DIGITAL INTERFACE DESIGN AND APPLICATION

JONATHAN A. DELL

WILEY

DIGITAL INTERFACE DESIGN AND APPLICATION

DIGITAL INTERFACE DESIGN AND APPLICATION

Jonathan A. Dell
University of York, UK

WILEY

This edition first published 2015
Copyright © 2015 by John Wiley & Sons, Ltd. All rights reserved

Registered Office
John Wiley & Sons, Ltd., The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at www.wiley.com.

The right of the author to be identified as the author of this work has been asserted in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by the UK Copyright, Designs and Patents Act 1988, without the prior permission of the publisher.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The publisher is not associated with any product or vendor mentioned in this book.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. It is sold on the understanding that the publisher is not engaged in rendering professional services and neither the publisher nor the author shall be liable for damages arising herefrom. If professional advice or other expert assistance is required, the services of a competent professional should be sought

Library of Congress Cataloging-in-Publication data applied for.

ISBN: 9781118974322

A catalogue record for this book is available from the British Library.

Cover Image: mishooo/iStockphoto

Set in 10.5/13pt Times by SPi Global, Pondicherry, India

1 2015

Contents

List of Figures x

List of Tables xiii

Preface xv

1 Review of Digital Electronics and Computer Architecture 1

1.1 Embedded Systems 1

1.1.1 Processor Architecture (Revision) 2

1.1.2 Interface Subsystem 3

1.2 Software Architecture 4

1.3 Essential Basic Logic Elements 5

1.3.1 The Basic Flip/Flop 5

1.3.2 The Edge-Triggered D-Type Flip/Flop (Latch) 7

1.3.3 Edge-Triggered Latch with Enable 8

1.3.4 Multi-Bit Registers 9

1.4 Output Configuration Options 10

1.4 1 Open Drain Configuration 10

1.5 The Address Decode 11

1.5.1 Partial Address Decode 12

1.6 ARM Architecture 14

1.7 Interface Software Development 14

1.7.1 Software Development for Embedded Systems 18

1.8 C Programming Revision 19
1.8.1 Arrays 19
1.8.2 Structures and typedef 21
1.8.3 Header Files 21
1.9 Conclusion 22
References 23
Further Reading 23

2 Simple Input and Output Functions **24**
2.1 Introduction 24
2.2 Computer Structure 25
2.3 Simple Interface Circuit Concepts 26
2.3.1 An Output Interface 26
2.3.2 Address Decode for Output 28
2.3.3 A Simple Input Interface 29
2.3.4 Address Decode for Input 29
2.4 Activation of I/O Circuits 30
2.4.1 Programming an Output 30
2.4.2 Programming an Input 31
2.5 Universal I/O Circuits 31
2.5.1 Combined I/O Address Decode 32
2.6 Practical I/O Circuits 33
2.6.1 STM32F4 Address Decoding 35
2.7 A Typical I/O Programme 35
2.7.1 Example GPIO Application 37
2.7.2 A Summary of Alternative I/O Operations 40
2.7.3 Programming I/O in Assembler Language 41
2.8 Suggested Design Challenge 41
2.9 Conclusion 43
References 44
Further Reading 44

3 Timer Subsystems **45**
3.1 Timer Subsystems 45
3.2 Basic Timer Configuration 46
3.3 The STM32F4 Timers 47
3.3.1 The Individual Timers 50
3.4 Programming the STM32F4 Timers 51
3.5 Timer Triggering 55

3.5.1 Setting up the Time-Base 55
3.5.2 Using the Timer for an Input Measurement 56
3.6 Basic Timers 58
3.7 PWM Applications 61
3.8 Programming Challenge 63
3.9 Conclusion 64
References 65

4 Analogue Interface Subsystems 66
4.1 Analogue Interfaces 66
4.2 Digital to Analogue 67
4.2.1 The STM32F4 DAC 69
4.3 Analogue to Digital Conversion 69
4.3.1 Sampling 70
4.3.2 Switched Capacitor Converter 72
4.3.3 The Software Interface 73
4.3.4 The STM32F4 ADC 74
4.4 Software Control of DAC 75
4.4.1 Waveform Generation 76
4.4.2 Waveform Timing 77
4.4.3 DAC Using DMA 79
4.5 Software Control of ADC 83
4.5.1 ADC Interface Using Timer and DMA 85
4.6 Programming Challenge 88
4.7 Conclusion 89
References 89
Further Reading 89

5 Serial Interface Subsystems 90
5.1 Introduction 90
5.2 RS232 Universal Asynchronous Receiver/Transmitter (UART) Communications 91
5.3 The I2C Interface 95
5.3.1 Using the Touch Screen with an I2C Interface 96
5.4 SPI Interface 101
5.4.1 SPI Interface to an Analogue to Digital Converter 103
5.5 HDLC Serial Communication 105
5.6 The Universal Serial Bus (USB) 107
5.6.1 Hand-shake Packets 109

5.6.2 Token Packets 109
5.6.3 Data Packets 109
5.6.4 USB Protocol 110
5.7 Programming Challenge 110
5.8 Conclusion 111
References 111

6 Advanced Functions **112**
6.1 Advanced Functions 112
6.2 Interrupts 112
6.2.1 Interrupts in the STM32F4 114
6.2.2 The Nested Vector Interrupt Controller (NVIC) 115
6.2.3 Exceptions 117
6.3 Direct Memory Access (DMA) 118
6.3.1 The STM32F4 DMA System 118
6.3.2 DMA Request Mapping 119
6.3.3 DMA Management 119
6.4 The LCD Display Module 121
6.4.1 Character Generation 125
6.4.2 Parallel Interface 127
6.4.3 Touch Screen 128
6.5 The Wireless Interface Module 131
6.6 Digital Camera Interface 133
6.7 Conclusion 134
Further Reading 134

7 Application Case Study Examples **135**
7.1 An Open-Loop Digital Compass 135
7.1.1 Program Design 136
7.1.2 Setting up the MAG3110 136
7.1.3 Programming Challenge: A 360° Servo 140
7.2 The MSF Time Decoder 140
7.2.1 MSF Receiver Circuit Arrangement 141
7.2.2 Program Design 141
7.2.3 Setting up for an Interrupt 142
7.2.4 Acquiring the Data Bits 143
7.2.5 Decoding the MSF Data 147
7.2.6 Displaying the MSF Time Data 150

7.3 Decoding GPS Signals 153
7.3.1 Acquiring the GPS Message 154
7.3.2 Decoding the GPS Message 155
7.3.3 Selecting the Massage Stream 158
7.4 Conclusion 159
References 160

Appendix A: uVision IDE Notes **161**
A.1 Getting Started 161
A.2 Help 162
A.3 Project Development 162
A.4 Debug Facilities 164
A.5 Conclusion 167

Appendix B: STM Discovery Examples Library **168**
B.1 Peripheral Examples 168
B.2 Example Application 171

Appendix C: DAC and ADC Support Software **175**
C.1 DAC Peripheral Features 175
C.2 How to Use the DAC Driver 176
C.3 ADC Peripheral Features 177
C.4 How to Use the ADC driver 177
C.5 Files for Reference 178

Appendix D: Example Keyboard Interface **179**

Index **185**

List of Figures

Figure 1.1	Fundamental computer architecture	2
Figure 1.2	Interface software structure	4
Figure 1.3	Cross-coupled flip/flop circuit	6
Figure 1.4	Uncoupled circuit	6
Figure 1.5	D-type edge-triggered latch	7
Figure 1.6	Timing diagram	7
Figure 1.7	Flip/flop with enable functionality	8
Figure 1.8	Four-bit register with group input enable IE and group output enable OE	9
Figure 1.9	Push/pull output driver	10
Figure 1.10	Wired-AND connection of multiple outputs	11
Figure 1.11	Full address decode example	12
Figure 1.12	Partial address decode	13
Figure 1.13	Block decode	13
Figure 1.14	Screen-shot for uVision4 IDE	15
Figure 1.15	IDE management window	20
Figure 2.1	Fundamental processor structure	25
Figure 2.2	Bus timing relationships	26
Figure 2.3	Basic latch	27
Figure 2.4	A simple output interface	27
Figure 2.5	Simple decode for output	28
Figure 2.6	A tri-state buffer	29
Figure 2.7	A simple input interface circuit	29

Figure 2.8 Combined I/O 32
Figure 2.9 GPIO pin circuit 34
Figure 2.10 Observed code output pattern 39
Figure 2.11 16-key matrix keyboard 42
Figure 2.12 Keyboard interface connections 43
Figure 2.13 Timing diagram for keyboard interface 43
Figure 3.1 A simple timer function 47
Figure 3.2 The simplified timer architecture 48
Figure 3.3 Upward count mode 49
Figure 3.4 Downward count mode 49
Figure 3.5 Centre aligned mode 49
Figure 3.6 Timing diagram for example code 54
Figure 3.7 Capture/compare input circuit 57
Figure 3.8 Simplified timers 59
Figure 3.9 A PWM signal set at 40% 61
Figure 3.10 Servo control with PWM 64
Figure 3.11 Servo controller circuit diagram 64
Figure 4.1 Binary weighted DAC network 68
Figure 4.2 DAC block diagram 68
Figure 4.3 A 3-bit DAC transfer characteristic 69
Figure 4.4 Simple ADC subsystem design 70
Figure 4.5 Sample and hold 71
Figure 4.6 Sampling a sine wave 71
Figure 4.7 Switched capacitor converter 72
Figure 4.8 ADC interface timing 73
Figure 4.9 Triangle waveform 76
Figure 4.10 Timer triggering of DAC 77
Figure 4.11 Temperature sensor circuit 88
Figure 5.1 Manchester code 91
Figure 5.2 RS232 data format 92
Figure 5.3 UART module 92
Figure 5.4 An I2C interface circuit 95
Figure 5.5 I2C read and write modes 96
Figure 5.6 SPI interface connections 102
Figure 5.7 AD7680 connections 104
Figure 5.8 HDLC frame structure 106
Figure 5.9 HDLC frame control byte 106
Figure 5.10 USB connections 107
Figure 5.11 Touch screen interface 111
Figure 6.1 Interrupt basic concepts 113

Figure 6.2 Part of the NVIC 115
Figure 6.3 Simple processor with DMA 118
Figure 6.4 DMA controller architecture 119
Figure 6.5 An upper-case A character 126
Figure 6.6 The WiFi module configuration 132
Figure 6.7 Digital camera interface (DCMI) configuration 133
Figure 7.1 The compass interface 136
Figure 7.2 MSF signal modulation format 140
Figure 7.3 MSF receiver circuit 141
Figure 7.4 GPS module connection 153
Figure A.1 The uVision screen (part) 162
Figure A.2 A typical help screen 163
Figure A.3 Context sensitive help 163
Figure A.4 Project file structure 164
Figure A.5 Project compiling and linking 164
Figure A.6 A debug run 165
Figure A.7 Debug tools 165
Figure A.8 A system viewer window 166
Figure A.9 Timer 3 registers 166
Figure A.10 Watch windows 167

List of Tables

Table 1.1	Memory map	3
Table 1.2	Truth table for the simple flip/flop	6
Table 1.3	Truth table for edge-triggered latch	7
Table 1.4	Memory map	13
Table 1.5	Code interpretation	18
Table 2.1	Output address allocation	28
Table 2.2	Input address allocation	30
Table 2.3	I/O address allocation	33
Table 2.4	STM32F4 address allocations	35
Table 2.5	Programme linked comments	37
Table 4.1	DAC output allocation	76
Table 4.2	ADC input allocations	84
Table 5.1	STMP811 system control register 2	98
Table 5.2	Temperature gauge control register	98
Table 5.3	USB PID byte functions	108
Table 6.1	An extract of the vector table; the full table has more than 81 entries!	114
Table 6.2	NVIC priority groups	116
Table 6.3	DMA allocated channels	120
Table 6.4	Interface port mappings	122
Table 6.5	Pixel assignments	125
Table 6.6	Demonstration functions	132

Table 7.1 MAG3110 register summary 137
Table 7.2 MSF data bit allocations 141
Table 7.3 GPGGA navigation message format 154

Preface

This book aims to provide a justified and well founded cornerstone in the development of techniques required to establish reliable interface designs used when embedded computers are deployed in any demanding application. The book will focus on the ARM Microprocessor, which is now a leading technology in the electronics industry and offers a wide range of performance optimised features for particular applications. By using simple practical examples, the link between the embedded hardware and the programming task will be clearly developed so that interface design can be undertaken with confidence and the resulting systems will exhibit high reliability.

ARM Microprocessors have developed quickly over the last few years and have been very widely adopted by the electronics industry, finding pervasive applications in mobile phones, sensor networks and server systems to pick just a few examples. Applications will continue to develop and become more diverse over the years to come so a wide knowledge background in interface design will be highly valuable.

This book will bring together aspects of digital hardware, interface design and software integration (not otherwise available in a single text) in a progressive arrangement to promote thorough comprehension of the examples. In particular, the intimate linkage between low and high level languages (HLLs) will be explained so that the advantages of optimisation can be considered carefully. In many cases, the HLL approach will deliver rapid productivity but performance optimisation will require a more detailed knowledge of the overheads involved.

Readers should have a basic knowledge of digital electronics such as that established in the early years of an undergraduate degree course. The book is aimed at any student studying the requirements of interface design and although starting at an introductory level it will also provide a reference work for those involved in a wide variety of interface projects. Comprehensive details will be provided to enable access for the widest possible readership including those with a more software oriented background.

The author has over 25 years' experience of teaching microprocessors and interfacing at York University, and previously, Sheffield Hallam University, although this was known as Sheffield Polytechnic at the time. Prior to his teaching career, the author worked for several industrial organisations mainly focussing on various projects involving microprocessors, digital electronics and computer automated electronic test systems. These teaching and development experiences have provided the author with an extensive knowledge of microcomputer architectures and interface design, which has enabled the development of this book.

The author's earliest work with microprocessors used a very inflexible four-bit single-chip computer produced by Texas Instruments. This had no integrated peripherals and a very awkward architecture based on its optimisation for a pocket calculator role. Programming this chip involved using raw machine code and some limited assembler facilities. Fortunately, this was soon superseded by 8 and 16 bit architectures that offered full assembler programming support but still required may additional components to provide memory and peripheral resources. These types of system required considerable hardware and software development to incorporate all the elements that would be needed for a particular application. The introduction of HLL compilers and integrated development systems (IDE) provided significant advantages in productivity for system developers and higher levels of hardware integration allowed more extensive programmable interface resources to be included so removing the need for extensive hardware design. The current availability of systems based on the ARM processor and integrated peripherals, such as those found in the STM product STM32F4, together with its IDE software package, bring together all the advantages offering amazing capabilities and diverse application possibilities with rapid development.

The author has dealt with the issue of learning about many different processor developments over his career but has always found that a small program designed to flash a single LED was the best way to make a rapid start with the new technology. This approach can be recommended wholeheartedly.

Jonathan A. Dell
University of York, UK
15 October 2014

1

Review of Digital Electronics and Computer Architecture

This chapter will focus on the operation of basic logic elements, which should be familiar to most readers, such as flip/flops and registers that form the basic building blocks for interface elements. It will also cover the issue of address decoding to enable these elements through a programme statement and provide an introduction to the ARM architecture and its built-in peripherals. Finally the linkage between C and low-level assembler code will be shown through a simple example.

1.1 Embedded Systems

Embedded systems take many different forms but all rely on some computing processor whether it has a physical interface with the outside world like the keyboard and screen of a typical PC, the physical connections forming a communication network or specialised sensor and actuator hardware to control an automated machine. The other essential element of any system making use of a computer is its controlling program and in fact most embedded system component vendors provide integrated development systems or environments (IDE) that run on a PC to facilitate the software development. Interface design requires careful assessment of both the hardware and software requirements

Digital Interface Design and Application, First Edition. Jonathan A. Dell.
© 2015 John Wiley & Sons, Ltd. Published 2015 by John Wiley & Sons, Ltd.

so that when the objectives are considered the most effective solution can be achieved. In summary, we can say that an embedded system is computerised and tailor-made for a particular application.

1.1.1 Processor Architecture (Revision)

It is important to have a clear understanding of the processor architecture so that the integration of interface peripherals can be appreciated. The fundamental hardware architecture of the computing element will be examined briefly to show where interface components of any variety link up. Figure 1.1 shows a simple architecture of processor and memory that is widely adopted; this is interconnected by address, data and control buses. It is assumed that the reader is familiar with the general operation of the processor in terms of the machine instructions and data held in the memory as well as the cyclic operations of instruction fetch and execution to perform the desired task. The interface circuits connect onto the same buses, so as far as the processor is concerned the interfaces look like an extension to the memory. An advantage of this arrangement is that the same processor instructions can be used to manipulate data in memory or interface values.

As a reminder, the function of the address bus is to differentiate all the different elements of the system, as no two parts can have the same address or a conflict and confusion will arise. It is usual to use a memory map in order to show how the range offered by the address bus is allocated to different parts of the system. Table 1.1 shows how memory and interface elements might be allocated in a simple system with a 16-bit address bus.

The data bus has an obvious function but it should be noted that its operation is bi-directional; except in special circumstances data is sent to the processor or delivered by it. The most important task of the control bus is to indicate the data bus flow direction and thus avoid a bus conflict. Memory and interface

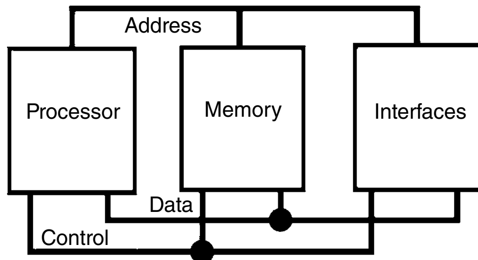


Figure 1.1 Fundamental computer architecture

Table 1.1 Memory map

Address range	System element
0xc000 to 0xffff	I/O interface
0xb000 to 0xbfff	Redundant space
0x9000 to 0xafff	12k Data memory
0x0000 to 0x8fff	36k Programme memory

elements can then be arranged to perform the appropriate read or write function as demanded. In a complex processor like the ARM the control bus may have many other functions to accommodate such as direct memory access (DMA), where the processor operation is temporarily suspended, and interrupt, which enable hardware signals and specific software instructions to be linked. These special functions will be examined in more detail in a later chapter.

The binary machine instructions held in the memory for the processor to access are usually determined from a high-level programming language, like C, for example by a Compiler but this approach will always involve some redundancy or overhead that may form an unacceptable burden in some demanding situations where memory is limited or processing time is at a premium. So for greater efficiency a low-level assembler language, which has a close linkage with each machine instruction, is sometimes employed. Unfortunately the assembler language approach introduces considerable complexity and is associated with low programming productivity, so it should only be considered for the most demanding applications.

1.1.2 Interface Subsystem

When an interface circuit is designed it is essential to link up with the bus signals using appropriate hardware circuits. In essence, the address bus must connect with an address decode function to create latch enable signals and the data bus must connect with output from registers or special input functions. Finally, the control line, representing data bus direction, is usually combined with the address decode logic so that an output function or input function can be enabled correctly when bus access is demanded. This is quite a complex task for an interface designer but some typical circuits will be discussed in more detail in the following sections to show the principles involved. In fact, the STM32F4 has many such interface subsystems integrated within the microprocessor chip itself making their application more straightforward for the user, removing a complex task for the designer. The resulting address map

for the STM32F4 device is shown in (ARM Cortex-4 Data Sheet Doc ID 022152 Rev 3 [1]) Section 1.4 and the principal eight 512 Mbyte blocks are shown on the left hand side. Each of these blocks is further sub-divided for the many different roles within the processor.

1.2 Software Architecture

Any interface design will involve elements of both hardware and software development. Fortunately for many applications the hardware design is almost complete because it can make use of the inbuilt components. So the essential tasks for the interface designer involve the use good software practice and implement reliable operation. The C language will be used for the examples throughout as this promotes a good software structure. If the design has special performance requirements where assembly language should be used much greater care in implementing the code is needed.

For the software design, a slightly more abstract model is required and it is most effective to start with a simple system block diagram arranged to emphasise the interface functions, as illustrated in Figure 1.2. Any potential application will require a diagram of this type so it is beneficial to start one at an early stage in the design process and keep it updated with changes as the design proceeds. An early development might identify the different aspects of initialisation and use of the particular interface elements, for example.

This diagram simply shows a conceptual application, which makes a call on specific interface elements to control an actuator of some kind or deliver on-screen status messages to the user. The links are shown as bi-directional to

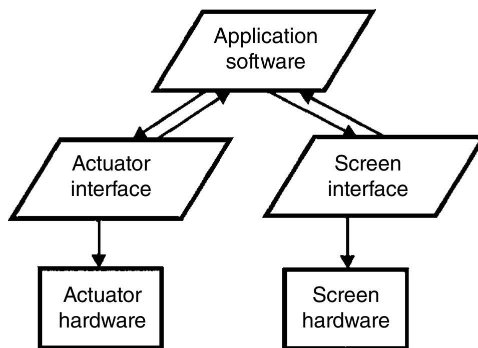


Figure 1.2 Interface software structure

account for the feedback that should be delivered in a good design to confirm correct interface operation. In a practical situation this may also require additional status signals from the actual hardware to provide the possibility of a hand-shake communication with the application program. Most of the built-in peripheral subsystems provide a host of status signals that can be used to implement various forms of hand-shake.

For the software aspects of interface design and development this book will make use of the Keil uVision4 Integrated Development Environment (IDE) provided by ST Microelectronics, which includes a C compiler. A brief introduction to the uVision4 platform will be provided towards the end of this chapter and Appendix A and B are included to provide a brief tutorial reference to help when readers lack experience with this particular package. The debug facilities included in the package provide valuable tools to track program flow, observe changes in variable data and examine information from the integrated peripherals.

1.3 Essential Basic Logic Elements

A brief resume of significant logic elements is provided in this section for reference to show in particularly how interface circuits are constructed. The interface designer will rarely need to develop new interface circuits but it is useful to have a clear understanding of the main principles so that system block diagrams of complex subsystems can be appreciated more fully.

It will be assumed that the reader has working familiarity with logic gates and the creation of combinational functions using several gates, as well as optimisation of the circuit using Boolean equations and Karnaugh mapping. If revision of these topics is required the reader should refer to a suitable reference text of which there are many, Wakerly in reference [2] is particularly recommended. For interface circuits the most important elements apart from simple gates are the flop/flops so this type of component will be reviewed in more detail in the following sections.

1.3.1 The Basic Flip/Flop

The simplest two-state flip/flop can be created from a pair of cross-coupled NAND gates as shown in the circuit Figure 1.3. This circuit is rather difficult to analyse because of this coupling so it is most convenient to temporarily break one of the links creating an extra input q as shown in Figure 1.4. If the

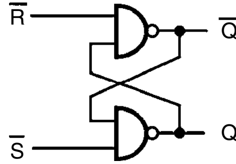


Figure 1.3 Cross-coupled flip/flop circuit

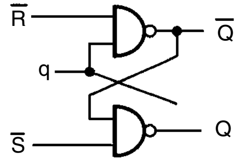


Figure 1.4 Uncoupled circuit

Table 1.2 Truth table for the simple flip/flop

$\bar{S}$	$\bar{R}$	Q	$\bar{Q}$
0	0	1	1
0	1	1	0
1	0	0	1
1	1	Last Q	Last $\bar{Q}$

inputs on $\bar{R}q\bar{S}$ are 111, $\bar{Q}$ will be 0 and Q will be 1 by virtue of the NAND logic so reconnecting will make no change and the circuit will be stable in the set state. If the inputs are 001 $\bar{Q}$ will be 1 and Q will be 0 by virtue of the logic so reconnecting will again make no change and the circuit will be stable in the reset state. In cases when q and Q are different when the reconnection is made the subsequent change brings the circuit to one of its stable conditions. For example, when $\bar{R}q\bar{S}$ is 011 $\bar{Q}$ is 1 and Q is 0 by virtue of the logic so the subsequent change due to reconnection makes q become 0 but this will not alter the circuit from its reset state.

However, this particular circuit is not functionally convenient in most applications because when the set and reset inputs are active (i.e. at logic 0) simultaneously the resulting state is not defined because both outputs take up the same logic level and the subsequent stable state when the inputs are deactivated is arbitrary. This conflicting situation is shown in the first row of the truth table in Table 1.2.

1.3.2 The Edge-Triggered D-Type Flip/Flop (Latch)

The most practical form of flip/flop, usually referred to as a *latch*, uses a more complex circuit containing a minimum of six gates, which will not be discussed in detail here, and introduces the idea of a 'data' input and a 'clock' as shown in the block diagram in Figure 1.5. A good description of this function can be found in Wakerly [2]. The functionality of this form of latch is most useful to know and can be described in several ways, that is truth table, Table 1.3 and timing diagram, Figure 1.6, both of which are useful in some circumstances. In either case the clock edge, the rising edge for the sake of this particular description, determines when the latch state changes and the D input determines the next state value, that is when D = 1 the resulting state is set (Q = 1) or when D = 0 the resulting state is reset (Q = 0), as shown in the timing diagram in Figure 1.6. Importantly, the input D has no effect and the state will not change until a clock edge arrives.

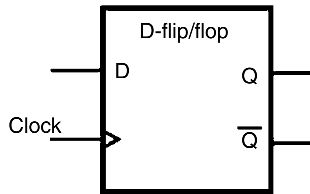


Figure 1.5 D-type edge-triggered latch

Table 1.3 Truth table for edge-triggered latch

Clock	D-input	Q	Q next
0	X	—	No change
1	X	—	No change
Edge	0	X	0
Edge	1	X	1

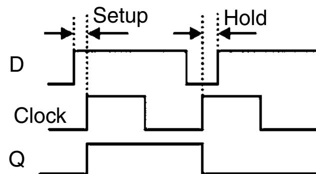


Figure 1.6 Timing diagram

It should be observed that in the example timing diagram shown the latch state only changes when there is a rising clock edge. If the clock is at a steady high or low the latch state does not change even when the D input changes. Note also that for reliable operation in practice the D input must be stable a few nano-seconds before the clock edge and remain stable for a few nano-seconds after. In logic documentation these times are referred to as the setup and hold time. In most practical applications these requirements are not difficult to achieve because the bus timing specifications are set up conveniently to avoid any critical timing situations so problems should not arise unless long chains of gate are employed at any point in the system.

Note that in a special alternative form of this function the latch state follows the D input when clock is high so the resulting state is only stable indefinitely when clock is taken low. In logic terminology this operation is referred to as a 'transparent' latch because when Q follows D while clock is high and the latch appears to do nothing. This type will not be used in the interface circuits discussed subsequently but may find application in some special circumstances.

For applications in interface circuits the edge-triggered D-type latch has significant advantages of reliability and will be employed throughout. It is, however, important to carefully consider the timing relationship between clock and the D input because when the criteria of setup and hold time is compromised the resulting state will not always follow the designer's expectations.

1.3.3 Edge-Triggered Latch with Enable

In practical situations an additional functionality of enable/disable is frequently required to ensure reliable operation under all possible conditions. This can be achieved quite simply with an additional gate circuit, as shown in Figure 1.7, when the latch is not enabled this makes the latch D input equal to its Q output so that the latch state doesn't change when active clock edges arrive. When

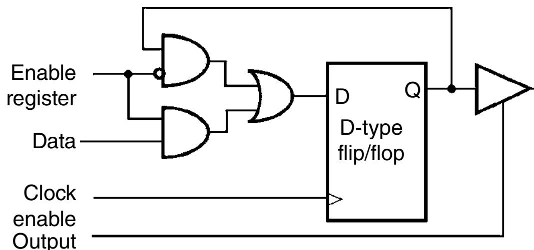


Figure 1.7 Flip/flop with enable functionality

enabled, the data input reaches the latch and the next clock edge sets it to the required state.

Importantly, this avoids the need to disable the latch by interrupting its clock signal, which could be achieved with an additional gate. If used this arrangement unfortunately has the side effect of delaying the clock edge by a few nano-seconds possibly compromising setup and hold times when least expected. The solution shown is much more reliable and predictable so is automatically adopted in most cases.

Note that in this circuit an output enable is also included by using a tristate buffer to improve functionality as this switches the output to the third high-impedance state when disabled. This will be needed, for example if the latch output is connected to a bus line where other circuits take the driving role some of the time. This structure, with or without the output buffer, will form the D-type functional block in many of the following interface circuit structures.

1.3.4 Multi-Bit Registers

For many interface applications multi-bit latches, often referred to as registers, are required to link up with the multi-bit data bus employed in a typical microcomputer design. This is quite simply achieved as illustrated by the 4-bit example, shown in Figure 1.8. Here, the clock and enable lines are taken in common across the structure of D-type latches. This scheme can be extended easily to accommodate as many bits as the application requires and a register block in a more complex diagram will be assumed to take this form internally.

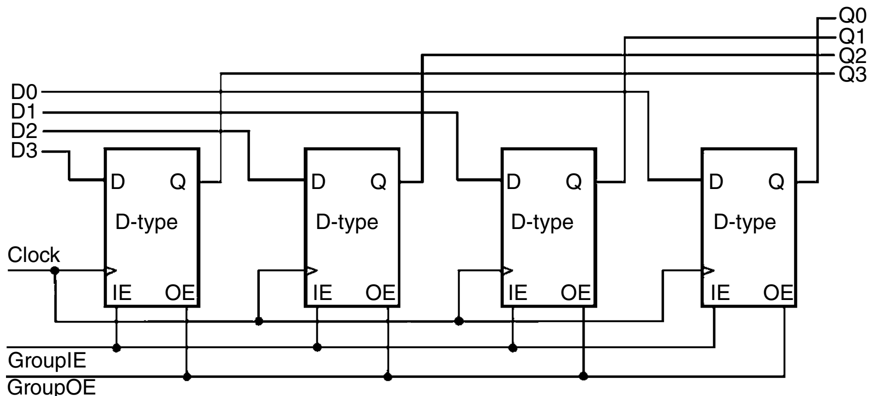


Figure 1.8 Four-bit register with group input enable IE and group output enable OE

In an interface circuit signals to drive the group input enable (IE) and group output enable (OE) inputs need to be determined and this is the main function of a block of logic called the address decode because it effectively places the interface at a unique position within the memory map of the processor. Note that the group output enable may be fixed if the Q outputs are driving external port connections but not if they are connected to a bus where conflicts could arise. It the case of external output ports the tristate buffer will not strictly be required.

1.4 Output Configuration Options

When registers are used in interface circuits the outputs may be required to drive other components and this presents various situations that need to be considered. The normal flip/flop output will drive a one or zero level with equal strength; this will be quite satisfactory if it is only driving a simple input on the connected circuit. The output circuit can be seen to have an active circuit to pull the pin up to a one and an active circuit to pull the pin down to a zero as shown in Figure 1.9. This bidirectional configuration is often referred to as a *push-pull output*.

1.4 1 Open Drain Configuration

An alternative option, which can be created using the output control block in Figure 1.9, is to disable the pull-up driver and replace it with a simple resistor either on-chip or externally; this is known as the open drain configuration.

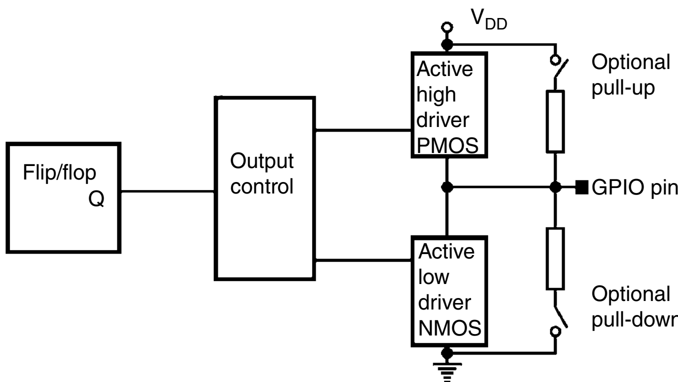


Figure 1.9 Push/pull output driver

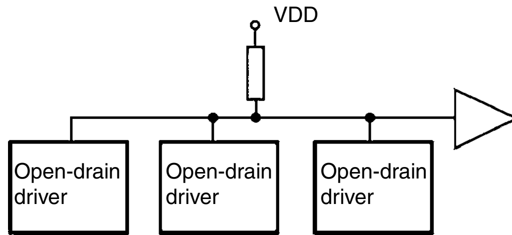


Figure 1.10 Wired-AND connection of multiple outputs

Note that this has a detrimental effect on the maximum speed because the resistor acts as a rather weak pull-up driver but importantly allows wired logic and multisource bus connections to be made and in fact will be used in connection with the I2C serial interface later in Chapter 5. The optional resistors are particularly useful in this situation. The diagram shown in Figure 1.10, with one pull-up resistor, exhibits a wired-AND configuration because the input to the system on the right will only get to a logic one when all the drivers are off, as soon as any one of the drivers is on the input will become zero.

So the designer's choice of push/pull or open-drain and the terminating resistor arrangement will depend on the application envisaged. The natural choice in many situations will be push/pull with no resistors; only in special circumstances will other configurations be needed. Careful analysis of the requirements will be the best way to ensure that a satisfactory configuration is created.

1.5 The Address Decode

As stated previously interface elements are usually required to be located at a particular point, or use a limited range of addresses, within the memory map of the microcomputer's architecture so that they can be accessed reliably by the software instructions. This is achieved through a combinational logic function taking the address bus as its inputs and delivering an enable for the interface register when the defined address is presented by the processor. In the arbitrarily chosen example shown in Figure 1.11, the interface is allocated the address 0x8FA3 on a 16-bit address bus so requires all 16 address bits to present appropriate states to the interconnected group of AND gates for the output enable to become active. This scheme is referred to as *full address decoding* because it takes all individual address lines into account. The circuit presented here is not sufficient for the IE and OE signals required because the state of the control bus is not included so further gate combinations will often be required in a practical situation.

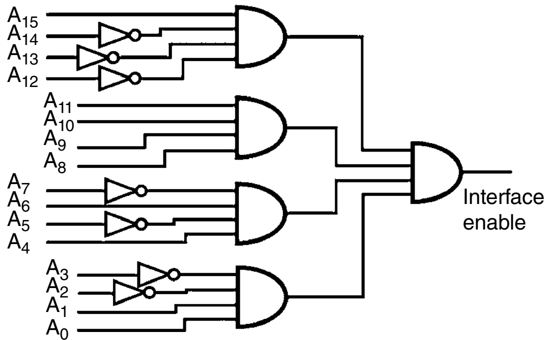


Figure 1.11 Full address decode example

Note that in modern practical processor devices the address bus typically has more bits, actually 32-bits in the ARM processor, and the decode circuit can become rather complex and unwieldy, requiring many gates to achieve the required function. A more economical scheme is widely employed and this is referred to as *partial address decode*. This uses fewer gates but has a serious side effect in that it creates a partitioned memory map with unusable gaps in some sections. With a vast 32-bit address range this, however, will not present a serious issue for most system designs.

1.5.1 Partial Address Decode

In the partial decode scheme a few of the most significant address bits and a few of the least significant address bits are employed as inputs to the decode circuit making the combinational function much simpler. The simplified circuit is shown in Figure 1.12. However, the resulting enable will inevitably be activated by a whole block of addresses within the memory map because the middle bits now don't care. Taking the same example previously but removing the eight middle connections and their gating, that is A4 through A11, means that any addresses between 0x8003 and 0x8FF3 will also activate the interface. So a whole block of nearly 4K address locations are locked out from other uses or otherwise redundant.

In practice the top group of address lines is fully decoded so as to divide the whole memory map into useful blocks; some of these can be allocated to memory and others to interface circuits as required by the system configuration. A simple example of this uses the top three address bits connected to a three to eight line decoder as shown in Figure 1.13. If address lines A15, A14 and A13 are used, eight blocks of 8 Kbytes are created in the memory map Table 1.4.

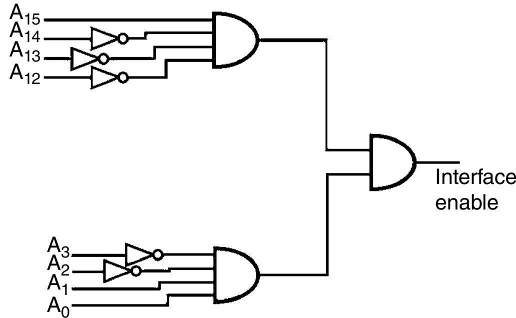


Figure 1.12 Partial address decode

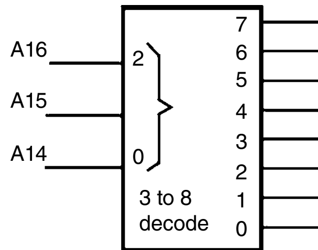


Figure 1.13 Block decode

Table 1.4 Memory map

Base address	Top address
0xe000	0xffff
0xc000	0xdfff
0xa000	0xbfff
0x8000	0x9fff
0x6000	0x7fff
0x4000	0x5fff
0x2000	0x3fff
0X0000	0x1fff

An examination of the memory map for the ARM processor (ARM Cortex 4 STM32F4 Data Sheet Chapter 4) reveals that the top four address bits of its 32-bit address bus are decoded to provide eight 512 Mbyte blocks. Some of these blocks are further subdivided by using lower address bits to activate the built-in memories and I/O functions; in particular most of the I/O functions make use of 1 Kbyte blocks.

1.6 ARM Architecture

The architecture of the STM32F4 ARM based microprocessor system is quite complex in comparison with the simple architecture shown earlier in Figure 1.1 and has introduced many advanced features during its development. It has several types of embedded memory and two high speed internal buses that link with the embedded interface subsystems. The STM32F4 Data sheet has a detailed block diagram in Section 2.2 Figure 2.5. This shows that the 32-bit floating point ARM cortex processor has three independent bus connections, the instruction bus I, the data bus D and the system bus S. These are optimised for particular types of interaction between the processor, the memories and the peripherals. The advanced high-performance bus (AHB) and the advanced peripheral bus (APB) form the main resources that link the memories and peripherals through the 32-bit multi-AHB bus matrix, see Figure 6 in the (ARM Cortex-4 Data Sheet Doc ID 022152 Rev 3).

The 32-bit multi-AHB bus matrix interconnects all the system masters (processor (CPU), DMA subsystems, Ethernet peripherals and USB High Speed peripherals) and the slaves (Flash memory, RAM, *Flexible static memory controller* (FSMC) as well as the AHB and APB that link up the peripherals) and ensures a seamless and efficient operation even when several high-speed peripherals are required to work simultaneously.

The peripherals provide a wide variety of functions but the most obvious in the block diagram are the set of up to 14 timer modules, six universal synchronous or asynchronous serial interfaces (universal synchronous/asynchronous receiver/ transmitter, USART), various proprietary interfaces as well as analogue to digital (AD) and digital to analogue (DA) converters for analogue interface requirements. Most of these will be discussed in detail in later chapters.

The examples given in the text make use of the STM32F4-Discovery board, which is available at low cost from various commercial distributors. The family of boards included provides some useful expansion capabilities that will be used for the case studies in Chapter 7.

1.7 Interface Software Development

The uVision4 IDE, which provides an extensive range of development and debug tools, can be downloaded freely from the ST Microelectronics web site and this package comes with a wide selection of examples set up to illustrate the operation and control of various specific interface subsystems. These will be used extensively throughout the text to show how practical design of interface modules is achieved.

It is important to gain working familiarity with the tools presented through the IDE screen and in overview it will be seen that this is split into several sub-windows. The top left window provides for project management so that the various files needed for a particular implementation can be determined. Note that this area has some alternative functions but more detail will be given when particular examples are discussed. The central window area presents resources for programming and debug that can be employed in the software development. Some useful word processing functions are provided and debug facilities include single-stepping through the code statements, setting break-points to halt execution at a particular point in the code so that the current state can be examined and trace facilities to observe programme sequence and flow. Various optional windows on the right allow the user to examine code action on important aspects of the system memory and the peripheral subsystems. The bottom central window shows the IDE command sequence and will reflect the compiling and liking process steps, for example. Finally, an optional window at the top of the central area shows the assembler language derived from the C code statements, which is very useful in some situations. A screen shot from a simple I/O programme is shown in Figure 1.14.

In summary, the user should be able to make use of the IDE to set up a project, edit, compile and download the designed code and finally use the debug facilities to confirm correct operation of the function. The various steps involved

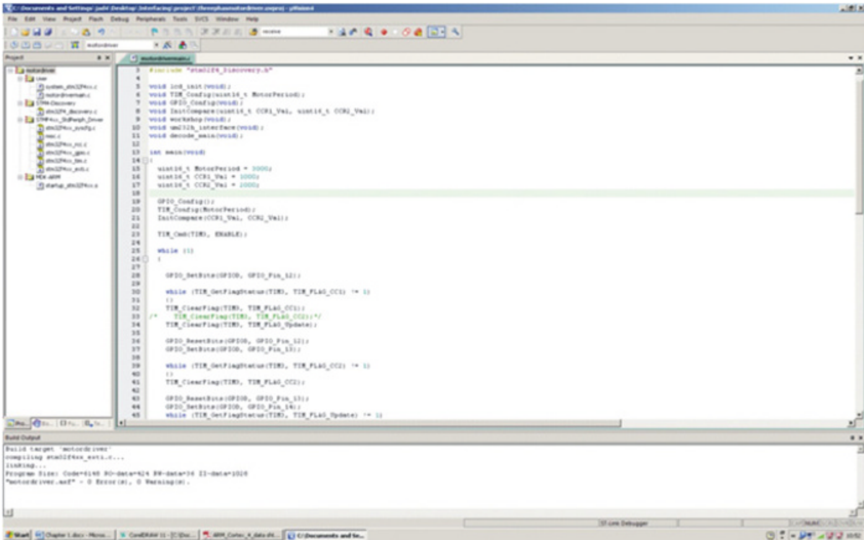


Figure 1.14 Screen-shot for uVision4 IDE

in these processes are described in more detail in Appendix A to help the reader gain familiarity with the IDE and the STM32F4-Discovery board if required.

The examples provided by ST Microelectronics are listed in Appendix B together with a brief summary of their intended target in terms of the peripherals that they address. These examples are written in C and all include the correct project settings so as to make it a simple matter for the user to get a practical implementation to work correctly. In many situations it will be most convenient to start with one of these examples so as to ensure that all the differing requirements are satisfied. Once the various aspects are well understood optimisation of the setup for a particular application will be possible.

The examples are set up to focus on a particular peripheral and the C language is employed to enable the potential user to gain a rapid understanding of the software requirements. It is also essential to use the product data sheet to provide the hardware description of the target peripheral so that the hardware and software aspects can be linked effectively. For example, the block diagram for an I/O port interface can be found in Section 6.3 of the STM32F4 Reference manual together with a tabulation of the port configuration bits, this information is essential to enable the user to control it more efficiently when required.

The C language examples activate and configure the port using a user friendly and well documented approach. However, it should be appreciated that this involves considerable overhead in terms of the individual machine instructions and processing cycles that will be required. Much greater efficiency can be achieved in terms of both processing cycles and memory requirements if Assembler language programming is adopted. This is not as challenging as might be expected because the debug facilities, provided within the Keil IDE, allow the Assembler translation of the C code statements to be inspected, thus allowing the user to rapidly learn how the assembler language statements are constructed.

To illustrate this linkage a simple C program is shown next where an array of consecutive numbers is declared and the program calculates the total summation.

```
main()
{
    int value[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i, sum = 0;

    for(i = 0; i <= 9; i++)
    {
        sum = sum + value[i];
    }
}
```

The assembler language code delivered by the Keil uVision 4 compiler in debug mode is shown next and it can be seen that the program contains many more statements than might be imagined from the brevity of the C statements. The first field shows the memory location, the second field the instruction code, which is 16-bits in most cases, and the third field the assembler mnemonic with its operands. The function of each instruction is fairly easy to understand remembering that the destination is the first operand but more detail can be obtained from the ARM documentation. In summary MOV and MOVS mean move, LDR means load register, ADD, ADDS and SUM perform the obvious calculations, CMP provides compare and B, BL and BLE provide program branches. The original C statements are included as comments to roughly indicate how the two representations are related.

```
0x0800038E   B08B           SUB          sp, sp, #0x2C
    3:       int value[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
0x08000390    2228        MOVS         r2, #0x28
0x08000392    4908        LDR        r1, [pc, #32];
@0x080003B4
0x08000394    A801        ADD        r0, sp, #0x04
0x08000396    F000F821    BL.W       __aeabi_memcpy
(0x080003DC)
     4:        int i, sum = 0;
     5:
0x0800039A     2500        MOVS          r5, #0x00
     6:        for(i = 0; i <= 9; i++)
     7:        {
0x0800039C     2400        MOVS          r4, #0x00
0x0800039E     E004        B          0x080003AA
     8:             sum = sum + value[i];
     9:        }
0x080003A0     A801        ADD      r0, sp, #0x04
0x080003A2     F8500024    LDR      r0, [r0, r4, LSL #2]
0x080003A6     4405        ADD          r5, r5, r0
0x080003A8     1C64        ADDS          r4, r4, #1
0x080003AA      C09        CMP          r4, #0x09
0x080003AC     DDF8        BLE          0x080003A0
    10: }
0x080003AE     2000        MOVS          r0, #0x00
0x080003B0     B00B        ADD          sp, sp, #0x2C
```

Table 1.5 Code interpretation

Instruction address	Interpretation
0800038E	40 bytes reserved on push down stack
08000394	R0 points to first stack location
08000394	Call function to initialise array data values
08000396	Initialise sum to zero
0800039E	Branch to end test
0800039C	Initialise i to zero
0800039E	Branch to test against limit
080003A0	R0 points to first stack location
080003A2	Next value retrieved from stack (at r0 indexed by four times loop count)
080003A6	Add next value to sum
080003A8	Loop counter increment
080003AA	End test, loop counter compared with limit value
080003AC	Branch back when incomplete
080003AE	End of program, r0 returned to 0
080003B0	Stack space given back

The data values are placed on the stack and 40 bytes are reserved for this. Register r4 is used as the loop counter in the for-loop. Register r0 is used as the stack address of the current integer value in some instructions and the actual representation of integer i in others. Register r5 represents the sum variable accumulation. The tabulation Table 1.5 gives some interpretation of the code to assist with comprehension in terms of the required operations.

In fact this code is quite efficient in most respects; in particular the instruction at 080003A2 does a lot of work in calculating the address on the stack of the data value required at each iteration of the program. Note that each value uses 4 bytes. The only small inefficiency comes from the call to a function, which initialises the data values on the stack.

1.7.1 Software Development for Embedded Systems

It is quite unusual to be able to address all the requirements of a system design within a single program module so the IDE presents an optimal way to combine all the resources that will be required. The development resources provided by ST include a library of routines for each of the integrated subsystems. For example, the General Purpose Input and Output (GPIO) functions are in the library stm32f4xx_gpio.c and, for example, this provides routines

for GPIO_SetBits() and GPIO_ResetBits(), which will be examined more extensively in the next chapter. Each of these files includes useful notes on the particular subsystem and the functionality provided. Rather lengthy names are used throughout these library modules to help the user understand their application.

The only complex C programming concept that is used extensively is the use of data structures throughout the peripheral driver package. In each driver a data structure is defined, using the C typedef, which contains all the aspects required in the initialisation of the particular module in question. All the required structures are defined in the drivers so all the user has to do is to assign appropriate values to the structure elements, that is structure_name. element_name=0xf0Ae. In many cases the structure elements have a limited range of possible values so in these situations a series of names are defined to handle the values in a more intelligible way.

Examining the top left hand corner of the IDE screen a small panel showing all of the file names involved in a particular development will be observed, see Figures 1.14 and 1.15 for greater detail. These are grouped together in four sub categories for convenience, user, STM_Discovery, StdPeriphDrivers and MDK_ARM, but can be modified as required. Notice that startup_stm32f4xx.s is always included in the MDK_ARM category to establish various initial system resources like the clock generation and will call the user's main code module. The file stm32f4_discovery.c in the STM_Discovery category provides board specific elements. The file system_stm32f4xx.c resides in the user category with any user designed code modules providing system specific elements. Finally the StdPeriphDrivers category should contain any peripheral drivers relevant to the particular application.

1.8 C Programming Revision

It will be found that the C programming examples used throughout the text and the support package rely heavily on structures and pointers as well as functions so a few notes on these have been included here for reference. If the reader is familiar with C programming conventions, this particular section can be safely ignored.

1.8.1 Arrays

Array names are themselves pointers so can be passed to functions where read and write access will be possible. In this example p[i] reads the message characters from main. The string can be altered if required `p[0] = 'X'` will

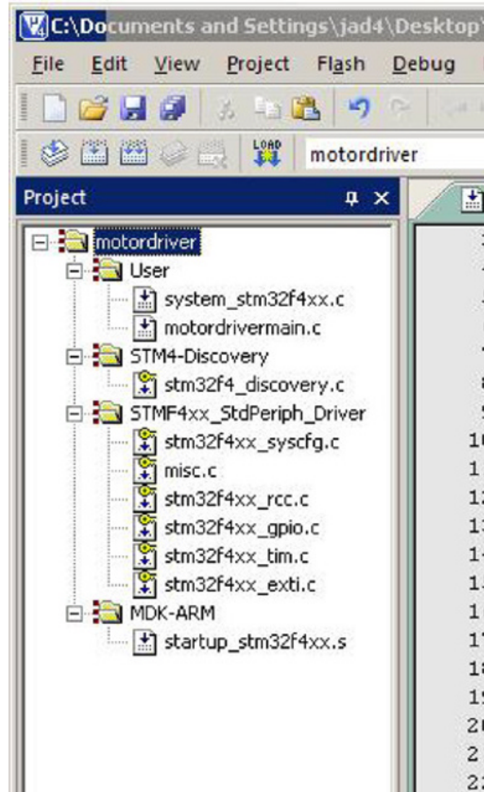


Figure 1.15 IDE management window

change the first character from T to X. Note that the declaration `char *p` can also use the alternative form `char p[]`.

```
int main(void)
{
    int length;
    char message[] = "TEST Message/r/n";
    length = my_length(message);
}

int my_length(char *p)
{
    int i = 0;

    while(p[i] != 0)
```

```
    {
        i++;
    }
    return(i);
}
```

1.8.2 *Structures and typedef*

These definitions allow a very convenient way to set up small data bases that are particularly useful in grouping together all the parameters that need to be initialised in a particular interface module. In this short example the structure is declared in main and the function call passes a pointer to it. The function can access the structure elements using the special `->` shorthand instead of `*(p).first`. If the structure is declared within a function, instead of in main, its pointer can be passed to another function quite easily.

```
typedef struct MY_DATA
{
    int first;
    int second;
} MY_STRUCT;

int main(void)
{
    MY_STRUCT record;

    init_structure(&record);
}

void init_structure(MY_STRUCT *p)
{
    p->first = 100;
    p->second = 200;
}
```

1.8.3 *Header Files*

Header files (such as stm32f4xx_gpio.h) are used extensively within the support package as there is a specific header file for each of the peripherals that define both reference values and data structures. In order to incorporate a

header file the #include statement is used and the header file location is specified. For example:

`#include "my_header.h"` if the file is in a local or specified directory or
`#include <my_header.h>` if the file is in a standard location.

It is usual to have a #include statement for each required file but as the application may involve many different peripherals this would become rather tedious to handle. In order to pick the header files up automatically it is only necessary to declare #include “stm32f4_discovery.h” to cover all possibilities. This file actually contains #include “stm32f4xx.h”, which contains all the references.

Remember that any functions that you create must be declared before the start of main(), for example:

```
void GPIO_setup_pins(void);
```

These declarations can advantageously be part of your own header file, as this is useful particularly when your source code spans several files, otherwise they would have to be declared in each of them.

In summary, remember to declare the structures you want to use either in main() or in your own functions. When access to the data structure is required, such as with a supporting function, you must use a pointer to the structure that has been defined correctly. Header files for the peripherals make their programming more straightforward, allowing realistic names to be associated with raw data values.

1.9 Conclusion

Some essential revision of processor architecture and a number of basic interface design hardware and software issues have been discussed in this chapter. It should be emphasised that familiarity with these aspects is key to establishing techniques that can be used in effective interface design and will also enable the reader to gain the maximum benefit from the following chapters.

The C language will be used extensively to promote rapid understanding of the issues so the most straightforward constructs will be used and the reader will not require knowledge of more than fundamental programming techniques. The linkage between C and assembler has been illustrated by a simple example but although this particular example is comparatively efficient assembler

language short cuts will be included where appropriate because it may help the interface designer to obtain the required efficiency in a demanding application. The uVision compiler and debug facilities always make it possible to examine the assembler code so shortcuts can be seen quite easily from careful analysis.

The next chapter will deal with simple parallel interface designs that can be used to link up with a variety of basic input and output devices. The ARM processor has several interfaces of this type built in so its user has to provide only straightforward hardware circuitry for many practical applications.

References

[1] ST Microelectronics (2012) STM32F405xx (ARM Cortex-4) Data Sheet Doc ID 022152 Rev 3, www.st.com (accessed 20 December 2014).

[2] Wakerly, J.F. (2006) *Digital Design Principles and Practices*; Prentice Hall; ISBN: 978 013 613 9874.

Further Reading

ST Microelectronics (2011) RM0090 (ARM Cortex-4) Reference Manual Doc ID 018909 Rev 1, www.st.com (accessed 20 December 2014).

2

Simple Input and Output Functions

Simple input and output (I/O) operations are a fundamental part of any embedded system where an LED indicator is turned on or a user input button is provided. All processors for embedded applications provide I/O functions that can perform this kind of job and the STM32F4 ARM based processor actually includes 144 pins (16 pins on each of 9 ports) for this kind of application but as will be shown later many are associated with other functions.

2.1 Introduction

This chapter will focus on the design and application of simple I/O techniques using exclusively the basic logic elements outlined in Chapter 1, specifically latches, tri-state gates and address decoding logic. Most importantly it will develop the concept of a programme controlled interface and C language program examples, using pointers to handle interface addresses, will be described. The chapter will also cover the development of these techniques for the extensive range of reconfigurable I/O functions provided by the STM32F4, a typical embedded processor design based on the ARM core. To enable the user to handle their configuration and use conveniently, C programme techniques will be discussed. At first acquaintance, the STM32F4 embedded

Digital Interface Design and Application, First Edition. Jonathan A. Dell.
© 2015 John Wiley & Sons, Ltd. Published 2015 by John Wiley & Sons, Ltd.

processor's interface modules appear extremely complex but approaching their functionality from a simple standpoint will facilitate their effective employment in any particular application. User I/O programming in assembler will be briefly discussed to show areas where it might be effective to employ this approach. The chapter will conclude with a design challenge for the reader based on the STM32F4 Discovery board and making use of the techniques discussed.

2.2 Computer Structure

All general purpose computers and computers for embedded applications rely on I/O interfaces to link up with their particular environment and the actual interface task will span a huge range of alternatives. It is the intention here to begin with very simple interface scenarios to establish a sound basis for the more complex interface subsystems and interface techniques that will be featured in later chapters. So to enable the general structure of an input or output interface to be established it is useful to review again the bus structure of a typical embedded computer design. The important elements of the overall system are the buses that enable communication between all the interdependent modules. These are the address, data and control buses as shown in Figure 2.1. It is important to remember, as explained in Chapter 1, that the data bus in particular is bi-directional and that, probably the most important, function of the control bus is to indicate when the data bus is in input or output mode so that bus conflicts can be eliminated. With these ideas in mind it will be easier to comprehend the important aspects of an interface design.

As also explained in Chapter 1 the address bus determines the physical location of all parts of the system within the memory map and is invariably used to identify and select I/O circuits through their associated address decoding logic. The data bus provides a multi-bit bi-directional pathway between

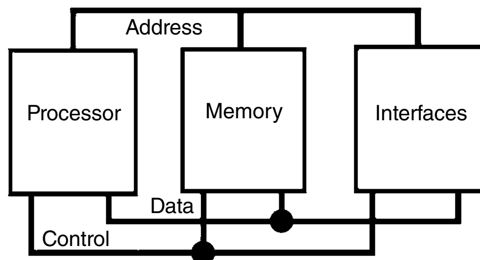


Figure 2.1 Fundamental processor structure

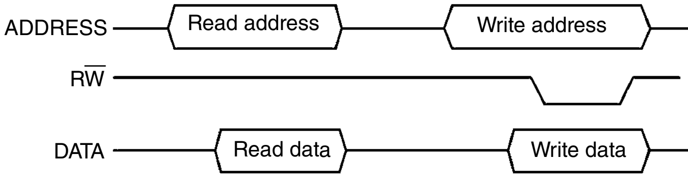


Figure 2.2 Bus timing relationships

the system components and also the I/O circuits although, in practice, these often employ only a few bits depending on the complexity of the particular function required. The most important signal in the control bus is that determining the data bus direction and thus whether an input, or output, interface should be enabled. A simple timing diagram is shown in Figure 2.2 to emphasise the relationships between these signals.

Note that read data becomes available shortly after the read address becomes stable and that the write address and write data must both be stable before the controlling RW signal goes high again.

2.3 Simple Interface Circuit Concepts

Users of the STM32F4 embedded computer are unlikely to actually design an interface module linking to the buses directly but to understand the programmable options available a fairly detailed knowledge of the basics is essential. The basic concepts for input and output interfaces will therefore be presented here because these form the core elements in a wide variety of practical interface designs. Although the circuits presented focus on a single bit interface they can easily be extended to accommodate as many bits as the application requires.

2.3.1 An Output Interface

The situation where the embedded system is required to control an output is probably the most common of all requirements. The simplest output interface that satisfies this requirement uses a basic D-type flip/flop or latch building block element as shown in the diagram in Figure 2.3. In fact this element is used in almost all interface structures so a clear understanding of its operation is very important.

As shown previously, the latch is designed to be edge sensitive and is thus updated with a new information bit from the D input by a positive edge on its

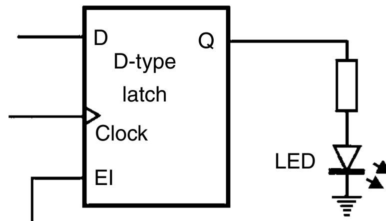


Figure 2.3 Basic latch

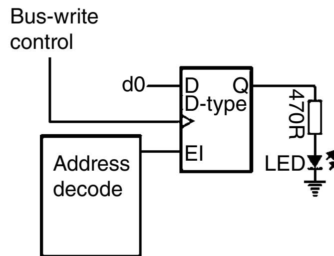


Figure 2.4 A simple output interface

clock input when the input enable (EI) is also active. An output enable as seen in the previous chapter is not required in this simple case.

The latch D input can be connected directly to a particular line in the data bus, d0, for example in this case. This connection will never create a bus conflict situation. The output Q, and possibly $\overline{Q}$ if this is also available, form the signals used to drive physical outputs such as an LED in this simple application. If the input enable signal (EI) is created from the address decode and the bus write direction controls the latch clock it will be updated only when the bus is performing an output correctly and the allocated interface address is simultaneously active. The sketch of a completed output interface circuit using the D-type latch to directly drive an LED is shown in Figure 2.4. The LED is driven through a resistor to limit current to an acceptable value, for reference this is given by the following equation where logic high is V_H and the LED forward voltage is V_{LED}.

$$I = \frac{V_H - V_{LED}}{R} = \frac{3.6 - 1.2}{470} = 5\text{ mA}$$

Once the latch is set by writing a one the LED will be illuminated until the flip/flop is reset once again. Notice that no extra gating is needed in the

bus-write control line as the latch will remain in the same state until address decode enables it to accept a new value.

Note that bus timing will ensure that the D-type is correctly clocked as long a positive edge sensitive design is employed. Although a simple LED has been used in this example there is practically no limit to other possibilities if a power control circuit such as a 'Smart Power' MOSFET is included in the design.

2.3.2 Address Decode for Output

It is of course essential to design the address decode logic so that there is no conflict with other elements of the system. Although not a particularly difficult issue in this case because the circuit cannot compromise other bus activity as it has no output driving the data bus lines. Table 2.1 shows an arbitrary interface assignment using a 16-bit address bus for the sake of discussion. A full or partial address decoding scheme can be employed according to the particular system requirements imposed. In fact, most practical address decoding schemes will employ partial decoding as explained in Chapter 1 to minimise the logical complexity of the combinational decoder circuit, a significant consideration for most designers.

In this case, providing that the I/O demands are minimal, a decoder connecting to the top four address bits and A0 will be sufficient. The circuit in Figure 2.5 will satisfy these requirements.

Table 2.1 Output address allocation

Allocated address	Name	Comment
0x8000	OUTBIT	LED driver

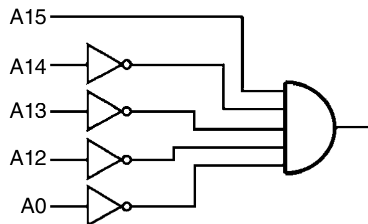


Figure 2.5 Simple decode for output

2.3.3 A Simple Input Interface

The simplest input interface uses a tri-state buffer element as shown in the diagram Figure 2.6. When the output is connected to a bus line the high impedance state is required so that when the interface is disabled the input circuit does not contend with or interrupt other activity on the bus lines.

2.3.4 Address Decode for Input

The tri-state output is connected to a particular line in the data bus, d0, for example in this case. The buffer input can then be connected directly to a simple switch to provide a binary input. The buffer enable is created from the address decode and the bus read direction control thus enabling the buffer only when the bus is performing an input correctly and the allocated address is simultaneously active. A sketch of a typical circuit is shown in Figure 2.7. The address decode, Table 2.2, is again arranged to avoid conflict but it can in fact use the same address as the output interface, given earlier, because the bus-read and bus-write controls are mutually exclusive.

Timing is quite satisfactory because the bus line will be set in the desired state as soon as the correct address is established because the read is high throughout the access cycle.

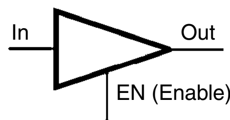


Figure 2.6 A tri-state buffer

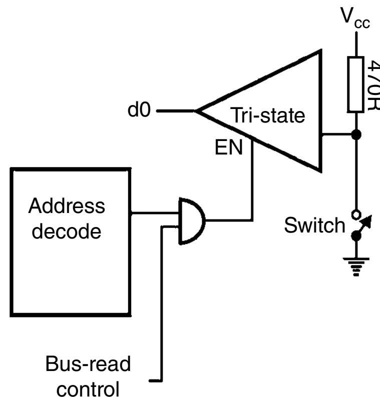


Figure 2.7 A simple input interface circuit

Table 2.2 Input address allocation

Allocated address	Name	Comment
0x8000	INBIT	Switch input

The input is zero when the switch is closed and a pull-up resistor is used to provide logic one when the switch is open. An alternative arrangement employing a pull-down resistor can be used if required.

2.4 Activation of I/O Circuits

In order to activate an input or output through the circuits described previously, a processor instruction involving a suitable 'move' operation will be needed, where data is written to the desired address or data is retrieved from the desired address. In high level languages a simple assignment is all that will be required but in low level languages Load, Store and Move types of operation will be effective as long as the interface address can be set up in a processor register, for example. The interface address is needed whatever language is being used so in the C language the following assignment statement will activate the output latch as long as the correct address is defined in the way suggested. In the C programme the interface address is handled through a pointer variable that must be declared together with the other variables used. The pointer itself is then set to the address of the interface circuits.

2.4.1 Programming an Output

The code that follows implements the actions needed to activate an output interface. This will work whether the output interface is a single bit or multiple bits up to the data bus width; that is 8, 16 or 32 bits. Notice how careful naming is used to help explain what relevance the numbers carry, this can improve the code in terms of readability and maintenance.

```
#define OUTBIT 0x8000;
#define INBIT 0x8000;

int a, *p;      /* declarations */

p = OUTBIT;     /* set up pointer */

*p = a;         /* output assignment */
```

After executing these operations the output port bit will be set according to the contents of the variable ‘a’ and if its least significant bit (LSB) is one, the LED will be illuminated. In any other situation the LED will be extinguished.

2.4.2 Programming an Input

In the case of an input interface reversing the assignment is all that is needed to initiate the data transfer to a program defined variable, providing the address is still correct.

```
a = *p;
```

Note that this assignment will read a whole word (i.e. 8 or 16 bits) when only one bit is actually determined by the input switch, so a logical AND operation on the variable ‘a’ will be required to remove the undefined bits and give the correct zero or one result that reflects the state of the switch.

```
a = a & 1;
```

In a practical application the variable ‘a’ can then be used to determine the function of an ‘if’ statement or the action of a more complex structure as required by the programme design.

```
if (a > 0)
{
   printf("Switch set\n");
}
else {
   printf("Switch not set\n");
}
```

2.5 Universal I/O Circuits

Many practical systems require a programme configurable I/O system that can be set to operate as input or output according the instructions. This can be achieved quite easily by combining the two simple one-directional circuits illustrated previously. Such a combination is shown in Figure 2.8. An extra D-type latch will be required to provide the direction control and this will need its own distinct address to avoid conflict with the I/O access operations.

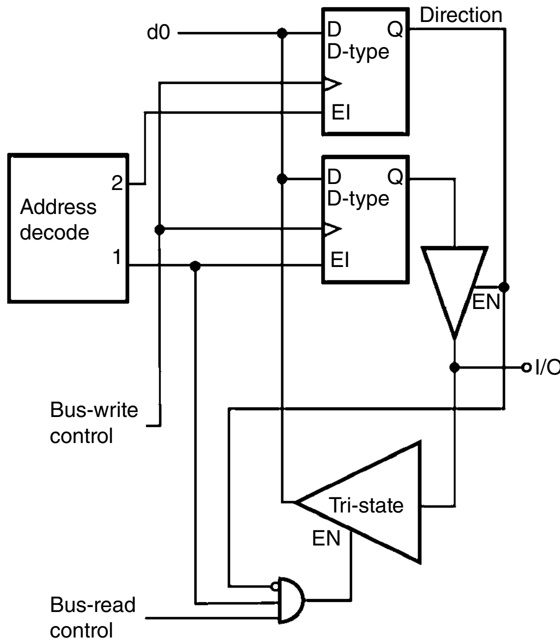


Figure 2.8 Combined I/O

The control provided by this latch is used to enable the tristate buffer in the output path and disable, by virtue of its inversion, the tristate buffer on the input path. The logic determines that input and output operations are mutually exclusive. Input bus conflicts are still avoided because the input buffer is controlled by the bus-read signal as well. The circuit will act as an output when the direction latch is set and as an input when the direction bit is reset. The short delay from the extra gate that combines address decode, direction and bus read will not cause a problem because read actually take place later in the processor cycle.

2.5.1 Combined I/O Address Decode

Address decode in this example will need to identify a new address for the direction control latch avoiding conflict with other system elements of course. Using a 16-bit address bus, for example it can be seen that a consecutive address can be allocated for simplicity as only a few gates will be required in the decode logic design. The same data bus LSB bit d0 is used for both the interface elements in this circuit (see Table 2.3).

Table 2.3 I/O address allocation

Allocated address	Name	Comment
0x8000	OUTBIT	In output mode
0x8000	INBIT	In input mode
0x8001	D_C	Direction, 1 output, 0 input

The controlling programme will need to first determine whether the circuit should operate as input or output, by setting or clearing the direction latch through address 2 (0x8001), then it can execute an appropriate read or write assignment using address 1 (0x8000). The elements of a C programme to select input or output mode is given next, pointers are used as before for both interface addressed assignments.

```
#define D_C 0x8001;
#define INOUT 0x8000;

int a, *p, *dc;      /* declarations */

p = INOUT;           /* set the pointers */
dc = D_C;

*dc = 0;     /* input mode */
a = *p;      /* input assignment */

*dc = 1;     /* output mode */
*p = a;      /* output assignment */
```

Any of the circuits discussed can easily be extended to accommodate multiple input bits or multiple output bits as required, up to the data bus width available in the system. Beyond this further addresses will need to be employed. Although it is unlikely that an interface designer will need to resort to the construction of circuits like these, it is useful to understand the principles behind their functionality as the practical circuits, which are implemented in the majority of programmable modules, have strong similarities with what has been described.

2.6 Practical I/O Circuits

The majority of processors for embedded applications, including the STM32F4 ARM based device, contain reconfigurable I/O blocks that can be set up according to the user's requirements. Although these circuits are quite complex

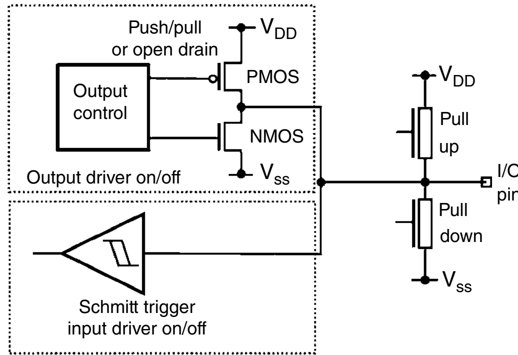


Figure 2.9 GPIO pin circuit

they are all based on the simple techniques described here and the vendors also provide useful example programmes and flexible device drivers to simplify their configuration for particular application requirements.

The diagram in Figure 2.9 shows a simplified circuit of each I/O pin to explain some of the programmable aspects that are available. The output has two modes, in the push/pull mode both PMOS and NMOS devices are activated whereas in open drain mode the PMOS device is disabled so the output is only driven by the NMOS device. The optional pull up and pull down resistors are effective whether the pin is acting as an input or output. These programmable options greatly enhance the functionality of the pins extending the range of external circuit designs that can be accommodated. A good explanation of the push/pull and open drain applications can be found in Wakerly [1].

On the STM32F4 processor each general-purpose 16-bit I/O port has no less than ten 32-bit configuration registers, which give it many alternative modes of operation. It has two 32-bit data registers for input and output and a 32-bit set/reset register for bitwise operations on the outputs when these are needed. A 32-bit locking register freezes the I/O configuration from further changes and two 32-bit alternate function selection registers are used to enable each of the pins to act as inputs or outputs for the various functional blocks like timers and analogue converters that are provided on the STM32F4 processor. Full details can be found in the data sheets and reference manuals for the STM32F4 [2]. Apart from the general I/O configuration the General Purpose Input and Output (GPIO) port also provides the ability for the processor to control the electrical characteristics of individual pins whether in input or output mode through configurable pull-up and pull-down resistors and speed selection options.

Table 2.4 STM32F4 address allocations

Address offset	Register name	Comment (number of bits)
0x00	GPIOx_MODER	Interface mode (2)
0x04	GPIOx_OTYPER	Output type (1)
0x08	GPIOx_OSPEED	Interface speed (2)
0x0C	GPIOx_PUPDR	Pull-up/pull-down (2)
0x10	GPIOx_IDR	Input data
0x14	GPIOx_ODR	Output data
0x18	GPIOx_BSRR	Bit set/reset (2)
0x1C	GPIOx_LCKR	Lock register
0x20	GPIOx_AFRL	Alternate function low (4)
0x24	GPIOx_AFRH	Alternate function high (4)

2.6.1 STM32F4 Address Decoding

The STM32F4 ARM based family of processors use a 32-bit address bus so an extensive address decode block will have been implemented on the chip. In fact the memory map is set up to accommodate quite complex peripheral interfaces where a 1 K block of addresses are allocated to each element. The interface for the 16-bit GPIO port D starts at 0x4002 0C00 and continues to 0x4002 0FFF; however, in this case only the first 10 locations, as listed in Table 2.4, are actually used. The number of bits listed in the table refers to the allocation for each pin. So each I/O pin is controlled by a single bit or possibly a group of bits in the respective controlling register as indicated.

The function of each I/O pin in the interface is independent and can be set as input, output or I/O. Most of the registers have read and write capability so that the application programme can determine the settings made by previous statements. Addresses above this point are not used and full details of the control registers are given in the STM32F4 processor reference manual Section 6.

2.7 A Typical I/O Programme

The programme to set up and control such an interface would leave the user to handle quite a lot of independent pointers so the ST Microelectronics support peripheral driver package provides a library of functions for handling the interface more conveniently. For this general purpose interface the specific library required is ‘stm32f4xx_gpio.c’, which uses the header file ‘stm32f4xx_gpio.h’ and users should ensure that their location can be found by the compiler. This header file defines most of the predefined options for the GPIO port operation.

For example, four possible GPIO modes are defined for its digital and analogue applications:

```
GPIO_Mode_IN
GPIO_Mode_OUT
GPIO_Mode_AF (Alternate function Mode for connections
to various peripherals)
GPIO_Mode_AN (Analogue Mode for AD and DA converter
connections)
```

The two predefined output types are, see Section 2.6 for a brief explanation of these two alternatives:

```
GPIO_OType_PP (push/pull)
GPIO_OType_OD (open drain)
```

There are four speed options; low, medium, fast and high that offer a closer match with design requirements particularly if minimal power consumption is desirable. The high speed mode is generally satisfactory for prototype evaluation when requirements are quite relaxed. The speed options have no relevance to pins configured as inputs.

```
GPIO_Speed_2MHz
GPIO_Speed_25MHz
GPIO_Speed_50MHz
GPIO_Speed_100MHz.
```

Finally, there are three possible pin termination arrangements, see Section 2.6 for a brief explanation of these options. The choice will depend mainly on the requirements of the connected circuit:

```
GPIO_PuPd_NOPULL
GPIO_PuPd_UP
GPIO_PuPd_DOWN.
```

The header file also provides a useful list of the functions implemented within the peripheral driver package and some of the most useful functions are summarised next with the details of their required parameters. The naming shows fairly explicitly what function is performed in each case.

```
/* Functions used to set and use the GPIO */
/* Initialization and Configuration function */
GPIO_Init(GPIOx, &GPIO_InitStruct);

/* GPIO Read and Write functions */
uint8_t GPIO_ReadInputDataBit(GPIOx, GPIO_Pin);
uint16_t GPIO_ReadInputData(GPIOx);
uint8_t GPIO_ReadOutputDataBit(GPIOx, GPIO_Pin);
uint16_t GPIO_ReadOutputData(GPIOx);
GPIO_SetBits(GPIOx, GPIO_Pin);
GPIO_ResetBits(GPIOx, GPIO_Pin);
GPIO_WriteBit(GPIOx, GPIO_Pin, BitAction BitVal);
GPIO_Write(GPIOx, PortVal);
GPIO_ToggleBits(GPIOx, GPIO_Pin);

/* GPIO Alternate functions configuration function */
void GPIO_PinAFConfig(GPIOx, GPIO_PinSource, GPIO_AF);
```

2.7.1 Example GPIO Application

To emphasise the principles and put the discussion into a more definite context a simple example C program, given next, shows how the STM32F4 processor is configured to provide a 4-bit parallel output using four pins on port GPIOD. Additional comments for explanation are given in Table 2.5. These pins are connected to LEDs on the STM32F4 Discovery prototype board so that, by

Table 2.5 Programme linked comments

Note #	Comments
1	typedef declaration for GPIO initialisation structure
2	Connect GPIOB module to the AHB clock so that its registers can be loaded
3	The port pin names are logically ORd together to form the pin specification value
4	Use GPIO_Init() lo load the registers with the setup data
5	Unterminated while loop
6	3 time units delay
7	7 time units delay
8	16 time units delay

using a function providing several seconds delay, the output change sequence can be easily observed with the naked eye.

```
#include "stm32f4_discvery.h"

int main(void)
{
   GPIO_InitTypeDef  GPIO_InitStructure;  /* note 1 */

   /* GPIOD Peripheral clock enable */
   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD,
ENABLE);  /*note 2 */

   /* Configure PD12, PD13, PD14 and PD15 in output
   push/pull mode */
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12 | GPIO_
Pin_13| GPIO_Pin_14| GPIO_Pin_15;     /* note 3 */
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
   GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
   GPIO_Init(GPIOD, &GPIO_InitStructure); /* note 4 */

   while (1)     /* note 5 */
   {
      /* PD12 to be toggled */
      GPIO_SetBits(GPIOD, GPIO_Pin_12);

      Delay(0x3FFFFF); /* Insert delay note 6 */

      /* PD13 to be toggled */
      GPIO_SetBits(GPIOD, GPIO_Pin_13);

      Delay(0x3FFFFF); /* Insert delay */

      /* PD14 to be toggled */
      GPIO_SetBits(GPIOD, GPIO_Pin_14);

      Delay(0x3FFFFF); /* Insert delay */

      /* PD15 to be toggled */
      GPIO_SetBits(GPIOD, GPIO_Pin_15);

      Delay(0x7FFFFF); /* Insert delay note 7*/
```

```
    GPIO_ResetBits(GPIOD, GPIO_Pin_12|GPIO_
    Pin_13|GPIO_Pin_14|GPIO_Pin_15);

        Delay(0xFFFFFF); /* Insert delay note 8*/
    }
}
```

It can be seen that the user's task in putting this code together is greatly simplified by making use of the standard peripheral driver package stm32f4xx_gpio.c as it defines all the useful functions and data structures to initialise the GPIO parameters and manipulate the module effectively as shown earlier. This helps to setup and use the GPIOD controlling registers in a more convenient and user friendly way. The first code statement in main() enables the clock for GPIOD (Table 2.5 note 2), this effectively switches the port on, and is necessary because only a few parts of the system are enabled initially by default. It will not be possible to communicate with the GPIO module before this step is completed. Note that the initialisation and access routines can apply to one or more pins as required. For this example, a standard GPIO configuration for all four pins is made use of. This approach to initialisation using a defined data structure helps to ensure that all the required settings are provided by the user.

The code presented and the peripheral driver package make use of C language functionality to provide procedures such as `GPIO_SetBits()` that automatically make the appropriate address assignments and thus the required changes on the port pins. The intention is to make the programming task for the user as straightforward and comprehensible as possible.

In terms of the actual time sequence the operation of this example will provide a four phase signal on the outputs as depicted in the diagram in Figure 2.10, these signals can be observed with an oscilloscope quite easily if the delays are shortened to multiples of a few milliseconds. The code next shows a very simple delay function that can be used for experiment.

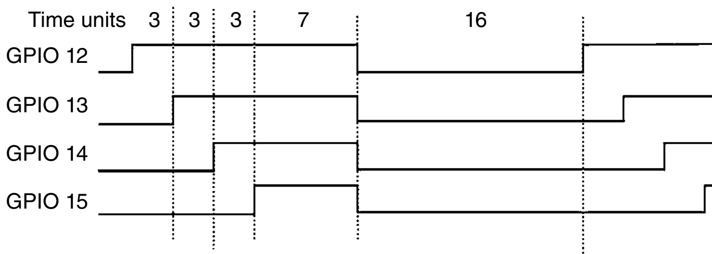


Figure 2.10 Observed code output pattern

```
void Delay(int time)
{
     int i;
     while (time > 0)
     {
          time--;
          i = 100;
          while(i > 0)
          {
               i--;
          }
     }
}
```

2.7.2 A Summary of Alternative I/O Operations

There are many ways to manipulate output pins beyond the set and reset that has been employed in the previous example. For instance the GPIO_Write() function can be used to set all the port bits simultaneously, as illustrated by the statement next, where the variable a provides the pattern of bits required.

```
GPIO_Write(GPIOD, a);
```

Alternatively to configure some port pins as input will require a small change to the port mode in the GPIO_InitStructure as shown here, the assignment for output type becomes irrelevant in this case so does not need to be included.

```
/* Configure PD0, PD1, PD2 and PD3 in input mode */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_
Pin_1 | GPIO_Pin_2 | GPIO_Pin_3;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
  GPIO_Init(GPIOD, &GPIO_InitStructure);
```

Once these changes are made, the data on the four allocated pins can be accessed by the GPIO_ReadInputData() function or individually by the GPIO_ReadInputDataBit() function as illustrated here.

```
a = GPIO_ReadInputData(GPIOD);
a = GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_0);
```

Note that to use these examples on the STM32F4 Discovery board the development system uVision4 from STM Microelectronics should be employed together with various support files provided such as 'startup_stm32f4xx.s', which configures the system clock structure by using the SystemInit() function that is called before branching to the users application main() block. Examples provided within the package give more examples of I/O programming techniques.

2.7.3 Programming I/O in Assembler Language

There will inevitably be some situations where the overhead of C procedures will not be acceptable and a resort to assembly language programming will have to be made. It is clear, however, that port initialisation is only done once so it would generally not be productive to use assembler for this part of the code design. Actual input or output interactions with the pins are more likely to be time critical and thus assembler programming would be most advantageous in this part. Taking an example of the GPIO_SetBits() function this only demands a new value to be written to the bit set/reset register GPIO_BSRR at address offset 0x18, that is 0x4002 0C18 for GPIOD.

An assembler instruction to perform this function can be defined if the GPIOD base address is in register r0 = 0x40020c00 and the bit mask for pin 12 is in register r1 = 0x1000. In this instance the store half word ARM instruction will be all that is needed to set GPIO pin 12 high:

```
strh        r1, [r0, #0x18]
```

Note the use of a half word instruction here because the upper half of this address is used for the reset bits and these should not be affected at this stage. It is evident the assembler code is not self-explanatory so great care has to be taken if it is utilised.

2.8 Suggested Design Challenge

For the Design Challenge readers should attempt to design a programme to implement an interface to a small 16-key matrix keyboard, of the type shown in Figure 2.11 available from Farnell under part number 113-0806 [3]. The

Figure 2.11 16-key matrix keyboard

suggested electrical arrangement for the keyboard involves eight independent connections for the rows and columns, that is four outputs and four inputs. These can be provided by port D on the STM32F4 device, for example as shown in Figure 2.12, but many alternative arrangements are possible. When a key is activated by the user a logic one placed on the related output line will force an input line to logic one through the switch at the crossing point. All four input lines have a pull-down resistor to ensure that the unconnected lines do not float to logic one possibly giving an erroneous key indication. Note that the specification for this key board suggests that a 5 ms de-bounce period should be allowed for reliable key recognition.

The main function of the required programme is to scan the keyboard one column at a time and read the inputs at each stage. A timing diagram for this is shown in Figure 2.13. When an input becomes one the key at the crossing point with the active output is the one that is depressed by the user. It can be assumed, in the first instance, that only one key at a time is depressed but in actual practice ‘rollover’ may occur, when two keys are depressed before one is released, and this should be accommodated by the controlling programme

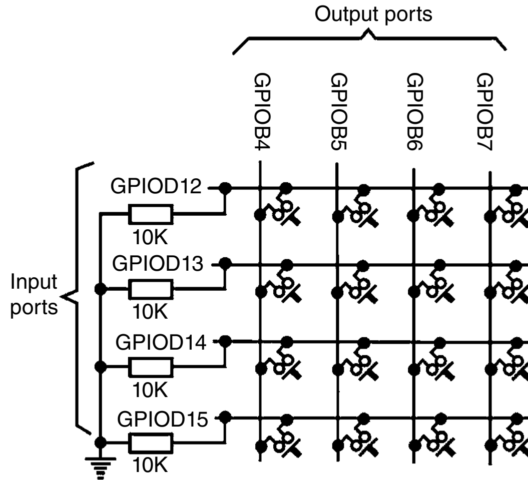


Figure 2.12 Keyboard interface connections

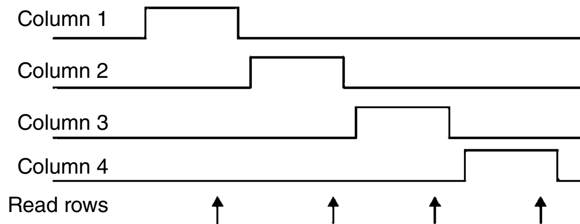


Figure 2.13 Timing diagram for keyboard interface

if possible. Also mechanical key bounce or instability may occur in practice so suitable delays of a few milliseconds should be allowed to elapse before a key signature is accepted as correct. This interface task will ensure a clear understanding of the principles described in this chapter and useful practice in employing the STM development environment. It is suggested that the code is developed in stages so as not to introduce too much complexity all at once.

2.9 Conclusion

Readers should refer to the ST web site [4] for access to the full range of microcontroller development tools and support libraries. Programming expertise and productivity will quickly develop as familiarity with the STM32F4 increases.

This chapter has provided an introduction to input and output interface circuits and the basic concept of their activation through a series of programme statements. The address decoding requirements were described to activate simple interfaces together with the use of pointers in the C language to handle them. The extensive I/O functionality of the STM32F4 processor, determined through its multiple control registers, has been discussed briefly together with its controlling program code.

Program examples are given throughout with a view to providing resources that can be adapted to specific user requirements. All these examples make use of the ST Microelectronics support tools and peripheral drivers, which are freely available together with low cost hardware boards for prototype development.

Although it is recommended that the reader attempt the design challenge independently a possible solution for reference is provided in the Appendix D at the end of the book.

In the next chapter, analogue interfaces, involving specifically AD and DA converter modules, will be described together with C program examples for their practical application.

References

[1] Wakerly, J.F. (2006) *Digital Design Principles and Practices*; Prentice Hall; ISBN: 978 013 613 9874.
[2] ST Microelectronics (2011) STM32F4 Reference Manual. RM009 (ARM Cortex-4) Reference Manual Doc ID 018909 Rev 1, www.st.com (accessed September 2014).
[3] Farnell element14 Data sheets from: http://www.farnell.com following links to STM32F4 Discovery (accessed 20 December 2014).
[4] ST Microelectronics www.st.com (accessed 20 December 2014).

Further Reading

ARM Ltd Keil uVision IDE Integrated Development Environment, www.keil.com/uvision/ide_ov_starting.asp (accesses September 2014).
ST Microelectronics (2012) STM201432F4 Data Sheet. S32F405xx (ARM Cortex-4) Data Sheet Doc ID 022152 Rev 3, www.st.com (accessed September 2014).

3

Timer Subsystems

3.1 Timer Subsystems

The need for timing in I/O applications has already been encountered in Chapter 2 where the de-bounce time for a simple keyboard had to be accommodated. The solution taken for this issue was to simply waste time by locking the computer into a repeated loop. There are two reasons why this is not a good approach one is that the actual time is difficult to predict with any degree of accuracy and the second is that the computer resources are wasted during this delay and could be employed much more productively. A better method to deliver the required time interval is clearly desirable in practical application designs. It turns out that a simple hardware system can deliver an accurately defined time interval but it should be understood that this will not release the computer resources for more productive work unless other techniques are also employed. These will be discussed in Chapter 5.

The de-bounce application does not itself present a particularly demanding timer requirement but in many applications of embedded systems accurate and predictable time intervals are a high priority. In a control system for example, the response to a change in demand must be delivered within a specific time interval or instability might result. So when the embedded application has a real time requirement, measuring time intervals or generating

Digital Interface Design and Application, First Edition. Jonathan A. Dell.
© 2015 John Wiley & Sons, Ltd. Published 2015 by John Wiley & Sons, Ltd.

time-based events, this clearly has to be addressed. Accurate timing is also essential if the embedded application has to manage multiple tasks to accommodate the different activities it is required to perform. This, however, needs other resources to enable the switch between tasks and this aspect will also be discussed later.

Another commonly used technique in embedded systems is the Watchdog where system integrity is monitored at specific time intervals and then the system is reset if a fault is detected. There are many different ways to implement a watchdog system monitor, some are more efficient than others, but a long period timer is usually one of the key elements.

To address the timing requirements of many embedded systems that require operations to be performed in real time this chapter will focus on the timer subsystems that form an important component in many of the commercial microcontrollers that are designed for these applications. A review of the block diagram for the STM32F4 microcontroller (in the ST Microelectronics Data Sheet, Figure 5 [1]) reveals that there are no less than eight independent timer units. Most of these are formed using very similar hardware elements. We will start by reviewing a simple logical structure to furnish this sort of function and go on to examine one of the STM32F4 units in more detail. The chapter will conclude with a short example application illustrating the flexibility of the timer to address a complex requirement.

In summary timers are required because it is impossible for the reasons given previously to create precise time intervals by software routines alone. For example, if an accurate 10 ms interval was required between particular interface functions a timer unit could provide and guarantee this aspect of the specification.

3.2 Basic Timer Configuration

A simple multi-bit logical counter can be used to perform a timer function as long as it is provided with an accurate clock derived from a specific crystal reference oscillator or a signal derived from the actual processor clock. The actual counter circuit will not be discussed here so the reader should refer to Synchronous clock driven counters in Wakerly [2] or other sources. For this simple timer, the availability of a synchronous up going counter with an initial preload capability will be assumed. The block diagram in Figure 3.1 shows how the basic blocks are arranged.

Before starting the n-bit timer operation, the preload register value, which could take on a value anywhere from zero to the maximum counter value

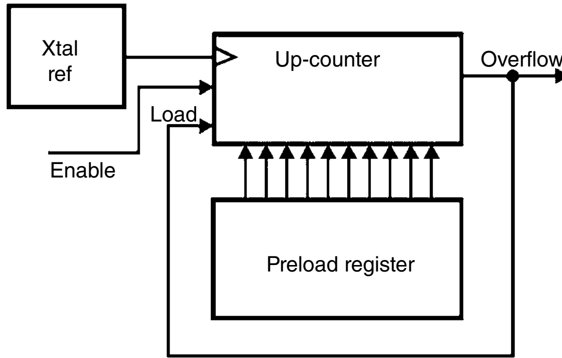


Figure 3.1 A simple timer function

$(2^n - 1)$ is loaded into the counter register. Once counting is actually enabled the upward counting will proceed from the preload value supplied. The key interval is then the number clock cycles required to run through the counter states until it reaches its maximum value, that is $(2^n - \text{Preload}) \times Tc$ the clock period. When the counter reaches its maximum value the overflow output becomes active and on the next cycle the counter starts over again from the preload value. For example, if the number of bits n is 4 and the preload value is eight the period will be $8 \times Tc$ and the count sequence will be 8, 9, 10, 11, 12, 13, 14, 15, 8, 9, 10, 11, 12, 13, 14, 15, 8, 9, 10, and so on. Note that the 8 count value will always appear at the start of each sequence because the preload action takes place during the final state of the sequence (15). If the preload value is changed the current sequence is completed before the new timing action takes effect.

3.3 The STM32F4 Timers

The timers implemented in the STM32F4 look much more involved and have many more additional features than have been discussed in Section 3.1. The actual block diagram is shown in Figure 65 in the ST Microelectronics Reference Manual [3] but its most significant features are summarised in Figure 3.2.

The actual hardware presents various options for the timer clock but in the default mode it is derived from the AHB (Advanced Host Bus) timing. This is calculated from the values set in the clock control module (see Section 2.2 in the STM32F4 Reference Manual [3]) because the input clock (8 MHz on the Discovery board) is multiplied in a Phase-Locked Loop (PLL) as follows.

$$f_{AHB} = 8\text{ MHz} \times \frac{PLLN}{PLLM \times PLLP} = 8\text{ MHz} \times \frac{336}{8 \times 2} = 168\text{ MHz}$$

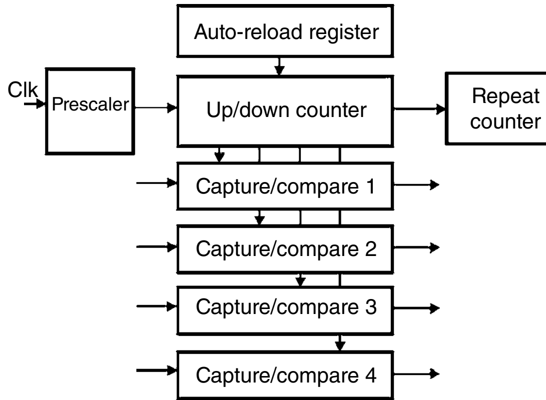


Figure 3.2 The simplified timer architecture

So these default settings make the AHB clock 168 MHz as shown and this is further divided by a prescaler to deliver the APB (Advanced Peripheral Bus) clocks.

$$f_{APB} = \frac{168\ \text{MHz}}{APBPSC}$$

The default setting of the prescaler (*APBPSC*) for APB1 is 4 and for APB2 is 2 so f_{APB1} is 42 MHz and f_{APB2} is 84 MHz. There is, however, a special arrangement for the timer clocks in that the APB clock is multiplied by 2 as long as the *APBPSC* is not 1.

Thus the timer clock becomes 84 MHz for timers attached to APB1 (2, 3, 4, 5, 12, 13 and 14) and 168 MHz for the other modules. The timer clock source is scaled down by its own prescaler value so when it is set to 84 in the APB1 timers for example the resulting timer clock will be 1 MHz.

In these modules the counter operates in a slightly different way to that described earlier. In the up mode the counter increases on each clock cycle until it reaches the value in the auto reload register (ARR) when it is reset to zero to start over again. This is shown schematically in Figure 3.3.

In the down mode the counter decreases on each clock cycle until it reaches zero when it is set to the ARR value to start over again. This is shown schematically in Figure 3.4.

In an alternative centre aligned mode the counter increases to the ARR value and then decreases to zero. This is used in Pulse Width Modulation (PWM) applications in particular and its operation is shown schematically in Figure 3.5.

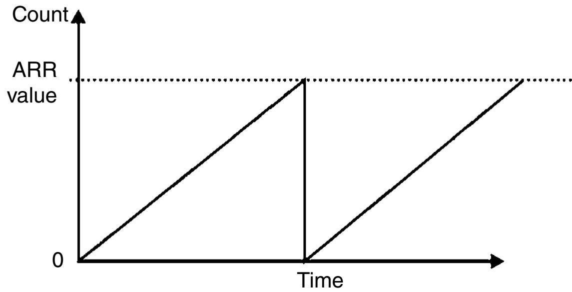


Figure 3.3 Upward count mode

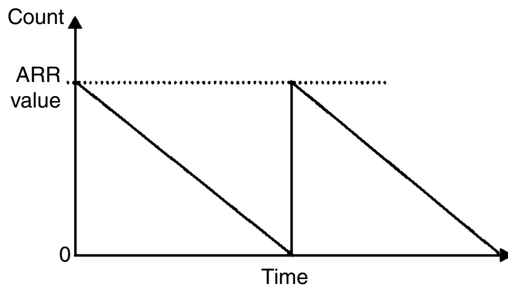


Figure 3.4 Downward count mode

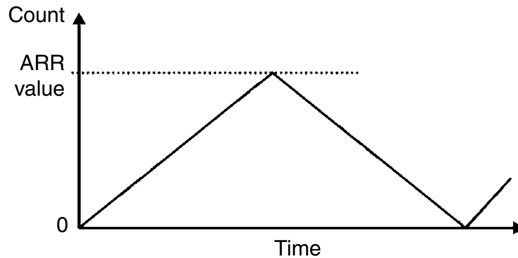


Figure 3.5 Centre aligned mode

The current timer value is constantly compared with the four capture/compare registers allowing different outputs related to the basic time-base to be created or different measurements of an input signal to be made for example.

If the repetition counter is used the timer update event trigger is generated after counting up to the reload value is repeated for the number of times programmed in the repetition counter register. This allows for much longer intervals between event triggers.

3.3.1 The Individual Timers

Some of the timer units have special features for specific applications but Timers 2–5 form a group with common features, all are general-purpose timers consisting of a 16-bit or 32-bit auto-reload counter, driven by a programmable prescaler that gives wide flexibility over the basic period supplied for counting. The connected registers can capture the current value when triggered by an external signal or continuously compare their preset value with the current counter value. They may be used for a variety of purposes, including measuring the pulse lengths of input signals (*input capture mode*) or generating output waveforms (*the output compare mode and the PWM mode*). Derived pulse lengths and waveform periods, from a few microseconds to several milliseconds, can be accommodated using a combination the timer prescaler and the options provided within the Reset and Clock Control (RCC) module resources, which will not be discussed further at this point. The timers are completely independent, and do not share any resources but they can be synchronised together if the application requires.

The main features of this group include 16-bit (TIM3 and TIM4) or 32-bit (TIM2 and TIM5) up counter, down counter or up/down auto-reload counter. Also each has a 16-bit programmable prescaler that can be used to divide the counter reference clock frequency by any factor between 1 and 65 535. Each unit has up to four independent channels for the different modes: Input capture, Output compare, PWM generation (Edge- and Centre-aligned modes are available) and the One-pulse output mode. A synchronisation circuit is included to control the timer reliably with external signals and to interconnect several timers when required.

The timers all include the capability to link up with advanced interface techniques like Interrupts or DMA, which will be discussed in a later chapter, but these can be initiated by requests based on the following four types of counter event:

1. An update event such as counter overflow/underflow or counter initialisation by software or internal or external trigger signals.
2. A trigger event such as counter start, counter stop, counter initialisation or count increment by an internal or external trigger.
3. An input capture event.
4. An output compare event.

Special hardware in these modules provides support for interfaces with incremental quadrature encoders and hall-sensor circuitry used in position

measurement applications. Finally a trigger input can be used as an external clock or cycle-by-cycle management of the counting process.

The other timers are very similar but offer a slightly different feature set aligned with specific applications. The STM32F4 ARM Reference Manual [1] should be consulted for the details of individual modules. For example TMR6 and TMR7 have no physical outputs but may be used as generic timers for time-base generation. They are also specifically used to drive the digital-to-analogue converter (DAC) and in fact, are internally connected to the DAC and are thus able to drive it through their trigger outputs (TRGO). Timers TIM1 and TIM8 have advanced features specifically optimised for advanced PWM applications.

3.4 Programming the STM32F4 Timers

ST Microelectronics provide a standard peripheral driver for the timer units in the file 'stm32f4xx_tim.c', which contains a package of useful support functions and also some useful notes on the timer set up options. All the preconfigured functionality can be found in the header file 'stm32f4xx_tim.h'. For example the counter mode options are:

```
TIM_CounterMode_Up
TIM_CounterMode_Down
TIM_CounterMode_CenterAligned1
TIM_CounterMode_CenterAligned2
TIM_CounterMode_CenterAligned3
```

The first two are self-explanatory but the centre-aligned modes are explained in the data sheet in Section 13 [1]. Basically in these modes the counter counts from 0 to the ARR value, generates a counter overflow event, then counts from the auto-reload value back down to 0 and generates a counter underflow event. Then it restarts counting from 0 again. These modes are particularly useful in the generation of PWM signals as will be shown in a later section.

The timer output compare modes options are:

```
TIM_OCMode_Timing where the output is frozen (match
used for timebase generation)
TIM_OCMode_Active where the output becomes active
on match
TIM_OCMode_Inactive where the output becomes
inactive on match
```

```
TIM_OCMode_Toggle where the output toggles on match
TIM_OCMode_PWM1
TIM_OCMode_PWM2
```

This functionality is used to control an output waveform or indicate when a period of time has elapsed. When a match is found between the capture/compare register and the counter, the output compare function assigns the corresponding output pin to a programmable value defined by the output compare mode and the output polarity. The output pin can keep its level, be set active, be set inactive or can toggle on the match condition. More detail on these modes can be found in the data sheet, particularly the PWM mode, which is used in the included example.

The device driver header file stm32f4xx_tim.h shows all the timer related functions that are supported and these are split into the following principle groups: Time Base management, Output Compare management, Input Compare management and Advanced control, Flag management, and so on. For example, TIM_OC1Init(TIM3, &TIM_OCInitStructure) from the OC management group uses the following data structure elements:

```
TIM_OCMode; Specifies the timer mode.
TIM_OutputState; Specifies the timer Output Compare state
TIM_Pulse; Specifies the pulse value to be loaded
into the CCR
TIM_OCPolarity; Specifies the timer output polarity
```

The code shown next uses Timer 3 to generate three complementary output signals each with different duty cycles. The main steps in the procedure start with configuring the output ports and switching them to their alternate function so as to pick up the timer signals, these settings are in the TIM_PinsConfig() function. Then the next step is to set up the fundamental time base parameters to provide required repeat interval. Next, specify the capture/compare configuration for each channel; this includes the setting of the compare register value. Once this is complete, the code goes into an endless loop but the timer output activity can be observed on the respective pins.

```
/* Timer 3 Functions */
#include "stm32f4_discovery.h"

void TIM_PinsConfig(void);

int main(void)
{
```

```
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
    TIM_OCInitTypeDef  TIM_OCInitStructure;

    TIM_PinConfig();

    /* Time base configuration */
    TIM_TimeBaseStructure.TIM_Prescaler = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_
CounterMode_Up;
    TIM_TimeBaseStructure.TIM_Period = 2000;
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;

    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);

    /* Channel Configuration */
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
    TIM_OCInitStructure.TIM_OutputState = TIM_
OutputState_Enable;
    TIM_OCInitStructure.TIM_Pulse = 0x1f4;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_
OCPolarity_High;

    TIM_OC1Init(TIM3, &TIM_OCInitStructure);

    TIM_OCInitStructure.TIM_Pulse = 0xfa;
    TIM_OC2Init(TIM3, &TIM_OCInitStructure);

    /* TIM3 counter enable */
    TIM_Cmd(TIM3, ENABLE);

    while (1)
    {
    }
}

void TIM_PinsConfig(void)
{
/* Configure I/O Pins for timer output lines*/
    GPIO_InitTypeDef GPIO_InitStructure;

    /* TIM3 clock enable */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

    /* GPIOB clock enable */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
```

```
    /* GPIOB Configuration:  TIM3  CH1 (PB4), CH2
(PB5), CH3 (PB0) and CH4 (PB1) */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_
Pin_1 | GPIO_Pin_4 | GPIO_Pin_5;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP ;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    /* Connect TIM3 pins to AF2 */
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource0, GPIO_
AF_TIM3);
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource1, GPIO_
AF_TIM3);
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource4, GPIO_
AF_TIM3);
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource5, GPIO_
AF_TIM3);
}
```

In this example, the ARR value is 2000, which is set as the timer period and the prescaler is zero. Note that the output compare mode ‘PWM1’ leaves the output active until the point at which the counter reaches the compare value provided. Also using the Discovery board standard settings the timer (TIM3) clock is attached to APB1 and is 84 MHz as shown earlier so the timing of the example waveforms shown in Figure 3.6 will be of the 23.8 µs period.

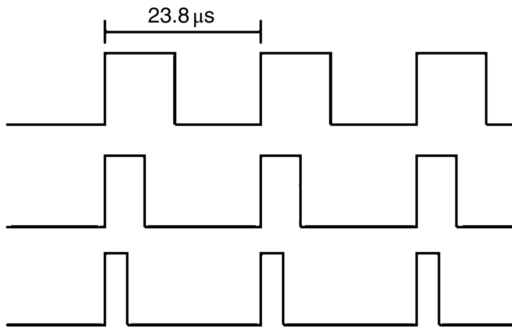


Figure 3.6 Timing diagram for example code

3.5 Timer Triggering

The timers can be used to provide a trigger for other functions within the STM32F4 device. For example TIM3_CH1 could be used to trigger the analogue to digital conversion (ADC) so that it samples an input at precisely 1 ms intervals.

It is also possible to cascade counters so that very long intervals can be created; here the first takes on a master mode while the second acts as a slave. Alternatively a group of timers can be synchronised together when the cascaded trigger is used to initiate reset action on the whole group.

The possible trigger source options that allow the selection of the information to be sent in master mode to slave timers for synchronisation through the TRGO signal are:

```
TIM_TRGOSource_Reset
TIM_TRGOSource_Enable used to synchronize a slave
or create a window for it
TIM_TRGOSource_Update where a master can act as a
prescaler for a slave
TIM_TRGOSource_OC1 pulse mode
TIM_TRGOSource_OC1Ref channel 1 compare match
TIM_TRGOSource_OC2Ref channel 2 compare match
TIM_TRGOSource_OC3Ref channel 3 compare match
TIM_TRGOSource_OC4Ref channel 4 compare match
```

The example function call here shows a typical TRGO setting:

```
TIM_SelectOutputTrigger(TIM3, TIM_TRGOSource_Update);
```

This function will in fact select the generation of a trigger when the counter reaches the ARR value in the upward direction or zero in the downward case.

3.5.1 Setting up the Time-Base

Once the general format of the signals required for the application has been decided on, it will then be essential to determine the time-base that will be needed. In most case this will require suitable values to be assigned to the prescaler (PCS) and the counter ARR. In general it will be better to make the ARR value as large as possible to provide the maximum resolution for time values set in the capture/compare registers.

Let's suppose the application requires a time base of 10 ms, from a knowledge of the counter clock (84 MHz if the timer is driven by APB1, otherwise 168 MHz) it is clear that a total count ratio will be $(10e^{-3} \times 84e6) = 840000$ (or 1 680 000 for a timer on APB2). This is much too big for the 16-bit counter itself so a prescale value of $(840\,000/2^{16}) = 13$ (or 25 for an APB2 case), using the nearest whole number, will be needed. The ARR value will then be (840 000/13) 64615 = 0xfc67, just fitting in the 16-bit counter. Working back shows that a period of 9.9999 ms will be achieved. In fact, if we select, say, 14 (or 28 in the APB2 case) for the prescale then a similar calculation shows that 60 000 will be required in the ARR and this will achieve a period much closer to 10 ms. No significant change to the timer resolution will result. It is frequently beneficial to try a few different alternatives to see which gives the most convenient result.

3.5.2 Using the Timer for an Input Measurement

In some applications it will be required to measure the period or time of arrival of an input signal so the following example will show how this can be achieved. It will be essential as a first step to establish the overall time base. Using TIM3 the channel one input is on GPIOB pin 4 and the channel configuration must determine the channel selected, the edge polarity, the capture mode, the prescaler value, if required, and the filter parameters, if needed. Figure 3.7 shows a simplified circuit of the input channel that delivers the activation to the capture register.

The options listed here are available to set up the input compare configuration.

```
TIM_Channel, that is one of the four possible
compare channels
TIM_ICPolarity either Rising, Falling or Both Edges
TIM_ICSelection, that is the signal source, direct
timer input, indirect timer input or input from a
slave counter
TIM_ICPrescaler the possible ratios are limited to
1, 2, 4 or 8
TIM_ICFilter this can take any value between 0 and
16 (0xf) and is used to determine the minimum
length of input to become an acceptable trigger.
```

Once the required configuration is determined the values for these parameters are assembled into the control register bits by the function TIM_ICInit();

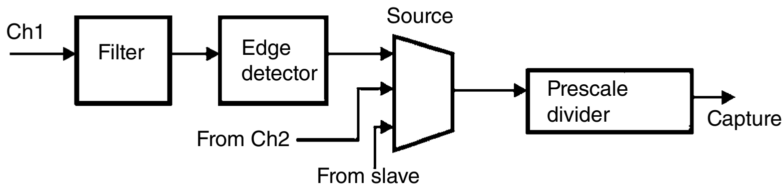


Figure 3.7 Capture/compare input circuit

```
TIM_ICInit(TIM4, &TIM_ICInitStructure);
```

The code modules shown here implement a typical configuration as a basis for further development.

```
#include "stm32f4xx_discovery.h"

TIM_ICInitTypeDef  TIM_ICInitStructure;

void TIM_Config(void);

int main(void)
{

   TIM_ICInitStructure.TIM_Channel = TIM_Channel_1;
   TIM_ICInitStructure.TIM_ICPolarity = TIM_
ICPolarity_Rising;
   TIM_ICInitStructure.TIM_ICSelection = TIM_
ICSelection_DirectTI;
   TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;
   TIM_ICInitStructure.TIM_ICFilter = 0x0;

   TIM_ICInit(TIM3, &TIM_ICInitStructure);

   /* Select the TIM3 Input Trigger: TI1FP1 */
   TIM_SelectInputTrigger(TIM3, TIM_TS_TI1FP1);

   /* TIM enable counter */
   TIM_Cmd(TIM3, ENABLE);

   /* Enable the CC1 Interrupt Request */
   TIM_ITConfig(TIM3, TIM_IT_CC1, ENABLE);

   while (1);
}
```

```
void TIM_Config(void)
{
   GPIO_InitTypeDef GPIO_InitStructure;
   NVIC_InitTypeDef NVIC_InitStructure;

/* TIM3 clock enable */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

/* GPIOB clock enable */
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);

   /* TIM3 chennel1 configuration: PB.04 */
   GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_4;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
   GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NoPull ;
   GPIO_Init(GPIOB, &GPIO_InitStructure);

   /* Connect TIM pin to AF2 */
   GPIO_PinAFConfig(GPIOB, GPIO_PinSource4, GPIO_
AF_TIM3);
}
```

This code includes a statement to set up an interrupt based on CCR1 but as will be shown later other code modules are actually needed to set this up.

Note that it is also possible to measure the frequency and the duty cycle of the input signal in two different capture registers if the timer is set up in the PWM mode. In this case both capture circuits are connected to the same input but one acts as a master and the other as a slave. There is an example of this in the STM32F4 application library provided.

3.6 Basic Timers

The STM32F4 also contains two simplified timer units (TIM6 and TIM7) that do not contain any capture/compare register logic and have no connections through physical pins. They do provide a 16-bit count register and ARR and a prescaler. The block diagram in Figure 3.8 shows the structure of these simplified modules.

The interface to these modules is entirely through software commands but they can also be used to trigger the DAC or generate interrupts and DMA requests at precisely determined intervals. When a software interface is

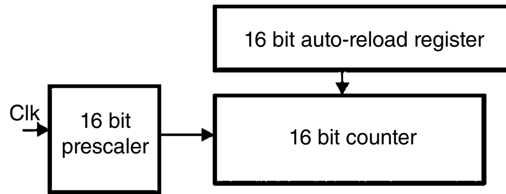


Figure 3.8 Simplified timers

employed the status register flags (e.g. TIM_FLAG_Update) can be used in a hand-shake driven procedure as illustrated in the following code. The timer in this example is set for 0.1 ms (the prescaler clock is 84 MHz as these timers are on APB1) so the output bit that has been set up (GPIOD pin 12) should toggle on each millisecond. Note the observed time will be slightly longer because of the handshake function but this can be corrected if the prescale value is adjusted. A corrected value of 7788 was found to give a very exact toggle period on the Discovery board.

```
#include "stm32f4xx_Discovery.h";

void GPIOD_Init(void);
void TIM6_Init();
void TIM6_Wait(uint16_t t_period);

int main(void)
{
   GPIOD_Init();
   TIM6_Init();

   while(1)
   {
      TIM6_Wait(10);
      GPIO_ToggleBits(GPIOD, GPIO_Pin_12);
   }
}

void TIM6_Init(uint16_t t_period)
{
   TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;

   /*TIM6 Clock Enable */
   RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM6, ENABLE);
   /* Time Base Configuration */
   TIM_TimeBaseStructure.TIM_Period = t_period;
```

```
    TIM_TimeBaseStructure.TIM_Prescaler = 8400;
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_
CounterMode_Up;

    TIM_TimeBaseInit(TIM6, &TIM_TimeBaseStructure);

    /* Prescaler Configuration */
    TIM_PrescalerConfig(TIM6, 8400, TIM_PSCReloadMode_
Immediate);

    TIM_Cmd(TIM6, ENABLE);
}

/* time given in tenths of a millisecond */
void TIM6_Wait(uint16_t t_period)
{
    /* set period in ARR */
    TIM_SetAutoreload(TIM6, t_period);
    TIM_SetCounter(TIM6, 0);
    while (TIM_GetFlagStatus(TIM6, TIM_FLAG_Update) != 1)
    {}
    TIM_ClearFlag(TIM6, TIM_FLAG_Update);
}

void GPIOD_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    /* PD12 */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;

    GPIO_Init(GPIOD, &GPIO_InitStructure);
}
```

In applications where the timer is used for other functions, such as DAC synchronisation, the set up will not have to encounter the additional overhead of handshaking so timing will be more accurately defined by the ARR and prescale values set up.

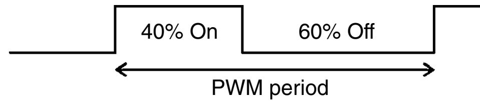


Figure 3.9 A PWM signal set at 40%

3.7 PWM Applications

Many systems involving power control or motor speed control in particular rely on a PWM scheme where the on to off ratio of a digital signal needs to vary. The diagram in Figure 3.9 shows a PWM signal set to about 40% of the whole period.

The sample program in this section uses PWM to control the brightness of an LED but could also be used to control the speed of a DC motor if a suitable interface was constructed. The basic PWM period is set to 100 Hz by defining the ARR and the on percentage can vary from 0 to 100% by setting the value in CCR1. One hundred steps in the brightness are determined by the for loop parameters and there is a lengthy delay between each iteration so that the changes are more visible to an observer.

```
#include "stm32ff_discovery.h"

#define SECOND 0xffff

void PWM_PinsConfig(void);
void PWM_TIMConfig(uin16_t base_period);
void PWM_SetPeriod(uint16_t period);
void PWM_SetFraction(uint16_t fraction);
void wait(int time);

int main(void)
{
   int time, on_time, i;

   PWM_PinsConfig();

   PWM_TIMConfig(time);

   PWM_SetPeriod(time);
   PWM_SetFraction(on_time);

   while(1)
   {
      for (i = 5; i < 95; i++)
```

```
        {
            on_time = i;
            PWM_SetFraction(on_time);
            wait(SECOND);
        }
    }
}

/* TIM4 PWM on Ch1 GPIOd12 */
void PWM_PinsConfig(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;

    GPIO_Init(GPIOD, &GPIO_InitStructure);

    GPIO_AFConfig(GPIOD, GPIO_PinSource12, TIM4_CH1);
    GPIOD->AFR[1] = 0x0200;
}

/* PWM period in ARR and fraction in CCR1 */
void PWM_TIMConfig(uin16_t base_period)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    TIM_OCInitTypeDef TIM_OCInitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);

    /* Time Base Configuration */
    TIM_TimeBaseStructure.TIM_Period = base_period;
    TIM_TimeBaseStructure.TIM_Prescaler = 8400;
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_
CounterMode_Up;

    TIM_TimeBaseInit(TIM4, &TIM_TimeBaseStructure);

    /* Channel Configuration */
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
```

```
    TIM_OCInitStructure.TIM_OutputState = TIM_
OutputState_Enable;
    TIM_OCInitStructure.TIM_Pulse = 0x1f4;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_
OCPolarity_High;

    TIM_OC1Init(TIM4, &TIM_OCInitStructure);

    /* Prescaler Configuration */
    TIM_PrescalerConfig(TIM4, 8400, TIM_PSCReloadMode_
Immediate);

    TIM_Cmd(TIM4, ENABLE);
}

void PWM_SetPeriod(uint16_t period)
{
    TIM_SetAutoreload(TIM4, period);
}

void PWM_SetFraction(uint16_t fraction)
{
    TIM_SetCompare1(TIM4, fraction);
}
```

This example program code has been arranged to link up with one of the LEDs installed on the STM32F4 Discovery board so that the effect of the code can be easily observed. The PWM output varies from 5 to 95% so the LED brightness will vary from dark to bright on a slowly changing basis. The pulse width modulation can also be shown on an oscilloscope attached to GPIOD Pin 12.

3.8 Programming Challenge

For this application, a control program is required for a miniature servo actuator such as those used in radio controlled models; the Hitec HS422, which can be obtained from servoshop.co.uk, is a good example. This requires a PWM signal with a period of 20 ms and the Hitec specifications show that when the on period is 0.9 ms the actuator setting is 0°, 1.5 ms on gives 90° and 2.1 ms on gives 180°. Although the variation of pulse width is fairly small, it can easily be defined by the compare value provided in the timer. The timing diagram in Figure 3.10 shows roughly what is required.

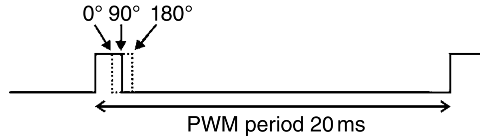


Figure 3.10 Servo control with PWM

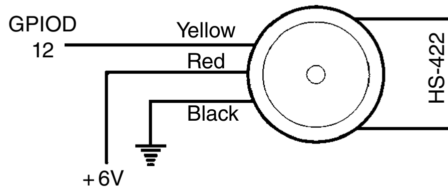


Figure 3.11 Servo controller circuit diagram

The program should be set up to run through the full range of positions slowly and then return slowly back to the origin. This operation can be observed on an oscilloscope but if the real actuator is connected it will require a separate power source as it represents too much load for the USB driven supply on the Discovery board. A diagram of the hardware configuration is shown in Figure 3.11.

3.9 Conclusion

All the timer modules provide an extensive range of facilities that can adapt to many different applications. Although several example applications have been examined in this chapter, as mentioned in several places the STM32F4 Discovery development support package includes other examples that can form the basis for accommodating new system requirements. A careful study of the STM32F4 reference manual is also recommended to show all the options that have not been covered specifically in this chapter.

For a new application a careful sketch of the requirements will quickly reveal the most appropriate timer mode and settings required. It is frequently beneficial to evaluate alternative settings to determine those that give the best results in terms of accurate timing required in a real time environment.

Subsequent chapters will use timers in various roles, for example, in connection with ADC and DAC converters because these systems frequently need precise synchronisation. Also different programming modes of operation including interrupts and DMA will frequently be triggered by timer activity so

the availability these modules is well justified and a good working knowledge of them will be invaluable.

References

[1] ST Microelectronics (2012) STM201432F4 Data Sheet. S32F405xx (ARM Cortex-4) Data Sheet Doc ID 022152 Rev 3, www.st.com (accessed September 2014).
[2] Wakerly, J.F. (2006) *Digital Design Principles and Practices*, Prentice Hall; ISBN: 978 013 613 9874.
[3] ST Microelectronics (2011) RM0090 (ARM Cortex-4) STM32F4 Reference Manual Doc ID 018909 Rev 1, www.st.com (accessed 20 December 2014).

4

Analogue Interface Subsystems

4.1 Analogue Interfaces

A quick review will reveal that many embedded system designs require some form of analogue interface because of the analogue nature of the real world in which such systems have to operate. This chapter will focus on analogue to digital and digital to analogue interface techniques because they will be required whenever an analogue signal has to be created or monitored in a digital computer algorithm.

Taking temperature measurement as just one analogue example, it is evident that the basic analogue measurement must be converted to digital form so that the computer can manipulate its digital representation in a control algorithm. Similarly, the drive for a DC servo will be calculated as a digital value in the control algorithm but will have to take analogue form to activate the motor control voltage. This chapter will present, in particular, the Analogue to Digital converter (ADC) and the Digital to Analogue converter (DAC) through a discussion of the conversion techniques employed in some popular designs. The ADC converter actually presents a considerable design challenge so there are a wide range of different solutions to be found in commercial practice. Also, as the rate of conversion is a key factor in many applications, the key aspects of performance for these subsystems will be summarised. The chapter

Digital Interface Design and Application, First Edition. Jonathan A. Dell.
© 2015 John Wiley & Sons, Ltd. Published 2015 by John Wiley & Sons, Ltd.

will include a description of simple sampling applications, for signal acquisition, in a variety of practical situations.

Although C code will be employed extensively to promote readability and understanding the linked assembler code will be explained in detail to enable the design of highly efficient interfaces where time considerations have a high priority.

4.2 Digital to Analogue

The digital to analogue conversion process will be presented initially because it is comparatively straightforward and it also forms an inherent part of one of the more popular ADC designs. This conversion process requires in principle a set of binary weighted resistors or a binary weighted current distribution network and a corresponding set of switches that are activated by the latched binary input from the data bus that is stored in the converter's input register.

The binary weighted resistor network is usually implemented by an R/2R network connected to provide binary weighted currents. These are summed at the input node of an op-amp as shown in the diagram Figure 4.1. This type of network is used because it can be fabricated reliably in production. A simpler binary weighted resistor network would have a huge range of values that are difficult to define accurately.

The current in the MSB resistor is simply $\frac{V_{REF}}{2R}$ and this will produce an output at V_{OUT} equal to $\frac{V_{REF}}{2}$. At point C, the two parallel resistors will share the current flow equally whether the switch is in the left or right position. A similar current sharing happens at point B and point A. The currents in the resistors below points A, B and C are $\frac{V_{REF}}{4R}$, $\frac{V_{REF}}{8R}$ and $\frac{V_{REF}}{16R}$, respectively. Thus, the overall transfer function is given by:

$$V_{OUT} = R \cdot \left(b_3 \cdot \frac{V_{REF}}{2R} + b_2 \cdot \frac{V_{REF}}{4R} + b_1 \cdot \frac{V_{REF}}{8R} + b_0 \cdot \frac{V_{REF}}{16R} \right)$$

where b_n represents the binary value, b_3 is the MSB (most significant bit) and b_0 is the least significant bit (LSB). This network can be easily extended to accommodate as many more bits as required and accuracy is maintained because the limited range of resistor values can be fabricated on an integrated circuit successfully.

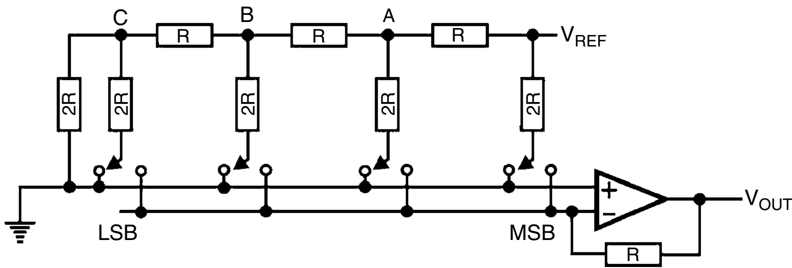


Figure 4.1 Binary weighted DAC network

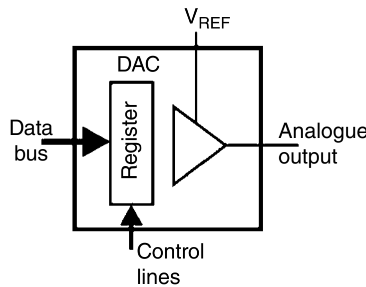


Figure 4.2 DAC block diagram

Detail of the actual circuit development is not important but an explanation can be found in reference books [1] if required. A simplified block diagram as shown in Figure 4.2 will suffice to explain the operation of the overall converter at this point and highlight the important interface connections. Note that the control lines must be derived logically from address decode and the bus write direction control signal.

Most DAC implementations use a linear encoding (i.e. $V_0 = V_{REF} \times \frac{D}{2^n}$) where D represents the binary value and n the number of bits. All the transitions in its conversion characteristic have an equal size, as shown in the 3-bit example in Figure 4.3. Apart from the obviously equal steps it should also be noted that the maximum output is one step lower than the reference or full scale value. Non-linear encoding based on a mathematical relationship like sine or cosine is used in some specialised applications and a different arrangement of resistors is used in this case.

The most significant performance characteristics are the reference voltage V_{REF}, because this limits the maximum possible output, the number of bits quantisation, because this determines the resolution, and the conversion time or its inverse the rate because this must match the application requirements. In

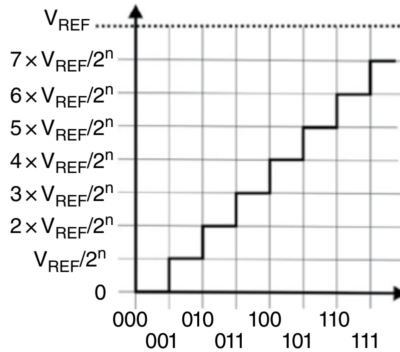


Figure 4.3 A 3-bit DAC transfer characteristic

a system with an audio output, for example a resolution of 16-bits and a rate of 44.1 kHz will be required to achieve the quality of performance provided by an audio CD.

4.2.1 The STM32F4 DAC

There are two DACs in the STM32F4 device, which can be used independently or in synchronism as required, these have 12-bit resolution and a conversion time of 6 μs for large value changes or a maximum rate of 1 M samples per second for small changes. The block diagram in the ARM Reference Manual [2], Figure 48, is much more complex than the simple diagram in Figure 4.2 reflecting the fact that it has a number of alternative modes of operation such as triangular wave generation and direct memory access (DMA), which can both relieve the processor from software servicing requirements. Also conversion can be triggered from an external signal if required. The reference is usually set to the processor supply voltage (Vdd), typically 3.3 V on the Discovery board, but can be provided externally if the application requires it.

4.3 Analogue to Digital Conversion

There is no simple way to make a conversion in this direction so a complex subsystem is used. There is no perfect solution for this conversion direction so a wide range of architectures have been used by different designers. In a simple form of ADC the output from a DAC is compared with the unknown analogue input and the digital value is adjusted until a match is achieved. One control strategy starts with all the bits cleared and the MSB set to one, this makes the

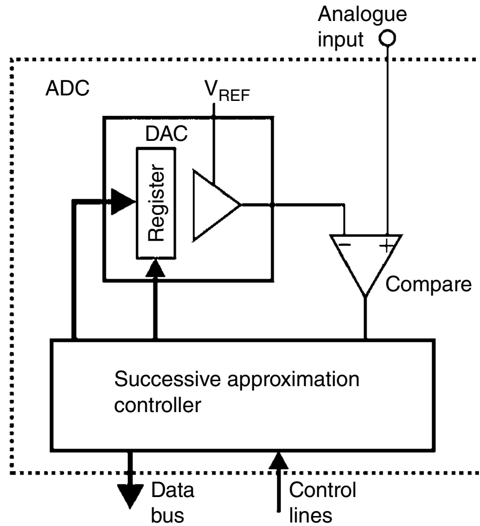


Figure 4.4 Simple ADC subsystem design

DAC output half full scale ($V_{FS}/2$). If the compare shows a greater than relationship then the MSB is needed, otherwise the MSB is reset to zero. The bit next to MSB ($V_{FS}/4$) is set and again if the compare shows greater than this bit will be needed, otherwise it is reset. This procedure is repeated for each of the bits in turn until the LSB is reached. This strategy is known as the successive approximation or binary search converter and is shown schematically in the block diagram in Figure 4.4. This form can be quite effective because it requires the minimum number of clock cycles to perform all the bit decisions, working down from the MSB to the LSB. Practical performance can typically achieve a 3 μs conversion time with this type of architecture. Note that output data is only available when all decisions are completed so careful control arrangements are needed to manage the conversion interface.

4.3.1 Sampling

To allow the converter subsystem described previously to operate correctly the analogue input must not change by more than one LSB until the successive approximation process has been completed otherwise early bit choices will become incorrect. From a knowledge of the conversion time (T_C) the maximum rate of change at the input can be determined and must not exceed a value of $V_{FS}/2^nT_C$. A typical 14-bit ADC with a conversion time of 6 μs and 3.3 V reference

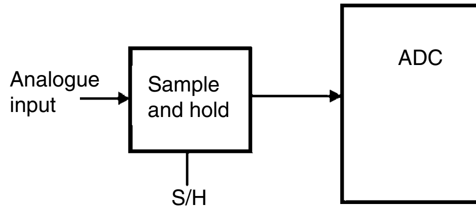


Figure 4.5 Sample and hold

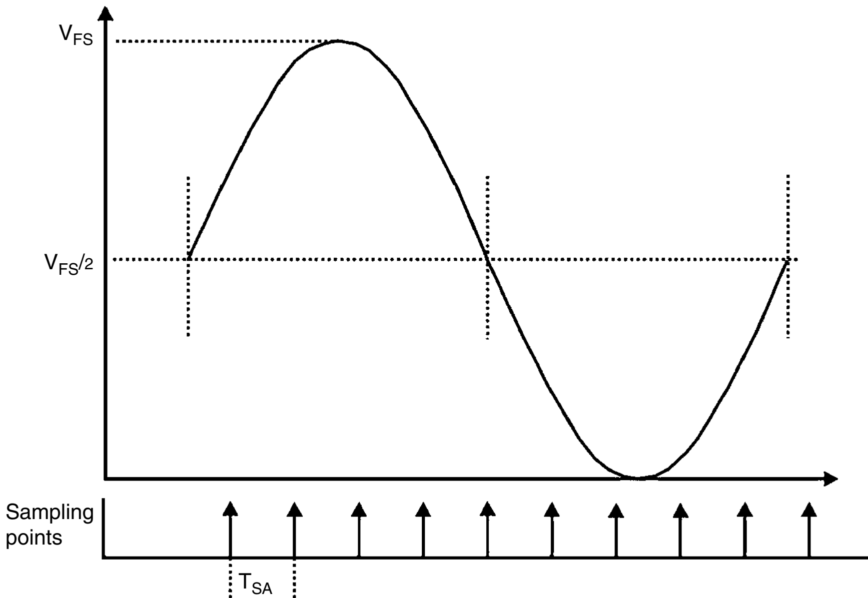


Figure 4.6 Sampling a sine wave

gives a maximum rate of 33 V/s, rather a low value. This stability requirement cannot be achieved easily other than by using a special sample and hold function as shown in Figure 4.5. It is assumed that the sampling process T_S is short compared with the hold T_H, which is usually matched to or exceeds the conversion time (T_C). The overall performance will be significantly improved by using this element.

If a simple sine-wave input, as shown in Figure 4.6, is considered the sampling process can be explained and the maximum input frequency at the converter input can be determined. If we consider a sine wave matched to the full scale converter range it can be seen that the rate reaches a maximum at the zero crossing point. The mathematical gradient at this point is $2\pi f V_{FS}$ and this

must be less than or equal the sample maximum rate of input change, that is $2\pi fV_{FS} \leq V_{FS}/2^n T_S$ so the maximum input frequency $f_{MAX} = \frac{1}{2^n \pi T_S}$ if we assume a maximum of 1 LSB change in the sample time (T_S). This, however, gives a very optimistic value and it is frequently found in practice that the performance is governed by the Nyquist sampling criterion because this determines that there must be at least two samples per cycle so $f_{MAX} \leq \frac{2}{T_{SA}}$ where $T_{SA} = T_C + T_S$ With a 6 µs conversion time and short sampling time the maximum input frequency will be more than 300 kHz, a much more useful performance.

4.3.2 Switched Capacitor Converter

Some types of ADC do not require a distinct sample and hold element because they effectively perform the sampling process internally. The implementation available commercially is known as the switched capacitor converter and its basic operation is described using the diagram in Figure 4.7. The conversion process requires four phases, three to set up the capacitor array and the fourth to assess the individual bits by cyclically testing each in turn.

The conversion process still requires careful management because the resulting value is not available until all bit decisions have been made.

In phase 1 the switch S1 is closed making the inverting node become zero and all the bit switches are to the left as shown so that all capacitors are discharged to zero.

In phase 2 switch S1 remains closed, S2 selects Vin and the bit switches are all set to the right. The inverting node is still zero so all capacitors are charged up to Vin.

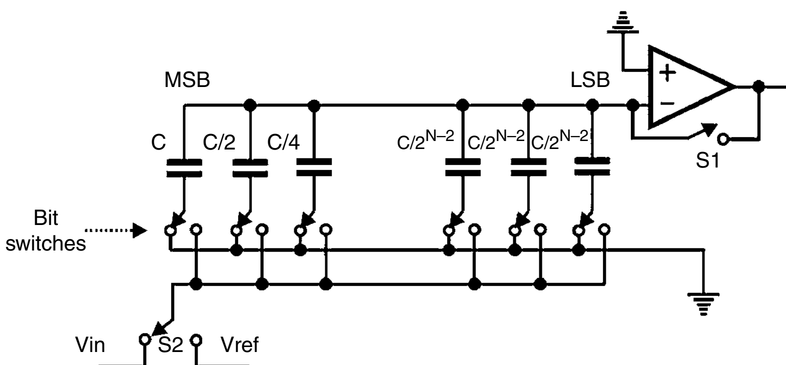


Figure 4.7 Switched capacitor converter

In phase 3 switch S1 is opened and the amplifier forms a comparator, releasing the inverting node. The bit switches are all switched to the left so that the negative node voltage becomes −Vin.

In phase 4 S1 is still open, S2 selects Vref and the cyclic conversion process takes place. First the MSB bit is switched to the right selecting Vref. The sum of the capacitors in the array to the right of the MSB is exactly C so this forms a 1:1 charge divider. The inverting node voltage is then (−Vin + Vref/2). Thus, if Vin is greater than Vref/2 the node voltage remains negative and the high compare output shows that the converter MSB bit is one otherwise the MSB is zero and the MSB switch is reset to zero. The next to MSB bit is switched to Vref. This forms a divider so that the inverting node voltage becomes (−Vin + Vref/2 + Vref/4) if the MSB is one or (−Vin + Vref/4) if the MSB is zero. In either case if Vin is greater the next bit is one otherwise it is zero and the bit switch is reset.

This process sounds quite complex but can be performed quickly. The main advantage of this architecture is that fabrication techniques allow capacitors to be defined more precisely than resistors so high ratios required can be more easily achieved.

4.3.3 The Software Interface

The software interface to an ADC is a little more complex than the diagram might suggest because the conversion process has to be allowed to complete its successive approximation or other conversion algorithm and thus the output data is not available until this point is reached. This timing is conveniently shown in a simple timing diagram as shown in Figure 4.8.

To manage this interface the processor must implement a handshake to determine when conversion is complete so that the new data can be collected when it is definitely ready. A simple while loop structure in C can achieve this hand-shake, assuming the availability of a read function, is shown here.

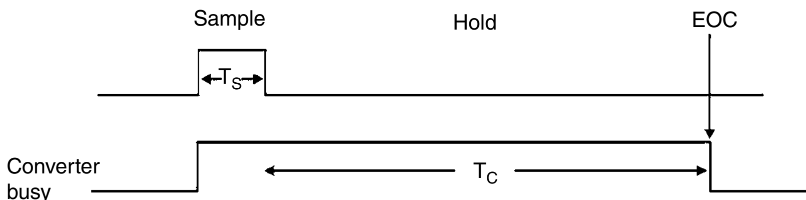


Figure 4.8 ADC interface timing

```
busy = read(busy_flag);
while (busy)
{
   busy = read(busy_flag);
}
read(ADC_data);
```

The while loop will continually read the busy flag and only exit when conversion is done. Notice that this is quite wasteful of processor time because it is effectively idle for a long period. Alternative methods to achieve more efficient interface management, such as interrupts or DMA, are available and these will be discussed later.

4.3.4 The STM32F4 ADC

There are three ADCs and their block diagram is shown in the ARM Reference Manual Figure 28 [2], this is much more complex than the basic converter shown in Figure 4.2 and includes a sample and hold system. The 12-bit ADC is a successive approximation converter with a built-in programmable sampling system. It has up to 19 selectable input channels allowing it to measure signals from 16 external sources, 2 internal sources and the V_{BAT} battery monitor in stand-alone applications. The analogue conversion of the channels can be performed using various user selected modes, single shot, continuous, scan, for automatic conversion of a group of channels, or discontinuous, for a short group of conversions, for example. The ADC resolution can be configured as 12-bit, 10-bit, 8-bit or 6-bit, to achieve optimum levels of performance, and the result is stored into a left or right-aligned 16-bit data register as required. The converters can be operated in various dual and triple modes to allow near simultaneous measurements when required by the application.

Various methods of converter synchronisation with external signals and timers are available and the special analogue watchdog feature allows the controlling application software to detect when the input voltage goes beyond the user-defined, higher or lower threshold. Automatic data transfer through DMA can be selected to optimise the level of service support in the application software and interrupt generation signalling the end of conversion and some other key situations are implemented.

The following examples illustrate the basic operation of the converter and the more advanced techniques are discussed in later chapters.

4.4 Software Control of DAC

Using the Keil peripherals package stm32f4xx_dac.c file provides firmware functions to manage the following functionalities of the DAC peripheral: DAC channels configuration, conversion trigger, output buffer, data format and DMA management, as well as interface management flags and interrupts. Appendix III gives a summary of the functions available and the physical pin connections are given in Table 4.1.

The example next shows DAC initialisation and control using a C implementation and the support functions provided by Keil. The infinite while loop generates a saw-tooth waveform.

```
#include "stm32f4_Discovery.h"
main()
{
    int i;
    uint16_t Data;

    /* Preconfiguration before using DAC & GPIO */
    GPIO_InitTypeDef  GPIO_InitStructure;
    DAC_InitTypeDef  DAC_InitStructure;

    /* Enable write access to DAC registers */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);

    /* Configure DAC_OUTx (DAC_OUT1: PA4) in analogue
mode.*/
    /* Enable the GPIO AHB clock using */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);

    /* DAC channel 1 (DAC_OUT1 = PA.4) configuration */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;

    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* DAC channel1 Configuration */
    DAC_InitStructure.DAC_Trigger = DAC_Trigger_None;
    DAC_InitStructure.DAC_WaveGeneration = DAC_
WaveGeneration_None;
    DAC_InitStructure.DAC_OutputBuffer = DAC_
OutputBuffer_Enable;
```

```
    DAC_Init(DAC_Channel_1, &DAC_InitStructure);

    /* Enable the DAC channel */
    DAC_Cmd(DAC_Channel_1, ENABLE);

    while (1)
    {
        for(i = 0; i < 4000; i++)
        {
            Data = i;
            DAC_SetChannel1Data(DAC_Align_12b_R, Data);
        }
    }
}
```

4.4.1 Waveform Generation

The DAC can be used in an alternative mode using the automatic triangle generation facility and this is shown in the next example. A sketch of the triangle wave is shown in Figure 4.9 for reference. The amplitude can be the full scale range of the DAC or any fraction of it. There are various ways to determine the period and some of these will be discussed.

A similar initialisation to that used in the previous example is required but in addition some form of activation or triggering will be needed to increment

Table 4.1 DAC output allocation

	OUT
DAC1	GPIOA4
DAC2	GPIOA5

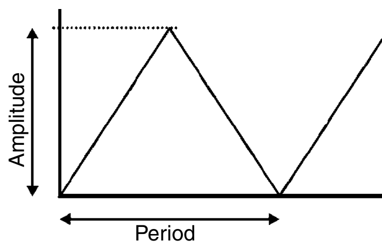


Figure 4.9 Triangle waveform

the current value as the generation of the triangle proceeds. In this case the DAC register value is changed as soon as the effecting statement is executed:

```
/* DAC channel1 Configuration */
   DAC_InitStructure.DAC_Trigger = DAC_Trigger_
Software;
   DAC_InitStructure.DAC_WaveGeneration = DAC_
WaveGeneration_Triangle;
   DAC_InitStructure.DAC_LFSRUnmask_
TriangleAmplitude = DAC_TriangleAmplitude_1023;
   DAC_InitStructure.DAC_OutputBuffer = DAC_
OutputBuffer_Enable;
   DAC_Init(DAC_Channel_1, &DAC_InitStructure);

   DAC_SoftwareTriggerCmd(DAC_Channel_1, ENABLE);
```

Note that the triangle wave amplitude is chosen from the limited range of values available, that is $\left(2^n - 1\right)$ where n is between 1 and 12.

4.4.2 Waveform Timing

The previous example will deliver a new sample value on every clock edge (APB1 42 MHz) but in many cases the rate will need to be controlled according to user requirements. This can be achieved quite conveniently by using a trigger from one of the timers as shown in the diagram Figure 4.10.

In the following example the timer TIM6 prescaler is set for a 10 µs clock interval and its auto reload register (ARR) is set to 2. This is used to synchronize the conversion through its trigger output (TRGO) and is accomplished by reassigning the DAC_Trigger to DAC_Trigger_T6_TRGO as shown in the following code. It is also possible to use an external trigger through a physical pin if required in the particular application. In this example TIM6 triggers the

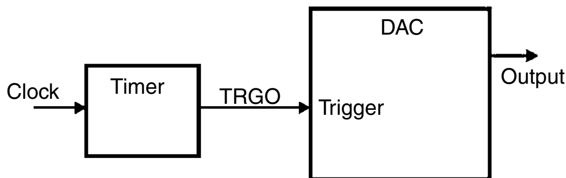


Figure 4.10 Timer triggering of DAC

DAC every 30 μs and as the triangle amplitude is 63 steps a half-period of just under 2 ms will be observed.

```
#include "stm32f4_Discovery.h"

void init_dac(void);
void init_timer(void);

int main(void)
{
   init_timer();
   init_dac();

   /* Enable the DAC channel */
   DAC_Cmd(DAC_Channel_1, ENABLE);

   while(1);
   {
   }
}

void init_dac(void)
{
/* DAC channel1 Configuration */

   GPIO_InitTypeDef  GPIO_InitStructure;
   DAC_InitTypeDef  DAC_InitStructure;

/* Enabled write access to DAC registers */
   RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);

/* Configure DAC_OUTx (DAC_OUT1: PA4) in analogue mode.*/
/* Enable the GPIO AHB clock using */
   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);

/* DAC channel 1 (DAC_OUT1 = PA.4) configuration */
   GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_4;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;

   GPIO_Init(GPIOA, &GPIO_InitStructure);

   DAC_InitStructure.DAC_Trigger = DAC_Trigger_T6_
TRGO;
   DAC_InitStructure.DAC_WaveGeneration = DAC_
WaveGeneration_Triangle;
```

```
    DAC_InitStructure.DAC_LFSRUnmask_
TriangleAmplitude = DAC_TriangleAmplitude_63;
    DAC_InitStructure.DAC_OutputBuffer = DAC_
OutputBuffer_Enable;

    DAC_Init(DAC_Channel_1, &DAC_InitStructure);
}

void init_timer(void)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;

    /*TIM6 Clock Enable */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM6, ENABLE);
    /* Time Base Configuration  84MHz / ((ARR + 1) * PSC)*/
    /* PrescalerValue = (uint16_t) ((SystemCoreClock / 2) /
28000000) -1;*/
    TIM_TimeBaseStructure.TIM_Period = 2;
    TIM_TimeBaseStructure.TIM_Prescaler = 840;
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_
CounterMode_Up;

    TIM_TimeBaseInit(TIM6, &TIM_TimeBaseStructure);

    /* Prescaler Configuration */
    TIM_PrescalerConfig(TIM6, 840, TIM_PSCReloadMode_
Immediate);

    TIM_SelectOutputTrigger(TIM6, TIM_TRGOSource_Update);

    TIM_Cmd(TIM6, ENABLE);
}
```

4.4.3 DAC Using DMA

When more complex waveforms are required, the new sample value can be calculated in software at each step or pre-calculated and placed in a table of consecutive values. When this is set up the DAC data can be supplied directly through DMA, this will be discussed more extensively in Chapter 6 but a simple example is given here as a taster. The DMA method is much more efficient because it only requires processor intervention during the setup phase and can be optimised to deliver the required waveform at the appropriate sample rate for the required application.

The main issues to take account of in the code are defining the address in memory from which the data will be accessible and the address of the DAC holding register (DHR), which is the destination of the DMA transfer. The Keil debug mode reveals all the DAC addresses, if this subunit is inspected it will be found that DHR is 0x40007408, and if the data is created as a constant array C can determine its address by using the '&' sign. The waveform timing between samples is determined entirely by the DAC trigger from TIM6 so in this example a frequency of 100 Hz was chosen so if there are to be n samples the sample rate s is (100 × n) and the TIM6 ARR must be set to $\frac{84 \times 10^6}{s}$ when the TIM6 prescaler is zero, that is $\frac{84 \times 10^6}{100 \times 32} = 26250$ when there are 32 samples. Note that the DMA mode is set to circular so that the waveform is automatically repeated.

```
#include "stm32f4_discovery.h"
#include "stm32f4xx_dma.h"

void dma_setup(void);
void init_timer(void);

#define DAC_DHR12R1_ADDRESS    0x40007408

const uint16_t Sine12bit[32] = {
   2047, 2447, 2831, 3185, 3498, 3750, 3939, 4056,
4095, 4056,
   3939, 3750, 3495, 3185, 2831, 2447, 2047, 1647,
1263, 909,
   599, 344, 155, 38, 0, 38, 155, 344, 599, 909,
1263, 1647};

int main(void)
{
   dma_setup();
   init_timer();

/*DMA through Stream 5 Channel 7*/
   DAC_DMACmd(DAC_Channel_1, ENABLE);

   while(1)
   {
   }
}
```

```
void dma_setup(void)
{
   DAC_InitTypeDef  DAC_InitStructure;
   DMA_InitTypeDef DMA_InitStructure;

   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1, ENABLE);
   RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);

/* DAC channel1 Configuration */
   DAC_InitStructure.DAC_Trigger = DAC_Trigger_T6_TRGO;
   DAC_InitStructure.DAC_WaveGeneration = DAC_
WaveGeneration_None;
   DAC_InitStructure.DAC_OutputBuffer = DAC_
OutputBuffer_Enable;
   DAC_Init(DAC_Channel_1, &DAC_InitStructure);

/* DMA1_Stream5 channel7 configuration */
   DMA_DeInit(DMA1_Stream5);
   DMA_InitStructure.DMA_Channel = DMA_Channel_7;
   DMA_InitStructure.DMA_PeripheralBaseAddr =
(uint32_t)DAC_DHR12R1_ADDRESS;
   DMA_InitStructure.DMA_Memory0BaseAddr =
(uint32_t)&Sine12bit;
   DMA_InitStructure.DMA_DIR = DMA_DIR_
MemoryToPeripheral;
   DMA_InitStructure.DMA_BufferSize = 32;
   DMA_InitStructure.DMA_PeripheralInc = DMA_
PeripheralInc_Disable;
   DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_
Enable;
   DMA_InitStructure.DMA_PeripheralDataSize =
DMA_PeripheralDataSize_HalfWord;
   DMA_InitStructure.DMA_MemoryDataSize = DMA_
MemoryDataSize_HalfWord;
   DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
   DMA_InitStructure.DMA_Priority = DMA_Priority_
High;
   DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_
Disable;
   DMA_InitStructure.DMA_FIFOThreshold = DMA_
FIFOThreshold_HalfFull;
```

```
    DMA_InitStructure.DMA_MemoryBurst = DMA_
MemoryBurst_Single;
    DMA_InitStructure.DMA_PeripheralBurst = DMA_
PeripheralBurst_Single;

    DMA_Init(DMA1_Stream5, &DMA_InitStructure);

/* Enable DMA1_Stream5 */
    DMA_Cmd(DMA1_Stream5, ENABLE);

/* Enable DAC Channel 1 */
    DAC_Cmd(DAC_Channel_1, ENABLE);

/* Enable DMA for DAC Channel1 */
    DAC_DMACmd(DAC_Channel_1, ENABLE);
}

void init_timer(void)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;

    /*TIM6 Clock Enable */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM6,
ENABLE);
    /* Time Base Configuration */
    /* PrescalerValue = (uint16_t) ((SystemCoreClock / 2) /
28000000) -1;*/
    TIM_TimeBaseStructure.TIM_Period = 0xff;
    TIM_TimeBaseStructure.TIM_Prescaler = 0;
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_
CounterMode_Up;

    TIM_TimeBaseInit(TIM6, &TIM_TimeBaseStructure);

    /* Prescaler Configuration */
    TIM_PrescalerConfig(TIM6, 0, TIM_PSCReloadMode_
Immediate);
    /* TRGO Source selection */
    TIM_SelectOutputTrigger(TIM6, TIM_TRGOSource_
Update);

    TIM_Cmd(TIM6, ENABLE);
}
```

4.5 Software Control of ADC

Using the STM32F4 peripherals package stm32f4xx_adc.c file provides firmware functions to manage the following functionalities of the ADC peripheral: Initialisation and Configuration (in addition to ADC multi-mode selection), Analogue Watchdog configuration, Temperature Sensor, Vrefint (Voltage Reference internal) and VBAT management, Regular Channels Configuration, Regular Channels DMA Configuration, Injected Channels Configuration, as well as Interrupts and Flags management. There are no less than 20 registers controlling various elements of the converter and Appendix C gives a summary of the functions available. The physical pin connections are summarised in Table 4.2, where it will be seen that some pins are shared between some of the converters.

The programs here illustrate the set up for a simple converter configuration and a handshake interface control based on the end-of-conversion flag (EOC). Note that the processor will wait until this event occurs so there is no chance of an erroneous value being retrieved from the converter data register.

```
int main(void)
{
   ADC_InitTypeDef       ADC_InitStructure;
   ADC_CommonInitTypeDef ADC_CommonInitStructure;
   GPIO_InitTypeDef      GPIO_InitStructure;

   int value;

   /* Enable ADC2, and GPIO clocks */
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC2, ENABLE);

   /* 2. ADC pins configuration */

   /* Configure ADC2 Channel 12 pin as analog input */
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
   GPIO_Init(GPIOC, &GPIO_InitStructure);

   /* 3. Configure the ADC Prescaler, conversion
resolution and data */
   /* alignment using the ADC_Init() function. */

   /* ADC Common Init */
   ADC_CommonInitStructure.ADC_Mode = ADC_Mode_
Independent;
```

```
    ADC_CommonInitStructure.ADC_Prescaler = ADC_
Prescaler_Div2;
    ADC_CommonInitStructure.ADC_DMAAccessMode =
ADC_DMAAccessMode_Disabled;
    ADC_CommonInitStructure.ADC_TwoSamplingDelay =
ADC_TwoSamplingDelay_5Cycles;
    ADC_CommonInit(&ADC_CommonInitStructure);

    /* ADC3 Init */
    ADC_InitStructure.ADC_Resolution = ADC_
Resolution_12b;
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
    ADC_InitStructure.ADC_ExternalTrigConvEdge =
ADC_ExternalTrigConvEdge_None;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStructure.ADC_NbrOfConversion = 1;
    ADC_Init(ADC3, &ADC_InitStructure);

    /* 4. Activate the ADC peripheral using ADC_
Cmd() function. */
    ADC_Cmd(ADC2, ENABLE);

    /* Start ADC2 Software Conversion */
    ADC_SoftwareStartConv(ADC2);

    while (ADC_GetFlagStatus(ADC2, ADC_FLAG_EOC) != 1)
    {
    }
    value = ADC_GetConversionValue(ADC2);
}
```

Table 4.2 ADC input allocations

	IN0	**IN1**	**IN2**	**IN3**	**IN4**	**IN5**	**IN6**	**IN7**
ADC1	A0	A1	A2	A3	A4	A5	A6	A7
ADC2	A0	A1	A2	A3	A4	A5	A6	A7
ADC3	A0	A1	A2	A3	F6	F7	F8	F9
	IN8	**IN9**	**IN10**	**IN11**	**IN12**	**In13**	**IN14**	**IN15**
ADC1	B0	B1	C0	C1	C2	C3	C4	C5
ADC2	B0	B1	C0	C1	C2	C3	C4	C5
ADC3	F10	F3	C0	C1	C2	C3	F4	F5

This procedure looks quite complex mainly because there are three discrete elements that must be dealt with in the set up phase, firstly the ADC analogue input pins configuration, secondly the configuration of the ADC common aspects such as the prescaler that determines the conversion clock rate and the sampling structure, thirdly the individual ADC aspects such as its conversion resolution and data alignment. Some form of trigger is needed to activate the conversion and a simple software trigger is used here. Note that ADC2 is used in this example and that its input on Channel 12 is connected via GPIOC pin 2.

A further example using a timer trigger and an efficient DMA interface will be given in the next section.

4.5.1 ADC Interface Using Timer and DMA

In this procedure the converter is used to make 100 measurements precisely timed at 100 μs intervals. When the block of readings is complete the DMA interface will signal the software so that it can process the results as required.

The main issues that the code has to address in addition to the usual aspects are configuring ADC3 for DMA in this case. The procedure follows the now familiar steps, enabling the peripheral clocks, setting up the physical pin, in this case GPIOC pin 2, as analogue input for channel 12 and configuring the ADC common and individual aspects as well as its channel configuration. It will be seen later that ADC3 uses the controller DMA2 stream 0 or 1 channel 2 and for this example stream 0 is chosen. The memory area receiving the ADC values is defined in the main module and the DMA controller needs to reflect the buffer length specified. The DMA process is managed in this example by monitoring the DMA flag status in DMA_FLAG_TCIFx to determine when the stream transfer is complete. A much more effective arrangement would be employ interrupts that can be generated by the DMA controller because the processor would not be idly waiting for the transfer to complete.

```
int main(void)
{
   uint16_t ADC3_Values[100];

   config_ADC_pins();        /* GPIOC pin 2 */
   config_ADC_common();        /* independent */
   config_ADC3_values();       /*ADC3 configuration */
   config_DMA2_stream();      /* stream 0 channel 2 */

   /* Enable DMA request after last transfer
(Single-ADC mode) */
   ADC_DMARequestAfterLastTransferCmd(ADC3, ENABLE);
```

```
    /* Enable ADC3 DMA */
    ADC_DMACmd(ADC3, ENABLE);

    /* Enable ADC3 */
    ADC_Cmd(ADC3, ENABLE);

    /* Start ADC3 Software Conversion */
    ADC_SoftwareStartConv(ADC3);

    while (1)
    {
        while(DMA_GetFlagStatus(DMA2_Stream0, DMA_
FLAG_TCIFx) != 1)
        {
        }
    /* process block of data */
   /* convert the ADC value (from 0 to 0xFFF) to a
voltage value (from 0V to 3.3V)*/
    ADC3ConvertedVoltage = ADC3ConvertedValue
*3300/0xFFF;
    }
}

void config_DMA2_stream (void)
{
    DMA_InitTypeDef DMA_InitStructure;

    /* Enable DMA2 clocks */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2, ENABLE);

    /* DMA2 Stream 0 channel 2 configuration */
    DMA_InitStructure.DMA_Channel = DMA_Channel_2;
    DMA_InitStructure.DMA_PeripheralBaseAddr =
(uint32_t)ADC3_DR_ADDRESS;
    DMA_InitStructure.DMA_Memory0BaseAddr =
(uint32_t)&ADC3_Values;
    DMA_InitStructure.DMA_DIR = DMA_DIR_
PeripheralToMemory;
    DMA_InitStructure.DMA_BufferSize = 1;
    DMA_InitStructure.DMA_PeripheralInc = DMA_
PeripheralInc_Disable;
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_
Disable;
    MA_InitStructure.DMA_PeripheralDataSize =
DMA_PeripheralDataSize_HalfWord;
```

```
    DMA_InitStructure.DMA_MemoryDataSize = DMA_
MemoryDataSize_HalfWord;
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;
    DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_
Disable;
    DMA_InitStructure.DMA_FIFOThreshold = DMA_
FIFOThreshold_HalfFull;
    DMA_InitStructure.DMA_MemoryBurst = DMA_
MemoryBurst_Single;
    DMA_InitStructure.DMA_PeripheralBurst = DMA_
PeripheralBurst_Single;
    DMA_Init(DMA2_Stream0, &DMA_InitStructure);
    DMA_Cmd(DMA2_Stream0, ENABLE);
}

/* Configure GPIO pin 2 for ADC3 Channel 12 as
analog input */
void config_ADC_pins(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOC, &GPIO_InitStructure);
}

/* ADC Common Init */
void config_ADC_common(void)
{
    ADC_CommonInitTypeDef ADC_CommonInitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC3, ENABLE);

    ADC_CommonInitStructure.ADC_Mode = ADC_Mode_
Independent;
    ADC_CommonInitStructure.ADC_Prescaler = ADC_
Prescaler_Div2;
    ADC_CommonInitStructure.ADC_DMAAccessMode =
ADC_DMAAccessMode_Disabled;
    ADC_CommonInitStructure.ADC_TwoSamplingDelay =
ADC_TwoSamplingDelay_5Cycles;
```

```
    ADC_CommonInit(&ADC_CommonInitStructure);
}

/* ADC3 Init */
void config_ADC3_values(void)
{
    ADC_InitTypeDef ADC_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC3, ENABLE);

    ADC_InitStructure.ADC_Resolution = ADC_
Resolution_12b;
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
    ADC_InitStructure.ADC_ExternalTrigConvEdge =
ADC_ExternalTrigConvEdge_None;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_
Right;
    ADC_InitStructure.ADC_NbrOfConversion = 1;
    ADC_Init(ADC3, &ADC_InitStructure);

    /* ADC3 regular channel 12 configuration */
    ADC_RegularChannelConfig(ADC3, ADC_Channel_12, 1,
ADC_SampleTime_3Cycles);
}
```

4.6 Programming Challenge

A data logging application is required that records temperature readings over a long period assessing the maximum and minimum values over a period of 24 h. Readings should be taken every minute and these should be derived from actual readings every 10 s. A simple temperature sensor such as LM35CZ can be connected up to the Discovery board as shown in the circuit Figure 4.11.

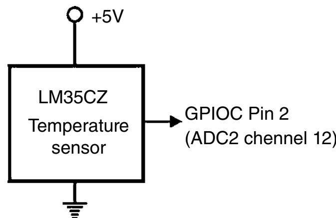


Figure 4.11 Temperature sensor circuit

The transfer relationship for the LMC35 in its simplest mode is 10 mV per °C so at 25°C the output will be 250 mV, conveniently in the converter input range.

4.7 Conclusion

This chapter has focussed on the analogue input and output interface techniques provided within the STM32F4 device. A brief introduction to converter technology has been included to establish the principles involved but this is by no means complete as there are many alternative designs in use that exhibit enhanced performance in terms of conversion rate, which is needed in more demanding applications. Many books focus on converter technology and these should be consulted for further explanation.

Many alternative modes of operation are available from the STM32F4 converter subsystems but it is hoped that the examples given will provide a starting point for a wide range of practical applications.

In a new application the overall requirements must be carefully defined and the key aspects of the conversion process such as resolution and throughput determined. If the STM32F4 converter modules can meet these basic requirements, the mode can then be selected to match the operating scheme demanded.

The STM32F4 Discovery applications package gives further examples. One describes how to use the ADC and DMA to transfer continuously converted data from ADC3 channel 7 to memory. Each time an end of conversion occurs, the DMA transfers the converted data in circular mode. Another describes using the ADC peripheral to convert a regular channel in triple interleaved mode and by using DMA in mode 2, where two values are transferred on each DMA request and achieving a very high throughput rate.

References

[1] Hoeschele, D.F. (1994) *Analog-to-Digital and Digital-to-Analog Conversion Techniques*, 2nd edn. John Wiley & Sons, Ltd.

[2] ST Microelectronics (2011) STM32F4 Reference Manual. RM009 (ARM Cortex-4) Reference Manual Doc ID 018909 Rev 1, www.st.com (accessed September 2014).

Further reading

ST Microelectronics (2012) STM32F4 Data Sheet. S32F405xx (ARM Cortex-4) Data Sheet Doc ID 022152 Rev 3, www.st.com (accessed September 2014).

5

Serial Interface Subsystems

This chapter will focus on serial communications interfaces because these form an important aspect of many embedded system designs. In particular industry standard communication protocols will be described together with the design of supporting software. Examples will feature RS232 Communications, inter-integrated circuit (I2C) and serial peripheral interface (SPI) protocols and the universal serial bus (USB). Although C code will be employed for the examples, particular aspects of the linked assembler code will be explained where this is significant for enhanced performance.

5.1 Introduction

Serial interfaces are required when the complexity and cost of a parallel interface cannot be justified such as where the communication distance is more than a few meters. In a serial interface the data bits are assembled into a serial pattern for transmission along a single wire to their destination. In order to recover the original data a synchronised clock is required in the receiver to extract the bits from the serial stream as they arrive. Some serial interfaces use a supplementary dedicated connection for clock, whereas others require the receiver to regenerate the clock locally, obviating the need for any extra connections.

Digital Interface Design and Application, First Edition. Jonathan A. Dell.
© 2015 John Wiley & Sons, Ltd. Published 2015 by John Wiley & Sons, Ltd.

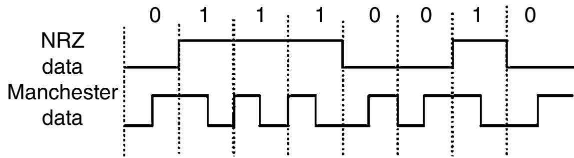


Figure 5.1 Manchester code

For short distance serial interfaces a clock connection can be accommodated because the cost will not be significant but for longer distance communications a local clock regeneration arrangement is the only feasible alternative. Some serial systems employ an encoding process where the data is delivered in a form that will enable self-clocking to be established, an example of this is the Manchester code. In essence this uses a high to low transition at the bit time centre to encode a one and a high to low transition at the bit centre to encode a zero. It can be observed from Figure 5.1 that this requires many extra signal transitions and consequently a greater bandwidth in a communications channel. It is, however, used in the 10 Mbs Ethernet local area network implementation. As a reminder NRZ means non-return to zero between bits.

5.2 RS232 Universal Asynchronous Receiver/Transmitter (UART) Communications

Historically, RS232 was used to interconnect computers and distant terminals and was the first serial protocol to be widely accepted throughout industry for these longer distance communications. It uses a single wire pair; that is signal and ground and delivers reliable communication at low cost. To solve the clock regeneration problem this scheme uses a local clock that is triggered by the falling edge at the start of the signal group and synchronism is assumed to be accurate enough over a limited group of following bits. Before the block of 8 data bits is transmitted the serial line is held high so that the start of data is clearly defined. The serial signal format including the essential stop bit to return the line to the high state is shown in Figure 5.2. The special hardware module designed for this function is known as a universal asynchronous receiver transmitter (UART) and this includes the clock regeneration. In practice the clock regeneration is usually based on a measurement of the start-bit length and then the following bits are accepted at the approximate centre of their bit position, thus avoiding the inevitable distortion of the signal edges that accumulate over a lengthy connection path.

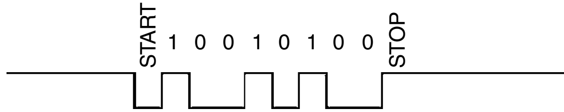


Figure 5.2 RS232 data format

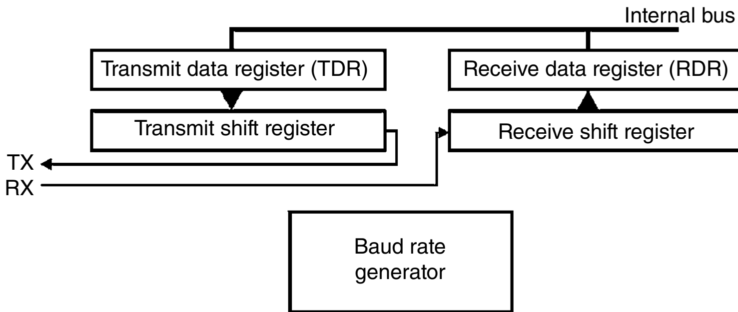


Figure 5.3 UART module

The STM32F4 includes six blocks that can implement UART functions but four of them can also implement synchronous communications where a controlling clock signal is generated, so they are actually called universal synchronous/asynchronous receiver/transmitter (USART) modules [1].

The diagram in Figure 5.3 shows the main blocks of a typical UART module. The baud rate generator provides the bit timing clocks for transmitter and receiver shift registers. A typical rate of 9600 bps will require a transmitter clock period of 1/9600 almost exactly 104 µs, this is usually derived from the internal system clock and the UART reference manual provides values for the BRG (Baud Rate Generator) registers to achieve a variety of standard rates [1].

In the default mode the equation for the baud rate divider (BRDIV) is $BRDIV = \dfrac{fck}{8 \times 2 \times BRATE}$ when f_{ck} is set to 8 MHz the divider becomes 52.0625. In many designs the fractional part is ignored resulting in a slight but insignificant error. In the STM32F4 USART implementation the fractional part of the divider is set in the least significant 4 bits $(0.0625 \times 16) = 1$ so the resulting value of baud rate register (USARTDIV) is 0x0341 [1]. This provides a considerable improvement in clock accuracy. The STM32F4 interface utilities actually perform this calculation for the user so only the actual baud rate required is submitted in the initialisation function call.

On the STM32F4DIS-BB base board USART6 is available and a special higher voltage buffer SP3232EEY to implement the standard ±12V signal is included for the 9-pin D connector COM1. Code to connect the pins and initiate USART6 is shown next. Note the USART nomenclature used on the STM32F4 to emphasise the additional synchronous option of this module [1].

```
void GPIO_set_ioports(void)
{
    /* USART6 uses Alternate Function 8 */
    /* TX is on GPIOC Pin6 and RX is on GPIOC Pin7 */

    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);

    /* TX output and RX input*/
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;

    GPIO_Init(GPIOC, &GPIO_InitStructure);

    /* select Alternate functions (GPIOC->AFR[0] =
0x88000000;)*/
    GPIO_PinAFConfig(GPIOC, GPIO_Pin_6, GPIO_AF_USART6);
    GPIO_PinAFConfig(GPIOC, GPIO_Pin_7, GPIO_AF_USART6);
}

void USART6_setup(void)
{
    USART_InitTypeDef  USART_InitStruct;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART6,
ENABLE);

    USART_InitStruct.USART_BaudRate = 9600;
    USART_InitStruct.USART_WordLength = USART_
WordLength_8b;
    USART_InitStruct.USART_StopBits = USART_StopBits_1;
    USART_InitStruct.USART_Parity = USART_Parity_No;
    USART_InitStruct.USART_Mode = USART_Mode_Rx |
USART_Mode_Tx;
```

```
    USART_InitStruct.USART_HardwareFlowControl =
USART_HardwareFlowControl_None;      /* if using
both _RTS_CTS */

    USART_Init(USART6, &USART_InitStruct);
    USART_Cmd(USART6, ENABLE);
}
```

As the serial interface is quite slow compared to code execution the handling of transfers to the transmit data register (TXDR) and from the receive data register (RXDR) must be constructed carefully to prevent possible overruns in the case of transmission or reception not being completed. The code that follows shows how the transmit register empty (TXE) flag is tested in the case of transmission, transmission complete (TC) is also tested to safeguard the process. For reception receiver not empty (RXNE) is tested to wait for the buffer to be full, then the received data register can be accessed.

```
void char_print(char p)
{
/* outut to serial interface on UART6 */
        /* wait for TXE */
        while (USART_GetFlagStatus(USART6, USART_
FLAG_TXE) != 1)
        {
        }
        USART_SendData(USART6, p);
        while (USART_GetFlagStatus(USART6, USART_
FLAG_TC) != 1)
        {
        }
}

char get_char(void)
{
    /* input via serial input on UART6 */
    char p;
    /* wait till buffer is full */
    while (USART_GetFlagStatus(USART6, USART_FLAG_
RXNE) != 1)
    {
    }
```

```
    p = USART_ReceiveData(USART6);
    return(p & 0x7f);
}
```

These functions could quite easily be implemented in assembler code because the USART6 registers addresses and the bit test positions are well documented. However, there would not be much advantage in using this approach as the data transfer rate is quite slow in any case.

Note that when the PC Hyper-terminal is used it will usually be essential to defeat the hardware flow control, which is implemented by default. This is achieved by supplying a signal for the request to send (RTS), pin 7 on the 9-way D-type connector, to activate transmission from the keyboard. This can be obtained from the clear to send (CTS), pin 8 on the D-type, so a wire link between these pins on the back of the Discovery base board will satisfy this requirement.

5.3 The I2C Interface

The I2C was developed for short interconnections between components, offering a considerable economy in situations where multiple connections for data, address and control would have been used otherwise. This greatly simplifies the printed circuit board (PCB) design as it only requires two connections, apart from the common ground. This is shown in the diagram Figure 5.4.

The interfaced components have a master/slave relationship and the connections are named serial clock (SCL) and serial data (SDA). Both use an open-drain configuration with pull-up resistors so that drive can be activated

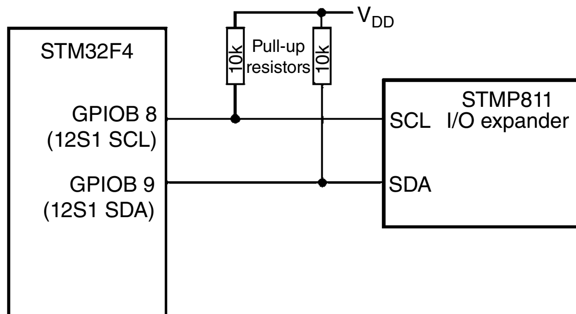


Figure 5.4 An I2C interface circuit

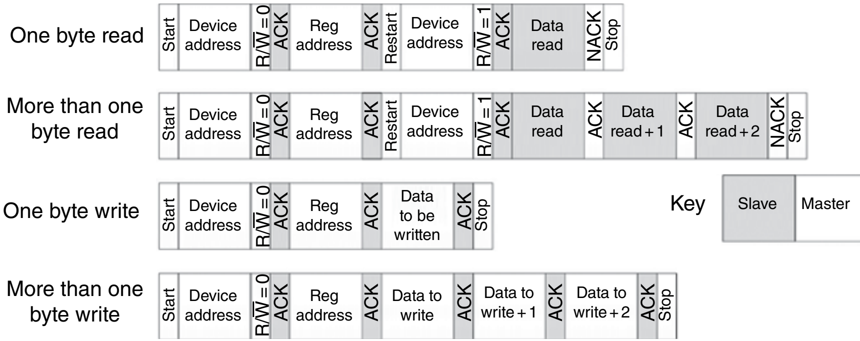


Figure 5.5 I2C read and write modes

in either device at appropriate times. The STM32F4 has three independent I2C interface modules and two further modules that can take on the I2C role if required. Read and write data transfers across the I2C bus are defined by the diagrams shown in Figure 5.5.

The start condition is defined when SCL is high and SDA falls and the stop condition is defined when SCL is high and SDA rises. The register address is 7 bits and the eighth least significant bit (LSB) indicates whether a read or write (R/W) operation follows. The STM32F4 support utilities file stm32f4xx_i2c.c provides software resources to initialise the I2C hardware block in preparation for communication and but it is up to the user to assemble message sequences correctly in software.

The acknowledge bit (ACK) is generated by the receiving device and the master provides an extra clock cycle for this to happen. Note that a new start (RESTART) is needed to switch the direction on SDA in the read processes. To terminate the read transfers the master issues a negative acknowledgement (NACK) on the final read cycle. This ensures that the slave releases the SDA line so that the master can generate the stop condition. In the write transfers the master is in control so it can terminate the sequence by creating the stop condition.

The STM32F4 I2C interface also offers a 10-bit addressing mode for components with extended capability.

5.3.1 Using the Touch Screen with an I2C Interface

A useful example of an I2C interface is set up on the STM32F4 DIS_LCD board forming an interface with the touch screen controller STMP811. This component also contains a temperature sensor so this forms the subject of this example. The code that follows shows how the I2C module connections are

set up and the initialisation is achieved. Notice that the pins are defined as open-drain (OD) so that the SCL and SDA lines can perform correctly. Also in this case the I2C clock speed is set to 100 kHz.

```
void I2C_InitPinsandI2C(void)
{
   I2C_InitTypeDef I2C_InitStructure;
   GPIO_InitTypeDef GPIO_InitStructure;

   RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);
   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);

   /* set up B8 (SCL) and B9 (SDA) */
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 |
GPIO_Pin_9;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
   GPIO_InitStructure.GPIO_OType = GPIO_OType_OD;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;

   GPIO_Init(GPIOB, &GPIO_InitStructure);

   GPIO_PinAFConfig(GPIOB, GPIO_Pin_8, GPIO_AF_I2C1);
   GPIO_PinAFConfig(GPIOB, GPIO_Pin_9, GPIO_AF_I2C1);
/* Note: the ST code module is incorrect so a
direct method is used here */
   GPIOB->AFR[1] = 0x044;

   I2C_InitStructure.I2C_ClockSpeed = 100000;
   I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
   I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;
   I2C_InitStructure.I2C_OwnAddress1 = 0x00;
   I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
   I2C_InitStructure.I2C_AcknowledgedAddress =
I2C_AcknowledgedAddress_7bit;

   I2C_Init(I2C1, &I2C_InitStructure);
   I2C_AcknowledgeConfig(I2C1, ENABLE);
   I2C_Cmd(I2C1, ENABLE);
}
```

A particular sequence of actions is required to set up the STMP811 initialising and reading the various registers. It has no less than 49 internal registers

Table 5.1 STMP811 system control register 2

SYS_CR2 Bit #	Name	Function
3	TS_OFF	Temperature gauge off
2	GPIO_OFF	I/O expansion off
1	TSC_OFF	Touch screen controller off
0	ADC_OFF	Analogue to digital converter off

Table 5.2 Temperature gauge control register

TEMP_CTRL Bit #	Name	Function
4	THRES_RANGE	0 – greater than or equal threshold 1 – otherwise
3	THRES_EN	Temperature threshold enable
2	ACQ_MODE	0 – once only 1 – every 10 ms
1	ACK	Acquire temperature
0	EN	Enable

controlling its four main functions, a GPIO expander, an analogue to digital (AD) converter, a touch-screen interface and a temperature gauge. System control register two at address 0x04 can select the parts that are actually needed as detailed in Table 5.1.

Firstly the elements I/O Expander are all switched on by writing zero to register 0x04, secondly the temperature subsystem mode is selected by writing to register TEMP_CTRL at address 0x60 where the bits have the functions shown in Table 5.2.

Finally the temperature value is read from register 0x61 (MSB) and 0x62 (LSB).

```
IOE_write_reg(I2C1, 0x04, 0);       /* SYS_CR2 */
IOE_write_reg(I2C1, 0x60, 7);     /* TEMP_CTRL */

temp1 = IOE_read_reg(I2C1, 0x61);   /* MS Byte */
temp0 = IOE_read_reg(I2C1, 0x62);   /* LS Byte */
```

The code for writing to a register within the structure of the STMPE811 I/O expander is shown next in the function IOE_write_register() [2]. This requires five I2C steps as determined by the 1-byte write sequence in Figure 5.5. Each

operation must check appropriate I2C event status bits to confirm that the appropriate operations have been accomplished before the next one is attempted.

```
void IOE_write_reg(uint8_t DeviceAddr, uint8_t
RegisterAddr, uint8_t RegisterValue)
{
   /* 1 send start */
   I2C_GenerateSTART(I2C1, ENABLE);
   while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
MODE_SELECT) != 1)
   {}
   /* 2 send device address */
   I2C_Send7bitAddress(I2C1, DeviceAddr, I2C_
Direction_Transmitter);
   while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
TRANSMITTER_MODE_SELECTED) != 1)
   {}
   /* 3 send register address */
   I2C_SendData(I2C1, RegisterAddr);   /*ident
register select*/
   while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
BYTE_TRANSMITTED) != 1)
   {}
   /* 4 send register value */
   I2C_SendData(I2C1, RegisterValue);
   while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
BYTE_TRANSMITTED) != 1)
   {}
   /* 5 send stop condition */
   I2C_GenerateSTOP(I2C1, ENABLE);
}
```

For reading the contents of a register the code next shows the seven step sequence required as shown in the 1-byte read case in Figure 5.5. This starts off in the same way as the write operation but at step 4 a RESTART must be issued so that the I2C interface can be switched to a read mode. Also it will be observed that the last I2C cycle will have to generate the negative acknowledge so the configuration is set to I2C_NACKPosition_Current just before the single read operation in step 6.

```
uint8_t IOE_read_reg(uint8_t DeviceAddr, uint8_t
RegisterAddr)
    {
    uint8_t temp;

    /* 1. generate START */
    I2C_GenerateSTART(I2C1, ENABLE);
    while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
MODE_SELECT) != 1)
    {}
    /* 2. ADDRESS and WRITE */
    I2C_Send7bitAddress(I2C1, DeviceAddr, I2C_
Direction_Transmitter);
    while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
TRANSMITTER_MODE_SELECTED) != 1)
    {}
    /* 3. REG address */
    I2C_SendData(I2C1, RegisterAddr);

    /*outp register select*/
    while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
BYTE_TRANSMITTED) != 1)
    {}
    /* 4. RESTART */
    I2C_GenerateSTART(I2C1, ENABLE);
    while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
MODE_SELECT) != 1)
    {}

    /* 5. ADDRESS and READ */
    I2C_Send7bitAddress(I2C1, DeviceAddr, I2C_
Direction_Receiver);
    while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
RECEIVER_MODE_SELECTED) != 1)
    {}
    /* 6. READ byte, NACK follows */
    I2C_NACKPositionConfig(I2C1, I2C_NACKPosition_
Current);

    while(I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_
BYTE_RECEIVED) != 1)
    {}
```

```
    temp = I2C_ReceiveData(I2C1);

    /* 7. send STOP*/
    I2C_GenerateSTOP(I2C1, ENABLE);

    return(temp);
}
```

It will be seen that the I/O Expander can also generate an interrupt from various sources within its logic such as the touch sensor that can be useful in its overall management.

Debug of interface modules of this type, where a number of different actions have to be taken, is quite difficult to approach. It would be advantageous to include a route to exit from the while loops if an error occurs or if the procedure takes too long to complete. This will avoid the problem when the code gets stuck for no apparent reason.

If one of the timers is used and set to a period of 100 ms, for example the code shown next illustrates a possible design, here the timeout() function returns zero until a specified time has elapsed. Different return values can be used to identify the error reporting code section.

```
    if (timeout())
    {
        DEBUG_PRINT("RX Timeout error\r\n");
        return(1);
    }
```

5.4 SPI Interface

The SPI was developed for similar short distance applications in embedded systems as I2C but offering a considerably higher data throughput based on a clock rate of several megahertz and considerably reduced protocol overhead. One of its many possible applications is to form the basic mode communication link with an SD memory card. It forms a synchronous data link performing full duplex communication between devices having a Master/Slave relationship. The Master always initiates the communication process and also provides the clock allowing one or multiple Slave units to be accommodated.

The SPI link requires four connections, as shown in Figure 5.6, apart from the common ground, these are serial clock (SCK in the figure), master out/slave in (MOSI), master in/slave out and active low slave select ($\overline{SS}$). Each of

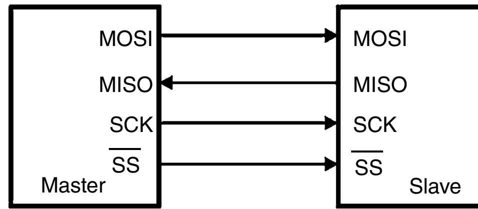


Figure 5.6 SPI interface connections

the three available SPI interface modules on the STM32F4 implement a single $\overline{SS}$ select and this is clearly redundant in many cases if only a single slave is envisaged in the design.

The SPI interface connections are mapped through GPIOA, GPIOB and GPIOC as shown in the STM32F4 data sheet Table 8 Alternate Function Mapping [2]. Taking SPI3 the assignment is mainly on GPIOC, SCL uses pin 10, MISO (Master In/Serial Out) uses pin 11, MOSI uses pin 12 and finally the slave select is assigned to GPIOA pin 15. The interface is quite straightforward to set up as shown by the initialisation code next. The main issue to consider is the clock edge on which you require data to become valid, this will usually be determined by the particular peripheral in question and its data sheet should be studied carefully.

```
void init_SPI_pins(void)
{
   GPIO_InitTypeDef GPIO_InitStructure;
   SPI_InitTypeDef SPI_InitStruct;

   RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI3, ENABLE);
   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC,
ENABLE);

   /* GPIOD Configuration: (SCK GPIOC10, MISO C11,
MOSI C12) */
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10 |
GPIO_Pin_11 | GPIO_Pin_12;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
   GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;

   GPIO_Init(GPIOD, &GPIO_InitStructure);
```

```
    GPIO_PinAFConfig(GPIOC, GPIO_Pin_10, GPIO_AF_SPI3);
    GPIO_PinAFConfig(GPIOC, GPIO_Pin_11, GPIO_AF_SPI3);
    GPIO_PinAFConfig(GPIOC, GPIO_Pin_12, GPIO_AF_SPI3);
/* note this function mishandles AFR[1] assignment */
    GPIOC->AFR[1] = 0x00066600;

    SPI_InitStruct.SPI_Direction = SPI_
Direction_2Lines_FullDuplex;
    SPI_InitStruct.SPI_Mode = SPI_Mode_Master;
    SPI_InitStruct.SPI_DataSize = SPI_DataSize_16b;
    SPI_InitStruct.SPI_CPOL = SPI_CPOL_Low;
    SPI_InitStruct.SPI_CPHA = SPI_CPHA_1Edge;
    SPI_InitStruct.SPI_NSS = SPI_NSS_Soft;
    SPI_InitStruct.SPI_BaudRatePrescaler = SPI_
BaudRatePrescaler_2;
    SPI_InitStruct.SPI_FirstBit = SPI_FirstBit_MSB;
    SPI_InitStruct.SPI_CRCPolynomial = 0;

    SPI_Init(SPI3, &SPI_InitStruct);

    SPI_Cmd(SPI3, ENABLE);
}
```

5.4.1 SPI Interface to an Analogue to Digital Converter

Although there are many components equipped with comprehensive implementations of the SPI interface many others use a more application oriented implementation set up to accomplish specific design objectives. In this example application the 16-bit AD7680 was designed to operate up to 100k samples per second so a dedicated SPI interface implementation was used. This AD converter only has a MISO connection besides the clock and its select signal is used to trigger new conversions. The circuit diagram is shown in Figure 5.7.

The SPI master mode will be used to generate clocks, although no command to the slave is actually required, so a dummy 0xff data byte is conventionally transmitted. To start with the converter has to be brought out of its power-down mode before a valid conversion can be made, this can be achieved by issuing a dummy 16 clock cycles with SS low. Conversion will start the next time SS goes low. It will be essential to generate the SS signal by manipulating a simple output because when the SPI is operating in master mode the slave select signal generated internally remains low continuously. For this

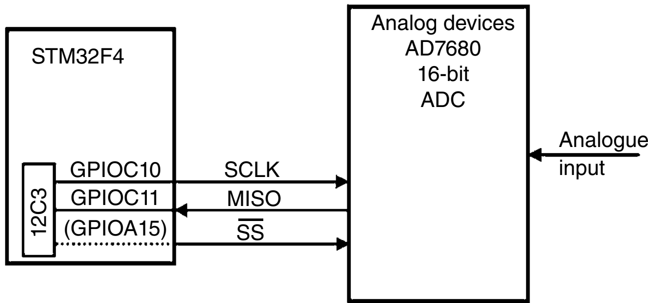


Figure 5.7 AD7680 connections

application GPIOA pin 15 is assigned as a simple output so that it can be set low before clocks are generated and return high afterwards.

The SPI direction can be temporarily reassigned if required:

```
SPI_BiDirectionalLineConfig(SPI_Direction_1Line_Tx);
```

The code shown next will generate 16 clock cycles and wake up the converter.

```
    GPIO_ResetBits(GPIOA, SS_Pin);
    SPI_I2S_SendData(SPI3, temp);
    while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_
TXE) != 1)
    {
    }
    SPI_I2S_SendData(SPI3, temp);
    while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_
TXE) != 1)
    {
    }
    GPIO_SetBits(GPIOA, SS_Pin);
```

Data can then be retrieved from the converter as soon as the receiver register is full. However, the converter requires 24 clock cycles on SCL to complete the conversion and deliver all the data bits. Within this bit-stream the data sheet shows that there are four leading and four trailing zeros surrounding the 16-bit converter data. Three bytes will need to be transmitted to achieve this. Reading out the received byte can be interleaved with the transmission so that the receiver buffer is ready for the next reception.

```
GPIO_ResetBits(GPIOA, SS_Pin);
SPI_I2S_SendData(SPI3, temp);
while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_
TXE) != 1)
{
}
SPI_I2S_SendData(SPI3, temp);
while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_
RXNE) != 1)
{
}
temp1 = SPI_I2S_ReceiveData(SPI3);
while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_
TXE) != 1)
{
}
while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_
RXNE) != 1)
{
}
temp2 = SPI_I2S_ReceiveData(SPI3);
```

Further code along the same lines will be needed to complete the required 24 clock cycles. The slave select pin can be taken high again after all the clocks have been delivered.

5.5 HDLC Serial Communication

As the demand for high speed digital communication increased the limitations and overheads of the simple strategies described earlier became evident. Also with the advent of improved techniques such as the phase-locked-loop (PLL) it became much more straightforward to regenerate a reliable clock in the receiver using the data edges as a reference. The PLL has inherent inertia like its mechanical counterpart so its operation will not be compromised if some of the data edges are missing, such as when there are consecutive ones or zeros. All that must be insured is that such a constant one or zero situation is not allowed to persist too long. The High-Level Data Link Control (HDLC) scheme employs extra bits stuffed into the data stream every time there are five consecutive ones, these can easily be removed in the receiver to restore the original data. The start of a data frame is still required so a special Flag

symbol of six consecutive ones is used, this can never occur anywhere else in the frame. The frame is transmitted LSB first in the NRZI format where consecutive ones remain high and zeros change at the centre of bit time. This ensures a data transition at least every 6 bit times during the frame and every 7 during the flags. Specialised hardware in the transmitter and receiver interface modules handles these operations, which mark out clearly the frames boundaries and provide enough edges to stabilise the PLL clock. The resulting frame structure is shown in Figure 5.8.

The Address byte effectively forms a channel number to distinguish primary and secondary sources.

The control byte distinguishes three types of frame as shown in Figure 5.9, Information (I), Supervisory (S) and Unnumbered (U). In the information frame control byte there are both send N(S) and receive N(R) sequence numbers encoded in three bits so error control information can be included. In a later version the sequence numbers were extended to 7 bits to improve efficiency. The supervisory frame is used for flow and error control when there is no data to send, there are only four possible frames in this group. The U frames are used mainly for link set up and management functions.

The P/F bit has two functions, Poll when set by the primary to obtain a response from a secondary, and Final when set by the secondary to indicate a response or the end of a transmission sequence.

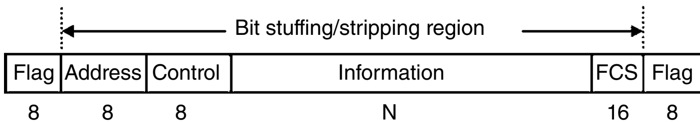


Figure 5.8 HDLC frame structure

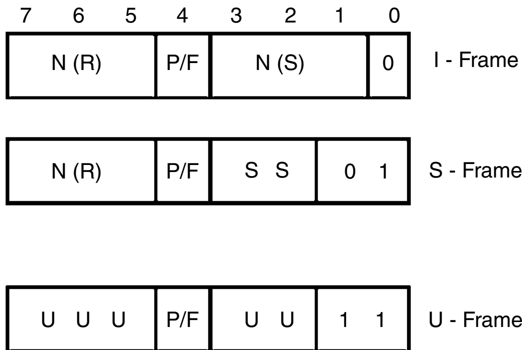


Figure 5.9 HDLC frame control byte

The frame check sequence (FCS) is a cyclic redundancy check (CRC) calculation that forms a very effective test for errors on the complete frame. The frame always finishes with another Flag, which may sometimes form the start Flag for the next frame.

Only a few of the details have been discussed here but HDLC was the inspiration for the standard IEEE 802.2, which is widely employed in connecting to the Internet so its relevance is obvious.

5.6 The Universal Serial Bus (USB)

The USB protocol was introduced to provide a more flexible and reconfigurable interconnection arrangement for PC peripherals, that would allow items to be added or removed without the need to modify and adapt the whole system. USB is now very familiar to PC users allowing a wide variety of devices such as digital cameras, MP3 audio players, webcams and memory sticks to be connected. It is another serial communications protocol with many similarities to HDLC that has been widely adopted throughout the PC industry but is much more complex than the other techniques that have been covered in this chapter so far. A sketch of the USB connector is shown in Figure 5.10 but a wide range of connector physical sizes are available.

USB employs a four wire connector comprising of data lines (D+ and D−), power and ground. The D+ and D− form a differential connection path using a pair of twisted wires, which allows the USB to work at high speeds across a link up to 5 m in length. The included power connection provides a 5 V supply, supporting loads up to 100 mA, which can be used by the linked device if it does not have its own power source. The connector is designed carefully to ensure the order of connection so that ‘hot swapping’ can be accomplished safely. The cable shield is connected first, then the power and ground then finally the data lines. Three classes of data rate are recognised standards, these are high speed (HS) 480 Mbps, full speed (FS) 12 Mbps and low speed (LS) 1.5 Mbps.

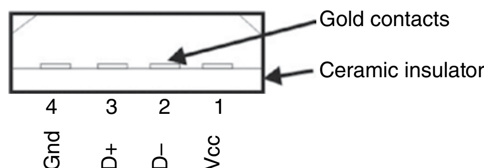


Figure 5.10 USB connections

During USB communication data is transmitted as packets in NRZI format, where a sequence of ones remain high but zeros always toggle at the bit centre. The packet always starts with a special sync (00000001) and bit stuffing is used to break up sequences of one. A special terminating end-of-packet (EOP) sequence is also defined. Initially all packets are sent from the host, via the root hub, to devices. Some of those packets direct a device to send some packets in reply. There are three main types of packet and these are Hand-shake, Token and Data, which all have a unique packet identifier (PID) as the first byte. Table 5.3 shows a basic set

Table 5.3 USB PID byte functions

PID #	PID byte	Name	Notes
0000	0000 1111		Reserved
1000	0001 1110	**SPLIT**	USB 2 split transaction
0100	0010 1101	**PING**	Does endpoint accept USB 2?
1100	0011 1100	**PRE**	Special for low-bandwidth USB
		ERR	Split transaction error for USB 2
0010	0100 1011	**ACK**	Data packet accepted
1010	0101 1010	**NAK**	Data packet not accepted; retransmit required
0110	0110 1001	**NYET**	Data not ready yet (USB 2)
1110	0111 1000	**STALL**	Transfer impossible; error recovery needed
0001	1000 0111	**OUT**	Host-to-device transfer address
1001	1001 0110	**IN**	Device-to-host transfer address
0101	1010 0101	**SOF**	Start of frame marker sent every millisecond
1101	1011 0100	**SETUP**	Host-to-device control transfer address
0011	1100 0011	**DATA0**	Even # data bytes
1010	0101 1010	**DATA1**	Odd # data bytes
0111	1110 0001	**DATA2**	USB 2 data packet
1111	1111 0000	**MDATA**	USB 2 data packet

Key:

Hand-shake	Token	Data

of PID values that have been extended to accommodate the USB 2 standard. The actual PID byte is transmitted LSB first and the last four bits are a complement of the first four. This helps with error checking when the packets are short.

5.6.1 Hand-shake Packets

These packets only use the PID byte; error checking relies on the byte structure alone. The only possible response from the host is ACK; if it is not ready it won't ask for data to be sent. NYET and ERR were added for USB2 extensions. NYET allows the device to indicate that its buffers are full so the host can use a PING before proceeding rather than sending a whole frame that will be rejected.

5.6.2 Token Packets

Tokens are only sent by the host and consist of the PID followed by an 11 bit field and a 5 bit CRC. The 7 bits of this data field are the device address and the last four give the number of data packets required. The appropriate handshake for **IN** and **OUT** tokens will be expected in response.

The **SETUP** token operates like an **OUT** token but is followed by an eight byte **DATA0** packet with a special format.

The USB host sends out a start of frame **SOF** token, which contains an 11-bit incrementing number field, every millisecond to synchronise data transfers. In USB 2 seven extra **SOF** tokens are introduced to define 'microframes' containing 60 000 bit times for the fastest data rate.

5.6.3 Data Packets

Up to 1024 data bytes can follow these PID values (64 for medium speed and 8 for low speed). The two forms DATA0 and DATA1 alternate and allow a better error management to be achieved. The receiver keeps track of the sequence so that when a packet is lost the sequence will be violated and a retransmission can be requested.

The most recent acknowledgement will indicate which type was received correctly. So unless the acknowledgement matches the last packet transmitted it is this that will need retransmission.

The PRE packet is a special preamble issued by the host just prior to a low bandwidth packet communication.

5.6.4 *USB Protocol*

When the device is first connected the host performs a process called enumeration where it first resets the device, then assigns it an address and proceeds to interrogate it to determine the basic operating characteristics, such as device type and data rate. All subsequent data communications are initiated by the host and use a hand-shake exchange to confirm success. The host uses the address of the target device and specifies the class type and direction of data transfer required. As devices are enumerated, the host keeps track of the total bandwidth that all of the devices are requesting whether they are using the isochronous or interrupt driven modes. They can consume up to 90% of the total bandwidth available (i.e. with USB 2.0 480 Mbps and with USB 3.0 4.8 Gbps). After the 90% allocation is used up, the host denies access to any other isochronous or interrupt driven devices. Control packets and packets for bulk transfers use any bandwidth left over (i.e. at least 10%).

The STM32F4 device contains two USB interface modules that can be configured to operate as host or slave device according to the particular application envisaged. These are both fully implemented on the Discovery board and one of them is used to download code from the PC host and interact with the integrated development environment (IDE) in Debug mode.

The ST Microelectronics support package contains valuable USB examples using both the host mode and the device mode. One of the host examples in the file USB_Host_Examples implements an inertial mouse based on the Discovery board using the USB human interface device (HID) class and another in the file USB_Device_Examples provides an interface, using the USB mass storage interface class (MSC), to a memory microSD card. Other examples using the host and device modes are also provided giving a range of possible applications.

5.7 Programming Challenge

The development of an interface for the touch-sensitive screen is required; this is shown in Figure 5.11 and uses the I2C interface to communicate with the STMP811 controller chip on the LDC35RT display board. Whenever a user input is detected a corresponding set of X-Y coordinates should be delivered to the hyper terminal on the PC. Do not attempt to use the interrupt at this stage unless you are familiar with the material in Chapter 7.

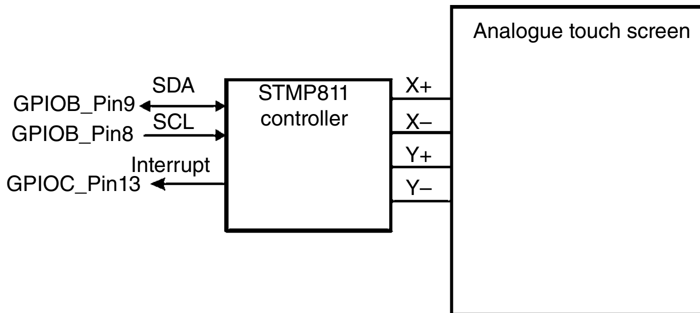


Figure 5.11 Touch screen interface

5.8 Conclusion

Various serial interface protocols including RS232, I2C, SPI and USB, have been examined in this chapter. These are widely utilised in common practice for a huge variety of different applications and a thorough knowledge of them will be invaluable when a new design is considered or new system components are required.

The software applications illustrated utilise simple polling techniques to determine when data can be transmitted or when data has been received but it should be becoming evident that more efficient interface management would be a significant advantage particularly when there is a large volume of data to deliver or retrieve. The interrupt techniques essential in achieving this will be described in Chapter 6.

Only brief details of advanced protocols like USB have been provided in this chapter because very comprehensive and well documented examples are provided in the STM32F4 Discovery support software package [1]. These examples will form a useful starting point for new design requirements.

References

[1] ST Microelectronics (2011) STM32F4 Reference Manual. RM009 (ARM Cortex-4) Reference Manual Doc ID 018909 Rev 1, www.st.com (accessed September 2014).

[2] ST Microelectronics (n.d.) STMPE811 Data Sheet, Doc ID 14489 Rev 5, www.st.com (accessed September 2014).

STM32F4 Peripheral Driver functions from the support library:
stm32f4xx_usart.c
stm32f4xx_i2c.c
stm32f4xx_spi.c
USB Protocol Documentation from various web sites.

6

Advanced Functions

6.1 Advanced Functions

This chapter will focus on techniques required in order to support more complex interfaced peripherals. In particular Interrupt and direct memory access (DMA) techniques are included to show how more efficient interface management can be achieved. The general requirements of wireless interfaces will be discussed and some practical examples including digital camera and LCD panel display interfaces will be described. The STM32F4 software examples included in the support package are an invaluable resource illustrating the different ways that these interfaces can be handled.

6.2 Interrupts

The general concept behind interrupts is that an event in a peripheral, such as an analogue to digital converter (ADC) value becoming available or a timer overflow occurring, can suspend the current processor background operation and instigate particular processor action to deal with that event automatically. When the event servicing action is complete the processor is allowed to return to its normal operation once again.

Digital Interface Design and Application, First Edition. Jonathan A. Dell.
© 2015 John Wiley & Sons, Ltd. Published 2015 by John Wiley & Sons, Ltd.

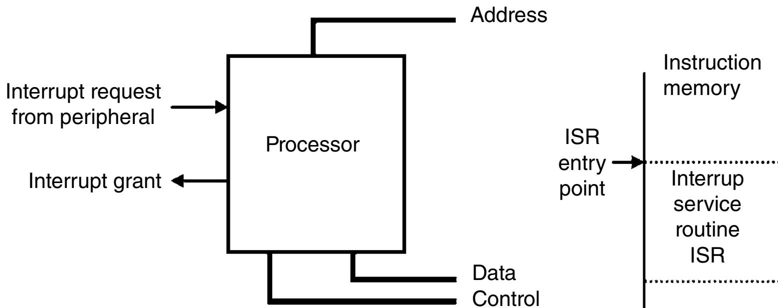


Figure 6.1 Interrupt basic concepts

These basic concepts are shown in Figure 6.1 and the sequence of action that will follow the arrival of the hardware interrupt request signal from the peripheral is explained later. First the processor will complete its current instruction, which may take a few processor clock cycles depending what stage execution has reached. This is known as the inherent latency and some processors are able to stop more quickly than others. In fact, the ARM Cortex 4 has very low latency for these interrupting actions. If the interrupt is accepted the interrupt grant signal is issued as soon as this situation is achieved. At this point the processor stores the location that the current code has reached and uses a vector to access the starting point of the service routine. When the service code is finally completed the processor retrieves the stored location and resumes whatever operations were stopped.

In practical situations, when several peripherals can generate competing interrupt requests, more complex logical arrangements are required to sort this out. Firstly a priority scheme is usually used to determine which interrupt will be acted upon and those that will be locked out for the present. Normally the higher priority interrupt will be serviced first and when it is complete the lower priority interrupt will take over. Secondly a more extensive vector table will be needed to link the various interrupts with their different service routines. In the STM32F4 the vector table is placed in a low memory area starting at (0x0000 0040) and each peripheral is assigned to a particular vector location so that it can be accessed efficiently and easily.

Interface processing organised around interrupts can be much more efficient in terms of processor resources than the more simple polling approach that has been used up to now. In earlier examples documented in this book, such as the interface involving a comparatively slow ADC, the polling approach employed would waste valuable processor resources while the converter finishes its job. Using an interrupt would allow the processor to proceed

with other background tasks and automatically process converter data when it becomes available and not before. A human user interface, such as a touch-screen, illustrates another application where interrupts are of great value because for much of the time the interface will be idle and processor resources would easily be wasted waiting for the user to take some action such as touching a new point on the screen.

So to provide interrupt capability most processors including the STM32F4 contain special hardware elements that allow an event in one of the subsystems or external signal to activate the interrupt and allow the processor to branch, using the vector table, to special related code known as the interrupt service routine (ISR). This special functionality also allows the interrupted processor task to be resumed automatically once the ISR is completed.

6.2.1 Interrupts in the STM32F4

The STM32F4 contains many subsystems such as timers, converters and serial interface modules that can generate an interrupt event so there are potentially many sources that need to be handled correctly. To facilitate interrupts in this processor a vectored and prioritised system is implemented. The vector table (see extract in Table 6.1) provides the link between the particular interrupt and its ISR and the priority controller sorts out the situation when several different events occur simultaneously through their predefined priority relationship.

Table 6.1 An extract of the vector table; the full table has more than 81 entries!

Position	Priority	Function	Address
6	13	EXT interrupt 0	0x0000 0058
7	14	EXT interrupt 1	0x0000 005C
11	18	DMA stream 0	0x0000 006C
12	19	DMA stream 1	0x0000 0070
18	25	ADC interrupts	0x0000 0088
28	35	TIM2 interrupt	0x0000 00B0
29	36	TIM3 interrupt	0x0000 00B4
31	38	I2C1 interrupt	0x0000 00BC

In practice the user has to complete a number of different steps to implement an interrupt controlled interface. Templates for some of these tasks, such as the file stm32f4xx._it.c that is for all exception handler and peripheral ISRs, are provided in the support software from ST Microelectronics. In essence the user will be required to satisfy the following steps in preparing for an implementation of an interrupt driven interface.

1. Set up the interrupt controller so that it can access the Vector Table entry related to the peripheral hardware in question, this will include the user selected part of priority assignment. A typical structure for this part of the task will be shown next.
2. Design the ISR that can handle the peripheral in question. This can be placed in the interrupt code module as long as it is named correctly to connect with the vector table entry definition found in the file startup_stm32f4xx.s.
3. Enable the particular event source by setting interrupt enable (IE) bits in the related peripheral control registers within the module involved. Notice that interrupt pending status bits are also included in most cases to enable interrupt management to be achieved when the interrupt is temporarily prevented by its priority relationship, for example.

If these steps are addressed carefully a successful interrupt driven interface will be achieved.

6.2.2 *The Nested Vector Interrupt Controller (NVIC)*

The function of the Nested Vector Interrupt Controller (NVIC) is to sort out the priority of the current interrupt request and if it is accepted it will activate the CPU in its interrupt mode. The NVIC will be used with the ADC3 in the following example and the steps outlined previously can be seen in the following code. Figure 6.2 shows the important features of the NVIC diagrammatically.

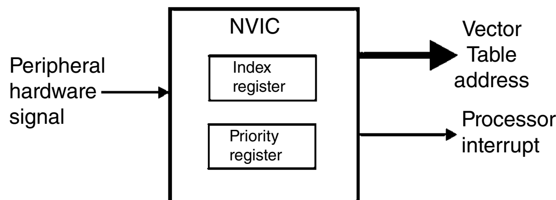


Figure 6.2 Part of the NVIC

Table 6.2 NVIC priority groups

NVIC group	Pre-emption bits	Sub-priority bits
0	0	4 (0–15)
1	1	3 (0–7)
2	2 (0–4)	2 (0–4)
3	3 (0–7)	1
4	4 (0–15)	0

The position in the vector table for ADC global interrupt is 0x0000 0088, this is calculated from the table base 0x0000 0040, the index position of the ADC global vector 18 (0x12) and four bytes per vector. Similarly the reference files show that the table entry for the TIM3 global vector is at index position 29 (0x1d) so its vector is at 0x0000 00b4. (Refer to files stm32f4_misc.h, stm23f4xx.h, stm32f4xx_it.c and startup_stm32f4xx.c in the ST support package.)

In addition to the hardware defined priority the user has an opportunity to select 1 of 16 priority levels for a particular interrupt source. Four additional bits in the NVIC define what are referred to as Pre-emption priority and Sub Priority relationships. The overall interrupt (IRQ) priority order is then sorted by highest to lowest priority as follows:

1. The lowest pre-emption priority
2. The lowest sub-priority
3. The lowest hardware priority, associated with the IRQ number.

The 4 bits allocated in the NVIC for these two user selectable relationships are split into five groupings, which are selected by using the library function NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0) that sets the required bits in the System Control register. The results of these groupings are shown in Table 6.2.

If the Priority Group is not initialised the priority is just taken as the raw value formed by combining the pre-emption and sub-priority fields. The grouping is not relevant when there are only one or two interrupt sources to handle.

```
   NVIC_InitTypeDef NVIC_InitStructure;
   NVIC_InitStructure.NVIC_IRQChannel = ADC_IRQn;
   NVIC_InitStructure.NVIC_IRQChannelPreemption
Priority = 0;
   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
```

```
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
```

The chosen ADC3 interrupt source will need to be configured and the module will need to be switched on.

```
    ADC_ITConfig(ADC3, ADC_IT_EOC, ENABLE);
    ADC_Cmd(ADC3, ENABLE);
```

The following function must appended to a copy of stm32f4xx_it.c, which also contains the exception handler templates, and saved in a file named along the lines of AtoD_it.c, which should then be included in the user group before compilation. The name of the function must conform to those given in startup_stm32f4xx.s, which sets up the vector table in memory when the code is compiled, that is `ADC_IRQHandler()` and `TIM3_IRQHandler()` in the case of timer TIM3, for example. A simplified ISR for the converter is shown here.

```
void ADC_IRQHandler(void)
{
    uint16_t temp;

    ITFlag = ADC_GetITStatus(ADC3, ADC_IT_EOC);

    /* Access ADC and store its value */
    temp = ADC_GetConversionValue(ADC3);

    ADC_ClearITFlag(ADC3, ADC_IT_EOC);
}
```

Once the ADC is started it will continue to generate interrupts every time a new value is ready. Don't forget to clear the flag so that it is ready to be set by the next conversion.

6.2.3 Exceptions

Processor exceptions are treated like interrupts and there are 16 possible exceptions that may be generated by the ARM CPU fault conditions that can arise during program execution; these are set out more extensively in the ARM-Cortex M4 documentation but will not be discussed in any detail here. It will be noted that the file stm32f4xx_it.c contains dummy functions to support these exceptions so that the user can provide their own handling routines if required.

6.3 Direct Memory Access (DMA)

DMA offers the most efficient way to move large volumes of data between interfaced peripheral modules and memory because it involves the minimum processor action to set up and no further interaction until the transfer is complete. Figure 6.3 summarises the main hardware concept.

The DMA controller was conceived to allow high speed interfaces such as Hard Disk bulk storage or Video output devices to have direct access to the system memory. In a simple processor as shown here the DMA controller would take over the address, data and control buses while the transfer proceeds. This means that the processor action must be suspended while the DMA controller is in operation or a bus conflict would arise. In early processor designs the DMA controller would issue a DMA request and the processor would respond with a DMA grant just before it suspended its action. When the data transfer came to an end the processor would be re-enabled. This is notably inefficient however as the processor action is stopped for significant periods while the DMA action takes over.

In a later development it was recognised that there are likely to be redundant bus cycles during normal program execution so it would be possible to interleaved DMA activity with processor activity. This would clearly be more efficient as the processor continues working but the maximum DMA rate might be compromised a little.

In the ARM processor the advanced bus matrix architecture allows the CPU to continue working with some resources while the DMA accesses others. This achieves the highest possible efficiency overall.

6.3.1 The STM32F4 DMA System

In the STM32F4 the DMA controller forms a powerful system resource and is used in order to provide efficient high-speed data transfer between peripherals and memory and between memory and memory if required. Data can be quickly moved by DMA without any CPU interaction leaving it free to process

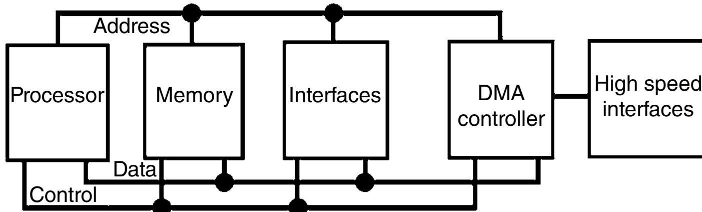


Figure 6.3 Simple processor with DMA

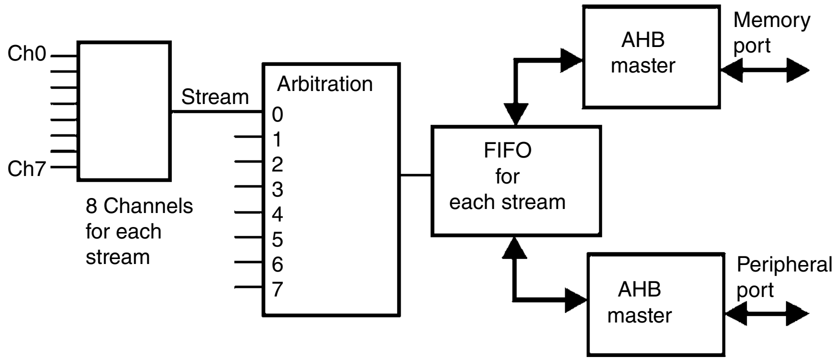


Figure 6.4 DMA controller architecture

application code. Based around the complex bus matrix architecture, the DMA controller combines a powerful master bus architecture based on dual AHBs with an independent FIFO to optimise the bandwidth of the system. A simplified block diagram is shown in Figure 6.4. The two DMA controllers each have eight streams (16 in total) enabling each of them to be dedicated to managing memory access requests from one or more peripherals. Each stream can have up to eight requests channels in total and each has an arbiter for handling the priority between DMA requests. Up to four priority levels can be assigned in the software to direct arbitration, very high, high, medium and low, but the hardware will determine the final choice in the case of equal priority situations, that is request 0 has priority over request 1, for example.

6.3.2 DMA Request Mapping

The various peripherals are mapped onto specific DMA channel requests by the hardware configuration of the STM32F4 device, specifically the digital-to-analogue converters (DACs) and ADCs are mapped as shown in Table 6.3, which is extracted from Tables 20 and 21 in the STM32F4 Reference Manual.

6.3.3 DMA Management

In general the DMA controllers can perform peripheral to memory and memory to peripheral transactions, also memory to memory transactions can be set up in the case of DMA2 because of its specific arrangement of connections to the bus matrix. A specified number of transactions take place and each DMA transaction follows a three stage process; firstly loading a word or byte

Table 6.3 DMA allocated channels

Peripheral	Controller	Stream	Channel
DAC1	DMA1	5	7
DAC2	DMA1	6	7
ADC1	DMA2	0 or 4	0
ADC2	DMA2	2 or 3	1
ADC3	DMA2	0 or 1	2

from peripheral register or memory location, then storing the data to peripheral register or memory location and finally decrementing the transaction counter so that it contains the number still left to complete. The DMA mode can be chosen to increment addresses in the source or destination as required and the circular mode can be used to handle continuous data flows or circular buffers. The waveform generation example in Chapter 5 used this configuration. When the circular mode is activated, the number of data items to be transferred is automatically reloaded with the initial value programmed during the stream configuration phase, and the DMA requests continue to be activated.

Each DMA stream has an independent four word FIFO that can be used to temporarily store data coming from the source before transmitting it to the destination. A software configurable threshold level, either 1/4, 1/2, 3/4 or full, is available to determine when the DMA request is finally initiated. The actual number of bytes transferred will depend whether bytes or words are being used in the particular application.

DMA transfer completion is indicated by the setting of the interrupt flag bit in the DMA interrupt status registers; this can be the result of different events depending on the DMA mode selected. If the DMA itself is providing flow control the flag is set when the transaction counter has reached zero in the memory to peripheral mode or when the stream is disabled before the end of the programmed transfer. Otherwise if the peripheral is providing flow control the flag is set when the last external data burst or single DMA request has been generated by the peripheral or the stream is disabled by a software command. In either case if the transfers involve the FIFO the flag is set when the remaining data has been transferred and the FIFO is again empty.

The interrupt flag can be inspected by the function shown here:

```
status = DMA_GetFlagStatus(DMA1_Stream0, DMA_FLAG_TCIF0);
```

There are good examples in the ST applications library that show the use of DMA in several different situations.

6.4 The LCD Display Module

The STM32F4DIS_LCD module uses a Solomon Semiconductors SSD2119 driver that provides an interface to a 320 by 240 TFT LCD matrix display on which the user can implement a wide range of designs. Each red, green or blue pixel is driven by a 6-bit value giving 262K ($2^6 \times 2^6 \times 2^6$) possible colour representations. The Discovery prototype board has the PS[3:0] bits hard wired to 0010 selecting a 8/16 bit interface using an historic protocol driven by write (WR) and read (RD). The data input port mapping shown on the data sheet, which is reproduced in Table 6.4, is distributed between GPIOD and GPIOE in an apparently rather awkward arrangement. One way to handle this is to set one bit at a time using a tabular driven procedure as illustrated next. The mapping is actually optimised to use the Flexible Static Memory controller (FSMC) and the ST Microelectronics example provided in the support package uses this approach.

```
/* set all 16 interface bits, split into two parts
low eight first */
void setdata_16(uint16_t data_16)
{
   /* low 8-bit assignments */
   GPIO_TypeDef* port_L8[] = {GPIOD, GPIOD, GPIOD,
GPIOD, GPIOE, GPIOE, GPIOE, GPIOE};
   uint16_t pin_L8[] = {GPIO_Pin_14, GPIO_Pin_15,
GPIO_Pin_0, GPIO_Pin_1, GPIO_Pin_7, GPIO_Pin_8,
GPIO_Pin_9, GPIO_Pin_10};
   /* high 8-bit assignments */
   GPIO_TypeDef* port_H8[] = {GPIOE, GPIOE, GPIOE,
GPIOE, GPIOD, GPIOD, GPIOD, GPIOD};
   uint16_t pin_H8[] = {GPIO_Pin_11, GPIO_
Pin_12,GPIO_Pin_13, GPIO_Pin_14, GPIO_Pin_15, GPIO_
Pin_8, GPIO_Pin_9, GPIO_Pin_10};

   int i;

   /* DC = 1 and RD = 1 */
   GPIO_SetBits(GPIOE, DC);
   GPIO_SetBits(GPIOD, RD);

   /* set low 8-bits */
   for(i = 0; i < 8; i++)
```

```
    {
       if (data_16 & 1)      /* test LSB */
       {
          GPIO_SetBits(port_L8[i], pin_L8[i]);
       }
       else
       {
          GPIO_ResetBits(port_L8[i], pin_L8[i]);
       }
       data_16 = data_16 >> 1;
    }

    /* set high 8-bits */
    for(i = 0; i < 8; i++)
    {
       if (data_16 & 1)
       {
          GPIO_SetBits(port_H8[i], pin_H8[i]);
       }
       else
       {
          GPIO_ResetBits(port_H8[i], pin_H8[i]);
       }
       data_16 = data_16 >> 1;
    }
    /* toggle WR to load data */
    GPIO_ResetBits(GPIOD, WR);
    TIM6_Wait(10);
    GPIO_SetBits(GPIOD, WR);
}
```

Interface timing characteristics (LCD data sheet section 14) show that the minimum time for WR low is 40 ns and if read is used the minimum time for

Table 6.4 Interface port mappings

Bit #	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**
GPIO Port	D10	D9	D8	E15	E14	E13	E12	E11
Bit #	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
GPIO Port	E10	E9	E8	E7	D1	D0	D15	D14

RD low is 500 ns. The WR low time is set at the end of the procedure mentioned previously using TIM6 giving a generous period of about 8 µs.

Before the interface can be utilised it must be initialised in the particular way described in the data sheet, basically turning the device on and waking it up from its sleep mode. Notice that there are two alternative interface modes that can be used. In the simplest mode the pixel RGB value is encoded in a single 16-bit word, that is 5 bits for R and B, 6 bits for G, giving a possibility of 64K different colours. In an alternative mode the RGB values are distributed between two consecutive words and 6 bits are allocated for each of the colours giving a possibility of all 262K possibilities! The initialisation code shown next sets important register values but most of these are actually the power-on reset (POR) values.

```
void display_on(void)
{
   uint8_t temp_8;
   uint16_t temp_16;

   /* set R07h with 0x0021 */
   temp_8 = 0x07;
   set_reg(temp_8);
   temp_16 = 0x0021;
   setdata_16(temp_16);

   /* set R00h with 0x0001 */
   temp_8 = 0x0;
   set_reg(temp_8);
   temp_16 = 0x0001;
   setdata_16(temp_16);

   /* set R07h with 0x0023 */
   temp_8 = 0x07;
   set_reg(temp_8);
   temp_16 = 0x0023;
   setdata_16(temp_16);

   /* set R10h with 0x0000  exit sleep mode */
   temp_8 = 0x10;
   set_reg(temp_8);
   temp_16 = 0x0000;
   setdata_16(temp_16);
```

```
    /* wait 30ms */
    TIM6_Wait(40000);

    /* set R07h with 0x0033 */
    temp_8 = 0x07;
    set_reg(temp_8);
    temp_16 = 0x0033;
    setdata_16(temp_16);

    /* set R11h with  TY1 = 1, TY0 = 0 for Type B
data format */
    temp_8 = 0x11;
    set_reg(temp_8);
    temp_16 = 0x42B0;            /* DFM bits 10 for
262k and TY bits 10 */
    setdata_16(temp_16);

    /* leave setting of R02h */
}
```

Writing information to the driver requires a two phase operation, first the 8 bit register address is provided while DC is set to zero and during the second phase 16 bit register data is provided while DC is set to one. It will be observed that an optimised code setting the lower 8 bits alone is provided by the function set_register(). Once the initialisation is complete the interface is ready to transfer pixel data this requires register 34 (0x22) to be selected first then the following data transfers contain the red, green and blue pixel values placed at specific points in the 16-bit word as shown in Table 6.5. The write sequence automatically increments the pixel address counter so each set of values set successive pixels across the page. The following code shows how the pixel colour values are encoded for this phases of the operation.

```
void ramdata_write(uint8_t red, uint8_t green,
uint8_t blue)
{
    /* set bits for R, G and B */
    data_16 = (red & 0x1f) << 10;
    data_16 = data_16 | ((green & 0x3f) << 5);
    data_16 = data_16 | (blue & 0x1f);
    setdata_16(data_16);
}
```

Table 6.5 Pixel assignments

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	R	R	R	R	G	G	G	G	G	G	B	B	B	B	B
4	3	2	1	0	5	4	3	2	1	0	4	3	2	1	0

For the full screen a sequence of 320×240 (76 800) consecutive write operations will be needed to complete the picture. Note that the interface controller provides an option to access a smaller screen window if required.

6.4.1 Character Generation

In many applications it will be desirable to establish a method to write alphanumeric symbols on the screen so a character generator will be needed to provide pixel values. In a typical case each character is described in terms of its pixel pattern using a 16 column by 24 row format. This fits conveniently into 24 consecutive words of 16-bit memory. Taking a capital A character, for example, will require only a few specific pixels to be activated and these patterns are encoded into 16-bit data as shown later. The blank rows below line 18 are reserved for the descending tails of characters, like g and y, for example.

The consecutive character codes can be described as a constant array in C and a complete set of 32 character codes is provided in the LCD library (see file fonts.c).

```
/* A */
const uint16_t coder[] = {0x0000, 0x0380, 0x0380,
0x06C0, 0x06C0, 0x06C0, 0x0C60, 0x0C60,
    0x1830, 0x1830, 0x1830, 0x3FF8, 0x3FF8, 0x701C,
0x600C, 0x600C,
    0xC006, 0xC006, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000,
    };
```

A complete character generator will take character and column as input and deliver a value which can be used to provide the row data. A typical character is shown in Figure 6.5.

To write a character each line must be assembled one pixel at a time then the cursor is moved one line down the screen and the process of assembly is repeated. As shown in the example next the assembly process uses a single bit

```
 1   ----------------   0×0000
 2   ------***-------   0×0380
 3   ------***-------   0×0380
 4   -----**-**------   0×06c0
 5   -----**-**------   0806c0
 6   -----**-**------   0×06c0
 7   ----**---**-----   0×0c60
 8   ----**---**-----   0×0c60
 9   ---**-----**----   0×1830
10   ---**-----**----   0×1830
11   ---**-----**----   0×1830
12   --***********---   0×3ff8
13   --***********---   0×3ff8
14   -***-------***--   0×701c
15   -**---------**--   0×600c
16   -**---------**--   0×600c
17   **-----------**-   0×c006
18   **-----------**-   0×c006
19   ----------------   0×0000
20   ----------------   0×0000
21   ----------------   0×0000
22   ----------------   0×0000
23   ----------------   0×0000
24   ----------------   0×0000
```

Figure 6.5 An upper-case A character

mask to test the character row value and determine whether the particular pixel should be white or black.

```
void DrawChar(uint16_t x_pos, uint16_t y_pos, const
uint16_t *c)
{
   uint16_t index, i;
   uint16_t x_addr, mask;

   x_addr = x_pos;
   /* set cursor start position */
   set_reg(0x4e);
   setdata_16(y_pos);
   set_reg(0x4f);
   setdata_16(x_addr);

   for (index = 0; index < 24; index++)
   {
      mask = 0x8000;
      set_reg(0x22);    /* prepare write to RAM */
      for (i = 0; i < 16; i++)
      {
```

```
            if ((c[index] & mask) == 0)
            {
                setdata_16(0);              /* BLACK */
            }
            else
            {
                setdata_16(0xFFFF);       /* WHITE */
            }
            mask = mask >> 1;
        }
        /* set next cursor start position */
        x_addr++;
        set_reg(0x4e);
        setdata_16(y_pos);
        set_reg(0x4f);
        setdata_16(x_addr);
    }
}
```

6.4.2 Parallel Interface

Although the interface described in the previous sections shows a simple and understandable structure a much more efficient parallel interface using the built-in FSMC is also possible and much more efficient in terms of the data handling requirement. The pin assignments are already set up for this mode of operation as shown by Table 7 in the STM32F4 Discovery data sheet. The important bus control signals provided by the FSMC are exactly matched to the requirements of the LCD:

Chip Select (CS) is driven by Not Enable 1 (NE1) as FSMC Bank 1 is selected,
Data/Control (DC) is driven by A19, a higher order address bit,
Read (RD) is driven by Not Output Enable (NOE),
and Write (WR) is driven by Not Write Enable (NWE)

Once the FSMC is configured it is only necessary to make assignments to the appropriate addresses to effect a register selection or data transfer operation.

```
#define LCD_CMD (*(uint16_t *)0x60000000)
#define LCD_DATA (*(uint16_t *)0x60100000)
```

```
LCD_CMD = LCD_Reg;
LCD_DATA = LCD_RegValue;
```

The example provided with the Discovery firmware support in the file stm32f4xx_discovery_lcd.c gives a working demonstration of this technique setting up the FSMC and using it to drive the LCD. The file also shows a wide range of useful character and simple graphic functions such as drawing lines and circles.

6.4.3 Touch Screen

The interface to the touch sensitive screen is provided via a special controller STMPE811 (see the resources provided in Further reading), this provides a built-in four-wire touch-screen interface using an enhanced movement tracking algorithm to avoid excessive data output and various other useful features such as a 8 bit GPIO extender, a 12 bit AD converter and a temperature sensor. The interface with the ARM chip is accomplished through the I2C interface, where transactions are passed to address 0x82 as the ADR0 pin is grounded.

The ARM cored STM32F4 device has an in-built I2C interface so this must be configured and enabled before any communication with the touch-screen interface can be established. The code shown next, as used for the examples in Chapter 5, enables the port pins involved in the two signals requires I2C_SCL and I2C_SDA and sets up the basic parameters for the I2C operation.

```
void I2C_InitPins(void)
{
   I2C_InitTypeDef I2C_InitStructure;
   GPIO_InitTypeDef GPIO_InitStructure;

   RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);
   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);

   /* set up B8 (SCL) and B9 (SDA) */
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 |
GPIO_Pin_9;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
   GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;

   GPIO_Init(GPIOB, &GPIO_InitStructure);
```

```
    GPIO_PinAFConfig(GPIOB, GPIO_Pin_8, GPIO_AF_I2C1);
    GPIO_PinAFConfig(GPIOB, GPIO_Pin_9, GPIO_AF_I2C1);
/* Note: the ST code module is incorrect so AFR[1]
is set directly */
    GPIOB->AFR[1] = 0x0044;

    I2C_InitStructure.I2C_ClockSpeed = 400000;
    I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
    I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;
    I2C_InitStructure.I2C_OwnAddress1 = 0x00;
    I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
    I2C_InitStructure.I2C_AcknowledgedAddress =
I2C_AcknowledgedAddress_7bit;

    I2C_Init(I2C1, &I2C_InitStructure);

    I2C_AcknowledgeConfig(I2C1, ENABLE);
    I2C_Cmd(I2C1, ENABLE);
}
```

Communication over the I2C interface must follow the correct procedure or protocol as described in Chapter 5. The basic sequence involves a start condition followed by the device address and register address.

```
void I2C_WriteDeviceReg(uint8_t DeviceAddr, uint8_t
RegAddr, uint8_t RegVal)
{
    I2C_GenerateSTART(I2C1, ENABLE);
    I2C_Send7bitAddress(I2C1, DeviceAddr, I2C_
DirectionTransmitter);
    I2C_GenerateSTOP(I2C1, ENABLE);
}
```

When I2C communication has been set up the data sheet suggests the following are the steps to initialise and configure the touch-screen controller (TSC):

1. Disable the clock gating for the TSC and ADC in the SYS_CFG2 register.
2. Configure the touch-screen operating mode and the window tracking index.
3. A touch detection status may also be detected by enabling the corresponding interrupt flag. With this interrupt, the user is informed when the touch is detected as well as when it is lifted again.

4. Configure the TSC_CFG register to specify the 'panel voltage settling time', touch detection delays and the averaging method to be used.
5. A windowing feature may also be enabled through TSCWdwTRX, TSCWdwTRY, TSCWdwBLX and TSCWdwBLY registers. By default, the windowing covers the entire touch panel.
6. Configure the TSC_FIFO_TH register to specify the threshold value to cause an interrupt. The corresponding interrupt bit in the interrupt module must also be enabled. This interrupt bit should be masked off during data fetching from the FIFO in order to prevent an unnecessary triggering of this interrupt. Upon completion of the data fetching, this bit can be re-enabled.
7. By default, the FIFO_RESET bit in the TSC_FIFO_CTRL_STA register holds the FIFO in Reset mode. Upon enabling the TSC (through the EN bit in TSC_CTRL), this FIFO reset is automatically de-asserted. The FIFO status may be observed from the TSC_FIFO_CTRL_STA register or alternatively through the interrupt.
8. Once the data is filled beyond the FIFO threshold value, an interrupt is triggered (assuming the corresponding interrupt has been enabled). The user is required to continuously read out the data set until the current FIFO content is below the threshold value, and then the user may clear the interrupt flag. As long as the current FIFO size exceeds the threshold value, an interrupt from the TSC is sent to the interrupt module. Therefore, even if the interrupt flag is cleared, the interrupt flag is automatically asserted, as long as the FIFO size exceeds the threshold value.
9. The current FIFO size can be obtained from the TSC_FIFO_SZ register. This information may assists the user in determining how many data sets are to be read out from the FIFO, if the user intends to read all in one shot. The user may also read one data set at a time.
10. The TSC_DATA_X register holds the X-coordinates. This register can be used in all touch-screen operating modes.
11. The TSC_DATA_Y register holds the Y-coordinates. TSC_DATA_Y register holds the Y-coordinates.
12. The TSC_DATA_Z register holds the Z value. TSC_DATA_Z register holds the Z-coordinates.
13. The TSCDATA_XYZ register holds the X, Y and Z values. These values are packed into 4 bytes. This register can only be used when the touch-screen operating mode is 000 and 001. This register is to facilitate fewer read operations and greater efficiency overall.
14. For the TSC_FRACT_Z register, the user may configure it based on the touch-screen panel resistance. This allows the user to specify the resolution

of the Z value. With the Z value obtained from the register, the user simply needs to multiply the Z value with the touch-screen panel resistance to obtain the touch resistance for the current position.

15. The TSC_DATA register allows an alternative reading format, with minimum I2C transaction overhead, by using the non-auto-increment mode (or an equivalent mode with the SPI interface). The data format is the same as TSC_DATA_XYZ, with the exception that all the data fetched are from the same address.
16. Enable the EN bit of the TSC_CTRL register to start the touch detection and data acquisition.
17. During the auto-hibernate mode, a touch detection can cause a wake-up to the device only when the TSC is enabled and the touch detect status interrupt mask is enabled.
18. In order to prevent confusion, it is recommended that the user does not mix the data fetching format (TSC_DATA_X, TSC_DATA_Y, TSC_DATA_Z, TSC_DATA_XYZ and TSC_DATA) between one reading and the next.
19. It is also recommended that the user should perform a FIFO reset and TSC disabling when the ADC or TSC settings are reconfigured.

After all these issues have been addressed the touch-screen should be ready for action and can be re-enabled through the SYS_CFG2 register.

6.5 The Wireless Interface Module

The Discover WiFi wireless module STM32F4DIS-WiFi provides ‘off the shelf’ serial to WiFi connectivity conforming to the 2.4 GHz IEEE802.11 standard that will be adequate for some applications (see Discover Wi-Fi UserManual_V1.5.pdf). It can also be used as a standalone WiFi station or network controller. The module uses the SN8200 from muRata which contains a controlling ARM processor and connects to the serial port USART1 on the Discovery board (TX on GPIOB pin 6 and RX on GPIOB pin 7) as shown in the diagram Figure 6.6. Note that an option to use an serial peripheral interface (SPI) is also included.

Extensive examples are provided in the Discovery support package which enables a wireless network to be established and a variety of operations to be tested. The software sets up communications sockets to enable a WiFi link to be established. The main functions of the demonstration software are listed in Table 6.6 and these are provided as a simple menu for the user to select from.

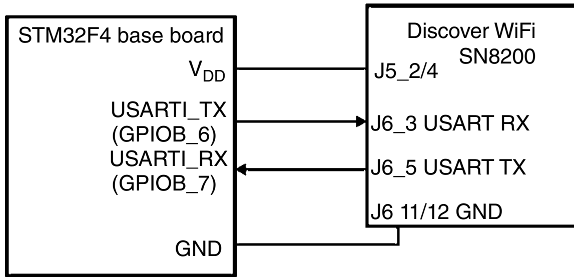


Figure 6.6 The WiFi module configuration

Table 6.6 Demonstration functions

0.	Get WiFi Status	10.	UDP Server
1.	WiFi Scan	11.	WiFi Off
2.	WiFi Join	12.	WiFi On
3.	Get IP	13.	HTTP get request
4.	TCP Client	14.	HTTP post request
5.	TCP Server	15.	HTTP post Json request
6.	Send from socket	16.	HTTP chunked post request
7.	WiFi Leave	17.	HTTS get request
8.	AP On/Off	18.	TLS (Transport Layer Security) Client
9.	UDP Client	19.	TLS Server (HTTPS Server)

Note that when using the Discovery base board the port connected to the 9-pin D socket is linked via USART6 whereas USART3 is used in the example code. Also the board rate for this link should be lower than suggested because USART6 is connected on the lower speed APB2. This involves some editing in the main code module only.

Selecting 1 (WiFi Scan) for example, delivers a message along the lines of the example here:

```
-WifiScan

SSID:       BTHomeHub-E45D CH:  5 RSSI: 171 Sec: 1
SSID:       BTOpenzone     CH:  5 RSSI: 171 Sec: 0
SSID:       SKYE9468       CH: 11 RSSI: 162 Sec: 4
```

Selecting menu item 2 will allow the user to choose the SSID channel and enter the password for the link. Further details of the other options are provided in the WiFi User manual.

6.6 Digital Camera Interface

A low cost digital camera module is available as part of the Discovery platform STM32F3DIS-CAM and the support package from ST gives an example showing how to use the digital camera interface (DCMI) module to control the OV9655 Camera board connected via the Discovery base board STM32F4DIS-BB.

The OV955 camera is a 1.3 MPixel low voltage CMOS device that provides the full functionality of a single-chip Super Extended Graphics Array (SXGA 1280 × 1024) camera and image processor in a physically small package. This camera has an image array capable of operating at up to 15 frames per second in SXGA resolution with complete user control over image quality, formatting and output data transfers. All required image processing functions are also programmable through the OmniVision Serial Camera Control Bus SCCB interface (like the I2C protocol).

In this example the DCMI, which is part of the STM32F4 device, is configured to communicate with the camera in continuous mode. The I2C1 module is used to configure the OV9655 camera in 8 bit RGB mode using 5:6:5 resolution. The user can select between two resolutions QQVGA(160 × 120) or QVGA(320 × 240) in order to display the captured image on the 320 × 240 LCD, this selection is performed in the 'main.h' file.

All camera data received by the DCMI are transferred through the DMA and displayed on the LCD panel that uses a parallel FSMC interface. As a result the CPU is released to execute other application tasks.

The flexible DCMI is a synchronous parallel interface that can receive high-speed data flows up to 54 Mbytes/s. It consists of up to 14 data lines (D13-D0) and a pixel clock line (PIXCLK). The pixel clock has a programmable polarity, so that data can be captured on either its rising or falling edge. A general block diagram of the DCMI is shown in Figure 6.7.

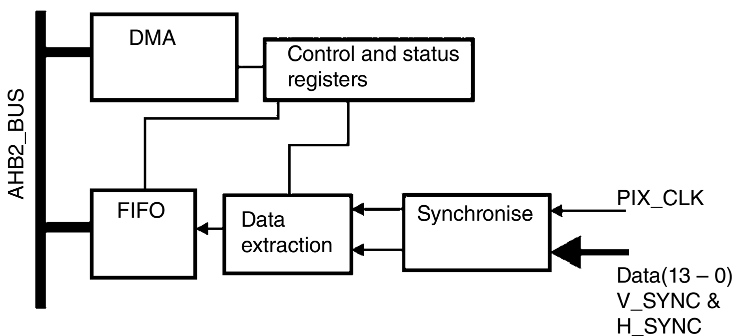


Figure 6.7 Digital camera interface (DCMI) configuration

Elements of the data stream are packed into a 32-bit data register (DCMI_DR) and then transferred through a general-purpose DMA channel. The image buffer in memory is managed by the DMA, not by the camera interface.

The data received from the camera can be organised in lines/frames (raw YUB/RGB/Bayer modes) or can be a sequence of JPEG images. To enable JPEG image reception, the JPEG bit (bit 3 of DCMI_CR register) must be set.

The data flow is synchronised either by hardware using the optional HSYNC (horizontal synchronisation) and VSYNC (vertical synchronisation) signals or by synchronisation codes embedded in the data flow.

6.7 Conclusion

This chapter has presented more detail about the Interrupt and DMA subsystems implemented in the STM32F4 device. Much more detail on both is provided in the ARM Reference Manual and this should be studied carefully before a new application is attempted although it is hoped that the examples provided will form a useable basis for development.

The LCD display example shows a more involved peripheral interface utilising the basic input and output facilities. It should be noted that a much more efficient interface is possible using the FSMC module and this is detailed as an example in the ST support package.

The WiFi interface is briefly introduced but the reader is recommended to download the support package and experiment with it to discover some of the features available.

The camera interface is also explained briefly as its support package covers the various aspects quite effectively.

Further Reading

The modules used in this chapter and their support software are all available from Farnell Electronics.

LCD Module STM32F4DIS-LCD:
SOLOMON SYSTECH SSD2119 TFT LCD Driver
ST Microelectronics STMPE811 resistive touch-screen controller
WiFi Module STM32F4-WiFi
Discover Wi-Fi UserManual_V1.5.pdf
Camera module STM32F4DIS-CAM:
OmniVision Serial Camera Control Bus (SCCB), www.ovt.com/download_document.php?type=document&DID=63 (accessed 10 October 2014).

7

Application Case Study Examples

This chapter will provide well documented case studies to illustrate the concepts described in earlier chapters and explain some useful concepts in C to process complex input data formats. Although these applications are based on the STM32F4 Discovery platform, many of the considerations discussed apply whatever platform is envisaged for a final product. So these examples should provide a useful resource for implementations on future products and applications. The particular applications that will be considered here include a Magnetic Compass module, an MSF time receiver and a simple GPS (Global Positioning System) navigator.

7.1 An Open-Loop Digital Compass

In this application a digital compass in the form of a Freescale Semiconductor MAG3110 Digital Magnetometer [1] is employed. A small module containing this sensor SEN-12670 is available from Sparkfun Electronics [2]. The MAG3110 sensor is actually a three-axis magnetometer and uses an I2C interface to control and access the three magnetic field sensors, which are mounted mutually at right angles to each other. Several suitable interface modules are available on the STM32F4 as there are no conflicting requirements but the

Digital Interface Design and Application, First Edition. Jonathan A. Dell.
© 2015 John Wiley & Sons, Ltd. Published 2015 by John Wiley & Sons, Ltd.

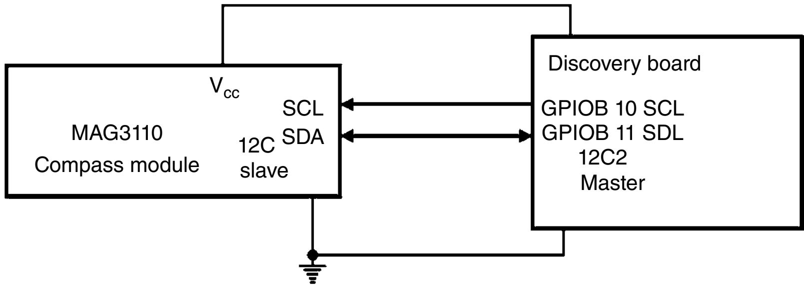


Figure 7.1 The compass interface

I2C2 module has been chosen in this case so the circuit arrangement is as shown in Figure 7.1.

7.1.1 Program Design

The program should start by initialising the two I/O port pins and configuring the I2C2 module. The main procedure then needs to follow the general scheme set out in the MAG3110 data sheet. To start with it shows that the I2C base address for the sensor is 0x0E so this needs to be part of the initialisation. To configure the sensor for action the sequence required is given in the follow notes.

A short summary of the register table provided in the MAG3110 product data sheet is given for reference in Table 7.1, those registers not listed here relate to the Z axis and user provided offset values that can be applied to each measurement axis when required by the application.

When CTRL_1 is 0x01 this gives the active mode and the highest sampling rate. When in standby mode setting the trigger immediate (TI) bit causes an immediate reading but in Active mode a measurement in progress will continue.

When CTRL_2 is set to 0xA0 the sensor is automatically reset before each reading (recommended) and the user set offsets are not applied.

The XYZDR bit can be polled to determine when new measurement data is available or more efficiently the interrupt can be used as this is linked to the state of the XYZDR bit.

7.1.2 Setting up the MAG3110

The MAG3110 data sheet describes many possible operating modes but in this case an example of triggered measurements with an output data rate (ODR) of 10 Hz and output sampling ratio (OSR) of 128 will be used.

Table 7.1 MAG3110 register summary

Register address	Name	Function
0x00	DR_STATUS	Per channel new data ready status Bit 4 XYZDR, Bit 1 YDR, Bit 0 XDR
0x01	OUT_X_MSB	Bits 15–8
0x02	OUT_X_LSB	Bits 7–0
0x03	OUT_Y_MSB	Bits15–8
0x04	OUT_Y_LSB	Bits 7–0
0x07	WHOAMI	Device ID (0xC4)
0x0F	DIE_TEMP	Signed 8-bit in °C
0x10	CTRL_1	Operating Mode Bit 0 Active Mode Bit 1 Trigger Immediate (TI) Bits 7–5 = data rate and oversampling ratio. All zero gives DR 80Hz and 16 sample average
0x11	CTRL_2	Operating Mode, Bit 5 RAW, Bit 4 reset sensor, Bit 7 Auto Reset

The register values required can be found in the data sheet and the following activation scheme should be applied.

1. Enable automatic magnetic sensor resets by setting bit 7 in CTRL_REG2. (CTRL_2 = 0x80)
2. Initiate a triggered measurement with OSR of 128 by writing 0x1A to CTRL_REG1 (CTRL_1 = 0x1A).
3. The MAG3110 will acquire the triggered measurement and go back into its STANDBY mode. It is possible at this point to synchronise with the data availability by polling the XYZDR bit status then the acquired data can be read out. Alternatively an interrupt can be activated when data becomes available.
4. When a new measurement is required go back to step 2 to initiate a new trigger.

When the I2C initialisation is complete a simple group of program statements will implement these requirements. The single byte data write function will accomplish the setting of CTRL_1 and CTRL_2, the singe byte read function will test the XYZDR status and the multi-byte read sequence will acquire the 6 bytes of measurement data. This automatically increments the register address between read operations.

```
    uint8_t MAG_XYZ[6];

    configure_GPIOB();
    configure_I2C1();

    /* configure MAG3110 */
    MAG_write_reg(0x0E, 0x12, 0x80);  /* CTRL_2 = 0x80 */
    MAG_write_reg(0x0E, 0x11, 0x1A);  /* CTRL_1 = 0x1A */

    while ((MAG_read_reg(0x0E, 0x00) & 0x08) == 0)  /*
read XYZDR */
    {
    }
    MAG_read_sixbyte(0x0E, 0x01, MAG_XYZ[]);/* read
X, Y and Z values */
```

The MAG_read and MAG_write functions are derived from the I2C functions described in Chapter 5. In particular the 6 byte read function follows the general format shown in Figure 5.5 and a coded implementation is given next. The code starts by following the steps of a single byte each read. Then after a restart multiple bytes of data are read from sequential registers automatically after each MAG3110 acknowledgement (ACK) is received until a no acknowledge (NAK) occurs from the master followed by a stop condition (STOP) signalling the end of transmission. As long as the fast read (FR) bit in CTRL_1 is clear, the complete 16-bit data for each measurement is read accessing all 6 bytes sequentially.

```
void MAG_read_sixbyte(uint8_t ADDR, uint8_t REGA,
uint8_t XYZ[])
{
    uint8_t i;
/* send start */
    I2C_GenerateSTART(I2C2, ENABLE);
    while(I2C_CheckEvent(I2C2, I2C_EVENT_MASTER_
MODE_SELECT) != 1)
    {}
    /* send device address */
    I2C_Send7bitAddress(I2C2, ADDR, I2C_Direction_
Transmitter);
    while(I2C_CheckEvent(I2C2, I2C_EVENT_MASTER_
TRANSMITTER_MODE_SELECTED) != 1)
    {}
```

```
    /* send register address */
*/
    I2C_SendData(I2C2, REGA);
    while(I2C_CheckEvent(I2C3, I2C_EVENT_MASTER_
BYTE_TRANSMITTED) != 1)
    {}
/* send RESTART */
    I2C_GenerateSTART(I2C2, ENABLE);
    while(I2C_CheckEvent(I2C2, I2C_EVENT_MASTER_
MODE_SELECT) != 1)
    {}
    I2C_Send7bitAddress(I2C2, ADDR, I2C_Direction_
Receiver);
    while(I2C_CheckEvent(I2C2, I2C_EVENT_MASTER_
RECEIVER_MODE_SELECTED) != 1)
    {}

/* read five bytes with normal ACK */
    for(i = 0; i < 5; i++)
    {
        while(I2C_CheckEvent(I2C2, I2C_EVENT_MASTER_
BYTE_RECEIVED) != 1)
        {}
        XYZ[i] = I2C_ReceiveData(I2C2);
    }
/* read last byte */
    I2C_NACKPositionConfig(I2C2, I2C_NACKPosition_
Next);
    while(I2C_CheckEvent(I2C2, I2C_EVENT_MASTER_
BYTE_RECEIVED) != 1)
    {}
    XYZ[5] = I2C_ReceiveData(I2C2);

    /* send stop */
    I2C_GenerateSTOP(I2C3, ENABLE);
}
```

Once the code modules have been set up the measurement data can be accessed as soon as a new measurement is available or whenever required if a triggered mode is being used.

7.1.3 Programming Challenge: A 360° Servo

Use the compass and a suitable actuator to implement a 360° servo system. The user will provide a desired heading and the servo should rotate automatically to match this requirement. Practically, this will require a mechanical construction to carry the compass module on an arm attached to the actuator motor and some care will be needed to accommodate the interconnection wiring.

7.2 The MSF Time Decoder

The MSF 60 kHz low frequency transmission from the Rugby station is the principle means of national distribution of standard time and frequency provided by the National Physical Laboratory (NPL). The transmitter power is quite high so that the signal strength is reliable across northern and western Europe. This application is based on a simple 60 kHz receiver (such as Model SYM-RFT-60 from PV Electronics [3]) that delivers a logic signal derived from the MSF Rugby transmission, which is modulated with time and date fields according to the specification in reference [3] and summarised in the following figures and tables.

Each minute is divided into 60 s except in the rare case when there is one extra when a leap second is required. The signal modulation format is shown in Figure 7.2.

Note that in this diagram high represents 0 bit data and low represents 1 bit data. The date and time is encoded in the A bits as shown in Table 7.2 and each field is encoded in BCD format.

British Summer time is indicated by bit B58, which is set during this period. More detail can be found directly from the NPL [4].

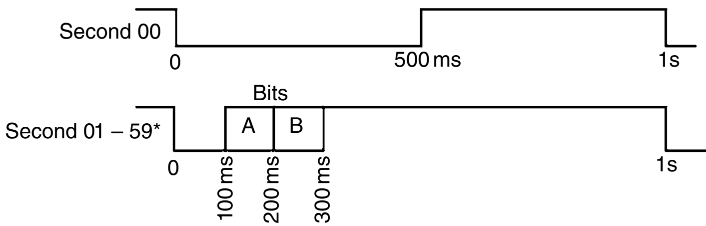


Figure 7.2 MSF signal modulation format

Table 7.2 MSF data bit allocations

A Bit #s	Weights	Group	Range
17–24	80, 40, 20, 10 8, 4, 2, 1	Year	00–99
25–29	10 8, 4, 2, 1	Month	01–12
30–35	20, 10 8, 4, 2, 1	Date	00–31
36–38	4, 2, 1	Weekday	0 (Sun) to 6 (Sat)
39–44	20, 10 8, 4, 2, 1	Hour	00–23
45–51	40, 20, 10 8, 4, 2, 1	Minute	00–59

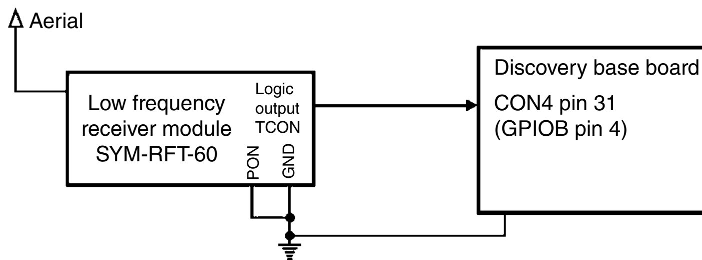


Figure 7.3 MSF receiver circuit

7.2.1 MSF Receiver Circuit Arrangement

The output from the MSF receiver can be connected to any GPIO input as long as it can be used to generate an interrupt, so GPIOB Pin 4 has been chosen for convenience as it is available on CON4 pin 31 on the Discovery board and uses a dedicated interrupt on the NVIC (Nested Vectored Interrupt Controller: see ST Microelectronics Reference Manual). The circuit is shown in Figure 7.3: Note that the output of this receiver is inverted compared with the earlier diagram Figure 7.2.

7.2.2 Program Design

The program design is centred on interrupts, which are triggered by the falling edge at each second boundary. A time delay is then used to select the strobe position for each data acquisition. In the zero second the data must

be low at 400 ms, for example and for the data bits 150 ms would place the strobe point in the centre of bit A and a further 100 ms (250 ms in total) would place the strobe point at the centre of bit B. The interrupt system should be re-enabled just before the start of the next second; this requires a further delay of 550 ms (950 ms in total) in the zero second and 700 ms (950 ms in total) after the data strobes in the following cycles. If the timer capture/compare registers are employed and the timer clock is set to 0.1 ms we can set ARR (auto reload register) to 9500, CC1 to 1500, CCR2 to 2500 and CCR3 to 4000. Every time the initial edge causes an interrupt further interrupts are disabled for the time being and the timer register is set to zero. When the timer reaches overflow the interrupt is re-enabled allowing the next edge to be detected.

7.2.3 Setting up for an Interrupt

To establish the interrupt system an external interrupt source (EXTI) is enabled using GPIOB pin 4 fall/rising edge to which the receiver output is connected. The code that follows shows how this is achieved. Note that EXTI line 4 has a dedicated interrupt service routine (ISR) name, see stm32f4xx.h.

```
void setup_EXTI(void)
{
   NVIC_InitTypeDef   NVIC_InitStructure;
   EXTI_InitTypeDef   EXTI_InitStructure;

   /* Enable SYSCFG clock */
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG,
ENABLE);

   /* Connect EXTI Line4 to PB4 pin */
   SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOB,
EXTI_PinSource4);

   /* Configure EXTI Line4 */
   EXTI_InitStructure.EXTI_Line = EXTI_Line4;
   EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
   EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_
Rising;
   EXTI_InitStructure.EXTI_LineCmd = ENABLE;
   EXTI_Init(&EXTI_InitStructure);
```

```
    /* Enable and set EXTI Line4 Interrupt to a low
priority */
    NVIC_InitStructure.NVIC_IRQChannel = EXTI4_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemption
Priority = 0x02;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

7.2.4 Acquiring the Data Bits

The related ISR that must be appended to the file stm32f4xx_it.c, as explained previously, is shown next. The function name must correspond with the definition is startup_stm32f4.s. Global variables, although not recommended, were set up for the variables data_bitA[], data_bitB[], second and first_second_captured to simplify the overall control. These variables must be declared outside main and referred to as extern in any other associated files. The first objective of the ISR is to capture the zero second then to acquire the data bits in following seconds. In all cases the process that follows the interrupt from the leading edge starts by disabling further interrupts by clearing the interrupt mask register (IMR) bit, re-enabling it just before the next second is about to begin. This avoids any spurious interrupts that could possibly occur during this rather long period. The interrupt is re-enabled by setting the IMR bit. Note that there is no built-in function for controlling the IMR so the user has to manipulate it directly. Examine the actual statements to see how this is achieved, remembering that the '~' character means bitwise complement. The interrupt pending bit must also be cleared to prevent the possibility of further interrupts from this edge. This mechanism is implemented for management when interrupts are temporarily blocked by a higher priority event.

TIM3 is then prepared as it remains in control for most of the following second. The first if statement captures the zero second as soon as it is first detected after switch on. Once synchronised, the data bits are accumulated from each following second by testing the receiver input at the two strobe points determined by CCR1 and CCR2, taking account of the data polarity shown earlier. The receiver input is also tested at the third strobe point determined by CCR3 because when this is at logical zero it will detect the next zero second when it eventually occurs. The second counter is reset to zero at this point. This will automatically take account of leap seconds, should one be added.

```
void EXTI4_IRQHandler(void)
{
    EXTI->IMR &= ~EXTI_Line4;    /* disable interrupt */
    EXTI_ClearITPendingBit(EXTI_Line4);

/* prepare TIM3 */
    TIM_SetCounter(TIM3, 0);
    TIM_ClearFlag(TIM3, TIM_FLAG_CC1);
    TIM_ClearFlag(TIM3, TIM_FLAG_CC2);
    TIM_ClearFlag(TIM3, TIM_FLAG_CC3);
    TIM_ClearFlag(TIM3, TIM_FLAG_Update);

    if (first_second_captured == 0)
    {
            while(TIM_GetFlagStatus(3, TIM_FLAG_CC3)
!= 1)  /* wait on CCR3 */
        {
        }
        TIM_ClearFlag(TIM3, TIM_FLAG_CC3);
        if (test_data() != 1)      /* should be low
for second 0 */
        {
            first_second_captured = 1;
            second++;
        }
        else
        {
            second = 0;    /* any non zero i.e. 1 - 59 */
        }
    }
    else          /* acquire data bits for A and B */
    {
        while(TIM_GetFlagStatus(TIM3, TIM_FLAG_CC1)
!= 1) /*wait for CCR1*/
        {
        }
        TIM_ClearFlag(TIM3, TIM_FLAG_CC1);
        if (test_data() != 0)
        {
            data_bitA[second] = 0;
        }
```

```
        else
        {
            data_bitA[second] = 1;
        }
        while(TIM_GetFlagStatus(TIM3, TIM_FLAG_CC2) != 1)
/*wait for CCR2*/
        {
        }
        TIM_ClearFlag(TIM3, TIM_FLAG_CC2);
        if (test_data() != 0)
        {
            data_bitB[second] = 0;
        }
        else
        {
            data_bitB[second] = 1;
        }
        /* check for next zero second */
        while(TIM_GetFlagStatus(TIM3, TIM_FLAG_CC3)
!= 1)   /*wait for CCR3*/
        {
        }
        TIM_ClearFlag(TIM3, TIM_FLAG_CC3);
        if (test_data() == 0)
        {
            second = 0;
            decode_MSFdata(ydt_vals);
            printMSF_ydt(ydt_vals);
        }
        STRING_PRINT("\b\b");
        BCD_PRINT(second);
        while(TIM_GetFlagStatus(TIM3, TIM_FLAG_Update)
!= 1) /* timer overflow? */
        {
        }
        TIM_ClearFlag(TIM3, TIM_FLAG_Update);
        second++;
    }
    EXTI->IMR |= EXTI_Line4; /* re-enable interrupt */
}
```

```
/* read input bit on GPIOB Pin 4 include inversion */
uint8_t test_data(void)
{
   if(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_4) != 0)
   {
      return 0;
   }
   else
   {
      return 1;
   }
}
```

The timer settings according to the calculations shown previously are shown next. The prescaler is set to 8400 giving a 0.1 s timer clock period.

```
void setup_TIM3(void)
{
   TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;

   /*TIM3 Clock Enable */
   RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

   /* Time Base Configuration */
   TIM_TimeBaseStructure.TIM_Period = 9500;
   TIM_TimeBaseStructure.TIM_Prescaler = 8400;
   TIM_TimeBaseStructure.TIM_ClockDivision = 0;
   TIM_TimeBaseStructure.TIM_CounterMode = TIM_
CounterMode_Up;

   TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);

   TIM_Cmd(TIM3, ENABLE);
}

void setup_Compare(uint16_t CCR1_Val, uint16_t
CCR2_Val, uint16_t CCR3_Val)
{
   TIM_OCInitTypeDef TIM_OCInitStructure;

   TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Timing;
   TIM_OCInitStructure.TIM_OutputState = TIM_
OutputState_Enable;
```

```
    TIM_OCInitStructure.TIM_Pulse = CCR1_Val;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_
OCPolarity_High;

    TIM_OC1Init(TIM3, &TIM_OCInitStructure);

    TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Disable);

    TIM_OCInitStructure.TIM_Pulse = CCR2_Val;

    TIM_OC2Init(TIM3, &TIM_OCInitStructure);

    TIM_OC2PreloadConfig(TIM3, TIM_OCPreload_Disable);

    TIM_OCInitStructure.TIM_Pulse = CCR3_Val;

    TIM_OC3Init(TIM3, &TIM_OCInitStructure);

    TIM_OC3PreloadConfig(TIM3, TIM_OCPreload_Disable);
}
```

7.2.5 Decoding the MSF Data

The received data bits are first decoded into integer values according the weightings assigned and these are stored in a date and time structure defined in the top module. The digits are read MSB first and the corresponding weight is accumulated when the bit is 1. A pointer to the structure is used so that the values are retained and the arrays of bit data need an `extern` declaration if this function is in a separate file.

```
extern char data_bitA[];
extern char data_bitB[];

void decode_MSFdata(struct MSF_Time *ytd_vals)
{
    year_weight[] = {80, 40, 20, 10, 8, 4, 2, 1};
    month_weight[] = {10, 8, 4, 2,1};
    date_weight[] = {20, 10, 8, 4, 2, 1};
    day_weight[] = {4, 2,1};
    hour_weight[] = {20, 10, 8, 4, 2, 1};
    minute_weight[] = {40, 20, 10, 8, 4, 2, 1};

    int year, month, date, day, hour, minute;

    index = 0;
```

```
/* decode year */
    i = 0;
    ytd_vals->year = 0;
    for (index = 17; index < 25; index++)
    {
        if (data_bitA[index] != 0)
        {
            ytd_vals->year = ytd_vals->year +
year_weight[i++];
        }
        else
        {
            i++;
        }
    }
/* decode month */
    i = 0;
    ytd_vals->month = 0;
    for (index = 25; index < 30; index++)
    {
        if (data_bitA[index] != 0)
        {
            ytd_vals->month = ytd_vals->month +
month_weight[i++];
        }
        else
        {
            i++;
        }
    }
/* decode date */
    i = 0;
    ytd_vals->date = 0;
    for (index = 30; index < 36; index++)
    {
        if (data_bitA[index] != 0)
        {
            ytd_vals->date = ytd_vals->date +
date_weight[i++];
```

```
        }
        else
        {
            i++;
        }
    }

/* decode day */
    i = 0;
    ytd_vals->day = 0;
    for (index = 36; index < 39; index++)
    {
        if (data_bitA[index] != 0)
        {
            ytd_vals->day = ytd_vals->day +
day_weight[i++];
        }
        else
        {
            i++;
        }
    }

/* decode hour */
    i = 0;
    ytd_vals->hour = 0;
    for (index = 39; index < 45; index++)
    {
        if (data_bitA[index] != 0)
        {
            ytd_vals->hour = ytd_vals->hour +
hour_weight[i++];
        }
        else
        {
            i++;
        }
    }

/* decode minute */
    i = 0;
```

```
    ytd_vals->minute = 0;
    for (index = 45; index < 52; index++)
    {
        if (data_bitA[index] != 0)
        {
            ytd_vals->minute = ytd_vals->minute +
minute_weight[i++];
        }
        else
        {
            i++;
        }
    }
}
```

7.2.6 Displaying the MSF Time Data

Once the bits are decoded a simple group of print operations can deliver the information to the hyper terminal interface. A pointer to the day and month names are provided by simple functions as shown and the STRING_PRINT() function using USART6 is included for completeness. The USART6 module and its I/O connections will also have to be set up as shown before.

```
void printMSF_ydt(struct MSF_Time *ydt)
{
    STRING_PRINT("MSF Time Decoder\r\n\n");

    STRING_PRINT("Year 20");
    BCD_PRINT(ydt->year);
    STRING_PRINT("\r\n");
    STRING_PRINT(day_name(ydt->day));
    STRING_PRINT(" ");
    STRING_PRINT(month_name(ydt->month));
    STRING_PRINT(" ");
    BCD_PRINT(ydt->date);
    STRING_PRINT("\r\n");
    BCD_PRINT(ydt->hour);
    STRING_PRINT(" : ");
    BCD_PRINT(ydt->minute);
    STRING_PRINT(" : ");
```

```
    BCD_PRINT(ydt->second);
    STRING_PRINT("\r\n");
}

void BCD_PRINT(int val)
{
    int i, bcd, mask = 1;
    int bcd1, bcd10;
    int mult[] = {1, 2, 4, 8, 0x16, 0x32, 0x64, 0x128};
    char string[10];

    bcd = 0;
    for (i = 0; i <= 8; i++)
    {
        if ((val & mask) != 0)
        {
            bcd = bcd_add(bcd, mult[i]);
        }
        mask = mask << 1;
    }
    bcd1 = bcd & 0x0f;
    bcd10 = (bcd & 0xf0) >> 4;

    string[0] = bcd10 + '0';
    string[1] = bcd1 + '0';
    string[2] = 0;             /* NULL termination */
    STRING_PRINT(string);
}

char *day_name(int day)
{
    static char *name[] = {
        "Sunday", "Monday", "Tuesday",
        "Wednesday", "Thursday", "Friday",
        "Saturday"};

        return name[day];
}

char *month_name(int month)
{
    static char *name[] = {
        "Illegal month", "January", "February", "March",
        "April", "May", "June", "July",
```

```
        "August", "September", "October",
        "November", "December"};

        return name[month];
}

int bcd_add(int a, int b)
{
    int bcd;

    bcd = a + b;

    if ((bcd & 0x0f) > 9)
    {
        bcd = bcd + 6;    /* adjust value */
    }

    return bcd;
}

void STRING_PRINT(char *p)
{
    int k = 0;

    while (p[k] != 0)
        {
            char_print(p[k++]);
        }
}

void char_print(char p)
{
/* output to serial interface on UART6 */
        /* wait for TXE */
        while (USART_GetFlagStatus(USART6, USART_
FLAG_TXE) != 1)
        {
        }
        USART_SendData(USART6, p);
        while (USART_GetFlagStatus(USART6, USART_
FLAG_TC) != 1)
        {
        }
}
```

7.3 Decoding GPS Signals

Many companies supply integrated GPS modules that contain a receiver for the satellite transmissions and a tracker/decoder that delivers one of the standard navigation message structures, NMEA (National Marine Electronics Association) or SiRF binary format. A suitable GPS module for evaluation EB056 is available from Matrix Technology Solutions Ltd, and this uses the Fastrax UP500 module that acquires signals from multiple orbiting satellites to calculate its current position [5]. Once an initial position (fix) has been acquired the module continues to send out navigation messages. This example will use the NMEA format as this delivers simple messages composed of ASCII characters.

The EN056 module is connected to the Discovery base board as shown in Figure 7.4 to link with an available USART interface.

After the initial fix the GPS receiver provides a constant stream of data as it delivers navigation information about the 3D location from the satellites currently in view. If the receiver is set in NMEA mode, the only format for the UP500, various navigation messages formed into simple ASCII strings are available and the user can select those that are relevant to the information required. An example of one of these is shown next, this contains both time and navigation data.

$GPGGA,161229.487,3723.2475,N,12157.3416,W,1,07,1.0,9.0,M,47.4,M,,*18

The string is broken into fields by the ',' character and the first field $GPGGA determines the type of message. Other message strings that may be encountered are $GPRMC (minimum navigation data), $GPGSA (overall satellite reception data) and $GPGSV (detailed satellite data) and they each have specific formats.

For the $GPGGA message the following fields are detailed in Table 7.3.

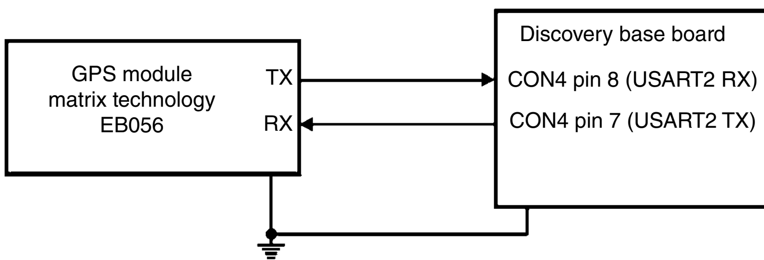


Figure 7.4 GPS module connection

Table 7.3 GPGGA navigation message format

Index	Function	Format	Example
1	Time	hhmmss.sss	161229.487
2	Latitude	ddmm.ssss	3723.2475
3	Direction	N or S	N
4	Longitude	dddmm.ssss	12157.3416
5	Direction	E or W	W
6	Fix Quality	1–8 (1 GPS) (0 invalid)	1
7	# satellites	One or two digits	07
8	Horizontal DOP	—	1.0
9	Height, M	Height above mean sea level	9.0,M
10	Geoid height, M	Above datum	47.4,M
11	Empty	—	—
12	Empty	—	—
13	*checksum	Two digits	*18

7.3.1 Acquiring the GPS Message

The GPS module is interfaced through an RS232 connection and USART2 was chosen for this link, its TX line is on GPIOD pin 5 and the RX line is on GPIOD pin 6. Once the USART is set up and its pins connected, the whole GPS messages can be acquired by a straightforward function that waits for the starting $ and a while loop that continues until the terminating * is encountered. The get_NMEA_char() function simply reads the next character on the USART2 interface as soon as it is ready. As the GPS module is usually set to deliver several types of message it may be more appropriate to test for the message type, that is $GPGGA, which is in the first field in the message string. Code performing this operation is included at the start of the message decode function discussed later.

```
    int i, j, k, started;
    char message_string[100];

    i = 0;
    k = 0;
    started = 0;
    /* read GPS until $ is found */
    while (started != 1)
    {
        message_string[i] = get_NMEA_char();
/*test_string[k++];*/
```

```
        if (message_string[i] != '$')
        {
            started = 0;
        }
        else
        {
            started = 1;
        }
    }
    while (message_string[i] != '*')      /* terminal
character */
    {
        message_string[++i] = get_NMEA_char();
/*test_string[k++];*/
    }
    message_string[i + 1] = 0; /* string termination
*/

char get_NMEA_char(void)
{
    char input_char;
    while (USART_GetFlagStatus(USART2, USART_FLAG_
RXNE) != 1)
    {
    }
    input_char = USART_ReceiveData(USART2);
    return(input_char & 0x7F);
}
```

7.3.2 Decoding the GPS Message

To split up the message into its different fields is fairly straightforward by noting each field separation character and using a case statement for successive fields as shown next. Only the fields one to seven have been processed in this example but this could easily be extended to others if required. The initial string is checked against the desired message type $GPGGA and each of the extracted strings are terminated with a NULL. The data structure GPGGA is assembled to contain all the separate fields conveniently and allow further field decode functions to be developed.

```
typedef struct parameter_strings
{
   char data_key[10];
   char time[11];
   char latitude[10];
   char latNSW[10];
   char longitude[10];
   char longEW[10];
   char fix[10];
   char sats[10];
}GPGGA;

void decode_main(GPGGA *ptr, char NMEA_string[])
{
   char str_buf[10];

   int i, j, k;

   i = 0;
   j = 0;
   k = 0;

   /* copy key to local buffer */
   while (NMEA_string[i] != ',')
   {
      ptr->data_key[i] = NMEA_string[i];
      i++;
   }
   ptr->data_key [i] = 0;      /* null termination
for first string */
   j++;
   if (strcmp(ptr->data_key, "$GPGGA") == 0)
/* message match? */
   {
      i++;
      while (j < 8)
      {
         k = 0;
         while (NMEA_string[i] != ',')
         {
            switch (j)
            {
```

```
                case 1 :
                    ptr->time[k++] = NMEA_string[i++];
                    break;
                case 2 :
                    ptr->latitude[k++] = NMEA_string[i++];
                    break;
                case 3 :
                    ptr->latNS[k++] = NMEA_string[i++];
                    break;
                case 4 :
                    ptr->longitude[k++] = NMEA_string[i++];
                    break;
                case 5 :
                    ptr->longEW[k++] = NMEA_string[i++];
                    break;
                case 6 :
                    ptr->fix[k++] = NMEA_string[i++];
/* fix status */
                    break;
                case 7 :
                    ptr->sats[k++] = NMEA_string[i++];
/* number x or xx */
                    break;
                default :
                    break;
                }
            }
            j++;
            i++;
        }
                  /* complete NULL terminations */
        ptr->data_key[6] = 0;
        ptr->time[10] = 0;
        ptr->latitude[9] = 0;
        ptr->latNS[1] = 0;
        ptr->longitude[9] = 0;
        ptr->longEW[1] = 0;
        ptr->fix[1] = 0;
        ptr->sats[k] = 0;   /* accounts for 1 or 2
characters */
```

```
    print_strings(ptr);
    }                           /* end of if $GGPA */
}
```

Once the message is split up in this way the various fields can be converted into numeric values if required. For example, taking latitude the first two characters represent degrees, the next five (two before and three after the decimal point) represent minutes of a degree as units and fraction so a floating point calculation would be needed to establish a real value. The three characters after the decimal point could be represented as a value for the seconds, by multiplying by 60 of course.

7.3.3 Selecting the Massage Stream

The GPS message stream can be tailored to specific needs by delivering short ASCII formatted messages to the module. For example, the group of strings shown here, turn off all streams except GPGGA. In each case the checksum is calculated carefully as the GPS module checks it on reception.

```
char  VTGOFF[] = "$PSRF103,05,00,00,01*21\r\n";
char  RMCOFF[] = "$PSRF103,04,00,00,01*20\r\n";
char  GSVOFF[] = "$PSRF103,03,00,00,01*27\r\n";
char  GSAOFF[] = "$PSRF103,02,00,00,01*26\r\n";
char  SETGGA[] = "$PSRF103,00,00,01,01*25\r\n";

    send_NMEA_string(VTGOFF);
    send_NMEA_string(RMCOFF);
    send_NMEA_string(GSVOFF);
    send_NMEA_string(GSAOFF);
    send_NMEA_string(SETGGA);
```

The function shown next, send_NMEA_string(), uses the TX output from USART2 to deliver the message to the module in a similar way to previous examples.

```
void send_NMEA_string(char *pt)
{
    uint16_t i;

    i = 0;
```

```
    while(pt[i] != 0)    /* continue until a NULL
termination is reached */
    {
        /* wait for TXE */
        while (USART_GetFlagStatus(USART2, USART_
FLAG_TXE) != 1)
        {
        }
        USART_SendData(USART3, pt[i++]);
        while (USART_GetFlagStatus(USART2, USART_
FLAG_TC) != 1)
        {
        }
    }
}
```

7.4 Conclusion

The code modules detailed in this chapter are provided to give further examples of the interface techniques discussed in earlier chapters and some useful extensions based on C programming techniques. Although these examples are incomplete programs, it should only require a little effort to construct a complete and useable application.

A themed approach has been taken to each of the chapters to link theory and practice in each subject area and promote a clear and well justified understanding. However, many embedded systems use a variety of different interface techniques simultaneously so the resources utilised need to be planned carefully to ensure that conflicts do not arise. In many cases the interface modules can be connected through different physical pins so physical conflicts can often be resolved simply by reallocation to an alternative set.

All the code modules discussed in these chapters have been evaluated on the STM32F4-Discovery kit using the Keil uVision IDE platform and rely on the interface drivers provided. In several sections the possibility of using Assembler language to achieve greater efficiency has been outlined but it should be clearly understood that the risks of using this approach outweigh the advantages in all but the most demanding situations.

Lastly, good luck with your applications using the STM32F4 ARM based device in particular but please appreciate that the principles discussed in these chapters have application to any embedded system design whatever core processor is chosen.

References

[1] Freescale Semiconductor, Inc. (n.d.) MAG3110 Data Sheet www.freescale.com/files/sensors/doc/data_sheet/MAG3110.pdf (accessed 11 October 2014).
[2] SparkFun Electronics (n.d.) SEN-12670 www.sparkfun.com/ (accessed 22 October 2014).
[3] PV Electronics (n.d.) www.pvelectronics.co.uk (accessed 22 October 2014).
[4] NPL Time & Frequency Services (n.d.) MSF Time Code, www.npl.co.uk/upload/pdf/MSF_Time_Date_Code.pdf (accessed 1 October 2014).
[5] Matrix Technology Solutions (n.d.) GPS Prototype Module, www.matrixtsl.com/eblocks.php (accessed 1 October 2014).

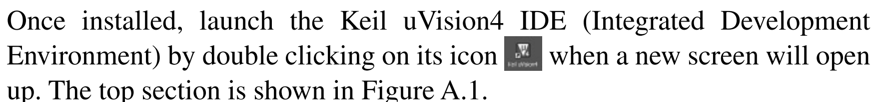
Appendix A

uVision IDE Notes

This appendix provides a brief introduction to the Keil uVision4 IDE for developing a project based on the STM32F4, Editing, Compiling and Downloading code to the Discovery prototype board. The uVision software and the STM32F4 support files can be downloaded from the web by going to:

www.keil.com/uvision/ide_ov_starting.asp for uVision and
www.farnell.com following links to STM32F4 Discovery

A.1 Getting Started

Once installed, launch the Keil uVision4 IDE (Integrated Development Environment) by double clicking on its icon  when a new screen will open up. The top section is shown in Figure A.1.

To get started you might expect to open a new project but it is strongly recommended that you use the examples provided by ST Microelectronics. It would be best to copy the whole project file from the downloaded example directories into your work area and rename it as appropriate. If you use finder to locate the downloaded directories you should find the directory STM32F4-Discovery_FW_V1.1.0\Project\Peripheral_Examples\TIM_ComplementarySignals,

Digital Interface Design and Application, First Edition. Jonathan A. Dell.
© 2015 John Wiley & Sons, Ltd. Published 2015 by John Wiley & Sons, Ltd.

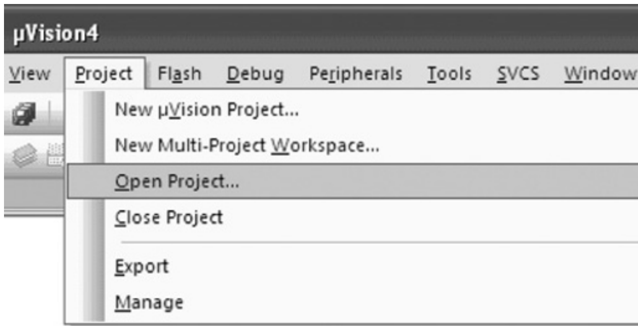


Figure A.1 The uVision screen (part)

which contains source files for one of the simplest application examples. Go to the directory MDK-ARM and in there you will see TIM_ComplementarySignals. uvproj, which contains the environment settings that will be needed.

Once copied you can open the project file by double clicking its new name 'filename.uvproj' and it should bring up the uVision IDE automatically. Proceed to open and modify the source files as required but don't forget to save your work at the end of an editing session.

A.2 Help

Help can be obtained at any time through a complete set of online, searchable manuals through the uVision4 environment. This is illustrated in Figure A.2.

uVision4 also provides context-sensitive help while you are editing your source file. You may request help for most keywords, standard library routines, directives and instructions. Select or click on the name you want help with and press F1. uVision4 will open the help file and locate the item required. In this case the C function getkey() is highlighted in Figure A.3.

A.3 Project Development

When uVision is running you will observe that a file hierarchy has been established in the project management area but you are quite free to change this later to incorporate new or other files if you wish (Figures A.4 and A.5).

On the drop-down window under Project you will find the entry 'Build Target …' this will compile your code, as long as no errors are found and it

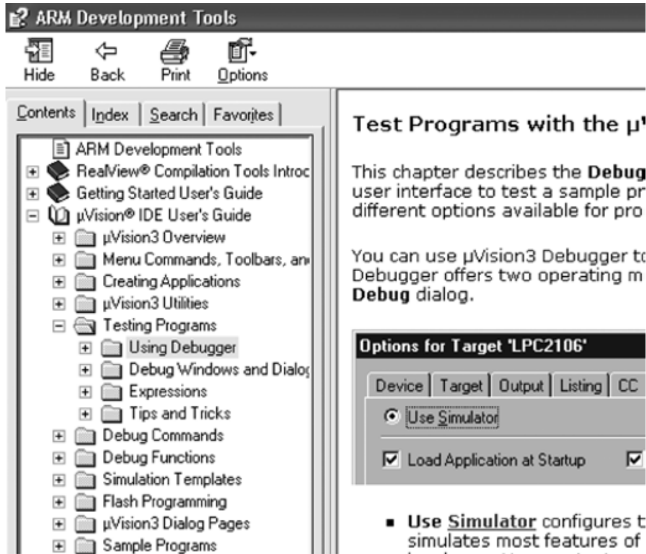


Figure A.2 A typical help screen

Figure A.3 Context sensitive help

can access all the files that are needed, in preparation for downloading to the target board.

You can now download the code into the target and observe the action of your code in real time. Using a logic analyser or an oscilloscope you can monitor the outputs and confirm that the selected pins deliver the correct waveform and timing.

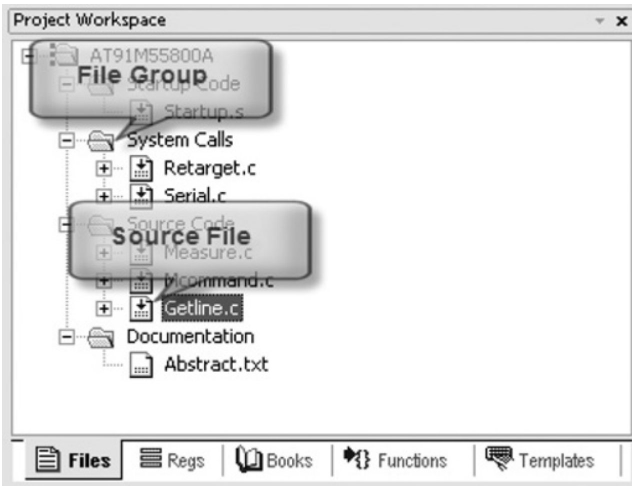


Figure A.4 Project file structure

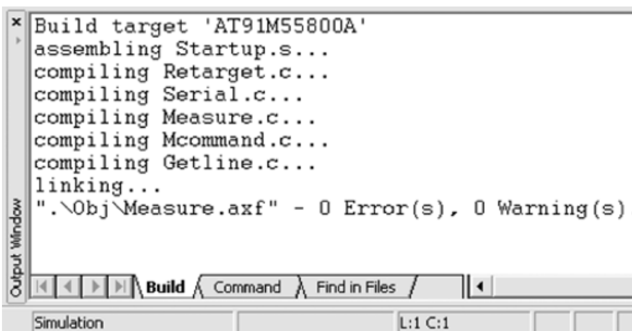


Figure A.5 Project compiling and linking

A.4 Debug Facilities

The Keil uVision IDE provides an extensive debug and simulation environment which you can use to test out your program and its interaction with the peripherals. Debug runs interactively with the hardware so you can still observe outputs that take new levels and changes that occur to variables, and so on.

To activate Debug press the ‘d’ button towards the right hand end of the toolbar and you will see the windows change dramatically. You will normally observe a two widow display consisting of your C code in the lower window and the compiled assembler code in the upper window. A typical view of the two debug screens is shown in Figure A.6.

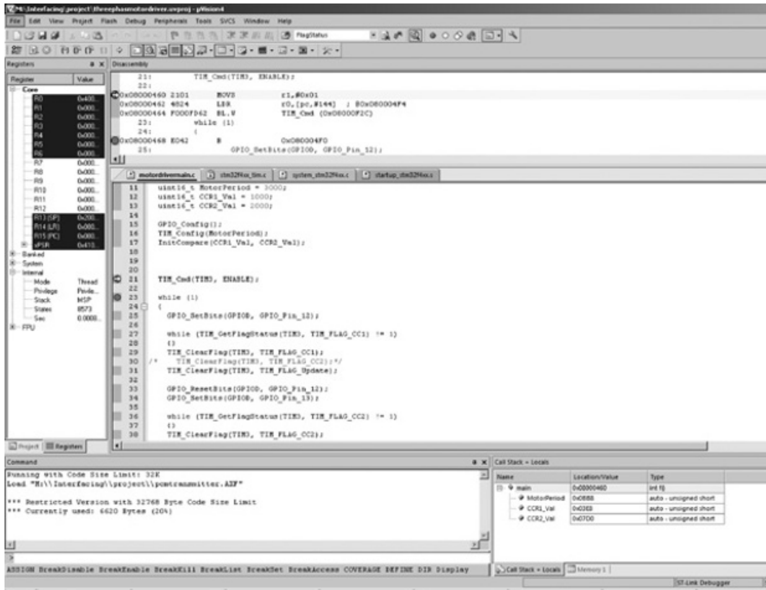


Figure A.6 A debug run

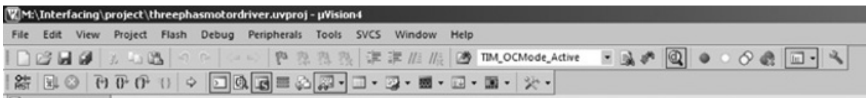


Figure A.7 Debug tools

Breakpoint and single step are some of the most powerful tools available, these and other techniques can be selected through the buttons and drop-down menus on the new toolbar along the top edge of the code windows, or the bottom row of the partial diagram as shown in Figure A.7.

The main debug tools are listed in the table:

Reset	Step into	Show next statement	Register window	Serial window
Run	Step over	Command window	Call stack	Analysis window
Stop	Step out	Disassembly window	Watch widow	Trace window
	Run to cursor	Symbol window	Memory window	System Viewer window

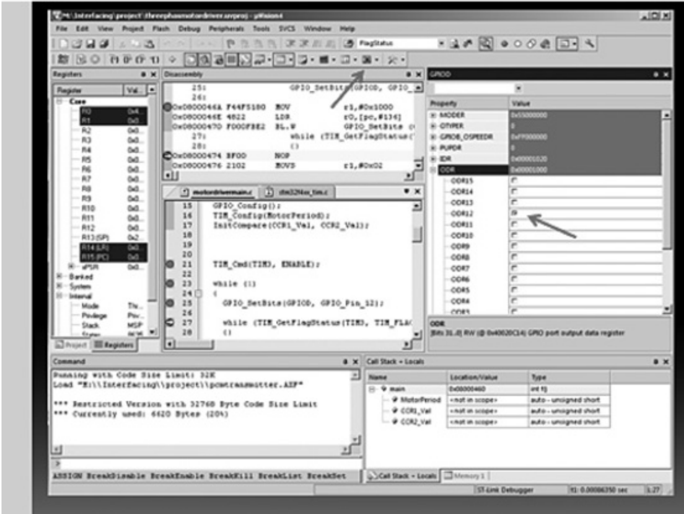


Figure A.8 A system viewer window

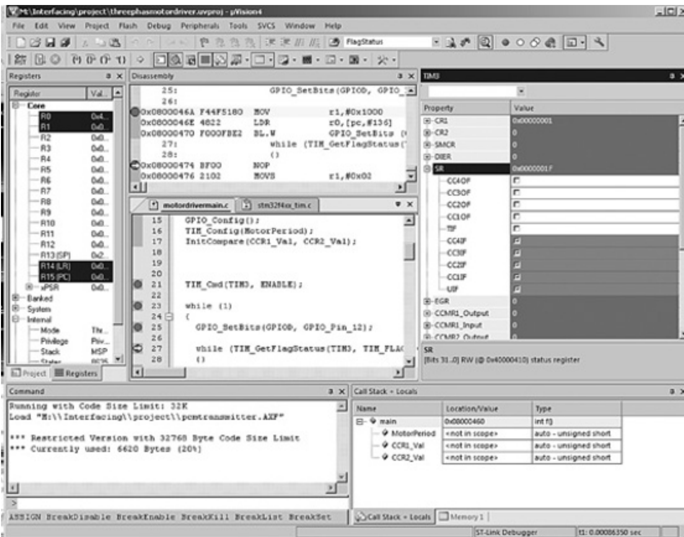


Figure A.9 Timer 3 registers

Most of these are self explanatory but a few experiments will reveal what is intended.

In particular the ‘system viewer window’ as shown if Figure A.8 enables you to examine the detailed programming of the registers in each GPIO, TIM or other peripheral.

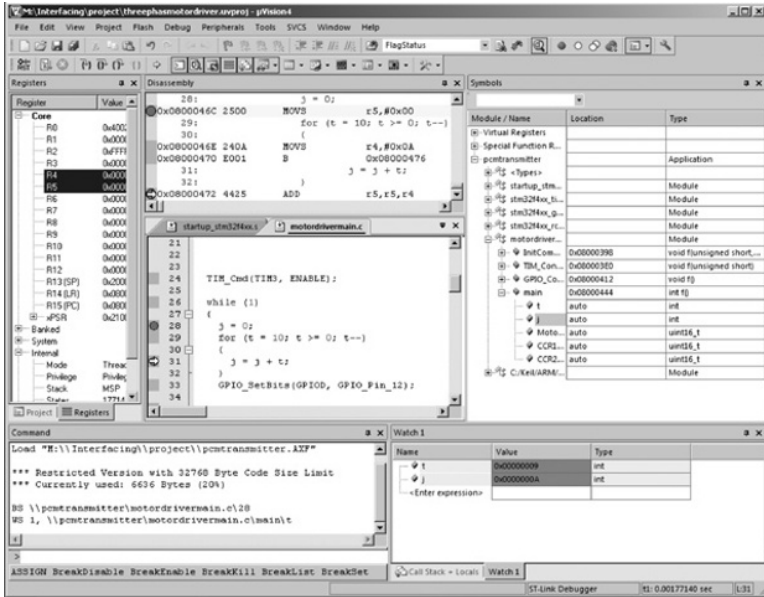


Figure A.10 Watch windows

In Figure A.9 the programming registers for TIM3 are shown and in particular the capture/compare status bits CC1IF, and so on, can be observed.

Watch windows, as shown in Figure A.10 at the lower right hand side, can be set up to monitor the dynamic variation of system variables and other aspects of the code as required.

A.5 Conclusion

In conclusion, the Keil uVision4 IDE provides very extensive tools for you to develop and debug programs for particular applications based on the standard device drivers provided for the STM32F4 device.

For a new assignment, select carefully all the device drivers that you are going to use then you will only need to focus on the C programme development itself. If you make use of an existing project essential initialization functions will be set up automatically, by including system_stm32f4.c, stm32f4_discovery.c and startup_stm32f4xx.s, so if you start a project from scratch you will need to use the same approach or use these as a basis for a developing your own implementation.

Appendix B

STM Discovery Examples Library

This appendix summarises the ST Microelectronics Keil Peripheral Example Library for STM32F4-Discovery and the advanced application examples.

B.1 Peripheral Examples

In the Peripheral Examples folder the following sub-folders will be found. A brief description of the functionality provided within each folder is given in the table.

Sub-folder name	Functionality of interface
ADC3_DMA	Describes how to use the ADC3 and DMA to transfer continuously converted data from ADC3 to memory. The ADC3 is configured to convert continuously channel7. Each time an end of conversion occurs the DMA transfers, in circular mode, the converted data from ADC3 DR register to the ADC3ConvertedValue variable

Digital Interface Design and Application, First Edition. Jonathan A. Dell.
© 2015 John Wiley & Sons, Ltd. Published 2015 by John Wiley & Sons, Ltd.

(*Continued*)

Sub-folder name	Functionality of interface
ADC_Interleaved_DMAmode2	Provides a short description of how to use the ADC peripheral to convert a regular channel in triple interleaved mode using DMA in mode 2 with 8.4 Msps
DAC_SignalsGeneration	Provides a short description of how to use the DAC peripheral to generate several signals using DMA controller. When the user presses the USER push-button, DMA transfers the two selected waveforms to the DAC
DMA_FLASH_RAM	Provides a description of how to use a DMA channel to transfer a word data buffer from FLASH memory to embedded SRAM memory
EXTI	Shows how to configure the external interrupt lines. In this example, the EXTI Line0 (connected to PA0 pin) is configured to generate an interrupt on each rising edge. In the interrupt routine a led connected to PD.12 pin on the STM32F4-Discovery board is toggled
FLASH_Program	Provides a description of how to program the STM32F4xx FLASH memory
FLASH_Write_Protection	Provides a description of how to enable and disable the write protection for the STM32F4xx FLASH memory
IO_Toggle	Describes how to toggle the GPIO pins connected on the AHB bus specifically LED4, LED3, LED5 and LED6 which are connected respectively to PD.12, PD.13, PD.14 and PD.15 on the STM32F4-Discovery board
IWDG	Shows how to update at regular period the IWDG reload counter and how to simulate a software fault generating an MCU IWDG reset on expiry of a programmed time period
MEMS	Shows how to configure the MEMS accelerometer which is part of the STM32F4-Discovery board to detect acceleration on X/Y axis and to detect the click/double click on its Z axis

(*Continued*)

(*Continued*)

Sub-folder name	Functionality of interface
PWR_CurrentConsumption	Shows how to configure the STM32F4xx system to obtain measurements of the different low power modes offering minimal current consumption
PWR_STANDBY	Shows how to switch the system to its STANDBY mode and wakeup from this situation again using various methods
PWR_STOP	Shows how to switch the system to its STOP mode and wakeup again from this mode using RTC wakeup timer event
RCC	Shows how to use, for debug purposes, the RCC_GetClocksFreq function to retrieve the current status and frequencies of different on chip clocks
SysTick	Shows how to configure the SysTick to generate a time base equal to 1 ms when the system clock is set to 168 MHz
TIM_Complementary Signals	Shows how to configure the TIM1 peripheral to generate three complementary TIM1 signals, to insert a defined dead time value, to use the break feature and to lock the desired parameters
TIM_PWM_Input	Shows how to use the TIM peripheral to measure the frequency and duty cycle of an external signal
TIM_PWM_Output	Shows how to configure the TIM peripheral in pulse width modulation (PWM) mode
TIM_TimeBase	Shows how to configure the TIM peripheral in output compare timing mode with the corresponding Interrupt requests for each channel in order to generate four different time base signals

DMA, direct memory access; ADC, analogue to digital; EXTI, external interrupt; GPIO, general purpose input and output; IWDG, Independent Watchdog; MCU, Microcontroller Unit; MEMS, Microelectromechanical System; RCC, reset and clock control and RTC Real Time Clock.

B.2 Example Application

In the folders provided in support of the Discovery Base Board (STM32F4DIS-BB) the following example applications will be found.

STM32F4xx_FPU_FFT_ Example	This example shows how the FPU unit in the Cortex-M4 is used and demonstrates the calculation of the maximum energy bin in the frequency domain of the input signal with the use of complex FFT, complex magnitude and maximum functions
STM32F4xx_Camera_ Example	This example shows how the DCMI is used to control the OV9655 Camera module (STM32F4DIS-CAM) connected via the STM32F4DIS-BB board All required image processing functions are programmable through the SCCB interface (I2C like protocol) In this example the DCMI is configured to interface with this 8 bit data camera in continuous mode. The I2C1 is used to configure the OV9655 in 8 bit RGB 5:6:5 mode. The user can select between two resolutions QQVGA(160 × 120) or QVGA(320 × 240) in order to display the captured image on the LCD(320 × 240), this selection is performed in main.h file. All camera data received by the DCMI are transferred through DMA and displayed on the LCD (connected to the FSMC in this case). As a result the CPU is free to execute other tasks
STM32F4xx_SDIO_ Example	1. This example provides an application showing how to use the SDIO firmware library and an associated driver to implement the FAT file system on the SD Card memory 2. This example provides basic suggestions of how to use the SDIO firmware library and an associated driver to perform read/write operations on the SD Card memory that could be mounted on the STM32F4DIS-BB board
STM32F4xx_USART_ Example	This example shows how to retarget the C library printf() function to the inbuilt USART. This implementation outputs the printf() message on the hyper-terminal using USART6

(*Continued*)

(*Continued*)

STM32F4xx_Ethernet_ Example (Using FreeRTOS)	**httpserver_netconn** This example implements a web server application, based on the netconn API **httpserver_socket** This example implements a web server application, based on the socket API **udptcp_echo_server_netconn** This example implements a UDP-TCP echo server demonstration Note that for http server netconn, http server socket and UDP/TCP echo server netconn demonstrations, LwIP v1.3.2 is used as the TCP/IP stack and FreeRTOS v6.1.0 is used as the Real Time Kernel
STM32F4xx_Ethernet_ Example (Stand alone)	**httpserver** This example implements a web server application **tcp_echo_server** This example implements a TCP echo server demonstration **tcp_echo_client** This example implements a TCP echo client demonstration **udp_echo_server** This example implement a UDP echo server demonstration **udp_echo_client** This example implements a UDP echo client demonstration Note that for all these STM32F4 demonstrations LwIP v1.3.2 is used as the TCP/IP stack
STM32F4xx_LCD_Example	**LCD_35T** This example implements a test of the LCD module, where a red, green and blue ribbon appear on the screen The LCD driver supports all bpp configurations but 16 bpp is used by default **LCD_Touch** This example shows how to proceed with touch screen calibration using four points at the corners

(*Continued*)

STM32F4xx_USB_Example	**USB_Device_Examples** **DFU** This example implements a device firmware upgrade (DFU) capability. It adheres to the DFU class specification, defined by the USB Implementers Forum, for reprogramming an application through the USB-FS-Device. The DFU principle is particularly well suited to applications that need to be reprogrammed in the field **MSC** The mass storage example provides a typical application using the USB OTG Device peripheral to communicate with a PC host using the bulk transfer method while a microSD card is used as the storage media. This example employs the BOT (bulk only transfer) protocol and all the required SCSI (small computer system interface) commands, and is windows compatible **VCP** This example implements a virtual COM port (VCP) capability which allows the STM32F4 device to behave as a USB-to-RS232 bridge. It uses an implementation of the CDC class following the PSTN sub-protocol **USB_Host_Examples** **HID** This example provides an implementation of a USB OTG host peripheral. When an USB device is attached to the Host port, the device is enumerated and checked whether it can support an HID device or not, if the attached device has HID capability, then the user can select a mouse or keyboard application The mouse application can highlighted a user selected screen area in green. In the keyboard application, the display shows the entered characters in green on the display screen

(*Continued*)

(*Continued*)

	MSC This example implements a USB OTG host peripheral where the STM32F4 behaves as a mass storage Host that can enumerate, show its contents and display the supported BMP images in the attached USB flash disk. The user has various options but is able to write a small file (less to 1 KB) on the disk
STM32F4xx_uCOSII_ Example	This example provides an implementation the small computer operating system uC/OS-II-V2.91 Two tasks are implemented App_TaskStar, which controls the LED blinking App_TaskKbd, which controls the LED blinking frequency through User button pressing

API, Application Programming Interface; DCIM, digital camera interface; HID, human interface device; FFT, Fast Fourier Transform; FSMC, Flexible Static Memory controller; FPU Floating Point Processor Unit; OTG USB, On-The-Go technology; SCCB, Serial Camera Control Bus; SDIO, Secure Digital memory card input/output; TCP, Transmission Control Protocol; UDP, User Datagram Protocol and USART, universal synchronous/asynchronous receiver/transmitter.

Appendix C

DAC and ADC Support Software

A summary of the software support provided for digital to analogue converter (DAC) and analogue to digital converter (ADC) subsystems in particular.

C.1 DAC Peripheral Features

DAC channels

The device integrates two 12-bit DACs that can be used independently or simultaneously (dual mode).

DAC triggers

DAC conversion can be non-triggered when the related output is available once data is written to the register.

DAC conversion can be triggered by:

1. An external event using a related input pin.
2. A timer generated event.
3. Using a software command.

DAC buffer mode feature

Each DAC channel integrates an output buffer that can be used to reduce the output impedance and to drive external loads directly without having to add

Digital Interface Design and Application, First Edition. Jonathan A. Dell.
© 2015 John Wiley & Sons, Ltd. Published 2015 by John Wiley & Sons, Ltd.

an external operational amplifier. The device data sheet gives more details about the impedance value that can be achieved.

DAC wave generation feature

Both DAC channels can be used to generate:

1. A pseudo-random Noise waveform,
2. A Triangle waveform.

DAC data format

The DAC data format can be:

1. 8-bit right alignment,
2. 12-bit left alignment,
3. 12-bit right alignment.

DAC data value to voltage correspondence

The analogue output voltage on each DAC channel pin is determined by the following equation:

$DAC_OUT = VREF + {}^*DOR / 4095$ where DOR is the DAC Data Register and VEF+ is the input reference voltage, typically 3.3 V on the Discovery board. The device data sheet explains other options.

DMA requests

When enabled a direct memory access (DMA) request can be generated when an external trigger (but not a software trigger) occurs. DMA requests are mapped to specific stream and channel configurations.

C.2 How to Use the DAC Driver

The following sequence of operations is always needed to set up the DAC.

DAC APB (advanced peripheral bus) clock must be enabled to get write access to DAC registers using the function:

RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE).

Configure the related outputs (DAC_OUT1: GPIOA_4 or DAC_OUT2: GPIOA_5) in analogue mode.

Configure the chosen DAC channel using the DAC_Init() function

Choose between possible timer sources, software or none for DAC_Trigger.

Select between none, noise or triangle by setting DAC_WaveGeneration.

Specify the LFSR number of bits for noise wave generation or the maximum triangle amplitude in DAC_LFSRUnmask_TriangleAmplitude, see stm32f4xx_dac.h for the range of values permitted.

Specify whether the output buffer is ENABLED or DISABLED in DAC_OutputBuffer.

Enable the chosen DAC channel using the DAC_Cmd(DACx, ENABLE) function.

When these steps are completed the DAC peripheral should be fully operational.

C.3 ADC Peripheral Features

Regular channels group configuration
These functions allow the user to configure the ADC and its channels group features.
Activate the continuous mode.
Configure and activate the Discontinuous mode.
Read the ADC converted values.

Multi mode ADCs Regular channels configuration
Refer to 'Regular channels group configuration' description to configure the ADC1, ADC2 and ADC3 regular channels.
Select the multi-mode ADC regular channels features (dual or triple mode)
Read the ADCs converted values as a group.

DMA for Regular channels group features configuration allows the user to:
Enable the DMA mode for a regular channel group,
Enable the generation of DMA requests continuously at the end of the last DMA transfer.

Injected channels group configuration allows the user to:
Configure the ADC Injected channels group features.
Activate the Continuous mode.
Activate the Injected Discontinuous mode.
Activate the Auto-Injected mode.
Read the ADC converted values.

C.4 How to Use the ADC driver

This sequence of operations will always be needed to setting up the ADC.

1. Enable the ADC interface clock with the function:
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC2, ENABLE)
2. ADC pins configuration
Enable the clock for the ADC's group of GPIO pins using the function: RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOx, ENABLE);
Configure these ADC pins in analogue mode

3. Configure the ADC Common features
   Select one of the 13 possible modes as ADC_Mode
   Select the ADC prescaler from one of the four (2, 4, 6 or 8) possible ratios in ADC_Prescaler
   Select the DMA transfer format from the three alternatives or disabled in ADC_DMAAccessMode
   Configure the ADC delay between two sampling phases from the range between 5 and 20 cycles ADC_TwoSamplingDelay.
4. Configure the ADC individual features
   Select the ADC bit resolution from 12, 10, 8 or 6 in ADC_Resolution
   Specify whether the conversion is to be performed in scan (multichannel) or single channel mode by setting ENABLE or DISABLE in ADC_ScanConvMode
   Choose between single (DISABLE) or continuous conversion (ENABLE) for the parameter ADC_ContinuousConvMode
   Select ADC external trigger edge for regular channel conversion from none, rising, falling or rising and falling in ADC_ExternalTrigConvEdge
   Configure the ADC external trigger sources for regular channel conversion from a selection of timers, capture/compare and external trigger sources in ADC_ExternalTrigConv
   Select from right or left alignment in ADC_DataAlign
   Select the number of conversions that will be performed in the sequencer for a regular channel group, this must be between 1 and 16, in ADC_Nbr OfConversion.
5. Activate the ADC peripheral by using ADC_Cmd(ADCx, ENABLE).

When all these steps have been taken the ADC should be ready to perform conversions when triggered.

C.5 Files for Reference

stm32f4xx_dac.h
stm32f4xx_dac.c
stm32f4xx_adc.h
stm32f4xx_adc.c

Appendix D

Example Keyboard Interface

A partial solution to the keyboard-scan design challenge is provided in the following code. This is based around a three-column by four-row keypad so will need a slight extension for the four by four keypad suggested. Functions are used to set up the required resources, inputs, outputs and timer. Other functions provide delays to control scanning and de-bounce tests. The operations to activate the scan columns and obtain row data are in further functions to simplify main() as much as possible. The translation of the row and column vales to the actual key-face signature is also provided.

```
#include "stm32f4_discovery.h"

void SetupOutPins(void);
void SetupInputPins(void);
void SetupTimerDelay(uint16_t time);
void Set_Column(int i);
void Timed_Wait(uint16_t time);
uint8_t get_row(void);
char get_key(uint8_t col, uint8_t row);
```

Digital Interface Design and Application, First Edition. Jonathan A. Dell.
© 2015 John Wiley & Sons, Ltd. Published 2015 by John Wiley & Sons, Ltd.

```
int main(void)
{
    uint8_t i, row;
    char key;

    SetupOutPins();
    SetupInputPins();
    SetupTimerDelay(1000); /* 1000us de-bounce */

    while(1)
    {
        for (i = 1; i <4; i++) /* scan columns */
        {
            Set_Column(i);
            row =get_row();
            if (row != 0)
            {
                key = get_key(i, row);
            }
            Timed_Wait(100); /* 100ms column scan */
        }
    }
}

uint8_t get_row()
{
    uint16_t row, rowx;

    row = (GPIO_Read_InputData(GPIOB) & 0xf000);
    if (row != 0)
    {
        Timed_Wait(10); /* try again in 1ms */
        rowx = (GPIO_Read_InputData(GPIOB) & 0xf000);
        if (rowx != row)
        {
            row = 0;
        }
        else                  /*wait till key lifted*/
        {
            while((GPIO_ReadInputData(GPIOD)& 0xf000)
            == row)
            {
```

```
            Timed_Wait(10);
            }
        }
    }
    return(row >> 12);
}

/* translate row (1, 2, 4,8) and column (1,2,3,4)
to key code */
char get_key(uint8_t col, uint8_t row)
{
    char keys[] = {'1', '2', '3', '4', '5', '6',
'7', '8', '9', '*', '0', '#'};
    char key;
    switch (row)
    {
        case 1
            key = keys[(col - 1)];
            break;
        case 2:
            key = keys[(col - 1) + 3];
            break;
        case 4:
            key = keys[(col - 1) + 6];
            break;
        case 8:
            key = keys[(col - 1) + 9];
            break;
        default:
            break;
    }
    return (key);
}

/* Use GPIOB pins 5, 6, 7, 8 for column outputs */
void SetupOutPins(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
/* GPIOD Peripheral clock enable */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD,
ENABLE);
```

```
/* Configure PB5, PB6, PB7 and PB8 in output
pushpull mode */
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_
Pin_6 | GPIO_Pin_7 | GPIO_Pin_8;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
   GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
   GPIO_Init(GPIOD, &GPIO_InitStructure);
}

/* Use GPIOD pins 12, 13, 14, 15 for row inputs*/
void SetupInputPins(void)
{
   GPIO_InitTypeDef GPIO_InitStructure;
/* GPIOD Peripheral clock enable */
   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB,
ENABLE);
/* Configure PD12, PD13, PD14 and PD15 in input mode */
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12 |
GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN;
   GPIO_Init(GPIOB, &GPIO_InitStructure);
}

/* Use TIM6 to provide one millisecond for de-bounce */
void SetupTimerDelay(uint16_t time)
{
   TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;

   /*TIM6 Clock Enable */
   RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM6,
ENABLE);

   /* Time Base Configuration */
   TIM_TimeBaseStructure.TIM_Period = time;
   TIM_TimeBaseStructure.TIM_Prescaler = 8400;
/* 1/10 ms */
   TIM_TimeBaseStructure.TIM_ClockDivision = 0;
   TIM_TimeBaseStructure.TIM_CounterMode = TIM_
CounterMode_Up;
```

```
    TIM_TimeBaseInit(TIM6, &TIM_TimeBaseStructure);

    /* Prescaler Configuration */
    TIM_PrescalerConfig(TIM6, 8400, TIM_
PSCReloadMode_Immediate);

    TIM_Cmd(TIM6, ENABLE);
}

void Set_Column(int i)
{
    switch(i)
    {
        case 1:
            GPIO_ResetBits(GPIOB, GPIO_Pin_7);
            GPIO_SetBits(GPIOB, GPIO_Pin_4);
            break;
        case 2:
            GPIO_ResetBits(GPIOD, GPIO_Pin_4);
            GPIO_SetBits(GPIOD, GPIO_Pin_5);
            break;
        case 3:
            GPIO_ResetBits(GPIOD, GPIO_Pin_5);
            GPIO_SetBits(GPIOD, GPIO_Pin_6);
            break;
        case 4:
            GPIO_ResetBits(GPIOD, GPIO_Pin_6);
            GPIO_SetBits(GPIOD, GPIO_Pin_7);
            break;
        default:
            break;
    }
}

void Timed_Wait(uint16_t time)
{
    /* set period in ARR */
    TIM_SetAutoreload(TIM6, time);
    TIM_SetCounter(TIM6, 0);
    while (TIM_GetFlagStatus(TIM6, TIM_FLAG_Update) != 1)
    {}
    TIM_ClearFlag(TIM6, TIM_FLAG_Update);
)
```

Note that it would be quite straightforward to establish an interrupt driven interface that could be triggered by user activation of the keypad. This would help to make the interface more efficient.

It will be found that some keypad assemblies contain a built-in encoder, which delivers both a character code and an interrupt signal, so the interface design is considerably simplified.

Index

address-bus, 2
address decode, 11
analogue to digital converter (ADC), 69
  binary search ADC, 70
  STM32F4 ADCs, 74
  support functions, 168
  switched capacitor ADC, 72
  timing, 73
ARM architecture, 14
arrays, 19
assembler language, 16
asynchronous, 91

baud rate, 92
binary search, 70
break-point, 164
bus matrix, 119
bus timing, 28

C language revision, 19
  arrays, 19
  functions, 19
capture/compare, 48
case studies, 135
character generator, 125
checksum, 154
clock, 50
compass module, 135

D-type, 7
data structures in C, 21
  arrays, 19
  structures, 21
debounce, 42
debug tools, 164
delay function, 39
  software, 40
  timer, 60
digital camera interface (DCMI), 133
digital compass, 135
digital to analogue converter (DAC), 67
  binary weighted network, 67
  programming, 75
  transfer characteristic, 69
  triggering, 77
  waveform generation, 76

Digital Interface Design and Application, First Edition. Jonathan A. Dell.
© 2015 John Wiley & Sons, Ltd. Published 2015 by John Wiley & Sons, Ltd.

direct memory access (DMA), 118
  ADC using DMA, 85
  DAC using DMA, 79
discovery support library, 168
display (LCD), 121
driver, 10

edge-triggered, 7
embedded system, 1
event
  timer, 55
  interrupt, 115
exceptions, 117

flags (status)
  ADC, 83
  USART, 94
  timer, 59
flip-flop, 5
frequency (clock), 48
function examples, 18
  bcd_add(), 152
  BCD_PRINT, 151
  char_print(), 152
  day_name(), 151
  decode_MSFdata(), 147
  EXTI4_IRQHandler(), 144
  get_NMEA_char(), 155
  MAG_read_sixbyte(), 138
  month_name(), 151
  printMSF_ydt(), 150
  setup_Compare(), 146
  setup_EXTI(), 142
  setup_TIM3(), 146
  STRING_PRINT(), 152

general purpose input and output (GPIO)
  pin circuit, 33
  pin programming, 34
global variables, 143
GPS navigation, 153

HDLC, 105
  control byte, 106
  frame structure, 106
header files, 21
high level C language, 18
hyper terminal, 95

I2C, 95
  read and write modes, 96
  temperature gauge, 98
  touch screen interface, 96
I/O pins, 31
inputs, 29
integrated development environment (IDE), 18
interface software, 14
interrupt mask register, 143
interrupt pending register, 143
interrupt service routine (ISR), 113
interrupts, 112

keyboard, 41

latency, 113
libraries, 18
light emitting diode (LED), 27
linking, 164
liquid crystal display (LCD), 121
logic elements, 5
loops (poling), 113

machine instructions, 2
magnetometer, 135
masking (interrupt), 143
master-in-slave-out (MISO), 101
master-out-slave-in (MOSI), 101
memory map, 3
motor, 61
MSF time signal, 140

naming (ISR), 117
Nested Vectored Interrupt Controller (NVIC), 115
networking protocols (HDLC), 105
Nyquist sampling, 72

open drain, 10
open loop, 135
outputs, 26

peripheral driver, 16
phase-locked loop, 47
pointers, 30
polling, 113
prescaler, 85

priority (interrupt), 114
processor architecture, 2
  ARM, 14
project build report, 164
project file structure, 164
pull-down, 10
pull-up, 10
pulse-width-modulation (PWM), 61
push-pull, 10

quantization, 68

receiver (MSF), 141
receiver (RS232), 91
register, 9
resolution, 69
RS232, 91
  data format, 92

sample and hold, 71
sampling, 70
sensor (temperature), 88
serial communication, 90
  Manchester coding, 91
  RS232, 91
serial peripheral interface (SPI), 101
  ADC interface, 103
servo, 63
  compass servo, 140
software architecture, 4
software program design, 18
start and stop bits (RS232), 91
status registers
  ADC, 83
  USART, 94
  timer, 59
structures, 21
successive approximation, 70
system clock, 48
system viewer window, 166

target system, 2
temperature sensor, 88
template (ISR), 117
timeout, 101
timer, 45
  count modes, 48
  STM32F4 timers, 47
  programming, 51
  time-base, 55
  time measurement, 56
  triggering, 55
transmitter (RS232), 91
typedef, 21

universal asynchronous receiver/transmitter (UART), 91
universal I/O, 31
universal serial bus (USB), 107
  connections, 107
  protocol, 110
universal synchronous/ asynchronous receiver/transmitter (USART), 92
uVision4, 14
  getting started, 161

vectors, 113
voltage reference , 67

watch window, 167

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.