Digital Signal Processing using Arm Cortex-M based Microcontrollers

Digital Signal Processing using Arm Cortex-M based Microcontrollers

Theory and Practice

arm Education Media

arm Education Media

Arm Education Media is an imprint of Arm Ltd. 110 Fullbourn Road, Cambridge, CBI 9NJ, UK

Notices

Knowledge and best practice in this field are constantly changing. As new research and experiences broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of product liability, negligence, or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

ISBN: 978-1-911531-16-6

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

Library of Congress Cataloging-in-Publication Data
A Catalog record for this book is available from the Library of Congress

Exercise solutions are available online via the Arm Connected Community: visit https://community.arm.com and search for the book title.
You may need to create an account first.

For information on all Arm Education Media publications, visit www.arm.com/education

To report errors or send feedback, please email edumedia@arm.com.

To our families

Contents

Preface

We live in a digital world surrounded by signals. Therefore, understanding and processing these signals is one of the most important skills a graduating or postgraduate engineer can possess. Arm Cortex-M based microcontrollers provide a low-cost and powerful platform for this purpose. This book aims to introduce the basics of digital signal processing on these microcontrollers and the theory behind it and includes a set of labs that handle the practical side of digital signal processing. We believe that this book will help undergraduate and postgraduate engineering students to bridge the gap between the theoretical and practical aspects of digital signal processing, so they can grasp its concepts completely. As a result, they will be ready to apply digital signal processing methods to solve real-life problems in an effective manner. This will be a valuable asset in today's competitive job market.

About This Book

This book is not purely about the theory of digital signal processing, nor is it solely about the practical aspects of digital signal processing. We believe that it is not possible to implement a DSP algorithm without knowing its theoretical merits and limitations. Moreover, theory is not sufficient alone to implement a DSP algorithm. Therefore, we cover DSP theory in this book, and then we explore these theoretical concepts through practical applications and labs. The aim is to bridge the gap between theory and practice.

In this book, we assume that you have some basic knowledge of signal processing. Therefore, theory is only provided when necessary and we do not go into great detail. However, we do provide a further reading section at the end of each chapter, where we provide references that are relevant to the concepts covered in that chapter. These may help you to extend your knowledge further.

The topics covered in this book are as follows. Chapter 2 introduces the mathematical and practical basics that will be used throughout the book. Chapter 3 discusses the Z-transform used to analyze discrete-time signals and systems in the complex domain. It will become clearer that some analyses will be easier to perform in this domain. In relation to this, Chapter 4 introduces frequency domain analysis in discrete time. In this chapter, you will learn how to analyze a discrete-time signal and system in the frequency domain. This will also uncover a method for designing and implementing filters in the frequency domain. Chapter 5 covers analog–digital conversions. These concepts are vital in understanding the relationship between analog and digital signals. Chapter 6

introduces methods of processing analog signals in a digital system. This chapter also focuses on methods of representing an analog system in digital form. Chapter 7 covers ways of representing a discrete-time system in different structural forms. Chapter 8 covers filter design, which is an important topic in digital signal processing. Chapter 9 focuses on adaptive signal processing concepts. This is one of the strengths of digital systems because their parameters can be easily changed on the fly. Chapter 10 explores fixed-point number representation issues. In this chapter, we observe that these issues have a direct effect on hardware usage, computation load, and the obtained result. Chapter 11 covers real-time digital signal processing concepts, which are extremely important for real-life applications with timing constraints.

A Note about Online Resources

This book includes a number of online resources that are accessible to readers of both the print and ebook versions, including answers to the end-of-chapter exercises and lab tasks, as well as code and other materials useful for the lab tasks. To access these resources, please visit the Arm Connected Community, https://community.arm.com/, and search for the book title. You may need to create an account first.

Acknowledgments

We would like to thank Arm Education for encouraging us to develop this book.

Author Biographies

Cem Ünsalan
Marmara University

Dr. Cem Ünsalan has worked on signal and image processing for 18 years. After receiving a Ph.D. degree from The Ohio State University, USA in 2003, he began working at Yeditepe University, Turkey. He now works at Marmara University, Turkey. He has been teaching microprocessor and digital signal processing courses for 10 years. He has published 20 articles in refereed journals. He has published five international books and holds one patent.

M. Erkin Yücel
Yeditepe University

M. Erkin Yücel received his B.Sc. and M.Sc. degrees from Yeditepe University. He is pursuing a Ph.D. degree on embedded systems at the same university. He has guided microprocessor and digital signal processing laboratory sessions for three years. Currently, he is working in research and development in industry.

H. Deniz Gürhan
Yeditepe University

H. Deniz Gürhan received his B.Sc. degree from Yeditepe University. He is pursuing a Ph.D. degree on embedded systems at the same university. For six years, he has been guiding microprocessor and digital signal processing laboratory sessions. He has published one international book on microcontrollers.

Digital Signal Processing Basics

Contents

1.1 Introduction

The aim of this book is to introduce you to the concepts of digital signal processing (DSP). To do so, the first step is explaining what a signal is and how to process it using a system. This chapter briefly introduces these concepts and explains why digital signal processing is important in today's world.

1.2 Definition of a Signal and Its Types

We perceive the world around us through our sensory organs. When an event occurs, a physical effect is generated. A sensor in our body receives this effect, transforms it into a specific form, and sends the result to our brain. The brain processes these data and commands the body to act accordingly. An electronic system works in a similar manner. There are various sensors (such as microphones, cameras, and pressure sensors) that transform a physical quantity into electronic form. If a sensor output changes with time (or another variable), we call it a *signal*. More generally, we can define a signal as data that change in relation to a dependent variable. A signal can be processed by a *system* to either obtain information from it or to modify it.

1.2.1 Continuous-Time Signals

If an acquired signal is in analog form, then we call it a continuous-time signal. A system that processes a continuous-time signal is called a continuous-time system. Assume that we form an analog microphone circuitry to display audio signals on an ***oscilloscope*** screen. If we say the word "HELLO," we will see a continuous-time signal on the screen as shown in Figure 1.1. This signal can be further processed by a continuous-time system composed of analog devices.

> **oscilloscope**
> *A laboratory instrument commonly used to display and analyze the waveform of electronic signals.*

1.2.2 Discrete-Time Signals

Although processing a continuous-time signal with a continuous-time system may seem reasonable, this is not the case for most applications. An alternative method is to sample the signal discretely. This corresponds to a discrete-time signal. Let us take a section of a continuous-time signal as in Figure 1.2(a). Taking 44,100 samples per second from it, we

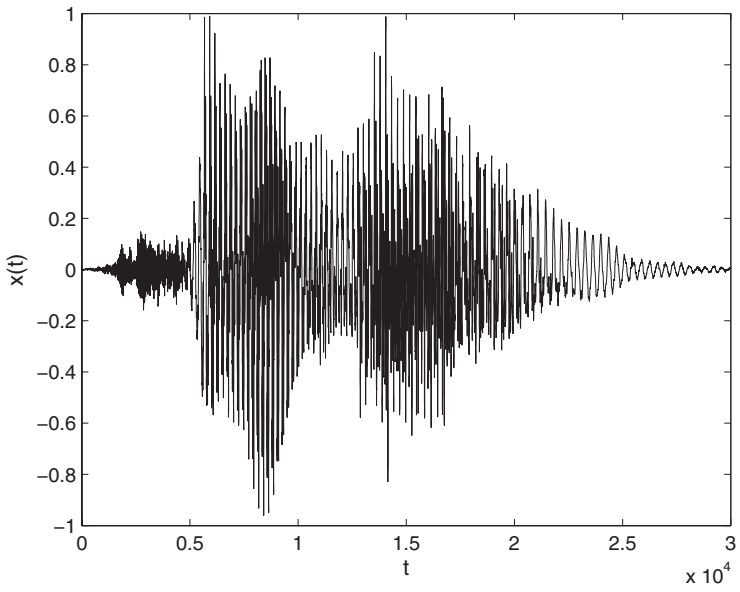

Figure 1.1 The word "HELLO" displayed as a continuous-time signal.

can obtain the corresponding discrete-time signal shown in Figure 1.2(b). In this case, the ***amplitude*** of each sample is real-valued.

> **amplitude**
> *The maximum deviation of a quantity from a reference level.*

1.2.3 Digital Signals

In a digital system (such as a microcontroller), a discrete-time signal can be represented as an array, where each sample in the signal is an array entry. This array cannot hold real values and can only store values with limited range and resolution. Think of an `integer` array in the C language. Each array entry can only be represented by a certain number of bits. This will also be the case if the array is composed of ***float*** or `double` values. Therefore, the amplitude of the discrete-time signal should be ***quantized***. The resulting signal is called a digital signal. Figure 1.3 shows the digital signal for the same section of signal we used in Figure 1.2. In this case, the signal is stored in a `float` array.

> **quantization**
> *The process of mapping sampled analog data into non-overlapping discrete subranges.*

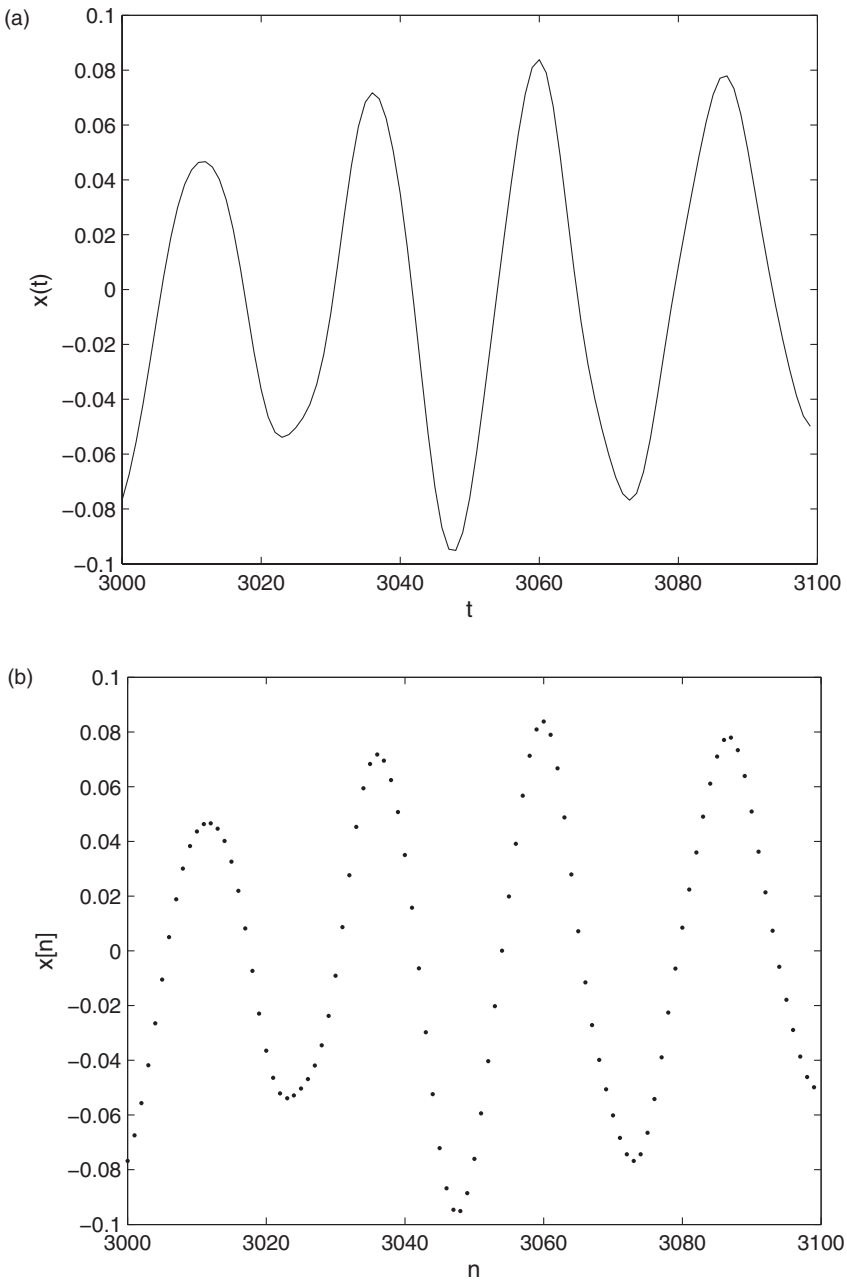

Figure 1.2 Part of the continuous-time signal and its sampled discrete-time version. (a) Continuous-time signal; (b) Discrete-time signal.

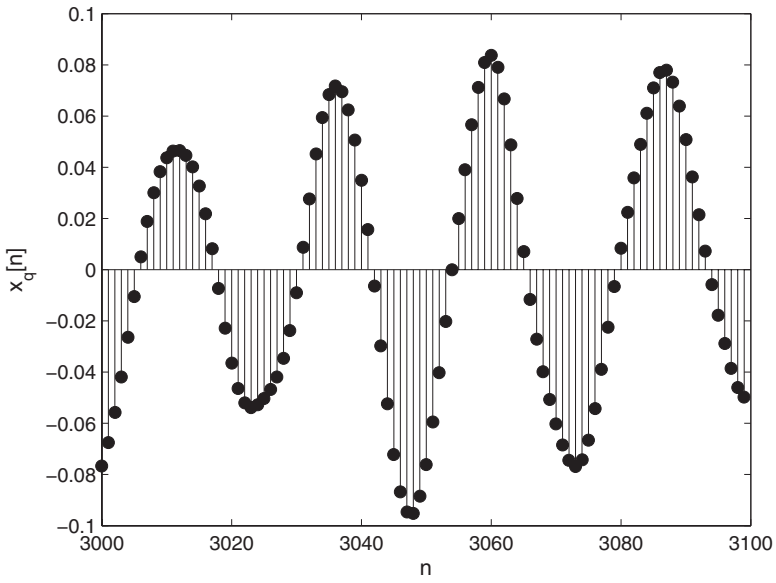

Figure 1.3 The digital signal.

float
The keyword for a floating-point data type with single precision.

Discrete-time signals and systems are preferred because they simplify the mathematical operations in theoretical derivations. On the contrary, digital signals and systems are necessary for implementation. This means that books focusing on the theoretical aspects of signal processing use discrete-time signal and system notations. Books on signal processing hardware and software use digital signal processing notations. Because this book aims to bridge the gap between theory and practice, we use the terms discrete time and digital interchangeably. To be more precise, we call a signal (or system) discrete-time when we explain its theoretical aspects, and we call it digital when we implement it.

1.3 Digital Signal Processing

In the past, analog systems were the only option for processing continuous-time signals. With the introduction of powerful hardware, digital signal processing has become more popular. Arm's Cortex-M based microcontrollers provide a low-cost and powerful platform for using DSP in practical applications. This book focuses on the theoretical and practical aspects of DSP concepts on Arm Cortex-M based microcontrollers.

The beauty of DSP is in its implementation because fixed analog hardware is generally not required. Instead, a suitable microcontroller is often sufficient for implementation. In fact, the DSP system will simply be a code fragment, which has several advantages. First, if the system does not satisfy design specifications, it can be redesigned by simply changing the relevant code block, and there is no need to change the hardware. Second, system parameters can be changed on the fly, which is extremely important for adaptive and learning systems. Moreover, if the microcontroller has communication capabilities, these parameters can be modified from a remote location. The abovementioned advantages make DSP more advantageous than continuous-time signal processing via analog equipment.

1.4 Lab 1

1.4.1 Introduction

The STM32F4 Discovery kit is the main hardware platform used in this book. It has a microcontroller, which is based on the Arm Cortex-M4 architecture. This lab introduces the properties of the kit and the microcontroller. While doing this, standard software packages will be used to simplify operations. Later on, the "AUP audio card"[1] will be introduced for use in audio processing. Being familiar with the properties of the discovery kit and the audio card will help in implementing and running the codes given.

1.4.2 Properties of the STM32F4 Discovery Kit

The STM32F4 Discovery kit has an STM32F407VGT6 microcontroller on it. In this section, we briefly introduce the microcontroller and the architecture of the kit.

The STM32F407VGT6 Microcontroller

The STM32F407VGT6 microcontroller is based on the 32-bit Arm Cortex-M4 architecture, which has ***Reduced Instruction Set Computing (RISC)*** structure. The microcontroller uses a decoupled three-stage pipeline to separately ***fetch***, ***decode***, and ***execute*** instructions. To be more specific, this operation can be summarized as follows. While the first instruction is being executed, the second one is decoded, and the third one is fetched. This way, most instructions are executed in a single CPU cycle. Related to this, with the help of pipelined RISC structure, Cortex-M4 CPU cores can reach a processing

1. Audio card from the Arm University Program

power of 1.27 and 1.25 ***DMIPS***/MHz with and without the floating-point unit (FPU), respectively.

Reduced Instruction Set Computing (RISC)
A microprocessor design strategy based on the simplified instruction set.

fetch
The first step of a CPU operation cycle. The next instruction is fetched from the memory address that is currently stored in the program counter (PC) and stored in the instruction register (IR).

decode
The second step of a CPU operation cycle, in which the instruction inside the instruction register is decoded.

execute
The last step of a CPU operation cycle in which the CPU carries out the decoded information.

DMIPS (Dhrystone Million Instructions per Second)
A measure of a computer's processor speed.

The Cortex-M4 CPU core uses a ***multiply and accumulate (MAC) unit*** for fixed-point calculations. This unit can execute most instructions in a single cycle with 32-bit data multiplication and hardware divide support. The hardware FPU can carry out floating-point calculations in a few cycles, compared to hundreds of cycles without the FPU. The FPU supports 32-bit instructions for single-precision (C float type) data-processing operations. It has hardware support for conversion, addition, subtraction, multiplication with optional accumulate, division, and square root operations. The Cortex-M4 CPU core has an additional set of instructions to perform parallel arithmetic operations in a single processor cycle for DSP-specific applications.

multiply and accumulate unit
A microprocessor circuit that carries a multiplication operation followed by accumulation.

The Cortex-M4 CPU core also has a ***nested vectored interrupt controller (NVIC)*** unit, which can handle 240 interrupt sources. The NVIC, which is closely integrated with the processor core, provides low latency interrupt handling. It takes 12 CPU cycles to reach the first line of the interrupt service routine code. Additionally, the Cortex-M4 structure has a ***memory protection unit*** (MPU), a ***wake-up interrupt controller*** (WIC) for ultra-low power sleep, instruction trace (ETM), data trace (DWT), instrumentation trace (ITM), serial/parallel debug interfaces for low-cost debug and trace operations, an ***advanced high-performance bus*** (AHB), and an ***advanced peripheral bus*** (APB) interface for high-speed operations.

interrupt
A signal to the processor emitted by hardware or software to indicate an event that needs immediate attention.

nested vectored interrupt controller (NVIC)
ARM-based interrupt handler hardware.

memory protection unit
The hardware in a computer that controls memory access rights.

wake-up interrupt controller
A peripheral that can detect an interrupt and wake a processor from deep sleep mode.

advanced high-performance bus
An on-chip bus specification to connect and manage high clock frequency system modules in embedded systems.

advanced peripheral bus
An on-chip bus specification with reduced power and interface complexity to connect and manage high clock frequency system modules in embedded systems.

The STM32F407VGT6 microcontroller has an Arm Cortex-M4 core with a 168-MHz clock frequency, 1 MB flash memory, 192 KB ***static random access memory*** (SRAM),

an extensive range of enhanced I/Os and peripherals connected to two APB buses, three AHB buses, and a 32-bit multi-AHB bus matrix. The STM32F407VGT6 microcontroller has three 12-bit ADCs, two ***digital to analog converters*** (DACs), a low-power real-time clock (RTC), two advanced-control timers, eight general-purpose timers, two basic timers, two ***watchdog*** timers, a SysTick timer, a true random number generator (RNG), three I^2C modules, three full-duplex ***serial peripheral interface*** (SPI) modules with I^2S support, four ***universal synchronous/asynchronous receiver/transmitter*** (USART) modules, two ***universal asynchronous receiver/transmitter*** (UART) modules, a high-speed and a full-speed USB ***on-the-go*** (OTG) module, two ***controller area network*** (CAN) modules, two ***direct memory access*** (DMA) modules, a ***secure digital input output*** (SDIO)/***multimedia card*** (MMC) interface, an ethernet interface, and a camera interface.

static random access memory
Static, as opposed to dynamic, RAM retains its data for as long as its power supply is maintained.

digital to analog converter
A device that converts a digital value to its corresponding analog value (e.g., voltage).

watchdog
A timer in an embedded system that is used to detect and recover from malfunctions.

serial peripheral interface
A serial communication bus used to send data, with high speed, between microcontrollers and small peripherals.

universal synchronous/asynchronous receiver/transmitter
A serial communication bus commonly used to send data, both synchronously and asynchronously, between microcontrollers and small peripherals.

universal asynchronous receiver/transmitter
A serial communication bus commonly used to send data, asynchronously, between microcontrollers and small peripherals.

on-the-go
A USB specification that allows a USB device to act as a host, allowing other USB devices to connect to themselves. Also called USB on-the-go.

controller area network
A vehicle bus standard designed to allow microcontrollers and devices to communicate with each other, in applications, without a host computer.

direct memory access
A mechanism whereby data may be transferred from one memory location to another (including memory-mapped peripheral interfaces) without loading, or independently of, the CPU.

secure digital input output
A circuit that allows the sending of data to external devices using Secure Digital (SD) specification.

multimedia card
A memory card standard used for solid-state storage.

The STM32F4 Discovery Kit

The STM32F4 Discovery kit has an ST-Link/V2 in-circuit debugger and programmer, a USB mini connector for power and debug operations, a USB Micro-AB connector for USB OTG operations, a reset button, a user push button, and four LEDs available to the user. It also has an 8-MHz main oscillator crystal, ST-***MEMS*** three-axis ***accelerometer***, ST-MEMS audio sensor, and a CS43L22 audio DAC with an integrated class-D speaker driver. The kit supports both 3.3 V and 5 V external power supply levels.

MEMS (Micro-Electro-Mechanical Systems)
Microscopic mechanical or electro-mechanical devices.

accelerometer
A device that measures change in velocity over time.

1.4.3 STM32Cube Embedded Software Package

STM32Cube is the software package released by STMicroelectronics for the STM32 family of Arm Cortex-M based microcontrollers. It contains a low-level hardware abstraction layer (HAL) and board support package (BSP) drivers for on-board hardware components. Additionally, a set of examples and middleware components are included in the STM32Cube package.

Including STM32Cube in a Project

STM32Cube is available in Keil μVision. In order to use it in a project, STM32Cube should be enabled from the **Manage Run-Time Environment** window by selecting **Classic** under **STM32Cube Framework (API)** as shown in Figure 1.4. Desired HAL drivers can then be selected from the **STM32Cube HAL** list.

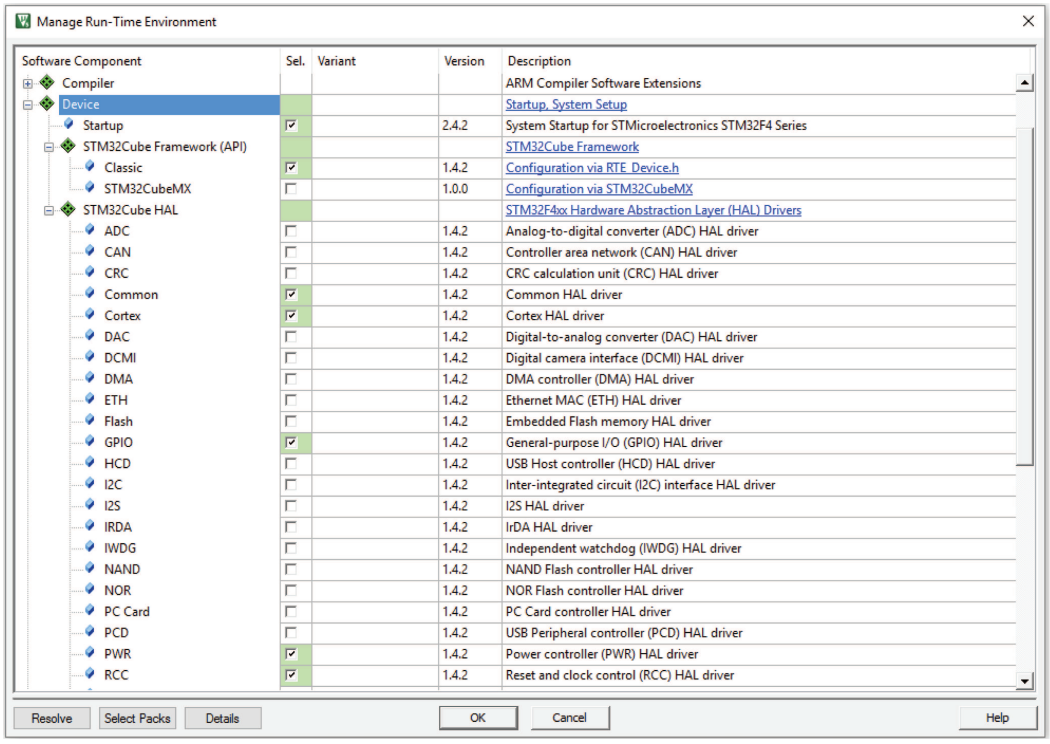

Figure 1.4 STM32Cube configuration by managing the run-time environment.

For our applications, **Common**, **Cortex**, **PWR**, **RCC**, and **GPIO** must be added before adding other HAL drivers. Some HAL drivers are connected, so they should be added together for them to work. Finally, the `stm32f4xx_hal.h` header file should be added at the top of the project's `main.c` file. Then, HAL drivers can be used in the source code.

Using BSP Drivers in the Project

To use BSP drivers in the project, first we need to specify the include folder for BSP drivers. To do this, click on the **Options for Target** button located in the toolbar, or select it from the **Project** tab. Click the **...** button next to the *Include Paths* text area from the **C/C++** tab. Then, add the full path of source files for BSP drivers under Keil's installation directory (`...\Arm\Pack\Keil\STM32F4xx_DFP\DFP_version\Drivers\BSP\`). Finally, click **OK** twice.

Application-specific header files must also be included in the source code. To add these specific files to the project, right click on the **Source Group 1** folder (in the **Project** window) and select **"Manage Project Items..."** as shown in Figure 1.5. From the pop-up window, click the **Add Files** button and navigate to the folders that contain the desired files.

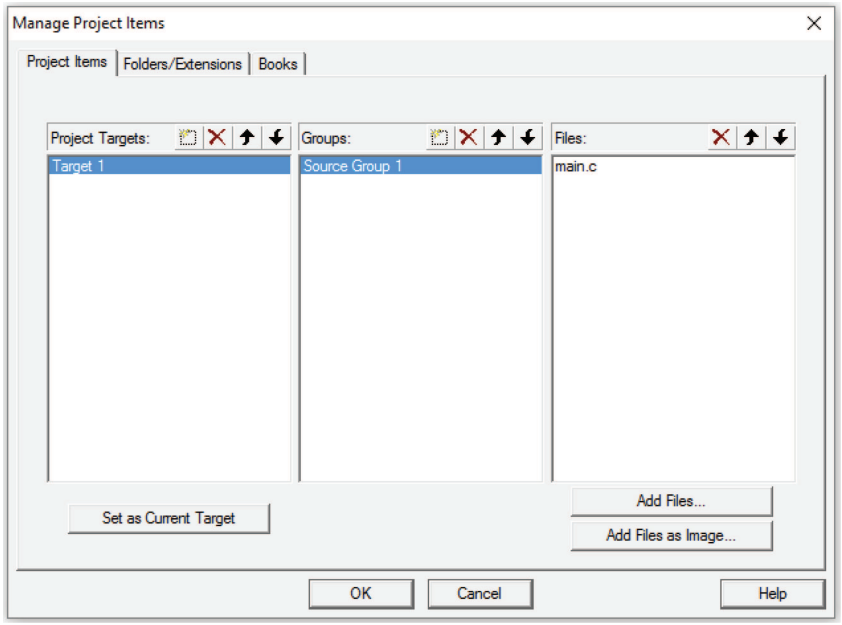

Figure 1.5 Adding BSP source files to the project.

Let us provide two examples on this topic. We use the onboard accelerometer from Section 1.4.6. To use it, you should add the `lis3dsh.c` and `lis302dl.c` files from the `...\Arm\Pack\Keil\STM32F4xx_DFP\DFP_version\Drivers\BSP\Components\lis3dsh` and `...\Arm\Pack\Keil\STM32F4xx_DFP\DFP_version\Drivers\BSP\Components\lis302dl` folders. You should also add the `stm32f4_discovery.c` and `stm32f4_discovery_accelerometer.c` files, along with their associated `.h` files, from the `...\Arm\Pack\Keil\STM32F4xx_DFP\DFP_version\Drivers\BSP\STM32F4-Discovery` folder. Similarly, if you want to use the onboard audio *codec*, then you should add

the `lis302dl.c` file from the `...\Arm\Pack\Keil\STM32F4xx_DFP\DFP_version\Drivers\BSP\Components\lis302dl` folder. The `stm32f4_discovery.c` and `stm32f4_discovery_audio.c` files should also be added from the `...\Arm\Pack\Keil\STM32F4xx_DFP\DFP_version\Drivers\BSP\STM32F4-Discovery` folder.

> **codec (coder-decoder)**
> *A device or computer program that allows the encoding or decoding of a digital data stream or signal.*

1.4.4 STM32F407VGT6 Microcontroller Peripheral Usage

We use the STM32F407VGT6 microcontroller peripherals through HAL drivers. The HAL driver library contains a set of generic and extension APIs, which can be used by peripheral drivers. While generic APIs are applicable to all STM32 devices, extension APIs can be used by a specific family or part number. The HAL library should be initialized first in order to use peripheral-specific HAL drivers in a project. We provide the general header file `hal_config.h` for peripheral configuration using HAL drivers in `Online_Student_Resources\Lab1`.

Although the STM32F407VGT6 microcontroller has a wide variety of peripherals, we will only mention the ones used in this book. We now focus on the peripherals we use.

Power, Reset, and Clock Control

HAL power control functions can be used to adjust the power configuration of the STM32F407VGT6 microcontroller. To do so, the APB interface should be enabled first after reset. Furthermore, low-power modes can be configured through these functions.

As mentioned in the previous section, we have the general header file `hal_config.h` for peripheral configuration. After initializing the HAL library, if this header file is included in the project, the `SystemClock_Config` function can be used to set the maximum clock frequency. You can make necessary changes to the power, reset, and clock configuration through this function.

General-Purpose Input and Output

The STM32F407VGT6 microcontroller has six physical general-purpose input and output (GPIO) ports, namely Port A, B, C, D, E, and H. These ports support up to 82 programmable ***input/output*** pins. These pins are called PA0-15, PB0-15, PC0-15, PD0-15, PE0-15, and PH0-1. All programmable input/output pins are accessible from the STM32F4 Discovery kit.

input/output
A circuit in an embedded system that connects the system to the external world.

The STM32F4 Discovery kit has one push button and four LEDs. The connections for these components are provided in Table 1.1. The push button is connected as weak pull-down, so the internal ***pull-up/down resistors*** must be used for it.

resistor
A passive electrical component designed to implement electrical resistance as a circuit element.

pull-down resistor
A pull-down resistor (to a negative power supply voltage) ensures that a signal conductor adopts a valid (electrical) logic level (low) in the absence of any other connection.

pull-up resistor
A pull-up resistor (to a positive power supply voltage) ensures that a signal conductor adopts a valid (electrical) logic level (high) in the absence of any other connection.

Basic HAL GPIO functions can be used to configure GPIO pins. In these, GPIO pins are configured as ***floating input***. During and after reset, all alternate functions and external interrupts are disabled by default. Here, the AHB clock used for the GPIO port must be enabled first using the `__GPIOx_CLK_ENABLE()` function. Then, the GPIO pins should be initialized using the `HAL_GPIO_Init()` function. Here, each GPIO pin can be configured as a digital input, digital output, or peripheral specific pin. Optionally, output drive strength, pull-up/down resistors, speed, and ***open-drain*** options for a GPIO pin can also be configured.

Table 1.1 Push button and LED connections.

GPIO Pin	Component
PD13	LD3 LED (Orange)
PD12	LD4 LED (Green)
PD14	LD5 LED (Red)
PD15	LD6 LED (Blue)
PA0	Push button 1 (B1)

floating input
An input pin with no signal source or termination connected.

open-drain
An output pin driven by a transistor, which pulls the pin to only one voltage.

We provide a sample project on the usage of GPIO functions in `Online_Student_Resources\Lab1\GPIO_Example`. This project uses the `hal_config.h` header file for peripheral configuration. If another configuration setting is required, such as using more than one port as input or output, then it should be performed manually. Here, pins 12, 13, 14, and 15 of the GPIO D port are defined as output. These are connected to the onboard LEDs of the STM32F4 Discovery kit. Pin 0 of the GPIO A port is defined as input, which is connected to the onboard push button of the STM32F4 Discovery kit. The sample project controls the status of LEDs with this push button.

There is also a counter in the code. Pressing the push button increases this counter. When the counter is at zero, all LEDs are turned off. When the counter is at one, the orange LED turns on. When the counter is at two, the green LED turns on. When the counter is at three, the red LED turns on. Finally, when the counter is at four, the blue LED turns on. Furthermore, there is a software debouncer to prevent any glitches due to a button press.

Interrupts

A total of 82 maskable interrupt channels and 16 interrupt lines in the STM32F407VGT6 microcontroller can be prioritized as 16 main and 16 sub-levels. Interrupts are controlled by the NVIC.

A brief summary of how the NVIC works is as follows. While the CPU is executing a non-interrupt code, it is said to be in thread mode. When an ***interrupt flag*** is raised, the NVIC will cause the CPU to jump to the appropriate interrupt service routine (ISR) and execute the code. At this stage, the CPU is said to be in handler mode. While switching from thread to handler mode, the NVIC performs two operations in parallel. First, it fetches the exception vector, which holds the address of the related ISR. Second, it pushes necessary key register content to the ***stack***. Whenever the CPU is in thread mode, this process takes exactly 12 clock cycles (independent of the code executed).

interrupt flag
A register bit used for indicating related interrupt status.

stack
A data structure in which items are removed in the reverse order from that in which they are added, so that the most recently added item is the first one removed.

Exception vectors are stored in an interrupt vector table. This table is located at the start of the address space, which is predefined as part of the startup code. A label for each ISR is stored at each interrupt vector location. To create an ISR function, a void C function must be declared using the same name as the interrupt vector label. When the program counter reaches the end of the ISR function, the NVIC forces the CPU to return from handler mode to the point in thread mode that it left. At this point, key register data are also retained from the stack.

The interrupt flag should be cleared at the beginning of handler mode in order to not miss any other generated interrupts. If the interrupt flag is cleared at the beginning of the ISR function and a new interrupt flag is raised while in handler mode, then the latter interrupt is nested. It waits until the former ISR function is executed. If two interrupt flags are raised at the same time, the CPU fetches the highest priority one first and nests the lower priority one. Moreover, if the interrupt flag is not cleared before returning from handler mode, the NVIC causes the CPU to incorrectly jump back to handler mode.

The HAL library has special functions to handle interrupts. These functions enable and disable interrupts, clear pending interrupt requests, and set the priority of interrupts.

The ISR function must be linked to the related interrupt vector before an interrupt is used. Normally, all unused interrupt vectors are linked to `Weak` interrupt functions in the `startup_stm32f407xx.s` file. When an interrupt is used, the related ISR function should be redefined. All ports have external interrupt/event capability. Each of the 16 external interrupt lines are connected to the multiplexed output of six GPIO ports of the microcontroller. For example, pin 0 of each port is connected to Line0, pin 1 of each port is connected to Line1, and so on. The port must be configured as the input mode to use external interrupt lines.

We provide a sample project on the usage of interrupt and GPIO functions in `Online_Student_Resources\Lab1\Interrupts_Example`. This project does the same job as the one given in `GPIO_Example`. The only difference here is that the program execution is controlled by interrupts. Again, pins 12, 13, 14, and 15 of the GPIO D port are defined as outputs, which are connected to the onboard LEDs of the STM32F4 Discovery kit. Pin 0 of the GPIO A port is defined as an external interrupt source, which is connected to the onboard push button of the STM32F4 Discovery kit. This pin is set to generate an interrupt when the button is pressed.

Timers

There are 17 timer modules in the STM32F407VGT6 microcontroller. These are two advanced-control timers, 10 general-purpose timers, two basic timers, two watchdog timers, and one SysTick timer. These are briefly described below.

Advanced-control timers (TIM1, TIM8): 16-bit three-phase ***pulse width modulation*** (PWM) generators multiplexed on six channels with full modulation. Can be used as a general-purpose timer.

> **pulse width modulation**
> *A modulation technique that generates variable-width pulses to represent the amplitude of an analog input signal.*

General-purpose timers (TIM3 and TIM4): Full-featured general purpose 16-bit up, down, and up/down auto-reload counter. Four independent channels for input capture/output compare, and PWM or one-pulse mode output.

General-purpose timers (TIM2 and TIM5): Full-featured general-purpose 32-bit up, down, and up/down auto-reload counter. Four independent channels for input capture/output compare, and PWM or one-pulse mode output.

General-purpose timers (TIM10, TIM11, TIM13, and TIM14): 16-bit auto-reload up counter and a 16-bit prescaler. One independent channel for input capture/output compare, and PWM or one-pulse mode output.

General-purpose timers (TIM9 and TIM12): 16-bit auto-reload up counter and a 16-bit prescaler. Two independent channels for input capture/output compare, and PWM or one-pulse mode output.

Basic timers (TIM6 and TIM7): 16-bit auto-reload up counter and a 16-bit prescaler. Mainly used for DAC triggering and waveform generation.

Independent watchdog timer: 12-bit down counter and 8-bit prescaler. Clocked from an independent 32 kHz internal RC oscillator. Operates independently of the main clock. Can be used as a free-running timer.

Window watchdog timer: 7-bit down counter. Clocked from the main clock. Can be used as a free-running timer.

SysTick timer: 24-bit down counter. Integrated into Cortex M4 core. Mainly used for generating a timer-based interrupt for use by an operating system. Whenever the predefined function `HAL_Init()` is added to the project, it is set to interrupt every millisecond. The `HAL_Delay()` function uses the SysTick to generate delay.

We will only use timers to generate time bases in this book. We will not consider other timer modes, such as capture, compare, PWM, one-shot, SysTick, and watchdog.

The HAL library provides generic and extended functions to control timers. These functions are used for enabling and disabling timers, setting time bases, configuring clock sources, and starting and stopping timers. In order to use a timer module, its

clock source, clock divider, count mode, frequency, and period must first be configured and initialized. Then, the AHB clock and interrupt for the timer module in use should be enabled. Interrupt priorities should also be set in the specific package initialize function. Finally, the timer module should be started in blocking (polling), non-blocking (interrupt), or DMA mode. The time duration generated by the timer module in seconds can be calculated as

$$Duration = \frac{(TimerPrescaler + 1)(TimerPeriod + 1)}{TimerClock} \tag{1.1}$$

We provide a sample project on the usage of timer functions and interrupts in `Online_Student_Resources\Lab1\Timers_Example`. This project uses the `hal_config.h` file to configure the timer module. Here, pins 12, 13, 14, and 15 of the GPIO D port are defined as outputs, which are connected to the onboard LEDs of the STM32F4 Discovery kit. The timer frequency is set to 10 kHz, and the timer period is set to 10000. This means an interrupt is generated every 10000 clock cycles. In other words, the interrupt period is 1 s. In the timer ISR, onboard LEDs are turned on one by one in a similar way to the `GPIO_Example`.

Analog to Digital Converter

There are three identical 12-bit ***analog to digital converter*** (ADC) modules in the STM32F407VGT6 microcontroller. These are called ADC0, ADC1, and ADC2. Each ADC module has 16 external inputs, two internal inputs, a V_{BAT} input, analog to digital converter block, data register blocks, interrupt control block, and trigger blocks. ADC modules share 16 analog input channels. The channels and pins related to them are listed in Table 1.2.

> **analog to digital converter**
> *A device that samples and quantizes an analog input signal to form a corresponding/representative digital signal.*

ADC modules in the STM32F407VGT6 microcontroller are based on the successive approximation register method. These can run in independent or dual/triple conversion modes. In the independent conversion mode, the ADC module can conduct single conversion from single-channel, single conversion from multichannel (scan), continuous conversion from single-channel, continuous conversion from multichannel (scan), and injected conversion. In dual or triple conversion modes, the ***sampling*** rate of the ADC module can be increased with configurable interleaved delays. Here, sampling rate is the number of digital samples obtained per second (sps). The speed of the analog to digital converter block defines the sampling rate of the ADC module in single mode. The clock for the ADC block (ADCCLK) is generated from the APB2 clock divided by a programmable prescaler. This allows the ADC module to work with clock speeds of

Table 1.2 STM32F407VGT6 microcontroller analog input channels.

Pin Number	Pin Name	Channel Name
23	PA0	ADC{1, 2, 3}_IN0
24	PA1	ADC{1, 2, 3}_IN1
25	PA2	ADC{1, 2, 3}_IN2
26	PA3	ADC{1, 2, 3}_IN3
29	PA4	ADC{1, 2}_IN4
30	PA5	ADC{1, 2}_IN5
31	PA6	ADC{1, 2}_IN6
32	PA7	ADC{1, 2}_IN7
35	PB0	ADC{1, 2}_IN8
36	PB1	ADC{1, 2}_IN9
15	PC0	ADC{1, 2, 3}_IN10
16	PC1	ADC{1, 2, 3}_IN11
17	PC2	ADC{1, 2, 3}_IN12
18	PC3	ADC{1, 2, 3}_IN13
33	PC4	ADC{1, 2}_IN14
34	PC5	ADC{1, 2}_IN15

up to 42 MHz. The total conversion time is the sampling time plus 12 clock cycles. This allows a sample and conversion rate of up to 2.8 Msps.

sampling
The process of obtaining values at specific time intervals.

The analog to digital conversion operation is initiated by a trigger in the ADC module. The source for this can be software, internal hardware, or external hardware in the STM32F407VGT6 microcontroller. A software trigger is generated by the `HAL_ADC_Start` function. The internal hardware trigger can be generated by timer events. The external hardware trigger can be generated by EXTI_11 or EXTI_15 pins.

In this book, we only use the ADC modules in single-channel and single conversion mode.

The HAL library provides generic and extended functions to control ADC modules. In order to use the ADC module, its clock divider, conversion mode, and resolution must first be configured and initialized. Then, the AHB clock and the interrupt used for the ADC module should be enabled. Next, the GPIO pin used in operation should be configured as analog input. Interrupt priorities should also be set in the specific package initialize function. Then, the sampling time and channel should be configured. The ADC module should be started in blocking (polling), non-blocking (interrupt), or DMA

mode. When the conversion is complete, the digital value should be read from the ADC register.

We provide three sample projects on the usage of ADC modules in `Online_Student_Resources\Lab1\ADC_Examples`. The first project uses the internal temperature sensor. Here, ADC1 is configured for 12-bit single conversion from Channel 16 (internal temperature sensor). It is set to be triggered with software. ADCCLK is set to 42 MHz. Sampling time is set to 84 cycles, which leads to approximately 437.5 Ksps. When the converted data are read, they are scaled to yield a temperature output in degrees Celsius. The second project uses the software trigger in the ADC module. Here, the PA1 pin is used as the ADC channel. The ADC1 module is configured for 12-bit single conversion from Channel 1. ADCCLK is set to 42 MHz. Sampling time is set to three cycles, which leads to approximately 2.8 Msps. The third project uses the ADC module with a timer trigger. Here, the PA1 pin is used for the ADC channel. The ADC1 module is configured for 12-bit single conversion from Channel 1. The timer module triggers the ADC every 1/10000 s. Moreover, the ADC interrupt is enabled so that an interrupt is generated when the conversion ends.

Digital to Analog Converter

There is one module to handle the digital to analog conversion operation in the STM32F407VGT6 microcontroller. There are two independent 12-bit DACs within this module. Each DAC has a buffered independent monotonic voltage output channel. Each DAC has one output, one reference input, a digital to analog converter block, a data register block, a control block, and a trigger block. Each DAC has a separate output channel. These channels and their related pins are listed in Table 1.3.

DACs can run in single or dual conversion modes. In the single conversion mode, each DAC can be configured, triggered, and controlled independently. In dual mode, DACs can be configured in 11 possible conversion modes. The DAC module is clocked directly from the APB1 clock and can give an output with or without a trigger. There can be no DAC triggers, or they can be set as internal hardware, external hardware, or software. If no hardware trigger is selected, data stored in the DAC data holding register, (`DAC_DHRx`), are automatically transferred to the DAC output register, (`DAC_DORx`), after one APB1 clock cycle. When a hardware trigger is selected and a trigger occurs, the transfer is performed three APB1 clock cycles later. The analog output voltage becomes available after a settling time that depends on the power supply voltage and the analog

Table 1.3 The STM32F4 microcontroller analog output channels.

Pin Number	Pin Name	Channel Name
29	PA4	DAC_CHANNEL1
30	PA5	DAC_CHANNEL2

output load. The DAC output register (DOR) value is converted to output voltage on a linear scale between zero and the reference voltage. This value can be calculated as

$$DAC_{output} = V_{REF} \times \frac{DOR}{4095} \tag{1.2}$$

DACs have output ***buffers*** that can be used to reduce the output impedance and to drive external loads directly without having to add an external operational amplifier. These buffers can be enabled or disabled through software. In this book, we will only use the single conversion operation mode in DAC.

> **buffer**
> *A circuit in which data are stored while it is being processed or transferred.*

The HAL library provides generic and extended functions to control the DAC module. The DAC module must be initialized before it is used. Then, the AHB clock should be enabled. The GPIO output pin should be configured as analog output in the specific package initialize function. Then, the output buffer and DAC channel trigger should be configured. Next, the DAC module should be started in blocking (polling), non-blocking (interrupt), or DMA mode. Finally, the digital data should be transferred to the output register to set the analog output value.

We provide three sample projects on the usage of DAC functions in `Online_Student_Resources\Lab1\DAC_Examples`. In the first project, the DAC module is configured for 12-bit single conversion to output from Channel 1 (PA4 pin) without triggering. In the second project, the DAC module is used with a software trigger. Here, the PA4 pin is used as the DAC channel output and DAC is configured for 12-bit conversion every second using delays. In the third project, the DAC module is used with a timer trigger. Here, the PA4 pin is used as the DAC channel output and the DAC module is configured for 12-bit conversion every second via the basic timer TIM6. TIM6 is configured to update TRGO output once the counter reaches its maximum value. DAC is triggered with this TIM6 TRGO output.

Direct Memory Access

DMA is a special module of the microcontroller. Through it, data transfer can be conducted in the background without using CPU resources, which means the CPU can conduct other tasks at the same time.

The STM32F407VGT6 microcontroller has two DMA modules with a total of 16 streams (8 per unit). Each stream has up to eight selectable channels. Each channel is dedicated to a specific peripheral and can be selected by software. Each channel has a ***First In First Out*** (FIFO) buffer with a length of four words (4 × 32 bits). These buffers can be used in FIFO mode to temporarily store the incoming data. Then, data

transfer can be initiated when the selected threshold value is reached. Threshold values can be 1/4, 2/4, 3/4, or full. FIFO buffers can also be used in direct mode to transfer data immediately. To transfer data, either normal or circular mode can be selected. The second mode will be especially useful while handling circular buffers, which we will consider in Chapter 11.

> **First In First Out**
> *FIFO is a method for organizing and manipulating a data buffer, where the oldest (first) entry, or "head" of the queue, is processed first.*

DMA modules in the STM32F407VGT6 microcontroller can initiate data transfer from memory to memory, from memory to peripheral, and from peripheral to memory. When the memory to memory transfer mode is selected, only the second DMA module can be used. Here, circular and direct modes are not allowed. The data width for the source and destination can be set as one byte (8 bits), half-word (16 bits), or word (32 bits). The DMA module can be configured for incrementing source and destination addresses automatically after each data transfer.

There are four different stream priorities for handling multiple DMA streams. If multiple DMA streams have the same priority, hardware priority is used (e.g., stream 0 has priority over stream 1). Burst transfer can be used when FIFO mode is selected. Here, the burst size can be selected as 4×, 8×, or 16× a data unit. FIFO buffers can also be configured as double buffer to support the ping–pong data transfer structure, which we will consider in Chapter 11.

We provide two sample projects on the usage of the DMA module in `Online_Student_Resources\Lab1\DMA_Examples`. In the first project, the content of a buffer is copied to another buffer in direct mode. In the second project, data are read from the ADC module and written into a buffer. The ADC module is triggered with software as explained in Section 5.3.

1.4.5 Measuring Execution Time by Setting Core Clock Frequency

We will use the DWT unit introduced in Section A.5 to measure the execution time of a code block. We will demonstrate its usage when the core clock is set to a certain frequency. As in Section A.5, the `time.h` header file should be added to the project. In order to set the core clock frequency, we will use the function `SystemClock_Config` in the `hal_config.h` header file.

Measuring Execution Time without the SysTick Timer

The first method of measuring the execution time does not use the SysTick timer. This timer is used as a time base. When the `HAL_Init` function is called in the code, it is

configured to create an interrupt every millisecond. However, the SysTick timer is not used when the DWT unit is used for measuring the execution time. If used by mistake, its interrupt generates additional clock cycles.

We provide a sample project on measuring the execution time of a code block in `Online_Student_Resources\Lab1\Measure_Execution_Time_I`. Here, the core clock is set to 168 MHz. There are some HAL functions used in this code. Therefore, you should arrange the project as explained in Section 1.4.3. Then, you can add the code block to be measured in the indicated place.

There are some differences between this code and the code mentioned in Section A.5. The first difference is in writing the core clock frequency. Here, the line `CoreClock = SystemCoreClock` is used instead of writing the core clock frequency manually to the `CoreClock` variable. This feature can only be used when the STM32Cube is enabled in the project. The second difference is in the `HAL_Init` and `SystemClock_Config` functions. They are used for setting the core clock frequency. The third difference is in the `HAL_SuspendTick` function. This function is used for disabling the SysTick timer interrupt.

In order to test the method introduced in this section, use the example given in Section A.5. Add a `for` loop with 2000000 iterations in the indicated place in the example code and run it. Observe the number of clock cycles and execution time. You can also use the State and Sec registers described in Section A.5 to measure the execution time.

Measuring Execution Time with the SysTick Timer

The second method of measuring the execution time is by using the SysTick timer. You should be aware that this method gives a time resolution at the millisecond level. We provide a sample project on measuring the execution time in `Online_Student_Resources\Lab1\Measure_Execution_Time_II`. Here, the core clock is set to 168 MHz. There are some HAL functions used in this code. Therefore, you should arrange the project as explained in Section 1.4.3. Then, you can add the code block to be measured in the indicated place.

If there is some additional code between the clock configuration and the code block to be measured, the SysTick timer must be disabled after configuring the clock. Then, using the `HAL_ResumeTick` function, the SysTick timer should be enabled before the code block to be executed. Execution time can be obtained in milliseconds using the `HAL_GetTick` function after the code block to be measured. Test this method by adding a `for` loop with 2000000 iterations in the indicated place in the code. Observe the execution time.

1.4.6 STM32F4 Discovery Kit Onboard Accelerometer

There are two accelerometer types available depending on the STM32F4 Discovery kit ***printed circuit board*** (PCB) revision. If the kit's PCB revision is MB997B (revision B),

then the *LIS302DL* accelerometer is available on the board. If the kit's PCB revision is MB997C (revision C), then the *LIS3DSH* accelerometer is available on the board. Both devices are controlled by the SPI interface.

> **printed circuit board**
> *A board made of fiberglass, composite epoxy, or other laminate material and used for connecting different electrical components via etched or printed pathways.*

Basic BSP accelerometer functions can be found in `Keil_v5\Arm\Pack\Keil\STM32F4xx_DFP\2.8.0\Drivers\BSP\STM32F4-Discovery\stm32f4_discovery_accelerometer.c`. In order to use these functions, the BSP must be included in the project as explained in Section 1.4.3. A sample project on the usage of accelerometer functions is given in `Online_Student_Resources\Lab1\Accelerometer_Usage_I`. Here, LEDs are turned on or off based on the board's motion. Four LEDs form a cross form between the two switches on the ST Discovery board. Each LED is closer to one edge of the board than the others. We process the accelerometer data so that the LED closest to the edge, facing downward, turns on.

1.4.7 The AUP Audio Card

The AUP audio card is a multipurpose audio expansion board. It is based on the TLV320AIC23B chip, which is a stereo audio codec with a headphone amplifier. The board is shown in Figure 1.6. We will use the AUP audio card to acquire audio signals. Therefore, we discuss its features in detail below.

- Analog line-level input and output via 3.5 mm line in and out sockets.
- High quality headphone output and microphone input via 3.5 mm headphone and microphone sockets.
- 8 to 96 kHz sampling-frequency support.
- ***Inter-IC sound*** (I2S)-compatible interface for audio stream.
- I2C-compatible interface for configuration.

> **Inter-IC sound**
> *A serial communication bus interface designed to connect digital audio devices.*

Connecting the AUP Audio Card

The AUP audio card is compatible with the STM32 Discovery kit. It can be connected as shown in Figure 1.6.

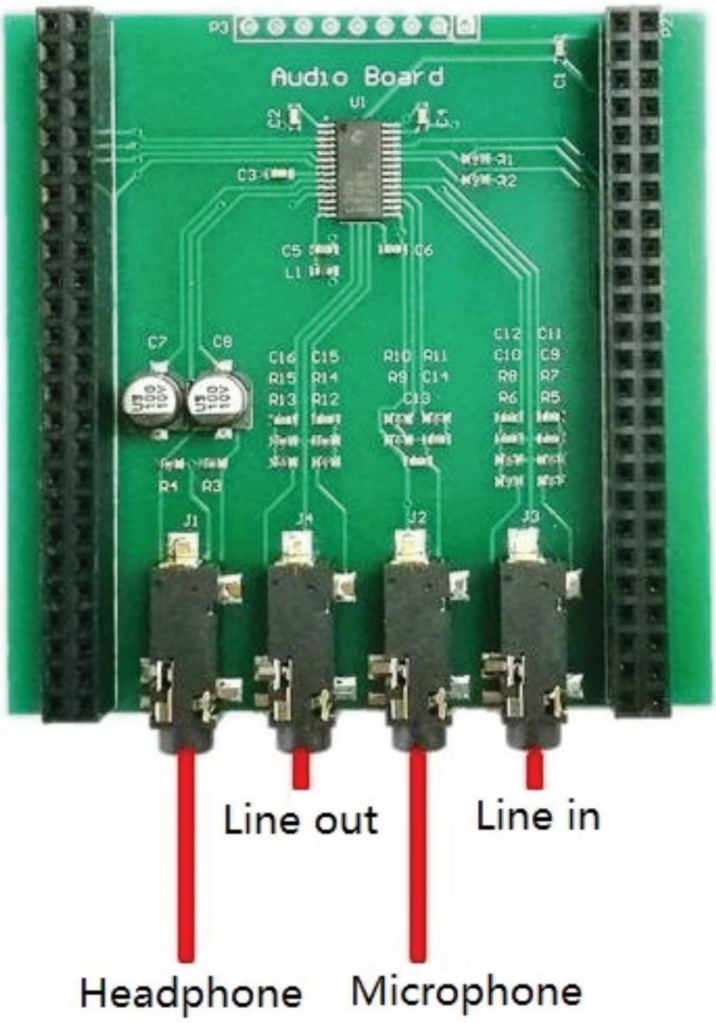

Figure 1.6 The AUP audio card. Photo by Arm Education.

Using the AUP Audio Card

In order to use the AUP audio card with the STM32F4 Discovery kit, the card must first be configured. The function to configure the AUP audio card is `AudioInit`. While calling the configuration function, the ***sampling frequency***, input source, and mode should be set. You can set the input source to digital microphone, analog microphone, or line-in using one of the code lines in Listing 1.1. You can set the sampling rate using one of the code lines in Listing 1.2 and set the mode using one of the code lines in Listing 1.3. We will only use the `IO_METHOD_INTR` mode, which is an interrupt-driven mode.

> **sampling frequency**
> *The reciprocal of the time delay between successive samples in a discrete-time signal.*

We provided the "predefined constants" for the "input source," "sampling rate," and "acquisition mode" of the audio card used in Listing 1.1, 1.2, and 1.3. In fact, all these definitions have been made in the `hal_config.h` file. The user can use these within the `AudioInit` function. For example, within the `LTEK_Example_Keil` project, they are used as `AudioInit(FS_32000_HZ,AUDIO_INPUT_MIC,IO_METHOD_INTR)`.

```
1                   AUDIO_INPUT_LINE
2                   AUDIO_INPUT_MIC
```

Listing 1.1 Setting the input source to digital microphone.

```
1                   FS_8000_HZ
2                   FS_16000_HZ
3                   FS_22050_HZ
4                   FS_32000_HZ
5                   FS_44100_HZ
6                   FS_48000_HZ
7                   FS_96000_HZ
```

Listing 1.2 Setting the sampling rate.

```
1                   IO_METHOD_INTR
2                   IO_METHOD_DMA
3                   IO_METHOD_POLL
```

Listing 1.3 Setting the mode.

```
1                   void ProcessData(I2S_Data_TypeDef* I2S_Data)
2                   /* Process data of the left channel. */
3                   if (IS_LEFT_CH_SELECT(I2S_Data->mask))
4                   I2S_Data->output_l=I2S_Data->input_l;
5
6                   /* Process data of the left channel. */
7                   if (IS_RIGHT_CH_SELECT(I2S_Data->mask))
8                   I2S_Data->output_r=I2S_Data->input_r;
9
10                  }
```

Listing 1.4 The ProcessData function.

The function for transmitting and receiving data from the AUP audio card is `ProcessData`. You should provide this function in the main code as given in Listing 1.4. The left and right audio channels can be selected using the `IS_LEFT_CH_SELECT` and `IS_RIGHT_CH_SELECT` macros. The `I2S_Data` structure contains the mask, input, and output data.

We provide a sample project on the usage of the TLV320AIC23B codec functions in `Online_Student_Resources\Lab1\L-Tek_Example`. The code in the project simply forms a loop between ADC input and DAC output on the AUP audio card. Data coming from microphones are directly fed to headphone output in the `ProcessData` subroutine.

1.4.8 Acquiring Sensor Data as a Digital Signal

We will use sensors on the STM32F4 Discovery kit and the AUP audio card to form digital signals. We will begin with the accelerometer.

Task 1.1

We provide the sample project to acquire accelerometer data in `Online_Student_Resources\Lab1\Accelerometer_Usage_II`. Use this project to acquire the sensor data for 1000 samples. Data are stored in the `buffer` array. While acquiring the data, shake the STM32F4 Discovery kit along the x-axis. To note here, the direction of this movement should be along the widest side of the ST Discovery kit with the mini USB connector facing forward. Observe the acquired data through the watch window as explained in Chapter 0.

Task 1.2

Now, we will use the internal temperature sensor. Use the ADC functions to acquire the temperature data for 1000 samples. To obtain data from the temperature sensor, make necessary adjustments as explained in Section 1.4.4. Observe the acquired data through the watch window as explained in Chapter 0.

Task 1.3

Finally, we will use the AUP audio card. Use the AUP audio card to acquire the audio signal when the word "HELLO" is spelled. Set the sampling rate to 32 kHz and buffer size to 32000. Store 1 s of audio signal in an array. Please refer to Section 1.4.7 to use the AUP audio card effectively. Observe the acquired data through the watch window as explained in Chapter 0.

Here, you should observe what actual sensor data look like. To note here, we will not process these data in real time until Chapter 11. We will only acquire the sensor data and process them offline. We follow this strategy so we do not have to deal with real-time signal processing concepts while introducing the basics of digital signal processing.