
    c = col + 1;
    if(row == 0)
        address = 0;
    if(row == 1)
        address = 0x40;
    address = address + c - 1;
    lcd_write(0x80 | address, 0);
}

void lcd_init()
{
        int i;
    MXC_Delay(120000);
    for(i = 0; i < 4; i++)
        MXC_GPIO_OutClr(gpio_out.port, DataPin[i]);
    MXC_Delay(50000);
    MXC_GPIO_OutSet(gpio_out.port, D4);
    MXC_GPIO_OutSet(gpio_out.port, D5);
    lcd_strobe();
    MXC_Delay(10000);
    lcd_strobe();
    MXC_Delay(10000);
    lcd_strobe();
    MXC_Delay(10000);
    MXC_GPIO_OutClr(gpio_out.port, D4);
    lcd_strobe();
    MXC_Delay(5000);
    lcd_write(0x28, 0);
    MXC_Delay(1000);
    lcd_write(0x08, 0);
    MXC_Delay(1000);
    lcd_write(0x01, 0);
    MXC_Delay(10000);
    lcd_write(0x06, 0);
    MXC_Delay(5000);
    lcd_write(0x0C, 0);
    MXC_Delay(10000);
}
//
// END OF LCD FUNCTIONS
//======================================================================

/***** Main program *****/
int main(void)
```

```
{
        int count = 0;
        char buff[10];

        /* Setup output pins */
        gpio_out.port = MXC_GPIO0;                                  // Port 0
        gpio_out.mask = D4 | D5 | D6 | D7 | RS | E;                 // D4,D5,D6,D7
        gpio_out.pad = MXC_GPIO_PAD_NONE;                           // None
        gpio_out.func = MXC_GPIO_FUNC_OUT;                          // Output
        gpio_out.vssel = MXC_GPIO_VSSEL_VDDIOH;                     // Set port to +3.3V
        MXC_GPIO_Config(&gpio_out);

        lcd_init();                                                 // Initialize LCD
        lcd_clear();                                                // Clear display
        lcd_puts("Counter");                                        // Display Counter

    while (1)                                                       // Do forever
    {
        lcd_goto(0, 1);                                             // Cursor at 0,1
        itoa(count, buff, 10);                                      // Convert to string
        lcd_puts(buff);                                             // Display count
        count++;                                                    // Increment count
        MXC_Delay(1000000);                                         // Wait 1 second
    }
}
```

Figure 4.36 Program LCD

You should press the Reset button on the MAX78000FTHR to start the program from the beginning.

## Storing the LCD functions in a library

We can easily combine all LCD functions in a library and then include the file at the beginning of our program. Perhaps the easiest option is to put all LCD functions in a .c file called **lcdfuncs.c** and save it in the working folder of the project (where the main program **main.c** program is). Figure 4.37 shows the program listing **lcdfuncs.c**.

```
//======================================================================
//                              LCD FUNCTIONS
//
//
#define D4 MXC_GPIO_PIN_8                                   // D4 pin
#define D5 MXC_GPIO_PIN_11                                  // D5 pin
#define D6 MXC_GPIO_PIN_19                                  // D6 pin
#define D7 MXC_GPIO_PIN_16                                  // D7 pin
#define RS MXC_GPIO_PIN_17                                  // RS pin
```

```
#define E MXC_GPIO_PIN_6                                // E pin

int DataPin[] = {D4, D5, D6, D7};

mxc_gpio_cfg_t gpio_out;

void lcd_write(char, int);
void lcd_clear();
void lcd_home();
void lcd_cursor_blink();
void lcd_cursor_on();
void lcd_cursor_off();
void lcd_putch(char);
void lcd_puts(char *);
void lcd_goto(int, int);
void lcd_init();

void Config()
{
        /* Setup output pins */
        gpio_out.port = MXC_GPIO0;                              // Port 0
        gpio_out.mask = D4 | D5 | D6 | D7 | RS | E;             // D4,D5,D6,D7
        gpio_out.pad = MXC_GPIO_PAD_NONE;                       // None
        gpio_out.func = MXC_GPIO_FUNC_OUT;                      // Output
        gpio_out.vssel = MXC_GPIO_VSSEL_VDDIOH;                 // Set port to +3.3V
        MXC_GPIO_Config(&gpio_out);
}

void lcd_strobe()
{
        MXC_GPIO_OutSet(gpio_out.port, E);
        MXC_Delay(1000);
        MXC_GPIO_OutClr(gpio_out.port, E);
        MXC_Delay(1000);
}

void lcd_write(char c, int mode)
{
        int i,b;
        char d;

    d = c;
    d = d >> 4;
    for(i = 0; i < 4; i++)
    {
        b = d & 1;
        if(b == 0)
                MXC_GPIO_OutClr(gpio_out.port, DataPin[i]);
        else
                MXC_GPIO_OutSet(gpio_out.port, DataPin[i]);
        d = d >> 1;
    }
    if(mode == 1)
        MXC_GPIO_OutSet(gpio_out.port, RS);
    else
        MXC_GPIO_OutClr(gpio_out.port, RS);
    lcd_strobe();

    d = c;
    for(i = 0; i < 4; i++)
    {
        b = d & 1;
        if(b == 0)
                MXC_GPIO_OutClr(gpio_out.port, DataPin[i]);
        else
                MXC_GPIO_OutSet(gpio_out.port, DataPin[i]);
        d = d >> 1;
    }
    if(mode == 1)
        MXC_GPIO_OutSet(gpio_out.port, RS);
     else
        MXC_GPIO_OutClr(gpio_out.port, RS);
     lcd_strobe();
     MXC_Delay(1000);
     MXC_GPIO_OutSet(gpio_out.port, RS);
}

void lcd_clear()
{
    lcd_write(0x01, 0);
    MXC_Delay(15000);
}

void lcd_home()
{
    lcd_write(0x02, 0);
    MXC_Delay(5000);
}

void lcd_cursor_blink()
{
    lcd_write(0x0D, 0);
    MXC_Delay(1000);
}

void lcd_cursor_on()
{
    lcd_write(0x0E, 0);
    MXC_Delay(1000);
}

void lcd_cursor_off()
{
    lcd_write(0x0C, 0);
    MXC_Delay(1000);
}

void lcd_putch(char c)
{
    lcd_write(c, 1);
    MXC_Delay(2000);
}

void lcd_puts(char *s)
{
    int i, l;
    l = strlen(s);
    for(i = 0; i < l; i++)
    {
       lcd_putch(*s);
       s++;
    }
}

void lcd_goto(int col, int row)
{
    int c, address;
    c = col + 1;
    if(row == 0)
        address = 0;
    if(row == 1)
        address = 0x40;
    address = address + c - 1;
    lcd_write(0x80 | address, 0);
}

void lcd_init()
{
    int i;
    Config();
    MXC_Delay(120000);
    for(i = 0; i < 4; i++)
        MXC_GPIO_OutClr(gpio_out.port, DataPin[i]);
    MXC_Delay(50000);
    MXC_GPIO_OutSet(gpio_out.port, D4);
    MXC_GPIO_OutSet(gpio_out.port, D5);
    lcd_strobe();
    MXC_Delay(10000);
    lcd_strobe();
    MXC_Delay(10000);
    lcd_strobe();
    MXC_Delay(10000);
    MXC_GPIO_OutClr(gpio_out.port, D4);
    lcd_strobe();
    MXC_Delay(5000);
    lcd_write(0x28, 0);
    MXC_Delay(1000);
    lcd_write(0x08, 0);
    MXC_Delay(1000);
    lcd_write(0x01, 0);
    MXC_Delay(10000);
    lcd_write(0x06, 0);
    MXC_Delay(5000);
    lcd_write(0x0C, 0);
    MXC_Delay(10000);
}
//
// END OF LCD FUNCTIONS
//=====================================================================
```

Figure 4.37 Program lcdfuncs.c

File lcdfuncs.c is included in the simplified main program (**LCD2**) as shown in Figure 4.38. Figure 4.39 shows the project built on a breadboard.

```
/*----------------------------------------------------------------------
                USING LCDs - LCD SECONDS COUNTER

In this program an LCD is connected to MAX78000FTHR. The program counts
up by one every second and displays the count on the LCD.

In this program the LCD functions are stored in lcdfuncs.c

Author: Dogan Ibrahim
Date  : March 2021
Work  : LCD2
----------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include "string.h"
#include "stdlib.h"
#include "lcdfuncs.c"

/******* Main program *******/
int main(void)
{
        int count = 0;
        char buff[10];

        lcd_init();                             // Initialize LCD
        lcd_clear();                            // Clear display
        lcd_puts("Counter");                    // Display Counter

    while (1)                                   // Do forever
    {
        lcd_goto(0, 1);                         // Cursor at 0,1
        itoa(count, buff, 10);                  // Convert to string
        lcd_puts(buff);                         // Display count
        count++;                                // Increment count
        MXC_Delay(1000000);                     // Wait 1 second
    }
}
```

Figure 4.38 Simplified program listing

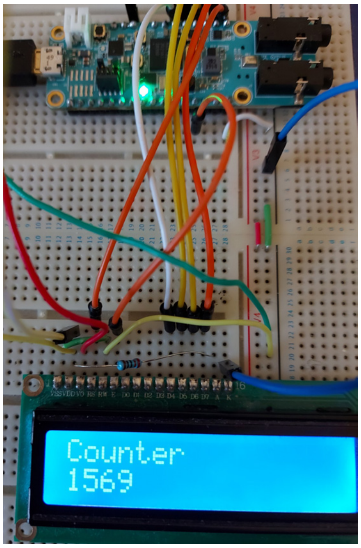


Figure 4.39 Project built on a breadboard

# Chapter 5 • Analog-To-Digital converters (ADC)

## 5.1 • Overview

Most real-world sensors provide an analog output such as resistance, voltage, or current which is proportional to a measured variable. Such sensors cannot be connected directly to digital computers without using an ADC. In this chapter, we will learn how to use the ADC channels of the MAX78000FTHR development board to interface to analog sensors.

Most ADCs for general purpose applications are 8 or 10-bits wide, although some higher-grade professional ones are 16 or even 32-bit wide. The conversion time of an ADC is one of its important specifications. This is the time taken for the ADC to convert an analog input into digital. The smaller the conversion time the better. Some cheaper ADCs give the converted digital data in serial format, while other more expensive professional ones provide parallel digital outputs.

The MAX78000 microcontroller has 8 ADC channels with a maximum clock rate of 8MHz. Only 2 of the ADC channels (AIN3 at pin 5, and AIN4 at pin 6) are available at the MAX78000FTHR GPIO pins (see Figure1.1).

The ADC has a resolution of 10-bits, thus converting an analog input voltage into 1024 (0 to 1023) levels. The reference voltage of the ADC can either be +1.22V or VDDA/2. Using the ADC with +1.22V reference voltage, the resolution is 1220 mV/1024 = 1.19 mV per bit. Therefore, an analog input voltage of 1.19 mV gives a digital output of 00 0000001, 2.38 mV gives 00 00000010, and so on. The default reference voltage is +1.22V.

The main features of the MAX78000FTHR ADC are:

- 8MHz maximum ADC clock rate
- Two reference sources, an internal 1.22V bandgap or the VDDA analog supply
- 8 External analog inputs that can be configured as 4 two-input comparators
- 10 Internal power supply monitor inputs
- Fixed 10-bit word conversion time of 1024 ADC clock cycles
- Programmable out-of-range (limit) detection
- Interrupt generation for limit detection, conversion start, conversion complete, and internal reference powered on
- Serial ADC data measurements
- ADC conversion 10 output either MSB or LSB aligned

In this chapter, we will develop several projects using the ADC of the MAX78000 microcontroller.

## 5.2 • Project 1 – Voltmeter with LCD

**Description**: This is a simple voltmeter project where the voltage of an external voltage source is measured and displayed on the LCD in millivolts every second.

**Aim**: This project aims to show how the MAX78000 ADC channels can be used to read analog input voltage

**Circuit diagram**: Figure 5.1 shows the circuit diagram of the project. In this project, the voltage to be measured is applied to analog input AIN3 (pin 5). You must make sure the input voltage does not exceed +1.22 V. If it is a requirement to measure higher voltages, use resistive potential divider circuits at the input of the ADC.

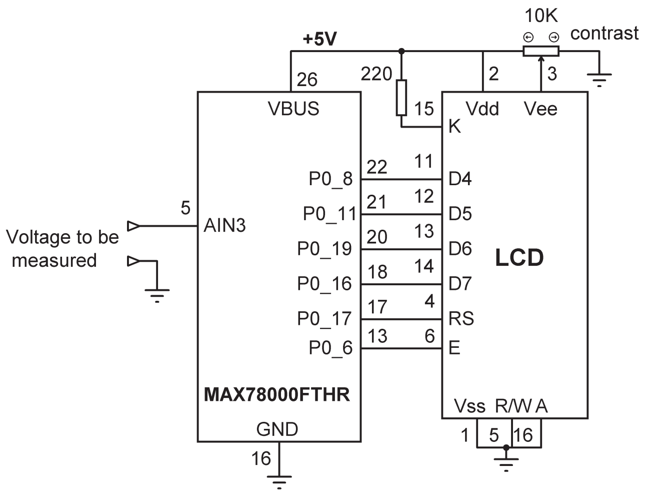


Figure 5.1 Circuit diagram of the project

**Program listing**: Figure 5.2 shows the program listing (**Voltmeter**). At the beginning of the program module channel 3 is defined, LCD and ADC are both initialised. The remaining parts of the program run in a while loop. Inside this loop, analog voltage is converted into millivolts and stored in variable **mV**. This variable is then converted into integer and string and displayed on the LCD. The above process is repeated every second. The following function reads data from ADC channel 3 and stores the converted data in variable **adc_val**:

```
adc_val = MXC_ADC_Start(ADC_CHANNEL);
```

```
/*----------------------------------------------------------------------
                VOLTMETER WITH LCD

In this program an LCD is connected to MAX78000FTHR. The program
measures analog voltage at pin AIN3 and displays on LCD in mV

Author: Dogan Ibrahim
Date  : March 2021
Work  : Voltmeter
----------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include <mxc.h>
#include "string.h"
#include "stdlib.h"
#include "lcdfuncs.c"

#define ADC_CHANNEL MXC_ADC_CH_3                        // Use CH 3

/******* Main program *******/
int main(void)
{
        int MV;
        int adc_val=0;
        float mV;
        char buff[10];

        lcd_init();                                     // Initialize LCD
        MXC_ADC_Init();                                 // Initialize ADC

    while (1)                                           // Do forever
    {
        adc_val=MXC_ADC_StartConversion(ADC_CHANNEL);   // Start conversion
        lcd_clear();
        mV = 1220.0 * adc_val / 1024;                   // Convert to mV
        MV = (int)mV;                                   // As integer
        itoa(MV, buff, 10);                             // Convert to string
        lcd_puts(buff);                                 // Display measured mV
        MXC_Delay(1000000);                             // Wait 1 second
    }
}
```

Figure 5.2 Program listing

An optional feature allows samples captured by the ADC to be automatically compared against user-programmable high and low limits. Up to four channel limit pairs can be configured in this way. The comparison allows the ADC to trigger an interrupt (and potentially wake the CPU from a power mode) when a captured sample goes outside the pre-programmed limit range. Since this comparison is performed directly by the sample limit monitors, it can be performed even while the CPU is in SLEEP, LOW POWER or MICROPOWER mode. The eight AIN[7:0] inputs can be configured as four pairs and deployed as four independent comparators.

Some frequently used ADC functions are (see adc.h for a complete list of functions):

**MXC_ADC_INIT(void)**: initialise the ADC

**MXC_ADC_Shutdown(void)**: shutdown the ADC

**MXC_ADC_Busy(void)**: check if ADC is busy

**MXC_ADC_EnableInt (uint32_t flags)**: enable specific ADC interrupts

**MXC_ADC_DisableInt (uint32_t flags)**: disable specific ADC interrupts

**MXC_ADC_SetConversionSpeed (uint32_t hz)**: set ADC conversion speed

**MXC_ADC_SetExtScale (mxc_adc_scale_t scale)**: set ADC scaling factor

**MXC_ADC_RefSelect (mxc_adc_ref_t ref)**: set ADC reference source

**MXC_ADC_EnableMonitor (mxc_adc_monitor_t monitors)**: enable channel HIGH/ LOW monitor

**MXC_ADC_DisableMonitor (mxc_adc_monitor_t monitors)**: disable channel HIGH/ LOW monitor

**MXC_ADC_SetMonitorHighThreshold (mxc_adc_monitor_t monitor, uint32_t threshold)**: set channel HIGH limit for monitoring

**MXC_ADC_SetMonitorLowThreshold (mxc_adc_monitor_t monitor, uint32_t threshold)**: set channel LOW limit for monitoring

**MXC_ADC_SetMonitorChannel (mxc_adc_monitor_t monitor, mxc_adc_chsel_t channel)**: Set a monitor to use a specific channel

**MXC_ADC_StartConversion (mxc_adc_chsel_t channel)**: perform conversion on a specific channel

**MXC_ADC_Convert (mxc_adc_conversion_req_t* req)**: perform conversion on a specific channel

**MXC_ADC_GetData (uint16_t *outdata)**: get results of the previous conversion

Figure 5.3 shows the block diagram of the MAX78000 ADC module. There are 8 external inputs and several internal inputs to the ADC multiplexer. The internal inputs are from various voltage sources of the microcontroller. For small signals, the ADC and reference (or both) can be scaled by 50%. This enables flexibility to achieve better resolution on the ADC conversion. Each input channel, supports the default of no scaling of the input (**ADC_CTRL.scale = 0**) and no scaling of the reference (**ADC_CTRL.ref_scale = 0**).

The external inputs, AIN0 through AIN7, support scaling of the input by 50%, the reference by 50%, or both by 50%. Also, the scaling can further be modified by additional factors of 2, 3, or 4 as defined by **ADC_CTRL.adc_divsel**. The converted data is 16-bits. It can either be LSB left-justified or LSB right-justified. By default, the data is LSB left-justified, so the upper 6 bits of the data are filled with zeroes.

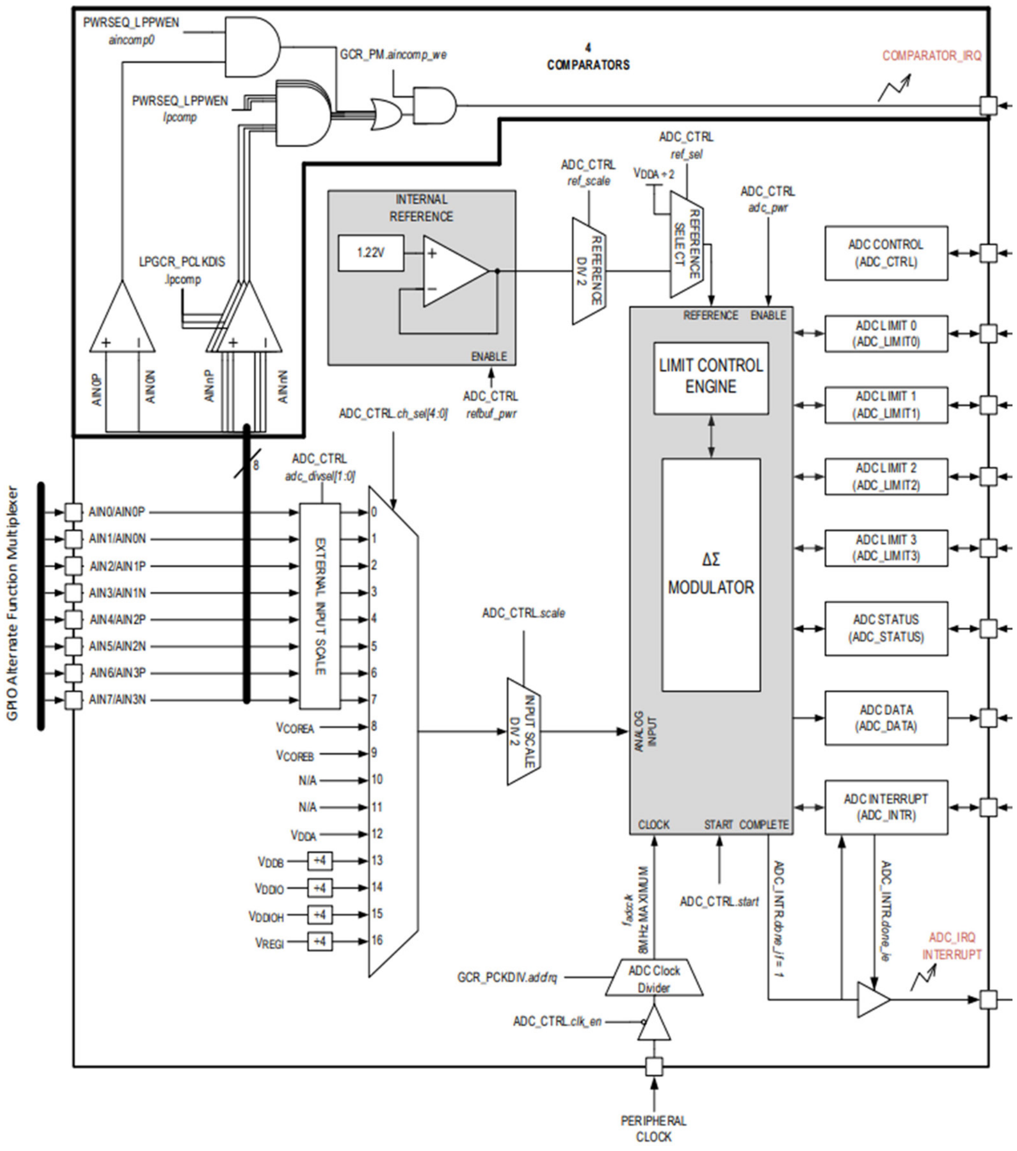


Figure 5.3 Block diagram of the ADC module

## 5.3 • Project 2 – Temperature measurement

**Description**: In this project, an external analog temperature sensor chip is used to measure and display the ambient temperature on the LCD

**Aim**: This project aims to show how an external analog temperature sensor chip can be used to measure external temperature

**Block Diagram**: Figure 5.4 shows the block diagram of the project

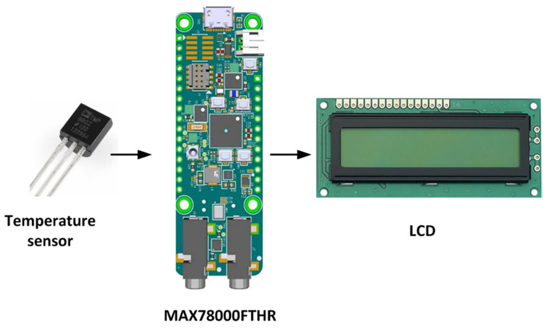


Figure 5.4 Block diagram of the project

**Circuit diagram**: The circuit diagram of the project is shown in Figure 5.5. In this project, a TMP36 type temperature sensor chip is used. (Figure 5.6) It is connected to ADC channel AIN3 (pin 5). The chip is powered from the +3.3V supply (pin 2) of the MAX78000FTHR. This chip provides an analog output voltage proportional to the measured temperature. The relationship between the measured temperature and the output voltage is given by:

T = (Vo – 500) / 10

Where **T** is the measured temperature in degrees Celsius, and **Vo** is the sensor output voltage in millivolts. For example, at 20ºC the output voltage is 700mV, at 30ºC it is 800mV, and so on.

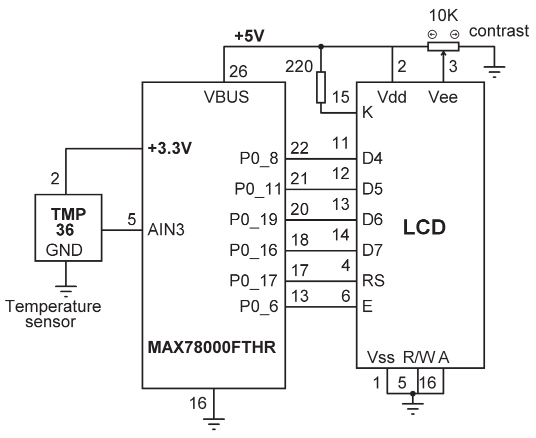


Figure 5.5 Circuit diagram of the project

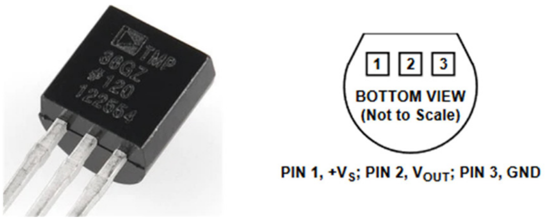


Figure 5.6 TMP36 sensor chip

**Program listing**: Figure 5.7 shows the program listing (**TMP36**). Sensor voltage is read by channel 3 of the ADC. This voltage is then converted into millivolts, 500 subtracted and divided by 10. The result is in degrees Celsius and is displayed on the LCD every second.

```
/*----------------------------------------------------------------------
                TEMPERATURE MEASUREMENT WITH LCD

In this program an LCD is connected to MAX78000FTHR. Also, a TMP36
type temperature sensor chip is connected to analog input AIN3. The
program displays teh ambient temperature on the LCD

Author: Dogan Ibrahim
Date  : March 2021
Work  : TMP36
----------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include <mxc.h>
#include "string.h"
#include "stdlib.h"
#include "lcdfuncs.c"

#define ADC_CHANNEL MXC_ADC_CH_3                // Use CH 3
static uint16_t adc_val;

/******* Main program *******/
int main(void)
{
        int T;
        float mV;
        char buff[10];

        lcd_init();                             // Initialize LCD
        MXC_ADC_Init();                         // Initialize ADC
```

```
    while (1)                                   // Do forever
    {
        adc_val = MXC_ADC_StartConversion(ADC_CHANNEL); // Start conversion
        lcd_clear();
        mV = 1220.0 * adc_val / 1023;           // Convert to mV
        mV = (mV - 500.0)/ 10.0;                // Temperature in C
        T = (int)mV;                            // As integer
        itoa(T, buff, 10);                      // Convert to string
        lcd_puts(buff);                         // Display measured mV
        MXC_Delay(1000000);                     // Wait 1 second
    }
}
```

Figure 5.7 Program listing

## 5.4 ● Project 3 – ON/OFF temperature controller

**Description**: Temperature control is important in many industrial, commercial, and domestic chemical applications. A temperature control system consists of a temperature sensor, heater, fan (optional), actuator to operate the heater, and a controller. Negative feedback is used to control the heater so the temperature is at the desired set-point value. Accurate temperature control systems are based on the PID (Proportional+Integral+Derivative) algorithm. In this project, an ON/OFF type simple control system is designed. ON/OFF temperature control systems commonly use relays to turn the heater ON or OFF depending on the set-point temperature and the measured temperature. If the measured temperature is below the set-point value, the relay is activated which turns the heater ON. If on the other hand, the measured temperature is above the set-point value, the relay is de-activated to turn OFF the heater so that the temperature is lowered.

In this project, a TMP36 type temperature sensor chip, a relay, and a heater are used to control the temperature of a small room. The heater is turned **ON** by the relay if the measured room temperature (**RoomTemp**) is below the set-point temperature (**SetTemp**), and it is turned **OFF** if it is above the set-point value. This process is repeated every 3 seconds.

**Aim**: This project aims to show how an ON/OFF temperature control system can be designed using a low-cost temperature sensor chip with the MAX78000FTHR development board.

**Block diagram**: Figure 5.8 shows the block diagram of the project.

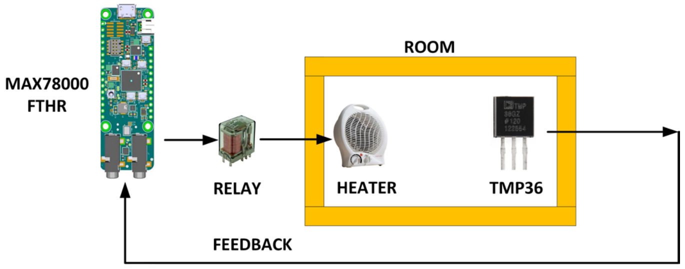


Figure 5.8 Block diagram of the project

**Circuit diagram**: The circuit diagram of the project is shown in Figure 5.9. A TMP36 sensor chip is connected to analog channel 3 as in the previous project. The relay is connected to port pin P0_5 (pin 12) through a transistor switch (any NPN type transistor) and is activated when logic 1 is applied to it. The LCD is connected as in the previous projects using LCDs.

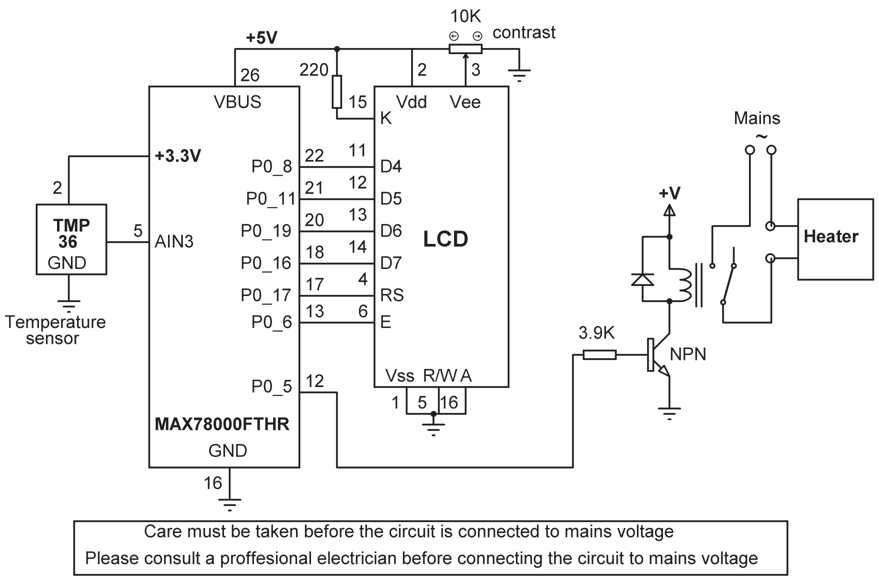


Figure 5.9 Circuit diagram of the project

## Operation of the project

The operation of the project is described in Figure 5.10 as a PDL (*Program Description Language*).

```
BEGIN
        Configure and turn OFF the relay
        Initialise the LCD
        Initialise the ADC
        Specify the Set temperature
        Display the Set temperature at top row of LCD
        DO WHILE
                Read the analog sensor data
                Convert to Room temperature in Celsius
                IF Set temperature > Room temperature THEN
                        Turn ON relay
                ELSE
                        Turn OFF relay
                ENDIF
                Display the Room temperature at bottom row of LCD
                Wait 3 seconds
        ENDDO
END
```

Figure 5.10 PDL of the project

**Program listing**: Figure 5.11 shows the program listing (**ONOFF)**. The desired temperature is set to 27ºC and is stored in variable **SetTemp**. The **Relay** is assigned to P0_5 and is turned OFF at the beginning of the program (so that the heater is OFF).

The LCD and the ADC are initialised and the **SetTemp** converted into a floating-point string (in **buffset**) so it can be displayed on the LCD. The remaining parts of the program run in a **while** loop. Inside this loop, the room temperature is read from the sensor, converted into digital, and stored in variable **RoomTemp**. If the **SetTemp** is higher than the **RoomTemp** then the relay is activated so the heater turns ON. If on the other hand, the **SetTemp** is less than or equal to the **RoomTemp** then the relay is deactivated so the heater is turned OFF. Both the **SetTemp** and the **RoomTemp** are displayed on the LCD. This process is repeated every 3 seconds.

```
/*----------------------------------------------------------------------
        ON-OFF TEMPERATURE CONTROL WITH LCD

In this program an LCD is connected to MAX78000FTHR. Also, a TMP36
type temperature sensor chip in a room is connected to analog input AIN3.
The program controls the temperature in the room using ON-OFF technique.
Both the measured and the set temperature are displayed on the LCD

Author: Dogan Ibrahim
Date  : March 2021
Work  : ONOFF
------------------------------------------------------------------------*/
/***** Includes *****/
```

```
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include <mxc.h>
#include "string.h"
#include "stdlib.h"
#include "lcdfuncs.c"

#define ADC_CHANNEL MXC_ADC_CH_3                        // Use CH 3
static uint16_t adc_val;
#define Relay MXC_GPIO_PIN_5                            // Relay at pin 5
mxc_gpio_cfg_t gpio_out;

/******* Main program *******/
int main(void)
{
        float mV;
        char buffroom[10], buffset[10];
        float SetTemp, RoomTemp;

        /* Setup output pins */
        gpio_out.port = MXC_GPIO0;                      // Port 0
        gpio_out.mask = Relay;                          // Relay
        gpio_out.pad = MXC_GPIO_PAD_NONE;               // None
        gpio_out.func = MXC_GPIO_FUNC_OUT;              // Output
        gpio_out.vssel = MXC_GPIO_VSSEL_VDDIOH;         // Set port to +3.3V
        MXC_GPIO_Config(&gpio_out);

        MXC_GPIO_OutClr(gpio_out.port, Relay);          // Relay OFF at beginning

        lcd_init();                                     // Initialize LCD
        MXC_ADC_Init();                                 // Initialize ADC
        SetTemp = 27.0;                                 // Set temp
        sprintf(buffset, "%5.2f", SetTemp);             // Convert to string

    while (1)                                           // Do forever
    {
        adc_val = MXC_ADC_StartConversion(ADC_CHANNEL); // Start conversion
        lcd_clear();                                    // Clear LCD
        lcd_puts("Set : ");                             // Display heading
        lcd_puts(buffset);                              // Display set point
        mV = 1220.0 * adc_val / 1023;                   // Convert to mV
        RoomTemp = (mV - 500.0)/ 10.0;                  // Temperature in C

        if(SetTemp > RoomTemp)                          // IF SetTemp > RoomTemp
```

```
                MXC_GPIO_OutSet(gpio_out.port, Relay);  // Relay ON
        else
                MXC_GPIO_OutClr(gpio_out.port, Relay);  // Relay OFF

        sprintf(buffroom, "%5.2f", RoomTemp);           // Convert to string
        lcd_goto(0, 1);                                 // Cursor at 2nd row
        lcd_puts("Room: ");                             // Display heading
        lcd_puts(buffroom);                             // Display room temp
        MXC_Delay(3000000);                             // Wait 3 second
    }
}
```

Figure 5.11 Program listing

Figure 5.12 shows an example display.

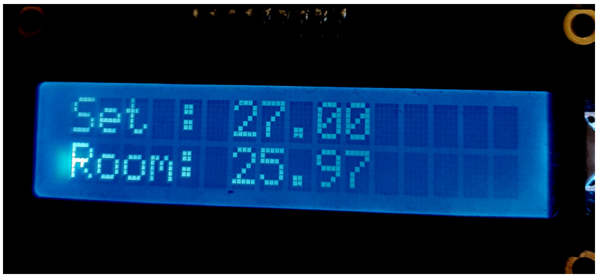


Figure 5.12 Example display

## 5.5 ● Project 4 – ADC with completion interrupt – displaying the temperature

**Description**: Interrupt-based ADC can be a very useful feature, especially in real-time systems where a quick response is required. The basic operation is such that the ADC generates an interrupt whenever the data has been converted. A flag is then set in the interrupt service routine to inform the main program that the ADC conversion has been completed. An example is given in this section. In this example, the temperature is read from the TMP36 temperature sensor chip after detecting that the conversion is complete, and is displayed on the LCD.

**Aim**: This project aims to show how the ADC can be configured to generate completion interrupts and how the converted data can be extracted.
The block diagram and circuit diagram of the project are as in Figures 5.4 and 5.5 respectively.

**Program listing**: Figure 5.13 shows the program listing (**ADCInt**). LCD and ADC are initialised in the main program and ADC interrupts are enabled. The program starts the conversion and waits until flag **adc_done** is set in the callback function **adc_complete**. The converted data is then read by calling function **MXC_ADC_GetData**. The temperature is then calculated in degrees Celsius and displayed on the LCD as in the previous projects.

The ADC interrupt must be enabled and **MXC_ADC_Handler** called in the ISR places data in the error parameter of the callback function.

```
/*----------------------------------------------------------------------
        ADC WITH COMPLETION INTERRUPT-DISPLAYING THE TEMPERATURE

In this program an LCD is connected to MAX78000FTHR. Also, a TMP36
type temperature sensor chip is conencted to the board. ADC comletion
interrupt sets a flag which enables the main program to continue and
display the temperature on the LCD

Author: Dogan Ibrahim
Date  : March 2021
Work  : ADCInt
----------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include <mxc.h>
#include "string.h"
#include "stdlib.h"
#include "lcdfuncs.c"

#define ADC_CHANNEL MXC_ADC_CH_3                        // Use CH 3
static uint16_t adc_val;
volatile unsigned int adc_done = 0;

//
// ADC interrupt service routine. Set flag adc_done
//
void adc_complete(void* req, int error)
{
    adc_done = 1;                                       // Interrupt occured flag
    return;
}

void ADC_IRQHandler(void)
{
    MXC_ADC_Handler();
}


/******* Main program *******/
int main(void)
```

```
{
        float mV, T;
        char buffer[10];

        lcd_init();                                         // Initialize LCD
        MXC_ADC_Init();                                     // Initialize ADC
    NVIC_EnableIRQ(ADC_IRQn);                               // Enable ADC interrupts

    while (1)                                               // Do forever
    {
        adc_done = 0;
        MXC_ADC_StartConversionAsync(ADC_CHANNEL, adc_complete);
        while (!adc_done) {};

        MXC_ADC_GetData(&adc_val);                          // GEt converted data
        lcd_clear();                                        // Clear LCD

        mV = 1220.0 * adc_val / 1023;                       // Convert to mV
        T = (mV - 500.0)/ 10.0;                             // Temperature in C
        sprintf(buffer, "%5.2f", T);                        // Convert to string
        lcd_puts(buffer);                                   // Display temperature
        MXC_Delay(3000000);                                 // Wait 3 second
    }
}
```

Figure 5.13 Program listing

# Chapter 6 • Serial Communication – UART

## 6.1 • Overview

Serial communication is a simple means of sending data over long distances quickly and reliably. The most commonly used serial communication method is based on the RS232 standard. In this standard, data is sent over a single line from a transmitting device to a receiving device in bit-serial format at a pre-specified speed, also known as the Baud rate, or the number of bits sent each second. Typical Baud rates are 4800, 9600, 19200, 38400, etc.

RS232 serial communication is a form of asynchronous data transmission where data is sent character by character. Each character is preceded with a Start bit, seven or eight data bits, an optional parity bit, and one or more stop bits. The most commonly used format is eight data bits, no parity bit, and one stop bit. Therefore, a data frame consists of 10-bits. With a Baud rate of 9600, we can transmit and receive 960 characters every second. The least significant data bit is transmitted first, and the most significant bit is transmitted last.

In standard RS232 communication, logic high is defined to be at -12V, and logic 0 at +12V. Figure 6.1 shows how character "A" (ASCII binary pattern 0010 0001) is transmitted over a serial line. The line is normally idle at -12V. The start bit is first sent by the line going from high to low. Then eight data bits are sent starting from the least significant bit. Finally, the stop bit is sent by raising the line from low to high.

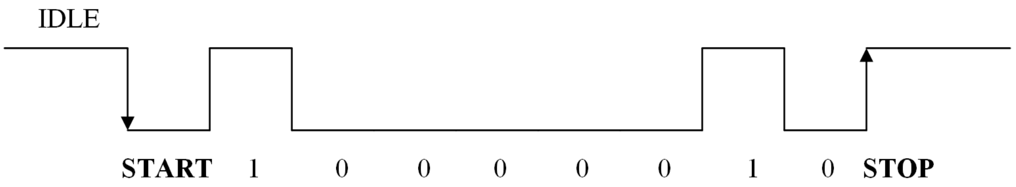


Figure 6.1 Sending character "A" in serial format

In serial communication a minimum of three lines are used for communication: transmit (TX), receive (RX), and ground (GND). Some high-speed serial communication systems use additional control signals for synchronisation, such as CTS, DTR, and so on. Some systems use software synchronisation techniques where a special character (XOFF) is used to tell the sender to stop sending. Another character (XON) is used to tell the sender to restart transmission. RS232 devices are connected using two types of connectors: 9-way and 25-way. Table 6.1 shows the TX, RX, and GND pins of each type of connector. The connectors used in RS232 serial communication are shown in Figure 6.2.

| **9-pin connector** | |
|---|---|
| **Pin** | **Function** |
| 2 | Transmit (TX) |
| 3 | Receive (RX) |
| 5 | Ground (GND) |

| **25-pin connector** | |
|---|---|
| **Pin** | **Function** |
| 2 | Transmit (TX) |
| 3 | Receive (RX) |
| 7 | Ground (GND) |

Table 6.1 Minimum pins required for RS232 serial communication

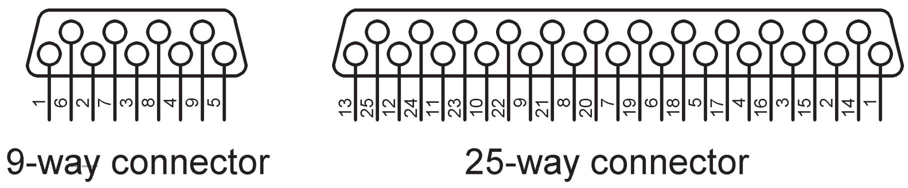


Figure 6.2 RS232 connectors

As described above, RS232 voltage levels are at ±12V. On the other hand, microcontroller input-output ports operate at 0 to +3.3V voltage levels. It is therefore necessary to translate voltage levels before a microcontroller can be connected to an RS232 compatible device. Thus, the output signal from the microcontroller has to be converted into ±12V, and the input from an RS232 device must be converted into 0 to +3.3V before it can be connected to a microcontroller. This voltage translation is normally done using special RS232 voltage converter chips. One such popular chip is the MAX232. This is a dual converter chip having the pin configuration as shown in Figure 6.3. This particular device requires four external 1µF capacitors for its operation.

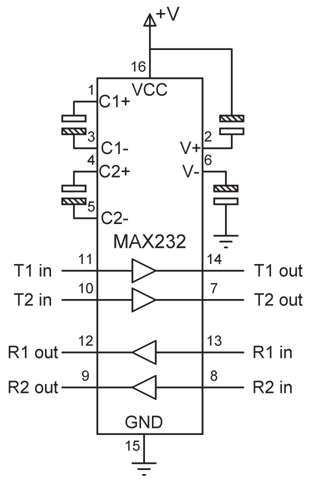


Figure 6.3 MAX232 pin configuration

Nowadays, serial communication is made using standard TTL logic levels instead of ±12V, where logic 1 is +5V (or greater than +3.3V) and logic 0 is 0V. A serial line is idle when the voltage is at +5V. The start bit is identified on the high-to-low transition of the line, i.e. the transition from +5V to 0V.
In this chapter, we will develop programs using the UART of MAX78000FTHR development board.

## 6.2 • MAX78000FTHR UART serial ports

The MAX78000 microcontroller has normal and low power UART (LPUART). The LPUART is a special version of UART that can receive characters at 9600 Baud while in a low power mode. The basic features of the UART are:

- Flexible baud rate generation up to 12.5Mbps
- Programmable character size of 5-bits to 8-bits
- Stop bit settings of 1, 1.5, or 2-bits
- Parity settings of even, odd, mark (always 1), space (always 0), and no parity
- Automatic parity error detection with selectable parity bias
- Automatic framing error detection
- Separate 8-byte transmit and receive FIFOs
- Flexible interrupt conditions
- Hardware flow control for RTS and CTS
- DMA capable

The LPUART provides these additional features:

- Flexible baud rate generation up to 1.85Mbps
- Ability to receive characters at 9600 Baud in ACTIVE, SLEEP, LPM, and μPM modes
- Fractional baud rate divisor settings to allow greater accuracy at slow baud rates

- Wakeup to ACTIVE on multiple RX FIFO conditions

The MAX78000 microcontroller has three UART ports and one LPUART port with the names:

**UART0 (includes hardware flow control)**
**UART1**
**UART2**
**LPUART**

UART0 is the console/DAPLink UART at pins P0_1 (TXD) and P0_0 (RXD). This UART is used to communicate with a PC over a virtual COM port using terminal emulation software as we have already seen while using the **printf** statement. This behaviour can be changed in **board.h** in **C:\MaximSDK\Libraries\Boards\MAX78000\FTHR_RevA\Include** by altering **#define CONSOLE_UART 0**. If you want to have a console on pins 7 and 8 change it to 2.

UART2 is at pins 7 (TX) and 8 (RX), and LPUART at pins 15 (TX) and 14 (RX) on the MAX78000FTHR development board (see Figure 6.4).

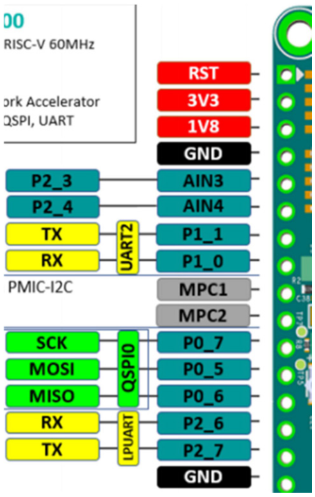


Figure 6.4 MAX78000FTHR board UART serial ports

Figure 6.5 shows the block diagram of the UART. Separate read (RX FIFO) and write (TX FIFO) FIFOs are provided. TX FIFO pointer is incremented after data is written to the data register. Similarly, reading data returns the character in the RX FIFO and decrements the FIFO pointer.

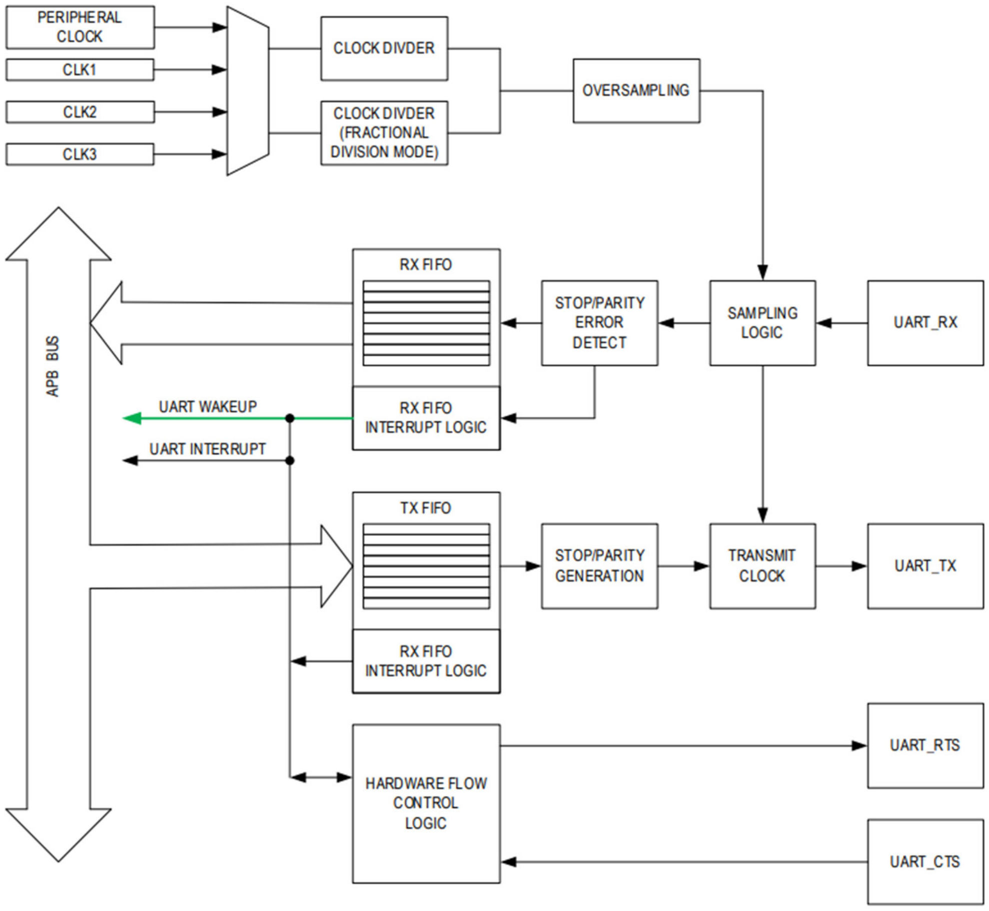


Figure 6.5 UART block diagram

Some example UART-based projects are given in the next sections.

## 6.3 • Project 1 – Sending the temperature readings to a terminal with relative time stamping

**Description**: In this project, we will be using a TMP36 temperature sensor chip to read ambient temperature every 5 seconds. The readings will be displayed on the terminal with relative seconds stampings since the program started.

**Aim**: This project aims to show how the UART can be used in a project.

**Block diagram**: Figure 6.6 shows the block diagram of the project. A virtual com port is used to get the readings from the development board and send them to a terminal emulation program running on the PC.

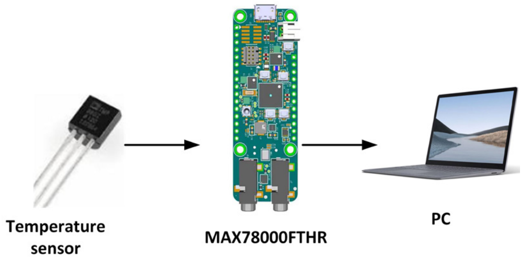


Figure 6.6 Block diagram of the project

**Circuit diagram**: The circuit diagram of the project is shown in Figure 6.7. The MAX78000FTHR is connected to a PC through its USB port.

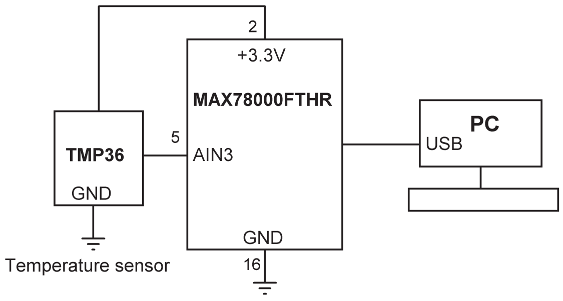


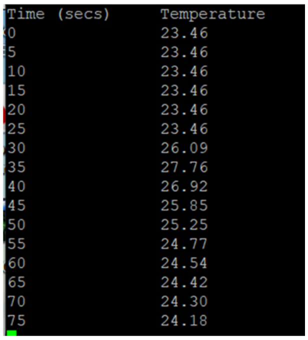
Figure 6.7 Circuit diagram of the project

**Program listing**: The program listing (**TMP36UART**) is shown in Figure 6.8. The program initialises the ADC and displays a heading message using the **printf** statement. The remaining parts of the program are executed in a **while** loop. Inside this loop, the temperature is read, converted into degrees Celcius and displayed on the terminal with relative time in seconds. The **printf** statement is used to display the data. Notice that **\t** in the **printf** statement is the TAB character which moves the cursor by one tab position.

```
/*------------------------------------------------------------------------
                TEMPERATURE MEASUREMENT WITH UART

In this program a TMP36 type temperature sensor chip is connected to analog
input AIN3.The program displays the ambient temperature on a terminal

Author: Dogan Ibrahim
Date  : March 2021
Work  : TMP36UART
------------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include <mxc.h>

#define ADC_CHANNEL MXC_ADC_CH_3                        // Use CH 3
static uint16_t adc_val;

/******* Main program *******/
int main(void)
{
        float mV, T;
        int Secs = 0;

        MXC_ADC_Init();                                 // Initialize ADC
        printf("Time (secs)\tTemperature\n");           // Display heading

    while (1)                                           // Do forever
    {
        adc_val = MXC_ADC_StartConversion(ADC_CHANNEL); // Read data from sensor
        mV = 1220.0 * adc_val / 1023;                   // Convert to mV
        T = (mV - 500.0)/ 10.0;                         // Temperature in C
        printf("%d\t\t%5.2f\n", Secs, T);               // Display temperature
        Secs = Secs + 5;
        MXC_Delay(5000000);                             // Wait 5 seconds
    }
}
```

Figure 6.8 Program listing

Figure 6.9 shows an example output from the program. In this project, the **Putty** terminal emulation software was used in serial mode with the Baud rate set to 115200 (the default Baud rate of the MAX78000FTHR), and port COM5 was used (you will have to find the serial port number of the virtual com port using **Device Manager** on your PC).



```
Time (secs)      Temperature
0                23.46
5                23.46
10               23.46
15               23.46
20               23.46
25               23.46
30               26.09
35               27.76
40               26.92
45               25.85
50               25.25
55               24.77
60               24.54
65               24.42
70               24.30
75               24.18
```

Figure 6.9 Example data displayed on the terminal

## 6.4 ● Project 2 – Calculator project using a terminal

**Description**: This is a 4-function calculator that can add, subtract, multiply, and divide numbers. In this project, we will be using a terminal to enter the numbers to be manipulated. The calculation is performed on the MAX78000FTHR development board and the result is sent and displayed on the terminal.

**Aim**: This project aims to show how UART output and input functions can be used in a simple program.

The block diagram of the project is as in Figure 6.6. There is no TMP36 in this project.

**Circuit diagram**: Figure 6.10 shows the circuit diagram of the project.

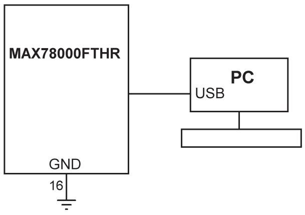


Figure 6.10 Circuit diagram of the project

**Program listing**: Figure 6.11 shows the program listing (UARTCALC). At the beginning of the program, the user is prompted to enter numbers and the required operation. Function scanf is used to read numbers and the required operation. For example, to add numbers 5 and 4, enter as:

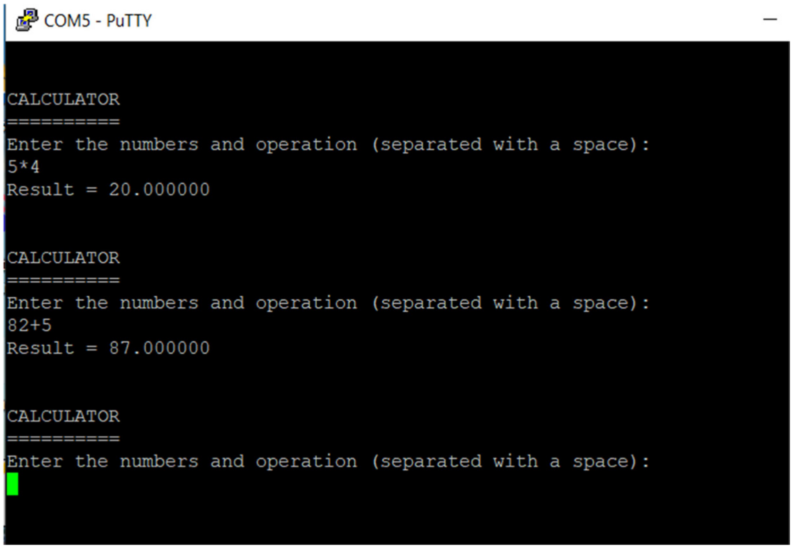
5+4 or as 5 + 4

The result is calculated, stored in variable result, and is displayed on the terminal.

```
/*------------------------------------------------------------------------
                        CALCULATOR PROGRAM

This is a simple 4 function calculator program that can add,subtarct,multiply
and divide the given numbers. The numebrs are entered from a terminal. The
program calculates the required operation and sends teh result to the terminal

Author: Dogan Ibrahim
Date  : March 2021
Work  : UARTCALC
------------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include <mxc.h>

/******* Main program *******/
int main(void)
{
        float no1, no2, result = 0;
        char oper;

    while (1)                                              // Do forever
    {
        printf("\n\nCALCULATOR");
        printf("\n==========\n");
        printf("Enter the numbers and operation (separated with a space):\n");
        scanf("%f %c %f", &no1, &oper, &no2);

        switch(oper)
        {
                case '+':                                  // Add?
                        result = no1 + no2;
                        break;
                case '-':                                  // Subtract?
                        result = no1 - no2;
                        break;
                case '*':                                  // Multiply?
                        result = no1 * no2;
                        break;
                case '/':                                  // Divide?
                        result = no1 / no2;
                        break;
        }
        printf("\nResult = %f\n", result);              // Display result
    }
}
```

Figure 6.11 Program listing

Figure 6.12 shows an example run of the program.



```
CALCULATOR
==========
Enter the numbers and operation (separated with a space):
5*4
Result = 20.000000


CALCULATOR
==========
Enter the numbers and operation (separated with a space):
82+5
Result = 87.000000


CALCULATOR
==========
Enter the numbers and operation (separated with a space):
```

Figure 6.12 Example run of the program

There are many functions related to UART. Some of the important functions are (see file **uart.h** for full details):

int MXC_UART_Init(mxc_uart_regs_t* uart, unsigned int baud, mxc_uart_clock_t clock): this function initialises UART. The following parameters are set by default:

*** UART Data Size - 8 bits**
*** UART Stop Bits - 1 bit**
*** UART Parity - None**
*** UART Flow Control - None**

**int MXC_UART_SetDataSize(mxc_uart_regs_t* uart, int dataSize)**: This function sets the data size (5-8 bits).

**int MXC_UART_SetStopBits(mxc_uart_regs_t* uart, mxc_uart_stop_t stopBits)**: This function sets the number of stop bits.

**int MXC_UART_SetParity(mxc_uart_regs_t* uart, mxc_uart_parity_t parity)**: This function sets the parity.

**int MXC_UART_SetFlowCtrl(mxc_uart_regs_t* uart, mxc_uart_flow_t flowCtrl, int rtsThreshold)**: This function sets the flow control.

**int MXC_UART_SetClockSource(mxc_uart_regs_t* uart, mxc_uart_clock_t clock)**: This function sets the clock source.

Stop bits can be:

MXC_UART_STOP_1
MXC_UART_STOP_2

Parity options are:

MXC_UART_PARITY_DISABLE
MXC_UART_PARITY_EVEN_0
MXC_UART_PARITY_EVEN_1
MXC_UART_PARITY_ODD_0
MXC_UART_PARITY_ODD_1

Flow control options are:

MXC_UART_FLOW_DIS
MXC_UART_FLOW_EN

Clock options are:

/*8MHz clock can only be used for UART 0, 1 & 2*/
MXC_UART_8M_CLK = 2,

Some commonly used low-level functions are:

**int MXC_UART_WriteCharacterRaw(mxc_uart_regs_t* uart, uint8_t character)**: This function writes a character to UART.

**int MXC_UART_ReadCharacterRaw(mxc_uart_regs_t* uart)**: This function reads the next available character from UART.

## 6.5 • Project 3 – MAX78000FTHR and Arduino Uno serial communication

**Description**: In this project, an Arduino Uno computer is connected to the MAX78000FTHR. The Arduino Uno counts up and sends the count over a serial link to the MAX78000FTHR which displays the count on an LCD.

The Baud rate will be set to 9600 in this project.

**Aim**: This project aims to show how the communication parameters (e.g. Baud rate) can be changed.

**Block diagram**: Figure 6.13 shows the block diagram of the project.

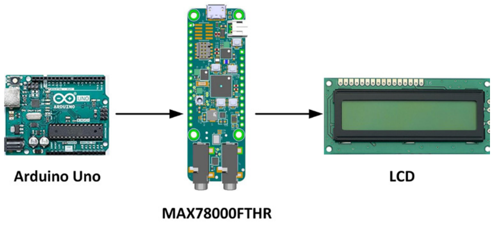


Figure 6.13 Block diagram of the project

**Circuit diagram**: The circuit diagram of the project is shown in Figure 6.14. Arduino pin 2 is used as a software serial port and this pin is connected to the UART2 RX pin (pin 8) of the MAX78000FTHR development board. A resistive potential divider circuit is used to lower output voltage (+5V) of the Arduino Uno to +3.3V so that it is compatible with the MAX78000FTHR input voltage range.

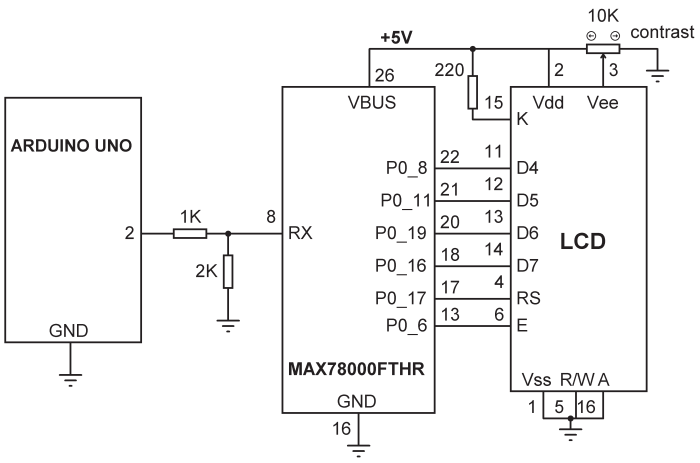


Figure 6.14 Circuit diagram of the project

**Program listing (MAX78000FTHR)**: Figure 6.15 shows the program listing **(UARTREAD)** of the project. At the beginning of the program, the Baud rate is defined as 9600. Function **MXC_UART_Init** is called in the main program to set the Baud rate. UART2 at pins 7 (TX) and 8 (RX) is selected in this project by setting **READING_UART** to 2 and using this variable as the first argument of function **MXC_UART_Init**. The LCD is initialised and the display is cleared. The remaining parts of the program run in a loop. Inside this loop, function **ReadData** is called to read data from the UART. This function reads characters until a new line is detected. The received characters are stored in the character array **RxData**. The data is terminated with a NULL character and is displayed on the LCD. The loop is repeated every 2 seconds.

```
/*------------------------------------------------------------------------
                READ FROM ARDUINO UNO

This program receives data (count) from Arduino Uno and displays on LCD.
The UART baud rate is set to 9600

Author: Dogan Ibrahim
Date  : March 2021
Work  : UARTREAD
-------------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include <mxc.h>
#include "string.h"
#include "uart.h"
#include "lcdfuncs.c"

#define READING_UART        2
#define UART_BAUD           9600
#define BUFF_SIZE           10
char RxData[BUFF_SIZE];

//
// This function reads UART data until newline character is detected.
// Data is returned in array RxData
//
void ReadData()
{
        int i,ch;
        i=0;

        while(1)
        {
                ch=MXC_UART_ReadCharacterRaw(MXC_UART_GET_UART(READING_UART));
                if(ch == '\n')break;
                if(ch > 0)
                {
                        RxData[i] = ch;
                        i++;
                }
        }
        RxData[i-1]='\0';
}


/******* Main program. Read data from Arduino every 2 seconds and display it
*******/
int main(void)
{
        lcd_init();
        lcd_clear();
        MXC_UART_Init(MXC_UART_GET_UART(READING_UART), UART_BAUD, MXC_UART_APB_CLK);

    while (1)                                                // Do forever
    {
        ReadData();                                          // Receive data
        lcd_clear();                                         // Clear LCD
        lcd_puts(RxData);                                    // Display data
        MXC_Delay(2000000);                                  // Wait 2 seconds
    }
}
```

Figure 6.15 Program listing (MAX78000FTHR)

**Program listing (Arduino Uno)**: Figure 6.16 shows the Arduino Uno program listing (**Counter.c**). Pins 2 and 3 are configured as soft serial RX and TX pins (the RX pin is not used in this project). The Baud rate is set to 9600. The program increments a variable called **count** every 3 seconds and sends its value to the serial line.

```
/*******************************************************
 *                    COUNTER
 *                    ========
 * This program counts up every 3 seconds and sends the count
 * to MAX78000FTHR for displaying on LCD
 *
 * Author: Dogan Ibrahim
 * Date  : February, 2021
 * File  : Counter.c
 *******************************************************/
#include <SoftwareSerial.h>
SoftwareSerial MySerial(2, 3);           // RX, TX

int count = 0;
String cntstr;

void setup()
{
  MySerial.begin(9600);                  // Soft serial speed 9600
}

void loop()
{
  cntstr = String(count);               // Convert to string
  MySerial.println(cntstr);             // Send to MAX78000
  count++;                              // Increment count
  delay(3000);                          // Wait 3 seconds
}
```

Figure 6.16 Program listing (Arduino Uno)

## 6.6 • Project 4 – UART interrupts

**Description**: This project is similar to the previous project where the count is displayed on LCD. In this project, UART interrupts are used to read the data from the Arduino.

The Baud rate is set to 9600 as in the previous project.

**Aim**: This project aims to show how UART interrupts can be configured and used.

The block and circuit diagrams are as in Figures 6.13 and 6.14 respectively.

The Arduino program listing is the same as in Figure 6.16.

**Program listing (MAX78000FTHR)**: Figure 6.17 shows the MAX78000FTHR program listing (**UARTINT**). Again, UART2 is used in this project with pin 7 as the TX, and pin 8 the RX. The main program configures UART interrupts. Function **UART_Handler** is called when a character is received by the UART. Received data is stored in array **RxData** as it arrives. The function extracts the characters up to a new line and displays them on the LCD. You should press the Reset button on the development board after loading the program.

```
/*---------------------------------------------------------------------------
                READ FROM ARDUINO UNO - UART INTERRUPTS

This program receives data (count) from Arduino Uno and displays on LCD.
The UART baud rate is set to 9600.

In this version of the program, UART interrupts are used.

Author: Dogan Ibrahim
Date  : March 2021
Work  : UARTINT
-------------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include <mxc.h>
#include "string.h"
#include "uart.h"
#include "nvic_table.h"
#include "lcdfuncs.c"

#define READING_UART     2
#define UART_BAUD        9600
#define BUFF_SIZE        1024

volatile int READ_FLAG;
volatile int indx=0;
volatile int indxs = 0;
mxc_uart_req_t read_req;
uint8_t RxData[BUFF_SIZE];

//
// UART interrupt service routine. Read and display the data on LCD
//
void UART_Handler(void)
{
        char buffer[10];
    MXC_UART_AsyncHandler(MXC_UART_GET_UART(READING_UART));
    lcd_goto(0,0);                                                 // Cursor at 0,0
    if(RxData[indx] == '\n')                                       // Newline?
    {
        RxData[indx-1] = '\0';                          // Terminate with Null
        sprintf(buffer, "%s", RxData+indxs);            // Convert to string
        lcd_puts(buffer);                               // Display on LCD
        indxs = indx+1;
    }
    indx++;
}


void readCallback(mxc_uart_req_t* req, int error)
{
    READ_FLAG = error;
}


/******* Main program. Read data from Arduino every 2 seconds and display it
*******/
int main(void)
{
        lcd_init();
        lcd_clear();
        MXC_UART_Init(MXC_UART_GET_UART(READING_UART), UART_BAUD, MXC_UART_APB_CLK);

    read_req.uart = MXC_UART_GET_UART(READING_UART);
    read_req.rxData = RxData;
    read_req.rxLen = BUFF_SIZE;
    read_req.txLen = 0;
    read_req.callback = readCallback;

    NVIC_ClearPendingIRQ(MXC_UART_GET_IRQ(READING_UART));
    NVIC_DisableIRQ(MXC_UART_GET_IRQ(READING_UART));
    NVIC_SetVector(MXC_UART_GET_IRQ(READING_UART), UART_Handler);
    NVIC_EnableIRQ(MXC_UART_GET_IRQ(READING_UART));

    MXC_UART_ClearRXFIFO(MXC_UART_GET_UART(READING_UART));
    MXC_UART_TransactionAsync(&read_req);

    while (1)                                                   // Do forever
    {
    }
}
```

Figure 6.17 Program listing

# Chapter 7 • I2C Bus Interface

## 7.1 • Overview

The I2C (or I$^2$C) bus is commonly used in microcontroller-based projects. In this chapter, we will look at the use of this bus on the MAX78000FTHR development board. The aim is to make the reader familiar with I2C bus library functions and to show how they can be used in real projects. Before looking at the details of the project, it is worthwhile to look at the basic principles of the I2C bus.

## 7.2 • The I2C Bus

The I2C bus is one of the most commonly used microcontroller communication protocols for communicating with external devices such as sensors and actuators. It is is a single master, multiple slave bus that can operate in standard mode: 100 Kbit/s, full speed: 400 Kbit/s, fast mode: 1 Mbit/s, and high speed: 3.2 Mbit/s. The bus consists of two open-drain wires, pulled-up with resistors:

**SDA: data line**
**SCL: clock line**

Figure 7.1 shows the structure of an I2C bus with one master and three slaves.

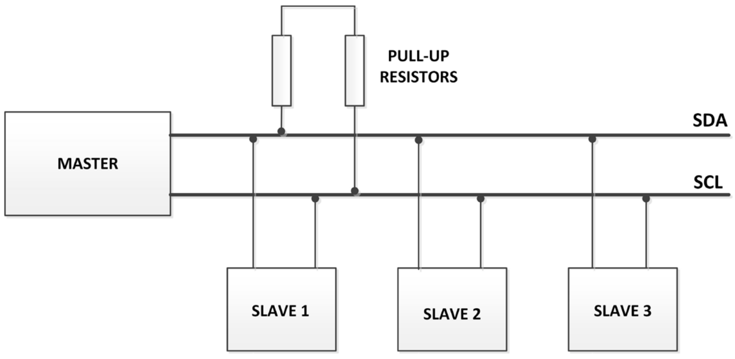


Figure 7.1 I2C bus with one master and three slaves

Because the I2C bus is based on just two wires, there should be a way to address an individual slave device on the same bus. For this reason, the protocol defines that each slave device provides a unique slave address for the given bus. This address is usually 7-bits wide. When the bus is free both lines are HIGH. All communication on the bus is initiated and completed by the master which initially sends a START bit, and completes a transaction by sending a STOP bit. This alerts all the slaves that some data is coming on the bus and all the slaves listen on the bus. After the start bit, 7 bits of unique slave address are sent. Each slave device on the bus has its own address ensuring only the addressed slave communicates on the bus at any time to avoid any collisions. The last sent bit is a read/write bit such that if this bit is 0, it means the master wishes to write to the bus (e.g.

to a register of a slave), if this bit is a 1, it means that the master wishes to read from the bus (e.g. from the register of a slave). The data is sent on the bus with the MSB bit first. An acknowledgment (ACK) bit takes place after every byte and this allows the receiver to signal to the transmitter that the byte was successfully received. As a result, another byte may be sent. ACK bit is sent at the 9th clock pulse.

The communication over the I2C bus is as follows:

- On the bus, the master sends the address of the slave it wants to communicate with
- The LSB is the R/W bit which establishes the direction of data transmission, i.e. from master to slave (R/W = 0), or from slave to master (R/W = 1)
- Required bytes are sent, each interleaved with an ACK bit until a stop condition occurs

Depending on the type of slave device used, some transactions may require a separate transaction. For example, the steps to read data from an I2C compatible memory device are:

- Master starts the transaction in write mode (R/W = 0) by sending the slave address on the bus
- The memory location to be retrieved are then sent as two bytes (assuming 64Kbit memory)
- The master sends a STOP condition to end the transaction
- The master starts a new transaction in read mode (R/W = 1) by sending the slave address on the bus
- The master reads the data from the memory. If reading the memory in sequential format, more than one byte will be read
- The master sets a stop condition on the bus

## 7.3 • I2C pins of the MAX78000 microcontroller

MAX78000 microcontroller I2C bus supports the following basic features:

- Operates as either a master or slave device as a transmitter or receiver
- Supports I2C Standard Mode, Fast Mode, Fast Mode Plus, and High Speed (Hs) mode
- Transfers data at rates up to 3.4Mbps (in High-Speed mode)
- Supports 7- and 10-bit addressing
- Supports RESTART condition and clock stretching
- Provides transfer status interrupts, DMA transfers, and flags
- Provides glitch filter and Schmitt trigger hysteresis on SDA and SCL
- Provides control, status, and interrupt events for maximum flexibility
- Provides independent 8-byte RX FIFO and 8-byte TX FIFO (with pre-loading)
- Provides programmable interrupt threshold levels for the TX and RX FIFO

There are 3 I2C bus interface modules on the MAX78000 microcontroller: I2C0, I2C1, and I2C2. The MAX78000FTHR development board only supports the I2C1at pins P0_16 (SCL), and P0_17 (SDA) as shown in Figure 7.2.

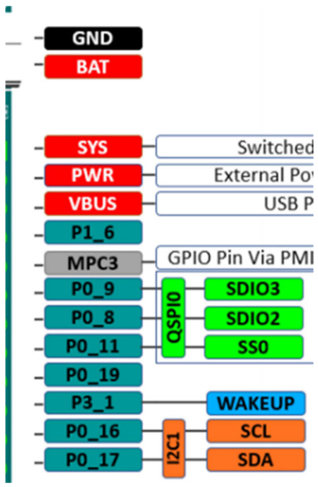


Figure 7.2 MAX78000FTHR I2C pins

In the remaining parts of this chapter, we will develop projects using the I2C bus.

## 7.4 ● Project 1 – I2C port expander

**Description**: A simple project is given in this section to show how the I2C functions can be used in a program. In this project, an I2C bus compatible Port Expander chip (MCP23017) is used to give an additional 16 I/O ports to the MAX78000FTHR development board. This is useful in some applications where a large number of I/O ports may be required. In this project, an LED is connected to MCP23017 port pin GPA0 (pin 21) and the LED is flashed ON and OFF every 500 milliseconds so that the operation of the program can be verified. A 1K current limiting resistor is used in series with the LED.

**Aim**: This project aims to show how the I2C bus can be used on the MAX78000FTHR development board.

**Block diagram**: The block diagram of the project is shown in Figure 7.3.

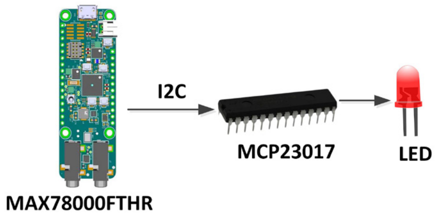


Figure 7.3 Block diagram of the project

## The MCP23017

The MCP23017 is a 28 pin chip with the following features. Pin configuration is shown in Figure 7.4:

- 16 bi-directional I/O ports
- Up to 1.7MHz operation on I2C bus
- Interrupt capability
- External reset input
- Low standby current
- +1.8 to +5.5V operation
- 3 address pins so that up to 8 devices can be used on the I2C bus
- 28-pin DIL package

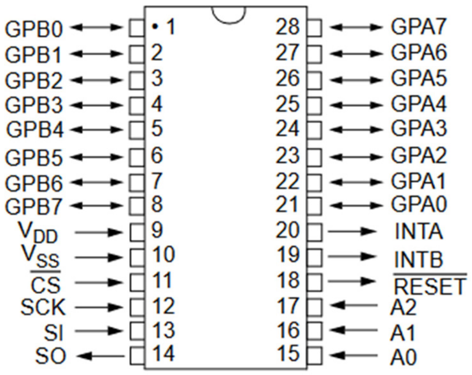


Figure 7.4 Pin configuration of the MCP23017

Pin descriptions are given in Table 7.1.

| Pin | Description |
|---|---|
| GPA0-GPA7 | Port A pins |
| GPB0-GPB7 | Port B pins |
| VDD | Power supply |
| VSS | Ground |
| SDA | I2C data pin |
| SCL | I2C clock pin |
| RESET | Reset pin |
| A0-A2 | I2C address pins |

Table 7.1 MCP23017 pin descriptions

The MCP23017 is addressed using pins A0 to A2. Table 7.2 shows the address selection. In this project, the address pins are connected to ground, thus the address of the chip is 0x20. The chip address is 7 bits wide with the low bit set or cleared depending on whether we wish to read data from the chip or write data to the chip respectively. Since in this project we will be writing to the MCP23017, the low bit should be 0, making the chip byte address (also called the device opcode) 0x40.

| A2 | A1 | A0 | Address |
|---|---|---|---|
| 0 | 0 | 0 | 0x20 |
| 0 | 0 | 1 | 0x21 |
| 0 | 1 | 0 | 0x22 |
| 0 | 1 | 1 | 0x23 |
| 1 | 0 | 0 | 0x24 |
| 1 | 0 | 1 | 0x25 |
| 1 | 1 | 0 | 0x26 |
| 1 | 1 | 1 | 0x27 |

Table 7.2 Address selection of the MCP23017

The MCP23017 chip has 8 internal registers that can be configured for its operation. The device can either be operated in 16 or two 8-bit modes by configuring bit IOCON.BANK. On power-up, this bit is cleared which chooses the two 8-bit mode by default.

The I/O direction of the port pins is controlled with registers IODIRA (at address 0x00) and IODIRB (at address 0x01). Clearing a bit to 0 in these registers makes the corresponding port pin(s) as output(s). Similarly, setting a bit to 1 in these registers makes the corresponding port pin(s) input(s). GPIOA and GPIOB register addresses are 0x12 and 0x13 respectively. This is shown in Figure 7.5.

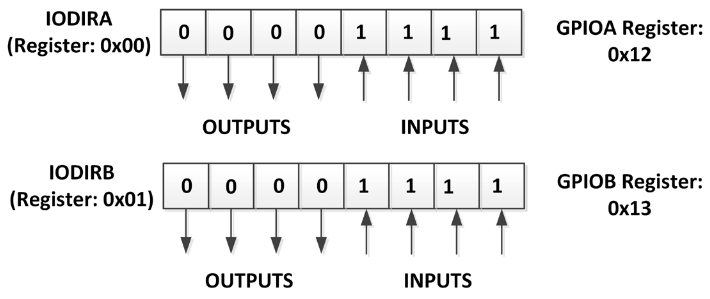


Figure 7.5 Configuring the I/O ports

Figure 7.6 shows the circuit diagram of the project. Notice the I2C pins of the port expander are connected to pins P0_17 (SDA) and P0_16 (I2C0 SCL) of the MAX78000FTHR development board and are pulled up using 10K resistors as required by the I2C specifications. The LED is connected to port pin GPA0 of the MCP23017 (pin 21) through a current limiting resistor.

The address select bits of the MCP23017 are all connected to ground.

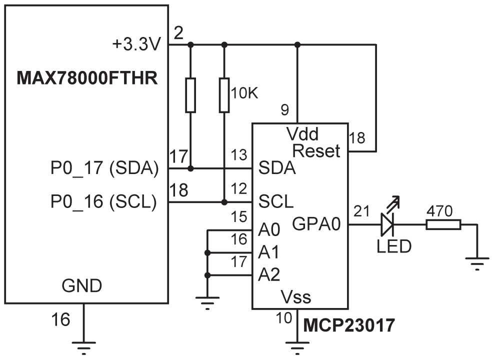


Figure 7.6 Circuit diagram of the project

More information on the MCP23017 chip can be obtained from the datasheet:

http://docs-europe.electrocomponents.com/webdocs/137e/0900766b8137eed4.pdf

**Program listing**: Figure 7.7 shows the program listing (**MCP23017**). At the beginning of the program, the MCP23017 device address and the registers used are defined. The I2C device address when A0, A1, and A2 pins are connected to GND is 0x20. The device address when A2 = A1 = 0 and A0 = 1 is 0x22 and so on. Up to 8 MCP23017 devices can be connected to the MAX78000FTHR, thus adding up to 8 x 16 = 128 additional GPIO ports. The I2C bus speed is set to 1MHz (the MCP23017 data sheet specifies that it can be from 100 kHz to up to 1.7 MHz):

```
#define MCP_SLAVE_ADDR 0x20                  // MCP23017 I2C address
#define MCP_GPIOA_REG 0x12                   // MCP23017 GPIOA address
#define MCP_IODIRA_REG 0                     // MCP23017 IODIRA Address
#define I2C_MASTER MXC_I2C1                  // Use I2C1
unsigned int hz= 1000000;                    // I2C frequency
```

The MAX78000 microcontroller is set as the bus MASTER, using I2C1 (P0_16 and P0_17):

```
       MXC_I2C_Init(I2C_MASTER, 1, 0);
```

**mxc_i2C_req_t** structure is then initialised with the following settings:

```
reqMaster.i2c = I2C_MASTER;                  // We are the MAster
reqMaster.addr = MCP_SLAVE_ADDR;             // Slave address
reqMaster.tx_buf = txdata;                   // Transmit buffer
reqMaster.tx_len = 2;                        // 2 bytes to transfer
reqMaster.rx_buf = rxdata;
reqMaster.rx_len = 0;                          // No receive data
reqMaster.restart = 0;
reqMaster.callback = 0;                        // No callback
```

The transmit buffer length is set to 2 since two bytes are sent to the slave on each transaction: the register address, followed by the data. Notice that setting the receive buffer length to 0 disables receiving data from the I2C bus which is the case in this example project. Also, this disables the **callback**.

Register IODIRA is set to 0 so that the GPA ports are all configured as outputs:

```
txdata[0]=MCP_IODIRA_REG;                              // IODIRA register
txdata[1]=0;                                           // Set to outputs
MXC_I2C_MasterTransaction(&reqMaster);
```

Port pin GPA0 is then flashed every 500 ms by setting it to 1, waiting for 500 ms, setting it to 0, and waiting for 500 ms again.

```
/*--------------------------------------------------------------------------
                I2C PORT EXPANDER

In this project an I2C compatible port expander chip (MCP23017) is connected
to the MAX78000FTHR. An LED is connected to the port expander. The program
flashes the LED every 0.5 second

Author: Dogan Ibrahim
Date  : March 2021
Work  : MCP23017
---------------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include <mxc.h>
#include "i2c_regs.h"
#include "i2c.h"

#define MCP_SLAVE_ADDR 0x20                              // MCP23017 I2C address
#define MCP_GPIOA_REG 0x12                               // MCP23017 GPIOA address
#define MCP_IODIRA_REG 0                                 // MCP23017 IODIRA Address
#define I2C_MASTER MXC_I2C1                              // Use I2C1
unsigned int hz= 1000000;                                // I2C frequency
unsigned char txdata[10];                                // Transmit buffer
unsigned char rxdata[10];                                // Receive buffer (unused)
```

```
mxc_i2c_req_t reqMaster;

/******* Main program *******/
int main(void)
{
        MXC_I2C_Init(I2C_MASTER, 1, 0);                // Initialize as MASTER
        MXC_I2C_SetFrequency (I2C_MASTER,hz);          // Set frequency

        reqMaster.i2c = I2C_MASTER;                    // We are the MAster
        reqMaster.addr = MCP_SLAVE_ADDR;               // Slave address
        reqMaster.tx_buf = txdata;                     // Transmit buffer
        reqMaster.tx_len = 2;                          // 2 bytes to tranfer
        reqMaster.rx_buf = rxdata;
        reqMaster.rx_len = 0;                          // No receive data
        reqMaster.restart = 0;
        reqMaster.callback = 0;                        // No callback

        txdata[0]=MCP_IODIRA_REG;                      // IODIRA register
        txdata[1]=0;                                   // Set to outputs
        MXC_I2C_MasterTransaction(&reqMaster);         // Send to slave

        while(1)                                       // Do forever
        {
            txdata[0]=MCP_GPIOA_REG;                   // GPIOA register
            txdata[1]=1;                               // Set GPA0 to 1
            MXC_I2C_MasterTransaction(&reqMaster);     // Send to slave
            MXC_Delay(500000);                         // WaIt 0.5 second

            txdata[0]=MCP_GPIOA_REG;                   // GOIOA register
            txdata[1]=0;                               // SEt GPA0 to 0
            MXC_I2C_MasterTransaction(&reqMaster);     // Send to slave
            MXC_Delay(500000);                         // Wait 0.5 second
         }
}
```

Figure 7.7 Program: MCP23017

## 7.5 • Project 2 – TMP102 temperature sensor

**Description**: In this project, the I2C compatible TMP102 temperature sensor chip is used. Ambient temperature is read every second and is sent and displayed on a terminal.

**Aim**: This project aims to show how a TMP102temperature sensor chip can be used in a program.

## **The TMP102**

The TMP102 is a highly accurate I2C compatible temperature sensor chip with a built-in thermostat, having the following basic features:

Supply voltage: 1.4V to 3.6V
Supply current: 10µA
Accuracy: ±0.5ºC
Resolution: 12 bits (0.0625ºC)
Operating range: -40ºC to +125ºC

The TMP102 is a 6-pin chip as shown in Figure 7.8. the pin descriptions are:

| Pin | Name | Description |
|---|---|---|
| 1 | SCL | I2C line |
| 2 | GND | power supply ground |
| 3 | ALERT | Over temperature alert. Open-drain output. requires a pull-up resistor |
| 4 | ADD0 | Address select |
| 5 | V+ | power supply |
| 6 | SDA | I2C line |

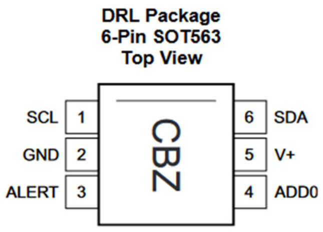


Figure 7.8 TMP102 pin layout

The TMP102 has the following operational modes:

- **Continuous conversion**: by default, an internal ADC converts the temperature into digital format with the default conversion rate of 4Hz, with a conversion time of 26ms. The conversion rate can be selected using bits **CR1** and **CR0** of the configuration register as: 0.25Hz, 1Hz, 4Hz (default), and 8Hz. In this project, the default 4Hz is used.
- **Extended mode**: Bit EM of the configuration register selects normal mode (EM = 0), or extended mode (EM = 1). In normal mode (default mode) the converted data is 12 bits. Extended mode is used if the temperature is above 128ºC and the converted data is 13 bits. In this project, normal mode is used.

- **Shutdown mode**: This mode is used to save power where the current consumption is reduced to less than 0.5µA. Shutdown mode is entered when configuration register bit SD = 1. The default mode is normal operation (SD = 0).
- **One-shot conversion**: Setting configuration register bit OS to 1 selects the one-shot mode which is a single conversion mode. The default mode is continuous conversion (OS = 0).
- **Thermostat mode**: This mode indicates whether to operate in comparator mode (TM = 0) or in interrupt mode (TM = 1). The default is comparator mode. In comparator mode, the Alert pin is activated when the temperature equals or exceeds the value in the THIGH register, and remains active until the temperature drops below $T_{LOW}$. In interrupt mode, the Alert pin is activated when the temperature exceeds $T_{HIGH}$ or goes below TLOW registers. The Alert pin is cleared when the host controller reads the temperature register.

A **Pointer Register** selects various registers in the chip as shown in Table 7.3. The upper 6 bits of this register are 0s.

| **P1** | **P0** | **Register Selected** |
|---|---|---|
| 0 | 0 | Temperature register (read only) |
| 0 | 1 | Configuration register |
| 1 | 0 | $T_{LOW}$ register |
| 1 | 1 | $T_{HIGH}$ register |

Table 7.3 Pointer register bits

Table 7.4 shows the temperature register bits in normal mode (EM = 0).

| **BYTE 1** | | | | | | | |
|---|---|---|---|---|---|---|---|
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
| T11 | T10 | T9 | T8 | T7 | T7 | T5 | T4 |

| **BYTE 2** | | | | | | | |
|---|---|---|---|---|---|---|---|
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
| T3 | T2 | T1 | T0 | 0 | 0 | 0 | 0 |

Table 7.4 Temperature register bits

Table 7.5 shows the configuration register bits. The power-up default bit configuration is shown in the table.

| **BYTE 1:** | | | | | | | |
|---|---|---|---|---|---|---|---|
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
| OS | R1 | R0 | F1 | F0 | POL | TM | SD |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

| **BYTE 2:** | | | | | | | |
|---|---|---|---|---|---|---|---|
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
| CR1 | CR0 | AL | EM | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

Table 7.5 Configuration register bits

The **Polarity** bit (POL) allows the user to adjust the polarity of the Alert pin output. If set to 0 (default), the Alert pin becomes active low. When set to 1, the Alert pin becomes active high.

The default device address is 0x48. TMP102 is available as a module (breakout module) as shown in Figure 7.9. The temperature register address is 0x00 and this should be sent after sending the device address. This is then followed by a read command where 2 bytes are read from the TMP102. These 2 bytes contain the temperature data.

The temperature read sequence is as follows:

- Master sends the device address 0x48 with the R/W set to 0
- Device responds with ACK
- Master sends the temperature register address 0x00
- Device responds with ACK
- Master re-sends device address 0x48 with the R/W bit set to 1
- Master reads upper byte of temperature data
- Device sends ACK
- Master reads lower byte of temperature data
- Device sends ACK
- Master sends stop condition on the bus

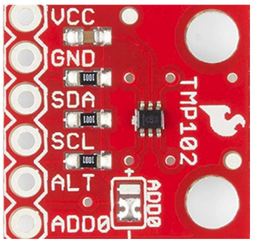


Figure 7.9 TMP102 as a module

**Block diagram**: Figure 7.10 shows the block diagram of the project.

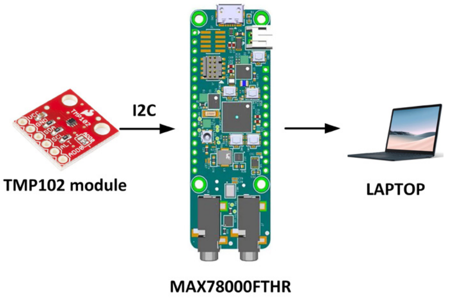


Figure 7.10 Block diagram of the project

**Circuit diagram**: The circuit diagram of the project is shown in Figure 7.11. On-chip pull-up resistors are available on the TMP102 I2C bus lines and therefore there is no need to use external pull-up resistors.

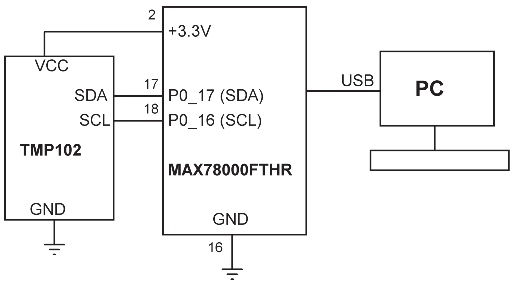


Figure 7.11 Circuit diagram of the project

**Program listing**: Figure 7.12 shows the program listing (**TMP102**). At the beginning of the program, the I2C address of TMP102 and the Pointer register addresses are defined. The Pointer register is set to 0 to select the temperature register.

The transmit buffer length **tx_len** is set to 1 and it is loaded initially with 0 so that a byte is sent to the TMP102 chip to configure it to send the temperature readings. The receive buffer **rxdata** length **rx_len** is set to 2 so that it receives the two bytes of temperature data.

The program runs inside a **while** loop every second. Two bytes of the temperature data are read and stored in **rxdata[0]** and **rxdata[1]**. The temperature is then converted into positive (or negative) degrees Celsius and stored in variable **temperature**. If the temperature is negative, it is in 2's complement form and its complement is taken and 1 is added to find the true negative value. By multiplying **temp** with the LSB we find the temperature in degrees Centigrade. The temperature reading is sent to the PC as a floating-point number using the **printf** statement.

Table 7.6 shows the data output format of the temperature. Let us look at two examples:

**Example 1**: Measured value = 0011 00100000 = 0x320 = 800 decimal

This is a positive temperature, so the temperature is 800 x 0.0625 = +50ºC

**Example 2**: Measured value = 1110 01110000 = 0xE70

This is a negative temperature, complement is 0001 10001111, adding 1 gives 0001 10010000 = 400 decimal. The temperature is 400 x 0.0625 = 25 or, -25ºC.

| **Temperature** | **Digital Output (Binary)** | **Digital Output (HEX)** |
|---|---|---|
| 128 | 011111111111 | 7FF |
| 100 | 011001000000 | 640 |
| 50 | 001100100000 | 320 |
| 0.25 | 000000000100 | 004 |
| -0.25 | 111111111100 | FFC |
| -25 | 111001110000 | E70 |
| -55 | 110010010000 | C90 |

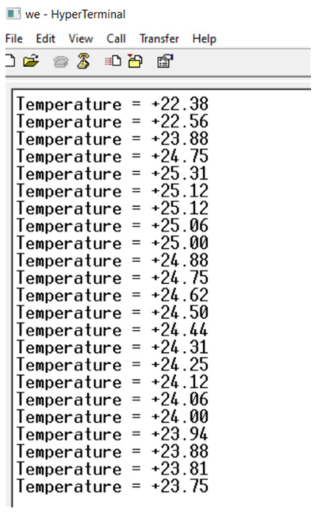
Table 7.6 The data output for some temperature readings

```
/*--------------------------------------------------------------------------
        I2C TMP102 TEMPERATURE SENSOR

In this project an I2C compatible TMP102 temperature sensor chip is connected
to the MAX78000FTHR. The program sends the temperature readings to a terminal
over serial link

Author: Dogan Ibrahim
Date  : March 2021
Work  : TMP102
---------------------------------------------------------------------------*/
/***** Includes *****/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include <mxc.h>
#include "i2c_regs.h"
#include "nvic_table.h"
#include "i2c.h"
#include "string.h"
```

```
#define TMP102_PointerReg 0                              //TMP102 register
#define TMP102_SLAVE_ADDR 0x48                           // MCP23017 I2C address
#define I2C_MASTER MXC_I2C1                              // Use I2C1
unsigned int hz= 1000000;                                // I2C frequency
unsigned char rxdata[10];                                // Receive buffer (unused)
unsigned char txdata[10];                                // Transmit buffer
volatile int I2C_FLAG;
volatile int flag=0;
mxc_i2c_req_t reqMaster;

void I2C1_IRQHandler(void)
{
    MXC_I2C_AsyncHandler(I2C_MASTER);
    return;
}

//
//I2C callback function
//
void I2C_Callback(mxc_i2c_req_t* req, int error)
{
    I2C_FLAG = error;
    return;
}

/******* Main program *******/
int main(void)
{
        float temperature;
        int temp;
        float LSB = 0.0625;
        MXC_I2C_Init(I2C_MASTER, 1, 0);                  // Initialize as MASTER
        MXC_I2C_SetFrequency (I2C_MASTER,hz);            // Set frequency

        NVIC_SetVector(I2C1_IRQn, I2C1_IRQHandler);
        NVIC_EnableIRQ(I2C1_IRQn);

        reqMaster.i2c = I2C_MASTER;                      // We are the MAster
        reqMaster.addr = TMP102_SLAVE_ADDR;              // Slave address
        reqMaster.tx_buf = txdata;                       // Transmit buffer
        reqMaster.tx_len = 1;                            // 1 byte to tranfer
        reqMaster.rx_buf = rxdata;                       // Receive buffer
        reqMaster.rx_len = 2;                            // Receive 2 bytes
        reqMaster.restart = 0;
        reqMaster.callback=I2C_Callback;                 // Callback routine
        I2C_FLAG = 1;
        txdata[0]=0;                                 // Prepare to send 0
        MXC_I2C_MasterTransactionAsync(&reqMaster);  // Configure to read temp
        reqMaster.tx_len=0;                          // No more to trenamit

        while(1)                                     // Do forever
        {
                MXC_I2C_MasterTransaction(&reqMaster);
                while(I2C_FLAG == 1) {};             // Wait for completion

            temp = (rxdata[0] << 4) | (rxdata[1] >> 4); // Extract temperature
            if(temp > 0x7FF)
            {
                temp = (~temp) & 0xFF;               // If negative temperature
                temp = temp + 1;
                temperature = -temp * LSB;
            }
            else
            {
                temperature = temp * LSB;
            }
                printf("Temperature = %+5.2f\n",temperature);   // Send readings
                MXC_Delay(1000000);                             // Wait 1 second

         }
}
```

Figure 7.12 Program listing

Example output from the program is shown in Figure 7.13.



```
Temperature = +22.38
Temperature = +22.56
Temperature = +23.88
Temperature = +24.75
Temperature = +25.31
Temperature = +25.12
Temperature = +25.12
Temperature = +25.06
Temperature = +25.00
Temperature = +24.88
Temperature = +24.75
Temperature = +24.62
Temperature = +24.50
Temperature = +24.44
Temperature = +24.31
Temperature = +24.25
Temperature = +24.12
Temperature = +24.06
Temperature = +24.00
Temperature = +23.94
Temperature = +23.88
Temperature = +23.81
Temperature = +23.75
```

Figure 7.13 Example output from the program

# Chapter 8 • SPI Bus Interface

## 8.1 • Overview

In this chapter, we will develop projects using the SPI bus (Serial Peripheral Interface) with the MAX78000FTHR development board. The SPI bus is one of the commonly used protocols to connect sensors and many other devices to microcontrollers. The SPI bus is a master-slave type bus protocol. In this protocol, one device (the microcontroller) is designated as the master, and one or more other devices (usually sensors) are designated as slaves. In a minimum bus configuration, there is one master and only one slave. The master establishes communication with the slaves and controls all the activity on the bus. Figure 8.1 shows an SPI bus example with one master and 3 slaves. The SPI bus uses 3 signals: clock (SCK), data in (SDI, or RX), and data out (SDO, or TX). The SDO of the master is connected to the SDIs of the slaves. SDOs of the slaves are connected to the SDI of the master. The master generates SCK signals to enable data to be transferred on the bus. In every clock pulse, one bit of data is moved from master to slave, or from slave to master. The communication is only between a master and a slave. Slaves cannot communicate with each other. It is important to note that only one slave can be active at any time since there are no mechanisms to identify the slaves. Thus, slave devices have enable lines (e.g. CS or CE) which are normally controlled by the master. Typical communication between a master and several slaves is as follows:

- Master enables slave 1
- Master sends SCK signals to read or write data to slave 1
- Master disables slave 1 and enables slave 2
- Master sends SCK signals to read or write data to slave 2
- The above process continues as required

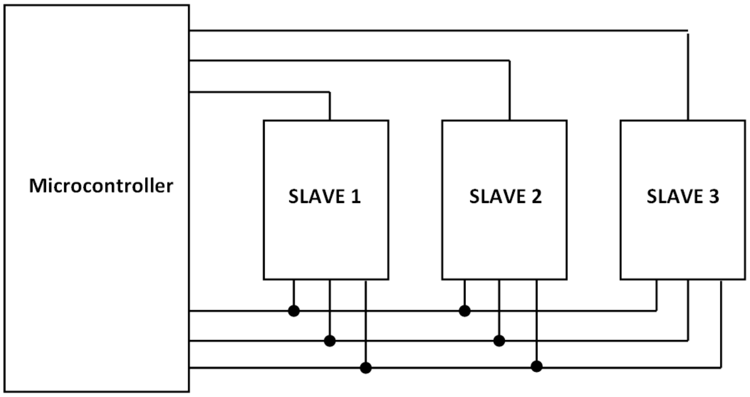


Figure 8.1 SPI bus with one master and 3 slaves

The SPI signal names are also called MISO (Master in, Slave out), and MOSI (Master out, Slave in). Clock signal SCK is also called SCLK and the CS is also called SSEL. In the SPI projects in this chapter, the MAX78000FTHR is the master and one or more slaves are connected to the bus. Transactions over the SPI bus are started by enabling the SCK line. The master then asserts the SSEL line LOW so data transmission can begin. Data transmission involves two registers, one in the master and one in the slave device. Data is shifted out from the master into the slave with the MSB bit first. If more data is to be transferred, the process is repeated. Data exchange is complete when the master stops sending clock pulses and deselects the slave device.

Both the master and slave must agree on clock polarity and phase on the line, which are known as the SPI bus modes. These two settings are named Clock Polarity (CPOL) and Clock Phase (CPHA) respectively. CPOL and CPHA can have the following values:

| **CPOL** | **Clock active state** |
|---|---|
| 0 | Clock active HIGH |
| 1 | Clock active LOW |

| **CPHA** | **Clock phase** |
|---|---|
| 0 | Clock out of phase with data |
| 1 | Clock in phase with data |

The four SPI modes are:

| **Mode** | **CPOL** | **CPHA** |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 1 | 0 |
| 3 | 1 | 1 |

When CPOL = 0, the active state of the clock is 1, and its idle state is 0. For CPHA = 0, data is captured on the rising clock, and data is shifted out on the falling clock. For CPHA = 1, data is captured on the falling edge of the clock and is shifted out on the rising edge of the clock.

When CPOL = 1, the active state of the clock is 0, and its idle state is 1. For CPHA = 0, data is captured on the falling edge of the clock and is output on the rising edge. For CPHA = 1, data is captured on the rising edge of the clock and is shifted out on the falling edge.

## 8.2 ● MAX78000 microcontroller SPI ports

There are 2 SPI ports on the MAX78000 microcontroller: SPI0 and SPI1. Only the SPI0 pins are available on the MAX78000FTHR development board: MOSI at P0_5 (pin 12), MISO at P0_6 (pin 13), and SCK at P0_7 (pin 11). These pins are shared with microSD pins at P0_8, P0_9, and P0_11.

## 8.3 ● Project 1 – SPI send/receive

**Description**: In this project, the MOSI and MISO pins of the MAX78000FTHR development board are connected. Numbers **0123456789** are sent to the SPI bus through port MOSI. This data is received by the MISO port and then sent to the PC where it is displayed.

**Aim**: This project aims to show how the SPI ports of the MAX78000FTHR development board can be used.

**Block diagram**: Figure 8.2 shows the block diagram of the project.

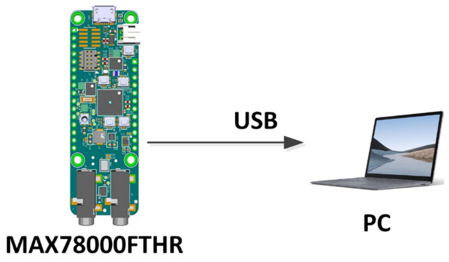


Figure 8.2 Block diagram of the project

**Circuit diagram**: Figure 8.3 shows the circuit diagram of the project. MOSI (P0_5) pin is connected to the MISO (P0_6).

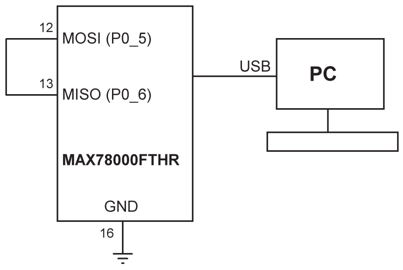


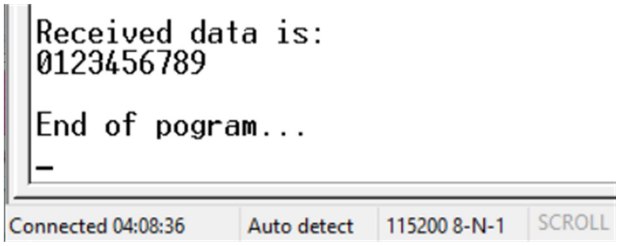
Figure 8.3 Circuit diagram of the project

**Program listing**: Figure 8.4 shows the program listing (**SPISNDRCV**). At the beginning of the program, SPI0 is defined, data length is set to 10 bytes, and the SPI bus speed is set to 1MHz. Inside the main program, the SPI pins are configured, data (**0123456789**) to be transmitted is stored in array **txdata**. The receive data **rxdata** is cleared to 0s. Then, the SPI request structure is configured with the transmit and receive data length set to **DATA_ LENGTH**, character size is set to 8bits/character. Function **MXC_SPI_MasterTransaction** is then called in blocking mode. The received data is sent to a terminal using the **printf** statement.

```
/*--------------------------------------------------------------------------
        SPI SEND/RECEIVE PROGRAM

In this program the MOSI (SPI output) of the MAX78000FTHR is connected to
its MISO (SPI input). The program sends out numbers 0123456789, which are
received and displayed on a terminal

Author: Dogan Ibrahim
Date  : March 2021
Work  : SPISNDRCV
---------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include "mxc_device.h"
#include "mxc_delay.h"
#include "mxc_pins.h"
#include "spi.h"
#include "board.h"
#include "stdlib.h"

#define SPI             MXC_SPI0                         // SPI port
#define DATA_LENGTH     10                               // Data length
#define SPI_SPEED       1000000                          // Bit Rate

char rxdata[DATA_LENGTH];                                // Receive data buffer
char txdata[DATA_LENGTH];                                // Transmit data buffer

//
// Start of MAIN program
//
int main(void)
{
    int j;
    mxc_spi_req_t req;
    mxc_spi_pins_t spi_pins;

//
// Configure the SPI pins
//
    spi_pins.clock = TRUE;
    spi_pins.miso = TRUE;
    spi_pins.mosi = TRUE;
    spi_pins.sdio2 = FALSE;
    spi_pins.sdio3 = FALSE;
    spi_pins.ss0 = TRUE;
    spi_pins.ss1 = FALSE;
    spi_pins.ss2 = FALSE;
//
// Convert j to character and store in txdata
//
    for (j = 0; j < DATA_LENGTH; j++)
    {
        itoa(j, &txdata[j], 10);                        // Load the TX buffer
    }

    MXC_SPI_Init(SPI, 1, 0, 1, 0, SPI_SPEED, spi_pins); //Init SPI
    memset(rxdata, 0x0, DATA_LENGTH);                   // Fill rxdata with 0s

//
// SPI Request structure
//
    req.spi = SPI;
    req.txData = (uint8_t*) txdata;
    req.rxData = (uint8_t*) rxdata;
    req.txLen = DATA_LENGTH;
    req.rxLen = DATA_LENGTH;
    req.ssIdx = 0;
    req.ssDeassert = 1;
    req.txCnt = 0;
    req.rxCnt = 0;
    req.completeCB = 0;

    MXC_SPI_SetDataSize(SPI, 8);                        // 8 bits/char
    MXC_SPI_SetWidth(SPI, SPI_WIDTH_STANDARD);          // Standard comms

    MXC_SPI_MasterTransaction(&req);                    // Master transaction

    printf("\nReceived data is:\n");                    // DIsplay received data
    for(j = 0; j < DATA_LENGTH; j++)
        printf("%c",rxdata[j]);

    MXC_SPI_Shutdown(SPI);                              // Shutdown SPI

    printf("\n\nEnd of pogram...\n");
}
```

Figure 8.4 Program listing

Figure 8.5 shows the received data displayed on a terminal. In this figure, **HyperTrm** terminal emulation software is used (you could use any other terminal emulation software if you wish, e.g. **Putty**, **Terra Term**, etc). The Baud rate is set to 115200.



```
Received data is:
0123456789

End of pogram...
_
```

Figure 8.5 Received data on the PC terminal

# Chapter 9 • Timers

## 9.1 • Overview

The MAX78000 microcontroller provides a large number of general-purpose timers with rich features. The timers can be used for timing, capture, compare, or for the generation of pulse-width-modulated (PWM) signals with minimal software interaction.

There are multiple 32-bit and dual 16-bit reloadable timers with the features including:

- Operation as a single 32-bit counter or single/dual 16-bit counters
- Programmable pre-scalers with values from 1 to 4096
- PWM output generation
- Capture, compare and capture/compare capability
- Configurable input pin for event triggering, clock gating, or capture signal
- Independent interrupts

The timers provide multiple operating modes:

- One-shot - counts up to a terminal value and halts
- Continuous – timer counts up to a terminal value and then repeats
- Counter – counts input edges on the input pin
- PWM output
- Capture – captures a snapshot of the current timer count on edge transitions
- Compare – the timer pin toggles when the timer exceeds the terminal count
- Gated – timer increments only when timer input is asserted
- Capture/compare – timer counts when timer input is asserted, captures timer count when input is de-asserted

The MAX78000 microcontroller provides 4 normal power timers (TMR0, TMR1, TR2, TMR3) and 2 low-power timers (LPTMR0 or TMR4 and LPTMR1 or TMR5). Normal power timers are cascadable to 32-bits or can be operated in dual 16-bit modes. Low power timers can only operate in 16-bit mode and are not cascadable.

Table 9.1 shows a list of the available timers with their output pin configurations. The terminologies **TimerA** and **TimerB** are used to differentiate the organisation of the 32-bit registers. Notice **N** as the last character in the signal names refers to complementary outputs.

| Instance | Register Access Name | TMRn_IOA | TMRn_IOB | TMRn_IOAN | TMRn_IOBN |
|---|---|---|---|---|---|
| TMR0 | TMR0 | TMR0A__IOA (P0.2)<br>TMR0B__IOA (P0.8) | TMR0A_IOB (P0.3)<br>TMR0B__IOB (P0.9) | TMR0B_IOAN (P0.4) | TMR0B_IOAN (P0.5) |
| TMR1 | TMR1 | TMR1A_IOA (P0.14) | TMR1A_IOB (P0.15) | TMR1B_IOAN (P0.12) | TMR1B_IOBN (P0.13) |
| TMR2 | TMR2 | TMR2A_IOA (P0.26) | TMR2A_IOB (P0.27) | - | - |
| TMR3 | TMR3 | TMR3B_IOA (P1.4)<br>TMR3B_IOA (P1.6) | TMR3A_IOB (P1.5)<br>TMR3A_IOB (P1.7) | - | - |
| LPTMR0 | TMR4 | LPTMR0B_IOA (P2.4) | - | - | - |
| LPTMR1 | TMR5 | LPTMR1_IOA (P2.5) | - | - | - |

Table 9.1 Timers and pin configurations

## 9.2 ● Timer operation

Timers operate by incrementing a readable timer counter register, driven by either the timer clock, an external stimulus on the timer pin, or a combination of both. Each timer has a user-configurable timer period which terminates on the timer clock cycle following the end of timer period condition. At the end of a timer period, a timer can:

- Change the state of the timer pin
- Capture a timer value
- Reload the counter register with a new starting value
- Disable the counter

A timer interrupt is generated at the end of a timer period. Although the timer counter is set to 0 after a reset, it is set to 1 at the end of a timer period.

## 9.3 ● 32-bit single/cascade and dual 16-bit modes

In cascade 32-bit mode, the timer counts [31:0] using the TimerA control registers. In dual 16-bit mode, the timer counts [15:0] using the TimerA control registers for the lower 16-bit counter, and counts [31:16] using the TimerB control registers for the upper 16-bit counter. In single 16-bit mode, the timer counts [15:0] using the TimerA control registers while the TimerB controls are ignored.

Some projects are given in the remaining parts of this chapter to show how timers can be used.

## 9.4 ● Project 1 – Time delay – using a one-shot timer (monostable)

**Description**: In this project, a timer is configured to operate as a one-shot (sometimes called a monostable). The onboard LED and button are used in this project. When the button is pressed, the LED is toggled after a given time. In this case, the time is taken as 10 seconds. For example, if the LED is OFF, it turns ON after 10 seconds of pressing the button. Similarly, if the LED is ON, it turns OFF after 10 seconds of pressing the button.

**Aim**: This project aims to show how a one-shot timer can be created on a MAX78000FTHR development board.

**Circuit diagram**: The onboard RED LED on port P2_0 and the button SW1 on port P0_2 are used in this program. The button output is debounced by the onboard hardware.
Program listing: Figure 9.1 shows the program listing (**ONESHOT**). At the beginning of the program TMR0 is used, and the clock source 32K (32768 Hz) is selected:

```
        #define OST_TIMER              MXC_TMR0            // Using TMR0
        #define OST_CLOCK_SOURCE    MXC_TMR_32K_CLK        // Timer clock
```

The on-board RED LED and the on-board SW1 button are defined as PIN_0 and PIN_2 respectively:

```
        #define RedLED MXC_GPIO_PIN_0                // On-board RED LED
        #define Button MXC_GPIO_PIN_2                // On board SW1 button
```

Inside the main program the TMR0 interrupts are enabled, and the LED and the button are configured as output and input respectively. The pull-up resistor is enabled at the input of the button (i.e. at port P0_2). The remainder of the main program consists of a **while** loop. Inside this loop, the program waits until button SW1 is pressed (i.e. until its output is at logic 0). When this happens, function **OneShot** is called:

```
while(1)
        {
                if(MXC_GPIO_InGet(gpio_in.port, Button) == 0)
                {
                        OneshotTimer();
                }
        }
```

Function **Oneshot** configures and starts the timer so that an interrupt is generated at the end of the timer period. This function performs the following operations:

1. Get the period count for the required frequency
2. Disable the timer
3. Set the prescaler value
4. Configure the timer for one-shot mode
5. Set polarity, timer parameters
6. Initialise timer clock
7. Enable timer
8. Start timer

The timer counts up until it reaches the specified compare register value (**cmp_cnt**) and it then generates a timer interrupt. At this point, it is automatically disabled and stops. In this program, the timer interrupt service routine is the function **OneshotTimerHandler** which toggles the LED every time it is called:

```
void OneshotTimerHandler()
{
       MXC_TMR_ClearFlags(OST_TIMER);      // Clear timer interrupt
       MXC_GPIO_OutToggle(gpio_out.port, RedLED);
}
```

Timer parameters are set using the structure: **mxc_tmr_cfg_t**. The following parameters are configured:

```
    tmr.pres = TMR_PRES_1;                   // Prescaler
    tmr.mode = TMR_MODE_ONESHOT;             // Mode=One shot
    tmr.bitMode = TMR_BIT_MODE_32;           // 32-bit
    tmr.clock = OST_CLOCK_SOURCE;            // Clock source
    tmr.cmp_cnt = periodTicks;               // Compare register value
    tmr.pol = 0;                             // Passive Polarity
```

The **Prescaler** value can only take the values of the powers of 2. i.e. valid prescaler values are 1, 2, 4, 8, 16, 32, 64. 128. 256, 512, 1024, 2048, 4096.

The timer mode is set to one shot with **tmr.mode = TMR_MODE_ONESHOT**

The timer bit mode is set to 32-bits using the statement **tmr.bitMode = TMR_BIT_MODE_32**. Valid timer bit modes are:

| | |
|---|---|
| TMR_BIT_MODE_32 | // Timer mode 32 bit |
| TMR_BIT_MODE_16A | // Timer mode lower 16 bit |
| TMR_BIT_MODE_16B | // Timer mode upper 16 bit |

The timer clock is set to **OST_CLOCK_SOURCE** which was defined as 32K (i.e. 32768 Hz) at the beginning of the program. i.e. `#define OST_CLOCK_SOURCE MXC_TMR_32K_CLK`. Different timers can have different clock rates as follows:

| | |
|---|---|
| Timers 0, 1, 2, and 3: | 8M and 60M |
| Timers 0, 1, 2, 3, and 4: | 32K |
| Timers 4 and 5: | 8K |

Timer frequencies are identified with the following names:

| | |
|---|---|
| 60M: | MXC_TMR_60M_CLK |
| 8M: | MXC_TMR_8M_CLK |
| 32K: | MXC_TMR_32K_CLK |
| 8K: | MXC_TMR_8K_CLK |

Timer polarity is the default logic value of the timer output. At the end of the timer period, the timer will be automatically disabled and the logic state of the timer output pin will generate a pulse with a duration equal to the timer clock period.

```
/*--------------------------------------------------------------------------
                ONESHOT TIME DELAY

In this project the on-board button and LED are used. Pressing the button
turns ON the Red LED toggles the LED after 10 second.

Author: Dogan Ibrahim
Date  : March 2021
Work  : ONESHOT
---------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_sys.h"
#include "nvic_table.h"
#include "mxc.h"
#include "led.h"

//
// Parameters for one shot  timer
//
#define OST_TIMER           MXC_TMR0                    // Using TMR0
#define OST_CLOCK_SOURCE    MXC_TMR_32K_CLK             // Timer clock

#define RedLED MXC_GPIO_PIN_0                           // On-board RED LED
#define Button MXC_GPIO_PIN_2                           // On board SW1 button
mxc_tmr_cfg_t tmr;
mxc_gpio_cfg_t gpio_out;
mxc_gpio_cfg_t gpio_in;

//
// One shot timer handler. Toggle the LED
//
void OneshotTimerHandler()
{
    MXC_TMR_ClearFlags(OST_TIMER);                      // Clear timer interrupt
    MXC_GPIO_OutToggle(gpio_out.port, RedLED);          // Toggle LED
}

void OneshotTimer()
{
    MXC_TMR_Shutdown(OST_TIMER);                        // Stop the timer
//
// Set timer parameters
//
    tmr.pres = TMR_PRES_128;                    // Prescaler
    tmr.mode = TMR_MODE_ONESHOT;                // Mode=One shot
    tmr.bitMode = TMR_BIT_MODE_32;              // 32-bit
    tmr.clock = OST_CLOCK_SOURCE;               // Clock source
    tmr.cmp_cnt = 2561;                         // Compare register for 10 secs
    tmr.pol = 0;                                // Passive Polarity

    MXC_TMR_Init(OST_TIMER, &tmr, true);        // Initialize timer clock
    MXC_TMR_EnableInt(OST_TIMER);               // Enable timer
    MXC_TMR_Start(OST_TIMER);                   // Start timer
}


int main(void)
{
    NVIC_SetVector(TMR0_IRQn, OneshotTimerHandler);     // Timer interrupt handler
    NVIC_EnableIRQ(TMR0_IRQn);                          // Enable timer interrupts

    /* Setup output pin P2_0 */
    gpio_out.port = MXC_GPIO2;                          // Port 2
    gpio_out.mask = RedLED;                             // Pin 0
    gpio_out.pad = MXC_GPIO_PAD_NONE;                   // None
    gpio_out.func = MXC_GPIO_FUNC_OUT;                  // Output
    MXC_GPIO_Config(&gpio_out);

    /* Setup input pin P0_2 */
    gpio_in.port = MXC_GPIO0;                           // Port 0
    gpio_in.mask = Button;                              // Pin 2
    gpio_in.pad = MXC_GPIO_PAD_PULL_UP;                 // Pull-up
    gpio_in.func = MXC_GPIO_FUNC_IN;                    // Input
    MXC_GPIO_Config(&gpio_in);

        while(1)
        {
                if(MXC_GPIO_InGet(gpio_in.port, Button) == 0)   // Wait for button
                {
                        OneshotTimer();                         // Call OneshotTimer
                }
        }
}
```

Figure 9.1 Program listing

**Testing the program**: Compile and load the program to the MAX78000FTHR. Press button SW1. You should see the RED LED turning ON after 10 seconds. Press the button again, and this time the RED LED will turn OFF after 10 seconds.

Figure 9.2 shows one shot operation mode. Notice when the timer is enabled that the count is compared to the compare value (green horizontal line). When the count is equal to the compare value, the count stops, a timer interrupt is generated by the rising edge of **TMRn_INTFL.irq** (cleared by software). At the same time, a rising pulse is generated, shown as **TMRn_CTRL0.pol=0** if the polarity was set to 0, or a falling pulse if the polarity was set to 1. The width of this pulse is equal to the width of the clock period.

At the end of the one shot, the counter is loaded with 1 (not 0). The required delay in seconds depends on the clock frequency and the selected prescaler value. The value to be loaded into the comparator register **cmp_cnt** for a required delay can be calculated using the following formula:

**cmp_cnt = counter clock frequency (Hz) x Required delay in seconds / Prescaler + 1**

As an example, assuming that the clock frequency is 32K, and the selected prescaler value is 128, we have:

```
cmp_cnt = 32768 x 10 / 128 + 1 = 2561
```

Therefore,

```
tmr.pres = TMR_PRES_128;
tmr.cmp_cnt = 2561;
```

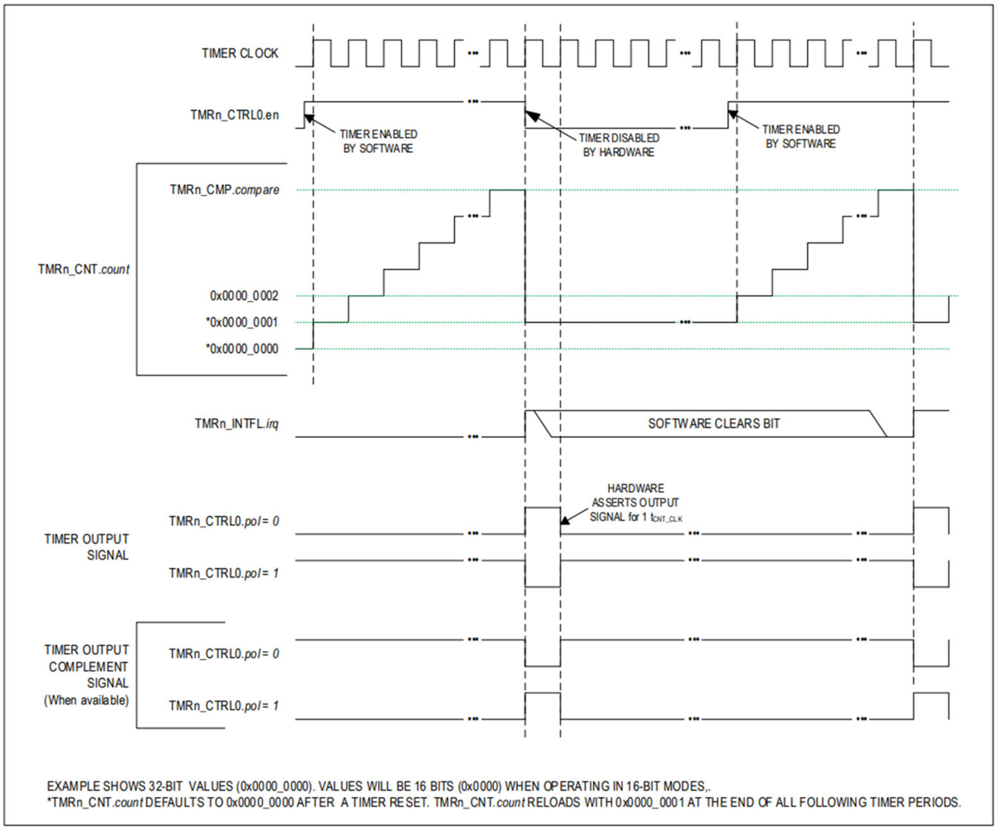

TIMER CLOCK
TMRn_CTRL0.en
TIMER ENABLED BY SOFTWARE
TIMER DISABLED BY HARDWARE
TIMER ENABLED BY SOFTWARE
TMRn_CMP.compare
TMRn_CNT.count
0x0000_0002
*0x0000_0001
*0x0000_0000
TMRn_INTFL.irq
SOFTWARE CLEARS BIT
HARDWARE ASSERTS OUTPUT SIGNAL for 1 tCNT_CLK
TIMER OUTPUT SIGNAL
TMRn_CTRL0.pol= 0
TMRn_CTRL0.pol= 1
TIMER OUTPUT COMPLEMENT SIGNAL (When available)
TMRn_CTRL0.pol= 0
TMRn_CTRL0.pol= 1
EXAMPLE SHOWS 32-BIT VALUES (0x0000_0000). VALUES WILL BE 16 BITS (0x0000) WHEN OPERATING IN 16-BIT MODES,.
*TMRn_CNT.count DEFAULTS TO 0x0000_0000 AFTER A TIMER RESET. TMRn_CNT.count RELOADS WITH 0x0000_0001 AT THE END OF ALL FOLLOWING TIMER PERIODS.

Figure 9.2 One shot mode

## 9.5 • Project 2 – Continuously running timer (astable)

**Description**: In this project, a timer is configured to operate continuously and generate interrupts every 2 seconds. The onboard LED is toggled in the interrupt service routine.

**Aim**: This project aims to show how a continuous timer can be created on a MAX78000FTHR development board.

**Circuit diagram**: The onboard RED LED at port P2_0 is used in this project. The LED flashes every 2 seconds.

**Program listing**: Figure 9.3 shows the program listing (**CONTTMR**). At the beginning of the program, TMR1 is used. The clock source 8M (8192 kHz) is selected:

```
#define CONT_TIMER          MXC_TMR1             // Use TMR1
#define CONT_CLOCK_SOURCE   MXC_TMR_8M_CLK       // Timer clock
```

Inside the main program, the onboard RED LED is configured as an output. Timer1 interrupts are configured and enabled. The interrupt service routine function is named **ContinuousTimerHandler**:

```
NVIC_SetVector(TMR1_IRQn, ContinuousTimerHandler);        // Timer interrupt
NVIC_EnableIRQ(TMR1_IRQn);                                // Enable interrupt
```

The operation of a timer in continuous mode is as follows. The counter starts counting up and when it reaches the compare register value (**cmp_cnt**), an interrupt is generated and the counter is reset to 1 and the counting continues. At the same time, a pulse is generated with the specified polarity at the end of every timer period.

The program then calls function ContinuousTimer where Timer1 is configured to operate continuously. The configuration of the timer in continuous mode is similar to the one shot mode. The timer is stopped and the following parameters are configured:

```
tmr.pres = TMR_PRES_128;                        // Prescaler
tmr.mode = TMR_MODE_CONTINUOUS;                 // Continuous
tmr.bitMode = TMR_BIT_MODE_16B;                 // Bit mode
tmr.clock = CONT_CLOCK_SOURCE;                  // Clock source
tmr.cmp_cnt = periodTicks;                      // Period ticks
tmr.pol = 0;
```

Time delay between the timer interrupts (i.e. the period of the timer interrupts) can be calculated using the following formula:

cmp_cnt = counter clock frequency (Hz) x Required delay between interrupts (secs) / Prescaler + 1

As an example, assuming the clock frequency is 8M (8192000 Hz), and the selected prescaler value is 1024, we have:

cmp_cnt = 8192000 x 2 / 1024 + 1 = 16001

Therefore,

```
tmr.pres = TMR_PRES_1024;
tmr.cmp_cnt = 16001;
```

The onboard RED LED is flashed inside the timer interrupt service routine. Notice that timer interrupts must be cleared inside the interrupt service routine so that further interrupts can be accepted by the processor:

```
void ContinuousTimerHandler()
{
    MXC_TMR_ClearFlags(CONT_TIMER);                         // Clear interrupt
    MXC_GPIO_OutToggle(gpio_out.port, RedLED);              // Toggle LED
}

/*---------------------------------------------------------------------------
        CONTINUOUS TIMER

In this project the on-board LED is flashed every 2 seconds using a
continuously running timer

Author: Dogan Ibrahim
Date  : March 2021
Work  : CONTTMR
---------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_sys.h"
#include "nvic_table.h"
#include "mxc.h"

//
// Parameters for Continuous timer
//
#define CONT_TIMER          MXC_TMR1                    // Use TMR1
#define CONT_CLOCK_SOURCE   MXC_TMR_8M_CLK              // Timer clock
#define RedLED MXC_GPIO_PIN_0                           // On-board RED LED

mxc_tmr_cfg_t tmr;
mxc_gpio_cfg_t gpio_out;

//
// Timer interrupt service routine. Toggle the LED
//
void ContinuousTimerHandler()
{
    MXC_TMR_ClearFlags(CONT_TIMER);                     // Clear interrupt
    MXC_GPIO_OutToggle(gpio_out.port, RedLED);          // Toggle LED
}

//
// COntinuous timer routine
//
void ContinuousTimer()
```

```
{
    MXC_TMR_Shutdown(CONT_TIMER);

    tmr.pres = TMR_PRES_1024;                            // Prescaler
    tmr.mode = TMR_MODE_CONTINUOUS;                      // Continuous
    tmr.bitMode = TMR_BIT_MODE_16B;                      // Bit mode
    tmr.clock = CONT_CLOCK_SOURCE;                       // Clock source
    tmr.cmp_cnt = 16001;                                 // Period ticks
    tmr.pol = 0;

    MXC_TMR_Init(CONT_TIMER, &tmr, true);                // Init timer
}


int main(void)
{
    /* Setup output pin P2_0 */
    gpio_out.port = MXC_GPIO2;                           // Port 2
    gpio_out.mask = RedLED;                              // Pin 0
    gpio_out.pad = MXC_GPIO_PAD_NONE;                    // None
    gpio_out.func = MXC_GPIO_FUNC_OUT;                   // Output
    MXC_GPIO_Config(&gpio_out);

    NVIC_SetVector(TMR1_IRQn, ContinuousTimerHandler);  // Timer interrupt
    NVIC_EnableIRQ(TMR1_IRQn);                           // Enable timer interrupt
    ContinuousTimer();                                   // Call timer routine
}
```

Figure 9.3 Program listing

Figure 9.4 shows the timer operating in continuous mode. Notice when the timer is enabled, the count is compared to the compare value (green horizontal line). When the count is equal to the compare value, the counter is reset to 1 and the counting continues. A timer interrupt is generated by the rising edge of **TMRn_INTFL.irq** (cleared by software). At the same time, a rising pulse is generated, shown as **TMRn_CTRL0.pol=0** if the polarity was set to 0, or a falling pulse if the polarity was set to 1. The width of this pulse is equal to the period of the timer cycle. i.e. 2 seconds in this example.

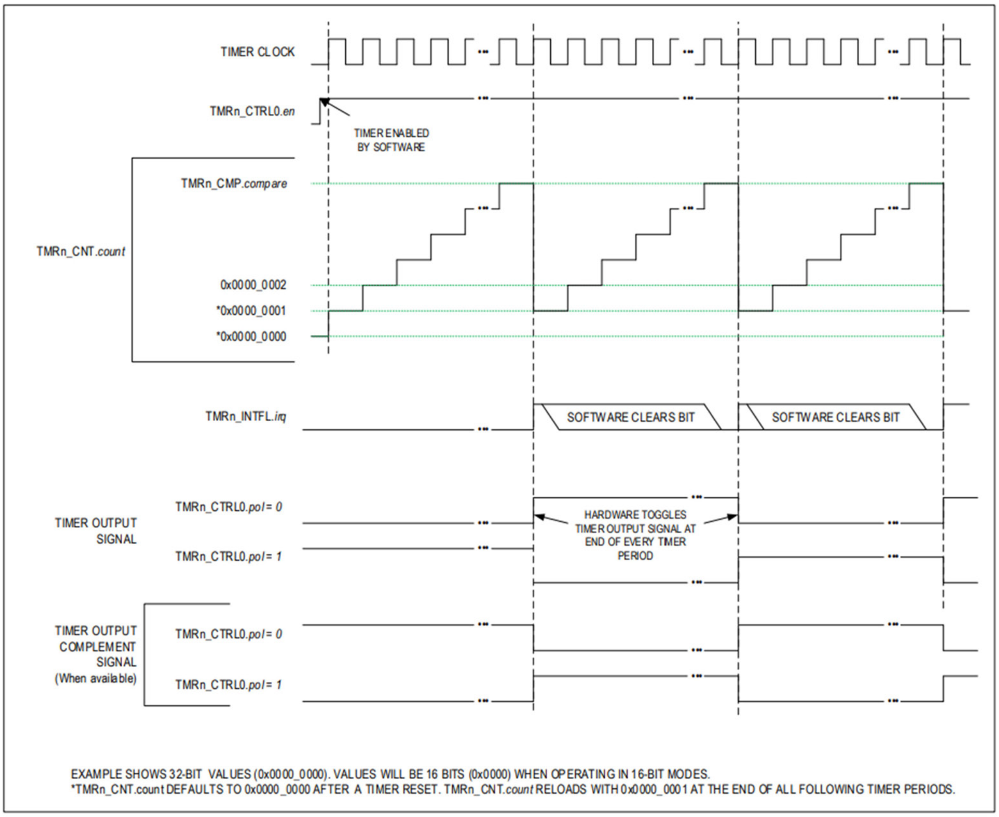

TIMER CLOCK
TMRn_CTRL0.en
TIMER ENABLED BY SOFTWARE
TMRn_CMP.compare
TMRn_CNT.count
0x0000_0002
*0x0000_0001
*0x0000_0000
TMRn_INTFL.irq
SOFTWARE CLEARS BIT
SOFTWARE CLEARS BIT
TIMER OUTPUT SIGNAL
TMRn_CTRL0.pol = 0
TMRn_CTRL0.pol = 1
HARDWARE TOGGLES TIMER OUTPUT SIGNAL AT END OF EVERY TIMER PERIOD
TIMER OUTPUT COMPLEMENT SIGNAL (When available)
TMRn_CTRL0.pol = 0
TMRn_CTRL0.pol = 1
EXAMPLE SHOWS 32-BIT VALUES (0x0000_0000). VALUES WILL BE 16 BITS (0x0000) WHEN OPERATING IN 16-BIT MODES.
*TMRn_CNT.count DEFAULTS TO 0x0000_0000 AFTER A TIMER RESET. TMRn_CNT.count RELOADS WITH 0x0000_0001 AT THE END OF ALL FOLLOWING TIMER PERIODS.

Figure 9.4 Continuous mode

## 9.6 • Project 3 – Refreshing a 2-digit 7-segment display – seconds counter

**Description**: In this project, a 7-segment LED display is used as a counter to count up every second from 0 to 99. Multi-digit 7-segment displays require continuous refreshing of their digits so that the human eye sees the digits as steady and non-flashing. The general technique used is to enable each digit for a short time (e.g. 10 ms) so that the human eye sees both digits ON at any time. This process requires the digits to be enabled alternately and continuously. As a result, the processor cannot perform any other task and is busy all the time, refreshing the digits. One technique used in non-multitasking systems is to use timer interrupts and refresh the digits in the timer interrupt service routines. In this project, we will be employing a multitasking approach to refresh the display digits so that the processor can carry out other tasks. The project aims to show how the digits of a multiplexed 2-digit 7-segment LED display can be refreshed, while the main program sends data to the display to count up in seconds from 00 to 99.

**Aim**: This project aims to show how a continuous timer can be created on a MAX78000FTHR development board.

**7-Segment LED Displays**: Displaying data is one of the fundamental output activities of any microcontroller system. For example, displays are used to show the sensor data such as temperature, humidity, pressure, etc. Several types of display devices can be used in microcontroller-based systems. LCDs and 7-segment displays are probably two of the most commonly used display devices. There are several types of LCD, such as text-based, graphics, colour, touch screen, etc. 7-segment displays are used to display numeric or alphanumeric values, and they can have one or more digits. One-digit displays can only display numbers from 0 to 9. Two-digit displays can display numbers from 0 to 99, three-digit displays numbers from 0 to 999, and so on. In this project, a two-digit 7-segment display is used.

As shown in Figure 9.5, a 7-segment LED display consists of 7 LEDs connected such that numbers from 0 to 9 and some letters can be displayed. Segments are identified by letters **a** to **g**. Figure 9.6 shows the segment names of a typical 7-segment display.

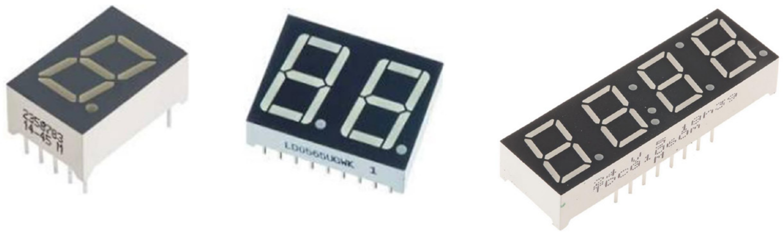


Figure 9.5 Some 7-segment displays

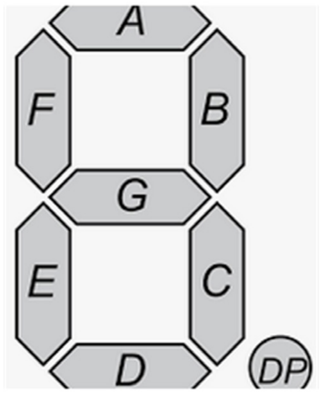


Figure 9.6 Segment names of a 7-segment display

Figure 9.7 shows how numbers 0 to 9 can be obtained by turning different segments of the display ON or OFF.

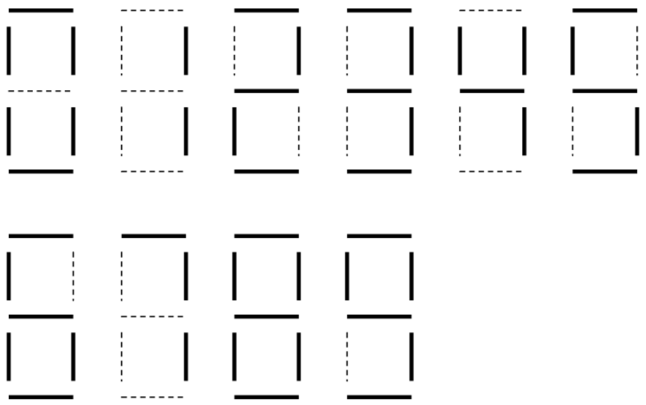


Figure 9.7 Displaying numbers 0 – 9

7-segment LED displays are available in two different configurations: common **cathode** and **anode**. As shown in Figure 9.8, in common cathode configuration all cathodes of all segment LEDs are connected to ground. The segments are turned ON by applying a logic 1 to the required segment LED via current limiting resistors. In common cathode configuration, the 7-segment LED is connected to the microcontroller in current sourcing mode.

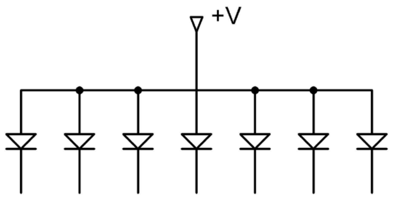


Figure 9.8 Common cathode 7-segment LED display

In a common anode configuration, the anode terminals of all the LEDs are connected as shown in Figure 9.9. This common point is normally then connected to the supply voltage. A segment is turned ON by connecting its cathode terminal to logic 0 via a current limiting resistor. In common anode configuration, the 7-segment LED is connected to the microcontroller in current sinking mode.

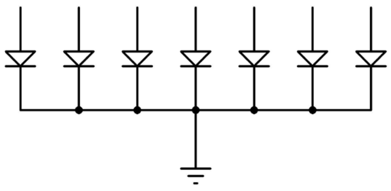


Figure 9.9 Common anode 7-segment LED display

In multiplexed LED applications (for example, see Figure 9.10 for a 2-digit multiplexed LED display), the LED segments of all digits are tied together and the common pins of each digit are turned ON separately by the microcontroller. By displaying each digit for several milliseconds, the eye can not distinguish that digits are not ON all of the time. This way we can multiplex any number of 7-segment displays together. For example, to display the number 57, we have to send 5 to the first digit and enable its common pin. After a few milliseconds, number 7 is sent to the second digit and the common point of the second digit is enabled. When this process is repeated continuously, the user sees as if both displays are always ON.

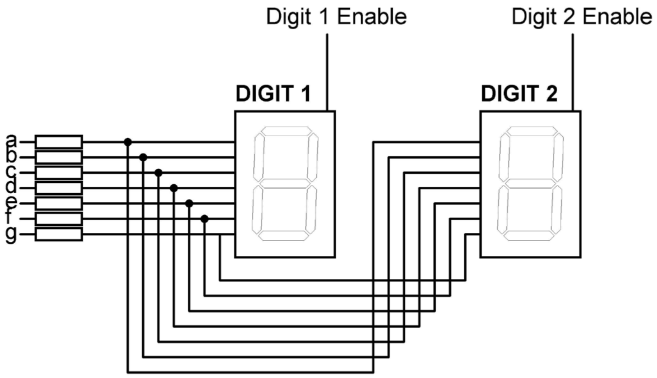


Figure 9.10 2-digit multiplexed 7-segment LED display

Some manufacturers provide multiplexed multi-digit displays in single packages. For example, we can purchase 2,4, or 8 digit multiplexed displays in a single package. The display used in this project is the DC56-11EWA which is a red 0.56 inch height **common-cathode** two-digit multiplexed display having 18 pins, where the pin configuration is shown in Table 9.1. This display can be controlled from the microcontroller as follows:

- Send the segment bit pattern for digit 1 to segments a to g
- Enable digit 1
- Wait for a few milliseconds
- Disable digit 1
- Send the segment bit patter for digit 2 to segments a to g
- Enable digit 2
- Wait for a few milliseconds
- Disable digit 2
- Repeat the above process continuously

| **Pin no** | **Segment** |
|---|---|
| 1,5 | e |
| 2,6 | d |
| 3,8 | c |
| 14 | digit 1 Enable |
| 17,7 | g |
| 15,10 | b |
| 16,11 | a |
| 18,12 | f |
| 13 | digit 2 Enable |
| 4 | decimal Point1 |
| 9 | decimal Point 2 |

Table 9.1 Pin configuration of DC56-11EWA dual display

The segment configuration of the DC56-11EWA display is shown in Figure 9.11. In a multiplexed display application, the segment pins of corresponding segments are connected. For example, pins 11 and 16 are connected as the common **a** segment. Similarly, pins 15 and 10 are connected as the common **b** segment and so on.

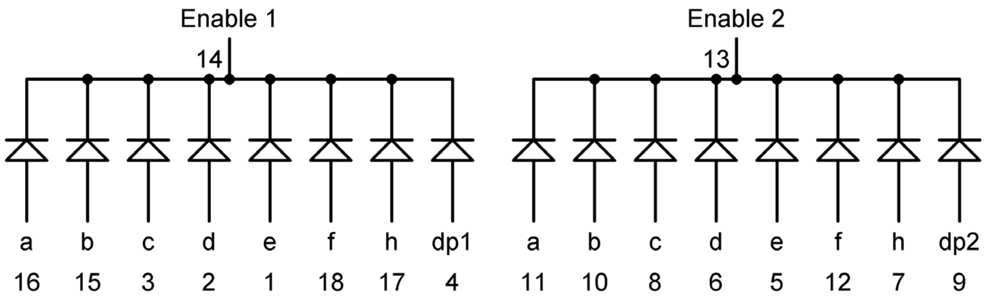


Figure 9.11 DC56-11EWA display segment configuration

**Block Diagram**: Figure 9.12 shows the block diagram of the project.

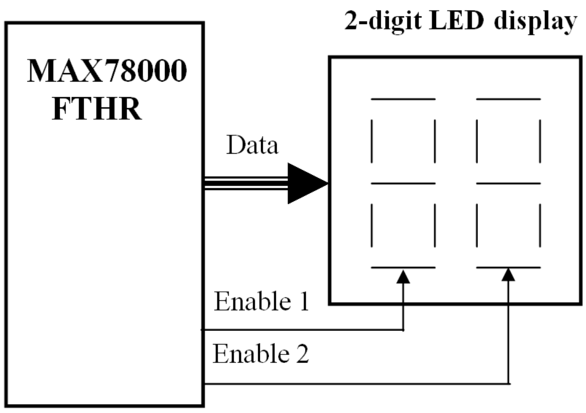


Figure 9.12 Block diagram of the project

**Circuit Diagram**: The circuit diagram of the project is shown in Figure 9.13. In this project, the following pins of the MA78000FTHR development board are used to interface with the 7-segment LED display:

| 7-Segment Display pin | MAX78000FTHR pin | Physical pin no |
|---|---|---|
| a | P0_7 | 11 |
| b | P0_5 | 12 |
| c | P0_6 | 13 |
| d | P0_19 | 20 |
| e | P0_11 | 21 |
| f | P0_8 | 22 |
| g | P0_9 | 23 |
| E1 | P2_7 (via transistor) | 15 |
| E2 | P2_6 (via transistor) | 14 |

7-segment display segments are driven from the port pins through 1K current limiting resistors. Digit enable pins E1 and E2 are driven from port pins P2_7 and P2_6 respectively through two BC108 type NPN transistors (any other NPN transistor can be used here), used as switches. The collectors of these transistors drive the segment digits. The segments are enabled when the base of the corresponding transistor is set to logic 1. Notice the following pins of the display are connected to form a multiplexed display:

16 and 11, 15 and 10, 3 and 8, 2 and 6, 1 and 5, 17 and 7, 18 and 12.

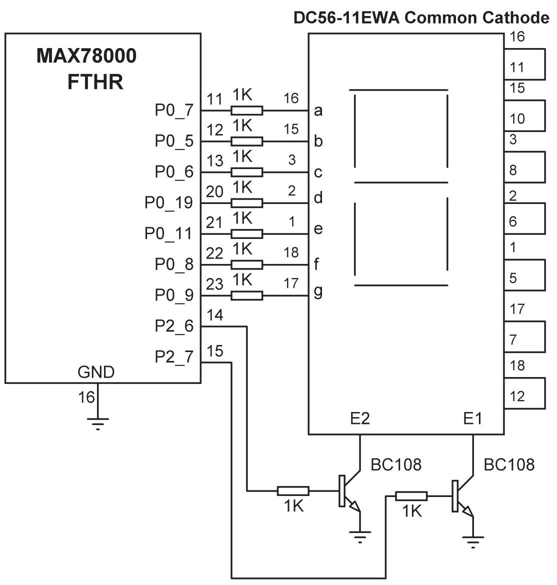
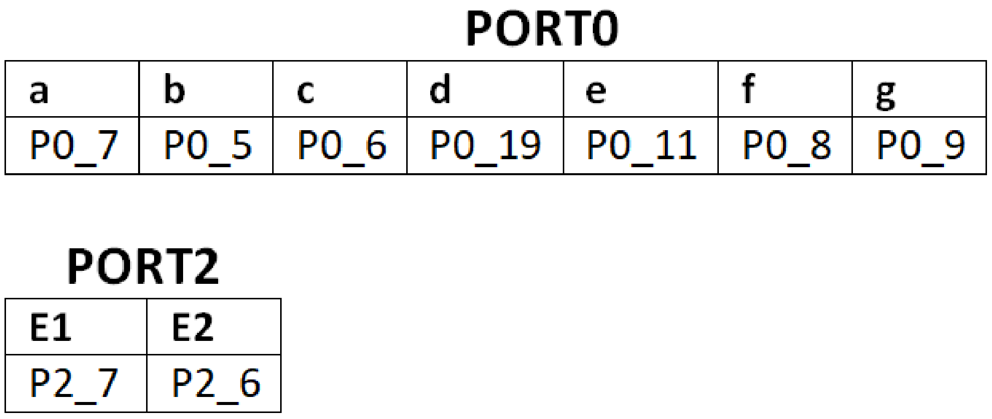


Figure 9.13 Circuit diagram of the project

The input-output map of the project is shown in Figure 9.14 where all port pins are outputs.



PORT0

| a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|
| P0_7 | P0_5 | P0_6 | P0_19 | P0_11 | P0_8 | P0_9 |

PORT2

| E1 | E2 |
|---|---|
| P2_7 | P2_6 |

Figure 9.14 Input-output map

**Program Listing**: Before driving the display, we have to know the relationship between the numbers to be displayed and the corresponding segments to be turned ON. This is shown below:

| Number to be displayed | LED bit pattern (a,b,c,d,e,f,g) |
|---|---|
| 0 | 1,1,1,1,1,1,0 (0x7E) |
| 1 | 0,1,1,0,0,0,0 (0x30) |
| 2 | 1,1,0,1,1,0,1 (0x6D) |
| 3 | 1,1,1,1,0,0,1 (0x79) |
| 4 | 0,1,1,0,0,1,1 (0x33) |
| 5 | 1,0,1,1,0,1,1 (0x5B) |
| 6 | 1,0,1,1,1,1,1 (0x5F) |
| 7 | 1,1,1,0,0,0,0 (0x70) |
| 8 | 1,1,1,1,1,1,1 (0x7F) |
| 9 | 1,1,1,1,0,1,1 (0x7B) |

Figure 9.15 shows the program listing (**SevenCount**). Timer1 (TMR1) is used in this program with the timer clock set to 8M.

The connections between the LED segments, segment control bits, and the MAX78000FTHR are then defined:

```
#define a MXC_GPIO_PIN_7                                    // a pin
#define b MXC_GPIO_PIN_5                                    // b pin
#define c MXC_GPIO_PIN_6                                    // c pin
#define d MXC_GPIO_PIN_19                                   // d pin
#define e MXC_GPIO_PIN_11                                   // e pin
#define f MXC_GPIO_PIN_8                                    // f pin
#define g MXC_GPIO_PIN_9                                    // g pin
#define E1 MXC_GPIO_PIN_7                                   // E1 pin
#define E2 MXC_GPIO_PIN_6                                   // E2 pin
```

Array **segs** stores the pin definitions a - g. Array **data** stores the bit patterns 0 to 9 corresponding to the digits to be displayed. For example, sending 0x30 displays number 1 and so on.
Inside the main program, PORT0 and PORT2 are configured as outputs, timer interrupts are configured, and a while loop is set up. Inside this while loop, variable count is incremented by one with a one-second delay between each count.

The main program calls function **ContinuousTimer** where the timer is configured to operate continuously and generate interrupts every 10 milliseconds. The following timer setup is used:

```
        tmr.pres = TMR_PRES_1024;                           // Prescaler
        tmr.mode = TMR_MODE_CONTINUOUS;                     // Continuous
        tmr.bitMode = TMR_BIT_MODE_16B;                     // Bit mode
        tmr.clock = CONT_CLOCK_SOURCE;                      // Clock source
        tmr.cmp_cnt = 81;                                   // 10 ms
        tmr.pol = 0;
```

With the Prescaler set to 1024 and a clock frequency of 8M, the required comparator value for 10ms (0.01 seconds) interrupts is calculated as:

cmp_cnt = counter clock frequency (Hz) x Required delay between interrupts (secs) / Prescaler + 1

or,

cmp_cnt = 8192000 x 0.01 / 1024 + 1 = 81

Function **send** groups several bits together and sends data to them. The function has two arguments: **No** is the number to be displayed and **L** is the number of bits to be grouped, which is 7 in this project. For example, if No is 0, the segments to display number 0 will be turned ON.

The timer interrupt service routine is the function **ContinuousTimerHandler**. On entry to this function, the timer interrupt flag is cleared so further timer interrupts can be accepted by the program. A variable called **flag** is used to determine whether to enable digit **E1** or digit **E2**. These digits are enabled alternately with each digit being ON for 10 milliseconds. **MSD** and **LSD** are the upper and the lower digits of the number to be displayed. For example, if the number is 25, MSD = 2 and LSD = 5. Leading zeroes are disabled if the number to be displayed is less than 10. Therefore, for example, the number 5 is displayed as 5 and not as 05. Function **send** is called inside the timer interrupt service routine to display the required number on the 7-segment LED. Each digit is enabled (i.e. refreshed) for 10 milliseconds and the human eye sees both digits as ON all the time.

```
/*----------------------------------------------------------------------
        7-SEGMENT 2-DIGIT COUNTER

In this project a 2-digit 7-segent display is connected to the MAX78000FTHR.
The display counts up from 00 to 99 with 1 second delay between each count

Author: Dogan Ibrahim
Date  : March 2021
Work  : SevenCount
----------------------------------------------------------------------*/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_sys.h"
```

```
#include "nvic_table.h"
#include "mxc.h"
#include "math.h"

//
// Parameters for Continuous timer
//
#define CONT_TIMER          MXC_TMR1                    // Use TMR1
#define CONT_CLOCK_SOURCE   MXC_TMR_8M_CLK              // Timer clock

#define a MXC_GPIO_PIN_7                                // a pin
#define b MXC_GPIO_PIN_5                                // b pin
#define c MXC_GPIO_PIN_6                                // c pin
#define d MXC_GPIO_PIN_19                               // d pin
#define e MXC_GPIO_PIN_11                               // e pin
#define f MXC_GPIO_PIN_8                                // f pin
#define g MXC_GPIO_PIN_9                                // g pin
#define E1 MXC_GPIO_PIN_7                               // E1 pin
#define E2 MXC_GPIO_PIN_6                               // E2 pin

mxc_tmr_cfg_t tmr;
mxc_gpio_cfg_t gpio_out;
mxc_gpio_cfg_t gpio2_out;

//
// GPIO pins used for the display (PORT0): 7,5,6,19,11,8,9
// GPIO pins used for display control (PORT2): 7, 6
//
int segs[] = {a, b, c, d, e, f, g};

//
// Define 7-segment display bit patterns for digita 0 - 9
//
int data[] = {0x7E, 0x30, 0x6D, 0x79, 0x33, 0x5B, 0x5F, 0x70, 0x7F, 0x7B};

int cnt = 0;
int flag = 0;

//
// This function groups 7 port pins together and sends data to them
//
void send(unsigned int No, unsigned int L)
{
        unsigned int j, i, m, r;
        m = L - 1;
        for(i = 0; i < L; i++)
        {
                j = 1;
                j=pow(2,m);
                r = No & j;
                if(r > 0)
                        MXC_GPIO_OutSet(gpio_out.port, segs[i]);  // Set a segment
                else
                        MXC_GPIO_OutClr(gpio_out.port, segs[i]);  // Clear a segment
                m--;
        }
}

//
// Timer interrupt service routine. The program jumps to this routine every 10 ms.
// Here, the count to be displayed is broken into its MSD and LSD digits and are
// displayed on the 7-segment LED. This routine refreshes the display every 10 ms
//
void ContinuousTimerHandler()
{
        unsigned int LSD, MSD;

    MXC_TMR_ClearFlags(CONT_TIMER);                            // Clear interrupt

        if(flag == 0)
        {
                flag = 1;
                MXC_GPIO_OutClr(gpio2_out.port, E1);            // Disable digit E1
                if(cnt > 9)                                     // If > 9
                {
                        MSD = cnt / 10;
                        send(data[MSD], 7);
                        MXC_GPIO_OutSet(gpio2_out.port, E2);    // Enable digit E2
                }
        }

        else
        {
                MXC_GPIO_OutClr(gpio2_out.port, E2);            // Disable digit E2
                LSD = cnt % 10;
                send(data[LSD], 7);
                MXC_GPIO_OutSet(gpio2_out.port, E1);            // Enable digit E1
                flag=0;
        }
}

//
```

```
// Configure the continuous timer for 10 ms
//
void ContinuousTimer()
{
    MXC_TMR_Shutdown(CONT_TIMER);

    tmr.pres = TMR_PRES_1024;                                 // Prescaler
    tmr.mode = TMR_MODE_CONTINUOUS;                           // Continuous
    tmr.bitMode = TMR_BIT_MODE_16B;                           // Bit mode
    tmr.clock = CONT_CLOCK_SOURCE;                            // Clock source
    tmr.cmp_cnt = 81;                                         // 10 ms
    tmr.pol = 0;

    MXC_TMR_Init(CONT_TIMER, &tmr, true);                     // Init timer
}

int main(void)
{
    /* Setup output pins of PORT0 */
    gpio_out.port = MXC_GPIO0;                                // Port 0
    gpio_out.mask = a|b|c|d|e|f|g;                            // Segments
    gpio_out.pad = MXC_GPIO_PAD_NONE;                         // None
    gpio_out.func = MXC_GPIO_FUNC_OUT;                        // Output
    gpio_out.vssel = MXC_GPIO_VSSEL_VDDIOH;                   // Set port to +3.3V
    MXC_GPIO_Config(&gpio_out);

    /* Setup output pins of PORT2 */
    gpio2_out.port = MXC_GPIO2;                               // Port 2
    gpio2_out.mask = E1|E2;                                   // Segment controls
    gpio2_out.pad = MXC_GPIO_PAD_NONE;                        // None
    gpio2_out.func = MXC_GPIO_FUNC_OUT;                       // Output
    gpio2_out.vssel = MXC_GPIO_VSSEL_VDDIOH;                  // Set port to +3.3V
    MXC_GPIO_Config(&gpio2_out);

    NVIC_SetVector(TMR1_IRQn, ContinuousTimerHandler);       // Timer interrupt
    NVIC_EnableIRQ(TMR1_IRQn);                                // Enable timer interrupt
    ContinuousTimer();                                        // Call timer routine

        while(1)                                              // Do Forever
        {
                cnt++;                                        // Increment count
                if(cnt == 100)cnt = 0;                        // If 100...
                MXC_Delay(1000000);                           // 1 sec delay
        }

}
```

Figure 9.15 Program listing

**Suggestions**: The 2-digit 7-segment display requires 9 output ports. You may try to use the MCP23017 port expander chip to expand the MAX78000FTHR development board input-output port count and use the 7-segment display with this chip.

## 9.7 ● Project 4 – Refreshing a 4-digit 7-segment display – seconds counter

**Description**: In this project, a 7-segment 4-digit multiplexed LED display is used as a counter to count up every second from 0 to 9999. The project is very similar to the previous one, except 4 digits are used instead of 2.

The operation of a 4 digit multiplexed display (Figure 9.16) is similar to the 2 digit display, where the LED segments of all the digits are tied together and the common pins of each digit are turned ON separately by the microcontroller. By displaying each digit for several milliseconds, the eye can not differentiate that the digits are not ON all the time. This way we can multiplex any number of 7-segment displays together. For example, to display the number 5734, we have to send 5 to the first digit and enable its common pin. After a few milliseconds, number 7 is sent to the second digit and the common point of the second digit is enabled, and so on. When this process is continuously repeated the user sees it as if both displays are continuously ON.

Figure 9.16 4-digit multiplexed 7-segment LED display

The display used in this project is the DC56-11EWA which is a red 0.56-inch height common-cathode two-digit multiplexed display having 18 pins. The pin configuration is shown in Table 3.1. Two such display modules are used to construct a 4 digit display. Each module has E1 and E2 enable pins.

In a multiplexed display application, the segment pins of corresponding segments are connected. For example, pins 11 and 16 are connected as the common a segment. Similarly, pins 15 and 10 are connected as the common b segment and so on.

**Block Diagram**: Figure 9.17 shows the block diagram of the project.

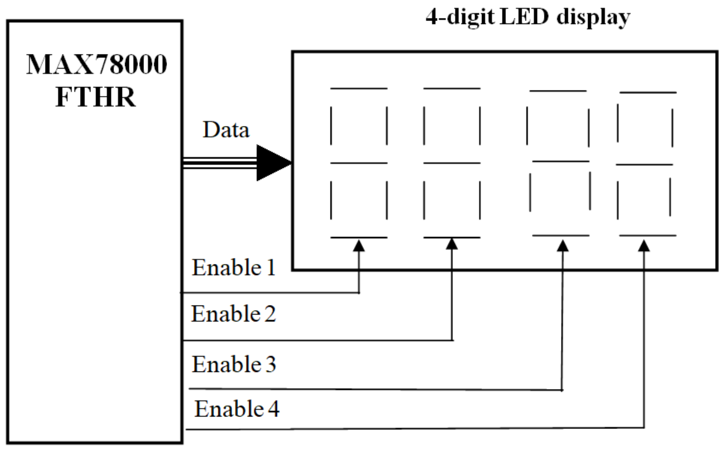


Figure 9.17 Block diagram of the project

Circuit Diagram: The circuit diagram of the project is shown in Figure 9.18. In this project, the following pins of the MAX78000FTHR development board are used to interface with the 7-segment LED display:

| 7-Segment Display pin | MAX78000FTHR pin | Physical pin no |
|---|---|---|
| a | P0_7 | 11 |
| b | P0_5 | 12 |
| c | P0_6 | 13 |
| d | P0_19 | 20 |
| e | P0_11 | 21 |
| f | P0_8 | 22 |
| g | P0_9 | 23 |
| E1 | P2_7 (via transistor) | 15 |
| E2 | P2_6 (via transistor) | 14 |
| E3 | P2_3 (via transistor) | 5 |
| E4 | P2_4 (via transistor) | 6 |

7-segment display segments are driven from the port pins through 1K current limiting resistors. Digit enable pins E1, E2, E3 and E4 are driven from port pins P2_7, P2_6, P2_3, and P2_4 respectively (in point of fact, E3 and E4 are the E1 and E2 pins of the second dual display module) through two BC108 type NPN transistors (any other NPN transistor can be used here), used as switches. The collectors of these transistors drive the segment digits. The segments are enabled when the base of the corresponding transistor is set to logic 1. Notice the following pins of the display are connected to form a multiplexed display:

16 and 11, 15 and 10, 3 and 8, 2 and 6, 1 and 5, 17 and 7, 18 and 12.

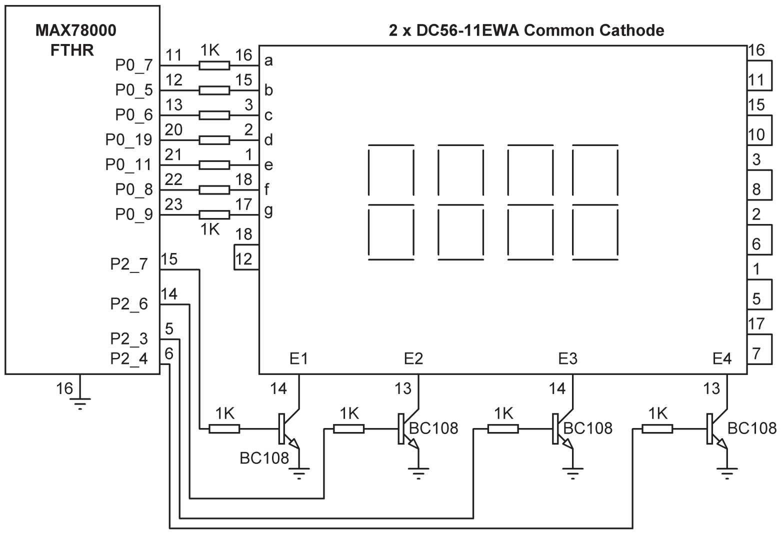


Figure 9.18 Circuit diagram of the project

**Program Listing**: Figure 9.19 shows the program listing (**SevenCount4**). The program is very similar to the one with 2 digits. Here, the connections to digit control bits E3 and E4 are also defined:

```
#define a MXC_GPIO_PIN_7                                          // a pin
#define b MXC_GPIO_PIN_5                                          // b pin
#define c MXC_GPIO_PIN_6                                          // c pin
#define d MXC_GPIO_PIN_19                                         // d pin
#define e MXC_GPIO_PIN_11                                         // e pin
#define f MXC_GPIO_PIN_8                                          // f pin
#define g MXC_GPIO_PIN_9                                          // g pin
#define E1 MXC_GPIO_PIN_7                                         // E1 pin
#define E2 MXC_GPIO_PIN_6                                         // E2 pin
#define E3 MXC_GPIO_PIN_3                                         // E3 pin
#define E4 MXC_GPIO_PIN_4                                         // E4 pin
```

The timer interrupt service routine **ContinuousTimerHandler** has been modified to display numeric data for 4 digits instead of 2. The timer interrupt processing time (LED refreshing rate) changed to 5ms since there are 4 digits and less time is required for each digit. The **Prescaler** is set to 1024 as before and the timer comparator value is set to 41 to give 5 ms time for each interrupt processing time. Variable **flag** determines which digit to process. It is initially set to 1 so that the first digit is processed initially. It is then set to 2 for the second digit and so on. The digits are extracted and each is displayed in its correct sequence.

```
/*--------------------------------------------------------------------------------
        7-SEGMENT 4-DIGIT COUNTER

In this project a 2 x 2-digit 7-segment displays are connected to the MAX78000FTHR.
to form a 4-digit display. The display counts up from 0000 to 9999 with 1 second
delay between each count

Author: Dogan Ibrahim
Date  : March 2021
Work  : SevenCount4
--------------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_sys.h"
#include "nvic_table.h"
#include "mxc.h"
#include "math.h"

//
// Parameters for Continuous timer
//
#define CONT_TIMER          MXC_TMR1                            // Use TMR1
#define CONT_CLOCK_SOURCE   MXC_TMR_8M_CLK                      // Timer clock

#define a MXC_GPIO_PIN_7                                        // a pin
#define b MXC_GPIO_PIN_5                                        // b pin
#define c MXC_GPIO_PIN_6                                        // c pin
#define d MXC_GPIO_PIN_19                                       // d pin
#define e MXC_GPIO_PIN_11                                       // e pin
#define f MXC_GPIO_PIN_8                                        // f pin
#define g MXC_GPIO_PIN_9                                        // g pin
#define E1 MXC_GPIO_PIN_7                                       // E1 pin
#define E2 MXC_GPIO_PIN_6                                       // E2 pin
#define E3 MXC_GPIO_PIN_3                                       // E3 pin
#define E4 MXC_GPIO_PIN_4                                       // E4 pin

mxc_tmr_cfg_t tmr;
mxc_gpio_cfg_t gpio_out;
mxc_gpio_cfg_t gpio2_out;

//
// GPIO pins used for the display (PORT0): 7,5,6,19,11,8,9
// GPIO pins used for display control (PORT2): 7, 6
//
int segs[] = {a, b, c, d, e, f, g};
```

```
//
// Define 7-segment display bit patterns for digita 0 - 9
//
int data[] = {0x7E, 0x30, 0x6D, 0x79, 0x33, 0x5B, 0x5F, 0x70, 0x7F, 0x7B};

int cnt = 0;
int flag = 1;
unsigned int LSD, MSD, MSDD, LSDD, LSDD2, LSDD3;

//
// This function groups 7 port pins together and sends data to them
//
void send(unsigned int No, unsigned int L)
{
        unsigned int j, i, m, r;
        m = L - 1;
        for(i = 0; i < L; i++)
        {
                j = 1;
                j=pow(2,m);
                r = No & j;
                if(r > 0)
                        MXC_GPIO_OutSet(gpio_out.port, segs[i]);  // Set a segment
                else
                        MXC_GPIO_OutClr(gpio_out.port, segs[i]);  // Clear a segment
                m--;
        }
}

//
// Timer interrupt service routine. The program jumps to this routine every 10 ms.
// Here, the count to be displayed is broken into its MSD and LSD digits and are
// displayed on the 7-segment LED. This routine refreshes the display every 10 ms
//
void ContinuousTimerHandler()
{
    MXC_TMR_ClearFlags(CONT_TIMER);                     // Clear interrupt

        if(flag == 1)                                   // Process first digit
        {
                flag = 2;                               // Next digit 2
                MXC_GPIO_OutClr(gpio2_out.port, E2);    // Disable digit E1
                MXC_GPIO_OutClr(gpio2_out.port, E3);    // Disable digit E1
                MXC_GPIO_OutClr(gpio2_out.port, E4);    / Disable digit E1
                MSD = cnt / 1000;                       // MSD
                MSDD = cnt % 1000;
                send(data[MSD], 7);
                MXC_GPIO_OutSet(gpio2_out.port, E1);    // Enable digit E2
        }

        else if(flag == 2)                              // Process second digit
        {
                flag = 3;                               // Next digit 3
                MXC_GPIO_OutClr(gpio2_out.port, E1);    // Disable digit E2
                MXC_GPIO_OutClr(gpio2_out.port, E3);    // Disable digit E1
                MXC_GPIO_OutClr(gpio2_out.port, E4);    // Disable digit E1
                LSDD = MSDD / 100;
                LSD = MSDD % 100;
                send(data[LSDD], 7);
                MXC_GPIO_OutSet(gpio2_out.port, E2);    // Enable digit E1
        }
        else if(flag == 3)                              // Process third digit
        {
                flag = 4;                               // Next digit 4
                MXC_GPIO_OutClr(gpio2_out.port, E1);    // Disable digit E2
                MXC_GPIO_OutClr(gpio2_out.port, E2);    // Disable digit E1
                MXC_GPIO_OutClr(gpio2_out.port, E4);    // Disable digit E1
                LSDD2 = LSD / 10;
                LSDD3 = LSD % 10;
                send(data[LSDD2], 7);
                MXC_GPIO_OutSet(gpio2_out.port, E3);    // Enable digit E1
        }
        else if(flag == 4)                              // Process fourth digit
        {
                flag = 1;                               // Next digit 1
                MXC_GPIO_OutClr(gpio2_out.port, E1);    // Disable digit E2
                MXC_GPIO_OutClr(gpio2_out.port, E2);    // Disable digit E1
                MXC_GPIO_OutClr(gpio2_out.port, E3);    // Disable digit E1
                send(data[LSDD3], 7);
                MXC_GPIO_OutSet(gpio2_out.port, E4);    // Enable digit E1

        }
}

//
// Configure the continuous timer for 5 ms
//
void ContinuousTimer()
{
    MXC_TMR_Shutdown(CONT_TIMER);

    tmr.pres = TMR_PRES_1024;                           // Prescaler
    tmr.mode = TMR_MODE_CONTINUOUS;                      // Continuous
    tmr.bitMode = TMR_BIT_MODE_16B;                      // Bit mode
    tmr.clock = CONT_CLOCK_SOURCE;                       // Clock source
    tmr.cmp_cnt = 41;                                    // 5 ms
    tmr.pol = 0;

    MXC_TMR_Init(CONT_TIMER, &tmr, true);                // Init timer
}

int main(void)
{
    /* Setup output pins of PORT0 */
    gpio_out.port = MXC_GPIO0;                           // Port 0
    gpio_out.mask = a|b|c|d|e|f|g;                       // Segments
    gpio_out.pad = MXC_GPIO_PAD_NONE;                    // None
    gpio_out.func = MXC_GPIO_FUNC_OUT;                   // Output
    gpio_out.vssel = MXC_GPIO_VSSEL_VDDIOH;              // Set port to +3.3V
    MXC_GPIO_Config(&gpio_out);

    /* Setup output pins of PORT2 */
    gpio2_out.port = MXC_GPIO2;                          // Port 2
    gpio2_out.mask = E1|E2|E3|E4;                        // Segment controls
    gpio2_out.pad = MXC_GPIO_PAD_NONE;                   // None
    gpio2_out.func = MXC_GPIO_FUNC_OUT;                  // Output
    gpio2_out.vssel = MXC_GPIO_VSSEL_VDDIOH;             // Set port to +3.3V
    MXC_GPIO_Config(&gpio2_out);

    NVIC_SetVector(TMR1_IRQn, ContinuousTimerHandler);  // Timer interrupt
    NVIC_EnableIRQ(TMR1_IRQn);                           // Enable timer interrupt
    ContinuousTimer();                                   // Call timer routine

        while(1)                                         // Do Forever
        {
                cnt++;                                   // Increment count
                if(cnt == 10000)cnt = 0;                 // If 10000...
                MXC_Delay(1000000);                      // 1 sec delay
        }

}
```

Figure 9.19 Program listing

## 9.8 ● Pulse width modulation (PWM)

Pulse Width Modulation (PWM) is a commonly used technique for controlling the power delivered to analog loads using digital waveforms. Although analog voltages (and currents) can be used to control the delivered power, they have several drawbacks. Controlling large analog loads requires large voltages and currents that cannot easily be obtained using standard analog circuits and DACs. Precision analog circuits can be heavy, large, and expensive and they are also sensitive to noise. By using the PWM technique, the average value of voltage (and current) fed to a load is controlled by switching the supply voltage ON and OFF at a fast rate. The longer the power-on time, the higher is the voltage supplied to the load.

Figure 9.20 shows a typical PWM waveform where the signal is a repetitive positive pulse, having the period T, ON time TON, and OFF time of T – TON seconds. The minimum and maximum values of voltage supplied to the load are 0 and VP respectively. PWM switching frequency is usually set very high (usually in the order of several kHz) so that it does not affect the load that uses the power. The main advantage of PWM is that the load is operated efficiently since power loss in the switching device is very low. When the switch is ON there is almost no voltage drop across the switch. When the switch is OFF, there is no current supplied to the load.

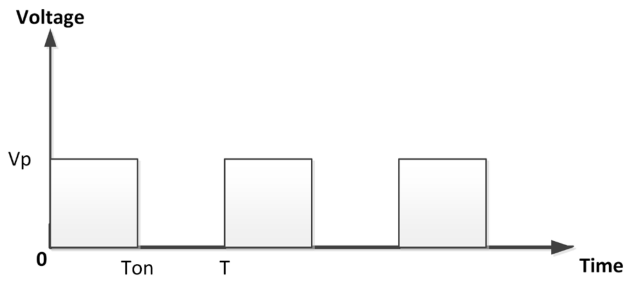


Figure 9.20 PWM waveform

The duty cycle (or D) of a PWM waveform is defined as the ratio of the ON time to its period.

Expressed mathematically,

$$\text{Duty Cycle (D)} = T_{ON} / T$$

The duty cycle is usually expressed as a percentage and therefore,

$$D = (T_{ON} / T_{OFF}) \times 100\ \%$$

By varying the duty cycle between 0% and 100% we can effectively control the average voltage supplied to the load between 0 and $V_p$.

The average value of voltage applied to the load can be calculated by considering a general PWM waveform shown in Figure 1. The average value A of waveform f(t) with period T and peak value $y_{max}$ and minimum value $y_{min}$ is calculated as:

$$A = \frac{1}{T}\int_{0}^{T} f(t)\,dt$$

or,

$$A = \frac{1}{T}\left(\int_{0}^{T_{ON}} y_{\max}\,dt + \int_{T_{ON}}^{T} y_{\min\,dt}\right)$$

In a PWM waveform ymin = 0 and the above equation becomes:

$$A = \frac{1}{T}\left(T_{ON}\;y_{\max}\right)$$

or,

$$A = D\,y_{\max}$$

As can be seen from the above equation, the average value of the voltage supplied to the load is directly proportional to the duty cycle of the PWM waveform and by varying the duty cycle we control the average load voltage. Figure 9.21 shows the average voltage for different values of the duty cycle.

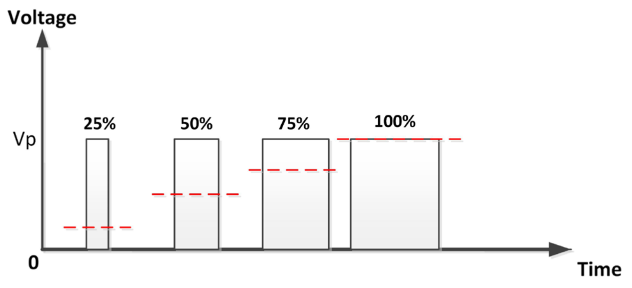


Figure 9.21 Average voltage (shown as dashed line) supplied to a load

## 9.8.1 • MAX78000 PWM

In PWM mode, the timer sends a Pulse-Width Modulated (PWM) output using the timer's output signal. The timer first counts up to the match value stored in the PWM register. At the end of the cycle where the timer count matches the PWM register value, the timer output signal toggles state. The timer continues counting until it reaches the timer comparator register value. The timer period ends on the rising edge of the timer clock, following the point when the timer matches the compare register value.

The timer peripheral automatically performs the following actions at the end of the timer period:

- The timer count is reset to 0x00000001, and the timer resumes counting.
- The timer output signal is toggled.
- The corresponding timer interrupt field will be set to 1 to indicate a timer interrupt event occurred.

When timer polarity is 0, the timer output signal starts low and then transitions to high when the timer count value matches the timer PWM value. The timer output signal remains high until the timer count value reaches the timer compare value, resulting in the timer output signal transitioning low, and the timer count value resetting to 0x0000 0001.
The PWM period in seconds is given by:

**PWM (secs) = TMR count value / timer clock**

If the polarity is set to 0, the Duty Cycle of the waveform is given by:

**Duty Cycle (%) = 100 x (Timer compare value – Timer PWM value) / Timer compare value**

If the polarity is set to 1, the Duty Cycle of the waveform is given by:

**Duty Cycle (%) = 100 x (Timer PWM value) / Timer compare value**

Two parameters are required to set a timer PWM waveform specifications: the PWM period and duty cycle.

Figure 9.22 shows a timer operating in PWM mode.

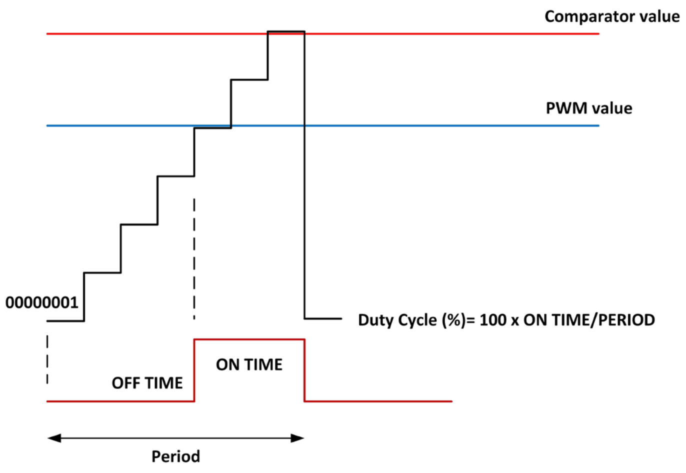


Figure 9.22 Timer operating in PWM mode

## 9.9 • Project 5 – Pulse width modulation (PWM) – generating 10 kHz square wave

**Description**: In this project, a 10 kHz PWM waveform is generated with a 50% duty cycle.

**Aim**: The project aims to show how a PWM waveform can be generated with the given period and duty cycle.

**Circuit diagram**: Figure 9.23 shows the circuit diagram of the project. In this project, Timer 4 is used with its output at pin P1_6 (see Table 9.1). This pin is connected to a digital oscilloscope to display the waveform.

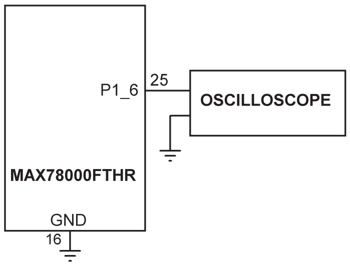


Figure 9.23 Circuit diagram of the project

**Program listing**: Figure 9.24 shows the program listing (**PWM10K**). At the beginning of the program, PWM parameters are defined. Timer 3 is used in this project with the clock source selected as 8 MHz, the frequency and duty cycle of the desired PWM waveform selected as 10K (10000 Hz) and 50% respectively.:

```
#define PWM_CLOCK_SOURCE  MXC_TMR_8M_CLK          // Timer clock
#define FREQ                 10000                // 10 kHz
#define DUTY_CYCLE           50                   // 50%
#define PWM_TIMER            MXC_TMR3             // Timer3
```

The PWM waveform is generated inside function **PWMTimer**. The number of ticks for the PWM period is stored in variable **periodTicks** using function **MXC_TMR_GetPeriod**. The first argument of this function is the timer used. The second is the clock used. The third is the prescaler value. The last argument is the desired frequency. Duty cycle is set to 50%.

The timer is configured with the Prescaler set to 16, mode set to PWM, 32-bit counter selected with the clock source as 8 MHz, comparator value set to **periodTicks**. Polarity is set to 0 so the output signal is normally at logic 0.

```
tmr.pres = TMR_PRES_16;                          // Prescaler=16
tmr.mode = TMR_MODE_PWM;                         // Mode=PWM
tmr.bitMode = TMR_BIT_MODE_32;                   // 32-bit mode
tmr.clock = PWM_CLOCK_SOURCE;                    // Clock source
tmr.cmp_cnt = periodTicks;                          // Comparator value
tmr.pol = 0;                                        // Polarity
```

The timer is initialised, and the duty cycle set using the **MXC_TMR_Init** function. Starting the timer generates the required PWM waveform.

```
/*--------------------------------------------------------------------------------
         10 kHz PULSE WIDTH MODULATED WAVEFORM

In this project a 10 kHz pulse width modulated waveform is generated using Timer 3
of the MAX78000 microcontroller. The waveform is output at pin P1_6

Author: Dogan Ibrahim
Date  : March 2021
Work  : PWM10K
--------------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_sys.h"
#include "mxc.h"

//
// Parameters for PWM timer
//
#define PWM_CLOCK_SOURCE    MXC_TMR_8M_CLK             // Timer clock
#define FREQ                10000                      // 10 kHz
#define DUTY_CYCLE          50                         // 50%
#define PWM_TIMER           MXC_TMR3                   // Timer3

mxc_tmr_cfg_t tmr;

void PWMTimer(void)
{
             unsigned int periodTicks = MXC_TMR_GetPeriod(PWM_TIMER, PWM_CLOCK_SOURCE, 16, FREQ);
             unsigned int dutyTicks   = periodTicks * DUTY_CYCLE / 100;

             MXC_TMR_Shutdown(PWM_TIMER);

             tmr.pres = TMR_PRES_16;                   // Prescaler=16
             tmr.mode = TMR_MODE_PWM;                  // Mode=PWM
             tmr.bitMode = TMR_BIT_MODE_32;            // 32-bit mode
             tmr.clock = PWM_CLOCK_SOURCE;             // Clock source (Timer3)
             tmr.cmp_cnt = periodTicks;                // Comparator value
             tmr.pol = 0;                              // Polarity
```

```
            MXC_TMR_Init(PWM_TIMER, &tmr, true);      // Initialize timer

            MXC_TMR_SetPWM(PWM_TIMER, dutyTicks);     // Set Duty Cycle
            MXC_TMR_Start(PWM_TIMER);                 // Start timer
}


int main(void)
{
         PWMTimer();

         while(1);
}
```

Figure 9.24 Program listing

The generated waveform is shown in Figure 9.25. In this display, the horizontal axis was 0.1ms/division. The vertical axis was 1V/division. The frequency of the generated waveform is 10 kHz.

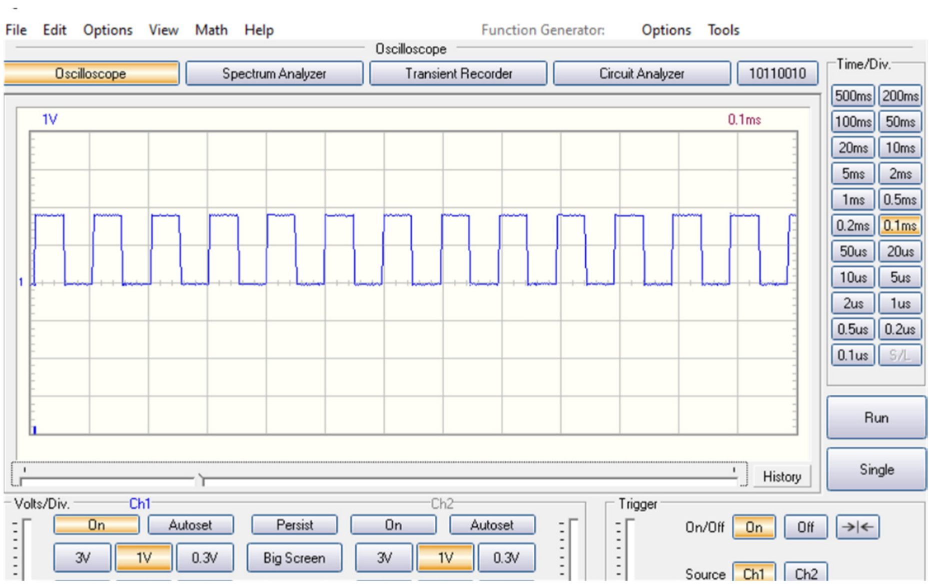


Figure 9.25 Generated waveform

## 9.10 • Project 6 – Pulse width modulation (PWM) – changing the brightness of an LED

**Description**: In this project, an external LED is connected to port pin P1_6 of the MAX78000FTHR development board through a current limiting resistor. The project sends a PWM waveform to the LED where the duty cycle is changed from 0% to 100%. As a result, the brightness of the LED changes accordingly.

**Aim**: This project aims to show how the duty cycle of a PWM waveform can be continuously changed.

**Circuit diagram**: Figure 9.26 shows the circuit diagram of the project. The LED is connected in current sinking mode with a 220 Ohm current limiting resistor. The LED is ON when the output of the microcontroller is at logic 0 and will be OFF when it is at 1. The LED is supplied from the +3.3V supply. Therefore, assuming a 2V voltage drop across the LED, the current through the LED will be about I = 3.3 – 2 / 470 = 6 mA at 100% duty cycle. The reason for using a low resistance value is so the LED is very bright when full voltage is applied.

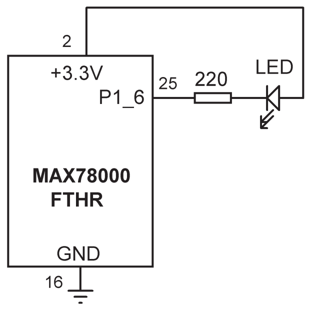


Figure 9.26 Circuit diagram of the project

**Program listing**: Figure 9.27 shows the program listing (**FADELED**). At the beginning of the program, PWM frequency is set to 1000 Hz and Timer3 is used as in the previous project with an 8 MHz clock rate. The part of the program that configures the timer for the PWM operation is the same as in the previous project. A **while** loop is formed inside the **PWMTimer** routine to vary the duty cycle of the generated signal. The duty cycle starts from 1% to 100% in steps of 10%. A 500-millisecond delay is inserted between each output. The result is that the brightness of the LED changed from OFF to fully ON. Variable **dutyTicks** sets the ratio of the **periodTicks** to the duty cycle and this value is used in function **MXC_TMR_SetPWM** to set the duty cycle by configuring the PWM register.

```
/*--------------------------------------------------------------------------------
         LED FADING USING PWM

In this project an external LED is connected to the MAX78000FTHR. The brightness
of the LED is changed by sending PWM signak to it with varying duty cycle

Author: Dogan Ibrahim
Date  : March 2021
Work  : FADELED
--------------------------------------------------------------------------------*/
#include <stdio.h>
```

```
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_sys.h"
#include "mxc.h"

//
// Parameters for PWM timer
//
#define PWM_CLOCK_SOURCE    MXC_TMR_8M_CLK                        // Timer clock
#define FREQ                1000                                  // 1000 Hz
#define PWM_TIMER           MXC_TMR3                              // Timer3

mxc_tmr_cfg_t tmr;

int DUTY_CYCLE = 1;                                               // 1%

void PWMTimer(void)
{
             unsigned int periodTicks = MXC_TMR_GetPeriod(PWM_TIMER, PWM_CLOCK_SOURCE, 16, FREQ);
             unsigned int dutyTicks   = periodTicks * DUTY_CYCLE / 100;

             MXC_TMR_Shutdown(PWM_TIMER);

             tmr.pres = TMR_PRES_16;                              // Prescaler=16
             tmr.mode = TMR_MODE_PWM;                             // Mode=PWM
             tmr.bitMode = TMR_BIT_MODE_32;                       // 32-bit mode
             tmr.clock = PWM_CLOCK_SOURCE;                        // Clock source (Timer3)
             tmr.cmp_cnt = periodTicks;                           // Comparator value
             tmr.pol = 0;                                         // Polarity

             MXC_TMR_Init(PWM_TIMER, &tmr, true);                 // Initialize timer

             MXC_TMR_SetPWM(PWM_TIMER, dutyTicks);                // Set Duty Cycle
             MXC_TMR_Start(PWM_TIMER);                            // Start timer

                  while(1)
                  {
                           dutyTicks   = periodTicks * DUTY_CYCLE / 100; // Cal periodtick ratio
                           MXC_TMR_SetPWM(PWM_TIMER, dutyTicks);         // Change duty cycle
                           DUTY_CYCLE = DUTY_CYCLE + 10;                 // Increment duty cycle
                           if(DUTY_CYCLE > 100)DUTY_CYCLE = 1;           // Reset suty cycle
                           MXC_Delay(500000);                            // Wait 500 ms
                  }
}
```

```
int main(void)
{
         PWMTimer();
}
```

Figure 9.27 Program listing

## 9.11 • Project 7 – Pulse width modulation (PWM) – brushed DC motor speed control

**Description**: This is a simple project where a small brushed DC motor is connected to the MAX78000FTHR development board through a power MOSFET transistor. In addition, a potentiometer is connected to one of the analog inputs of the microcontroller. In this project, motor speed is varied by moving the potentiometer arm.

**Block Diagram**: Figure 9.28 shows the block diagram of the project. A motor driver (MOSFET transistor) and potentiometer are connected to the microcontroller.

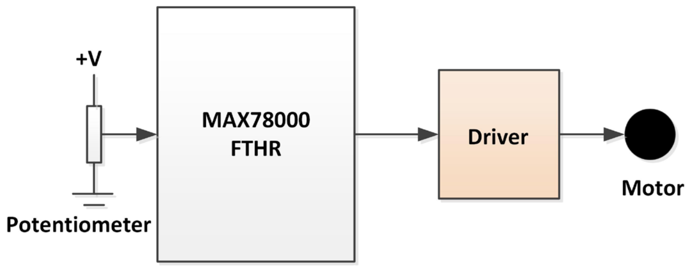


Figure 9.28 Block diagram of the project

The DC motor in this project is controlled using PWM waves as in the previous project. By varying the potentiometer arm, the analog voltage read by the microcontroller is varied and this, in turn, changes the PWM duty cycle of the voltage applied to the motor, thus causing the motor speed to change.

**Circuit diagram**: The circuit diagram of the project is shown in Figure 9.29. The potentiometer is connected to analog input AIN3 (pin 5). The DC motor is connected to PWM output P1_6 (pin 25) through an IRL540 type MOSFET switch. It is suggested to use an external power supply for the motor. Additionally, the output voltage of the MAX78000 microcontroller is +1.8V which is not enough to drive the MOSFET. A logic level converter module is used to raise +1.8V to +5V as shown in the figure.

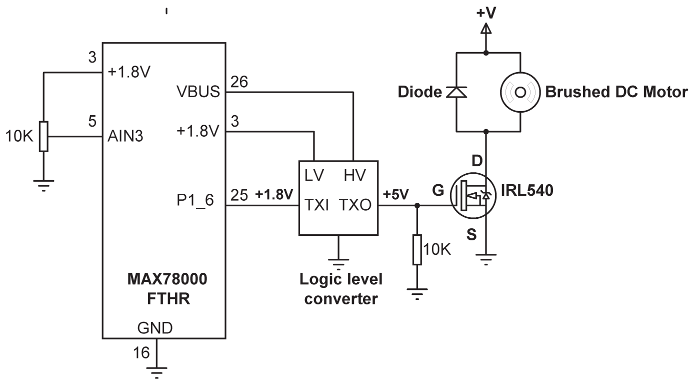


Figure 9.29 Circuit diagram of the project

**Program listing**: Figure 9.30 shows the program listing (**Motor**). The data read from the ADC varies between 0 and 1023 as the potentiometer arm is fully moved from one side to the other. This data is used to change the duty cycle from 0% to 100%. What is necessary is to map 0 to 1023 to 0 to 100. This can be done easily using the following formula:

$$y = x * 100 / 1023$$

Where x is the input (i.e. the ADC reading) and y is the required output (i.e. the duty cycle). For example, when x = 0, y becomes 0. When x is 512, y becomes 50. When x is 1023, y becomes 100. Variables **ADCMAX** and **DUTYMAX** are set to 1023 and 100 respectively, and variable conv is the ratio of **DUTYMAX** to **ADCMAX**. The timer is then started and the ADC is initialised. The remainder of the program runs in a while loop. Inside this loop, the potentiometer setting is read by the ADC and is converted to **DUTY_CYCLE** by mapping as described above. The duty cycle must not be less than or equal to zero. It is set to 1 if less than or equal to zero. **dutyTicks** (ratio to **periodTicks**) is calculated and the duty cycle of the waveform is set by calling function **MXC_TMR_SetPWM**.

Note the speed of the motor will change as the potentiometer arm is moved.

```
/*--------------------------------------------------------------------------------
        DC MOTOR SPEED CONTROL

In this project a small DC motor is connected to the MAX78000FTHR development
board. Also a potentiometer is connected. The speed of the motor is controlled
by varying the potentiometer arm

Author: Dogan Ibrahim
Date  : March 2021
Work  : MOTOR
--------------------------------------------------------------------------------*/
#include <stdio.h>
```

```
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_sys.h"
#include "mxc.h"

//
// Parameters for PWM timer
//
#define PWM_CLOCK_SOURCE        MXC_TMR_8M_CLK          // Timer clock
#define FREQ                    1000                    // 1000 Hz
#define PWM_TIMER               MXC_TMR3                // Timer3

#define ADC_CHANNEL MXC_ADC_CH_3                        // Use CH 3

mxc_tmr_cfg_t tmr;

unsigned int dutyTicks;
unsigned int periodTicks;
int DUTY_CYCLE = 1;                                     // 1%

void PWMTimer(void)
{
            periodTicks = MXC_TMR_GetPeriod(PWM_TIMER, PWM_CLOCK_SOURCE, 16, FREQ);
            dutyTicks   = periodTicks * DUTY_CYCLE / 100;

            MXC_TMR_Shutdown(PWM_TIMER);

            tmr.pres = TMR_PRES_16;                     // Prescaler=16
            tmr.mode = TMR_MODE_PWM;                    // Mode=PWM
            tmr.bitMode = TMR_BIT_MODE_32;              // 32-bit mode
            tmr.clock = PWM_CLOCK_SOURCE;               // Clock source (Timer3)
            tmr.cmp_cnt = periodTicks;                  // Comparator value
            tmr.pol = 0;                                // Polarity

            MXC_TMR_Init(PWM_TIMER, &tmr, true);        // Initialize timer
            MXC_TMR_SetPWM(PWM_TIMER, dutyTicks);       // Set Duty Cycle
            MXC_TMR_Start(PWM_TIMER);                   // Start timer
}


int main(void)
{
        int adc_val=0;
        float conv;
        float ADCMAX = 1023.0;
        float DUTYMAX = 100.0;
        conv = DUTYMAX / ADCMAX;                                // e.g. 100/1024
        PWMTimer();
        MXC_ADC_Init();                                         // Initialize ADC

        while(1)
        {
                adc_val=MXC_ADC_StartConversion(ADC_CHANNEL);   // Start conversion
                DUTY_CYCLE = (int)(adc_val * conv);
                if(DUTY_CYCLE > 100)DUTY_CYCLE = 100;
                if(DUTY_CYCLE <= 0)DUTY_CYCLE = 1;
                dutyTicks   = periodTicks * DUTY_CYCLE / 100;   // Calc periodTicks
                MXC_TMR_SetPWM(PWM_TIMER, dutyTicks);           // Change duty cycle
        }
}
```

Figure 9.30 Program listing

# Chapter 10 • Pulse Train Engine (PT)

## 10.1 • Overview

The Pulse Train engine (PT) generates either square wave signals with a 50% duty cycle, or a continuous bit pattern with a length of 2 to 32 bits. There are 16 PTs, and each one can be used independently. They may also be synchronised together. The frequency of each generated output can be set separately.

The basic features of the PT are:

- Independent pulse train outputs
- Square wave output (50% duty cycle)
- Pattern output mode (2 to 32 bits)
- Global clock for all outputs
- Individual rate configuration for each output
- Pulse train outputs can be halted and resumed at the same point

The MAX78000 provides up to four instances of the pulse train engine peripheral, as PT0, PT1, PT2, and PT3. Only PT1 at port pin P0_19 is available on the MAX78000FTHR development board.

In pulse train mode, bits are sent out with the LSB bit sent out first.
Two example projects are given in this section to show how the pulse train engine can be used.

## 10.2 • Project 1 – Generate a pulse train with a specified sequence

**Description**: In this project, an 8-bit pulse train is generated with the sequence 0x96 and bit rate set to 2 bits/second. The generated pulse sequence is **1001 0110**, with the LSB bit sent out first. The waveform is output from port P0_19 of the MAX78000 development board.

**Aim**: This project aims to show how a pulse train can be generated using the MAX78000.

**Circuit diagram**: Port P0_19 (pin 20) of the MAX78000FTHR development board is connected to an oscilloscope to display the generated waveform.

**Program listing**: Figure 10.1 shows the program listing (**PT**). At the beginning of the program the bit rate and the required bit pattern are specified:

```
#define CONT_WV_BPS          2             // Bit rate (Bits/sec)
#define CONT_WV_PATTERN      0x96          // Bit pattern (1001 0110)
```

The main program initialises the pulse train and calls function **PulseTrain** to create the required waveform. Inside this function, the PT channel number is set to 1 (i.e. port pin

P0_19), the bit rate is set to 2, the number of bits is set to 8, the required pattern is specified, the continuous loop is set to 0 so that the loop is repeated continuously, and the delay is set to 0 so that there are no delays between the loops. The pulse train is then configured and is started.

```
ptConfig.channel = 1;                              // PT1 (Port P0_9)
ptConfig.bps = CONT_WV_BPS;                        // Bit rate
ptConfig.ptLength = 8;                             // No of bits
ptConfig.pattern = CONT_WV_PATTERN;                // Pattern
ptConfig.loop = 0;                                 // Continuous loop
ptConfig.loopDelay = 0;                            // No delay

MXC_PT_Config(&ptConfig);                          // Configure PT
MXC_PT_Start(MXC_F_PTG_ENABLE_PT1);                // Start PT1
```

```
/*-------------------------------------------------------------------------------
                        GENERATE PULSE TRAIN

In this project the pulse sequence 0x96 (1001 0110) is generated and output from
port P0_9 of the MAX78000 development board.

Author: Dogan Ibrahim
Date  : March 2021
Work  : PT
--------------------------------------------------------------------------------*/
#include <mxc.h>

#define        CONT_WV_BPS            2                // Bit rate (Bits/sec)
#define        CONT_WV_PATTERN        0x96             // Bit pattern (1001 0110)

mxc_pt_cfg_t ptConfig;

//
// This function configures the Pulse Train PT1 (channel 1)
//
void PulseTrain(void)
{
    ptConfig.channel = 1;                              // PT1 (Port P0_9)
    ptConfig.bps = CONT_WV_BPS;                        // Bit rate
    ptConfig.ptLength = 8;                             // No of bits
    ptConfig.pattern = CONT_WV_PATTERN;                // Pattern
    ptConfig.loop = 0;                                 // Continuous loop
    ptConfig.loopDelay = 0;                            // No delay

    MXC_PT_Config(&ptConfig);                          // Configure PT
    MXC_PT_Start(MXC_F_PTG_ENABLE_PT1);                // Start PT1
}


int main(void)
{
    MXC_PT_Init(MXC_PT_CLK_DIV1);                         // Initialize PT

    PulseTrain();                                         // Call function PulseTrain

    while (1);
}
```

Figure 10.1 Program listing

Figure 10.2 shows the signal generated by the program on a digital oscilloscope. Notice the pulse train is **1001 0110** with the LSB sent first. i.e. from left to right the generated sequence is: **0110 0110**.

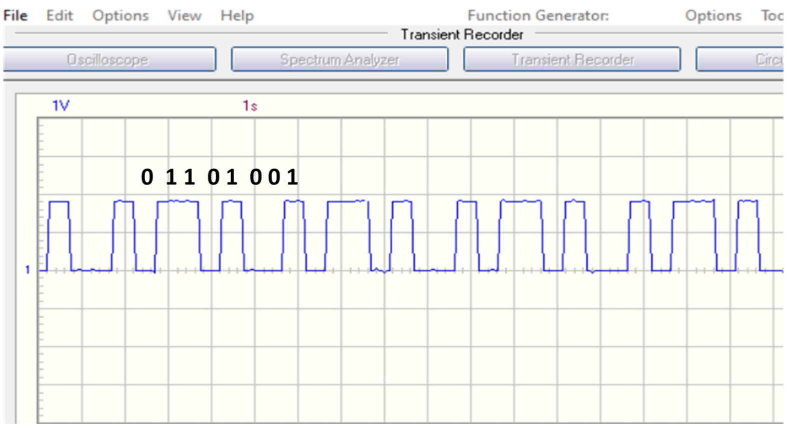


Figure 10.2 Signal generated

### 10.3 • Project 2 – Generate a pulse train with a specified frequency

**Description**: In this project, a 2 kHz square wave signal is generated using the Pulse Train Engine. The duty cycle is set to 50% by default. The output is available on pin P0_19.

**Aim**: This project aims to show how the Pulse Train Engine can be used to generate a square wave signal.

**Circuit diagram**: Port P0_19 (pin 20) of the MAX78000FTHR development board is connected to an oscilloscope to display the generated waveform.

**Program listing**: Figure 10.3 shows the program listing (**PTSQR**). The frequency is defined as 2 kHz. The main program calls function **SquareWave**. Inside this function channel 1 and the frequency are specified, and the pulse train is started:

```
    uint32_t freq = SQ_WV_HZ;                // Frequency (Hz)
    MXC_PT_SqrWaveConfig(1, freq);           // Configure channel & freq
    MXC_PT_Start(MXC_F_PTG_ENABLE_PT1);     // Enable PT1
```

```
/*---------------------------------------------------------------------------------
                        GENERATE SQUARE WAVE

In this project the pulse train engine is used to generate a 2 kHz square wave
signal with 50% duty cycle

Author: Dogan Ibrahim
Date  : March 2021
Work  : PTSQR
---------------------------------------------------------------------------------*/
#include <mxc.h>

#define SQ_WV_HZ 2000                                        // 2 kHz square wave

//
// Generate square wave with frequency 2000 Hz, 50% duty cycle on channel 1
// (port pin P0_19), enable PT1
//
void SquareWave(void)
{
    uint32_t freq = SQ_WV_HZ;                                // Frequency (Hz)
    MXC_PT_SqrWaveConfig(1, freq);                           // Configure channel & freq
    MXC_PT_Start(MXC_F_PTG_ENABLE_PT1);                      // Enable PT1
}


int main(void)
{
    MXC_PT_Init(MXC_PT_CLK_DIV1);                            //init PT with the clock

        SquareWave();                                        // Generate square wave

    while (1) {}
}
```

Figure 10.3 Program listing

Figure 10.4 shows the waveform generated on an oscilloscope. Here, the horizontal axis was 0.5 ms and the vertical axis was 1V/division. The frequency is 2 kHz and the duty cycle is 50%.

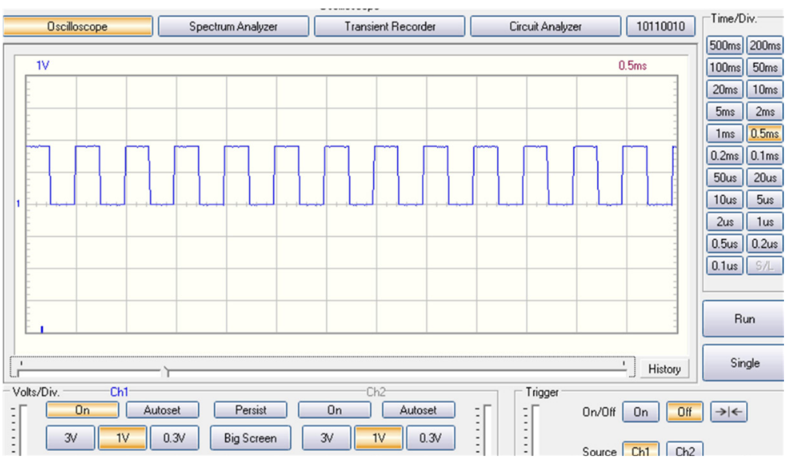


Figure 10.4 Signal generated

# Chapter 11 • True Random Number Generator Engine (TRNG)

## 11.1 • Overview

The True Random Number Generator Engine (TRNG) is a non-deterministic engine that can be used to generate true random numbers in security applications, such as encryption keys (e.g. as AES keys), or in other applications which require true random numbers.
In this chapter, we will develop an application to generate random numbers.

## 11.2 • Project 1 – Generate random numbers

**Description**: In this project, five 32-bit numbers and five 1-byte numbers are generated. The numbers are sent to a PC and displayed on a terminal.

**Aim**: This project aims to show how true random numbers can be generated.

**Circuit diagram**: Figure 11.1 shows the circuit diagram of the project. The MAX78000FTHR development board is connected to a PC through a USB cable as in the previous projects.

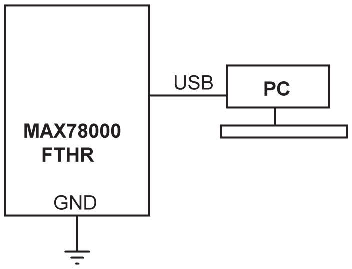


Figure 11.1 Circuit diagram of the project

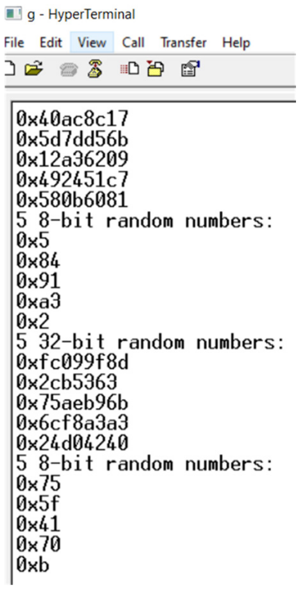
**Program listing**: Figure 11.2 shows the program listing (**RND**). Function **GenerateTRNG** is used to generate the numbers. Function **MXC_TRNG_RandomInt** generates a 32-bit true random number. Similarly, function **MXC_TRNG_random** generates a true random number of the required number of bytes. The results are sent to a terminal emulation software running on a PC.

```
/*------------------------------------------------------------------------------
                        GENERATE RANDOM NUMBERS

In this project 5 32-bit and 5 1-byte numbers are generated and sent to a terminal
through a TTL-USB module

Author: Dogan Ibrahim
Date  : March 2021
Work  : RND
```

```
---------------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "trng.h"

volatile int wait;
volatile int callback_result;

//
// Generate 32-bit and 8-bit random numbers
//
void GenerateTRNG()
{
    uint32_t rnd32;
    uint8_t rnd8;
    int i;

    MXC_TRNG_Init();                                    // Initialize TRNG

    printf("5 32-bit random numbers:\n");
    for(i = 0; i < 5; i++)
    {
        rnd32 = MXC_TRNG_RandomInt();                   // Generate 32-bit number
        printf("0x%x\n", rnd32);                        // Print them
    }

    printf("5 8-bit random numbers:\n");
    for(i = 0; i < 5; i++)
    {
        MXC_TRNG_Random(&rnd8, 1);                      // Generate 1 byte number
        printf("0x%x\n", rnd8);                         // Print them
    }

    MXC_TRNG_Shutdown();                                // Shutdown TRNG engine
}


int main(void)
{
    GenerateTRNG();
}
```

Figure 11.2 Program listing

Figure 11.3 shows an example output from the program. In this project, HyperTerm terminal emulation software is used on a PC. Baud rate was set to 115200 with no flow control.



```
0x40ac8c17
0x5d7dd56b
0x12a36209
0x492451c7
0x580b6081
5 8-bit random numbers:
0x5
0x84
0x91
0xa3
0x2
5 32-bit random numbers:
0xfc099f8d
0x2cb5363
0x75aeb96b
0x6cf8a3a3
0x24d04240
5 8-bit random numbers:
0x75
0x5f
0x41
0x70
0xb
```

Figure 11.3 Example output

# Chapter 12 • 1-Wire Master (OWM)

## 12.1 • Overview

The MAX78000 microcontroller provides a 1-Wire master that can be used to communicate with one or more external 1-Wire slaves.

1-Wire master has the basic features:

- 1-Wire timing generation
- 1-Wire reset generation and presence-pulse detection
- Generation of 1-Wire read and write time slots for single and eight-bit byte transmissions.
- Search ROM Accelerator (SRA) mode, which simplifies the generation of multiple-bit time slots and discrepancy resolution required when completing the Search ROM function to determine the IDs of multiple, unknown 1-Wire slaves on the bus
- Transmit data completion, received data available, presence pulse detection, and 1-Wire line-error condition
- Long-line compensation

## 12.2 • MAX78000 microcontroller 1-Wire pins

1-Wire devices require a bi-directional I/O pin (OWM_IO) for their operation. Optionally, a pull-up enable (OWM_PE) signal can be provided to enable an external pull-up device on the 1-Wire bus. The MAX78000 microcontroller supports the following 1-Wire pins:

1-Wire OWM_IO at P0_6

1-Wire OWM_PE at P0_7

A project is given in this chapter to show how the 1-Wire protocol can be used with the MAX78000 microcontroller.

## 12.3 • Project 1 – DS1820 1-Wire digital thermometer

**Description**: In this project a DS1820 type 1-Wire compatible temperature sensor chip is used. Ambient temperature is measured and sent to a PC where it is displayed.

**Aim**: This project aims to show how the 1-Wire protocol can be used.

**Block diagram**: Figure 12.1 shows the block diagram of the project.

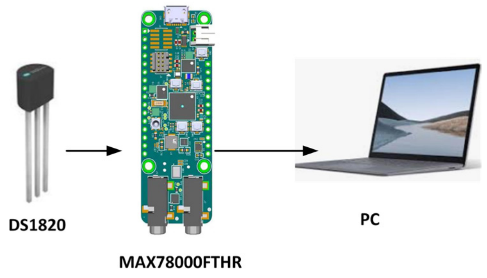


Figure 12.1 Block diagram of the project

**Circuit diagram**: The circuit diagram of the project is shown in Figure 12.2. The DS1820 chip has 3 pins: DQ, GND, and VDD. Pin DQ is the bi-directional input-output pin that is connected to port pin P0_6 of the microcontroller. VDD is connected to +3.3V and GND is connected to the supply ground pin. Notice VDD is optional and can be connected to GND for operation in parasite power mode. In this project, the DS1820 is powered from the +3.3V pin of the MAX78000FTHR development board.

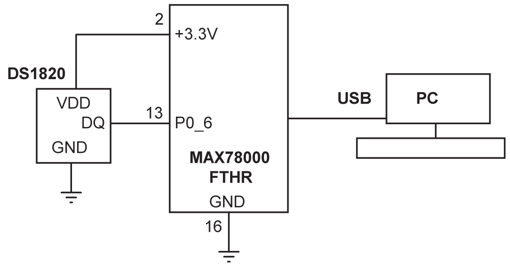


Figure 12.2 Circuit diagram of the project

## DS1820 chip

It is worthwhile to review the operation of the DS1820 temperature sensor chip before developing the program. The DS18S20 digital thermometer chip (Figure 12.3) operates over a 1-Wire interface and provides 9-bit Celsius temperature measurements and has an alarm function with non-volatile user-programmable upper and lower trigger points. The chip can derive power directly from the data line ("parasite power"), eliminating the need for an external power supply. Each DS18S20 chip has a unique 64-bit serial code, allowing multiple DS18S20s to function on the same 1-Wire bus. The chip operates from +3.3V to +5.5V and can measure temperature from -10ºC to +85ºC with ±0.5ºC accuracy (or -55ºC to +125ºC with ±2ºC accuracy). The chip is calibrated to operate in degrees Centigrade (for Fahrenheit applications, a look-up table or a conversion routine can be used).

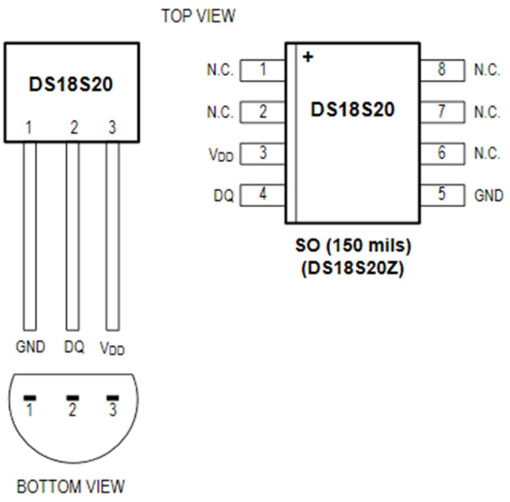


Figure 12.3 The DS1820 chip

The temperature sensor output has a 9-bit resolution, which corresponds to 0.5°C steps. The DS1820 powers up in a low-power idle state.

The transaction sequence is:

1. Initialisation
2. ROM commands
3. Function commands

**Initialisation**: All transactions on the 1-Wire bus begin with an initialisation sequence. The initialisation sequence consists of a reset pulse transmitted by the bus master followed by a presence pulse(s) transmitted by the slave(s). The presence pulse lets the bus master know that slave devices (such as the DS1820) are on the bus and ready to operate.

**ROM commands**: After the bus master has detected a presence pulse, it can issue a ROM command. These commands operate on the unique 64-bit ROM codes of each slave device and allow the master to single out a specific device if many are present on the 1-Wire bus. Some commonly used ROM commands are:

**Search ROM (0xF0)**

When a system is initially powered up, the master must identify the ROM codes of all slave devices on the bus, which allows the master to determine the number of slaves and their device types. If there is only one slave on the bus, which is the case in this project, the simpler Read ROM command can be used in place of the **Search ROM** process.

**Read ROM (0x33)**

This command can only be used when there is only one slave on the bus. It allows the bus master to read the slave's 64-bit ROM code without using the Search ROM procedure.

**Skip ROM (0xCC)**

The master can use this command to address all devices on the bus simultaneously without sending out any ROM code information. For example, the master can make all DS1820s on the bus perform simultaneous temperature conversions by issuing a **Skip ROM** command followed by a **Convert T** (0x44h) command. The **Read Scratchpad** (0xBE) command can follow the Skip ROM command if there is a single slave device on the bus. In this case, time is saved by allowing the master to read from the slave without sending the device's 64-bit ROM code.

**Convert T (0x44)**

This command initiates a single temperature conversion. Following the conversion, the resulting thermal data is stored in the 2-byte temperature register in the scratch-pad memory and the DS1820 returns to its low-power idle state.

**Read Scratchpad (0xBE)**

This command allows the master to read the contents of the scratchpad. The data transfer starts with the LSB bit of byte 0 and continues through the scratchpad until the 9th byte (byte 8 – CRC) is read.

There are some other ROM commands but they are not covered in this book. Interested readers can refer to the DS1820 datasheet.

The steps to read the temperature when there is only one DS1820 on the bus are as follows:

- Master issues a **Skip ROM** command (0xCC)
- Master issues a **Convert T** command (0x44)
- Wait until the data is ready. If the DS18S20 is powered by an external supply, the master can issue "read-time slots" after the Convert command and the DS1820 will respond by transmitting 0 while the temperature conversion is in progress and 1 when the conversion is done.
- Send a Reset command
- Send a Skip ROM command
- Issue a Read Scratchpad command (0xBE)
- The resulting 16-bit sign-extended two's complement temperature data is stored in the scratchpad memory register. The sign bit indicates if the temperature is positive (S = 0) or negative (S = 1)
- DS1820 returns to its idle state

Table 12.1 gives some examples of the temperature/data relationship of the DS1820 chip.

| TEMPERATURE (°C) | DIGITAL OUTPUT (BINARY) | DIGITAL OUTPUT (HEX) |
|---|---|---|
| +85.0* | 0000 0000 1010 1010 | 00AAh |
| +25.0 | 0000 0000 0011 0010 | 0032h |
| +0.5 | 0000 0000 0000 0001 | 0001h |
| 0 | 0000 0000 0000 0000 | 0000h |
| -0.5 | 1111 1111 1111 1111 | FFFFh |
| -25.0 | 1111 1111 1100 1110 | FFCEh |
| -55.0 | 1111 1111 1001 0010 | FF92h |

Table 12.1 Temperature/data relationship

Each DS1820 contains a unique 64-bit code stored in ROM. The least significant 8 bits of the ROM code contain the DS1820's 1-Wire family code: 10h. The next 48 bits contain a unique serial number. The most significant 8 bits contain a cyclic redundancy check (CRC) byte that is calculated from the first 56 bits of the ROM code.

Figure 12.4 shows the memory map of the DS1820 chip. The memory consists of an SRAM scratchpad with nonvolatile EEPROM storage for the temperature alarm (TH and TL) bytes. Byte 0 and byte 1 of the scratchpad contain the LSB and the MSB of the temperature register, respectively.

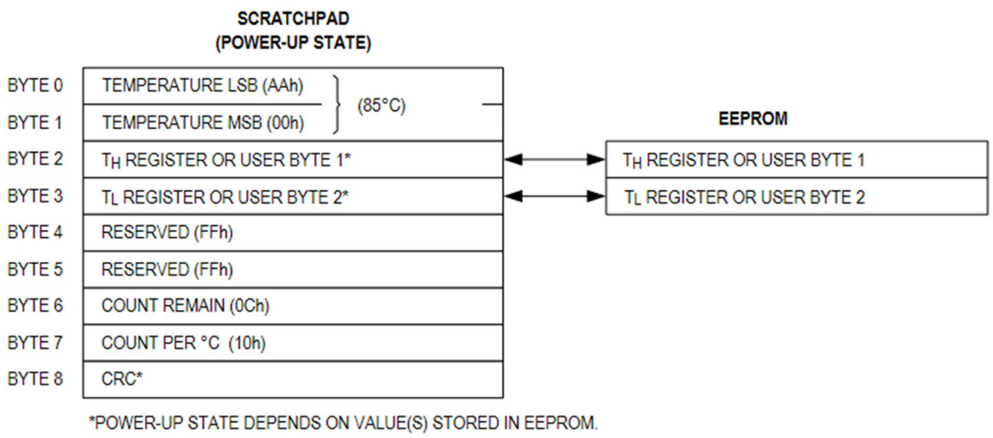


Figure 12.4 Memory map of the DS1820

**Program listing**: Figure 12.5 shows the program listing (**DS1820**). At the beginning of the program, the **ConvertT** and **ReadScractchPad** commands are defined. Inside the main program, P0_6 is configured as an input with the pull-up resistor enabled. The 1-Wire bus is configured by specifying that an internal pull-up resistor is used and there is no long line. The remaining parts of the program run in a **while** loop. Inside this loop, the 1-Wire bus is initialised, the **Skip ROM** command, and the **ConvertT** commands are issued. The program then waits until the data is ready. Port pin P0_6 (i.e. DQ pin of DS1820 is 0 while polling, and becomes 1 when data is ready). The program then resets the bus, issues a **Skip ROM** command and a **ReadScratchPad** command. 9 bytes of data is read into integer array **buffer**.

The MSD and LSD bytes of the temperature are then combined into a 16-bit register called **Temp**. The temperature can be positive or negative. If the temperature is negative, the most significant bit of **Temp** is 1. Negative temperature is in 2's complement form and its value is calculated by taking its complement and adding 1. Finally, the resulting temperature is stored in the floating point variable **Temperature** and displayed on the terminal.

```
MXC_OWM_Init (&owm);                            // Init 1-Wire bus
MXC_OWM_Reset ();                               // Reset the bus
MXC_OWM_WriteByte (SKIP_ROM_COMMAND );          // Send SKIP ROM command
MXC_OWM_WriteByte(ConvertT_COMMAND);            // END ConvertT command
while(MXC_GPIO_InGet(gpio_in.port, DQ) == 0);  // Wait until ready

MXC_OWM_Reset();                                // Reset 1-Wire bus
MXC_OWM_WriteByte(SKIP_ROM_COMMAND);            // Send SKIP ROM command
MXC_OWM_WriteByte(ReadScratchPad_COMMAND);      // Send ReadScratchPad
MXC_OWM_Read(buffer, len);                      // Read ScratchPad (9 bytes)
Temp = (buffer[1] << 8) | buffer[0];            // Temperature in Temp
```

Negative temperature is handled as follows:

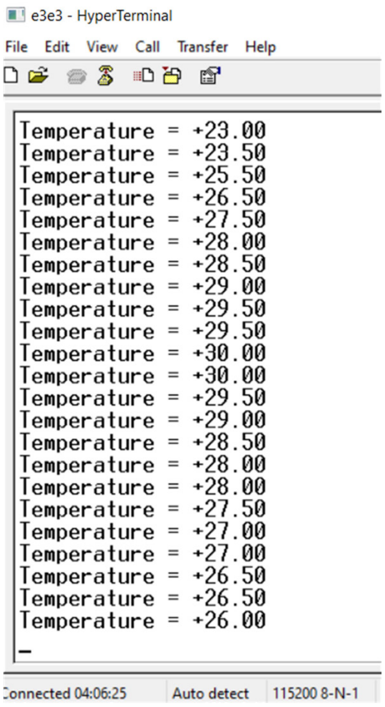
```
if((Temp & 0x8000) != 0)                        // If negative
{
       Temp = ~Temp;                            // Complement
       Temp++;                                  // 2s complement
       Temperature = -Temp * 0.5;               // Negative
}
```

```
/*---------------------------------------------------------------------------------
                          1-WIRE TEMPERATURE SENSOR

In this project a DS1820 type 1-Wire temperature sensor chip is used. The ambient
temperature is read, sent to the PC, and displayed on a terminal emulation window
every second

Author: Dogan Ibrahim
Date  : March 2021
Work  : DS1820
----------------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdint.h>
#include "mxc_device.h"
#include "mxc_sys.h"
#include "owm_regs.h"
```

```
#include "owm.h"
#include "mxc_delay.h"

#define ConvertT_COMMAND   0x44                             // ConvertT
#define ReadScratchPad_COMMAND   0xBE                       // Read ScratchPad

#define DQ MXC_GPIO_PIN_6                                   // DQ at P0_6

mxc_owm_cfg_t owm;
mxc_gpio_cfg_t gpio_in;

//
// Main program
//
int main(void)
{
        int len = 9;
        float Temperature;
        uint16_t Temp;
        uint8_t buffer[9];

        //
        // Configure input pin at P0_6 (DQ)*/
        //
        gpio_in.port = MXC_GPIO0;                           // PORT0
        gpio_in.mask = DQ;                                  // Pin DQ
        gpio_in.pad = MXC_GPIO_PAD_PULL_UP;                 // Internal pull-up
        gpio_in.func = MXC_GPIO_FUNC_IN;                    // Input
        MXC_GPIO_Config(&gpio_in);

        //
        // Configure 1-Wire bus
        //
        owm.int_pu_en = 1;                                  // Internal pull-up
        owm.ext_pu_mode = MXC_OWM_EXT_PU_UNUSED;            // No external pull-up
        owm.long_line_mode=0;                               // No long line

        while(1)
        {
                MXC_OWM_Init (&owm);                        // Init 1-Wire bus
                MXC_OWM_Reset ();                           // Reset the bus
                MXC_OWM_WriteByte (SKIP_ROM_COMMAND );      // Send SKIP ROM command

                MXC_OWM_WriteByte(ConvertT_COMMAND);        // END ConvertT command
                while(MXC_GPIO_InGet(gpio_in.port, DQ) == 0);      // Wait until ready
                MXC_OWM_Reset();                           // Reset 1-Wire bus
                MXC_OWM_WriteByte(SKIP_ROM_COMMAND);       // Send SKIP ROM command
                MXC_OWM_WriteByte(ReadScratchPad_COMMAND); // Send ReadScratchPad command
                MXC_OWM_Read(buffer, len);                 // Read ScratchPad (9 bytes)

                Temp = (buffer[1] << 8) | buffer[0];       // Temperature

                if((Temp & 0x8000) != 0)                   // If negative
                {
                        Temp = ~Temp;                      // Complement
                        Temp++;                            // 2s complement
                        Temperature = -Temp * 0.5;         // Negative
                }
                else
                        Temperature = Temp * 0.5;                // Positive
                printf("Temperature = %+5.2f\n", Temperature);   // Display temperature
                MXC_Delay(1000000);                              // Wait 1 second
        }
}
```

Figure 12.5 Program listing

Figure 12.6 shows example output from the program.



```
Temperature = +23.00
Temperature = +23.50
Temperature = +25.50
Temperature = +26.50
Temperature = +27.50
Temperature = +28.00
Temperature = +28.50
Temperature = +29.00
Temperature = +29.50
Temperature = +29.50
Temperature = +30.00
Temperature = +30.00
Temperature = +29.50
Temperature = +29.00
Temperature = +28.50
Temperature = +28.00
Temperature = +28.00
Temperature = +27.50
Temperature = +27.00
Temperature = +27.00
Temperature = +26.50
Temperature = +26.50
Temperature = +26.00
_
```

Figure 12.6 shows example output from the program

# Chapter 13 • I2S Bus Interface

## 13.1 • Overview

I²S (or I2S, or Inter-IC Sound) is a serial audio interface, created by Philips, for communicating PCM data information between devices. I2S was created in the 1980s for the transmission of audio data using a standardised interface. It is a two-channel protocol because it was designed for stereo sound.

As shown in Figure 13.1, in I2S bus applications, 3 signals are mainly used: Data is on the SD line, the WS line corresponds to the audio channel (left or right) that is being transmitted, and the SCK carries the serial clock. A device in the I2S bus can be a master or a slave. The WS and SCK are generated by the master. A master can be a transmitter or a receiver.

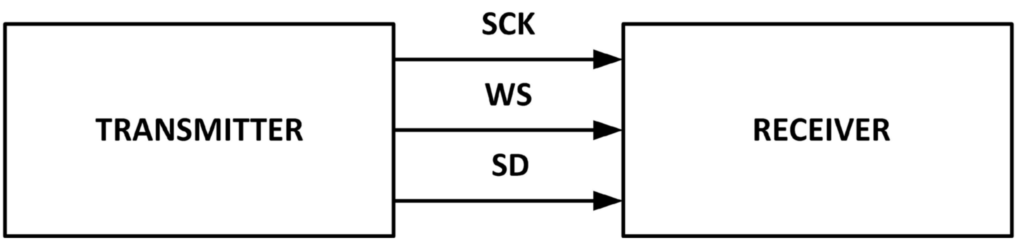


Figure 13.1 Basic I2S bus

**Serial data (SD)**: The serial data is transmitted to MSB first. The master and slave do not need to have an agreed word length. The slave receives whatever is sent by the master. Data is out from the master on the rising or the falling edge of the clock. Data must however be received on the rising edge of the clock.

**Word select (WS)**: A logic HIGH indicates that data is currently being transmitted for the right-channel, and a logic LOW indicates left-channel.

**Clock (SCK)**: The clock runs continuously and the maximum data rate is not specified by the protocol.

Some devices can act both as master and slave, in which case they may have additional lines, such as serial data input (SDI), interrupt lines, etc. Figure 13.2 shows a typical I2S timing diagram.

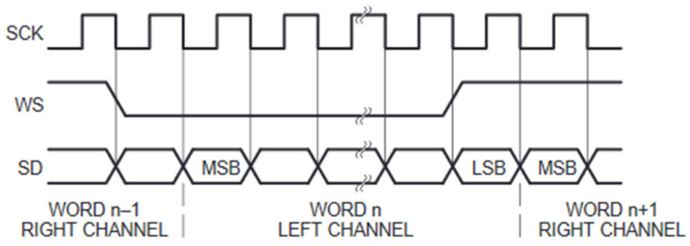


Figure 13.2 I2S timing diagram

## 13.2 ● MAX78000 microcontroller I2S support

The I2S bus on the MAX78000 microcontroller is a bidirectional 4-wire serial bus, having the following basic features:

- Master and slave operation
- 8, 16, 24, and 32-bit frames
- DMA support for both receive and transmit
- Stereo (2 channel) and mono (one channel) formats
- Configurable sampling rate
- Programmable word size
- Word select polarity control
- 8-byte receiver FIFO and 8-byte transmitter FIFO

## 13.3 ● MAX78000 microcontroller I2S pins

MAX78000 microcontroller supports an I2S channel with the name I2S0, having different input (SDI) and output (SDO) lines. The I2S pins are as follows:

| **I2S pin** | **MAX78000 pin** |
|---|---|
| SCK | P1_2 |
| WS | P1_3 |
| SDI | P1_4 |
| SDO | P1_5 |

The MAX78000FTHR development board incorporates a digital microphone, audio CODEC, and a stereo in- and output on 3.5mm sockets placed at the edge of the board. The SPH0645LM4H-B type I2S bus compatible digital microphone is used on the board, having the following basic features:

- 1.8V - 3.6V operating voltage
- 600μA supply current
- 100Hz - 10kHz flat frequency response

The SPH0645LM4H microphone operates as an I2S slave. The master must provide the SCK and WS signals. WS signal must be SCK/64 and synchronized to the SCK. Clock frequencies from 2,048Mhz to 4,096MHz are supported so sampling rates from 32KHz (2,048 / 64) to 64KHz (4,096 / 64) can be used. The Data Format is 24-bit, 2s complement, MSB first, where data precision is 18 bits with unused bits set to zeroes.

Figure 13.3 shows the SPH0645LM4H-B microphone and its connection diagram on the MAX78000FTHR development board.

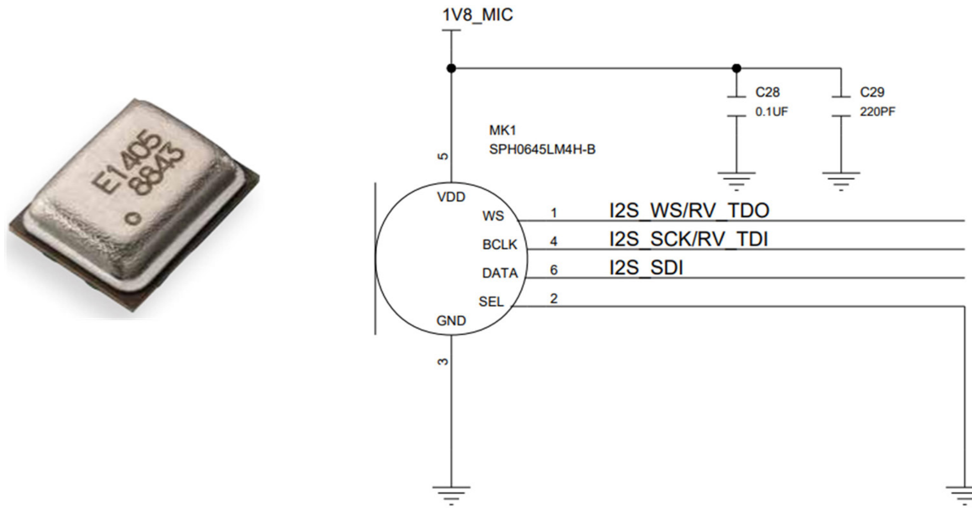


Figure 13.3 SPH0645LM4H-B microphone

A project is given in this chapter to show how the I2S bus can be used.

## 13.4 ● Project 1 - I2S Bus – receiving microphone data

**Description**: This project shows how microphone data can be received. In this project the microphone captures data either until the specified receive buffer is full, or if the elapsed time is one second. Messages are sent to a terminal as the program is started and data is captured.

**Program listing**: Figure 13.4 shows the program listing (I2S). The program uses a one shot timer with the timeout period set to one second. Data capturing starts when the one shot is started. At the end of one second, the timer stops and a timer interrupt is generated. A flag is set in the timer interrupt service routine to inform the main program that one second has elapsed. The program stops when the receive buffer is full, or when the timer expires after one second.

At the beginning of the program the receive buffer size, receive buffer, and I2S interrupt flag are defined:

```
#define RX_BUFFER_SIZE  30000                     // I2S RX buffer size
int32_t RX_BUFFER[RX_BUFFER_SIZE];                // I2S BUFFER
volatile uint8_t I2S_Flag = 0;
int Wait_Flag = 0;
```

Also, the one shot timer parameters and the timer interrupt flag are defined:

```
#define OST_TIMER           MXC_TMR0                      // Using TMR0
#define OST_CLOCK_SOURCE    MXC_TMR_32K_CLK               // Timer clock
volatile uint8_t Timer_Flag = 0;                          // Timer flag
```

The main program enables the cache, sets the system clock to 100MHz, initialises and turns ON the microphone, and initialises the receive buffer. The I2S parameters are then set as follows:

Sample size of 32, normal polarity, left channel used, MSB first clock divider of 5, and the receive buffer and its size specified.

```
    req.wordSize    = MXC_I2S_DATASIZE_WORD;
    req.sampleSize  = MXC_I2S_SAMPLESIZE_THIRTYTWO;
    req.justify     = MXC_I2S_MSB_JUSTIFY;
    req.wsPolarity  = MXC_I2S_POL_NORMAL;
    req.channelMode = MXC_I2S_INTERNAL_SCK_WS_0;
    req.stereoMode  = MXC_I2S_MONO_LEFT_CH;
    req.bitOrder    = MXC_I2S_MSB_FIRST;
    req.clkdiv      = 5;
    req.rawData     = NULL;
    req.txData      = NULL;
    req.rxData      = RX_BUFFER;
    req.length      = RX_BUFFER_SIZE;
```

The I2S clock rate is given by:

I2S clock = 12,288MHz / (2*(clkdiv+1)) = 1,024MHz

The one shot timer parameters are defined inside function **OneShotTimer** as follows:

```
    tmr.pres = TMR_PRES_128;                      // Prescaler
    tmr.mode = TMR_MODE_ONESHOT;                  // Mode=One shot
    tmr.bitMode = TMR_BIT_MODE_32;                // 32-bit
    tmr.clock = OST_CLOCK_SOURCE;                 // Clock source
    tmr.cmp_cnt = 254;                            // For 1 second
    tmr.pol = 0;
```

The one shot timer is set for one second (see Chapter 9) so it generates an interrupt when the timer expires.

Receive FIFO threshold is then configured and I2S interrupts are enabled. Also, one shot timer interrupts are enabled. The I2S interrupt service routine is named **I2SHandler**. Similarly, the one shot timer interrupt service routine is named **OneshotTimerHandler**. The remaining parts of the program run in a **while** loop. Inside this loop, the program waits until microphone data arrives, reads the data, and shifts it right by 14 bits since the actual data is 18 bits. The data capturing continues until the received buffer is full, or until the one shot timer expires. The captured data in the integer array **buf_current** is displayed on a terminal emulation program running on a PC. Remember to set the Baud rate to 115,200 with 8 bits of data, no parity, 1 stop bit, and no flow control.

```
/*--------------------------------------------------------------------------------
                        I2S - CAPTURE MICROPHONE DATA

This project shows how data can be captured from the microphone on the MAX7800FTHR
development board. Data is captured for 1 second or until the specified buffer is
full

Author: Dogan Ibrahim
File  : I2S
Date  : March 2020
----------------------------------------------------------------------------------
*/
#include <stdio.h>
#include <string.h>
#include "board.h"
#include "icc.h"
#include "i2s.h"
#include "i2s_regs.h"
#include "max20303.h"
#include "nvic_table.h"
#include "mxc_device.h"
#include "mxc_sys.h"
#include "mxc.h"

//
// I2S Definitions
//
#define RX_BUFFER_SIZE  30000                           // I2S RX buffer size
int32_t RX_BUFFER[RX_BUFFER_SIZE];                      // I2S BUFFER
volatile uint8_t I2S_Flag = 0;
int Wait_Flag = 0;
mxc_i2s_req_t req;

//
// OneShot timer definitions
//
```

```
#define OST_TIMER           MXC_TMR0                    // Using TMR0
#define OST_CLOCK_SOURCE    MXC_TMR_32K_CLK             // Timer clock
volatile uint8_t Timer_Flag = 0;                        // Timer flag
mxc_tmr_cfg_t tmr;

//
// One shot interrupt service routine. Set variable Timer_Flag when oneshot finished
//
void OneshotTimerHandler()
{
    MXC_TMR_ClearFlags(OST_TIMER);                      // Clear timer interrupt
    Timer_Flag = 1;                                     // Set timer flag
}

//
// I2S interrupt service routine. Set variable I2S_Flag when data is captured
//
void I2SHandler(void)
{
    I2S_Flag = 1;                                       // Set I2S flag
    MXC_I2S_ClearFlags(MXC_F_I2S_INTFL_RX_THD_CH0);     // Clear interrupt flag
}

//
// This function implements the OneShot timer. The timer expires after 1 second
// and this causes a timer interrupt where a flag is set to stop data capturing
//
void OneShotTimer()
{
    MXC_TMR_Shutdown(OST_TIMER);                        // Stop the timer
//
// Set timer parameters
//
    tmr.pres = TMR_PRES_128;                            // Prescaler
    tmr.mode = TMR_MODE_ONESHOT;                        // Mode=One shot
    tmr.bitMode = TMR_BIT_MODE_32;                      // 32-bit
    tmr.clock = OST_CLOCK_SOURCE;                       // Clock source
    tmr.cmp_cnt = 254;                                  // For 1 second
    tmr.pol = 0;                                        // Passive Polarity

    MXC_TMR_Init(OST_TIMER, &tmr, true);                // Initialize timer clock
    MXC_TMR_EnableInt(OST_TIMER);                       // Enable timer
    MXC_TMR_Start(OST_TIMER);                           // Start timer
}

//
```

```
// Start of MAIN program
//
int main()
{
    uint32_t cnt, i, rx_size;
    int32_t* buf_current, *buf_start;

    MXC_ICC_Enable(MXC_ICC0);                          // Enable cache
    MXC_SYS_Clock_Select(MXC_SYS_CLOCK_IPO);           // System clock to 100 MHz
    SystemCoreClockUpdate();

    max20303_init(MXC_I2C1);                           // Init microphone
    max20303_mic_power(1);                             // Turn ON microphone
    MXC_Delay(MXC_DELAY_MSEC(200));                    // Wait 200ms

    memset(RX_BUFFER, 0, sizeof(RX_BUFFER));           // Clear I2S_RX buffer

    /* Configure I2S interface parameters */
    req.wordSize    = MXC_I2S_DATASIZE_WORD;           // Datasize Word
    req.sampleSize  = MXC_I2S_SAMPLESIZE_THIRTYTWO;    // Sample size 32
    req.justify     = MXC_I2S_MSB_JUSTIFY;             // MSB justify
    req.wsPolarity  = MXC_I2S_POL_NORMAL;              // Polarity normal
    req.channelMode = MXC_I2S_INTERNAL_SCK_WS_0;
    req.stereoMode  = MXC_I2S_MONO_LEFT_CH;            // Only left channel
    req.bitOrder    = MXC_I2S_MSB_FIRST;               // MSB first
    req.clkdiv      = 5;                               //
    req.rawData     = NULL;                            //
    req.txData      = NULL;                            // No TX data
    req.rxData      = RX_BUFFER;                       // I2S buffer
    req.length      = RX_BUFFER_SIZE;                  // I2S buffer size

    MXC_I2S_Init(&req);                                // Init I2S

    MXC_I2S_SetRXThreshold(4);                         // RX FIFO threshold
    NVIC_SetVector(I2S_IRQn, I2SHandler);              // I2S interrupt vector
    NVIC_EnableIRQ(I2S_IRQn);
    MXC_I2S_EnableInt(MXC_F_I2S_INTEN_RX_THD_CH0);     // Enable RX FIFO thresh int
    MXC_I2S_RXEnable();                                // Enable RX

    NVIC_SetVector(TMR0_IRQn, OneshotTimerHandler);    // Timer interrupt handler
    NVIC_EnableIRQ(TMR0_IRQn);                         // Enable timer interrupts

    buf_start = &RX_BUFFER[0];                         // Buffer start
    buf_current = buf_start;

    i = 0;
    OneShotTimer();                                     // Call OneShot timer

    while(Timer_Flag == 0)
    {
        while (I2S_Flag == 0);                          // Wait for I2S interrupt

        I2S_Flag = 0;                                             // Clear flag
        rx_size = MXC_I2S->dmach0 >> MXC_F_I2S_DMACH0_RX_LVL_POS; // RX FIFO samples
        i = i + rx_size;

        if(i > RX_BUFFER_SIZE)break;                            // If buffer is full

        while (rx_size--)
        {
            *buf_current++ = ((int32_t) MXC_I2S->fifoch0) >> 14; // 18BitActualvalue
        }

       if(!Wait_Flag && *(buf_current-1) != 0)
        {
                printf("Capturing microphone data...\n");
                Wait_Flag = 1;
        }
    }

//
// End of data capture from microphone (Either buffer is full or timer has elapsed)
//
    buf_current=buf_start;
    for(cnt = 0; cnt < i; cnt++)printf("%x",*buf_current++);
    printf("\nEnd\n");
}
```

Figure 13.4 Program listing

# Chapter 14 • Using the Camera

## 14.1 • Overview

The MAX78000FTHR development board has an onboard OVM7692 type miniature camera (link: https://www.ovt.com/sensors/OVM7692). There is a small plastic lid on the camera lens that must be removed before using it. This is a 640 x 480 pixel VGA camera in a 1/3 inch optical format, having the following basic features (see Figure 14.1):

- Automatic exposure control (AEC)
- Automatic gain control (AGC)
- Automatic black level calibration (ABLC)
- 50/60Hz luminance detection
- Programmable gamma correction
- Flip, scaling, and windowing
- Image quality controls
- 2.8V power supply with built-in 1.5V regulator
- Digital video port
- 1.15mm focal length
- Output format: RAW 8-bit, YUV422, RGB565/444
- Built-in 10-bit ADC
- Input clock frequency: 6-24 MHz
- Maximum image transfer rate: VGA (640x480 – 30fps), CIF (352x288 – 20fps), QVGA(320x240 – 60fps), QCIF(176x144 – 60fps)

Figure 14.1 shows the OVM7692 camera and its interface to the MAX78000FTHR development board.

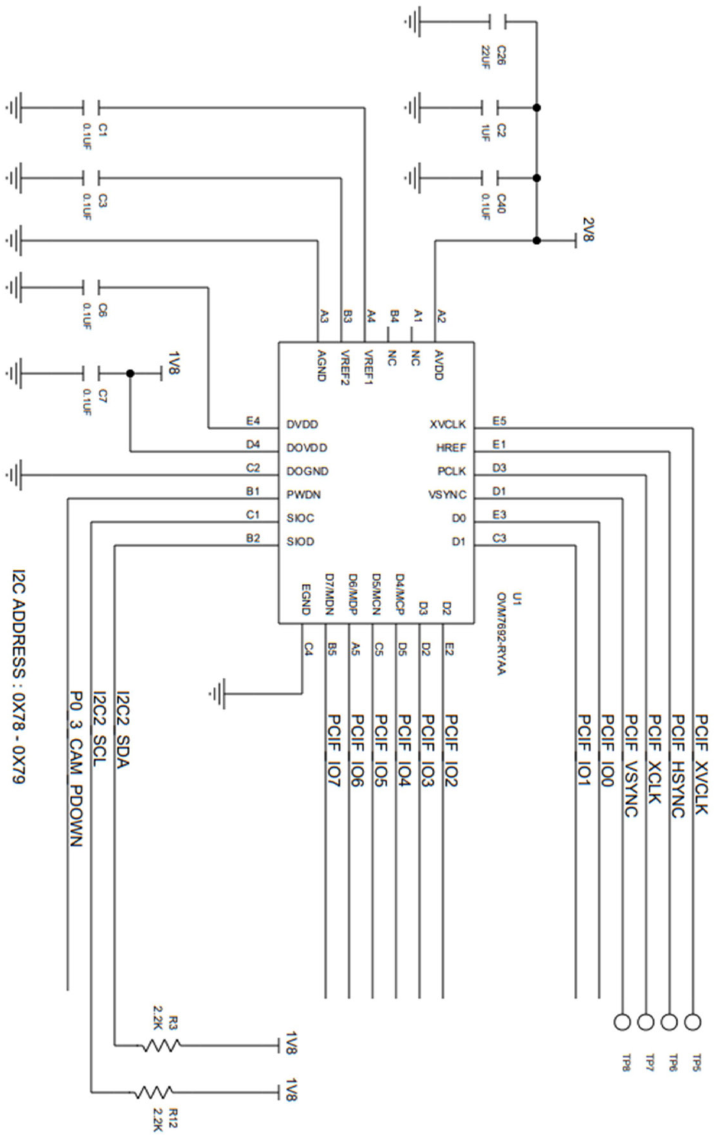


Figure 14.1 OVM7692 camera

In this chapter, we will learn how to use the camera in a project.

## 14.2 • Project 1 – Using the camera

**Description**: In his project, an image is captured and its properties are displayed on a terminal using emulation software (e.g. Putty, Terraterm, HyperTerm, etc).

**Camera functions**

The following camera functions are commonly used (see file **camera.h** for full descriptions):

**int camera_init(camera_freq, MXC_TMRn)**: This function initialises the camera interface. This function must be called before any other camera functions. **camera_freq** is the camera frequency and **MXC_TMRn** is the timer used.

**int camera_get_slave_address()**: returns the I2C slave address of the camera. The onboard camera slave address in this project is 0x3C.

**int camera_get_product_id(&id)**: returns the product ID of the camera. The on-board camera product ID in this project is 7692

**int camera_get_manufacture_id(&id)**: returns the manufacturer ID of the camera. The on-board camera manufacturer ID in this project is 0x7FA2

**int camera_dump_registers()**: dump all registers of the camera

**int camera_reset()**: reset the sensor to its default state

**int camera_setup(int xres, int yres, pixformat_t pixformat, fifomode_t fifo_mode, dmamode_t dma_mode, int dma_channel)**: this function sets the camera resolution, pixel format, expand bits option, FIFO byte mode, and DMA option.

RGB888 format is supported by expanding from RGB565 by camera interface hardware. With RGB888 we have 8-bits of Red, Green, and Blue, making 24-bits. The following pixel formats are supported:

```
PIXFORMAT_GRAYSCALE          // 2BPP/GRAYSCALE
PIXFORMAT_RGB444             // 2BPP/RGB444
PIXFORMAT_RGB565             // 2BPP/RGB565
PIXFORMAT_RGB888             // 3BPP/RGB888
PIXFORMAT_YUV422             // 2BPP/YUV422
PIXFORMAT_BAYER              // 1BPP/RAW
```

Supported FIFO modes are:

**FIFO_THREE_BYTE**
**FIFO_FOUR_BYTE**

Supported DMA modes are:

**NO_DMA**
**USE_DMA** (the DMA channel must be specified)

Supported expand bit modes are:

**BIT_EXPAND_OFF**
**BIT_EXPAND_565_TO_888**

**int camera_set_contrast(int level)**: set the sensor contrast level (-3 to +3)

**int camera_set_brightness(int level)**: set the sensor brightness level (-3 to +3)

**int camera_set_saturation(int level)**: set the sensor saturation level (-3 to +3)

**int camera_start_capture_image(void)**: start to capture image

**int camera_is_image_rcv(void)**: check whether all image data received or not

**uint8_t* camera_get_pixel_format(void)**: retrieve the camera pixel format

**void camera_get_image(uint8_t** img, uint32_t* imgLen, uint32_t* w, uint32_t* h)**: get the camera frame buffer, image length and resolution

As an example, to return and display the I2C slave address of the on-board camera on a terminal, use the following statements:

```
CamAddress = camera_get_slave_address();
printf("Camera slave address is: %x\n", CamAddress);
```

Similarly, to return the on-board camera product ID, use the following statements. It is always good programming practice to check the return status of functions:

```
RetStatus = camera_get_product_id(&id);
if (RetStatus != STATUS_OK)
{
        printf("Error reading camera id, error: %d\n", RetStatus);
        return -1;
}
```

Figure 14.2 shows the program listing (**CAMCAPTURE**). At the beginning of the program, the resolution of the image and camera frequency are defined. A DMA channel is thennn specified and the camera is initialised.

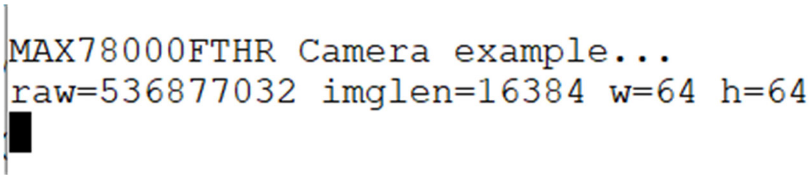
The camera is set up with the following parameters:

x-resolution: 64
y-resolution: 64
pixel format: RGB888
FIFO byte format: FIFI_THREE_BYTE
DMA used

An image is then captured by calling function **camera_start_capture_image**. When an image is received, its properties are displayed by calling function **camera_get_image**. Figure 14.3 shows the data displayed on a terminal emulation screen.

```
/*-------------------------------------------------------------------------------
                           CAPTURE CAMERA DATA

This project shows how an image can be captured from the camera on the MAX7800FTHR
development board

Author: Dogan Ibrahim
File  : CAMCAPTURE
Date  : March 2020
-----------------------------------------------------------------------------------*/
#include <stdio.h>
#include "mxc_device.h"
#include "board.h"
#include "mxc_delay.h"
#include "camera.h"
#include "dma.h"

#define IMAGE_XRES  64                                          // X-resolution
#define IMAGE_YRES  64                                          // Y-resolution
#define CAMERA_FREQ 10000000                                    // Frequency (10M)


int main(void)
{
    int dma_channel;
    uint8_t*   RawData;
    uint32_t  ImageLength, ImageWidth, ImageHeight;

    MXC_DMA_Init();                                             // Initialize the DMA
    dma_channel = MXC_DMA_AcquireChannel();                     // DMA channel
    printf("\nMAX78000FTHR Camera example...\n");               // Heading
    camera_init(CAMERA_FREQ, MXC_TMR1);                         // Initialize the camera

//
// Setup the camera properties: camera resolution,pixel format,FIFO byte mode,DMA channel
//
    camera_setup(IMAGE_XRES, IMAGE_YRES, PIXFORMAT_RGB888, FIFO_THREE_BYTE, USE_DMA, dma_channel);

    camera_start_capture_image();                               // Capture image

//
// Capture an image and display its properties
//
    while(1)
    {
        if (camera_is_image_rcv())                              // If image received...
        {
            camera_get_image(&RawData, &ImageLength, &ImageWidth, &ImageHeight);
            printf("raw=%d imglen=%d w=%d h=%d\n", RawData,ImageLength,ImageWidth,ImageHeight);
            MXC_Delay(SEC(10));
            break;
        }
    }
```

Figure 14.2 Program listing



```
MAX78000FTHR Camera example...
raw=536877032 imglen=16384 w=64 h=64
```

Figure 14.3 Data displayed on a terminal emulation screen

# Chapter 15 • The Instruction Cache

## 15.1 • Overview

The instruction cache helps speed up the CPU processing. In this chapter, we will present an example project to see how enabling the cache improves the performance enormously.

## 15.2 • Project 1 – Enabling/disabling the instruction cache

**Description**: In this project, we will write a program and measure the processing speed by enabling and then disabling the instruction cache. Terminal emulation software is started on a PC at 115200 Baud to display the data.

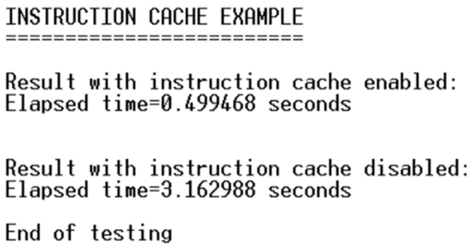
**Program listing**: Figure 15.1 shows the program listing (Cache).

```
/*------------------------------------------------------------------
                INSTRUCTION CACHE ENABLE/DISABLE

This program displays the elapsed time when executing a time consuming
function with and without the instruction cache

Author: Dogan Ibrahim
Date  : March 2021
Work  : Cache
-------------------------------------------------------------------*/
#include <stdio.h>
#include "mxc_device.h"
#include "icc.h"
#include "tmr.h"

#define ICC MXC_ICC0

//
// This is a dummy function which does some calculations to waste some time
//
void Dummy(void)
{
    float i, j, k = 0.0;

    for (i = 1; i < 1000; i++)
    {
        for (j = 1; j < 5000; j++)
        {
            k = k + i * j;
        }
    }
}

//
// This function starts the timer
//
void start_timer(void)
{
    MXC_TMR_SW_Start(MXC_TMR0);
    return;
}

//
// This function stops the timer and calculates the elapsed time
//
void stop_timer(void)
{
    int time_elapsed = MXC_TMR_SW_Stop(MXC_TMR0);        // Elapsed time in us
    float sec = (float)time_elapsed / 1000000.0;         // In seconds
    printf("Elapsed time=%f seconds\n", sec);            // Display elapsed time
}

//
// Start of MAIN program
//
int main(void)
{
    printf("\nINSTRUCTION CACHE EXAMPLE\n");
    printf("========================\n");

    MXC_ICC_Flush(ICC);

    printf("\nResult with instruction cache enabled:\n");
    MXC_ICC_Enable(ICC);                                 // Enable ICC
    start_timer();                                       // Start stopwatch
    Dummy();                                             // Dummy function
    stop_timer();                                        // Stop stopwatch

    printf("\n\nResult with instruction cache disabled:\n");
    MXC_ICC_Disable(ICC);                                // Disable ICC
    start_timer();                                       // Start stopwatch
    Dummy();                                             // Dummy function
    stop_timer();                                        // Stop stopwatch

    printf("\nEnd of testing\n");
    while (1);
}
```

Figure 15.1 Program listing

The program initially enables the cache, starts the stopwatch, and calls function **DUMMY** which executes nested **for** loops to waste time. The stopwatch is stopped at the end of the function and the elapsed time is displayed in seconds. Then, the cache is disabled and the same process is repeated by starting the stopwatch, calling the same function, and then stopping the stopwatch. Again the elapsed time is displayed on the terminal. It is clear from the figure that the processing time with the instruction cache enabled is much faster.

Figure 15.2 shows the processing powers with and without the instruction cache.



```
INSTRUCTION CACHE EXAMPLE
=========================

Result with instruction cache enabled:
Elapsed time=0.499468 seconds


Result with instruction cache disabled:
Elapsed time=3.162988 seconds

End of testing
```

Figure 15.2 Example output

The following functions are available for controlling the instruction cache (see header file icc.h for more details):

**MXC_ICC_Enable (mxc_icc_regs_t* icc)**: Enable the instruction cache controller

**MXC_ICC_Disable (mxc_icc_regs_t* icc)**: Disable the instruction cache controller

**MXC_ICC_Flush (mxc_icc_regs_t* icc)**: Flush the instruction cache controller

**MXC_ICC_ID (mxc_icc_regs_t* icc, mxc_icc_info_t cid)**: Read the data from the cache ID register

The enumeration type for the cache ID register is:

```
ICC_INFO_RELNUM         // Identifies the RTL release version
ICC_INFO_PARTNUM        // Specifies the value of C_ID Port Number
ICC_INFO_ID             // Specifies the value of Cache ID
```

# Chapter 16 ● Using a TFT Display

## 16.1 ● Overview

A TFT display is a kind of LCD that uses thin-film-transistor technology. These displays are active-matrix LCDs and offer improved image qualities compared to standard passive matrix LCDs. TFT-based displays have a transistor for each pixel on the screen. As a result, each pixel can be controlled at a fast rate resulting in bright and smooth displays. Several layers of filters are used sandwiched between two glass panels. Standard TFT displays use a white back-light to generate the picture. Newer panels utilise LEDs to generate their light. There are several differences between the LCDs and TFT. A standard LCD works well for low-resolution simple displays but performs poorly for larger more graphic display intended designs, such as televisions, computer displays, games consoles, etc. Before TFTs, it was not possible to use fast graphics with standard LCDs in game consoles. There are several variants of TFT displays, such as TN, STN, FSTN, DSTN, etc. Such displays create high contrast, sharp and fast images compared to standard LCDs. Another important advantage of TFT displays is that they are available in many colours. Perhaps one of the main disadvantages of TFT displays is their higher cost compared to LCDs. TFT displays also use more power.

In this chapter, we will learn how to use TFT displays in MAX78000FTHR applications. A 2.4 inch TFT colour display will be used in a project as an example.

## 16.2 ● 2.4 inch TFT display

The display used in this Chapter is the 2.4-inch **Adafruit FeatherWing** TFT display with touch screen (Figure 16.1), having the following features:

- 240x320 pixels
- 16-bit color pixel control
- On-board SD card slot
- SPI interface
- Size: 65 x 53 x 9.5mm
- 2.4 Diagonal LCD TFT display with resistive touch screen
- 4-white LED backlight
- On-board 3.3 V (300mA LDO regulator)
- 5 V compatible, use with 3.3 V or 5 V logic

Figure 16.2 shows the rear side of the display with the pin configuration. The display has two columns of pins which is compatible with the MAX78000FTHR development board. All you have to do is plug the display into the development board (see Figure 16.3, where the rear side of the display is shown).

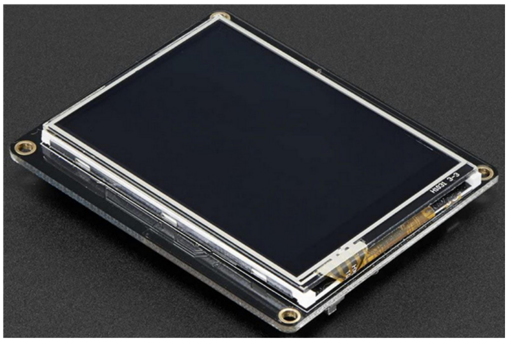


Figure 16.1 2.4 inch TFT display

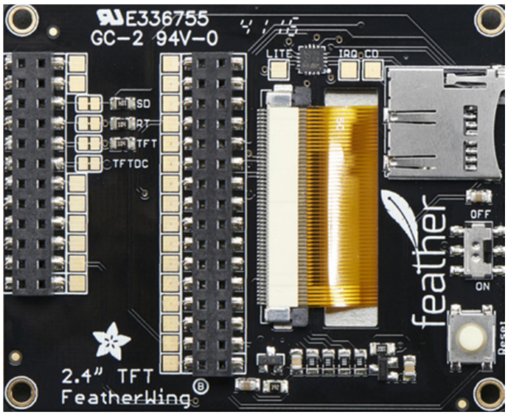


Figure 16.2 Rear side of the display

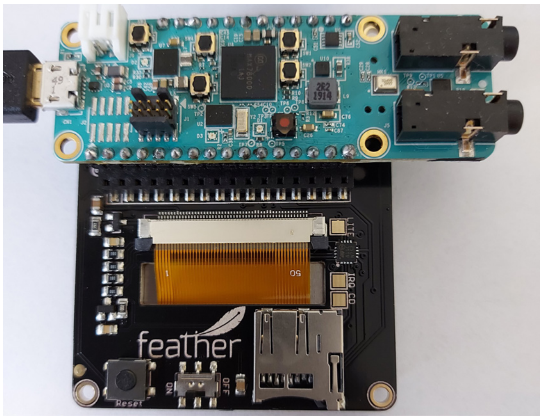


Figure 16.3 Connecting the display to the development board

Figure 16.4 shows the coordinates of the display, where (0, 0) is the top left corner, X-axis runs horizontally from left to right, and Y-axis runs vertically from top to bottom.

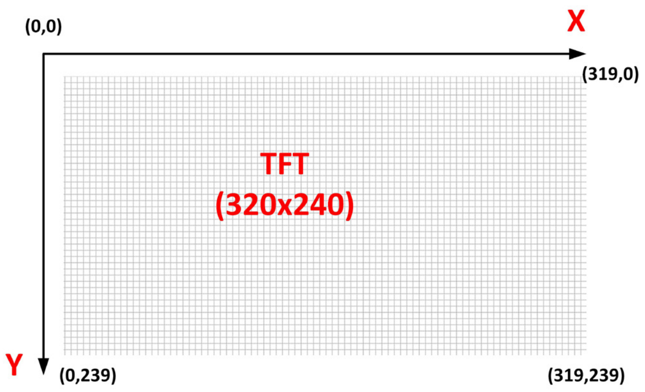


Figure 16.4 Coordinates of the screen

## 16.3 ● Project 1 – Displaying various shapes and text on the display

**Description**: In this project, we display various geometrical shapes and text on the TFT display.

**Aim**: This project aims to show how a TFT display can be used with the MAX78000FTHR development board.

**Program listing**: The shapes and texts shown in Figure 16.5 are drawn on the display. The coordinates of the shapes and the texts are shown in the Figure. Figure 16.6 shows the program listing (**TFTDEMO**). The program draws the following shapes and writes the texts. Some delay is introduced between each output to see the shapes being drawn:

- Set background color to white
- Clear the screen
- Draw a line
- Draw a rectangle
- Draw a circle
- Draw a circle filled with green colour
- Draw a rectangle filled with red colour
- Write text MAX78000 in font Arial 12x12
- Write text MAX78000 in font Arial 24x23

The instruction cache is enabled and the system clock speed is set to 100 MHz to speed up the processing.

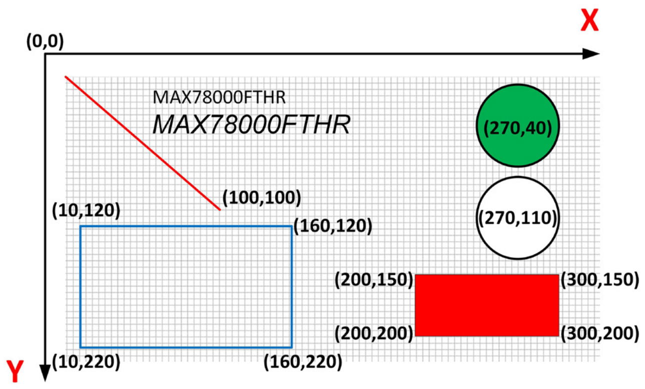


Figure 16.5 Drawing shapes on the display (not to scale)

```
/*--------------------------------------------------------------------
        TFT - DISPLAYING VARIOUS SHAPES AND TEXT

In this program a 2.4 inch FeatherWing TFT display is connected to
the MAX78000FTHR development board. Various shapes and text are drawn

Author: Dogan Ibrahim
Date  : March 2021
Work  : TFTDEMO
---------------------------------------------------------------------*/
#include "mxc.h"
#include "icc.h"
#include "mxc_device.h"
#include "mxc_delay.h"
#include "utils.h"
#include "state.h"
#include "tft_fthr.h"

text_t text;

//
// Display various shapes and text
//
void DisplayTest(void)
{
        area_t _area;
        area_t* area;

        MXC_TFT_SetRotation(ROTATE_90);                // 90 Degrees rotation
        MXC_Delay(SEC(2));
        MXC_TFT_SetBackGroundColor(WHITE);             // Background colour White
        MXC_Delay(SEC(2));
        MXC_TFT_ClearScreen();                           // Clear screen
        MXC_Delay(SEC(1));

        MXC_TFT_Line(0, 0, 100, 100, RED);               // Draw a line
        MXC_Delay(SEC(1));
        MXC_TFT_Rectangle(10, 120, 160, 220, BLUE);      // Draw a rectangle
        MXC_Delay(SEC(1));
        MXC_TFT_Circle(270, 110, 30, BLACK);             // Draw a circle
        MXC_Delay(SEC(1));
        MXC_TFT_FillCircle(270, 40, 30, GREEN);          // Draw a filled circle
        MXC_Delay(SEC(1));

        area = &_area;
        area->x = 200;                                   // X coordinate
        area->y = 150;                                   // Y coordinate
        area->w = 100;                                   // Width
        area->h = 50;                                    // HEight
        MXC_TFT_FillRect(area, RED);                     // Draw a filled rectangle
        MXC_Delay(SEC(1));

        text.data = "MAX78000";                          // Text to be displayed
        text.len = 8;
        MXC_TFT_PrintFont(60, 10, (int)&Arial12x12[0], &text, NULL);
        MXC_TFT_PrintFont(60, 30, (int)&Arial24x23[0], &text, NULL);
}


int main(void)
{
        MXC_Delay(200000);                       // Wait for 1.8V to be available
        MXC_ICC_Enable(MXC_ICC0);                // Enable instr cache

        MXC_SYS_Clock_Select(MXC_SYS_CLOCK_IPO);         // Set clock to 100MHz
        SystemCoreClockUpdate();

        MXC_TFT_Init(MXC_SPI0, 1, NULL, NULL);           // Initialize TFT display
        DisplayTest();                                   // Call display

        while(1);
}
```

Figure 16.6 Program listing

Figure 16.7 shows the output of the program on the display.

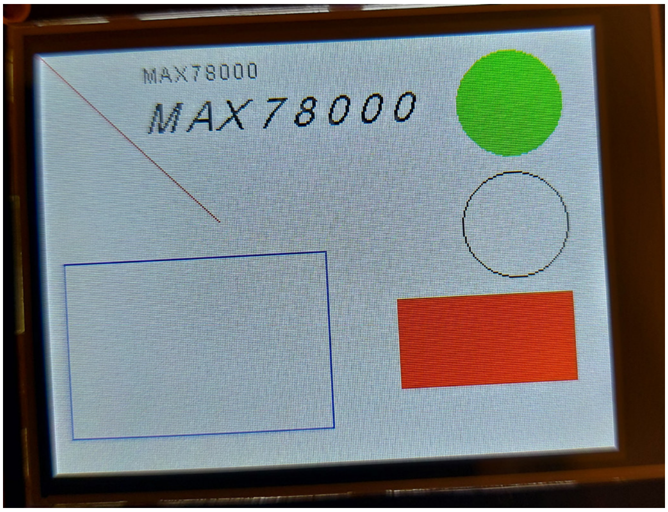


Figure 16.7 Output of the program

The MAX78000FTHR supports large number of TFT display functions (see header file **tft-fthr.h** for full details). Some commonly used functions are described below:

**MXC_TFT_Init(mxc_spi_regs_t* tft_spi, int ss_idx, mxc_gpio_cfg_t* reset_ctrl, mxc_gpio_cfg_t* bl_ctrl)**: Initialize the TFT display. The arguments are:

**tft_spi**: The SPI instance the TFT is connected to

**ss_idx**: The SSEL index to use when communicating with the attached TFT

**reset_ctrl**: The GPIO pin configuration for the TFT's reset pin. Use NULL if the reset pin of the TFT is not connected to the microcontroller.

**bl_ctrl**: The GPIO pin configuration for the backlight enable pin. Use NULL if the microcontroller does not have control of the backlight enable.

**MXC_TFT_Backlight(int on)**: Turn the backlight ON

**MXC_TFT_ClearScreen(void)**: Clear the screen

**MXC_TFT_FillRect(area_t* area, int color)**: Draw and fill a rectangle. The arguments are:

area_t: Location and size of rectangle

color: Palette index of rectangle color

area_t is a structure with the following members

| | | |
|---|---|---|
| **uint16_t** | **x** | **(X coordinate)** |
| **uint16_t** | **y** | **(Y coordinate)** |
| **uint16_t** | **w** | **(Width in pixels)** |
| **uint16_t** | **h** | **(Height in pixels)** |

color can take the following values:

**BLACK**
**NAVY**
**DARK_GREEN**
**DARK_CYAN**
**MAROON**
**PURPLE**
**OLIVE**
**LIGHT_GREY**
**DARK_GREY**
**BLUE**
**GREEN**
**CYAN**
**RED**
**MAGENTA**
**YELLOW**
**WHITE**
**ORANGE**
**GREEN_YELLOW**

**MXC_TFT_WritePixel(int pixelX, int pixelY, int width, int height, uint32_t color)**: Write a pixel in the display. The arguments are:

**pixelX**: x location of image
**pixelY**: y location of image
**width**: width of pixel
**height**: height of pixel
**color**: RGB value of image

**MXC_TFT_SetBackGroundColor(unsigned int index_color)**: Set the background color

**MXC_TFT_SetForeGroundColor(unsigned int color)**: Set the foreground color

**MXC_TFT_ConfigPrintf(area_t* area)**: Set the area of the printf

**MXC_TFT_ResetCursor(void)**: Set cursor to top left for printf

**MXC_TFT_Print(int x0, int y0, text_t* str, area_t* area)**: Print string with current font

**MXC_TFT_SetRotation(tft_rotation_t rotation)**: Set TFT screen rotation
tft_rotation_t is a structure with following members:

  **ROTATE_0**
  **ROTATE_90**
  **ROTATE_180**
  **ROTATE_270**

**MXC_TFT_Circle(int x0, int y0, int r, int color)**: Draw a circle. The arguments are:

x0: x location on screen
y0: y location on screen
r: circle radius
color: circle color

**MXC_TFT_FillCircle(int x0, int y0, int r, int color)**: Fill a circle with specified color

**MXC_TFT_Line(int x0, int y0, int x1, int y1, int color)**: Draw a line from x0,y0 to x1,y1 with specified color

**MXC_TFT_Rectangle(int x0, int y0, int x1, int y1, int color)**: Draw a rectangle with a specified color. x0 and y0 are the starting points x1,y1 are the endpoints.

**MXC_TFT_PrintFont(x, y, font, &text, NULL)**: Display text with the specified font. The following fonts are available:

Arial12x12
Arial24x23
Arial28x28
SansSerif16x16
SansSerif19x19

# Chapter 17 • Convolutional Neural Networks (CNN)

## 17.1 • Overview

The most important feature of the MAX78000 microcontroller that makes it unique is its Convolutional Neural Network (CNN) accelerator. This feature allows the microcontroller to be used in Artificial Intelligence (AI) applications, such as image processing (e.g. face and object recognition), sound processing (e.g. voice-based command system design), machine intelligence, and so on. In this chapter, we will briefly review the principles of Artificial Neural Networks (ANN) which help us to understand CNNs. There are many books, tutorials, and papers published on ANNs and interested readers can find tons of information on the Internet. The principles and use of the MAX78000 CNN accelerator will be presented in later sections of this chapter with some examples.

## 17.2 • Artificial neural networks (ANNs)

ANNs are software imitations of the structure of our brains. Our brains contain neurons that act like organic switches. These switches can change their output states depending on their inputs which can be electrical or chemical stimuli. Neural networks in our brain are a very complex interconnected network of neurons where the output of a neuron may be input to thousands of other neurons. Learning is a result of activating certain neural connections so that these connections are reinforced.

ANNs can be trained in a supervised or unsupervised manner and they attempt to mimic the behaviour of the brain in software. Supervised ANNs learn by providing large numbers of input and output data samples so that they learn and provide the desired output for a given input. For example, an ANN can be trained to learn to identify mature apples by passing large numbers of apples over a conveyor belt. The network can then learn to identify and notify when it encounters unripe apples. Supervised learning is the most commonly used form of ANNs.

Unsupervised learning is done without supervision or a teacher. There is no feedback from the environment as to what should be the desired output (and whether it is correct or not). Here, the inputs are combined to form clusters, and when a new input pattern is applied, the neural network gives an output response to indicate the class to which the input pattern belongs. ANNs attempt to learn the structure of the input data on its own.

## 17.3 • The ANN structure

In biological neurons, the outputs of some neurons are inputs to others. This is represented in software by connected layers of nodes, where each node has multiple weighted inputs and the sum of these inputs generates an output as shown in Figure 17.1. The circle represents the node that takes the weighted inputs, sums them, and inputs them to the activation function. The output of the activation function is shown as yj. Notice weights are not binary values but are real-valued numbers.

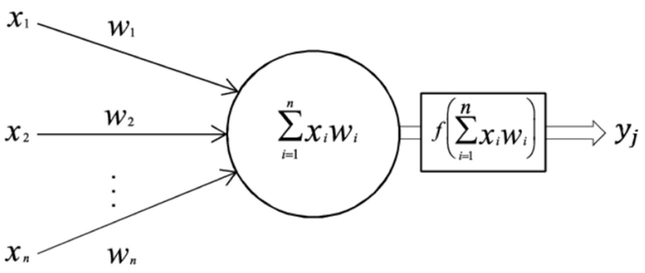


Figure 17.1 A node with inputs

The activation function has switching characteristics such that if its input is greater than a certain value, its output changes state (e.g. from 0 to 1, from -1 to 1, or from 0 to greater than 0), therefore simulating the activation of a biological neuron. A commonly used activation function is the sigmoid function, given by the following equation (note the **sigmoid function** is not the only activation function, and there are many activation functions). As can be seen in Figure 17.2, the function is switched (i.e. activated) from 0 to 1 when the input z is greater than a certain value. Notice that the sigmoid function looks like an S shape and the output does not instantaneously change.

$$\phi(z) = \frac{1}{1+e^{-z}}$$

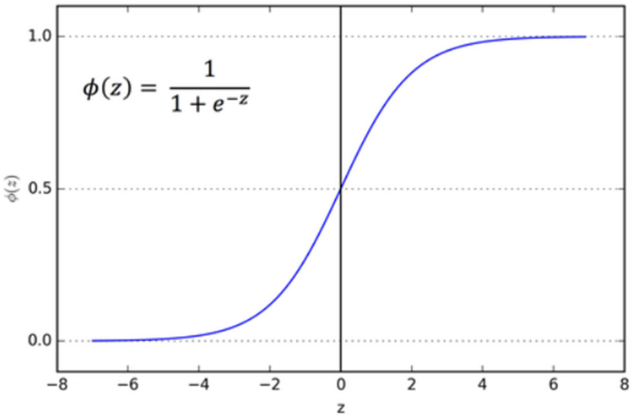


Figure 17.2 The sigmoid function

It is interesting to know how the output changes with the change of the input weightings. Let us consider a simple node with only one input x with weighing W, where the input to the activation function is xW. Figure 17.3 shows the change of the output as the weighing is changed between 0.5 and 2. Note that changing the weighing changes the slope of the output of the sigmoid function.

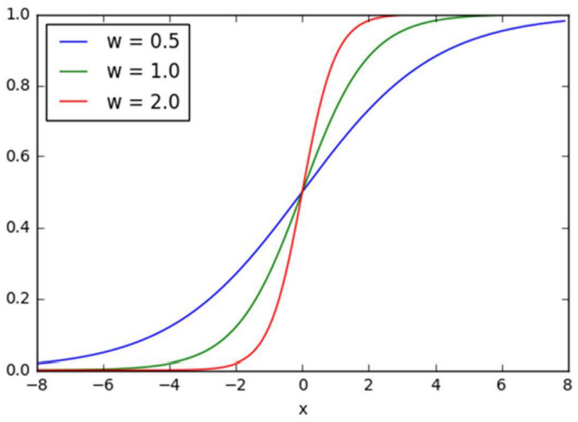


Figure 17.3 Changing the input weighing

In some applications, we may want the output of the sigmoid function to change when its input is greater than or less than 0. This can be done by applying a bias to the input node as shown in Figure 17.4.

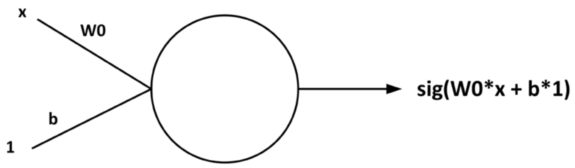


Figure 17.4 Applying bias to the node

Figure 17.5 shows the output as the bias b is changed. Notice that the switching point changes as **b** is changed.

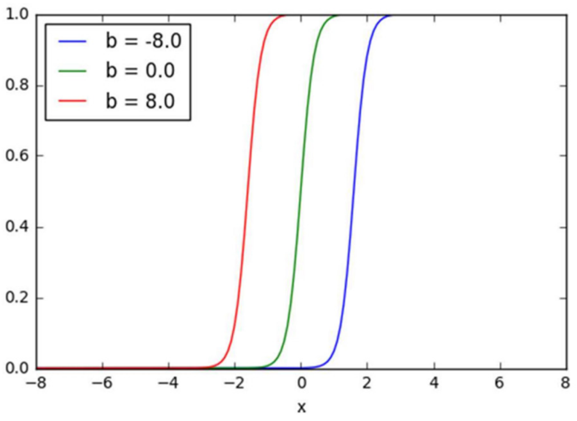


Figure 17.5 Changing the input bias

Just like the biological interconnected network of neurons, ANNs are in the form of interconnected structures. There are three layers: input, hidden, and output. As shown in Figure 17.6, data enters the ANN through the input layer. Hidden layers are between the input and output layers. The output layer is where the output of the ANN is available. There are many connections between the input and hidden layers. Each node of the input layer is connected to all nodes of the hidden layer. Similarly, each hidden layer is connected to the output layer. In real applications, there could be many inputs and outputs.

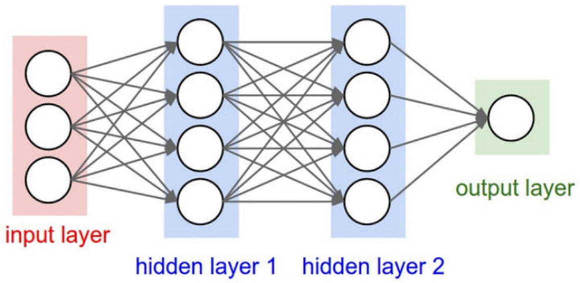


Figure 17.6 ANN structure with multiple hidden layers

A neural network can be in one of two states at any time: learning state (i.e. being trained) or normally operating state (i.e. after being trained). In normal operation, information flows from the input to the output through the hidden layers. This form of data flow in the network is called a feedforward network.

Learning requires feedback and this is also true for neural networks. Children learn by being told what they are doing is right or wrong, and next time they are expected not to do the same thing wrongly. Let us look at a simple learning process:

When an input is given to a neural network, it returns an output. On the first try, the network cannot get the correct output on its own (except by luck). Next, the weights are changed and the output is observed again. The weights are the only variables that can be changed during the learning process. To determine which weights are better to modify, the process called backpropagation is performed. In summary, neural networks learn things by a feedback process called backpropagation. This involves comparing the network output with the expected output and using the difference between them (i.e. the error) to modify the weights of the connections in the network, working from the outputs to the inputs, through the hidden layers. This process of going backward causes the network to learn in time, and hence reduces the error. The learning process requires the network to be trained with enough learning examples.

After a network has learned enough, it can be presented with an entirely new set of inputs it has never seen before and it is expected to respond correctly. For example, suppose you have trained a network by showing it many pictures of cars and busses. If you now show the picture of a lorry, the network will attempt to categorise the lorry as either a car or a bus, depending on its past experience. Properly trained neural networks can have many domestic, commercial, and industrial applications. Some application areas are listed below.

- **Automotive**: Intelligent automotive guidance systems
- **Financial**: Loan advisor, financial analysis, credit application evaluation
- **Electronics**: Chip failure analysis, machine vision
- **Industrial**: Product design and analysis, quality inspection, quality prediction, planning and management
- **Medical imaging**: ECG and EEG analysis, cancer analysis, medical diagnosis, examining pathology reports
- **Speech**: Speech recognition, voice-activated command systems, text to speech conversion
- **Imaging**: Face recognition, object recognition, pattern recognition
- **Signal processing**: hearing aid design
- **Transportation**: vehicle recognition, vehicle scheduling
- **Communication**: Data compression, image compression, real-time spoken language translation
- **Security**: Face recognition, fire detection, brake-in detection
- **Domestic**: Intelligent appliances, such as washing machine, TV, dishwasher, microwave, etc

The backpropagation algorithm is complex and requires good knowledge of higher mathematics, and it is not covered in any more detail in this book. Interested readers can find many books, tutorials, and papers on this topic.

## 17.4 ● Convolutional Neural Networks (CNNs)

CNNs are network architectures for deep learning where learning is directly from data, eliminating the need for feature extraction. CNNs are used in recognising images and objects (object recognition and computer vision), as well as in non-image data such as in audio voice recognition, signal data, etc.

Some important features of CNNs are:

- CNNs produce very accurate recognition results
- CNNs eliminate manual feature extraction since the features are learned directly
- CNNs can be used to build on pre-existing networks, thus they can be retained for new recognition tasks

Just like neural networks, CNNs have input, output, and many hidden layers in between. A CNN can have hundreds of layers where each layer learns to detect different features of an image or identify different parts of a sound. Like traditional neural networks, a CNN has neurons with weights and biases. The values of the weights are learned during the training process.

In CNN, instead of feeding entire images into our neural network as one set of numbers, we divide the image into small parts using filters and then feed these parts of the image.

Filters are used at different resolutions, and the output of each convolved image is used as the input to the next layer in the network. The complexity of the filters can vary, as they can start as simple features such as edges or brightness of an image, and then gradually increase in complexity to features that uniquely define the object to be recognised.

The most commonly used operations are: **Convolution**, **Rectified linear unit (ReLU)**, **Pooling (or subsampling)**, and **Classification** as described below briefly. These operations are repeated over many layers, where each layer identifies different features of the object.

**Convolution** is used to put the input through a set of filters, each of which activates certain features of the object. Convolution networks drive their names from the **convolution** operator. The primary purpose of the convolution is to extract features from the input, where small squares of the input data are extracted. An example 5 x 5-pixel image is shown in Figure 17.7, having pixel values of only 0 and 1 (in a grayscale image the pixel values vary between 0 and 255). Also, a 3 x 3-pixel matrix is shown in the same figure. The convolution of the 5 x 5 matrix by the 3 x 3 matrix is computed by multiplying and summing the elements of the two matrices as the 3 x 3 matrix is moved over (or **strided**) the 5 x 5 matrix. The 3 x 3 matrix is known as the **filter** or **feature detector**. The resultant matrix is the **convolved feature** (or the **feature map**).

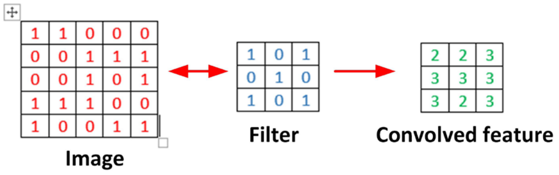


Figure 17.7 5 x 5 and 3 x 3 matrices

The filters slide over the input image to produce the feature maps. The convolution of another filter over the same image gives a different feature map. Users specify the number of filters, filter sizes, the architecture of the network, etc and the CNN learns the values of these filters during the training process. In general, more filters result in more image features to be extracted and this causes the network to recognise unseen image patterns more effectively.

**ReLU** is used after every convolution operation to map the negative values to zero, and maintain only the positive values so that faster and more effective training can be done. ReLU is a non-linear operation with its output shown in Figure 17.8. After ReLU, only the activated features are sent to the next layer.

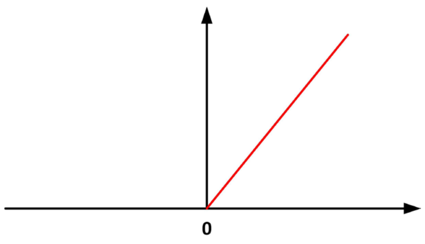


Figure 17.8 ReLu output

**Pooling** reduces the number of parameters (i.e. dimensionality) of each feature map while retaining the most important information. Pooling can be Max, Average, Sum, etc. In Max pooling, we define a window and take the largest element from the feature map within that window (see Figure 17.9). We could also take the average, sum, etc.

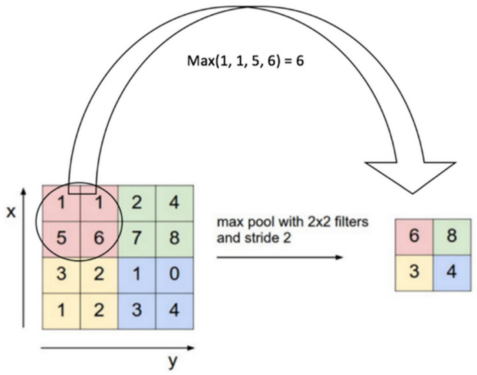


Figure 17.9 Max pooling

Figure 17.10 shows the basic CNN operations applied to an image. In this figure, we have two sets of Convolution, ReLU, and Pooling layers. The 2nd Convolution layer performs convolution on the output of the first Pooling Layer using six filters to produce a total of six feature maps. ReLU is then applied individually on all of these six feature maps. We then perform the Max Pooling operation separately on each of the six rectified feature maps, thus extracting the useful features from the images and reducing the dimensions.

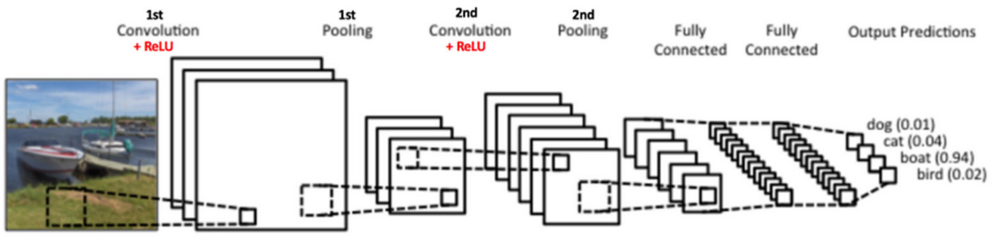


Figure 17.10 Basic CNN operations

The Fully Connected layer uses the features extracted by the convolution and pooling layers to classify the input image based on the training dataset. The Fully Connected layer gives probabilities for the identified objects, where the sum of these probabilities is 1, and the object with the highest probability is the identified object.

Although it is a complex process mathematically, the training of the CNN may be summarised as follows in non-mathematical terms:

- Initialise the parameters and all filters with random values
- Find the output probabilities by using **Forwardpropagation**. i.e. by going through convolution, ReLU, pooling, and classification. The output probabilities will be random for the first training
- Calculate the total error at the output layer using the target and output probabilities
- Use Backpropagation to calculate the gradients of the error concerning all weights and use gradient descent to update all filter values/weights to optimise these parameters and minimise the output error. Repeat the process until the probability of the identified object is high (ideally 1).

Further details on the simplified CNN operation can be obtained from the following links:

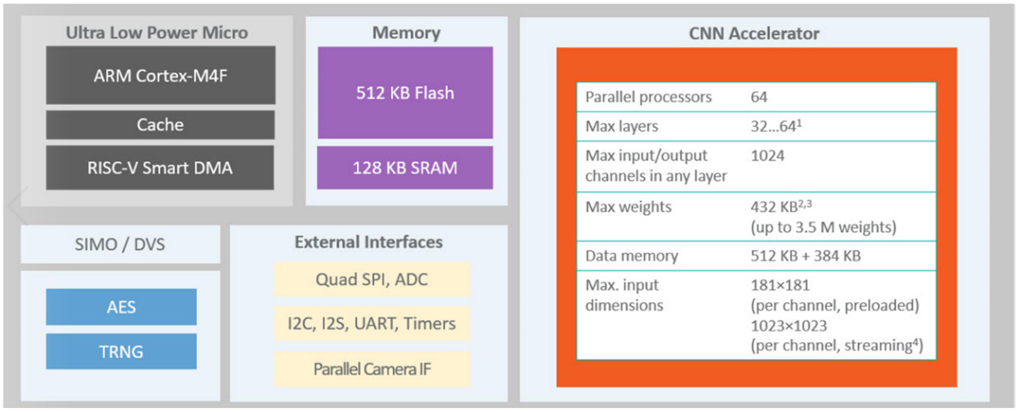
https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

https://medium.com/@ageitgey/machine-learning-is-fun-part-3-deep-learning-and-convolutional-neural-networks-f40359318721#.2gfx5zcw3

## 17.5 • The MAX78000 CNN accelerator

The hardware-based CNN accelerator enables battery-powered applications to execute AI inferences while spending only microjoules of energy. As shown in Figure 17.11, the MAX78000 microcontroller has two functional parts: the embedded microcontroller part, and CNN accelerator. Developing AI applications requires a new set of skills. People who are well trained to create neural networks may find it hard to enter the world of embedded development. Similarly, people with embedded developing training may find it hard to enter the AI field. It is usually difficult to start an AI application from the scratch. The author recommends readers use an existing AI application and modify it for their own needs.



| | |
|---|---|
| Parallel processors | 64 |
| Max layers | 32...64[1] |
| Max input/output channels in any layer | 1024 |
| Max weights | 432 KB[2,3] (up to 3.5 M weights) |
| Data memory | 512 KB + 384 KB |
| Max. input dimensions | 181×181 (per channel, preloaded) 1023×1023 (per channel, streaming[4]) |

Figure 17.11 Embedded part and the CNN engine

The CNN accelerator on the MAX78000 microcontroller consists of 64 parallel processors with 512KB of SRAM-based storage. Each processor includes a pooling unit and a convolutional engine with dedicated weight memory. Four processors share one 32KB data memory. These are further organised into groups of 16 processors that share common controls. A group of 16 processors operates as a slave to another group or independently. Data is read from SRAM associated with each processor and written to any data memory located within the accelerator. A given processor has visibility of its dedicated weight memory and to the data memory instance it shares with the three others.

The full features of the MAX78000 microcontroller CNN can be found at the **MAX78000 User Guide, UG7456; Rev 0; 03/2021**, which can be found on GitHub in the MaximIntegratedAI/MaximAI_Documentation repo. The README file in the MaximIntegratedAI/ai8x-training repo further describes the MAX78000 CNN architecture and features.

## 17.6 ● Demonstration programs

Several AI-based convolutional neural network demonstration programs are supplied with the Eclipse MaximSDK. These programs are available in the following folder:

**MaximSDK\Examples\Maxim78000\CNN**

Users can modify and debug the existing CNN demonstration programs. The following programs are compatible with the MAX78000FTHR development board (all of the AI programs are compatible with the MAX78000EVKIT development board):

- cifar-10
- cifar-100
- mnist
- mnist-riscv
- mnist-streaming
- kws20
- kws20_demo

The steps to load and run a CNN-based program are given below.

## 17.7 • Using the kws20_demo

The applications of digital voice-activated user interfaces have increased in recent years. This is a Keyword Spotting Demo that demonstrates recognition of several keywords using the MAX78000FTHR development board. The program recognises the following 20 words (from a set of 31 words):

**'up', 'down', 'left', 'right', 'stop', 'go', 'yes', 'no', 'on', 'off', 'one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'zero'**

The program recognises the above keywords and reports results and confidence levels. The steps to load this program into Eclipse are as follows:

- Start Eclipse MaximSDK and give a name to the new Workspace (e.g. Workspace20)
- Click **File -> Import** and then, when prompted select **General -> Existing Projects** into Workspace (Figure 17.12) and click **Next**

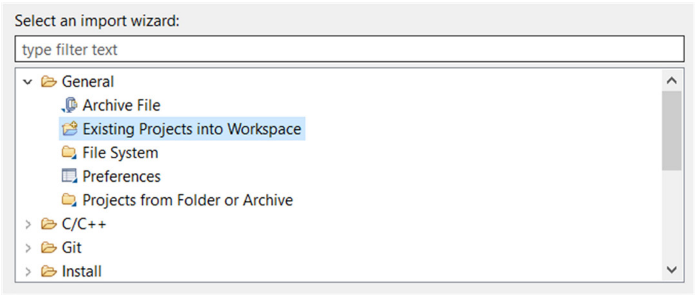
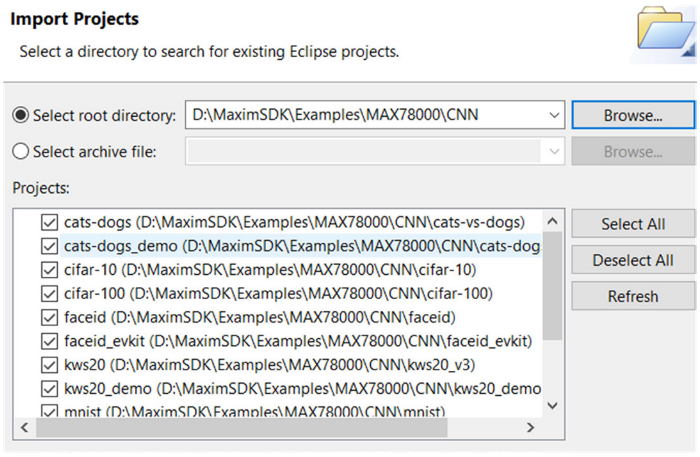


Figure 17.12 Select Existing Projects into Workspace

- Navigate to **Examples\MAX78000\CNN** and you should see a list of all the CNN based demonstration programs (Figure 17.13)



**Import Projects**

Select a directory to search for existing Eclipse projects.

● Select root directory: D:\MaximSDK\Examples\MAX78000\CNN [Browse...]

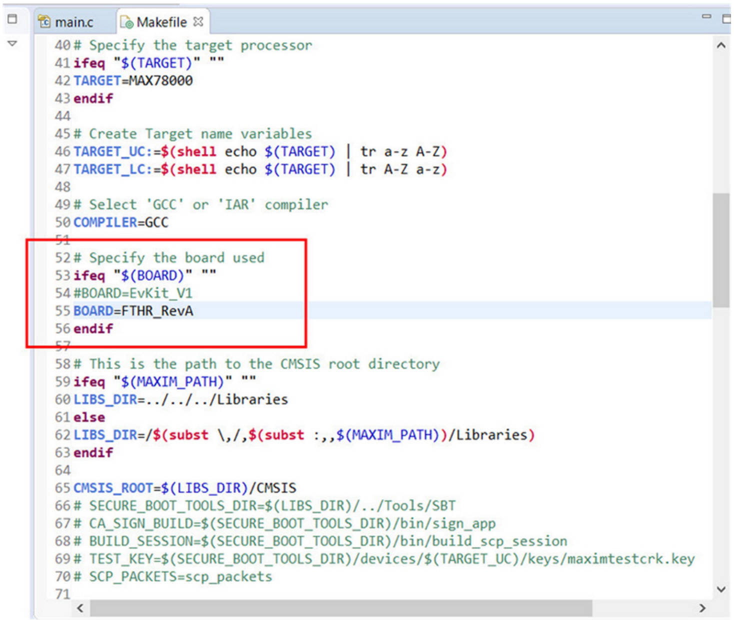
○ Select archive file: [Browse...]

Projects:

- ☑ cats-dogs (D:\MaximSDK\Examples\MAX78000\CNN\cats-vs-dogs)
- ☑ cats-dogs_demo (D:\MaximSDK\Examples\MAX78000\CNN\cats-dog
- ☑ cifar-10 (D:\MaximSDK\Examples\MAX78000\CNN\cifar-10)
- ☑ cifar-100 (D:\MaximSDK\Examples\MAX78000\CNN\cifar-100)
- ☑ faceid (D:\MaximSDK\Examples\MAX78000\CNN\faceid)
- ☑ faceid_evkit (D:\MaximSDK\Examples\MAX78000\CNN\faceid_evkit)
- ☑ kws20 (D:\MaximSDK\Examples\MAX78000\CNN\kws20_v3)
- ☑ kws20_demo (D:\MaximSDK\Examples\MAX78000\CNN\kws20_demo
- ☑ mnist (D:\MaximSDK\Examples\MAX78000\CNN\mnist)

Figure 17.13 CNN demonstration programs

- Click on the left to select **kws20_demo** and click **Finish (make sure to check "copy projects into workspace", otherwise, changes will be made in the source files)**
- You should see the program loaded into the workspace
- All example programs are targeted for the EVKIT by default. To change to the MAX78000FTHR, double click to open file **Makefile**, and change the board type to **FTHR_RevA** as shown in Figure 17.14.

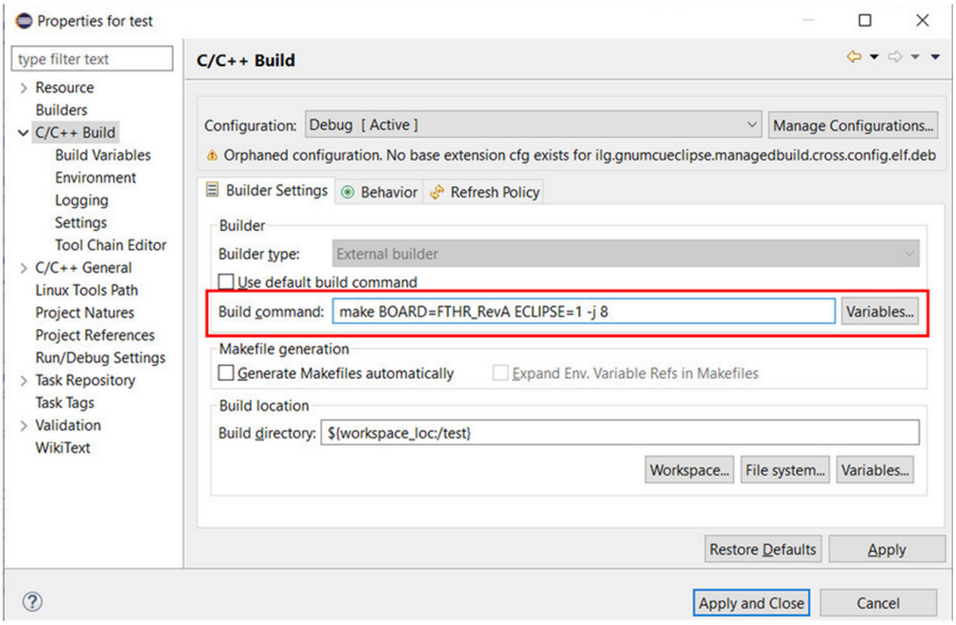


```
# Specify the target processor
ifeq "$(TARGET)" ""
TARGET=MAX78000
endif

# Create Target name variables
TARGET_UC:=$(shell echo $(TARGET) | tr a-z A-Z)
TARGET_LC:=$(shell echo $(TARGET) | tr A-Z a-z)

# Select 'GCC' or 'IAR' compiler
COMPILER=GCC

# Specify the board used
ifeq "$(BOARD)" ""
#BOARD=EvKit_V1
BOARD=FTHR_RevA
endif

# This is the path to the CMSIS root directory
ifeq "$(MAXIM_PATH)" ""
LIBS_DIR=../../../Libraries
else
LIBS_DIR=/$(subst \,/,$(subst :,,$(MAXIM_PATH))/Libraries)
endif

CMSIS_ROOT=$(LIBS_DIR)/CMSIS
# SECURE_BOOT_TOOLS_DIR=$(LIBS_DIR)/../Tools/SBT
# CA_SIGN_BUILD=$(SECURE_BOOT_TOOLS_DIR)/bin/sign_app
# BUILD_SESSION=$(SECURE_BOOT_TOOLS_DIR)/bin/build_scp_session
# TEST_KEY=$(SECURE_BOOT_TOOLS_DIR)/devices/$(TARGET_UC)/keys/maximtestcrk.key
# SCP_PACKETS=scp_packets

```

Figure 17.14 Change the board type to FTHR_RevA

- Click the cross to close **Makefile** and save it

- Click **Project -> Properties** followed by **C/C++ Build** and make the change shown in Figure 17.15.



Properties for test

C/C++ Build

Configuration: Debug [ Active ]

Orphaned configuration. No base extension cfg exists for ilg.gnumcueclipse.managedbuild.cross.config.elf.deb

Builder Settings | Behavior | Refresh Policy

Builder

Builder type: External builder

[ ] Use default build command

Build command: make BOARD=FTHR_RevA ECLIPSE=1 -j 8

Makefile generation

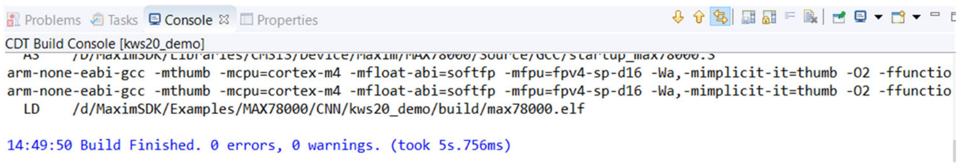
[ ] Generate Makefiles automatically    [ ] Expand Env. Variable Refs in Makefiles

Build location

Build directory: ${workspace_loc:/test}

Figure 17.15 Modify the build configuration

- Click **Apply** and **Close**
- Click **Project** followed by **Clean**
- We are now ready to compile the program
- Click **Project -> Build All** to compile the program
- Check the Console tab at the bottom of the screen and make sure there are no build errors (see Figure 17.16)

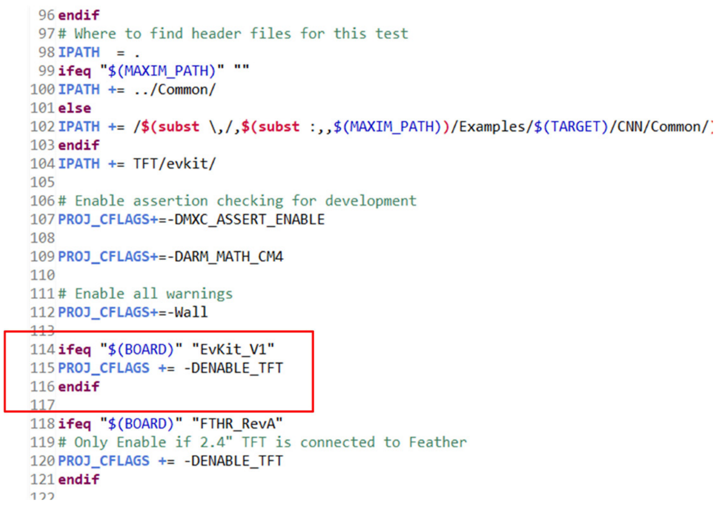


```
CDT Build Console [kws20_demo]
  AS     /D/MaximSDK/Libraries/CMSIS/Device/Maxim/MAX78000/Source/GCC/startup_max78000.s
arm-none-eabi-gcc -mthumb -mcpu=cortex-m4 -mfloat-abi=softfp -mfpu=fpv4-sp-d16 -Wa,-mimplicit-it=thumb -O2 -ffunctio
arm-none-eabi-gcc -mthumb -mcpu=cortex-m4 -mfloat-abi=softfp -mfpu=fpv4-sp-d16 -Wa,-mimplicit-it=thumb -O2 -ffunctio
  LD     /d/MaximSDK/Examples/MAX78000/CNN/kws20_demo/build/max78000.elf

14:49:50 Build Finished. 0 errors, 0 warnings. (took 5s.756ms)
```

Figure 17.16 Make sure there are no build errors

## Testing the program

If you have an Adafruit 2.4-inch FeatherWing TFT display, make sure you plug it into the MAX78000FTHR development board and enable the TFT display by uncommenting the line in the file **Makefile** as shown in Figure 17.17.

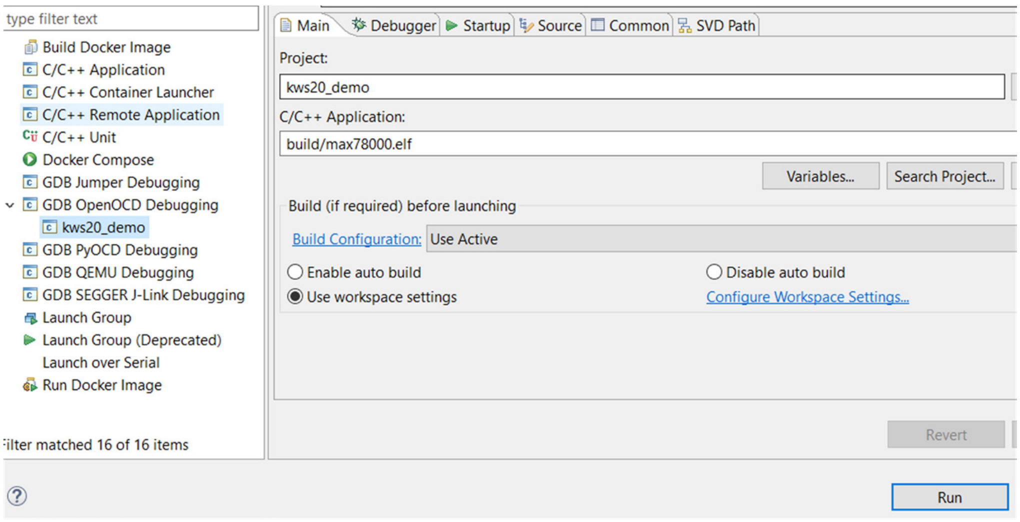


```
endif
# Where to find header files for this test
IPATH  = .
ifeq "$(MAXIM_PATH)" ""
IPATH += ../Common/
else
IPATH += /$(subst \,/,$(subst :,,$(MAXIM_PATH))/Examples/$(TARGET)/CNN/Common/)
endif
IPATH += TFT/evkit/

# Enable assertion checking for development
PROJ_CFLAGS+=-DMXC_ASSERT_ENABLE

PROJ_CFLAGS+=-DARM_MATH_CM4

# Enable all warnings
PROJ_CFLAGS+=-Wall

ifeq "$(BOARD)" "EvKit_V1"
PROJ_CFLAGS += -DENABLE_TFT
endif

ifeq "$(BOARD)" "FTHR_RevA"
# Only Enable if 2.4" TFT is connected to Feather
PROJ_CFLAGS += -DENABLE_TFT
endif
```

Figure 17.17 Enable the TFT display

Also, connect the TTL-USB module to the TX pin of the development board and connect the module to the USB port of your PC. Start a terminal emulation program at 115200 Baud rate.

Click **Run -> Run Configurations** and expand **GDB OpenOCD Debugging**. Click to select **kws20_demo**, and then **Run** to start the program running (Figure 17.18)



Project: kws20_demo

C/C++ Application: build/max78000.elf

Build (if required) before launching

Build Configuration: Use Active

○ Enable auto build    ○ Disable auto build

● Use workspace settings

Figure 17.18 Running the program

You should see the TFT displaying the text as shown in Figure 17.19. Also, the data displayed on the terminal is shown in Figure 17.20. Press button PB1 to start the program.

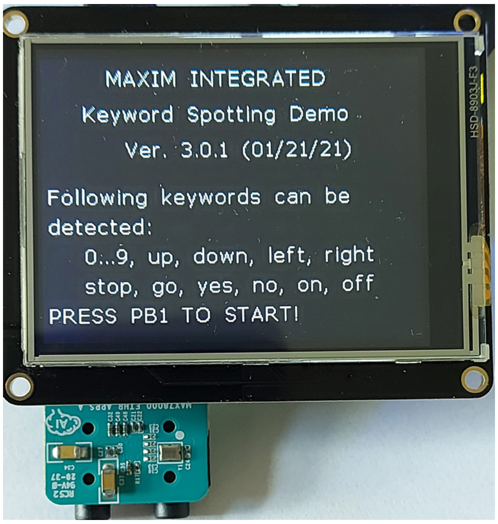

```
MAXIM INTEGRATED
Keyword Spotting Demo
Ver. 3.0.1 (01/21/21)

Following keywords can be
detected:
   0...9, up, down, left, right
   stop, go, yes, no, on, off
PRESS PB1 TO START!
```

Figure 17.19 TFT display

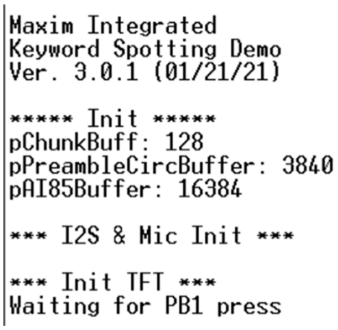

```
Maxim Integrated
Keyword Spotting Demo
Ver. 3.0.1 (01/21/21)

***** Init *****
pChunkBuff: 128
pPreambleCircBuffer: 3840
pAI85Buffer: 16384

*** I2S & Mic Init ***

*** Init TFT ***
Waiting for PB1 press
```

Figure 17.20 Terminal display

Now, say the word **One**. You should see the terminal and TFT display showing the **Detected word** as **One** as shown in Figure 17.21 and 17.22 respectively.

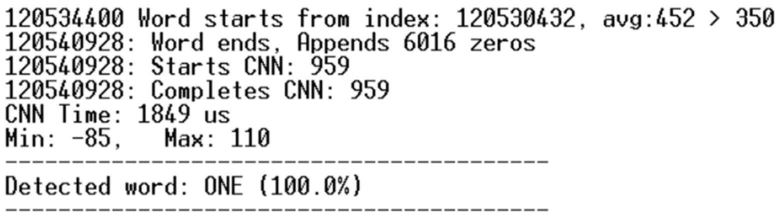
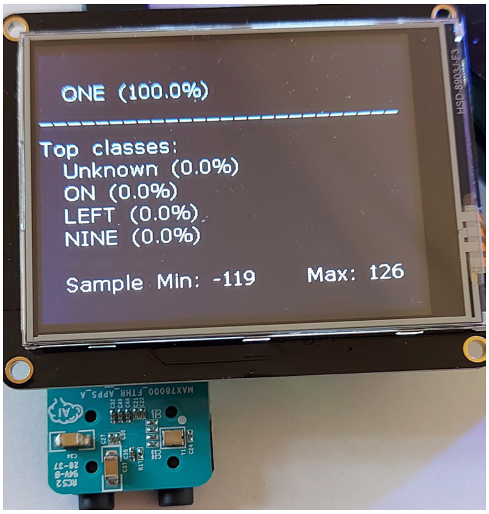

```
120534400 Word starts from index: 120530432, avg:452 > 350
120540928: Word ends, Appends 6016 zeros
120540928: Starts CNN: 959
120540928: Completes CNN: 959
CNN Time: 1849 us
Min: -85,   Max: 110
-----------------------------------------
Detected word: ONE (100.0%)
-----------------------------------------
```

Figure 17.21 The word One is detected (terminal)



```
ONE (100.0%)
------------------------------
Top classes:
  Unknown (0.0%)
  ON (0.0%)
  LEFT (0.0%)
  NINE (0.0%)

  Sample Min: -119     Max: 126
```

Figure 17.22 Word One is detected (TFT display)

## 17.8 ● Project 1 - Modified program - flashing the on-board LED with the spoken word count

In this section, we will modify the **kws20_demo** program so that the onboard red LED flashes the number of times specified by the spoken word. For example, the LED will 5 times if the word FIVE is spoken, etc.

Two parts of the main **kws20_demo** program have been modified. Figure 17.23 shows a function called **Detected_Word** which should be inserted just before the main program. This function uses a built-in function **strcmp** to compare the spoken words (in **keywords[out_class]**) with the numbers **ONE** to **NINE**. For example, if the spoken word is FIVE, the function returns the integer number 5 and so on.

```
int Detected_Word(int16_t out_class)
{
        if(strcmp(keywords[out_class], "ONE") == 0)return 1;
        else if(strcmp(keywords[out_class], "TWO") == 0)return 2;
        else if(strcmp(keywords[out_class], "THREE") == 0)return 3;
        else if(strcmp(keywords[out_class], "FOUR") == 0)return 4;
        else if(strcmp(keywords[out_class], "FIVE") == 0)return 5;
        else if(strcmp(keywords[out_class], "SIX") == 0)return 6;
        else if(strcmp(keywords[out_class], "SEVEN") == 0)return 7;
        else if(strcmp(keywords[out_class], "EIGHT") == 0)return 8;
        else if(strcmp(keywords[out_class], "NINE") == 0)return 9;
        else return 0;
}
```

Figure 17.23 Function Detected_Word

Figure 17.24 shows the other code added to the main program. This code is added just after the main program code shown in the Figure (the code added by the author is enclosed within the comments //******* DI). A **for** loop is used which calls function **Detected_Word** to find out what the spoken number is, and then flashes the red LED with this count.

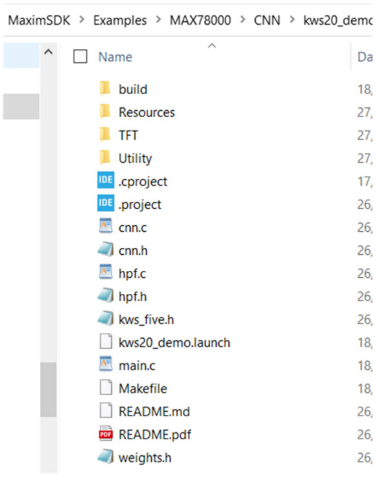
```
printf("Detected word: %s (%0.1f%%)", keywords[out_class],
       probability);
printf("\n---------------------------------------- \n");

//******** DI START
if(probability > 95)
{
      for(Max = 0; Max < Detected_Word(out_class); Max++)
      {
                LED_On(LED_RED);
                MXC_Delay(SEC(1));
                LED_Off(LED_RED);
                MXC_Delay(SEC(1));
      }
}
//******** DI END

Max = 0;
Min = 0
```

Figure 17.24 Added program code

If you have a TFT display you should connect it to the MAX78000FTHR development board. You may also connect a TTL-USB module and start a terminal emulation software on the PC. The program is ready when the green LED is ON. Speak a number and you should see the red LED flashing the required amount of times.

## 17.9 ● Operation of the kws20_demo program

Figure 17.25 shows the files and folders inside the kws20_demo folder.


MaximSDK > Examples > MAX78000 > CNN > kws20_demc

| Name | Da |
|---|---|
| build | 18, |
| Resources | 27, |
| TFT | 27, |
| Utility | 27, |
| .cproject | 17, |
| .project | 26, |
| cnn.c | 26, |
| cnn.h | 26, |
| hpf.c | 26, |
| hpf.h | 26, |
| kws_five.h | 26, |
| kws20_demo.launch | 18, |
| main.c | 18, |
| Makefile | 18, |
| README.md | 26, |
| README.pdf | 26, |
| weights.h | 26, |

Figure 17.25 kws20_demo folder

The flow of the demo software is shown in Figure 17.26 (see **Maxim Application Note: 7359 by Niktash & Loginov**). The onboard I2S microphone samples an 18-bit, 16kHz audio signal, and streams it to the MAX78000. A simple high-pass filter is used to remove the DC offset of the microphone and store the samples in a circular buffer. The signal level is averaged over 128-sample windows and compared to an adjustable threshold to find the beginning of a word. The level below this threshold is categorised as the silence before the word in an utterance. The beginning of a word is detected once the signal level passes the threshold. 16kHz, 8-bit samples (one second) are needed to start an inference on the CNN accelerator. The signal level at the end of a spoken word is monitored. The inference can start if the average level goes and stays below an adjustable threshold for several back-to-back 128-sample windows or if the 16k samples are already collected. The inference on the CNN accelerator for this network takes about 2.5ms. The inference result and confidence level are shown on the TFT display and serial port.

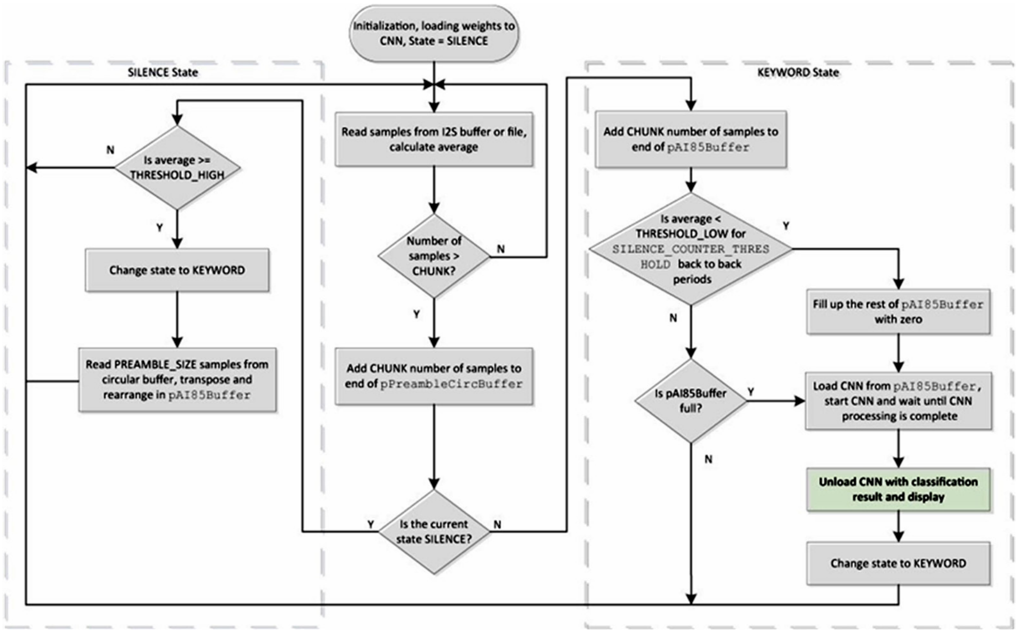

Initialization, loading weights to CNN, State = SILENCE
SILENCE State
KEYWORD State
Is average >= THRESHOLD_HIGH
N
Y
Change state to KEYWORD
Read PREAMBLE_SIZE samples from circular buffer, transpose and rearrange in pAI85Buffer
Read samples from I2S buffer or file, calculate average
Number of samples > CHUNK?
Add CHUNK number of samples to end of pPreambleCircBuffer
Is the current state SILENCE?
Add CHUNK number of samples to end of pAI85Buffer
Is average < THRESHOLD_LOW for SILENCE_COUNTER_THRESHOLD back to back periods
Fill up the rest of pAI85Buffer with zero
Is pAI85Buffer full?
Load CNN from pAI85Buffer, start CNN and wait until CNN processing is complete
Unload CNN with classification result and display
Change state to KEYWORD

Figure 17.26 Flow of the demo software

The 20 words to be recognised are stored in character array **keywords**:

```
const char keywords[NUM_OUTPUTS][10] =
{
"UP", "DOWN", "LEFT", "RIGHT", "STOP", "GO", "YES", "NO", "ON", "OFF", "ONE",
"TWO", "THREE", "FOUR", "FIVE", "SIX", "SEVEN", "EIGHT", "NINE", "ZERO",
"Unknown"
};
```

## 17.10 • Modelling, Training, and Synthesis

The development of an AI software project is done in 5 stages:

- Modelling (e.g. using PyTorch or TensorFlow-Keras)
- Training (e.g. using ai8x-training)
- Synthesis (e.g. using ai8x-synthesis)
- C Code generation for MAX78000 (e.g. using ai8x-synthesis)
- Embedded design (e.g. using Eclipse MaximSDK)

### Modelling

**PyTorch** or **TensorFlow-Keras** can be used to develop a model for the MAX78000 (see Figure 17.27). The details of PyTorch or TensorFlow are beyond the scope of this book, but brief details of the modelling process will be discussed in this section. A series of subclasses are used to represent the hardware. Pooling and activations are fused to 1D or 2D convolution layers. The model is trained with floating-point weights and training data. The result of quantisation can be evaluated over the evaluation dataset to check the accuracy degradation due to weight quantisation.

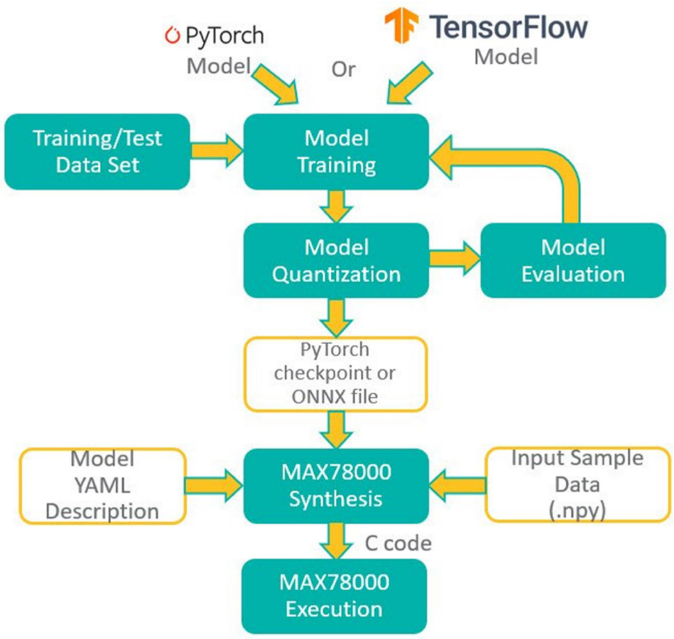


Figure 17.27 AI development flow

In this section, we will consider the kws20_demo. Figure 17.28 shows the PyTorch model of the project (see file: **ai8x/-training/models/ai85net-kws20.py** in link: https://github.com/MaximIntegratedAI/ai8x-training/tree/master/models. The model consists of two back-to-back CNNs: 1D (Conv1D) and 2D (Conv2D) convolutional networks. The Conv1D CNN includes four layers and extracts speech features. The Conv2D CNN is comprised of five layers, followed by a fully connected layer to classify the utterances. The model is trained with an augmented dataset for 20 keywords.

```
############################################################################################
########
#
# Copyright (C) Maxim Integrated Products, Inc. All Rights Reserved.
#
# Maxim Integrated Products, Inc. Default Copyright Notice:
# https://www.maximintegrated.com/en/aboutus/legal/copyrights.html
#
############################################################################################
########
```

```
"""
Keyword spotting network for AI85/AI86
"""
import torch.nn as nn

import ai8x


class AI85KWS20Net(nn.Module):
    """
    Compound KWS20 Audio net, starting with Conv1Ds with kernel_size=1
    and then switching to Conv2Ds
    """

    # num_classes = n keywords + 1 unknown
    def __init__(
            self,
            num_classes=21,
            num_channels=128,
            dimensions=(128, 1),  # pylint: disable=unused-argument
            fc_inputs=7,
            bias=False,
            **kwargs
    ):
        super().__init__()

        self.voice_conv1 = ai8x.FusedConv1dReLU(num_channels, 100, 1, stride=1, padding=0,
                                                bias=bias, **kwargs)

        self.voice_conv2 = ai8x.FusedConv1dReLU(100, 100, 1, stride=1, padding=0,
                                                bias=bias, **kwargs)

        self.voice_conv3 = ai8x.FusedConv1dReLU(100, 50, 1, stride=1, padding=0,
                                                bias=bias, **kwargs)

        self.voice_conv4 = ai8x.FusedConv1dReLU(50, 16, 1, stride=1, padding=0,
                                                bias=bias, **kwargs)

        self.kws_conv1 = ai8x.FusedConv2dReLU(16, 32, 3, stride=1, padding=1,
                                              bias=bias, **kwargs)

        self.kws_conv2 = ai8x.FusedConv2dReLU(32, 64, 3, stride=1, padding=1,
                                              bias=bias, **kwargs)

        self.kws_conv3 = ai8x.FusedConv2dReLU(64, 64, 3, stride=1, padding=1,
                                              bias=bias, **kwargs)
        self.kws_conv4 = ai8x.FusedConv2dReLU(64, 30, 3, stride=1, padding=1,
                                              bias=bias, **kwargs)

        self.kws_conv5 = ai8x.FusedConv2dReLU(30, fc_inputs, 3, stride=1, padding=1,
                                              bias=bias, **kwargs)

        self.fc = ai8x.Linear(fc_inputs * 128, num_classes, bias=bias, wide=True, **kwargs)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')

    def forward(self, x):  # pylint: disable=arguments-differ
        """Forward prop"""
        # Run CNN
        x = self.voice_conv1(x)
        x = self.voice_conv2(x)
        x = self.voice_conv3(x)
        x = self.voice_conv4(x)
        x = x.view(x.shape[0], x.shape[1], 16, -1)
        x = self.kws_conv1(x)
        x = self.kws_conv2(x)
        x = self.kws_conv3(x)
        x = self.kws_conv4(x)
        x = self.kws_conv5(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x


def ai85kws20net(pretrained=False, **kwargs):
    """
    Constructs a AI85KWS20Net model.
    """
    assert not pretrained
    return AI85KWS20Net(**kwargs)


models = [
    {
        'name': 'ai85kws20net',
        'min_input': 1,
        'dim': 1,
    },
]
```

Figure 17.28 kws20_demo PyTorch model

### Training

The training process (**ai8x-training**) optimises the network weights to minimise the output error. Depending on the complexity of the network, the type of processor, and the graphics card used, the training process can take many hours.

### Synthesis

The MAX78000 synthesiser tool (**ai8xize**) accepts the PyTorch checkpoint or TensorFlow exported ONNX files as an input, as well as the model description in the YAML format. The MAX78000 synthesiser automatically generates C code, which can be compiled and executed on the MAX78000. The C code includes Application Programming Interface (API) calls to load the weights as well as the provided sample data to the hardware to execute an inference on the sample data and compare the classification outcome with the expected result as a pass/fail sanity test. This generated C code can be used as an example to create own applications.

## 17.11 • Software/hardware requirements and software installation for training and synthesis

To train neural networks for the MAX78000, it is strongly recommended that you use an NVIDIA CUDA capable GPU inside your computer. It is recommendable to have a Maxwell generation chip (NVIDIA GTX9 series) or NVIDIA Tesla K80 at the very least. When going beyond simple models, model training does not work well without CUDA hardware acceleration. Such GPUs are costly and you may prefer to use your CPU to do all the required calculations for the training process. This may however increase the required processing time by a factor of 10 or more.

The software supports Ubuntu Linux 18.04 LTS and 20.04 LTS. The server version is sufficient (see link: https://ubuntu.com/download/server). Alternatively, Ubuntu Linux can also be used inside the Windows Subsystem for Linux (WSL2) by following the guide: https://docs.nvidia.com/cuda/wsl-user-guide/. Please note however that WSL2 with CUDA is a pre-release, and unexpected behaviour might occur. The recommendation is to install the latest version of CUDA 11 on Ubuntu 20.04 LTS (see link: https://developer.nvidia.com/cuda-toolkit-archive).

### Software installation

The following software should be installed on your Ubuntu 20.04 system before you can train and synthesise a network (for full details, see the Maxim link:
https://github.com/MaximIntegratedAI/ai8x-training)

```
$ sudo apt-get install -y make build-essential libssl-dev zlib1g-dev \
  libbz2-dev libreadline-dev libsqlite3-dev wget curl llvm \
  libncurses5-dev libncursesw5-dev xz-utils tk-dev libffi-dev liblzma-dev \
  libsndfile-dev portaudio19-dev

$  curl  -L  https://github.com/pyenv/pyenv-installer/raw/master/bin/pyenv-
installer | bash
```

Then, add to either ~/.bash_profile, ~/.bashrc, or ~/.profile (as shown by the terminal output of the previous step):

```
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
```

For convenience, define a shell variable named AI_PROJECT_ROOT:

```
$ export AI_PROJECT_ROOT="$HOME/Documents/Source/AI"
```

Add this line to `~/.profile`.

Install the following packages:

```
$ mkdir -p $AI_PROJECT_ROOT
$ cd $AI_PROJECT_ROOT
$ curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -
$ echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee /etc/apt/
sources.list.d/yarn.list

$ curl -sL https://deb.nodesource.com/setup_13.x | sudo -E bash -
$ sudo apt-get update
$ sudo apt-get install -y nodejs yarn

$ git clone https://github.com/uber/manifold.git
$ cd manifold
$ yarn
# ignore warnings
$ cd examples/manifold
$ yarn
# ignore warnings
```

Change to the project root and run the following commands. Use your GitHub credentials when prompted.

```
$ cd $AI_PROJECT_ROOT
$ git clone https://github.com/MaximIntegratedAI/ai8x-training.git
$ git clone https://github.com/MaximIntegratedAI/ai8x-synthesis.git
```

To create the virtual environment and install basic wheels:

```
$ cd ai8x-training
```

Then continue with the following (make sure a local Python3 version is already installed, using: pyenv install 3.8.6):

```
$ git submodule update --init
$ pyenv local 3.8.6
$ python3 -m venv .
$ source bin/activate
(ai8x-training) $ pip3 install -U pip wheel setuptools
```

The next step differs depending on whether the system uses Linux with CUDA 11.x, or any other setup.

For CUDA 11.x on Linux:

```
(ai8x-training) $ pip3 install -r requirements-cu111.txt
```

For all other systems, including CUDA 10.2 on Linux:

```
(ai8x-training) $ pip3 install -r requirements.txt
```

You are advised to check the link: https://github.com/MaximIntegratedAI/ai8x-training for updates.

We are now ready to train and synthesise a CNN with an available model. An example project is given below.

## 17.12 ● Project 2 – Training for a new keyword

**Description**: In this project, we will introduce a new keyword (**HAPPY**), and when this keyword is detected, the onboard red LED will flash 5 times. We will be using the kws20_ demo skeleton programs in this project.

**Aim**: This project aims to show a new keyword can be trained and synthesised for the MAX78000.

The kws20_demo application has been trained to recognise the following 20 words out of a total set of 35 words:

**"up", "down", "left", "right", "stop", "go", "yes", "no", "on", "off", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "zero"**

The complete set of 35 words is taken from version 2 of the speech command dataset created by Google (see: *Pete Warden, "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition", Apr. 2018*). The dataset consists of over 100k of utterances of 35 different words stored as one-second wave format files sampled at 16kHz:

**"backward", "bed", "bird", "cat", "dog", "down", "eight", "five", "follow", "forward", "four", "go", "happy", "house", "learn", "left", "marvin", "nine", "no", "off", "on", "one", "right", "seven", "sheila", "six", "stop", "three", "tree", "two", "up", "visual", "wow", "yes", "zero""**

For this application, we will still train 20 words from the same set but replace the word go with the word happy. The 20 words that we will be training are therefore as follows:

**"up", "down", "left", "right", "stop", "happy", "yes", "no", "on", "off", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "zero"**

In our application, the onboard red LED will flash 5 times when the word happy is detected.

**1. The Training (see link: https://github.com/MaximIntegratedAI/ai8x-training)**

The computer used by the author for training had the following specifications:

- Operating system: Windows 10, version: 10.0.18363 Build 18363
- System model: HP ENVY x360 Convertible 15-cn0xxx
- System type: x64-based PC
- Processor: Intel Core i7-8550U CPU @ 1.80GHZ
- Memory: 16GB
- Graphics card: NVIDIA GeForce MX150
- Hard disk: 512GB SSD

VMWare was used to create a workstation with the Ubuntu 20.04 operating system. The workstation was configured as follows:

- 100GB SSD drive
- 12GB memory
- 1 processor, 8 cores

The steps to start the training process are as follows:

- Install the required software on your Ubuntu machine
- Click **Files** on your Desktop to navigate to folder **ai8x-training** (see Figure 17.29)

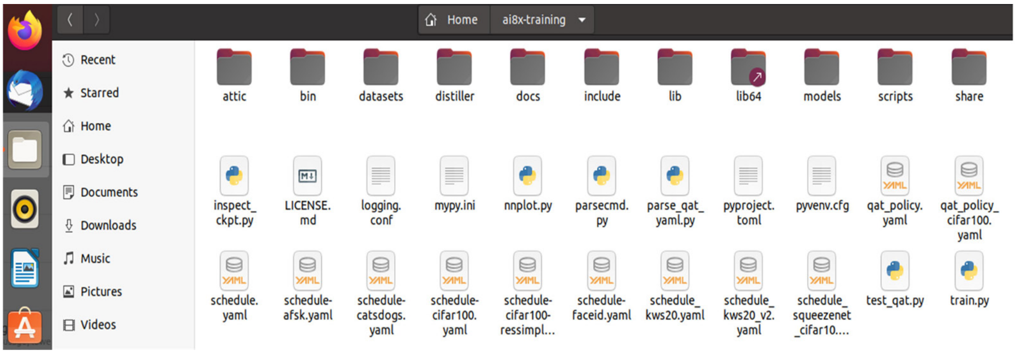
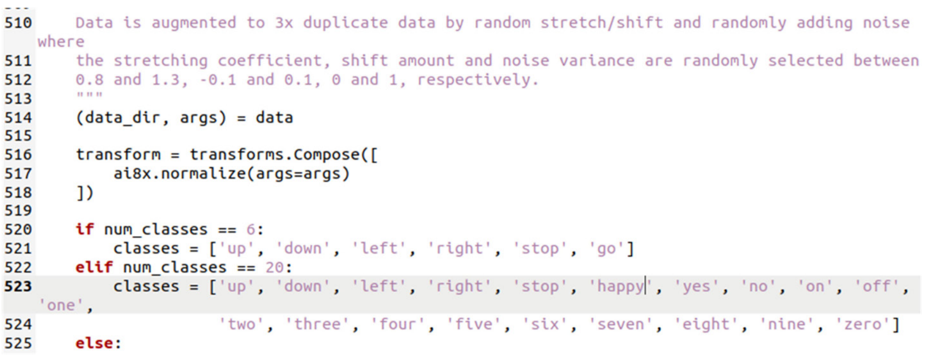


Figure 17.29 Navigate to folder ai8x-training

- Inside the folder, edit the file: **./datasets/kws20.py** and replace the word **go** with **happy** (Figure 17.30). Save and exit the file



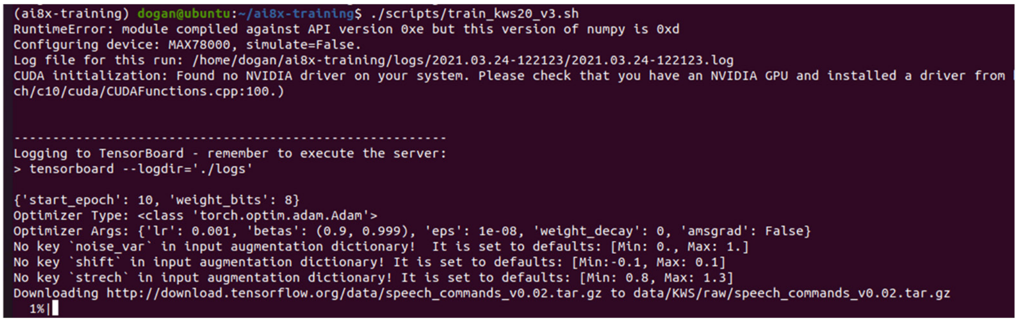
```
    Data is augmented to 3x duplicate data by random stretch/shift and randomly adding noise
where
    the stretching coefficient, shift amount and noise variance are randomly selected between
    0.8 and 1.3, -0.1 and 0.1, 0 and 1, respectively.
    """
    (data_dir, args) = data

    transform = transforms.Compose([
        ai8x.normalize(args=args)
    ])

    if num_classes == 6:
        classes = ['up', 'down', 'left', 'right', 'stop', 'go']
    elif num_classes == 20:
        classes = ['up', 'down', 'left', 'right', 'stop', 'happy', 'yes', 'no', 'on', 'off',
'one',
                   'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'zero']
    else:
```

Figure 17.30 Edit file kws20.py in folder datasets

- Open the terminal mode and navigate to the **ai8x-training** folder. Start the training by entering the following command (see Figure 17.31):

```
./scripts/train_kws20_v3.sh
```

Optionally, the above command can be terminated using the keyword **--cpu** for only CPU based training without a GPU, or the keyword **--gpu 0** for GPU based (CUDA) training.



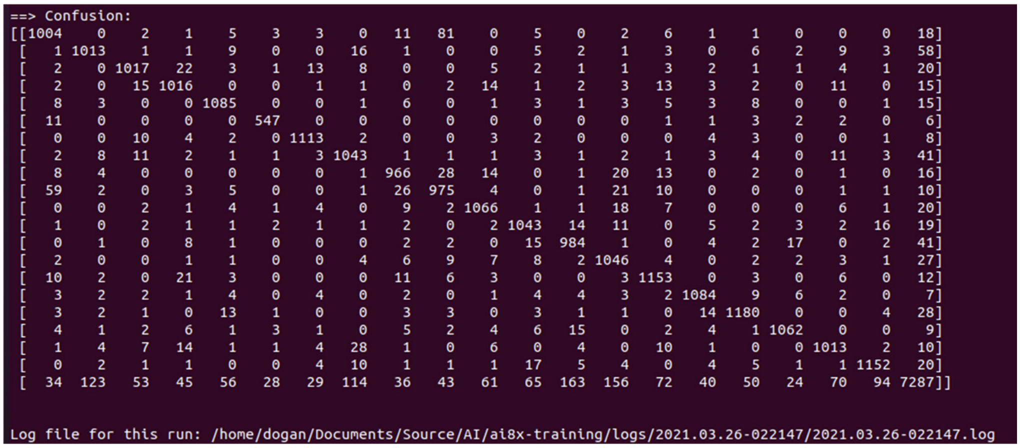
```
(ai8x-training) dogan@ubuntu:~/ai8x-training$ ./scripts/train_kws20_v3.sh
RuntimeError: module compiled against API version 0xe but this version of numpy is 0xd
Configuring device: MAX78000, simulate=False.
Log file for this run: /home/dogan/ai8x-training/logs/2021.03.24-122123/2021.03.24-122123.log
CUDA initialization: Found no NVIDIA driver on your system. Please check that you have an NVIDIA GPU and installed a driver from
ch/c10/cuda/CUDAFunctions.cpp:100.)


-------------------------------------------------------
Logging to TensorBoard - remember to execute the server:
> tensorboard --logdir='./logs'

{'start_epoch': 10, 'weight_bits': 8}
Optimizer Type: <class 'torch.optim.adam.Adam'>
Optimizer Args: {'lr': 0.001, 'betas': (0.9, 0.999), 'eps': 1e-08, 'weight_decay': 0, 'amsgrad': False}
No key `noise_var` in input augmentation dictionary!  It is set to defaults: [Min: 0., Max: 1.]
No key `shift` in input augmentation dictionary! It is set to defaults: [Min:-0.1, Max: 0.1]
No key `strech` in input augmentation dictionary! It is set to defaults: [Min: 0.8, Max: 1.3]
Downloading http://download.tensorflow.org/data/speech_commands_v0.02.tar.gz to data/KWS/raw/speech_commands_v0.02.tar.gz
  1%|
```

Figure 17.31 Starting the training process

The contents of the above shell command is as follows:

```
/train.py --epochs 200 --optimizer Adam --lr 0.001 --deterministic
--compress schedule_kws20.yaml --model ai85kws20netv3 --dataset KWS_20
--confusion --device MAX78000 "$@"
```

Depending on your hardware, the training process can take many hours or even days. It took about 18 hours on the author's computer to complete the training. Figure 17.32 shows the last page of the training process.



```
==> Confusion:
[[1004    0    2    1    5    3    3    0   11   81    0    5    0    2    6    1    1    0    0    0   18]
 [   1 1013    1    1    9    0    0   16    1    0    0    5    2    1    3    0    6    2    9    3   58]
 [   2    0 1017   22    3    1   13    8    0    0    5    2    1    1    3    2    1    1    4    1   20]
 [   2    0   15 1016    0    0    1    1    0    2   14    1    2    3   13    3    2    0   11    0   15]
 [   8    3    0    0 1085    0    0    1    6    0    1    3    1    3    5    3    8    0    0    1   15]
 [  11    0    0    0    0  547    0    0    0    0    0    0    0    0    1    1    3    2    2    0    6]
 [   0    0   10    4    2    0 1113    2    0    0    3    2    0    0    0    4    3    0    0    1    8]
 [   2    8   11    2    1    1    3 1043    1    1    1    3    1    2    1    3    4    0   11    3   41]
 [   8    4    0    0    0    0    0    1  966   28   14    0    1   20   13    0    2    0    1    0   16]
 [  59    2    0    3    5    0    0    1   26  975    4    0    1   21   10    0    0    0    1    1   10]
 [   0    0    2    1    4    1    4    0    9    2 1066    1    1   18    7    0    0    0    6    1   20]
 [   1    0    2    1    1    2    1    1    2    0    2 1043   14   11    0    5    2    3    2   16   19]
 [   0    1    0    8    1    0    0    0    2    2    0   15  984    1    0    4    2   17    0    2   41]
 [   2    0    0    1    1    0    0    4    6    9    7    8    2 1046    4    0    2    2    3    1   27]
 [  10    2    0   21    3    0    0    0   11    6    3    0    0    3 1153    0    3    0    6    0   12]
 [   3    2    2    1    4    0    4    0    2    0    1    4    4    3    2 1084    9    6    2    0    7]
 [   3    2    1    0   13    1    0    0    3    3    0    3    1    1    0   14 1180    0    0    4   28]
 [   4    1    2    6    1    3    1    0    5    2    4    6   15    0    2    4    1 1062    0    0    9]
 [   1    4    7   14    1    1    4   28    1    0    6    0    4    0   10    1    0    0 1013    2   10]
 [   0    2    1    1    0    0    4   10    1    1    1   17    5    4    0    4    5    1    1 1152   20]
 [  34  123   53   45   56   28   29  114   36   43   61   65  163  156   72   40   50   24   70   94 7287]]


Log file for this run: /home/dogan/Documents/Source/AI/ai8x-training/logs/2021.03.26-022147/2021.03.26-022147.log
```

Figure 17.32 Last page of the training

When the training ends, you will find in **./logs/** a new time and date encoded folder that contains your training results. In this example project, the following folder is created in folder **logs** after the training (notice the path **latest_log_dir** points to the latest log folder):

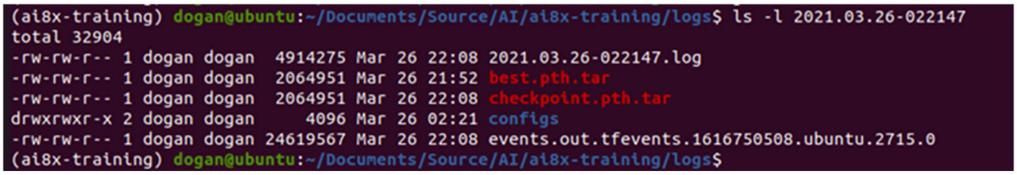
**2021.03.26-022147**

After the training, the log folder contained the following files (see Figure 17.33):

2021.03.26-022147.log
best.pth.tar
checkpoint.pth.tar
configs (folder)
events.out.tfevents.1616750508.ubuntu.2715.0

The **configs** folder contains the following file:

schedule-kws20.yaml



```
(ai8x-training) dogan@ubuntu:~/Documents/Source/AI/ai8x-training/logs$ ls -l 2021.03.26-022147
total 32904
-rw-rw-r-- 1 dogan dogan  4914275 Mar 26 22:08 2021.03.26-022147.log
-rw-rw-r-- 1 dogan dogan  2064951 Mar 26 21:52 best.pth.tar
-rw-rw-r-- 1 dogan dogan  2064951 Mar 26 22:08 checkpoint.pth.tar
drwxrwxr-x 2 dogan dogan     4096 Mar 26 02:21 configs
-rw-rw-r-- 1 dogan dogan 24619567 Mar 26 22:08 events.out.tfevents.1616750508.ubuntu.2715.0
(ai8x-training) dogan@ubuntu:~/Documents/Source/AI/ai8x-training/logs$
```

Figure 17.33 logs folder after the training

The files **best.pth.tar** (or **checkpoint.pth.tar**) and **schedule-kws20.yaml** will be inputs to the synthesiser. We will choose **best.pth.tar** which contains the best-quantised weights of the network. The file **schedule-kws20.yaml** contains the network configuration.

**2. The Synthesis (see link: https://github.com/MaximIntegratedAI/ai8x-synthesis)**

The synthesis is done after training and puts the files into a form that can be loaded to the MAX78000. Quantisation converts the floating-point values into an integer which is the format the CNN accelerator understands. The synthesis does not require CUDA.

The steps for the synthesis are as follows:

- Deactivate the ai8x-training environment if it is active:

```
(ai8x-training) $ deactivate
```

- Create a virtual environment and move to folder **ai8x-synthesis**:

```
cd $AI_PROJECT_ROOT
cd ai8x-synthesis
```

- Delete all contents of folder **trained** to make sure that any old data is not used
- Move to folder **ai8x-training**:

```
cd $AI_PROJECT_ROOT
cd ai8x-training
```

- Copy files from the **logs** folder in the **ai8x-training** folder into folder trained in **ai8x-synthesis**:

```
cp -R logs/2021.03.26-022147/*.* $AI_PROJECT_ROOT/ai8x-synthesis/trained/
```

- Move to folder ai8x-synthesis:

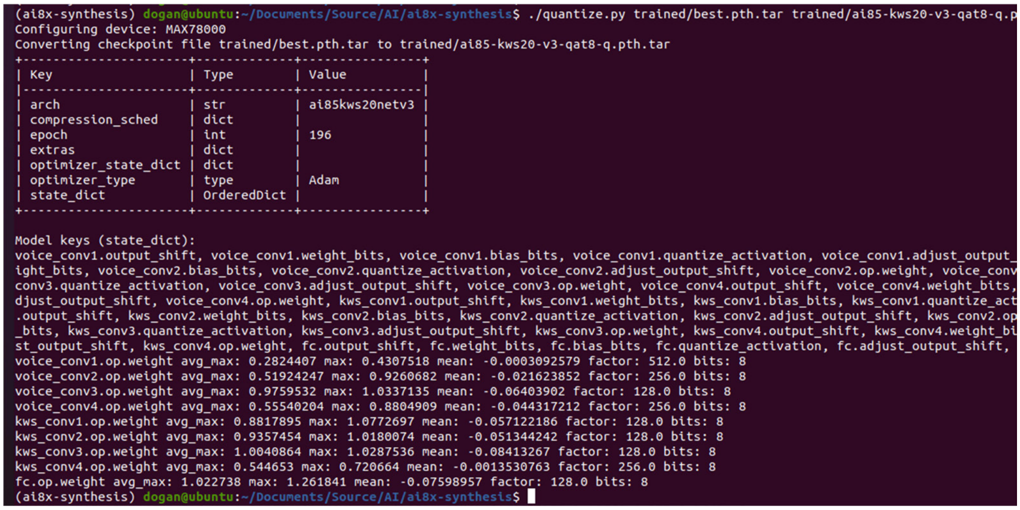
```
              cd $AI_PROJECT_ROOT
              cd ai8x-synthesis
```

- Install the following packages:

```
       $ git submodule update --init
       $ pyenv local 3.8.6
       $ python3 -m venv .
       $ source bin/activate
       (ai8x-synthesis) $ pip3 install -U pip setuptools
       (ai8x-synthesis) $ pip3 install -r requirements.txt
```

- We can now do a quantisation of our copied trained model with the following command (see Figure 17.34). This command will generate the quantised file called **ai85-kws20-v3-qat8-q.pth.tar** inside the **trained** folder:

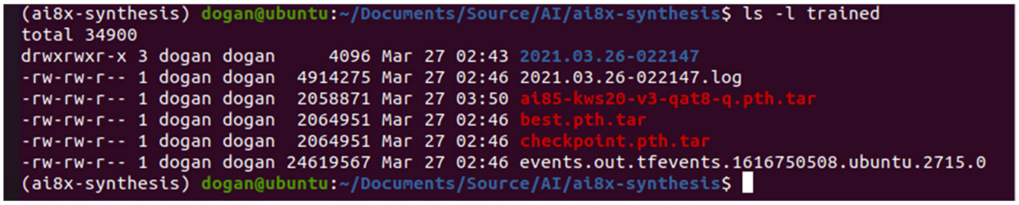
```
./quantize.py trained/best.pth.tar trained/ai85-kws20-v3-qat8-q.pth.tar
--device MAX78000 -v "$@"
```



```
(ai8x-synthesis) dogan@ubuntu:~/Documents/Source/AI/ai8x-synthesis$ ./quantize.py trained/best.pth.tar trained/ai85-kws20-v3-qat8-q.p
Configuring device: MAX78000
Converting checkpoint file trained/best.pth.tar to trained/ai85-kws20-v3-qat8-q.pth.tar
+----------------------+------------+---------------+
| Key                  | Type       | Value         |
|----------------------+------------+---------------|
| arch                 | str        | ai85kws20netv3 |
| compression_sched    | dict       |               |
| epoch                | int        | 196           |
| extras               | dict       |               |
| optimizer_state_dict | dict       |               |
| optimizer_type       | type       | Adam          |
| state_dict           | OrderedDict |              |
+----------------------+------------+---------------+

Model keys (state_dict):
voice_conv1.output_shift, voice_conv1.weight_bits, voice_conv1.bias_bits, voice_conv1.quantize_activation, voice_conv1.adjust_output_
ight_bits, voice_conv2.bias_bits, voice_conv2.quantize_activation, voice_conv2.adjust_output_shift, voice_conv2.op.weight, voice_conv
conv3.quantize_activation, voice_conv3.adjust_output_shift, voice_conv3.op.weight, voice_conv4.output_shift, voice_conv4.weight_bits,
djust_output_shift, voice_conv4.op.weight, kws_conv1.output_shift, kws_conv1.weight_bits, kws_conv1.bias_bits, kws_conv1.quantize_act
.output_shift, kws_conv2.weight_bits, kws_conv2.bias_bits, kws_conv2.quantize_activation, kws_conv2.adjust_output_shift, kws_conv2.op
_bits, kws_conv3.quantize_activation, kws_conv3.adjust_output_shift, kws_conv3.op.weight, kws_conv4.output_shift, kws_conv4.weight_bi
st_output_shift, kws_conv4.op.weight, fc.output_shift, fc.weight_bits, fc.bias_bits, fc.quantize_activation, fc.adjust_output_shift,
voice_conv1.op.weight avg_max: 0.2824407 max: 0.4307518 mean: -0.0003092579 factor: 512.0 bits: 8
voice_conv2.op.weight avg_max: 0.51924247 max: 0.9260682 mean: -0.021623852 factor: 256.0 bits: 8
voice_conv3.op.weight avg_max: 0.9759532 max: 1.0337135 mean: -0.06403902 factor: 128.0 bits: 8
voice_conv4.op.weight avg_max: 0.55540204 max: 0.8804909 mean: -0.044317212 factor: 256.0 bits: 8
kws_conv1.op.weight avg_max: 0.8817895 max: 1.0772697 mean: -0.057122186 factor: 128.0 bits: 8
kws_conv2.op.weight avg_max: 0.9357454 max: 1.0180074 mean: -0.051344242 factor: 128.0 bits: 8
kws_conv3.op.weight avg_max: 1.0040864 max: 1.0287536 mean: -0.08413267 factor: 128.0 bits: 8
kws_conv4.op.weight avg_max: 0.544653 max: 0.720664 mean: -0.0013530763 factor: 256.0 bits: 8
fc.op.weight avg_max: 1.022738 max: 1.261841 mean: -0.07598957 factor: 128.0 bits: 8
(ai8x-synthesis) dogan@ubuntu:~/Documents/Source/AI/ai8x-synthesis$
```

Figure 17.34 Quantise the trained model

Figure 17.35 shows the contents of the trained folder after the quantisation.

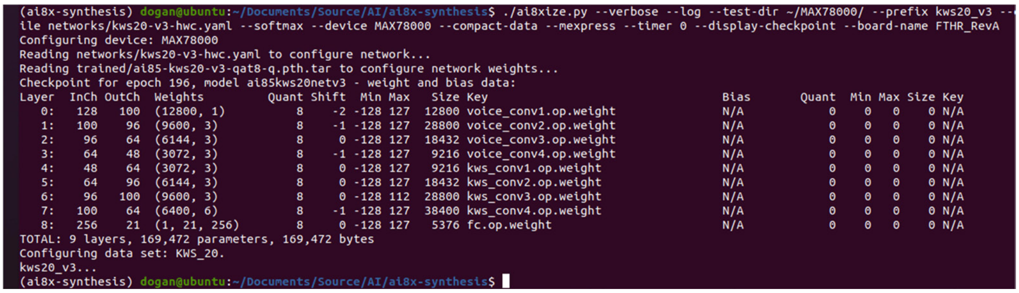


```
(ai8x-synthesis) dogan@ubuntu:~/Documents/Source/AI/ai8x-synthesis$ ls -l trained
total 34900
drwxrwxr-x 3 dogan dogan     4096 Mar 27 02:43 2021.03.26-022147
-rw-rw-r-- 1 dogan dogan  4914275 Mar 27 02:46 2021.03.26-022147.log
-rw-rw-r-- 1 dogan dogan  2058871 Mar 27 03:50 ai85-kws20-v3-qat8-q.pth.tar
-rw-rw-r-- 1 dogan dogan  2064951 Mar 27 02:46 best.pth.tar
-rw-rw-r-- 1 dogan dogan  2064951 Mar 27 02:46 checkpoint.pth.tar
-rw-rw-r-- 1 dogan dogan 24619567 Mar 27 02:46 events.out.tfevents.1616750508.ubuntu.2715.0
(ai8x-synthesis) dogan@ubuntu:~/Documents/Source/AI/ai8x-synthesis$
```

Figure 17.35 Folder trained after the quantisation

- After quantisation we can generate code and the final usable files for the MAX78000. Enter the following command in the **ai8x-synthesis** command prompt (see Figure 17.36):

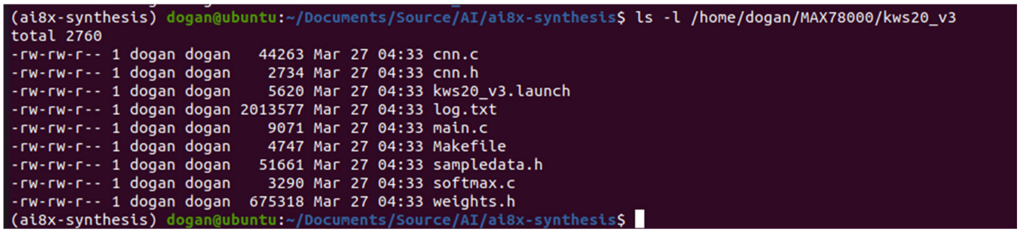
```
./ai8xize.py --verbose --log --test-dir ~/MAX78000/ --prefix kws20_v3
--checkpoint-file trained/ai85-kws20-v3-qat8-q.pth.tar --config-file
networks/kws20-v3-hwc.yaml --softmax --device MAX78000 --compact-data
--mexpress --timer 0 --display-checkpoint --board-name FTHR_RevA
```



```
(ai8x-synthesis) dogan@ubuntu:~/Documents/Source/AI/ai8x-synthesis$ ./ai8xize.py --verbose --log --test-dir ~/MAX78000/ --prefix kws20_v3 --
ile networks/kws20-v3-hwc.yaml --softmax --device MAX78000 --compact-data --mexpress --timer 0 --display-checkpoint --board-name FTHR_RevA
Configuring device: MAX78000
Reading networks/kws20-v3-hwc.yaml to configure network...
Reading trained/ai85-kws20-v3-qat8-q.pth.tar to configure network weights...
Checkpoint for epoch 196, model ai85kws20netv3 - weight and bias data:
Layer  InCh OutCh  Weights         Quant Shift  Min Max   Size Key                                       Bias       Quant  Min Max Size Key
   0:   128   100  (12800, 1)          8    -2 -128 127  12800 voice_conv1.op.weight                     N/A            0    0   0    0 N/A
   1:   100    96  (9600, 3)           8    -1 -128 127  28800 voice_conv2.op.weight                     N/A            0    0   0    0 N/A
   2:    96    64  (6144, 3)           8     0 -128 127  18432 voice_conv3.op.weight                     N/A            0    0   0    0 N/A
   3:    64    48  (3072, 3)           8    -1 -128 127   9216 voice_conv4.op.weight                     N/A            0    0   0    0 N/A
   4:    48    64  (3072, 3)           8     0 -128 127   9216 kws_conv1.op.weight                       N/A            0    0   0    0 N/A
   5:    64    96  (6144, 3)           8     0 -128 127  18432 kws_conv2.op.weight                       N/A            0    0   0    0 N/A
   6:    96   100  (9600, 3)           8     0 -128 112  28800 kws_conv3.op.weight                       N/A            0    0   0    0 N/A
   7:   100    64  (6400, 6)           8    -1 -128 127  38400 kws_conv4.op.weight                       N/A            0    0   0    0 N/A
   8:   256    21  (1, 21, 256)        8     0 -128 127   5376 fc.op.weight                              N/A            0    0   0    0 N/A
TOTAL: 9 layers, 169,472 parameters, 169,472 bytes
Configuring data set: KWS_20.
kws20_v3...
(ai8x-synthesis) dogan@ubuntu:~/Documents/Source/AI/ai8x-synthesis$
```

Figure 17.36 Generate final usable files

The above command will generate a folder called **MAX78000** inside the Home directory (**/home/dogan/** in the author's computer) that contains a small basic application that tests itself against predefined test data inside the code. Folder MAX78000 contains a folder called **kws20_v3** with the contents like **cnn.c, cnn.h, main.c, weights.h** etc. as shown in Figure 17.37. The example test code can be compiled and deployed to the MAX78000.



```
(ai8x-synthesis) dogan@ubuntu:~/Documents/Source/AI/ai8x-synthesis$ ls -l /home/dogan/MAX78000/kws20_v3
total 2760
-rw-rw-r-- 1 dogan dogan   44263 Mar 27 04:33 cnn.c
-rw-rw-r-- 1 dogan dogan    2734 Mar 27 04:33 cnn.h
-rw-rw-r-- 1 dogan dogan    5620 Mar 27 04:33 kws20_v3.launch
-rw-rw-r-- 1 dogan dogan 2013577 Mar 27 04:33 log.txt
-rw-rw-r-- 1 dogan dogan    9071 Mar 27 04:33 main.c
-rw-rw-r-- 1 dogan dogan    4747 Mar 27 04:33 Makefile
-rw-rw-r-- 1 dogan dogan   51661 Mar 27 04:33 sampledata.h
-rw-rw-r-- 1 dogan dogan    3290 Mar 27 04:33 softmax.c
-rw-rw-r-- 1 dogan dogan  675318 Mar 27 04:33 weights.h
(ai8x-synthesis) dogan@ubuntu:~/Documents/Source/AI/ai8x-synthesis$
```

Figure 17.37 Contents of folder MAX78000/kws20_v3

The synthesised model parameters and the created basic program main.c can be evaluated by compiling and running the created program. This is left as an exercise to the reader.

We are now ready to develop our embedded application and use the generated files **cnn.c**, **cnn.h**, **weights.h** etc.

## 3. Developing the embedded program

Now that we have trained and synthesised our network, the next stage is the development of the embedded program. In this section, this will be done using the Eclipse MaximSDK as per our other projects.

In this section, we will be using the **kws20_demo** project supplied by Maxim, and modify this program as required. Just to remind ourselves, we wish to flash the red LED 5 times when the word **HAPPY** is detected. Before modifying the original program, we need to copy the generated CNN files **cnn.c**, **cnn.h**, and **weights.h** to the following folder on our PC:

**MaximSDK/Examples/MAX78000/kws20_demo**

**(you may want to rename and save the original files before copying the new ones to the folder)**

Training and synthesising were done using Ubuntu Linux. The required files can be copied to our Windows-based system using the following steps:

Install the following packages on your Ubuntu system:

- `sudo apt install net-tools`
- `sudo apt install openssh-server`
- `sudo service ssh start`
- `sudo service ssh enable`

Enter the following command to see the status of SSH:

- `sudo service ssh status`

Enter the following command on your Linux system to get the IP address of your Linux system:

- ifconfig
- Start the file copy program **WinSCP** on your PC.
- Enter the IP address, username, and password for your Linux machine
- Set to the HOME directory on the Linux machine and drag and drop the required files from Linux to the **kws20_demo** folder on the PC
- Exit WinSCP

Start Eclipse, expand program **main.c**, and modify/add the following statements:

- Edit the **Makefile** and set it to **FTHR_RevA** as described earlier

- Enable the TFT display as described earlier if you have a TFT display attached to MAX78000 FTHR development board
- Connect the TX pin of the MAX78000FTHR development board to a TTL-USB module and connect the module to your PC. Start a terminal emulation program on your PC with the Baud rate set to 115200
- Change array **keywords** as follows:

```
const char keywords[NUM_OUTPUTS][10] = { "UP", "DOWN", "LEFT", "RIGHT",
"STOP", "HAPPY", "YES", "NO", "ON", "OFF", "ONE", "TWO", "THREE", "FOUR",
"FIVE", "SIX", "SEVEN", "EIGHT", "NINE", "ZERO", "Unknown"
                       };
```

- Insert the following function before the main program main.c (the points where the code is inserted are marked with comments //******* DI ):

```
int Detected_Word(int16_t out_class)
{
       if(strcmp(keywords[out_class], "HAPPY") == 0)
              return 1;
       else
              return 0;
}
```

- Insert the following statements to flash the Red LED 5 times when word HAPPY is detected:

```
                printf("Detected word: %s (%0.1f%%)", keywords[out_class],
                       probability);

                printf("\n---------------------------------------- \n");
//******** DI START
                if(probability > 95)
                {
                      if(Detected_Word(out_class) == 1)
                      {
                           for(Max = 0; Max < 5; Max++)
                           {
                              LED_On(LED_RED);
                              MXC_Delay(SEC(1));
                              LED_Off(LED_RED);
                              MXC_Delay(SEC(1));
                           }
                      }
                }
//******** DI END
                Max = 0;
                Min = 0;
              //--------------------------------------------------------------
```

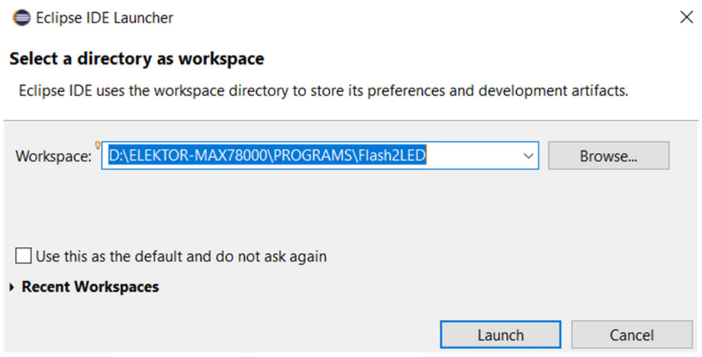
## Testing the program

- Compile the program, making sure there are no errors, and run in debug mode as described earlier.
- You should see a message asking you to press the button PB1 (SW1) to start detecting spoken words. Press the button
- Speak the word **HAPPY**. The red LED should flash 5 times

# Appendix A • Running the Programs in this Book

All programs in this book are available online. Each program and its associated files are in separate folders. This appendix shows the steps on how the programs given in this book can be loaded, compiled, and run.

- Make sure the MAX78000FTHR development board is connected to your PC
- You may have to start a terminal emulation program (e.g. Putty, Terraterm. HyperTerm, etc) on your PC for some of the programs where data is sent to the PC over the serial link
- Start Eclipse MaximSDK
- Select a Workspace by specifying the folder name of the program. For example, to run the program **Flash2LED**, select folder **Flash2LED** as the Workspace name by browsing the programs folder (see Figure A.1), and click **Launch**



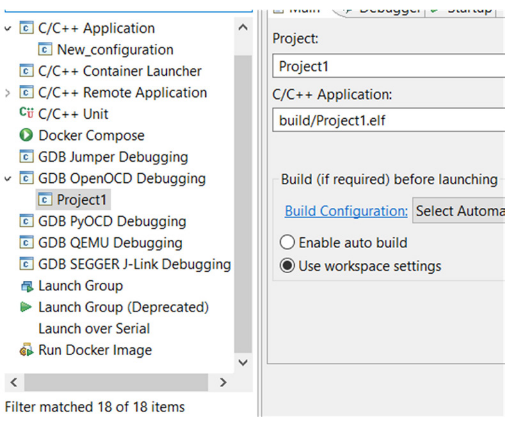
Eclipse IDE uses the workspace directory to store its preferences and development artifacts.

Workspace: D:\ELEKTOR-MAX78000\PROGRAMS\Flash2LED

Use this as the default and do not ask again

Figure A.1 Select a Workspace by the name of the program

- Click **Project -> Build All** to build the project
- Click **Run -> Run Configurations**
- Click **GDB OpenOCD Debugging** and select **Project 1** (Figure A.2)



Project: Project1

C/C++ Application: build/Project1.elf

Figure A.2 Select Project 1

- Click **Run** to run the program

**Note**: As described in chapter 3, you can also load all the programs into your workspace and select the ones you wish to compile and run.

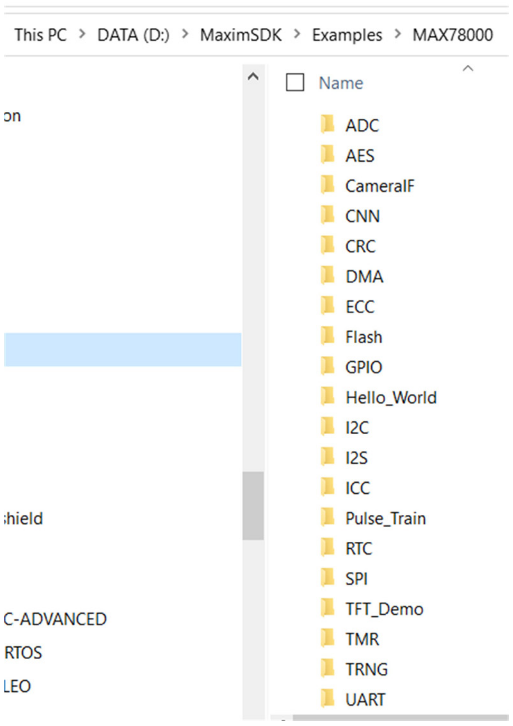
# Appendix B • References to Useful Files and Web Links

## B.1 • Useful files

There are many MAX78000 related files distributed with the MaximSDK that can be very useful during program development. The folders containing these files are given below:

### 1. Folder: MaximSDK\Examples\MAX78000

This folder contains many example MAX78000 programs (Figure B.1)

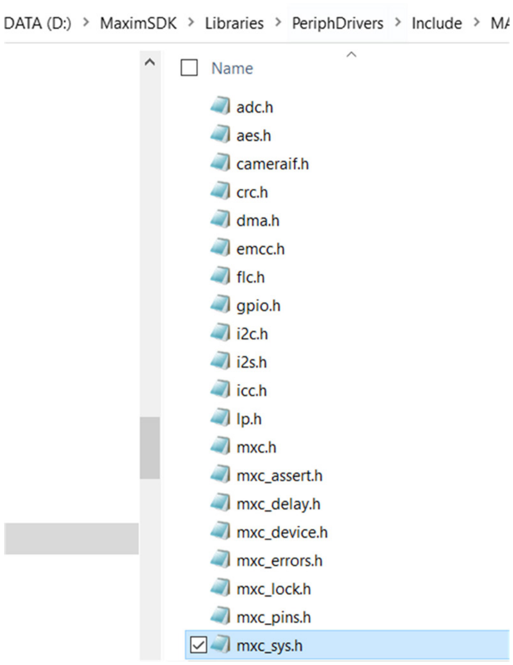


ADC
AES
CameraIF
CNN
CRC
DMA
ECC
Flash
GPIO
Hello_World
I2C
I2S
ICC
Pulse_Train
RTC
SPI
TFT_Demo
TMR
TRNG
UART

Figure B.1 Example programs

### 2. Folder: MaximSDK\Libraries\PeriphDrivers\Include\MAX78000

This folder contains the header files (.h) for the example programs. One can see the available structures and functions for a given utility by examining these header files (Figure B.2)



DATA (D:) > MaximSDK > Libraries > PeriphDrivers > Include > MA

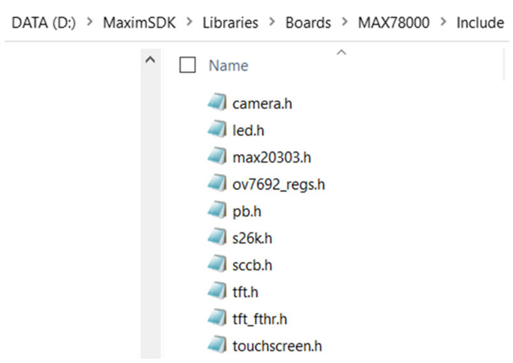
Name

adc.h
aes.h
cameraif.h
crc.h
dma.h
emcc.h
flc.h
gpio.h
i2c.h
i2s.h
icc.h
lp.h
mxc.h
mxc_assert.h
mxc_delay.h
mxc_device.h
mxc_errors.h
mxc_lock.h
mxc_pins.h
mxc_sys.h

Figure B.2 Header files

### 3. Folder: MaximSDK\Libraries\Boards\MAX78000\Include

This folder contains some additional header files (.h) for the example programs (Figure B.3)



DATA (D:) > MaximSDK > Libraries > Boards > MAX78000 > Include

Name

camera.h
led.h
max20303.h
ov7692_regs.h
pb.h
s26k.h
sccb.h
tft.h
tft_fthr.h
touchscreen.h

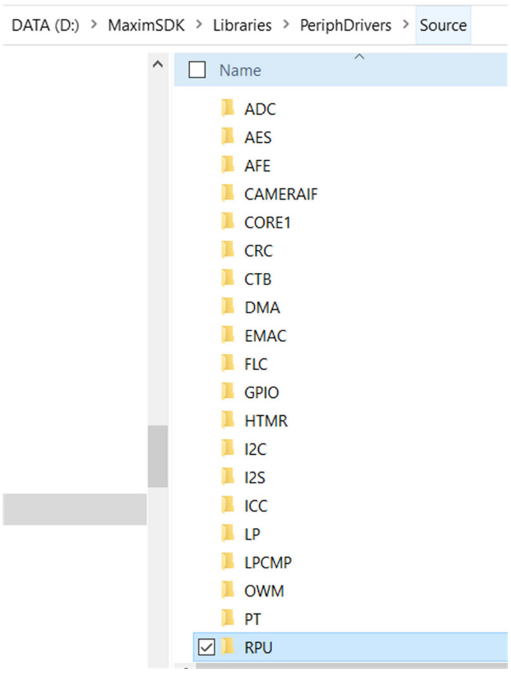
Figure B.3 Additional header files

### 4. Folder: MaximSDK\Libraries\PeriphDrivers\Documentation

This folder contains useful documents on various functions

## 5. Folder: MaximSDK\Libraries\PeriphDrivers\Source

This folder contains the source files (.c) of the functions (Figure B.4)



DATA (D:) > MaximSDK > Libraries > PeriphDrivers > Source

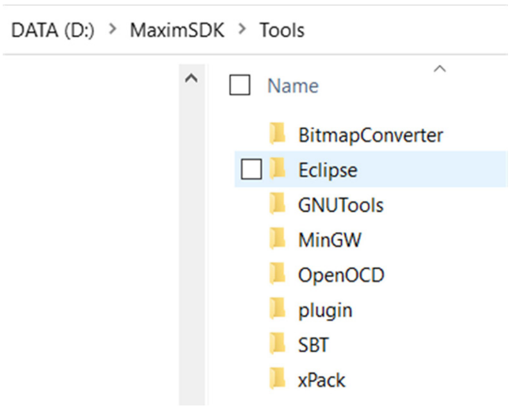
Name

ADC
AES
AFE
CAMERAIF
CORE1
CRC
CTB
DMA
EMAC
FLC
GPIO
HTMR
I2C
I2S
ICC
LP
LPCMP
OWM
PT
RPU

Figure B.4 Source files

## 6. This folder contains some useful software tools (Figure B.5)

This folder contains some useful software tools (Figure B.5)



DATA (D:) > MaximSDK > Tools

Name

BitmapConverter
Eclipse
GNUTools
MinGW
OpenOCD
plugin
SBT
xPack

Figure B.5 Useful tools

**7. Folder: MaximSDK\Libraries\Boards\MAX78000\Include**

This folder contains TFT, camera, and microphone header files.

## B.2 ● Useful Weblinks

You will find below data sheets and application notes on the MAX78000 microcontroller and the MAX78000FTHR development board.

https://datasheets.maximintegrated.com/en/ds/MAX78000.pdf

https://www.maximintegrated.com/en/design/technical-documents/app-notes/7/7417.html

https://www.maximintegrated.com/en/design/videos.html/vd_1_rtp4xipe#popupmodal

https://datasheets.maximintegrated.com/en/ds/MAX78000FTHR.pdf

https://www.maximintegrated.com/en/design/software-description.html/swpart=SFW0010820A

https://github.com/MaximIntegratedAI/MaximAI_Documentation/blob/master/MAX78000_Feather/README.md

https://www.maximintegrated.com/en/design/technical-documents/app-notes/7/7359.html

https://www.maximintegrated.com/en/products/microcontrollers/MAX78000.html

https://www.maximintegrated.com/en/design/videos.html

https://www.maximintegrated.com/en/design/technical-documents/app-notes/7/7364.html

https://www.elektormagazine.com/articles/ai-edge-get-started-maxim-max78000fthr

https://www.electronicproducts.com/maxims-first-ai-chip-a-game-changer-for-battery-powered-devices/#

https://www.nasdaq.com/press-release/maxim-integrateds-neural-network-accelerator-chip-enables-iot-artificial-intelligence

## • Index

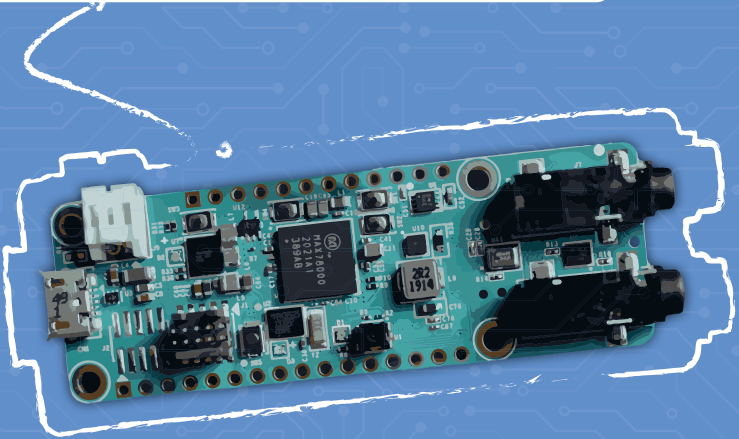
The MAX78000FTHR from Maxim Integrated is a small development board based on the MAX78000 MCU. The main usage of this board is in artificial intelligence applications (AI) which generally require large amounts of processing power and memory. It marries an Arm Cortex-M4 processor with a floating-point unit (FPU), convolutional neural network (CNN) accelerator, and RISC-V core into a single device. It is designed for ultra-low power consumption, making it ideal for many portable AI-based applications.

This book is project-based and aims to teach the basic features of the MAX78000FTHR. It demonstrates how it can be used in various classical and AI-based projects. Each project is described in detail and complete program listings are provided. Readers should be able to use the projects as they are, or modify them to suit their applications. This book covers the following features of the MAX78000FTHR microcontroller development board:

- Onboard LEDs and buttons
- External LEDs and buttons
- Using analog-to-digital converters
- $I^2C$ projects
- SPI projects
- UART projects
- External interrupts and timer interrupts
- Using the onboard microphone
- Using the onboard camera
- Convolutional Neural Network (CNN)
- Artificial Intelligence projects

**About the Author**

Prof Dr Dogan Ibrahim has a BSc degree in electronic engineering, an MSc degree in automatic control engineering, and a PhD degree in digital signal processing. Dogan has worked in many industrial organizations before he returned to academic life. Prof Ibrahim is the author of over 60 technical books and over 200 technical articles on microcontrollers, microprocessors, and related fields. He is a Chartered electrical engineer and a Fellow of the Institution of Engineering Technology.



ISBN 978-3-89576-447-9

**Elektor International Media BV**
www.elektor.com