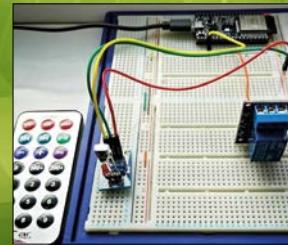
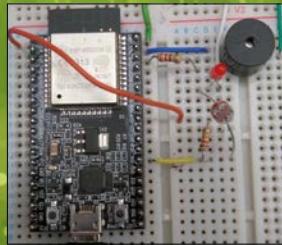
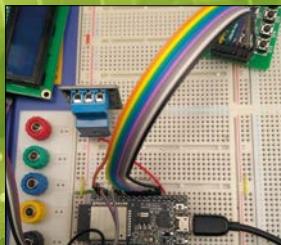
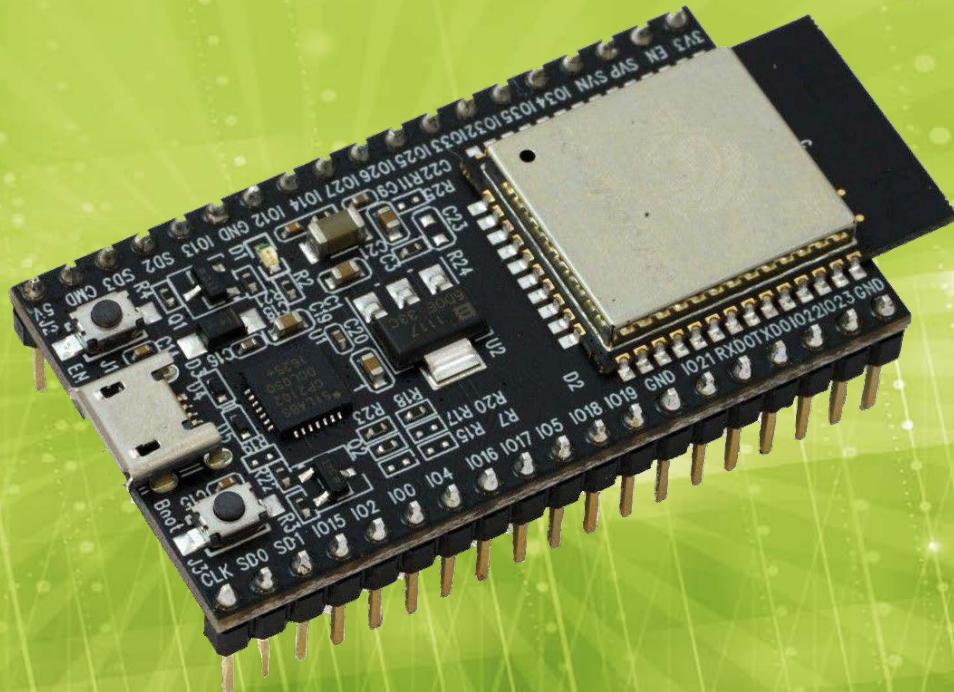


59 Experiments with Arduino IDE and Python

The Complete ESP32 Projects Guide



Dogan Ibrahim

The logo for elektor magazine, featuring a stylized lowercase 'e' inside a red circle followed by the word 'elektor' in a bold, sans-serif font.

LEARN ➤ DESIGN ➤ SHARE



The Complete ESP32 Projects Guide

59 Experiments with Arduino IDE and Python



Dogan Ibrahim

- This is an Elektor Publication. Elektor is the media brand of Elektor International Media B.V.
78 York Street, London W1H 1DP, UK
Phone: (+44) (0)20 7692 8344
- All rights reserved. No part of this book may be reproduced in any material form, including photocopying, or storing in any medium by electronic means and whether or not transiently or incidentally to some other use of this publication, without the written permission of the copyright holder except in accordance with the provisions of the Copyright Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd., 90 Tottenham Court Road, London, England W1P 9HE. Applications for the copyright holder's permission to reproduce any part of the publication should be addressed to the publishers.

- Declaration

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

- Acknowledgement

The authors would like to express their thanks to Espressif Systems for giving permission for the ESP32 related figures and tables to be included in this book.

- British Library Cataloguing in Publication Data

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

- Disclaimer

The material in this publication is of the nature of general comment only, and does not represent professional advice. It is not intended to provide specific guidance for particular circumstances and it should not be relied on as the basis for any decision to take action or not take action on any matter which it covers. Readers should obtain professional advice where appropriate, before making any such decision. To the maximum extent permitted by law, the author and publisher disclaim all responsibility and liability to any person, arising directly or indirectly from any person taking or not taking action based on the information in this publication.

- **ISBN 978-1-907920-75-2**

© Copyright 2019: Elektor International Media b.v.

Prepress Production: D-Vision, Julian van den Berg

First published in the United Kingdom 2019

Printed in the Netherlands by Wilco



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

LEARN ➤ DESIGN ➤ SHARE

Preface	25
Chapter 1 • The ESP32 Processor	28
1.1 Overview	28
1.2 The Architecture of ESP32	29
1.2.1 The CPU	30
1.2.2 Internal Memory	31
1.2.3 External Memory	31
1.2.4 General Purpose Timers	31
1.2.5 Watchdog Timers	31
1.2.6 The System Clock	31
1.2.7 Real Time Clock (RTC)	31
1.2.8 General Purpose Input-Outputs (GPIOs)	31
1.2.9 Analog To Digital Converter (ADC)	31
1.2.10 Digital To Analog Converter (DAC)	32
1.2.11 Hall Sensor	32
1.2.12 Temperature Sensor	32
1.2.13 Touch Sensor	32
1.2.14 UART	32
1.2.15 I2C Interface	32
1.2.16 I2S Interface	32
1.2.17 Infrared Controller	32
1.2.18 Pulse Width Modulation	32
1.2.19 LED PWM	32
1.2.20 Pulse Counter	33
1.2.21 SPI Interface	33
1.2.22 Hardware Accelerators	33
1.2.23 Wi-Fi	33
1.2.24 Bluetooth	33
1.2.25 Controller Area Network (CAN)	33
1.2.26 SD Card Support	33
1.3 ESP32 Development Boards	33
1.4 Summary	34

Chapter 2 • ESP32 Devkitc Development Board.....	35
2.1 Overview	35
2.2 ESP32 DevKitC Hardware	35
2.3 Powering Up the ESP32 DevKitC	37
2.4 Summary	40
Chapter 3 • Arduino IDE for the ESP32 DevKitC	41
3.1 Overview	41
3.2 Installing the Arduino IDE for the ESP32 DevKitC	41
3.3 Summary	46
Chapter 4 • Basic projects using the Arduino IDE and the ESP32 DevKitC	47
4.1 Overview	47
4.2 PROJECT 1 – Flashing LED	47
4.2.1 Description	47
4.2.2 The Aim.	47
4.2.3 Block diagram	48
4.2.4 Circuit Diagram	48
4.2.5 Construction.	49
4.2.6 PDL of the Project.	49
4.2.7 Program Listing	50
4.2.8 Program Description	51
4.2.9 Suggestions	51
4.3 PROJECT 2 – Lighthouse Flashing LED	51
4.3.1 Description	51
4.3.2 The Aim.	51
4.3.3 Block diagram:.....	51
4.3.4 Circuit Diagram	51
4.3.5 Construction.	51
4.3.6 PDL of the Project.	52
4.3.7 Program Listing	52
4.3.8 Program Description	53
4.4 PROJECT 3 – Alternately Flashing LEDs	53
4.4.1 Description	53
4.4.2 The Aim.	53

4.4.3 Block diagram:	53
4.4.4 Circuit Diagram	53
4.4.5 Construction.	54
4.4.6 PDL of the Project	55
4.4.7 Program Listing	55
4.4.8 Program Description	56
4.5 PROJECT 4 – Rotating LEDs	56
4.5.1 Description	56
4.5.2 The Aim.	56
4.5.3 Block diagram:	57
4.5.4 Circuit Diagram	57
4.5.5 Construction.	57
4.5.6 PDL of the Project	58
4.5.7 Program Listing	59
4.5.8 Program Description	60
4.5.9 Suggestions	60
4.6 PROJECT 5 – Christmas Lights.	60
4.6.1 Description	60
4.6.2 The Aim.	60
4.6.3 Block diagram:	60
4.6.4 Circuit Diagram	60
4.6.5 Construction.	60
4.6.6 PDL of the Project	60
4.6.7 Program Listing	61
4.6.8 Program Description	63
4.6.9 Modified Program	63
4.6.10 Suggestions	64
4.7 PROJECT 6 – Binary Up Counter with LEDs	64
4.7.1 Description	64
4.7.2 The Aim.	65
4.7.3 Block diagram:	65
4.7.4 Circuit Diagram	65
4.7.5 Construction.	66

4.7.6 PDL of the Project	66
4.7.7 Program Listing	67
4.7.8 Program Description	68
4.8 PROJECT 7 – Binary Up/Down Counter with LEDs	69
4.8.1 Description	69
4.8.2 The Aim.	69
4.8.3 Block diagram:	69
4.8.4 Circuit Diagram	69
4.8.5 Construction.	70
4.8.6 PDL of the Project.	71
4.8.7 Program Listing	72
4.8.8 Program Description	74
4.9 PROJECT 8 – Knight Rider Car LEDs.	74
4.9.1 Description	74
4.9.2 The Aim.	74
4.9.3 Block diagram:	74
4.9.4 Circuit Diagram	74
4.9.5 Construction.	74
4.9.6 PDL of the Project.	74
4.9.7 Program Listing	75
4.9.8 Program Description	76
4.9.10 Suggestions	76
4.10 PROJECT 9 – Changing the Brightness of an LED	76
4.10.1 Description.	76
4.10.2 The Aim.	76
4.10.3 Block diagram:	76
4.10.4 Circuit Diagram.	77
4.10.5 Construction.	77
4.10.6 PDL of the Project.	77
4.10.7 Program Listing	77
4.10.8 Program Description	78
4.10.9 Suggestions	80
4.11 PROJECT 10 – Generating Random Sounds Using a Buzzer	80

4.11.1 Description	80
4.11.2 The Aim	80
4.11.3 Block diagram:	80
4.11.4 Circuit Diagram	80
4.11.5 Construction	81
4.11.6 PDL of the Project	81
4.11.7 Program Listing	82
4.11.8 Program Description	83
4.11.9 Suggestion	83
4.12 PROJECT 11 – LED Colour Wand	83
4.12.1 Description	83
4.12.2 The Aim	83
4.12.3 Block diagram:	84
4.12.4 Circuit Diagram	84
4.12.5 Construction	84
4.12.6 PDL of the Project	85
4.12.7 Program Listing	86
4.12.8 Program Description	87
4.12.9 Suggestions	87
4.13 PROJECT 12 – Using the Built-in Hall Sensor – Door Alarm	87
4.13.1 Description	87
4.13.2 The Aim	87
4.13.3 Block diagram:	87
4.13.4 Circuit Diagram	88
4.13.5 PDL of the Project	88
4.13.6 Program Listing	89
4.14 PROJECT 13 – Using the Built-in Temperature Sensor	91
4.14.1 Description	91
4.14.2 The Aim	91
4.14.3 Program Listing	91
4.14.4 Program Description	92
4.15 PROJECT 14 – Chip Identity	92
4.15.1 Description	92

4.15.2 The Aim	92
4.15.3 Program Listing	93
4.16 Summary	93
Chapter 5 • Simple projects using the Arduino IDE and the ESP32 DevKitC	94
5.1 Overview	94
5.2 PROJECT 1 – Thermometer with Serial Monitor	94
5.2.1 Description	94
5.2.2 The Aim.	94
5.2.3 Block diagram:	94
5.2.4 Circuit Diagram	94
5.2.5 Construction.	95
5.2.6 PDL of the Project.	96
5.2.7 Program Listing	97
5.2.8 Program Description	97
5.3 PROJECT 2 – Temperature and Relative Humidity with Serial Monitor.	99
5.3.1 Description	99
5.3.2 The Aim.	99
5.3.3 Block diagram:	99
5.3.4 Circuit Diagram	99
5.3.5 Construction.	101
5.3.6 PDL of the Project.	101
5.3.7 Program Listing	102
5.3.8 Program Description	104
5.4 PROJECT 3 – Measuring the Light Level	104
5.4.1 Description	104
5.4.2 The Aim.	104
5.4.3 Block diagram:	104
5.4.4 Circuit Diagram	105
5.4.5 Construction.	106
5.4.6 PDL of the Project.	106
5.4.7 Program Listing	106
5.4.8 Program Description	107
5.4.9 Suggestions	108

5.5 PROJECT 4 – Darkness Reminder.....	108
5.5.1 Description	108
5.5.2 The Aim.....	108
5.5.3 Block diagram:.....	108
5.5.4 Circuit Diagram	109
5.5.5 Construction.....	109
5.5.6 PDL of the Project.....	110
5.5.7 Program Listing	110
5.5.8 Program Description	111
5.5.9 Suggestions	111
5.6 PROJECT 5 – LED Dice	112
5.6.1 Description	112
5.6.2 The Aim.....	112
5.6.3 Block diagram:.....	112
5.6.4 Circuit Diagram	112
5.6.5 Construction.....	113
5.6.6 PDL of the Project.....	113
5.6.7 Program Listing	114
5.6.8 Program Description	117
5.7 PROJECT 6 – Logic Probe	117
5.7.1 Description	117
5.7.2 The Aim.....	117
5.7.3 Block diagram:.....	118
5.7.4 Circuit Diagram	118
5.7.5 Construction.....	118
5.7.6 PDL of the Project.....	119
5.7.7 Program Listing	120
5.7.8 Program Description	121
5.7.9 Modified Program	121
5.8 PROJECT 7 – 7 - Segment LED Display Counter.....	124
5.8.1 Description	124
5.8.2 The Aim.....	127
5.8.3 Block diagram	127

5.8.4 Circuit Diagram	127
5.8.5 Construction.	128
5.8.6 PDL of the Project.	128
5.8.7 Program Listing	129
5.8.8 Program Description	131
5.8.9 Modified Program	131
5.9 PROJECT 8 – Clap ON – Clap OFF	134
5.9.1 Description	134
5.9.2 The Aim.	134
5.9.3 Block diagram	134
5.9.4 Circuit Diagram	135
5.9.5 Construction.	135
5.9.6 PDL of the Project.	136
5.9.7 Program Listing	136
5.9.8 Program Description	137
5.10 PROJECT 9 – LCD "Hello from ESP32".	137
5.10.1 Description.	137
5.10.2 The Aim.	137
5.10.3 Block diagram	137
5.10.4 Circuit Diagram.	138
5.10.5 Construction.	139
5.10.6 PDL of the Project.	139
5.10.7 Program Listing	140
5.10.8 Program Description	141
5.11 PROJECT 10 – LCD Event Counter	141
5.11.1 Description.	141
5.11.2 The Aim.	142
5.11.3 Block diagram	142
5.11.4 Diagram.	142
5.11.5 Construction.	142
5.11.6 PDL of the Project.	143
5.11.8 Program Description	145
5.12 PROJECT 11 – LCD Command	145

5.12.1 Description	145
5.12.2 The Aim	145
5.12.3 Block diagram	145
5.12.4 Circuit Diagram	145
5.12.5 Construction	145
5.12.6 LCD Commands	145
5.12.7 Program Listing	146
5.12.8 Program Description	148
5.13 PROJECT 12 – External Interrupts	149
5.13.1 Description	149
5.13.2 The Aim	149
5.13.3 Block diagram	149
5.13.4 Circuit Diagram	149
5.13.5 Construction	150
5.13.6 PDL of the Project	150
5.13.7 Program Listing	151
5.13.8 Program Description	152
5.14 PROJECT 13 – External Interrupts	153
5.14.1 Description	153
5.14.2 The Aim	153
5.14.3 Block diagram	153
5.14.4 Circuit Diagram	153
5.14.5 Construction	153
5.14.6 PDL of the Project	153
5.14.7 Program Listing	154
5.14.8 Program Description	155
5.15 PROJECT 14 – Using the Touch Sensitive Inputs – Touch Based LED Control	156
5.15.1 Description	156
5.15.2 The Aim	156
5.15.3 Block diagram	156
5.15.4 Circuit Diagram	156
5.15.5 Construction	157
5.15.6 PDL of the Project	158

5.15.7 Program Description	158
5.15.8 Using Touch Interrupts	160
5.16 PROJECT 15 – Using the Touch Sensitive Inputs – Changing the LED Brightness	161
5.16.1 Description	161
5.16.2 The Aim	161
5.16.3 Block diagram	161
5.16.4 Circuit Diagram	161
5.16.5 PDL of the Project	161
5.16.6 Program Description	161
5.17 PROJECT 16 – Using Multiple Touch Sensitive Inputs – Electronic Organ	163
5.17.1 Description	163
5.17.2 The Aim	163
5.17.3 Block diagram	163
5.17.4 Circuit Diagram	163
5.17.5 PDL of the Project	164
5.17.6 Program Description	165
5.17.7 Suggestions For Additional Work	167
5.18 PROJECT 17 – Using the SPI Bus – Digital to Analog Converter (DAC)	167
5.18.1 Description	167
5.18.2 The Aim	169
5.18.3 Block diagram	169
5.18.4 Circuit Diagram	169
5.18.5 The Construction	171
5.18.6 The PDL	171
5.18.7 Program Listing	172
5.18.8 Program Description	173
5.19 PROJECT 18 – Using the UARTs	175
5.19.1 Description	175
5.19.2 The Aim	175
5.19.3 Block diagram	175
5.19.4 Circuit Diagram	175
5.19.5 The PDL	176
5.19.6 Program Listing	177

5.19.7 Program Description	178
5.20 PROJECT 19 – Writing Data to the Flash Memory	179
5.20.1 Description	179
5.20.2 The Aim	179
5.20.3 Block diagram	179
5.20.4 Circuit Diagram	179
5.20.5 The PDL	179
5.20.6 Program Listing	180
5.20.7 Program Description	181
5.21 PROJECT 20 – Reading Data from the Flash Memory	181
5.21.1 Description	181
5.21.2 The Aim	181
5.21.3 Block diagram	182
5.21.4 Circuit Diagram	182
5.21.5 The PDL	182
5.21.6 Program Listing	182
5.21.7 Program Description	183
5.22 PROJECT 21 – Writing Floating Point Data to the Flash Memory	183
5.22.1 Description	183
5.22.2 The Aim	183
5.22.3 Block diagram	183
5.22.4 Circuit Diagram	183
5.22.5 Program Listing	184
5.22.6 Program Description	185
5.23 PROJECT 22 – Reading Floating Point Data from the Flash Memory	185
5.23.1 Description	185
5.23.2 The Aim	185
5.23.3 Block diagram	185
5.23.4 Circuit Diagram	185
5.23.5 Program Listing	185
5.23.6 Program Description	186
5.24 Summary	186

Chapter 6 • Intermediate projects using the Arduino IDE and the ESP32 DevKitC . 187	
6.1 Overview	187
6.2 PROJECT 1 – ON-OFF Temperature Control	187
6.2.1 Description	187
6.2.2 The Aim.	187
6.2.3 Block diagram:	187
6.2.4 Circuit Diagram	188
6.2.5 Construction.	189
6.2.6 PDL of the Project.	189
6.2.7 Program Listing	190
6.2.8 Program Description	193
6.3 PROJECT 2 – Generating Waveforms – Sawtooth Waveform	194
6.3.1 Description	194
6.3.2 The Aim.	195
6.3.3 Block diagram:	195
6.3.4 The DAC	195
6.3.5 Circuit Diagram	196
6.3.6 Construction.	196
6.3.7 PDL of the Project.	197
6.3.8 Program Listing	197
6.3.9 Program Description	198
6.4 PROJECT 3 – Generating Waveforms – Triangle Waveform	198
6.4.1 Description	198
6.4.2 The Aim.	198
6.4.3 Block diagram:	198
6.4.4 Circuit Diagram	199
6.4.5 PDL of the Project.	199
6.4.6 Program Listing	199
6.4.7 Program Description	200
6.5 PROJECT 4 – Port Expander	201
6.5.1 Description	201
6.5.2 The Aim.	201
6.5.3 Block diagram:	201

6.5.4 Circuit Diagram	201
6.5.5 The MCP23017	202
6.5.6 Construction.	204
6.5.7 PDL of the Project	204
6.5.8 Program Listing	205
6.5.9 Program Description	206
6.6 PROJECT 5 – Mini Electronic Organ	207
6.6.1 Description	207
6.6.2 The Aim.	207
6.6.3 Block diagram:	207
6.6.4 Circuit Diagram	207
6.6.5 Construction.	209
6.6.6 PDL of the Project	210
6.6.7 Program Listing	211
6.6.8 Program Description	214
6.7 PROJECT 6 – Calculator with Keypad and LCD	215
6.7.1 Description	215
6.7.2 The Aim.	215
6.7.3 Block diagram:	215
6.7.4 Circuit Diagram	216
6.7.5 Construction.	217
6.7.6 PDL of the Project	217
6.7.7 Program Listing	218
6.7.8 Program Description	223
6.8 PROJECT 7 – High-Low Game	224
6.8.1 Description	224
6.8.2 The Aim.	224
6.8.3 Block diagram:	225
6.8.4 Circuit Diagram	225
6.8.5 Construction.	225
6.8.6 PDL of the Project	225
6.8.7 Program Listing	226
6.8.8 Program Description	229

6.9 PROJECT 8 – Learning the Times Table	230
6.9.1 Description	230
6.9.2 The Aim	230
6.9.3 Block diagram:	230
6.9.4 Circuit Diagram	230
6.9.5 Construction	230
6.9.6 PDL of the Project	231
6.9.7 Program Listing	231
6.9.8 Program Description	234
6.10 PROJECT 9 – Learning Basic Mathematics	235
6.10.1 Description	235
6.10.2 The Aim	235
6.10.3 Block diagram:	235
6.10.4 Circuit Diagram	235
6.10.5 Construction	235
6.10.6 PDL of the Project	235
6.10.7 Program Listing	236
6.10.8 Program Description	237
6.10.9 Suggestions	237
6.11 PROJECT 10 - Keypad Door Lock	238
6.11.1 Description	238
6.11.2 The Aim	238
6.11.3 Block diagram:	238
6.11.4 Circuit Diagram	238
6.11.5 Construction	239
6.11.6 PDL of the Project	240
6.11.7 Program Listing	240
6.11.8 Program Description	244
6.11.9 Suggestions	244
6.12 PROJECT 11 – Using SD Cards – Writing to a Card	244
6.12.1 Description	244
6.12.2 The Aim	244
6.12.3 Block diagram:	244

6.12.4 Circuit Diagram	245
6.12.5 Construction	247
6.12.6 PDL of the Project	247
6.12.7 Program Listing	248
6.12.8 Program Description	249
6.13 PROJECT 12 – Using SD Cards – Reading from a Card	251
6.13.1 Description	251
6.13.2 The Aim	251
6.13.3 Block diagram:	251
6.13.4 Circuit Diagram	251
6.13.6 PDL of the Project	251
6.13.7 Program Listing	252
6.13.8 Program Description	253
6.14 PROJECT 13 – Infrared Receiver (IR)	253
6.14.1 Description	253
6.14.2 The Aim	254
6.14.3 Block diagram:	254
6.14.4 Circuit Diagram	254
6.14.5 PDL of the Project	255
6.14.6 Program Listing	256
6.14.7 Modifications	259
6.15 Low Power Operation	260
6.15.1 Deep Sleep Mode	262
6.16 PROJECT 14 – Using Timer to Wake Up.	262
6.16.1 Description	262
6.16.2 The Aim	262
6.16.3 Block diagram:	262
6.16.4 Circuit Diagram	262
6.16.5 PDL of the Project	262
6.16.6 Program Listing	263
6.16.7 Program Description	264
6.16.8 Wake Up Reason	265
6.17 PROJECT 15 – Using Touchpad to Wake Up	267

6.17.1 Description	267
6.17.2 The Aim	267
6.17.3 Block diagram:	267
6.17.4 Circuit Diagram	267
6.17.5 PDL of the Project	267
6.17.6 Program Listing	268
6.17.7 Program Description	269
6.18 PROJECT 16 – Using External Trigger to Wake Up	270
6.18.1 Description	270
6.18.2 The Aim	270
6.18.3 Block diagram:	270
6.18.4 Circuit Diagram	270
6.18.5 PDL of the Project	271
6.18.6 Program Listing	271
6.18.7 Program Description	272
Chapter 7 • ESP32 DevKitC network programming using the Arduino IDE	276
7.1 Overview	276
7.2 Scanning the Surrounding Wi-Fi Networks	276
7.3 Connecting to a Wi-Fi Network	278
7.4 HTTP GET Requests	281
7.5 Using the Socket Library	283
7.5.1 UDP Programs	284
7.5.2 TCP/IP Programs	288
7.6 Getting the Time from NTP Client	290
7.7 Summary	292
Chapter 8 • Project – The temperature and humidity on the cloud	293
8.1 Overview	293
8.2 The Block Diagram	293
8.3 The Cloud	293
8.4 Program Listing	295
8.5 Summary	298
Chapter 9 • Remote Web Based Control – Web Server	300
9.1 Overview	300

9.2 The Block Diagram	300
9.3 HTTP Web Server/Client	301
9.4 ESP32 DevKitC Program Listing	302
9.5 Accessing Web Server From Anywhere	306
9.6 Summary	308
Chapter 10 • Remote control using mobile phone	309
10.1 Overview	309
10.2 The Block Diagram	309
10.3 Mobile Phone Application	310
10.4 ESP32 DevKitC Program Listing	310
10.5 Summary	314
Chapter 11 • Send temperature and humidity to a mobile phone	315
11.1 Overview	315
11.2 The Block Diagram	315
11.3 Mobile Phone Application	315
11.4 ESP32 DevKitC Program Listing	316
11.5 Summary	320
Chapter 12 • Web server with relay	321
12.1 Overview	321
12.2 The Aim	321
12.3 The Block Diagram	321
12.4 Circuit Diagram	321
12.5 The Construction	323
12.6 Program Listing	324
12.7 Program Description	326
12.8 Summary	328
Chapter 13 • ESP32 DevKitC Bluetooth programming	329
13.1 Overview	329
13.2 Bluetooth BLE	330
13.2.1 Bluetooth BLE Software Model	331
13.3 PROJECT 1 – Sending Data to Mobile Phone Using Bluetooth BLE	332
13.3.1 Description	332
13.3.2 Program Listing	332

13.3.3 Program PDL	334
13.3.4 Program Description	335
13.3.5 Testing the Program	335
13.4 PROJECT 2 – 4-Channel Relay Control Using Bluetooth BLE.	337
13.4.1 Description.	337
13.4.2 The Aim.	337
13.4.3 The Block Diagram	337
13.4.4 Circuit Diagram.	337
13.4.5 Program PDL	338
13.4.6 Program Listing	338
13.4.7 Program Description	341
13.4.8 Testing the Program	341
13.5 PROJECT 3 – Serial Communication Over Bluetooth Classic.	343
13.5.1 Description.	343
13.5.2 The Aim.	343
13.5.3 Program Listing	343
13.5.4 Program Description	344
13.5.5 Testing the Program on the PC	344
13.5.6 Testing the Program on Mobile Phone	345
13.6 PROJECT 4 – 4-Channel Relay Control Using Bluetooth Classic	346
13.6.1 Description.	346
13.6.2 The Aim.	347
13.6.3 The Block Diagram	347
13.6.4 Circuit Diagram.	347
13.6.5 Program Listing	347
13.6.6 Program Description	349
13.6.7 Testing the Program on Mobile Phone	349
13.7 PROJECT 5 – Sending Temperature and Humidity to Mobile Phone Using Bluetooth Classic	351
13.7.1 Description.	351
13.7.2 The Aim.	351
13.7.3 The Block Diagram	351
13.7.4 Circuit Diagram.	351

13.7.5 Program Listing	351
13.7.6 Program Description	352
13.7.7 Testing the Program on a Mobile Phone	352
13.8 Summary	353
Chapter 14 • Using Micropython with the ESP32 DevKitC	354
14.1 Overview	354
14.2 Installing MicroPython on ESP32 DevKitC	354
14.2.1 Testing the MicroPython Installation	360
14.3 Flashing LED	360
14.4 LED With Push-Button Switch	361
14.5 Temperature and Humidity	361
14.6 Connecting to a Wi-Fi	362
14.7 MicroPython UDP Programs	363
14.8 Storing Temperature and Humidity on the Cloud	366
14.9 Remote Control Using Mobile Phone (Web Server)	368
14.10 Loading MicroPython Programs to the ESP32 DevKitC	372
14.10.1 Using the ampmy	372
14.10.2 Creating and Running a Program	374
14.10.3 Running a Program at Boot Time	375
14.11 Using the Real Time Clock (RTC)	378
14.11.1 Timestamping Temperature and Humidity Readings	379
Chapter 15 • Using ESP-IDF for programming the ESP32 DevKitC	381
15.1 Overview	381
15.2 Installing the ESP-IDF	381
15.2.1 Setting Up the Toolchain	381
15.2.2 Getting ESP-IDF	382
15.2.3 Path to ESP-IDF	383
15.2.4 Required Python Packages	383
15.3 PROJECT 1 – Flashing an LED	384
15.3.1 Description	384
15.3.2 Aim	384
15.3.3 Programming and Uploading Steps	384
15.3.4 Description of the Program	387

15.3.5 Project Files	388
15.4 PROJECT 2 – Three LEDs Flashing at Different Rates	388
15.4.1 Description.	388
15.4.2 Aim	388
15.4.3 Circuit Diagram.	388
15.4.4 Programming and Uploading Steps	389
15.4.5 Description of the Program.	391
15.5 Summary	391
Chapter 16 • Security features Of ESP32.	392
16.1 Overview	392
16.2 Flash Encryption	392
16.3 Secure Boot	393
16.4 Using espefuse.py	393
16.4.1 eFUSE Summary.	393
16.4.2 Write Protecting FLASH_CRYPT_CNT	395
16.5 Summary	395
APPENDIX A - List of Components used in the book	396
APPENDIX B - ESP32 DevKitC features used in projects in the book.	397

Preface

Wi-Fi (or WiFi) networks are currently used nearly in all homes, offices, and public places in developed countries to connect devices such as personal computers, smart mobile phones, modern printers, and tablets to the internet. A wireless Access Point (AP) is used to provide interconnection between all these devices. APs normally operate in the 2.4 GHz frequency band and their ranges are limited to a maximum of 100 metres.

Connection to a Wi-Fi AP requires a compatible wireless network interface module on the devices to be connected to the internet. Laptop and desktop computers use wireless Network Interface Cards (NICs) to establish communication with the AP and then with the internet. All that is required for the communication is the Wi-Fi name (SSID name) of the AP and the password. In the past, such NIC hardware used to be bulky and expensive and used to consume high currents, requiring large power supplies. Nowadays, NIC hardware is available in many different forms, sizes and prices, such as built-in electronic modules, external cards, or external flash memory size USB modules.

Microcontrollers are very popular electronic chips and are commonly used in many domestic, commercial, and industrial electronic monitoring and control applications. It is estimated that there are more than 50 microcontrollers in every house in developed countries. Some domestic equipments that have embedded microcontrollers are microwave ovens, printers, keyboards, computers, tablets, washing machines, dish washers, smart televisions, mobile phones, and many more.

A tiny chip called the ESP8266 has recently made it possible to interface any type of microcontroller to a Wi-Fi AP. The ESP8266 is a low-cost tiny Wi-Fi chip that has a full built-in TCP/IP stack and a 32-bit microcontroller unit. This chip, produced by Shanghai based Chinese manufacturer Espressif System is IEEE 802.11 b/g/n Wi-Fi compatible with on-chip program and data memory, and general purpose input-output ports. Several manufacturers have incorporated the ESP8266 chip in their hardware products (e.g. ESP-xx, NodeMCU etc) and offer these products as a means of connecting a microcontroller system such as the Android, PIC microcontroller or others to a Wi-Fi. The ESP8266 is a low-power chip and costs only a few dollars.

In addition to their ESP8266 chip, the same company has recently developed a new microcontroller chip named the ESP32. This is big brother of the ESP8266 and can be used in all projects that the ESP8266 is currently used. In addition to all the ESP8266 features, the ESP32 provides Bluetooth communications capability, larger SRAM data memory, more GPIOs, more interface signals, touch sensor, temperature sensor, higher CPU speed, CAN bus connectivity, higher resolution ADC converters, DAC converters, and security features.

This book compliments the earlier book by the author, entitled "The Official ESP32 Book: Discover All The Power" and is an advanced version of the earlier book. The book describes main hardware and software features of the ESP32 chip and provides many projects using these features. The main aim of the book is to teach how to use the ESP32 hardware and software in practical projects, especially using the highly popular ESP32 development board

DevKitC. Many basic, simple, and intermediate level projects are given in the book are based on the ESP32 DevKitC development board, using the highly popular Arduino IDE and also the MicroPython programming language.

Attempt has been made by the author to cover all features of the ESP32 in projects. In particular, the following features of the ESP32 processor are covered in projects in this advanced version of the book:

- General purpose input-output ports
- Touch sensors
- External interrupts
- Timer interrupts
- I2C
- I2S
- SPI
- PWM
- ADC
- DAC
- UART
- Hall sensor
- Temperature sensor
- Infrared controller
- Reading and writing to SD card
- Reading and writing to flash memory
- RTC timer
- Chip ID
- Security and encryption
- Wi-Fi and network programming
- Bluetooth BLE and Bluetooth classic programming
- Communication with the Android mobile phone
- Low power design
- ESP-IDF programming

There is some level of overlap with the earlier ESP32 book of the author. Additionally, this advanced version of the book makes an introduction to the native Espressif Systems ESP32 language ESP-IDF, and gives example projects using this powerful programming language in projects. The projects have been organized in increasing levels of difficulty and the readers are encouraged to read the projects in the given order.

A hardware kit is available from Elektor, especially prepared for the book. The kit contains all the components used in the projects in this book. With the help of this hardware kit it should be easy and fun to build the projects in the book.

We hope that you enjoy reading the book and at the same time learn how to use the ESP32 processor in your future projects.

Dogan Ibrahim

London, 2018

Chapter 1 • The ESP32 Processor

1.1 Overview

The ESP8266 processor has been a highly popular processor costing less than \$10 and it is basically a Wi-Fi enabled microcontroller with GPIOs that can be used in small monitoring and control applications. The ESP8266 was developed by Shanghai based Chinese manufacturer Espressif Systems and incorporates full TCP/IP stack. There is vast amount of information, tutorials, data sheets, applications notes, books, and projects based on the ESP8266. Several companies have created small development boards based on this processor, such as the ESP8266 Arduino and NodeMCU series.

Recently, Espressif has released a new and more powerful processor than the ESP8266, called the ESP32. Although ESP32 has not been developed to replace the ESP8266, it improves on it in many aspects. The new ESP32 processor not only has Wi-Fi support, but it also has a Bluetooth communications module, enabling the processor to communicate with Bluetooth compatible devices. The ESP32 CPU is the 32-bit Xtensa LX6, which is very similar to the ESP8266 CPU, but in addition it has two cores, more data memory, more GPIOs, higher CPU speed, ADC converters with higher resolution, DAC converter, and CAN bus connectivity.

The basic specifications of the ESP32 processor are summarised below:

- 32-bit Xtensa RISC CPU: Tensilica Xtensa LX6 dual-core microcontroller
- Operation speed 160 to 240 MHz
- 520 KB SRAM memory
- 448 KB ROM
- 16 KB SRAM (in RTC)
- IEEE 802.11 b/g/ne/I Wi-Fi
- Bluetooth 4.2
- 18 channels x 12-bit ADC
- 2 channel x 8-bit DAC
- 10 touch sensors
- Temperature sensor
- 36 GPIOs
- 4 x SPI
- 2 x I2C
- 2 x I2S
- 3 x UART
- 1 x CAN bus 2.0
- SD memory card support
- 2.2 V – 3.36 V operation
- RTC timer and watchdog
- Hall sensor
- 16 channels PWM
- Ethernet interface
- Internal 8 MHz, and RC oscillator

- External 2 MHz – 60 MHz, and 32 kHz oscillator
- Cryptographic hardware acceleration (AES, HASH, RSA, ECC, RNG)
- IEEE 802.11 security features
- 5 µA sleep current

Table 1.1 shows a comparison of the basic features of ESP32 and ESP8266 processors.

Specifications	ESP32	ESP8266
CPU	32-bit Xtensa LX6 dual-core	32-bit Xtensa L106 single-core
Operating frequency	160 MHz	80 MHz
Bluetooth	Bluetooth 4.2	None
Wi-Fi	Yes (HT40)	Yes (HT20)
SRAM	512 KB	160 KB
GPIOs	36	17
Hardware PWM	1	None
Software PWM	16	8
SPI/I2C/I2S/UART	4/2/2/2	2/1/2/2
CAN	1	None
ADC	12-bit	10-bit
Touch sensor	10	None
Temperature sensor	1	None
Ethernet MAC interface	1	None

Table 1.1 Comparison of ESP32 and ESP8266

1.2 The Architecture of ESP32

Figure 1.1 shows the functional block diagram of the ESP32 processor (see ESP32 Datasheet, Espressif Systems, 2017). At the heart of the block is the dual-core Xtensa LX6 processor and memory. At the left hand side we can see the peripheral interface blocks such as SPI, I2C, I2S, SDIO, UART, CAN, ETH, IR, PWM, temperature sensor, touch sensor, DAC, and ADC. The Bluetooth and Wi-Fi modules are situated at the top middle part of the block diagram. The clock generator and the RF transceiver are located at the top right hand of the block. The middle right hand is reserved for the cryptographic hardware accelerator modules such as the SHA, RSA, AES, and RNG. Finally, the bottom middle part is where the RTC, PMU, co-processor, and the recovery memory are.

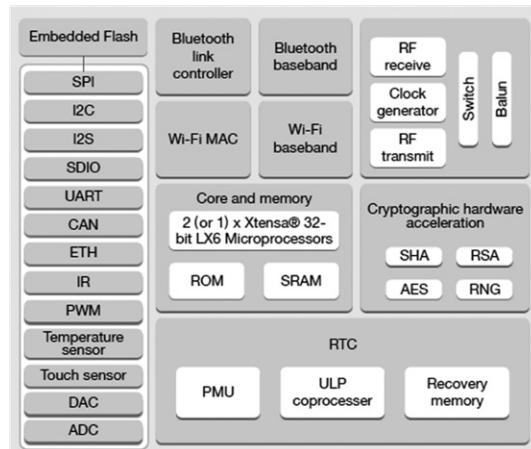


Figure 1.1 Functional block diagram of ESP32 processor

Figure 1.2 shows the system structure, consisting of two core Harvard architecture CPUs named PRO_CPU (for Protocol CPU) and APP_CPU (for Application CPU). The modules in the middle of the two CPUs are common to both CPUs. Detailed information about the internal architecture of the ESP32 can be obtained from the ESP32 Technical Reference Manual (Espressif Systems, 2017). Some information about the internal modules are given below.

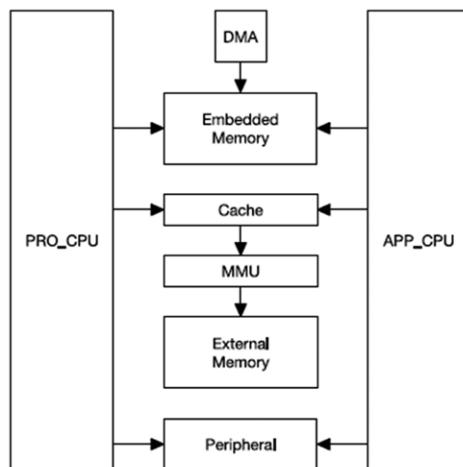


Figure 1.2 System structure

1.2.1 The CPU

The CPU can operate at up to 240 MHz and supports 7-stage pipelining with a 16/24-bit instruction set. Floating Point Unit and DSP instructions such as 32-bit multiplier, 32-bit divider, and 40-bit MAC are supported. Up to 70 external and internal interrupt sources with 32 interrupt vectors are available. Debugging can be done with the JTAG interface.

1.2.2 Internal Memory

520 KB SRAM and 448 KB ROM (for booting) are available on-chip. The Real Time Clock module contains 8 KB slow memory and 8 KB fast memory. 1 Kbit of eFuse is available with 256 bits used for the MAC address and chip configuration, and the remaining 768 bits reserved for customer applications.

1.2.3 External Memory

Up to 4 x 16 MB external flash and SRAM memory that can be accessed through a high-speed cache are supported. Up to 16 MB of the external flash are mapped onto the CPU code space and up to 8 MB of the external flash/SRAM are mapped onto the CPU data space. Although data read is supported both on the flash and SRAM, data writing is supported only on the SRAM.

1.2.4 General Purpose Timers

4 x 64-bit general purpose software controllable timers are supported by the ESP32 processor. The timers have 16-bit pre-scalers (2 to 65535) and auto-reload up/down counters. The timers can generate interrupts if configured.

1.2.5 Watchdog Timers

Three watchdog timers with programmable timeout values are available. Two watchdog timers, called the main Watchdog Timers are inside the general purpose timers, while the third one, called the RTC Watchdog Timer, is inside the RTC module. The actions taken when a watchdog timer resets can be: interrupt, CPU reset, core reset, and system reset.

1.2.6 The System Clock

An external crystal clock controls the system timing when the processor is reset. The clock frequency is typically 160 MHz, configured with the help of a PLL.

An 8 MHz accurate internal clock is also available. The programmer can either select the external or the internal clock.

1.2.7 Real Time Clock (RTC)

An RTC is provided that can be clocked using an external 32 kHz crystal, an internal RC oscillator (typically 150 kHz), an internal 8 MHz oscillator, or an internal 31.25 kHz clock derived by dividing the 8 MHz internal oscillator by 256.

1.2.8 General Purpose Input-Outputs (GPIOs)

There are 34 GPIOs that can be configured as digital, analog, or capacitive touch screen. Digital GPIOs can be configured to have internal pull-up resistors or pull-down resistors or set to high impedance state. Input pins can be configured to accept interrupts on either edges or on level changes.

1.2.9 Analog To Digital Converter (ADC)

The ESP32 processor includes 18 channels of 12-bit ADC. Small analog voltages can be measured by configuring some of the pins as programmable gain amplifiers.

1.2.10 Digital To Analog Converter (DAC)

The ESP32 processor includes two independent 8-bit DACs.

1.2.11 Hall Sensor

A hall sensor is available on the processor based on a resistor. A small voltage that can be measured by the ADC is developed when the sensor is in a magnetic field.

1.2.12 Temperature Sensor

An analog internal temperature sensor is available that can measure the temperature in the range -40 °C to +125 °C. The measured temperature is converted into digital form using an ADC. The measurement is affected by the temperature of the chip and the modules active inside the chip and thus the temperature sensor is only suitable for measuring temperature changes rather than measuring the absolute temperature.

1.2.13 Touch Sensor

Up to 10 capacitive touch sensors are provided that can detect the capacitive changes when a GPIO pin is in direct contact with a finger or any other suitable object.

1.2.14 UART

Three UARTs with speeds up to 5 Mbps are provided for RS232, RS485 and IrDA serial communications.

1.2.15 I2C Interface

ESP32 processor supports up to two I2C bus interfaces that can be configured as master or slave modes. The interface supports 400 Kbits/s fast transfer mode with 7-bit/10-bit addressing mode. External devices compatible with the I2C bus can be connected to these pins.

1.2.16 I2S Interface

ESP32 processor supports up to two I2S bus interfaces that can be configured in master or slave modes, in full or half duplex. The clock frequency can be from 10 kHz to 40 MHz.

1.2.17 Infrared Controller

Up to 8 channels of programmable infrared remote controller transmissions are supported by ESP32. The transmitting and receiving waveforms can be stored in shared 512 x 32-bit memory.

1.2.18 Pulse Width Modulation

Pulse Width Modulation (PWM) is used to control devices such as motors, electric heaters, smart lights and so on. ESP32 offers one programmable hardware PWM module and 16 software configurable PWM modules.

1.2.19 LED PWM

The LED PWM can be used to generate up to 16 independent digital waveforms with configurable duty cycles and periods. The duty cycle can be changed by software in a step by step mode.

1.2.20 Pulse Counter

Up to 8 channels of pulse counters are provided to capture pulses and count pulse edges. An interrupt can be generated when the count reaches a pre-defined value.

1.2.21 SPI Interface

Up to 4 SPI interfaces are supported by ESP32 in master and slave modes. External devices compatible with the SPI bus interface can be connected to these pins.

1.2.22 Hardware Accelerators

ESP32 supports hardware accelerators for implementing mathematical operations on algorithms such as AES, SHA, RSA and ECC. These accelerators help to increase the operation speed and also reduce the software complexity.

1.2.23 Wi-Fi

ESP32 includes a Wi-Fi module that can be used in projects to communicate with other Wi-Fi devices, such as mobile phones, PCs, laptops and iPads, through a network router.

1.2.24 Bluetooth

A Bluetooth module is included on the ESP32 processor. With the help of this module we can develop projects to communicate with other Bluetooth compatible devices, such as mobile phones, PCs, iPads, and others.

1.2.25 Controller Area Network (CAN)

ESP32 includes a CAN bus controller that can be programmed to communicate with other ESP32 processors or other CAN bus compatible devices.

1.2.26 SD Card Support

ESP32 supports SD memory cards, thus making it possible to store data on a memory card, for example.

1.3 ESP32 Development Boards

The ESP32 chip is highly complex and cannot easily be used on its own. There are several development boards available on the market based on the ESP32 chip. These development boards incorporate the ESP32 chip and associated hardware to simplify the task of project development based on the ESP32.

Some of the popular ESP32 development boards are:

- SparkFun ESP32 thing
- Geekcreit ESP32 Development Board
- HiLetgo ESP-WROOM-32 Development Board
- LoLin32 ESP32 Development Board
- Pycom LoPy Development Board
- ESP32 OLED Development Board
- Makerfocus ESP32 Development Board
- ESPS32 Test Board

- ESP32-EVB
- ESP32 Development Board by Pesky Products
- MakerHawk ESP32 Development Board
- Huzzah32 Development Board
- ESPe32
- NodeMCU-32s
- Node32S
- ESP32 DevKitC

1.4 Summary

In this chapter we have briefly looked at the architecture of the ESP32 chip. Additionally, a list of some of the popular ESP32 development boards are given in the chapter. The ESP32 DevKitC will be used as the development board in the remainder of this book.

In the next chapter we shall be looking at the hardware details of the ESP32 DevKitC and also see how this development board can be used in ESP32 based projects.

Chapter 2 • ESP32 DevKitC Development Board

2.1 Overview

In the last chapter we had a look at the architecture of the ESP32 processor and its basic features and advantages. We have also listed some of the popular ESP32 development boards available in the marketplace.

Currently, ESP32 DevKitC is one of the most popular development boards based on the ESP32 processor. All projects in this book are based on this development board. It is therefore important that we learn the architecture and the features of this board in detail.

In this chapter we shall be looking at the features of the ESP32 DevKitC development board in greater detail.

2.2 ESP32 DevKitC Hardware

ESP32 DevKitC is a small ESP32 processor based board developed and manufactured by Espressif. As shown in Figure 2.1, the board is breadboard compatible and has the dimensions 55 mm x 27.9 mm.

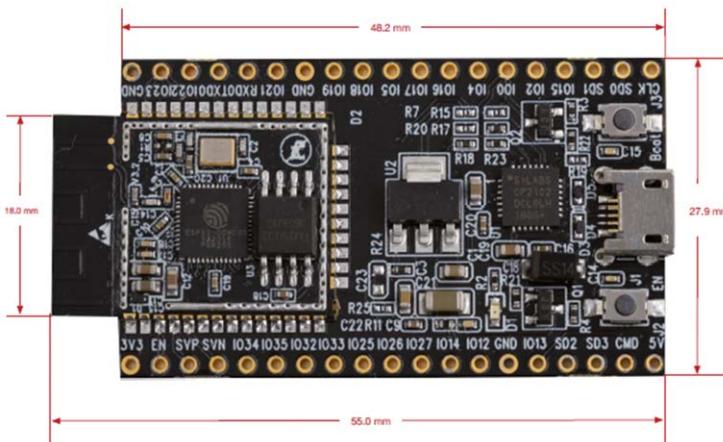


Figure 2.1 ESP32 DevKitC development board

The board has two connectors located along each side of the board for GPIO, clock, and power line interfaces. Each connector has 19 pins. As shown in Figure 2.2, the two connectors carry the following signals:

Left Connector	Right Connector
+3.3V	GND
EN	IO23
SVP	IO22
SVN	TXD0
IO34	RXD0
IO35	IO21
IO32	GND

IO33	IO19
IO25	IO18
IO26	IO5
IO27	IO17
IO14	IO16
IO12	IO4
GND	IO0
IO13	IO2
SD2	IO15
SD3	SD1
CMD	SD0
+5V	CLK

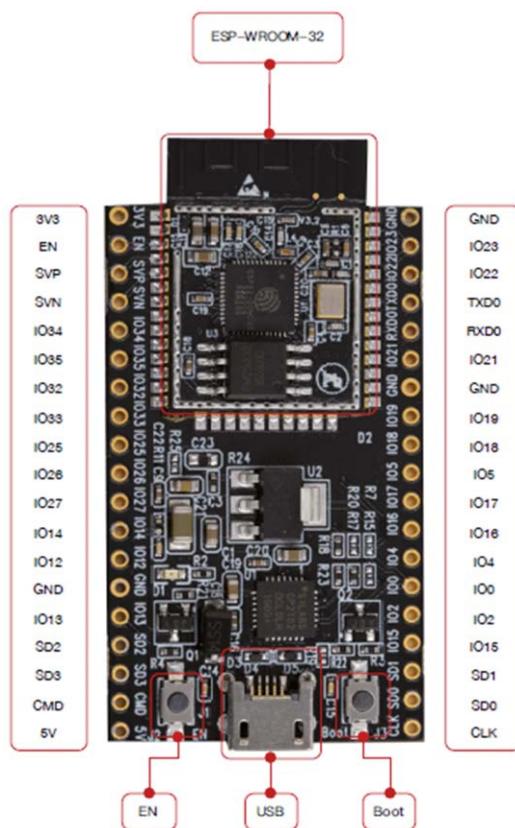


Figure 2.2 ESP32 DevKitC connectors

The board has a mini USB connector for connecting it to a PC. The board also receives its power from the USB port. Standard +5 V from the USB port is converted into +3.3 V on the board. In addition, two buttons are provided on the board named EN and BOOT, having the following functions:

EN: This is the reset button where pressing this button resets the board

BOOT: This is the download button. The board is normally in operation mode where the button is not pressed. Pressing and holding down this button and at the same time pressing the EN button starts the firmware download mode where firmware can be downloaded to the processor through the USB serial port.

The pins on the ESP32 DevKitC board have multiple functions. Figure 2.3 shows the functions of each pin. For example, pin 10 is shared with functions GPIO port 26, DAC channel 2, ADC channel 19, RTC channel 7, and RX01.

Figure 2.3 Multiple functions of each pin. Source: www.cnx-software.com

Note that GPIO34, GPIO35, GPIO36, GPIO37, GPIO38 and GPIO39 ports are input only and cannot be used as output ports (GPIO37 and GPIO38 are not available on the ESP32 board).

The board operates with a typical power supply of +3.3 V, although the absolute maximum is specified as +3.6 V. It is recommended that the current capacity of each pin should not exceed 6 mA, although the absolute maximum current capacity is specified as 12 mA. It is therefore important to use current limiting resistors while driving external loads such as LEDs.

Depending upon the configuration the RF power consumption during reception is around 80 mA and it can be in excess of 200 mA during a transmission.

2.3 Powering Up the ESP32 DevKitC

Programming of the ESP32 DevKitC requires the board to be connected to a PC through its mini USB port. The communication between the board and the PC takes place using the standard serial communication protocol.

ESP32 DevKitC is preloaded with firmware that can be used to test the board. This firmware is activated by default when power is applied to the board. Before communicating with the board, we have to run a terminal emulation software on our PC. There are several terminal emulation software available free of charge. Some examples are HyperTerm, Putty, X-CTU and so on. In this book we shall be using the Putty.

Putty is a popular terminal emulation program that runs on the PC. The program is available on the following website:

www.putty.org

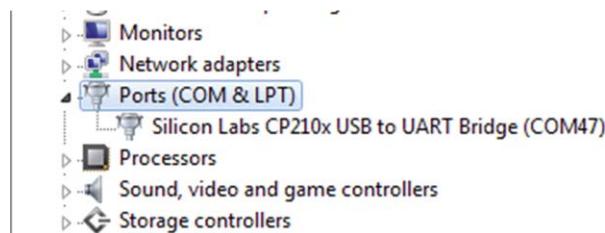


Figure 2.4 Hardware manager

The steps to start communication with the ESP32 DevKitC board are given below:

- Connect your ESP32 DevKitC to one of the USB ports of your PC. You should see the red LED on the board to turn on to indicate that power is applied to the board
- Find out the serial port number assigned to the USB port. This can be found from a display of the **Hardware Manager** as shown in Figure 2.4. In this example the required port number is COM47.
- Activate Putty by clicking on its icon.
- Set the following communication options in Putty (see Figure 2.5):

Connection type: Serial
Host name: COM47
Speed: 115200

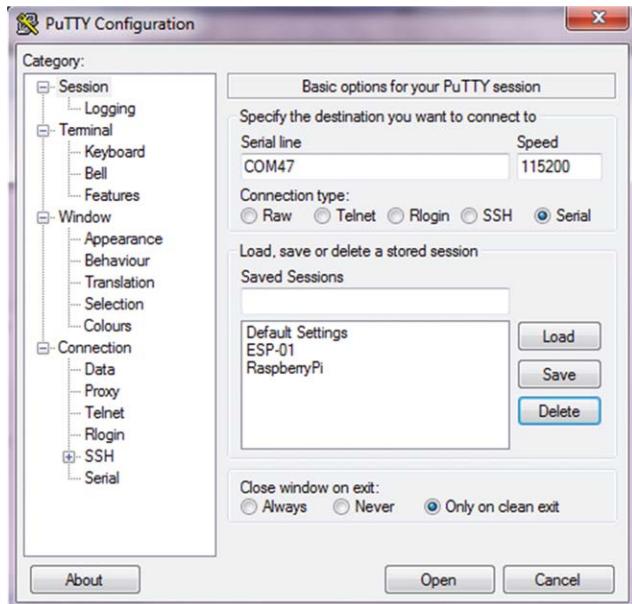


Figure 2.5 Putty communication options

- In addition, you may like to make the following configuration settings in Putty to make the display more readable. Click **Colours** under **Window** in the left pane and then:
 - Click on **Default Foreground**
 - Click on **Modify** and select black colour
 - Click on **Default Backround**
 - Click on **Modify** and select white colour
 - Click on **Cursor Text**
 - Click on **Modify** and select black colour
 - Click on **Cursor Colour**
 - Click on **Modify** and select black colour
- In addition, the font size can be increased if desired. Click on **Appearance** under **Windows** in the left pane and then click to change the font to 12 points, bold.
- Click on **Session** in the left pane and give a name to your session. e.g. ESP32 and click **Save** to save your configuration.
- Click **Open** to start the terminal emulation.

You should now reset your ESP32 DevKitC board by pressing the reset button (the button on the bottom left). The board will reset and will send messages to your terminal similar to the ones shown in Figure 2.6 (only part of the display is shown here).

```
:>ets Jun 8 2016 00:22:57
rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
ets Jun 8 2016 00:22:57

rst:0x10 (RTCWDT_RTC_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0x00
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3ffc0000,len:0
load:0x3ffc0000,len:2304
load:0x40078000,len:3788
ho 0 tail 12 room 4
load:0x40098000,len:532
entry 0x4009813c

*****
*      hello espressif ESP32!      *
*      2nd boot is running!       *
*      version (V0.1)            *
*****
```

Figure 2.6 Messages after the *ESP32 DevKitC* is reset

If you do not see anything on your display then you should check the connection between the *ESP32 DevKitC* board and your PC. Also make sure that the correct port number and communication speed are selected in Putty.

2.4 Summary

In this chapter we have seen the basic features and pin layout of the *ESP32 DevKitC* hardware development board.

In the next chapter we shall be learning how to program the *ESP32* processor using the *Arduino IDE*, and also how to upload an executable code to the *ESP32* processor.

Chapter 3 • Arduino IDE for the ESP32 DevKitC

3.1 Overview

By default, the ESP32 DevKitC is distributed with no programming firmware installed. It is therefore necessary to install a programming language firmware on the processor so that user programs can be developed and uploaded to the processor. Just like the ESP8266, the ESP32 processor is compatible with various programming languages such as C, MicroPython and so on.

Arduino IDE is one of the most commonly used development environments for microcontrollers, especially for the Arduino family of microcontrollers. This IDE is easy to use, supports many microcontrollers, and includes a very rich library of functions that make the programming easier. Most electrical/electronic engineering students and people whose hobby is electronics are familiar with using the Arduino IDE. In this section we shall be seeing how to install the ESP32 processor into the Arduino IDE on a Windows PC.

Several simple projects are given in the next chapter on using the ESP32 DevKitC board with the Arduino IDE. It is important to note that using the Arduino IDE has some limitations and not all features of the ESP32 can be programmed with this IDE. We shall be seeing in future chapters how to install and create the ESP-IDF development environment which is the framework developed and supported by Espressif for the ESP32 processors. Although using the ESP-IDF may be more complex than the Arduino IDE programming, it has the advantage that it enables the programmer to access all the features of the ESP32 processor.

3.2 Installing the Arduino IDE for the ESP32 DevKitC

The steps on installing the ESP32 support to the Arduino IDE on a Windows PC are provided in this section. If you have already installed ESP32 on Arduino IDE then you must delete the Espressif folder from your Arduino directory before re-installing it.

The steps to install the ESP32 on Arduino IDE are as follows:

- Download and install the latest version of Arduino IDE from the following web site:

<https://www.arduino.cc/en/Main/Software>

- Open your Arduino IDE and click File -> Preferences to open the Preferences window. Locate text box Additional Board Manager URLs at the bottom of the window and enter the following text as shown in Figure 3.1. If the text box contains another URL, add the new URL after separating it with a comma:

https://dl.espressif.com/dl/package_esp32_index.json

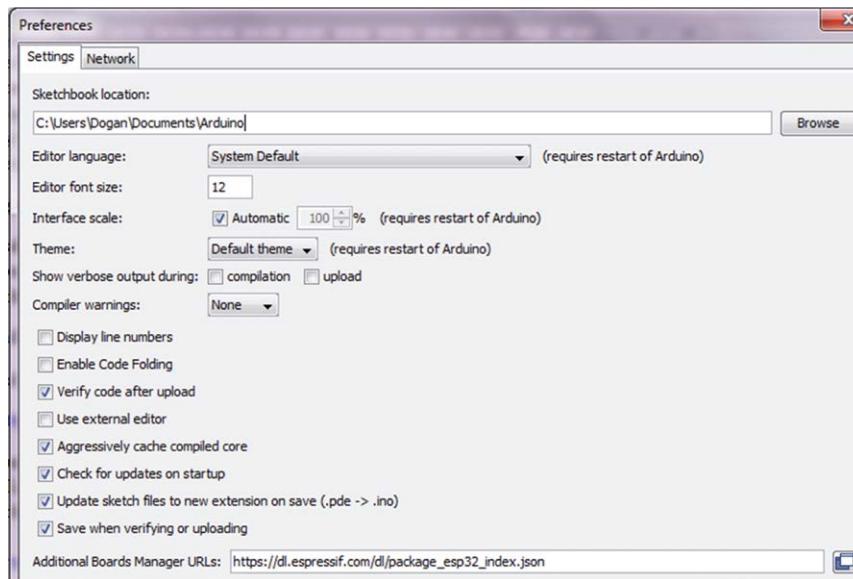


Figure 3.1 Preferences window

- Click **OK**
- Click **Tools -> Boards -> Board Managers** window and search for ESP32 as shown in Figure 3.2.

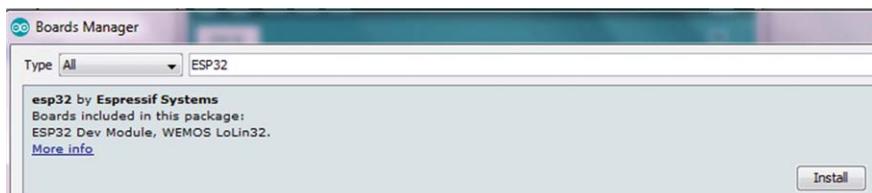


Figure 3.2 Search for ESP32]

- Click **Install** button. You should see the Installed message as shown in Figure 3.3. **Close** the window.

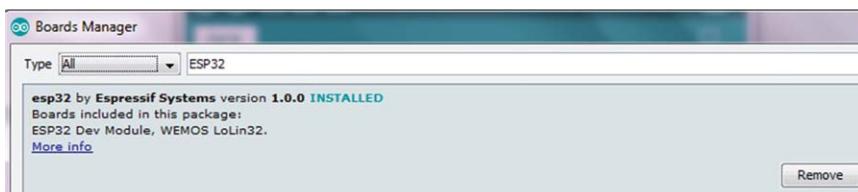


Figure 3.3 ESP32 installed

- We should now test the installation to make sure that all the necessary files have been loaded. We shall make use of one of the supplied example applications to make sure that our program can be uploaded to the ESP32 processor and is working correctly
- Plug in your ESP32 DevKitC to your PC and start the Arduino IDE
- Select **Tools -> Board ->ESP32 Dev Module** as shown in Figure 3.4.

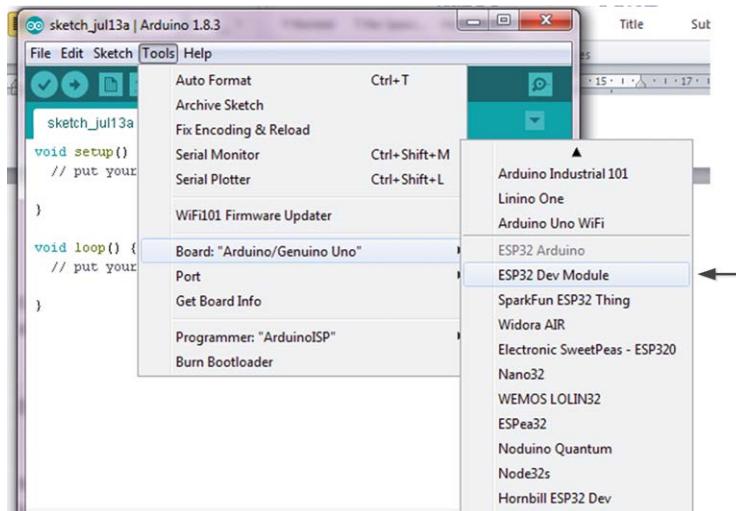


Figure 3.4 Select the ESP32 Dev Module

- Select the serial port number. In this example this is COM47 as shown in Figure 3.5

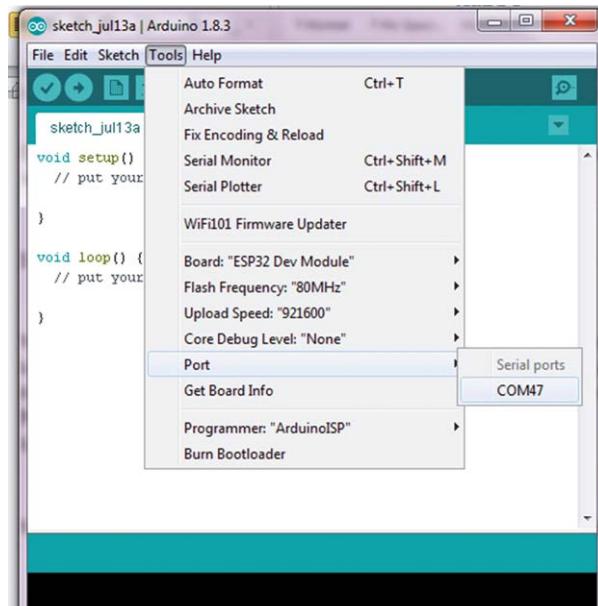


Figure 3.5 Select the serial port

- Open the example program in **File -> Examples -> WiFi (in ESP32) -> WiFi Scan** as shown in Figure 3.6

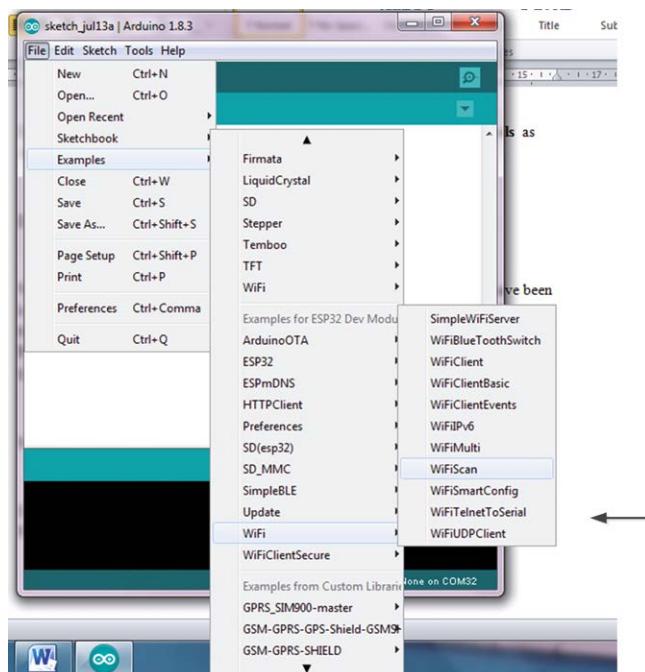


Figure 3.6 Open the example program WiFi Scan

- You should see the program as in Figure 3.7



```

  WiFiScan | Arduino 1.8.3
  File Edit Sketch Tools Help
  WiFiScan
  /*
  * This sketch demonstrates how to scan WiFi networks.
  * The API is almost the same as with the WiFi Shield library,
  * the most obvious difference being the different file you need
  */
  #include "WiFi.h"

  void setup()
  {
    Serial.begin(115200);

    // Set WiFi to station mode and disconnect from an AP if it was
    WiFi.mode(WIFI_STA);
    WiFi.disconnect();
    delay(100);

    Serial.println("Setup done");
  }

```

Figure 3.7 Example program to test the installation

- Now, we will upload the program to our ESP 32 DevKitC. First of all we have to put the device into firmware upload mode. Press and hold the **BOOT** button (the one on the right hand side). Press the **Sketch -> Upload** button in Arduino IDE to compile and upload your program to the ESP32 DevKitC. Wait until finished. After a successful process you should see a screen similar to the one shown in Figure 3.8 Release the **BOOT** button at the end of uploading.

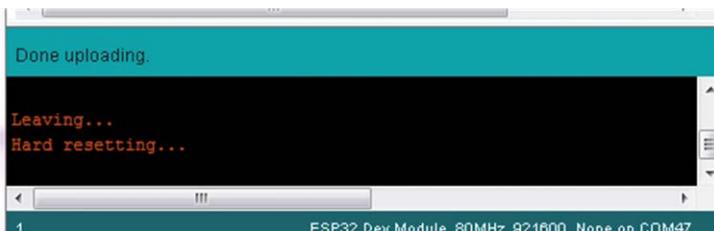
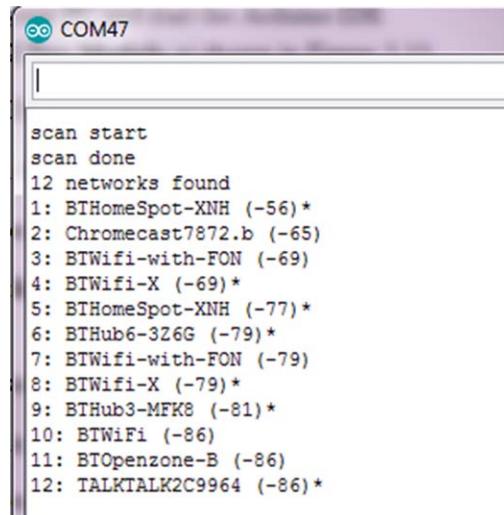


Figure 3.8 End of uploading

Now, open the Arduino IDE Serial monitor by clicking **Tools -> Serial Monitor**. Make sure that the communication baud rate is set to 115200. You should see a list of the access points close to you as shown in Figure 3.9. The list is updated every 5 seconds.

At this point we have successfully installed the ESP32 processor into our Arduino IDE.



```
scan start
scan done
12 networks found
1: BTHomeSpot-XNH (-56)*
2: Chromecast7872.b (-65)
3: BTWifi-with-FON (-69)
4: BTWifi-X (-69)*
5: BTHomeSpot-XNH (-77)*
6: BTHub6-3Z6G (-79)*
7: BTWifi-with-FON (-79)
8: BTWifi-X (-79)*
9: BTHub3-MFK8 (-81)*
10: BTWiFi (-86)
11: BTOpenzone-B (-86)
12: TALKTALK2C9964 (-86)*
```

Figure 3.9 List of close access points

3.3 Summary

In this chapter we have seen how to install the ESP32 processor into the Arduino IDE so that we can develop programs using the Arduino IDE and then upload them to our ESP32 DevKitC.

In the next chapter we shall be developing simple projects using the ESP32 DevKitC development board with the Arduino IDE.

Chapter 4 • Basic projects using the Arduino IDE and the ESP32 DevKitC

4.1 Overview

In the last chapter we have seen how to update the Arduino IDE so that the programs developed using this IDE can be compiled and downloaded to the ESP32 DevKitC development board.

In this chapter we will be developing basic projects using the Arduino IDE as the Integrated Development Environment, and then compile and download these projects to our ESP32 DevKitC. Notice that all the project given in this book have been tested and are working. The following sub-headings will be used for each project so that the projects can be described fully:

- Project title
- Project description
- Aim of the project
- Project block diagram
- Project circuit diagram
- Project construction
- Project operation (PDL)
- Project program listing
- Description of the program
- Additional work (optional)

The operation of the projects will be described using the **Program Description Language** (PDL). This is a free format English like description of the operation of a program using keywords such as BEGIN, END, IF, THEN, ELSE, ENDIF, WHILE, REPEAT, DO, DO FOREVER, ENDDO and so on. PDLs are preferred to Flow Charts as they consists of simple texts and do not have any graphics. Also, using PDLs create structured programs without any **goto** statements.

4.2 PROJECT 1 – Flashing LED

4.2.1 Description

In this project an LED is connected to GPIO port 23 of the ESP32 DevKitC. The LED is flashed every second.

4.2.2 The Aim

The aim of this project is to show how an LED can be connected to a GPIO port and how it can be turned on and off.

4.2.3 Block diagram

Figure 4.1 shows the block diagram of the project.



Figure 4.1 Block diagram of the project

4.2.4 Circuit Diagram

Figure 4.2 shows the pin configuration at the top right hand side of the ESP32 DevKitC. The circuit diagram of the project is shown in Figure 4.3. The LED is connected to GPIO port 23 (second pin at the top right hand side) through a current limiting resistor. It can be assumed that the voltage drop across the LED is 2 V. Therefore, for an LED current of 4 mA, the required value of the current limiting resistor R is found as:

$$R = (3.3 \text{ V} - 2 \text{ V}) / 4 \text{ mA} = 325 \text{ ohm. The nearest resistor chosen is } 330 \text{ ohm.}$$

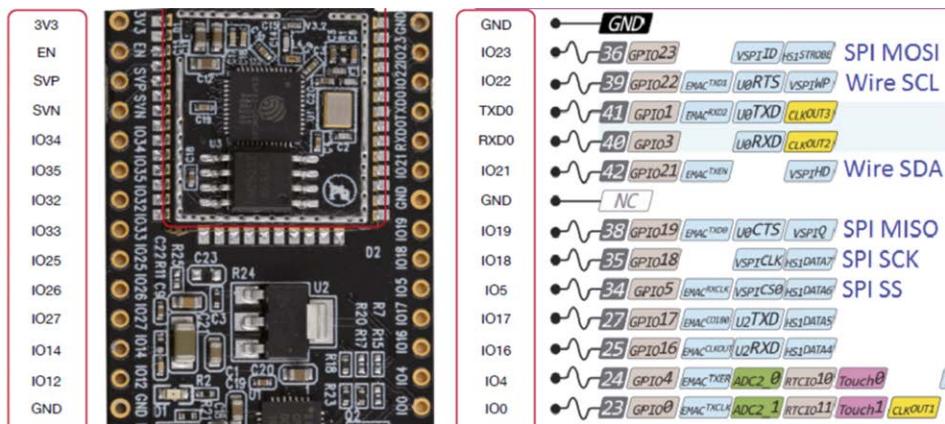


Figure 4.2 Top right hand side ESP32 DevKitC pin configuration

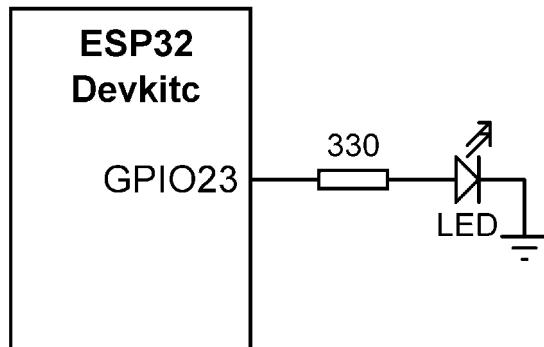


Figure 4.3 Circuit diagram of the project

4.2.5 Construction

The ESP32 DevKitC board is mounted on a breadboard as shown in Figure 4.4. The LED is connected to the board through a current limiting resistor.

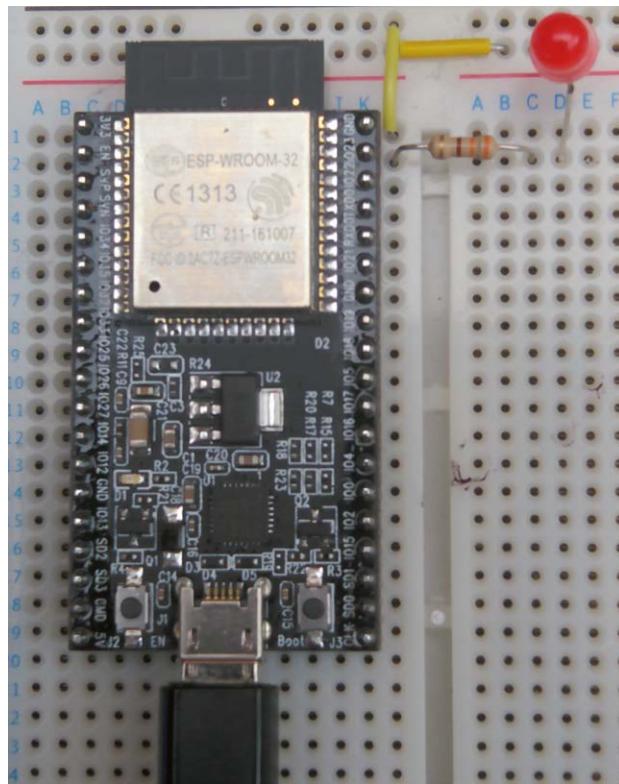


Figure 4.4 Project on a breadboard

4.2.6 PDL of the Project

The PDL of the project is shown in Figure 4.5. The program runs in an endless loop in which the LED is flashed every second.

```
BEGIN
    Assign LED to port pin GPIO23
    Configure port pin GPIO23 as output
DO FOREVER
    Set LED HIGH
    Wait 1 second
    Set LED LOW
    Wait 1 second
ENDDO
END
```

Figure 4.5 PDL of the project

4.2.7 Program Listing

The program listing of the project is very simple and is shown in Figure 4.6 (program: FlashLED).

```
*****FLASHING LED*****
=====
*
* In this program an LED is connected to port GPIO23 of the
* ESP32 DevKitC. The program flashes the LED every second
*
* Program: FlashLED
* Date   : July, 2017
*****/
#define LED 23

void setup()
{
    pinMode(LED, OUTPUT);
}

void loop()
{
    digitalWrite(LED, HIGH);
    delay(1000);
    digitalWrite(LED, LOW);
    delay(1000);
}
```

Figure 4.6 Program listing

4.2.8 Program Description

At the beginning of the program variable name **LED** is assigned to GPIO port 23. Then the LED port is configured as an output port. The main program is run continuously in a loop, where inside this loop the LED is turned ON (HIGH) and OFF (LOW) with one second delay between each output.

Remember that you should press and hold the **BOOT** button on the ESP32 DevKitC during the program compile and download process.

4.2.9 Suggestions

Modify the program in Figure 4.6 so that the LED ON time is 2 seconds and the OFF time is 5 seconds.

4.3 PROJECT 2 – Lighthouse Flashing LED

4.3.1 Description

In this project an LED is connected to GPIO port 23 of the ESP32 DevKitC as in Project 1. The LED is flashed in a group of two quick flashes every second. The flashing rate is assumed to be 200 ms. This type of flashing is identified as **GpFl(2)** in maritime lighthouse lights. Thus, the required flashing sequence can be expressed as follows:

```
LED ON  
Wait 200 ms  
LED OFF  
Wait 100 ms  
LED ON  
Wait 200 ms  
LED OFF  
Wait 100 ms  
Wait 400 ms
```

4.3.2 The Aim

The aim of this project is to show how an LED can be connected to a GPIO port and how it can be turned on and off at different rates.

4.3.3 Block diagram:

The block diagram of the project is as in Figure 4.1.

4.3.4 Circuit Diagram

The circuit diagram of the project is as in Figure 4.3.

4.3.5 Construction

The LED is connected to the ESP32 DevKitC board through the current limiting resistor as in Figure 4.4.

4.3.6 PDL of the Project

The PDL of the project is shown in Figure 4.7. The program runs in an endless loop where the LED is flashed every second with the lighthouse sequence **GpFl(2)**.

```
BEGIN
    Assign LED to port pin GPIO23
    Configure port pin GPIO23 as output
    DO FOREVER
        Set LED HIGH
        Wait 200 ms
        Set LED LOW
        Wait 100 ms
        Set LED HIGH
        Wait 200 ms
        Set LED LOW
        Wait 100 ms
        Wait 400 ms
    ENDDO
END
```

Figure 4.7 PDL of the project

4.3.7 Program Listing

The program listing of the project is very simple and is shown in Figure 4.8 (program: Lighthouse).

```
/*
 *          LIGHHOUSE FLASHING LED
 *=====
 *
 * In this program an LED is connected to port GPIO23 of the
 * ESP32 DevKitC. The program flashes the LED with the lighthouse
 * lighting sequence GpFl(2)
 *
 * Program: Lighthouse
 * Date   : July, 2017
 */
#define LED 23
#define ON HIGH
#define OFF LOW

void setup()
{
    pinMode(LED, OUTPUT);
}

void loop()
```

```
{
    digitalWrite(LED, ON);
    delay(200);
    digitalWrite(LED, OFF);
    delay(100);
    digitalWrite(LED, ON);
    delay(200);
    digitalWrite(LED, OFF);
    delay(100);
    delay(400);
}
```

Figure 4.8 Program listing

4.3.8 Program Description

At the beginning of the program variable name **LED** is assigned to GPIO port 23 as in Project 1. Then the LED port is configured as an output port. The main program is run continuously in a loop, in which the LED is turned ON (HIGH) and OFF (LOW) as specified in the lighthouse sequence **GpFl(2)**.

4.4 PROJECT 3 – Alternately Flashing LEDs

4.4.1 Description

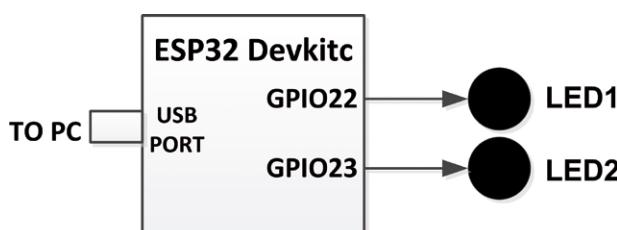
In this project two LEDs are connected to GPIO ports 22 and 23 of the ESP32 DevKitC.

4.4.2 The Aim

The aim of this project is to show how LEDs can be connected to a GPIO port and how they can be turned on and off alternately.

4.4.3 Block diagram:

The block diagram of the project is shown in Figure 4.9. LED1 is connected to GPIO port 22 and LED2 to GPIO port 23.

*Figure 4.9 Block diagram of the project*

4.4.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 4.10. The LEDs are connected to the port pins through 330 ohm current limiting resistors. See Figure 4.2 for pin configuration of ESP32 DevKitC.

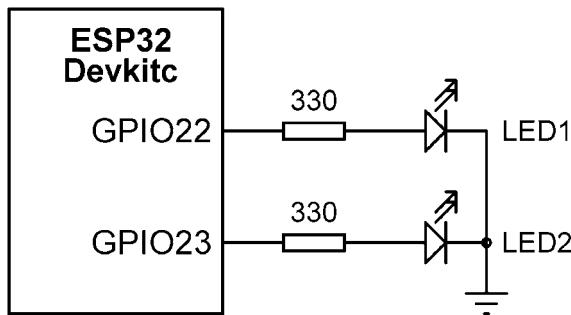


Figure 4.10 Circuit diagram of the project

4.4.5 Construction

A breadboard is used to construct the project. The LEDs are connected to the ESP32 DevKitC through the current limiting resistors as shown in Figure 4.11.

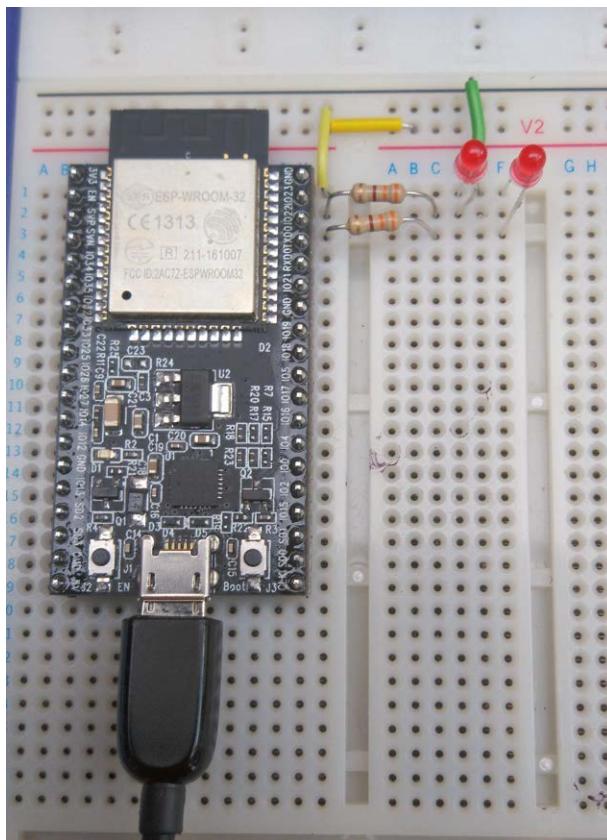


Figure 4.11 Project on a breadboard

4.4.6 PDL of the Project

The PDL of the project is shown in Figure 4.12. The program runs in an endless loop in which the LEDs are flashed alternately every second.

```
BEGIN
    Assign LED1 to port pin GPIO22
    Assign LED2 to port pin GPIO23
    Configure port pins GPIO22 and GPIO23 as outputs
DO FOREVER
    Set LED1 HIGH
    Set LED2 LOW
    Wait 1 second
    Set LED1 LOW
    Set LED2 HIGH
    Wait 1 second
ENDDO
END
```

Figure 4.12 PDL of the project

4.4.7 Program Listing

The program listing of the project is very simple and is shown in Figure 4.13 (program: **Altflash**).

```
*****
*                      ALTERNATE FLASHING LEDs
*                      =====
*
* In this program two LEDs are connected to port GPIO22 and
* GPIO23 of the ESP32 DevKitC. The program flashes the LEDs
* alternately with one second delay between each output
*
* Program: Altflash
* Date   : July, 2017
*****
#define LED1 22
#define LED2 23
#define ON HIGH
#define OFF LOW

void setup()
{
    pinMode(LED1, OUTPUT);
    pinMode(LED2, OUTPUT);
}
```

```
void loop()
{
    digitalWrite(LED1, HIGH);
    digitalWrite(LED2, LOW);
    delay(1000);
    digitalWrite(LED1, LOW);
    digitalWrite(LED2, HIGH);
    delay(1000);
}
```

Figure 4.13 Program listing

4.4.8 Program Description

At the beginning of the program variable names **LED1** and **LED2** are assigned to GPIO port 22 and 23 respectively. Then the port pins are configured outputs. The main program is run continuously in a loop, in which LED1 and LED2 are turned ON and OFF alternately with one second delay between each output.

4.5 PROJECT 4 – Rotating LEDs

4.5.1 Description

In this project four LEDs are connected to GPIO ports 23, 22, 1, and 3 (as shown in Figure 4.2, these port pins are next to each other). The LEDs are turned ON/OFF in a rotating manner where only one LED is ON at any given time. i.e. the required LED pattern is as follows:

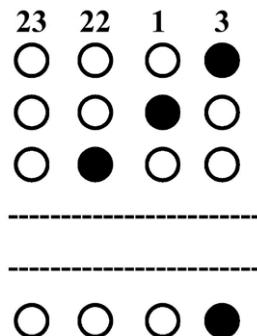


Figure 4.14 Rotating LEDs

The LED ON and OFF times are to be chosen as 500 ms and 100 ms respectively.

4.5.2 The Aim

The aim of this project is to show how an array and a **for** loop can be used in a program to control multiple devices (LEDs) connected to the GPIOs.

4.5.3 Block diagram:

The block diagram of the project is shown in Figure 4.15. LED1 to LED4 are connected to GPIO ports 23, 22, 1 and 3 respectively.

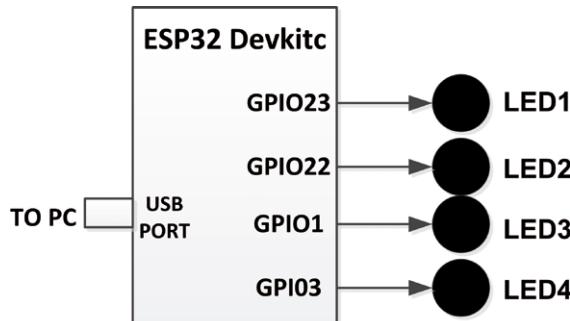


Figure 4.15 Block diagram of the project

4.5.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 4.16. The LEDs are connected to the port pins through 330 ohm current limiting resistors (see Figure 4.2 for the ESP32 DevKitC pin configuration)

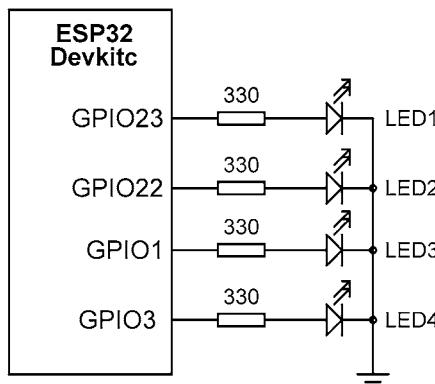


Figure 4.16 Circuit diagram of the project

4.5.5 Construction

A breadboard is used to construct the project. The LEDs are connected to the ESP32 DevKitC through the current limiting resistors as shown in Figure 4.17.

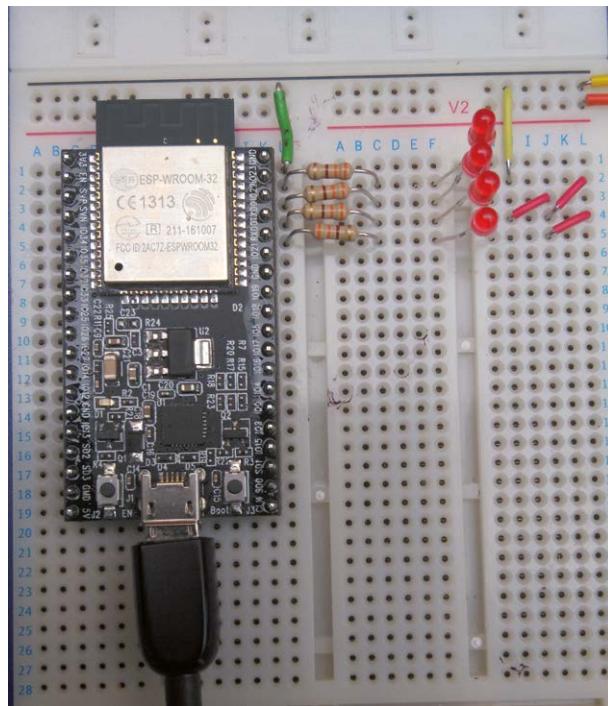


Figure 4.17 Project on a breadboard

4.5.6 PDL of the Project

The PDL of the project is shown in Figure 4.18. The program runs in an endless loop where the LEDs turn ON/OFF in a rotating pattern.

BEGIN

 Store LED port numbers in array LEDs

 Configure LED port pins as outputs and turn OFF the LEDs

DO FOREVER

DO FOR J = 0 TO 3

 Turn ON LED indexed by LEDs[J]

 Wait 500 ms

 Turn OFF LED indexed by LEDs[J]

 Wait 100 ms

ENDDO

ENDDO

END

Figure 4.18 PDL of the project

4.5.7 Program Listing

The program listing of the project is very simple and is shown in Figure 4.19 (program: **RotateLEDs**).

```
*****  
* ROTATING LEDs  
* ======  
  
* In this program four LEDs are connected to port GPIO23,  
* GPIO22, GPIO1, and GPIO3 and of the ESP32 Devkitc. The  
* program turns ON the LEDs in a rotating pattern. The ON  
* and OFF times are chosen as 500 ms and 100 ms so that a  
* nice rotating effect is displayed  
  
* Program: RotateLEDs  
* Date   : July, 2017  
*****/  
int LEDs[] = {23, 22, 1, 3};  
#define ON HIGH  
#define OFF LOW  
  
//  
// Set GPIO pins 23,22,1,3 as outputs and turn OFF the LEDs  
// to start with  
//  
void setup()  
{  
    unsigned char i;  
    for(i=0; i <=3; i++)  
    {  
        pinMode(LEDs[i], OUTPUT);  
        digitalWrite(LEDs[i], OFF);  
    }  
}  
  
//  
// Turn ON/OFF the LEDs in a rotating pattern  
//  
void loop()  
{  
    unsigned char j;  
    for(j = 0; j <=3; j++)  
    {  
        digitalWrite(LEDs[j], ON);  
    }  
}
```

```
    delay(500);
    digitalWrite(LEDs[j], OFF);
    delay(100);
}
}
```

Figure 4.19 Program listing

4.5.8 Program Description

At the beginning of the program an array called **LEDs** is set up to store the port numbers of the LEDs used in the project. Then, the GPIO ports that the LEDs are connected to are configured as output ports and the LEDs are turned OFF to start with. Inside the main program a **for** loop is set up to turn ON/OFF the LEDs connected to ports 23, 22, 1 and 3 in that order. The ON and OFF times are chosen as 500 ms and 100 ms so that a nice rotating LED effect is displayed.

4.5.9 Suggestions

You should increase the LED number to 8 and modify program 4.19 accordingly.

4.6 PROJECT 5 – Christmas Lights

4.6.1 Description

In this project 4 LEDs are connected to the ESP32 DevKitC as in the previous project. The LEDs are turned ON and OFF randomly every second.

4.6.2 The Aim

The aim of this project is to show how the random number generator function can be used in a program.

4.6.3 Block diagram:

The block diagram of the project is as in Figure 4.15.

4.6.4 Circuit Diagram

The circuit diagram of the project is as in Figure 4.16.

4.6.5 Construction

The project is built on a breadboard as shown in Figure 4.17.

4.6.6 PDL of the Project

The PDL of the project is shown in Figure 4.20.

```
BEGIN/Main
  Store LED port numbers in array LEDs
  Configure LED port pins as outputs
  DO FOREVER
    Generate a random number between 1 and 15
    Call Display with the number to turn ON the appropriate LED
```

```

Wait 1 second
ENDDO
END/Main

BEGIN/Display
Extract bit 3 of the number
IF bit = 1 THEN
    Turn ON LED1
ELSE
    Turn OFF LED1
ENDIF
Extract bit 2 of the number
IF bit = 1 THEN
    Turn ON LED2
ELSE
    Turn OFF LED2
ENDIF
Extract bit 1 of the number
IF bit = 1 THEN
    Turn ON LED3
ELSE
    Turn OFF LED3
ENDIF
Extract bit 0 of the number
IF bit = 1 THEN
    Turn ON LED4
ELSE
    Turn OFF LED4
ENDIF
END/Display

```

Figure 4.20 PDL of the project

4.6.7 Program Listing

The program listing of the project is very simple and is shown in Figure 4.21 (program: **Christmas**).

```

*****
*                               CHRISTMAS LIGHTS
*                               =====
*
* In this program four LEDs are connected to port GPIO23,
* GPIO22, GPIO1, and GPIO3 and of the ESP32 DevKitC. The
* program turns ON the LEDs in a random manner every second
* to give the effect of for example Christmas lights

```

```
*  
* Program: Christmas  
* Date   : July, 2017  
*****  
int LEDs[] = {23, 22, 1, 3};  
unsigned char Ran;  
  
//  
// Set GPIO pins 23,22,1,3 as outputs  
//  
void setup()  
{  
    unsigned char i;  
    for(i=0; i <=3; i++)  
    {  
        pinMode(LEDs[i], OUTPUT);  
    }  
    randomSeed(10);  
}  
  
//  
// Turn ON the appropriate LED  
//  
void Display(unsigned char No)  
{  
    digitalWrite(23,(No & B00001000));  
    digitalWrite(22,(No & B00000100));  
    digitalWrite(1, (No & B00000010));  
    digitalWrite(3, (No & B00000001));  
}  
  
//  
// Turn ON/OFF the LEDs randomly by generating a random number  
// between 1 and 15  
//  
void loop()  
{  
    Ran = random(1, 16);  
    Display(Ran);  
    delay(1000);  
}
```

Figure 4.21 Program listing

4.6.8 Program Description

At the beginning of the program an array called **LEDs** is set up to store the port numbers of the LEDs used in the project. Then, the GPIO ports that the LEDs are connected to are configured as output ports. Function **randomSeed** is called with an integer number to start the random number generator. Inside the main program a random number is generated between 1 and 15 (notice that the **random** function includes the lower bound, but excludes the upper bound). Then, function **Display** is called to extract the bits of this number and to turn ON or OFF the appropriate LED. Function **Display** receives an integer number between 1 and 15 and extracts the bits of this number. For example, if the number is 12 (binary 1100) then the left two LEDs (LED1 and LED2) are turned ON.

4.6.9 Modified Program

The program given in Figure 4.21 can be made more efficient by modifying the function **Display**. The new program listing is shown in Figure 4.22 (program: **Christmas2**). Here, function **Display** has two arguments: **No** is the number to be displayed as a binary bit pattern on the LEDs, and **L** is the width of the number in bits. Bits are extracted from the number and are sent to the LEDs to turn ON/OFF the correct LED. The program runs in an endless loop, where a random number is generated between 1 and 15 and stored in variable **Ran**. Function **Display** is then called as **Display(Ran, 4)** to turn ON/OFF the appropriate LEDs. Notice that here the width of the number is 4-bits (there are 4 LEDs).

```
/*
 *          CHRISTMAS LIGHTS
 *          =====
 *
 * In this program four LEDs are connected to port GPIO23,
 * GPIO22, GPIO1, and GPIO3 and of the ESP32 DevKitC. The
 * program turns ON the LEDs in a random manner every second
 * to give the effect of for example Christmas lights.
 *
 * In this modified program function Display is modified and
 * the program is more efficient and easier to maintain
 *
 * Program: Christmas2
 * Date    : July, 2017
 */
int LEDs[] = {23, 22, 1, 3};
unsigned char Ran;

//
// Set GPIO pins 23,22,1,3 as outputs
//
void setup()
{
    unsigned char i;
    for(i=0; i <=3; i++)
```

```
{  
    pinMode(LEDs[i], OUTPUT);  
}  
randomSeed(10);  
  
//  
// Turn ON the appropriate LED  
//  
void Display(unsigned char No, unsigned char L)  
{  
    unsigned char j, m, i;  
    m = L - 1;  
    for(i = 0; i < L; i++)  
    {  
        j = pow(2, m);  
        digitalWrite(LEDs[i], (No & j));  
        m--;  
    }  
}  
  
//  
// Turn ON/OFF the LEDs randomly by generating a random number  
// between 1 and 15  
//  
void loop()  
{  
    Ran = random(1, 15);  
    Display(Ran, 4);  
    delay(1000);  
}
```

Figure 4.22 Modified program

4.6.10 Suggestions

The number of LEDs can be increased by connecting more in series and in parallel and placed for example on a Christmas tree.

4.7 PROJECT 6 – Binary Up Counter with LEDs

4.7.1 Description

In this project 8 LEDs are connected to the ESP32 DevKitC. The program counts up from 0 to 255 where the count is displayed on the 8 LEDs in binary format.

4.7.2 The Aim

The aim of this project is to show how any 8 port pins can be grouped together and treated like an 8-bit output port.

4.7.3 Block diagram:

The block diagram of the project is shown in Figure 4.23. The LEDs are connected to the following GPIO port pins (see ESP32 DevKitC pin configuration in Figure 4.2):

23 (MSB)
22
1
3
21
19
18
5 (LSB)

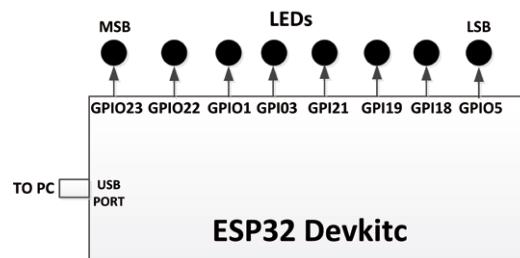


Figure 4.23 Block diagram of the project

4.7.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 4.24. 8 LEDs are connected to the GPIO ports through 330 ohm current limiting resistors.

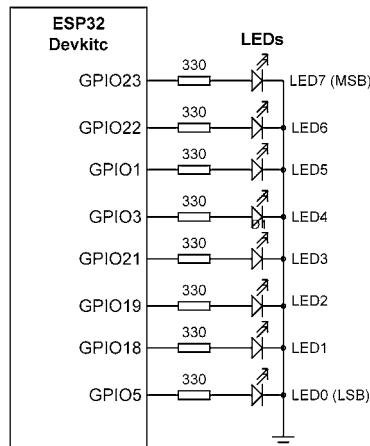


Figure 4.24 Circuit diagram of the project

4.7.5 Construction

The project is built on a breadboard as shown in Figure 4.25.

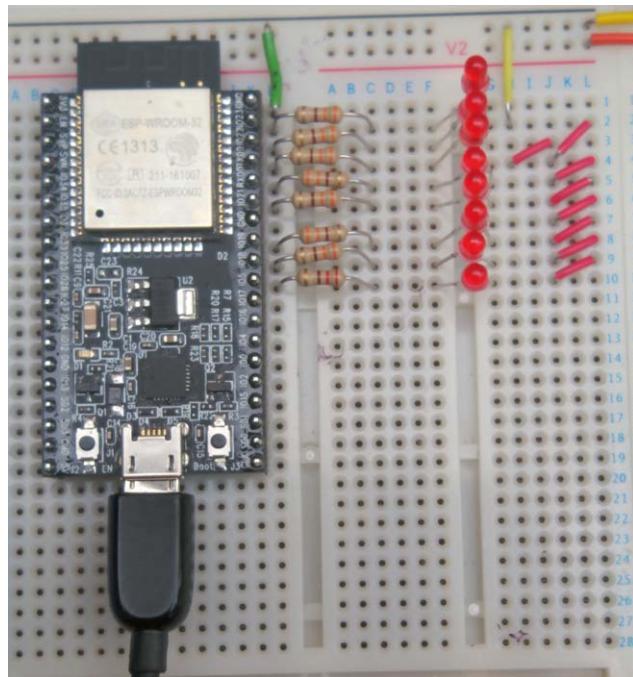


Figure 4.25 Constructing the project on a breadboard

4.7.6 PDL of the Project

The PDL of the project is shown in Figure 4.26.

BEGIN/Main

 Store LED port numbers in array LEDs

 Configure LED port pins as outputs

 Set Count to 0

DO FOREVER

Call Display with Count and 8 (width) as the arguments

 Increment Count

 Wait 1 second

ENDDO

END/Main

BEGIN/Display

 Extract bits of the number and send to appropriate GPIO pins

END/Display

Figure 4.26 PDL of the project

4.7.7 Program Listing

The program listing of the project is very simple and is shown in Figure 4.27 (program: **Counter**).

```
*****  
*          BINARY UP COUNTER  
*          ======  
*  
* In this program 8 LEDs are connected to port pins GPIO23,  
* GPIO22, GPIO1, GPIO3, GPIO21, GPIO19, GPIO18, and GPIO5  
* of the ESP32 DevKitC. The program counts up by one and  
* sends the count to the GPIO pins to turn ON/OFF the  
* appropriate LEDs so that the LEDs count up by one in binary.  
* One second delay is inserted between each output  
*  
*  
* Program: Counter  
* Date   : July, 2017  
*****  
int LEDs[] = {23, 22, 1, 3, 21, 19, 18, 5};  
unsigned char Count = 0;  
  
//  
// Set GPIO pins 23,22,1,3,21,19,18,5 as outputs  
//  
void setup()  
{  
    unsigned char i;  
    for(i=0; i < 8; i++)  
    {  
        pinMode(LEDs[i], OUTPUT);  
    }  
}  
  
//  
// Turn ON the appropriate LED  
//  
void Display(unsigned char No, unsigned char L)  
{  
    unsigned char j, m, i;  
    m = L - 1;  
    for(i = 0; i < L; i++)  
    {  
        j = pow(2, m);  
        digitalWrite(LEDs[i], (No & j));  
    }  
}
```

```
    m--;
}

}

//  
// Increment Count every second and send Count to GPIO  
// pins. Notice that the width is 8-bits (i.e. there are  
// 8 LEDs)  
//  
void loop()  
{  
    Display(Count, 8);  
    if(Count == 255)  
        Count = 0;  
    else  
        Count++;  
    delay(1000);  
}
```

Figure 4.27 Program listing

4.7.8 Program Description

At the beginning of the program an array called **LEDs** is set up to store the port numbers of the LEDs used in the project. Global variable **Count** is initialised to zero. Then, the GPIO ports where the LEDs are connected to are configured as output ports. Inside the main program function **Display** is called and **Count** is sent as the argument. This function extracts the bits of **Count** and turns ON/OFF the appropriate LEDs. Number 8 is also sent to function **Display** as an argument since there are 8 LEDs (i.e. the width of the number is 8 bits). The LEDs count up in binary from 0 to 255 continuously with one second delay between each count. You should see the pattern shown in Figure 4.28 on the LEDs.

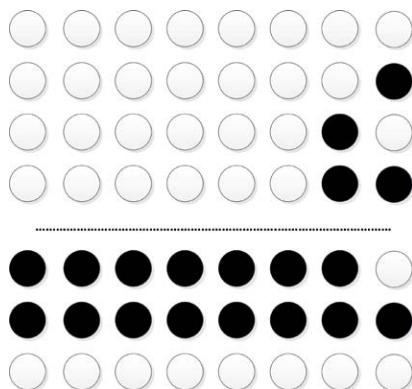


Figure 4.28 LED pattern

4.8 PROJECT 7 – Binary Up/Down Counter with LEDs

4.8.1 Description

In this project 8 LEDs are connected to the ESP32 DevKitC as in the previous project. In addition a push-button switch (or simply a **button**) is connected to GPIO pin 17. When the button is not pressed the program counts up, and when the button is held pressed the program counts down.

4.8.2 The Aim

The aim of this project is to show how a push-button switch can be connected to an ESP32 DevKitC GPIO port and how the port can be configured as an input.

4.8.3 Block diagram:

The block diagram of the project is shown in Figure 4.29. The LEDs are connected to the following GPIO port pins (see ESP32 DevKitC pin configuration in Figure 4.2):

23 (MSB)
22
1
3
21
19
18
5 (LSB)

The button is connected to GPIO pin 17.

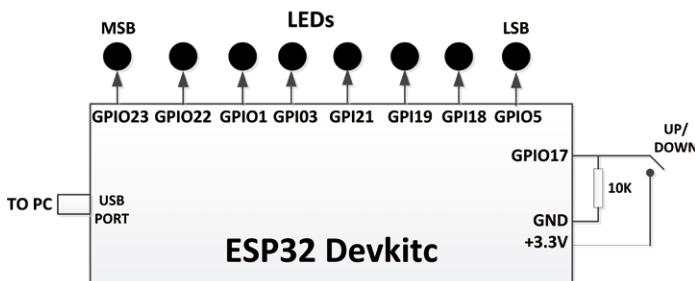


Figure 4.29 Block diagram of the project

4.8.4 Circuit Diagram

The circuit diagram of the project is as in Figure 4.30. 8 LEDs are connected to the GPIO ports through 330 ohm current limiting resistors. Notice that the button can be connected in two different ways to a GPIO port. In Figure 4.31, the output of the button is at logic 1 and goes to logic 0 when the button is pressed. In figure 4.32, the output of the button goes from logic 0 to logic 1 when the button is pressed. In this example the second method is used. Thus, GPIO pin is normally at logic 0 and goes to logic 1 when the button is pressed.

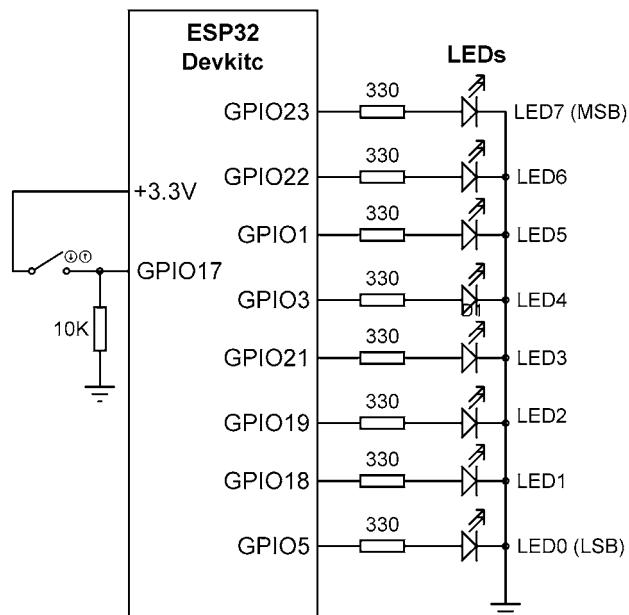


Figure 4.30 Circuit diagram of the project

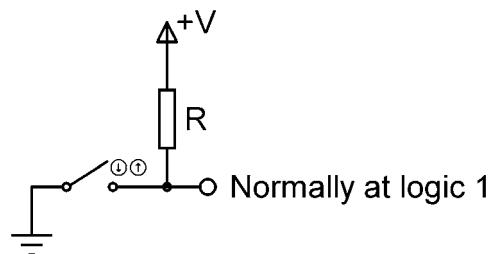


Figure 4.31 Button normally at logic 1

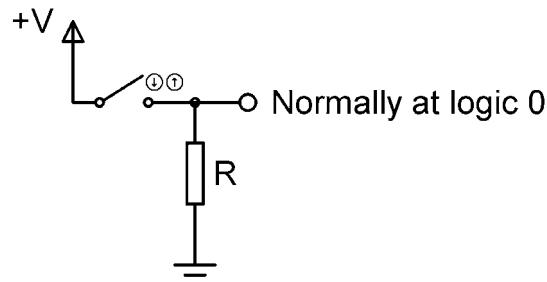


Figure 4.32 Button normally at logic 0

4.8.5 Construction

The project is built on a breadboard as shown in Figure 4.33.

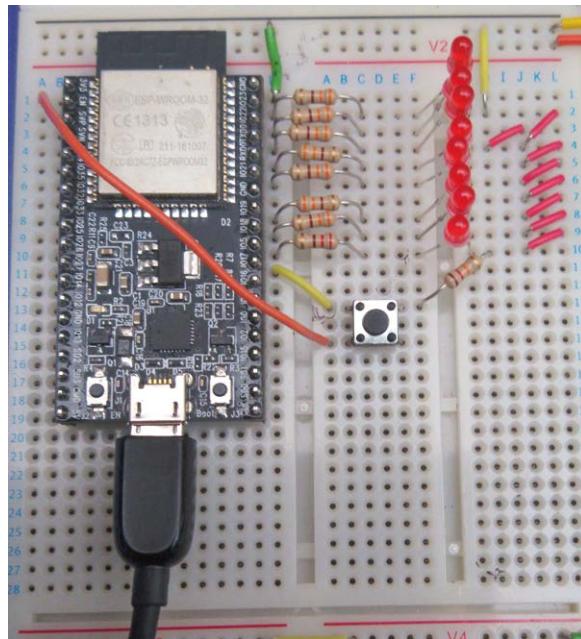


Figure 4.33 Constructing the project on a breadboard

4.8.6 PDL of the Project

The PDL of the project is shown in Figure 4.34.

```
BEGIN/Main
    Store LED port numbers in array LEDs
    Configure LED port pins as outputs
    Set Count to 0
DO FOREVER
    Call Display with Count and 8 (width) as the arguments
    IF Button not pressed THEN
        Increment Count
    ELSE IF Button is pressed THEN
        Decrement Count
    ENDIF
    Wait 1 second
ENDDO
END/Main

BEGIN/Display
    Extract bits of the number and send to appropriate GPIO pins
END/Display
```

Figure 4.34 PDL of the project

4.8.7 Program Listing

The program listing of the project is very simple and is shown in Figure 4.35 (program: **UpDown**).

```
/*********************************************
*                                BINARY UP/DOWN COUNTER
*                                =====
*
* In this program 8 LEDs are connected to port pins GPIO23,
* GPIO22, GPIO1, GPIO3, GPIO21, GPIO19, GPIO18, and GPIO5
* of the ESP32 DevKitC. In addition, a push-button switch
* is connected to port pin 17. Normally the switch output is
* at logic 0 and when the switch is pressed its output goes
* to logic 1. The program normally counts up. When the switch
* is pressed and held down then the program starts to count
* down from its last value. The appropriate LEDs are turned
* ON/OFF
*
*
* Program: UpDown
* Date   : July, 2017
*****/
#define Button 17
#define UP 0
#define DOWN 1
int LEDs[] = {23, 22, 1, 3, 21, 19, 18, 5};
unsigned char Count = 0;
unsigned char Button_State;

//
// Set GPIO pins 23,22,1,3,21,19,18,5 as outputs and GPIO pin 17
// as input
//
void setup()
{
    unsigned char i;
    for(i=0; i < 8; i++)
    {
        pinMode(LEDs[i], OUTPUT);
    }
    pinMode(Button, INPUT);
}

//
// Turn ON the appropriate LED
```

```
//  
void Display(unsigned char No, unsigned char L)  
{  
    unsigned char j, m, i;  
    m = L - 1;  
    for(i = 0; i < L; i++)  
    {  
        j = pow(2, m);  
        digitalWrite(LEDs[i], (No & j));  
        m--;  
    }  
}  
  
//  
// When teh button is not pressed increment Count every second  
// and send Count to GPIO pins. When the button is pressed  
// count down from the last value and send Count to GPIO pins  
// Notice that the width is 8-bits (i.e. there are 8 LEDs)  
//  
void loop()  
{  
    Display(Count, 8);  
    Button_State = digitalRead(Button);      // Read Button state  
    if(Button_State == UP)                  // If UP count  
    {  
        if(Count == 255)  
            Count = 0;                      // Reset Count  
        else  
            Count++;                      // Increment Count  
    }  
    else if(Button_State == DOWN)          // If DOWN count  
    {  
        if(Count == 0)  
            Count = 255;                  // Reset Count  
        else  
            Count--;                     // Decreament Count  
    }  
    delay(1000);                         // Wait 1 second  
}
```

Figure 4.35 Program listing

4.8.8 Program Description

At the beginning of the program an array called **LEDs** is set up to store the port numbers of the LEDs used in the project. Name **Button** is assigned to GPIO port 17, and names **UP** and **DOWN** are assigned to 0 and 1 respectively. Global variable **Count** is initialised to zero. Then, the GPIO ports that the LEDs are connected to are configured as output ports, and the GPIO port where the **Button** is connected to is configured as an input port. Inside the main program function **Display** is called and **Count** is sent as the argument. Then, the state of the **Button** is read: if the **Button** is not pressed (**Button_State** equal to **UP**) then an UP count is performed. If on the other hand the **Button** is pressed (**Button_State** equal to **DOWN**) then a DOWN count is performed. Notice that variable **Count** is reset at the end of the counts. i.e. during an up count when the count reaches to 255 it is reset to 0 on the next cycle. Similarly, on a down count when the count reaches to 0 it is reset to 255 on the next cycle.

4.9 PROJECT 8 – Knight Rider Car LEDs

4.9.1 Description

Knight Rider is a TV action movie where a super intelligent car that talk and self-navigate. A strip of lights are mounted in-front of the car. These lights turn ON one by one in one direction, and then in the reverse direction back and forth continuously. In this project 8 LEDs are connected to the ESP32 DevKitC and the LEDs simulate the Knight Rider car lights.

4.9.2 The Aim

The aim of this project is to show how the Knight Rider car lights can be simulated with the ESP32 processor.

4.9.3 Block diagram:

The block diagram of the project is as shown in Figure 4.23.

4.9.4 Circuit Diagram

The circuit diagram of the project is as shown in Figure 4.24. 8 LEDs are connected to the GPIO ports through 330 ohm current limiting resistors.

4.9.5 Construction

The project is built on a breadboard as shown in Figure 4.25.

4.9.6 PDL of the Project

The PDL of the project is shown in Figure 4.36.

```
BEGIN
    Store LED port numbers in array LEDs
    Configure LED port pins as outputs
    DO FOREVER
        Do k From 0 to 8
            Turn ON LED at index LEDs[k]
            Wait 100 ms
```

```

        Turn OFF LED at index LEDs[k]
ENDDO
DO k From 6 to 0
    TURN ON LED at index LEDs[k]
    Wait 100 ms
    Turn OFF LED at index LEDs[k]
ENDDO
ENDDO
END

```

Figure 4.36 PDL of the project

4.9.7 Program Listing

The program listing of the project is very simple and is shown in Figure 4.37 (program: **KnightRider**).

```

/*****
*           KNIGHT RIDER LEDs
*=====
*
* In this program 8 LEDs are connected to port pins GPIO23,
* GPIO22, GPIO1, GPIO3, GPIO21, GPIO19, GPIO18, and GPIO5
* of the ESP32 DevKitC. The program simulates the lights of
* the Knight Rider car as in the TV action movie Knight Rider.
*
*
* Program: KnightRider
* Date   : July, 2017
*****
int LEDs[] = {23, 22, 1, 3, 21, 19, 18, 5};
unsigned char Count = 0;
unsigned char del = 100;

//
// Set GPIO pins 23,22,1,3,21,19,18,5 as outputs
//
void setup()
{
    unsigned char i;
    for(i=0; i < 8; i++)
    {
        pinMode(LEDs[i], OUTPUT);
    }
}

```

```
//  
// Turn the LEDs ON/OFF to simulate the Knight Rider car  
//  
void loop()  
{  
    for(int k = 0; k < 8; k++)  
    {  
        digitalWrite(LEDs[k], HIGH);  
        delay(del);  
        digitalWrite(LEDs[k], LOW);  
    }  
  
    for(int k = 6; k > 0; k--)  
    {  
        digitalWrite(LEDs[k], HIGH);  
        delay(del);  
        digitalWrite(LEDs[k], LOW);  
    }  
}
```

Figure 4.37 Program listing

4.9.8 Program Description

At the beginning of the program an array called **LEDs** is set up to store the port numbers of the LEDs used in the project. Then, the GPIO ports that the LEDs are connected to are configured as output ports. Inside the main program two **for** loops are established. Inside the first **for** loop the LEDs from MSB to LSB are turned on for 100 ms. Inside the second loop the LEDs from LSB to MSB are turned ON for 100 ms. Thus, the net effect is that the LEDs chase each other in both directions.

4.9.10 Suggestions

In Figure 4.37, the delay time between each output is set to 100 ms. Try modifying this time and see the effects on the display.

4.10 PROJECT 9 – Changing the Brightness of an LED

4.10.1 Description

In this project an LED is connected to the ESP32 DevKitC. The program changes the brightness of the LED by changing the voltage applied to the LED.

4.10.2 The Aim

The aim of this project is to show how PWM type waveform can be generated in a program and how this waveform can be used to effectively vary the voltage applied to an LED.

4.10.3 Block diagram:

The block diagram of the project is as shown in Figure 4.1.

4.10.4 Circuit Diagram

The circuit diagram of the project is as shown in Figure 4.3. An LED is connected to the GPIO port 23 through 330 ohm current limiting resistor as in Project 1.

4.10.5 Construction

The project is built on a breadboard as shown in Figure 4.4.

4.10.6 PDL of the Project

The PDL of the project is shown in Figure 4.38.

```

BEGIN
  Define LED port
  Define PWM frequency, channel, and resolution
  Configure LED port pin as output
  Setup and attach port 23 to PWM channel 0
DO FOREVER
  DO 20 Times
    Increment the duty cycle by 5%, from 0% to 100%
    Send the waveform to port 23
    Wait 50 ms
  ENDDO

  DO 20 Times
    Decrement the duty cycle by 5%, from 100% to 0%
    Send the waveform to port 23
    Wait 50 ms
  ENDDO
ENDDO
END
```

Figure 4.38 PDL of the project

4.10.7 Program Listing

The program listing of the project is very simple and is shown in Figure 4.39 (program: **FadeLED**).

```

/*****+
 *          CHANGING THE LED BRIGHTNESS
 *          =====
 *
 * In this program an LED is connected to GPIO port pin 23.
 * The program changes the brightness of the LED by varying
 * the voltage applied to the LED. The LED PWM function is used
 * in this project where the duty cycle of the PWM waveform is
 * varied from 0 to its full value of 255
*****+
```

```
*  
* Program: Brightness  
* Date   : July, 2017  
*****  
int LED = 23;  
int frequency = 1000;  
int resolution = 8;  
int channel = 0;  
  
//  
// Set GPIO pin 23 as output. Setup the PWM channel 0 and attach  
// GPIO pin 23 to this channel  
//  
void setup()  
{  
    pinMode(LED, OUTPUT);  
    ledcSetup(channel, frequency, resolution);  
    ledcAttachPin(LED, channel);  
}  
  
//  
// Change the brightness of the LED by varying the duty cycle  
// from 0% (0) to 100% (255)  
//  
void loop()  
{  
    for(int duty = 0; duty < 255; duty +=5)  
    {  
        ledcWrite(channel, duty);  
        delay(50);  
    }  
  
    for(int duty = 255; duty >= 0; duty -=5)  
    {  
        ledcWrite(channel, duty);  
        delay(50);  
    }  
    delay(500);  
}
```

Figure 4.39 Program listing

4.10.8 Program Description

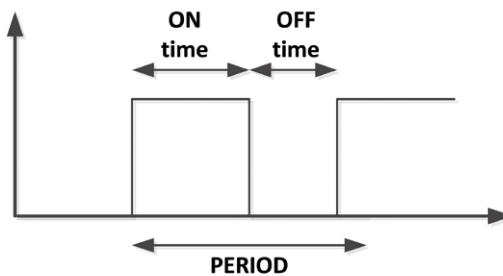
The fading of the LED is done by applying a PWM waveform to the LED and then changing the duty cycle of this waveform, which effectively changes the average voltage applied to the LED. Figure 4.40 shows a typical PWM waveform. The duty cycle is defined as the ratio

of the ON time to the period:

$$\text{Duty cycle} = \frac{\text{ONtime}}{\text{Period}} \times 100\%$$

or

$$\text{Duty cycle} = \frac{\text{ONtime}}{(\text{ONtime} + \text{OFFtime})} \times 100\%$$



$$\text{Duty Cycle} = (\text{ON time}) / (\text{Period}) \times 100\%$$

Figure 4.40 A typical PWM waveform

Thus, for example, when the duty cycle is 0% the ON time is 0, and when the duty cycle is 100% the OFF time is 0.

The program uses the LED PWM function of the ESP32 which consists of 16 independent channels with configurable duty cycles, frequencies, and resolutions. A channel from 0 to 15, and a resolution from 1 to 16 bits can be selected. The PWM setup function is called **ledcSetup** and it has the following format with integer parameters:

```
ledcSetup(channel, frequency, resolution)
```

After the setup, we have to attach the GPIO pin that we will be using as the PWM pin to the selected channel. This is done using function **ledcAttach** which has the format:

```
ledcAttachPin(GPIO pin, channel)
```

The PWM waveform is then sent to the selected GPIO pin using function **ledcWrite** which has the format:

```
ledcWrite(channel, duty cycle)
```

For an 8-bit resolution, the duty cycle is 0 for 0% and 255 for 100%.

In this project, the frequency is set to 1000 Hz, and channel 0 is used with the GPIO pin

23. The resolution is set to 8 bits. The duty cycle is changed from 0 to 100% in steps of 5% every 50 milliseconds. The effect is that the brightness of the LED changes from no light (0% duty cycle) to full brightness (100% duty cycle) where the brightness is increased by 5% every 50 ms. Similarly, after the full brightness is reached, the program waits for 500 ms and starts to fade the LED from full brightness to no light, again by 5% every 50 ms.

4.10.9 Suggestions

In Figure 4.39, the delay time is set to 50 ms and the duty cycle step is set to 5. Try modifying both of these values see their effects on the LED brightness.

4.11 PROJECT 10 – Generating Random Sounds Using a Buzzer

4.11.1 Description

In this project a passive buzzer is connected to GPIO port pin 23 of the ESP32 DevKitC. The buzzer generates sound with frequencies randomly changing between 100 Hz and 5 kHz.

4.11.2 The Aim

The aim of this project is to show how a passive buzzer can be connected to the ESP32 DevKitC and how sound with different frequencies can be generated using a buzzer.

4.11.3 Block diagram:

The block diagram of the project is as shown in Figure 4.41.

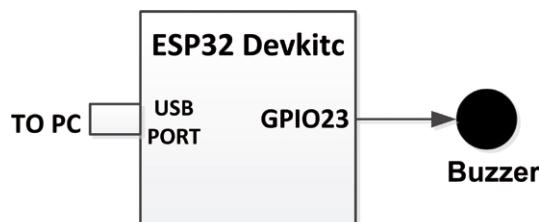


Figure 4.41 Block diagram of the project

4.11.4 Circuit Diagram

The circuit diagram of the project is as shown in Figure 4.42. The buzzer is connected to GPIO port 23.

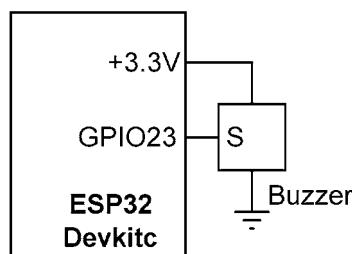


Figure 4.42 Circuit diagram of the project

4.11.5 Construction

The project is built on a breadboard as shown in Figure 4.43. Figure 4.44 shows the passive buzzer used in this project.

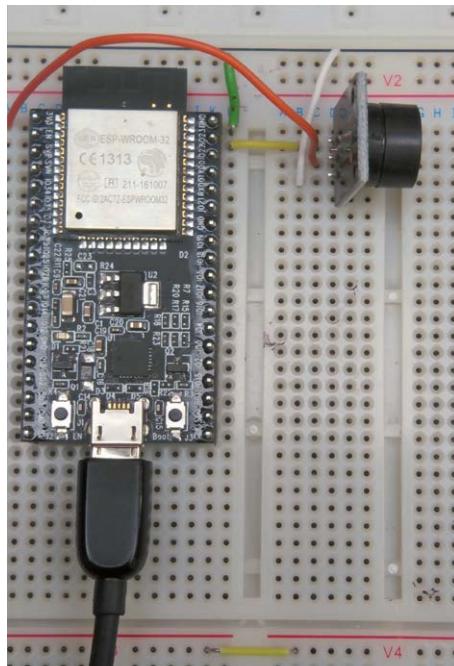


Figure 4.43 The project constructed on a breadboard

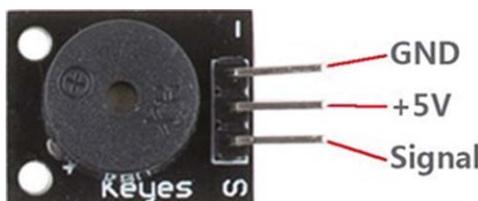


Figure 4.44 Passive buzzer used in the project

4.11.6 PDL of the Project

The PDL of the project is shown in Figure 4.45.

BEGIN

Define LED port

Define PWM initial frequency, channel, and resolution

Configure LED port pin as output

Setup and attach port 23 to PWM channel 0

DO FOREVER

```
Create a random number between 1000 and 5000  
Use this number as the frequency and send to the buzzer  
Wait 500 ms  
ENDDO  
END
```

Figure 4.45 PDL of the project

4.11.7 Program Listing

The program listing of the project is very simple and is shown in Figure 4.46 (program: **RandomBuzzer**).

```
/*********************************************  
* Random Buzzer  
* ======  
* In this program a passive Buzzer is connected to GPIO port  
* pin 23. The program generates random numbers between 1000  
* and 5000 and these numbers are used as the random frequency  
* to drive the Buzzer. The net effect is that the Buzzer  
* generates sound with random frequencies. The duty cycle of  
* the waveform is set to 50% (127). 500 ms delay is inserted  
* between each output.  
*  
*  
* Program: RandomBuzzer  
* Date : July, 2017  
******/  
int Buzzer = 23; // Buzzer on GPIO pin 23  
int duty = 127; // 50% duty cycle  
int freq = 1000; // Initial freq 1000 Hz  
int resolution = 8; // Resolution 8 bits  
int channel = 0; // Channel 0  
  
//  
// Set GPIO pin 23 as output. Setup the PWM channel 0 and attach  
// GPIO pin 23 to this channel  
//  
void setup()  
{  
    pinMode(Buzzer, OUTPUT);  
    ledcSetup(channel, freq, resolution);  
    ledcAttachPin(Buzzer, channel);  
    randomSeed(10);  
}
```

```

// 
// Use the random number generator to generate the frequency
// values between 1000 Hz and 5000 Hz. The duty cycle is set
// to 50% (127)
//
void loop()
{
    freq = random(1000, 5000);
    ledcWriteTone(channel, freq);
    delay(500);
}

```

Figure 4.46 Program listing

4.11.8 Program Description

There are two types of buzzers: passive buzzers, and active buzzers. Passive buzzers require an AC signal in the audible frequency range to operate. Active buzzers, on the other hand, have built in oscillators which generate a fixed frequency sound when logic 1 is applied to them, just like turning ON an LED. The frequency of an active buzzer can be changed by the ON/OFF frequency of the applied 1/0 logic signal. In this project a passive buzzer is used and a PWM signal with different frequencies is applied to the buzzer.

As in the previous project, the **ledcSetup** and **ledcAttach** functions are used to setup and attach the required GPIO pin to the PWM channel. Function **ledcWriteTone** is used to change the frequency of the PWM signal. The format of this function is:

```
ledcWriteTone(channel, required frequency)
```

In this project, channel 0 is used with the GPIO pin 23 and the resolution is set to 8 bits. The duty cycle is set to 50% (127). A random number is generated between 1000 and 5000 and this number is used as the frequency to activate the buzzer. 500 ms delay is used between each output.

4.11.9 Suggestion

In Figure 4.46 only the frequency is changed randomly. Modify the program to change the duty cycle as well and observe the difference.

4.12 PROJECT 11 – LED Colour Wand

4.12.1 Description

In this project an RGB LED is used to generate different colours of light, just like a colour wand.

4.12.2 The Aim

The aim of this project is to show how an RGB LED can be used in a program to generate different colours of light.

4.12.3 Block diagram:

The block diagram of the project is as shown in Figure 4.47.

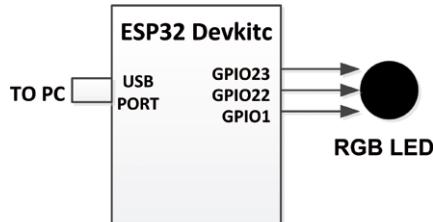


Figure 4.47 Block diagram of the project

4.12.4 Circuit Diagram

The circuit diagram of the project is as shown in Figure 4.48. The R, G, and B LED pins are connected to GPIO pins 23, 22 and 1 of the ESP32 DevKitC respectively through 330 ohm current limiting resistors.

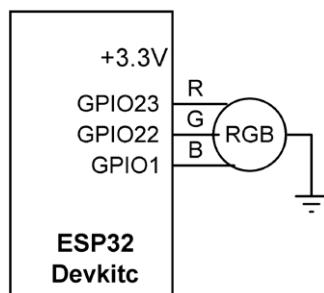


Figure 4.48 Circuit diagram of the project

4.12.5 Construction

The project is built on a breadboard as shown in Figure 4.49. It can be built into a wand and used in games, in a flower vase, or in an aquarium. Figure 4.50 shows a typical RGB LED. Notice that the component has 4 legs where the longest leg is the common pin (cathode or anode).

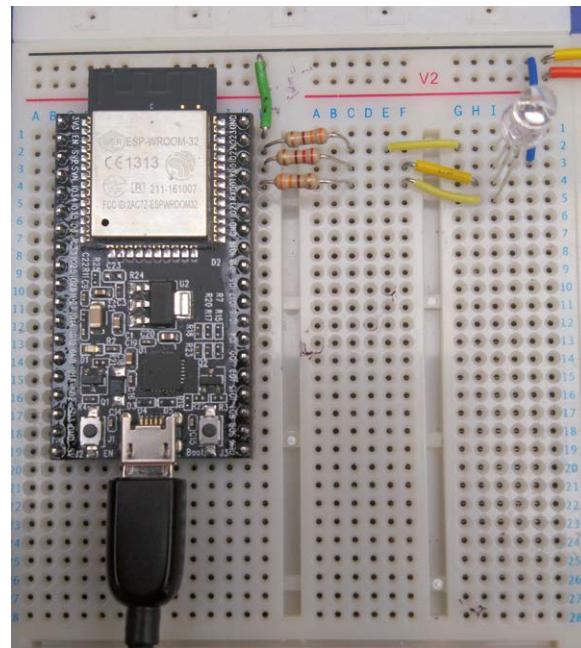


Figure 4.49 The project constructed on a breadboard

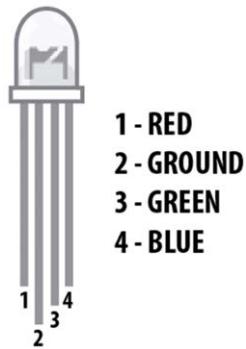


Figure 4.50 A typical RGB LED (common cathode)

4.12.6 PDL of the Project

The PDL of the project is shown in Figure 4.51.

```
BEGIN
  Define port numbers for Red, Green and Blue pins
  Configure ports as outputs
DO FOREVER
  Create a random number between 1 and 1 for Red
  Create a random number between 1 and 1 for Green
  Create a random number between 1 and 1 for Blue
  Write Red value to Red port
```

```
    Write Green value to Green port
    Write Blue value to Blue port
    Wait 500 ms
ENDDO
END
```

Figure 4.51 PDL of the project

4.12.7 Program Listing

The program listing of the project is very simple and is shown in Figure 4.52 (program: **RGB**).

```
/*********************************************
*           RGB LED
* =====
*
* In this program an RGB LED is connected to the ESP32 DevKitC
* where Red pin to GPIO23, Green pin to GPIO22, and the Blue pin
* to GPIO1. The LEDs are turned ON and OFF randomly.
*
*
* Program: RGB
* Date   : July, 2017
******/
#define RED 23
#define GREEN 22
#define BLUE 1
int R, G, B;

//
// Set GPIO pin 23, 22 and 1 as outputs
//
void setup()
{
    pinMode(RED, OUTPUT);
    pinMode(GREEN, OUTPUT);
    pinMode(BLUE, OUTPUT);
    randomSeed(10);
}

//
// Use the random number generator to generate ON (1) or OFF
// (0) values for the three colours
//
void loop()
```

```

{
    R = random(0, 2);
    G = random(0, 2);
    B = random(0, 2);

    digitalWrite(RED, R);
    digitalWrite(GREEN, G);
    digitalWrite(BLUE, B);
    delay(500);
}

```

Figure 4.52 Program listing

4.12.8 Program Description

At the beginning of the program the GPIO pins that the Red, Green and Blue pins are connected to are defined. Then these port pins are configured as outputs. The remainder of the program runs in an endless loop in which random numbers are generated between 0 and 1 (notice that when using the **random** function, the lower bound is included, but the upper bound is excluded. Because of this the arguments to this function must be 0, 2) for all three colours and the generated numbers are sent to the corresponding ports. Thus, for example, if number 1 is generated for the Red port then the red colour is turned ON, and so on.

4.12.9 Suggestions

In the program in Figure 4.52 the delay time is set to 500 ms. Try cg=hanging this time and see its effects.

4.13 PROJECT 12 – Using the Built-in Hall Sensor – Door Alarm

4.13.1 Description

This is a door lock alarm project where a magnet, an active buzzer, and a DevKitC development board are used. When the door is closed the magnet is very close to the DevKitC development board. The development board reads the magnetic field using its on-board Hall sensor and then turns OFF the buzzer. As soon as the door opens the magnet moves away from the development board and the Hall sensor returns a different value. The processor reads this value and turns ON the buzzer to indicate that the door is opened.

4.13.2 The Aim

The aim of this project is to show how the Hall sensor output can be read.

4.13.3 Block diagram:

The block diagram of the project is as shown in Figure 4.53.

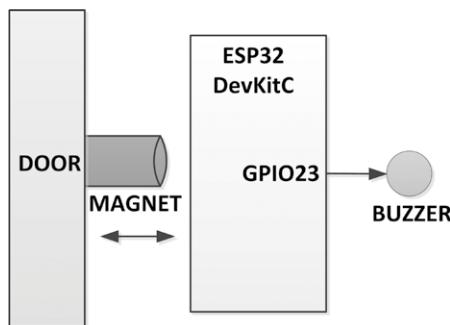


Figure 4.53 Block diagram of the project

4.13.4 Circuit Diagram

The circuit diagram of the project is as shown in Figure 4.54. The active buzzer is connected to GPIO pin 23 of the DevKitC development board.

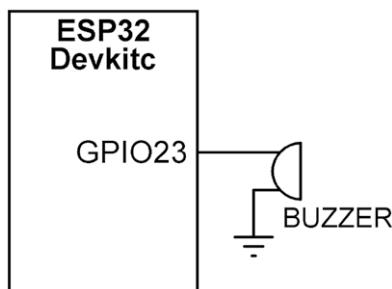


Figure 4.54 Circuit diagram of the project

4.13.5 PDL of the Project

The PDL of the project is very simple and is shown in Figure 4.55.

```

BEGIN
    Configure Buzzer as output
    Turn OFF the Buzzer
    DO FOREVER
        Read Hall sensor output
        IF returned value > 10 THEN
            Turn Buzzer ON
        ELSE
            Turn Buzzer OFF
        ENDIF
    ENDDO
END
  
```

Figure 4.55 PDL of the project

4.13.6 Program Listing

The value returned from the Hall sensor depends on the type of magnet used and the strength of the magnet. It is probably a good idea to write a short program code to display the value returned from the Hall sensor when the magnet to be used is close and also away from the development board. The following program code displays the values returned from the Hall sensor on the PC screen (using the **Putty** terminal emulator) when a small magnet is used:

```
void setup()
{
    Serial.begin(9600);
}

void loop()
{
    int magnet;
    magnet = hallRead();
    Serial.print("Hall sensor output = ");
    Serial.println(magnet);
    delay(1000);
}
```

Figure 4.56 shows the values returned from the Hall sensor. Without any magnet the returned values are between 18 and 21. When a small magnet is placed near the development board the readings fall to around 3. We can now write our main program based on this simple experiment.

```
COM47 - PuTTY
Hall sensor output = 21
Hall sensor output = 18
Hall sensor output = 19
Hall sensor output = 19
Hall sensor output = 18
Hall sensor output = 14
Hall sensor output = 18
Hall sensor output = 18
Hall sensor output = 21
Hall sensor output = 19
Hall sensor output = 18
Hall sensor output = 19
Hall sensor output = 21
Hall sensor output = 8
Hall sensor output = 7
Hall sensor output = 3
Hall sensor output = 4
Hall sensor output = 3
Hall sensor output = 3
```

Figure 4.56 Values returned from the Hall sensor

The program listing of the project is very simple and is shown in Figure 4.57 (program: **Hall**). At the beginning of the program, a buzzer is assigned to GPIO port 23. Inside the main program loop the program reads output of the Hall sensor using function **hallRead**. If the returned value is greater than 10 then it is assumed that the door is opened (magnet is away from the Hall sensor) and the buzzer is turned ON, otherwise the buzzer is turned OFF.

```
/*****
 *          HALL SENSOR DOOR ALARM
 *          =====
 *
 * In this program a Buzzer is connected to the ESP32 DevKitC GPIO
 * port 23. The built-in Hall sensor is used in this project. The
 * magnet is near the development board when the door is closed
 * and therefore the values returned by the Hall sensor are around
 * 3. When the door opens the magnet gets away from the door and
 * the Hall sensor returns values greater than 10, which is detected
 * by the program and a buzzer is activated to indicate that the
 * door is opened.
 *
 * Program: Hall
 * Date   : October, 2018
 *****/
#define Buzzer 23
#define ON HIGH
#define OFF LOW

void setup()
{
    pinMode(Buzzer, OUTPUT);           // Buzzer is output
    digitalWrite(Buzzer, OFF);         // Buzzer OFF initially
}

void loop()
{
    int magnet;

    while(1)                         // Do Forever
    {
        magnet = hallRead();          // Read Hall sensor
        if(magnet > 10)              // If door opened
            digitalWrite(Buzzer, ON);  // Buzzer ON
        else                          // Else...
            digitalWrite(Buzzer, OFF); // Buzzer OFF
    }
}
```

Figure 4.57 Program listing

4.14 PROJECT 13 – Using the Built-in Temperature Sensor

4.14.1 Description

The built-in temperature sensor of the ESP32 processor is not meant for measuring the absolute ambient temperature. This sensor measures the chip temperature and can be used to measure delta temperatures, i.e. changes in the temperature. One application of this temperature sensor could be to make sure that the processor temperature is stable. In this project, the chip temperature is measured and displayed on the PC screen using the Putty terminal emulator program. The measured value is returned in Degrees Fahrenheit. In this project this is converted into Degrees Centigrade before displayed.

4.14.2 The Aim

The aim of this project is to show how the built-in temperature sensor can be used in a project.

4.14.3 Program Listing

The program listing (program: **OnBoardTemp**) is shown in Figure 4.58.

```
*****  
*          ON-CHIP TEMPERATURE SENSOR  
*          ======  
  
* This program uses the on-chip temperature sensor of the DevKitC  
* development board to display the temperature. But note that the  
* absolute temperature is not measured. The measured value can be  
* used as delta temperature. i.e. to compare the change in the chip  
* temperature. The returned value is in Fahrenheit and can easily  
* be converted into Centigrade.  
  
* Program: OnChipTemp  
* Date   : October, 2018  
*****  
#ifdef __cplusplus  
extern "C"  
{  
    #endif  
    uint8_t temprature_sens_read();  
    #ifdef __cplusplus  
}  
#endif  
    uint8_t temprature_sens_read();  
  
void setup()  
{  
    Serial.begin(9600);  
}
```

```
void loop()
{
    while(1)
    {
        Serial.print("temperature = ");
        Serial.println((temprature_sens_read() - 32.0) / 1.8);
        delay(1000);
    }
}
```

Figure 4.58 Program listing

4.14.4 Program Description

Function **temperature_sens_read** returns the chip temperature in Degrees Fahrenheit. This is converted into Degrees Centigrade by subtracting 32 and dividing by 1.8. The resulting temperature is displayed on the PC screen every second. Notice that the conditional code at the beginning of the program ensures that the C library where the function is can be called from our C++ code.

Figure 4.59 shows a typical output from the program.

Figure 4.59 Output from the program

4.15 PROJECT 14 – Chip Identity

4.15.1 Description

The chip ID is basically same as the 6 byte chip MAC address. This project reads the ESP32 chip ID and displays on the PC screen.

4.15.2 The Aim

The aim of this project is to show how the chip ID can be read from the chip.

4.15.3 Program Listing

The program listing (program: **ChipId**) is shown in Figure 4.60. Function **ESP.getEfuseMac()** returns the chip ID which is then displayed on the PC screen. Notice that because the returned data is too long, it is divided into two parts: 2 upper bytes are stored in variable **MSD** and the 4 lower bytes are stored in variable **LSD**. The chip ID is displayed both in decimal and in hexadecimal formats as shown in Figure 4.61.

```
/*
 *          READ THE CHIP ID
 *          =====
 *
 * This program reads and then displays the chip id on the PC screen
 *
 * File:    ChipId
 * Author:  Dogan Ibrahim
 * Date:   November, 2018
 */
unsigned long long ChipID;
unsigned short MSD;           // Unsigned 16 bits
unsigned long LSD;            // Unsigned 32 bits

void setup()
{
    Serial.begin(9600);
}

void loop()
{
    ChipID = ESP.getEfuseMac();
    MSD = ChipID >> 32;
    LSD = ChipID & 0xFFFFFFFF;
    Serial.printf("\n\rChip Id (Dec)= %d%d\n\r", MSD,LSD);
    Serial.printf("Chip Id (Hex)= %04X%08X\n\r", MSD,LSD);

    while(1);
}
```

Figure 4.60 Program listing

```
Chip Id (Dec)= 5743594678576
Chip Id (Hex)= E05B05A4AE30
```

Figure 4.61 Displaying the chip id

4.16 Summary

In this chapter we have seen the development of some very basic projects using the ESP32 DevKitC. In the next chapter we shall be looking at the development of slightly more complex projects.

Chapter 5 • Simple projects using the Arduino IDE and the ESP32 DevKitC

5.1 Overview

In the last chapter we have seen the development of some basic projects using the ESP32 DevKitC. In this chapter we shall be developing slightly more complex projects using the Arduino IDE as the development software.

As with the previous projects, the title, description, aim, block diagram and so on of all the projects will be given.

5.2 PROJECT 1 – Thermometer with Serial Monitor

5.2.1 Description

In this project an analog temperature sensor chip is used to get the ambient temperature. The temperature is displayed every second on the PC screen using the Arduino IDE Serial Monitor.

5.2.2 The Aim

The aim of this project is to show how an analog input of the ESP32 DevKitC can be used to read analog data. In addition, it is shown how to display data on the PC screen.

5.2.3 Block diagram:

Figure 5.1 shows the block diagram of the project.

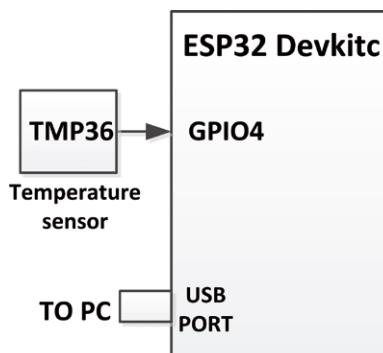


Figure 5.1 Block diagram of the project

5.2.4 Circuit Diagram

In this project the TMP36 analog temperature chip is used to get the ambient temperature. TMP36 temperature sensor chip is a 3-pin device having the pin layout as shown in Figure 5.2. The device operates with a power supply in the range 2.7 V to 5.5 V and can measure the temperature in the range -40 °C to +125 °C. The output voltage of the device is proportional to the measured temperature and is given by:

$$C = (V - 500) / 10$$

Where C is the measured temperature in $^{\circ}\text{C}$, V is the output voltage of the sensor chip in mV. Thus, for example, an output voltage of 800 mV corresponds to 30 $^{\circ}\text{C}$ and so on.

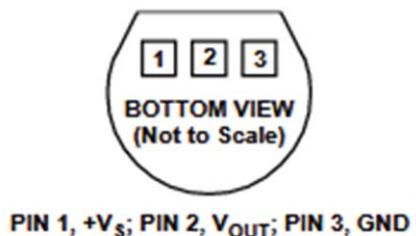


Figure 5.2 Pin layout of TMP36 temperature sensor chip

Figure 5.3 shows the circuit diagram of the project. Two pins of the TMP36 are connected to the +3.3 V power supply and the ground. The output pin of the TMP36 temperature sensor chip is connected to GPIO4 which is also the ADC2_0 of the ESP32 DevKitC (see Figure 4.2). The ADC is 12-bits wide and has a reference voltage of +3.3 V.

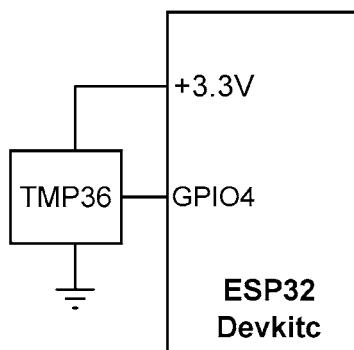


Figure 5.3 Circuit diagram of the project

5.2.5 Construction

The ESP32 DevKitC board is mounted on a breadboard as shown in Figure 5.4. The output of the TMP36 is connected to GPIO pin 4.

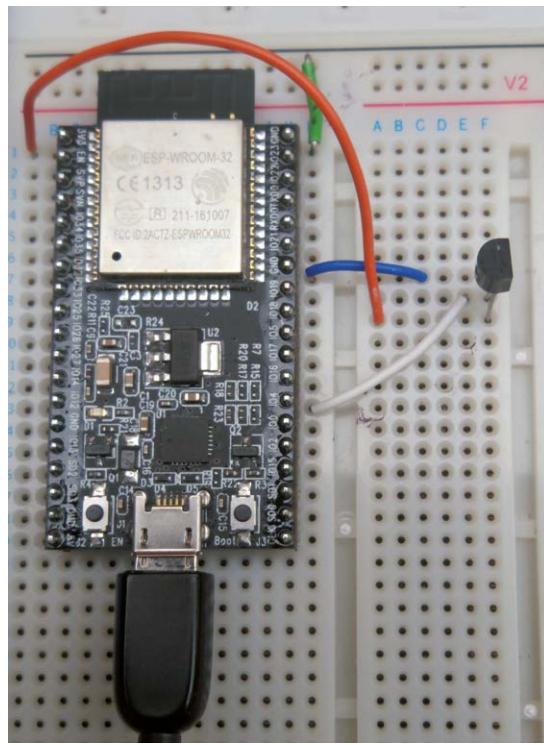


Figure 5.4 Project on a breadboard

5.2.6 PDL of the Project

The PDL of the project is shown in Figure 5.5. The program runs in an endless loop in which the temperature is read every second, and converted into degrees Celsius so that it can be displayed via the Serial Monitor.

```
BEGIN
    Define TMP36 as GPIO pin 4
    Set the Serial Monitor baud rate to 9600
    DO FOREVER
        Read the ambient temperature
        Convert the temperature into Degrees Centigrade
        Send the readings to Serial Monitor
        Wait 1 second
    ENDO
END
```

Figure 5.5 PDL of the project

5.2.7 Program Listing

The program listing of the project is shown in Figure 5.6 (program: **TMP36**).

```
/*
 *          THERMOMETER WITH SERIAL MONITOR
 *
 * =====
 *
 * In this program a TMP36 type analog temperature sensor chip
 * is connected to GPIO pin 4 which is also an analog input.
 * The program reads the ambient temperature every second and
 * sends the readings to the Serial Monitor
 *
 *
 * Program: TMP36
 * Date   : July, 2017
 */
#define TMP36 4

void setup()
{
    Serial.begin(9600);
}

// 
// Read the ambient temperaure, convert into Degrees Centigrade
// and send the readings to the Serial Monitor. Repeat this
// process every second
//
void loop()
{
    int Temp = analogRead(TMP36);
    float mV = Temp * 3300.0 / 4096.0;
    float Temperature = (mV - 500.0) / 10.0;
    Serial.println(Temperature);
    delay(1000);
}
```

Figure 5.6 Program listing

5.2.8 Program Description

At the beginning of the program **TMP36** is defined as port GPIO4 which is one of the analog input ports of the ESP32 DevKitC, and the Serial Monitor baud rate is set to 9600. The remainder of the program is executed in an endless loop. Inside this loop, the analog output of the TMP36 is read using the **analogRead** library function. The reading is then converted into absolute millivolts. Notice that the ADC reference voltage is 3300 mV and the ADC is 12-bits wide (0 to 4095). The reading is then converted into Degrees Celsius by subtracting

500 and dividing by 10. The temperature values are then sent to the Serial Monitor so that they can be displayed on the PC screen. To do this, click **Tool -> Serial Monitor** and select the baud rate as 9600. A typical output from the Serial Monitor is shown in Figure 5.7. We can make the display more user-friendly by displaying the text **Temperature** before its value by modifying the code as follows. The resultant display is shown in Figure 5.8.

```
float Temperature = (mV - 500.0) / 10.0;  
int len = Serial.write("Temperature = ");  
Serial.println(Temperature);
```

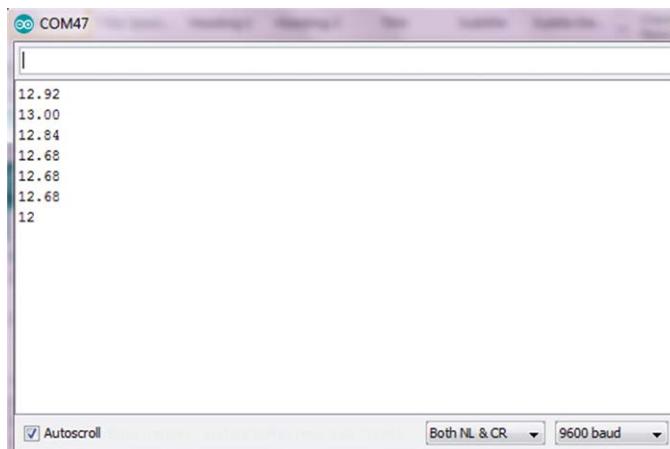


Figure 5.7 Typical output from the Serial Monitor

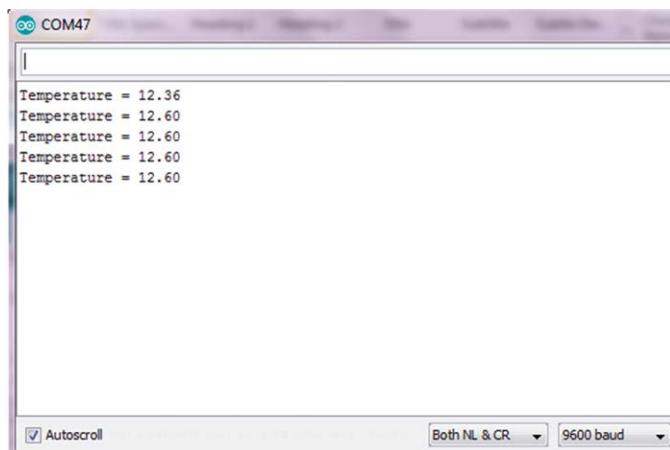


Figure 5.8 Modified output

It is important to mention at this point that the ESP32 DevKitC used by the author (any many others as reported on the Internet) has ADC linearity problems which may result in inaccurate readings. In order to get accurate results from the ADC it is recommended to

carry out an experiment and create a table or an equation showing the actual relationship between the analog input voltages and the digital values returned by the ADC. This table or the equation can then be used to correct the ADC readings.

5.3 PROJECT 2 – Temperature and Relative Humidity with Serial Monitor

5.3.1 Description

In this project the DHT11 temperature and relative humidity sensor chip is used to get the ambient temperature and the relative humidity. The readings are displayed every second on the PC screen using the Arduino Serial Monitor.

5.3.2 The Aim

The aim of this project is to show how the popular DHT11 relative humidity and temperature sensor chip can be used in projects.

5.3.3 Block diagram:

Figure 5.9 shows the block diagram of the project.

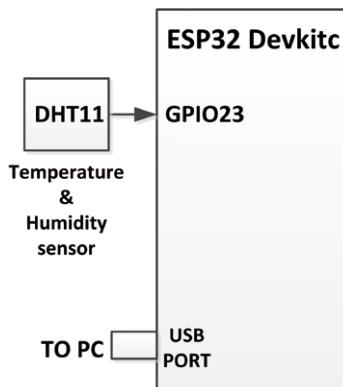


Figure 5.9 Block diagram of the project

5.3.4 Circuit Diagram

In this project the DHT11 relative humidity and temperature sensor chip is used. The standard DHT11 is a 4-pin digital output device (only 3 pins are used) as shown in Figure 5.10, having pins +V, GND, and Data. The Data pin must be pulled-up to +V through a 10K resistor. The chip uses capacitive humidity sensor and a thermistor to measure the ambient temperature. Data output is available from the chip around every second. The basic features of DHT11 are:

- 3 to 5 V operation
- 2.5mA current consumption (during a conversion)
- Temperature reading in the range 0-50 °C with an accuracy of ±2 °C
- Humidity reading in the range 20-80% with 5% accuracy
- Breadboard compatible with 0.1 inch pin spacings

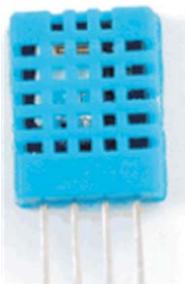


Figure 5.10 The standard DHT11 chip

In this example the DHT11 module with built-in 10K pull-up resistor, available from Elektor, is used. This is a 3-pin device with the pin layout as shown in Figure 5.11.

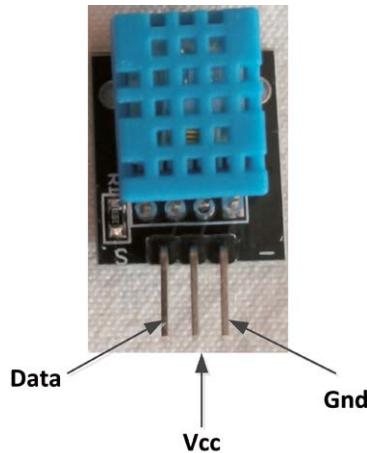


Figure 5.11 Elektor DHT11 module

Figure 5.12 shows the circuit diagram of the project. Here, the Data output of the DHT11 is connected to pin GPIO23 of the ESP32 DevKitC.

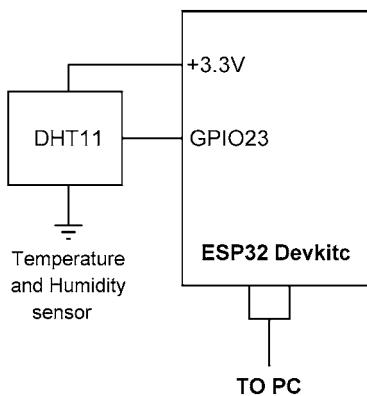


Figure 5.12 Circuit diagram of the project

5.3.5 Construction

The ESP32 DevKitC board is mounted on a breadboard as shown in Figure 5.13 with the DHT11 sensor chip connected to +3.3 V, GND and GPIO23 of the development board.

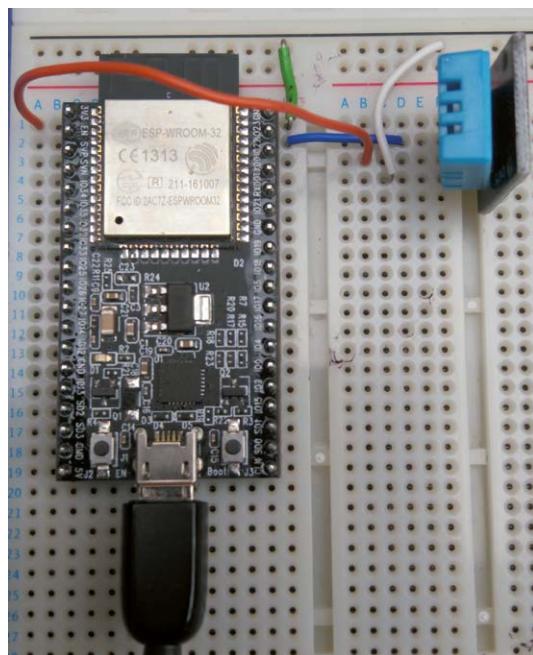


Figure 5.13 Project on a breadboard

5.3.6 PDL of the Project

The PDL of the project is shown in Figure 5.14. The program runs in an endless loop where the temperature and relative humidity are read every second and the readings sent to the Serial Monitor.

```

BEGIN
    Include the necessary header files
    Define GPIO23 as the DHT11 port
    Define the DHT type as DHT11
    Set the Serial Monitor baud rate to 9600
    Initialize the DHT library
DO FOREVER
    Read the relative humidity
    Read the ambient temperature
    Display the relative humidity and temperature on Serial Monitor
    Wait 1 second
ENDO
END

```

Figure 5.14 PDL of the project

5.3.7 Program Listing

It is necessary to install the DHT sensor library before the sensor chip can be used in an ESP32 processor based project. The steps to install this library into the Arduino IDE environment are given below:

- Install the **DHT-sensor-library**. Click **Sketch -> Include Library -> Manage Libraries**.
- Search for library **DHT-sensor-library** as shown in Figure 5.15 and click to install the library.

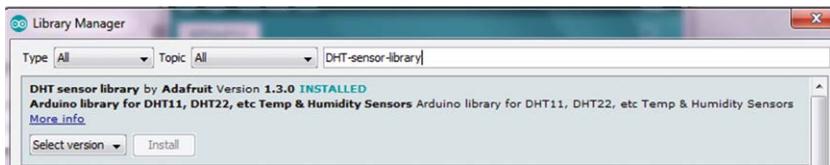


Figure 5.15 Install the DHT-sensor-library to Arduino library

- We now have to install the **Adafruit Unified Sensor** library. Click **Sketch -> Include Library -> Manage Libraries**.
- Search for library **Adafruit Unified Sensor** as shown in Figure 5.16 and click to install the library.

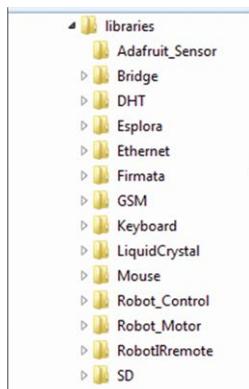


Figure 5.16 Install the Adafruit Unified Sensor library

- Make sure that you have the following two folders under your **Libraries** folder inside the Arduino main folder: **DHT_Sensor_Library** and **Adafruit_Unified_Sensor**.
- Re-start the Arduino IDE.
- We are now ready to develop our program to read the relative humidity and the temperature. Figure 5.17 shows the program listing (program: **DHT11**).

```
/*
 *      RELATIVE HUMIDITY AND TEMPERATURE WITH SERIAL MONITOR
 *      =====
 *
 * In this program a DHT11 type relative humidity and temperature
 * sensor chip is connected to GPIO pin 23. The relative humidity
 * and the ambient temperature are read every second and the
 * readings are displayed on the Serial monitor.
 *
 * Program: DHT11
 * Date   : July, 2017
 */
#include <Adafruit_Sensor.h>
#include "DHT.h"
#define DHT11_PIN 23
#define DHTTYPE DHT11

DHT dht(DHT11_PIN, DHTTYPE);

//
// Set the Serial Monitor baud rate to 9600 and initisize the
// DHT library
//
void setup()
{
    Serial.begin(9600);
    dht.begin();
}

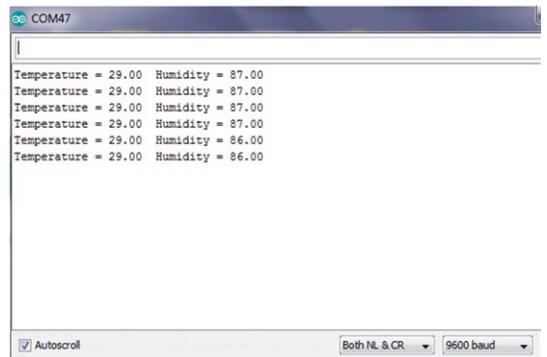
//
// Read the relative humidity and the ambient tempeaura and
// display the results on the Serial Monitor
//
void loop()
{
    float humidity = dht.readHumidity();           // Read humidity
    float temperature = dht.readTemperature();       // Read temperature
    Serial.print("Temperature = ");                // Display "Temperature="
    Serial.print(temperature);                     // Display temperature
    Serial.print(" Humidity = ");                  // Display "Humidity="
    Serial.println(humidity);                      // Display humidity
    delay(1000);                                 // Wait 1 second
}
```

Figure 5.17 Program listing

5.3.8 Program Description

At the beginning of the program header files **Adafruit_Sensor.h** and **DHT.h** are included in the program. GPIO pin that the DHT11 is connected to is defined as pin 23, and the DHT type is specified as DHT11. Then, the Serial Monitor baud rate is set to 9600 and the DHT library is initialised.

The remainder of the program is executed in an endless loop. Inside this loop the relative humidity and the ambient temperature are read and displayed on the Serial Monitor every second. Figure 5.18 shows a sample output displayed on the Serial Monitor.



A screenshot of a Windows-style Serial Monitor window titled "COM47". The window displays a series of text entries, each consisting of "Temperature = 29.00" followed by "Humidity = 87.00". Below the text area, there are three buttons: "Autoscroll" (unchecked), "Both NL & CR" (selected), and "9600 baud" (selected).

Figure 5.18 Sample output

5.4 PROJECT 3 – Measuring the Light Level

5.4.1 Description

In this project a light dependent resistor (LDR) is used to measure the light level every second. The reading is then displayed on the Serial Monitor.

5.4.2 The Aim

The aim of this project is to show how an LDR can be used in ESP32 processor based projects.

5.4.3 Block diagram:

Figure 5.19 shows the block diagram of the project.

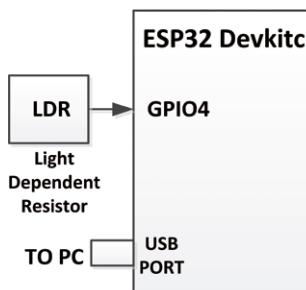


Figure 5.19 Block diagram of the project

5.4.4 Circuit Diagram

LDRs are simple resistors (Figure 5.20) whose resistances decrease with increasing incident light. These devices are usually used in light level control applications. Figure 5.21 shows the characteristic curve of a typical LDR. As you can see, the resistance of the device decreases as the light intensity increases.

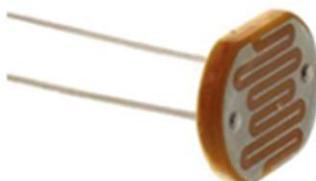


Figure 5.20 A typical light dependent resistor

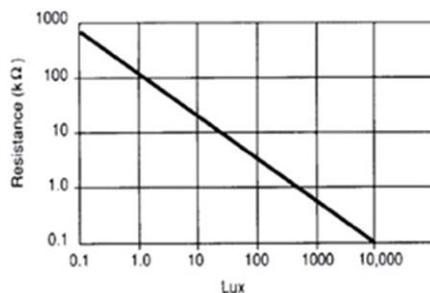


Figure 5.21 Characteristic curve of a typical LDR

The circuit diagram of the project is shown in Figure 5.22. One leg of the LDR is connected to ground (0 V). The other leg is connected to +3.3 V (3 V) through a 10K resistor. The junction of the LDR with the 10K resistor is connected to analog input GPIO4 of the ESP32 DevKitC. Notice that, just like resistors, LDRs have no polarities.

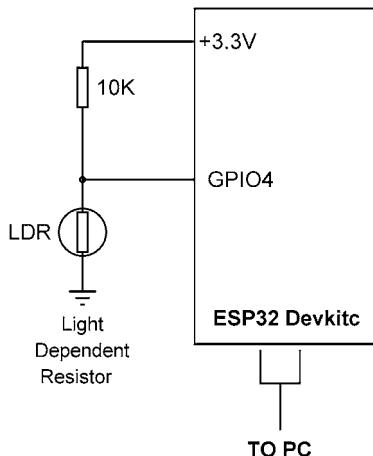


Figure 5.22 Circuit diagram of the project

5.4.5 Construction

The ESP32 DevKitC board is mounted on a breadboard as shown in Figure 5.23 with the LDR and a 10K resistor.

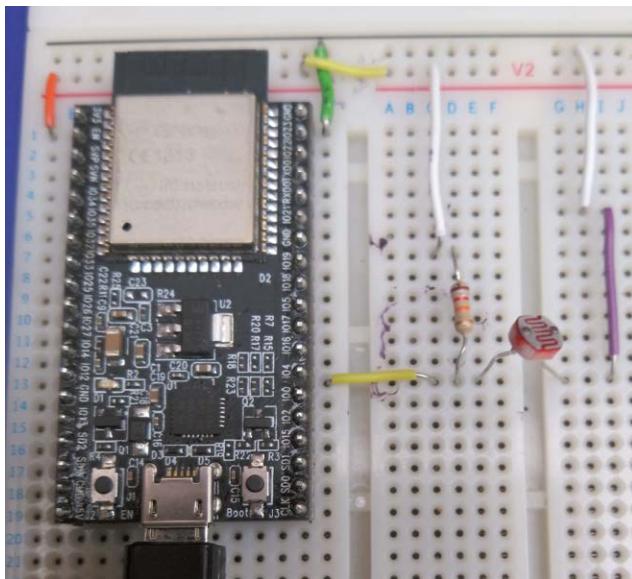


Figure 5.23 Project on a breadboard

5.4.6 PDL of the Project

The PDL of the project is shown in Figure 5.24. The program runs in an endless loop where the ambient light level is measured and displayed on the Serial Monitor.

```
BEGIN
    Define LDR as GPIO4
    Set the Serial Monitor baud rate to 9600
    DO FOREVER
        Read the voltage across the LDR through the ADC
        Display the reading on the Serial Monitor
        Wait 1 second
    ENDO
END
```

Figure 5.24 PDL of the project

5.4.7 Program Listing

The program is very simple and its listing is shown in Figure 5.25 (program: **LDR**).

```
/*
 *      MEASURING THE LIGHT LEVEL
 *      =====
 *
 * In this program a light dependent resistor is connected
 * to analog input GPIO4 of the ESP32 Devkitc. The program
 * measures and then displays the light level through the
 * Serial Monitor. The resistance of the LDR increases in
 * dark and as a result a higher reading is obtained.
 *
 *
 * Program: LDR
 * Date   : July, 2017
 */
#define LDR 4
//
// Set the Serial Monitor baud rate to 9600
//
void setup()
{
    Serial.begin(9600);
}

//
// Read the light level and display through the Serial Monitor
//
void loop()
{
    int ldr = analogRead(LDR);
    Serial.println(ldr);
    delay(1000);
}
```

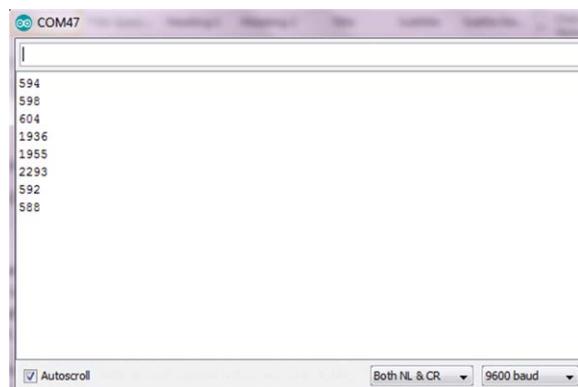
Figure 5.25 Program listing

5.4.8 Program Description

At the beginning of the program **LDR** is assigned to analog input port GPIO4. Inside the main program the voltage across the LDR is read through the ADC and is displayed on the Serial Monitor. Notice that what is displayed is the binary value read by the ADC and not the actual physical voltage across the LDR.

In normal light conditions the readings are around a few hundred. In the dark the resistance of the LDR increases and the reading goes over 2000. The program should be calibrated using a commercial light meter so that correct light level readings are correct.

Figure 5.26 shows a typical output from the program.

A screenshot of a Windows-style Serial Monitor window titled "COM47". The window has a menu bar with "File", "Edit", "Messages", "Help", and "About". Below the menu is a text area containing the following data:

```
594
598
604
1936
1955
2293
592
588
```

The bottom of the window shows settings: "Autoscroll" checked, "Both NL & CR" selected, and "9600 baud" selected.

Figure 5.26 A typical output on Serial Monitor

5.4.9 Suggestions

A variable resistor can be used instead of the 10K fixed resistor. This way, the circuit can be made more sensitive to other light levels.

5.5 PROJECT 4 – Darkness Reminder

5.5.1 Description

In this project a light dependent resistor (LDR) is used as in the previous project. Here, an active buzzer and an LED are connected to GPIO pins 23 and 22 respectively. The project reminds darkness where both the buzzer and the LED are turned ON when the LDR senses darkness.

5.5.2 The Aim

The aim of this project is to show how a simple darkness reminder project can be designed using the ESP32 DevKitC.

5.5.3 Block diagram:

Figure 5.27 shows the block diagram of the project.

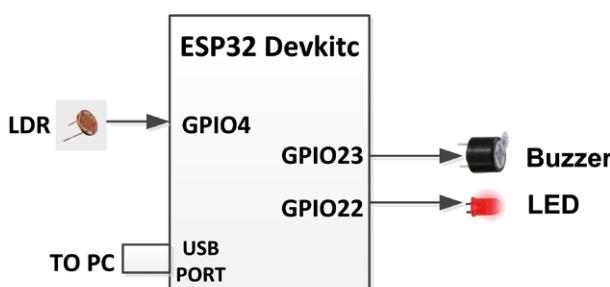


Figure 5.27 Block diagram of the project

5.5.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 5.28.

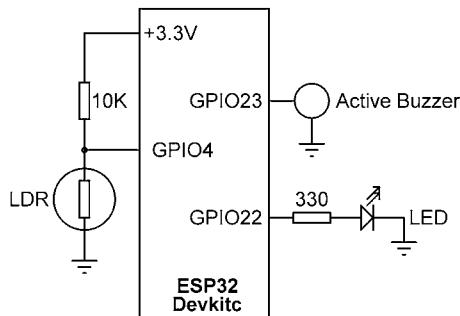


Figure 5.28 Circuit diagram of the project

5.5.5 Construction

The ESP32 DevKitC board is mounted on a breadboard as shown in Figure 5.29 with the LDR, a 10K resistor, and an active buzzer.

Make sure that the active buzzer you are using does not require more than about 5 mA and it can operate with +3.3 V. A transistor switch may be required to protect the output port loading if the buzzer requires more than 5 mA.

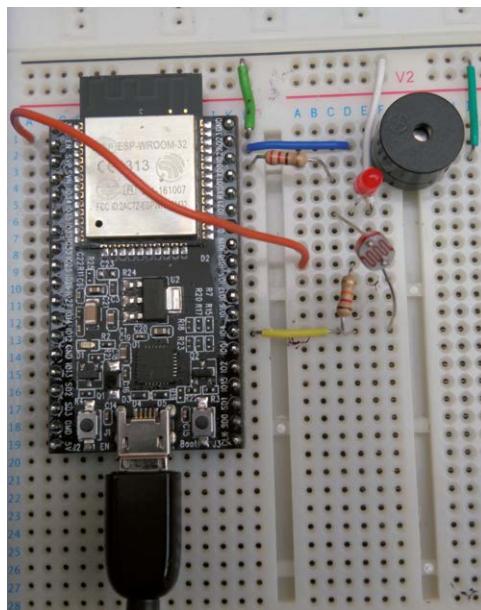


Figure 5.29 Project on a breadboard

5.5.6 PDL of the Project

The PDL of the project is shown in Figure 5.30. The program runs in an endless loop sensing the ambient light level, and when it is dark both the buzzer and the LED are turned ON. Darkness is assumed when the reading of the LDR is above 2000.

```
BEGIN
    Define LDR as GPIO4
    Define BUZZER as GPIO23
    Define LED as GPIO22
    Configure GPIO22 and GPIO23 as outputs
    Turn OFF both BUZZER and LED to start with
    Set the Serial Monitor baud rate to 9600
DO FOREVER
    Read the voltage across the LDR through the ADC
    IF reading > 2000 THEN
        Turn ON BUZZER
        Turn ON LED
    ELSE
        Turn OFF BUZZER
        Turn OFF LED
    ENDIF
ENDO
END
```

Figure 5.30 PDL of the project

5.5.7 Program Listing

The program is very simple and its listing is shown in Figure 5.31 (program: **LDRALARM**).

```
*****
*          DARKNESS REMINDER
*          =====
*
* In this program a light dependent resistor is connected
* to analog input GPIO4 of the ESP32 Devkitc. In addition,
* a buzzer and an LED are connecte dto GPIO ports 23 and 22
* respectively. Normally both the buzzer and the LED are OFF
* and they turn ON when it becomes dark. Darkness is assumed
* when the LDR reading is above 2000.
*
*
* Program: LDRALARM
* Date   : July, 2017
*****
#define LDR 4
#define BUZZER 23
```

```
#define LED 22

//
// Configure both the BUZZER and the LED as outputs
//
void setup()
{
    pinMode(BUZZER, OUTPUT);
    pinMode(LED, OUTPUT);
    digitalWrite(BUZZER, LOW);
    digitalWrite(LED, LOW);
}

//
// Read the light level and turn on both the BUZZER and the LED
// when it becomes dark. In this project the darkness has been
// assumed when the LDR reading is above 2000
//
void loop()
{
    int ldr = analogRead(LDR);
    if(ldr > 2000)
    {
        digitalWrite(BUZZER, HIGH);
        digitalWrite(LED, HIGH);
    }
    else
    {
        digitalWrite(BUZZER, LOW);
        digitalWrite(LED, LOW);
    }
}
```

Figure 5.31 Program listing

5.5.8 Program Description

At the beginning of the program **LDR** is defined as port 4, BUZZER as port 23 and LED as port 22. Digital ports 23 and 22 are configured as outputs. Inside the main program the output of the LDR is read and stored in integer variable **ldr**. Darkness is assumed if **ldr** is greater than 2000 (see the previous project) and when this happens both the buzzer and the LED are turned ON to indicate the darkness.

5.5.9 Suggestions

In this project a buzzer and an LED are used as the output devices. It is possible to connect a relay to an output port of the ESP32 DevKitC and, for example, turn ON the room lights automatically when it becomes dark.

5.6 PROJECT 5 – LED Dice

5.6.1 Description

This is a simple dice project based on LEDs and a push-button switch. The LEDs are organised to simulate the look of the faces of a real dice. When the push-button switch is pressed a random number is generated between 1 and 6 and is displayed on the LEDs. Normally the LEDs are all OFF to indicate that the system is ready to generate a new dice number. After 3 seconds the LEDs turn OFF again.

5.6.2 The Aim

The aim of this project is to show how an LED based dice can be designed.

5.6.3 Block diagram:

Figure 5.32 shows the block diagram of the project.

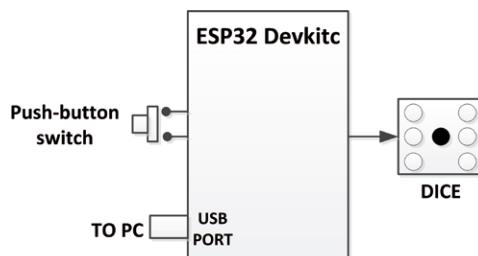


Figure 5.32 Block diagram of the project

As shown in Figure 5.33, the LEDs are organised such that when they turn ON, they indicate the numbers as in a real dice. Operation of the project is as follows: Normally the LEDs are all OFF to indicate that the system is ready to generate a new dice number. Pressing the switch generates a random dice number between 1 and 6 and displays on the LEDs for 3 seconds. After 3 seconds the LEDs turn OFF again.

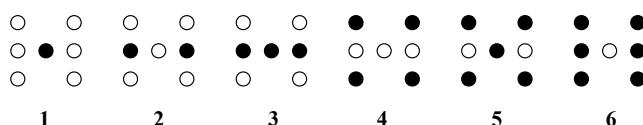


Figure 5.33 IFD dice

5.6.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 5.34. The 7 LEDs are connected to GPIO pins 23, 22, 1, 3, 21, 19, and 18 through 330 ohm current limiting resistors. The connection is as follows:

LED	GPIO pin
D1	23
D2	1
D3	19

D4	21
D5	22
D6	3
D7	18

The push-button switch is connected to GPIO pin 5 through a 10K pull-up resistor so that normally the button output is at logic 1 and it goes to logic 0 when the button is pressed.

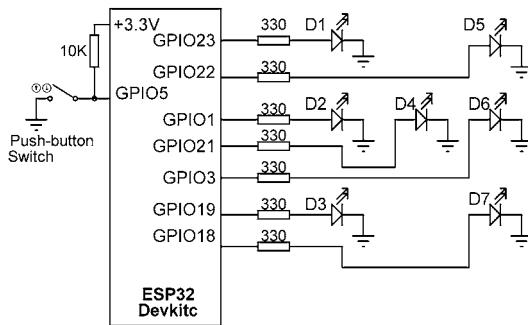


Figure 5.34 Circuit diagram of the project

5.6.5 Construction

The ESP32 DevKitC board is mounted on a breadboard with the 7 LEDs and the push-button switch as shown in Figure 5.35.

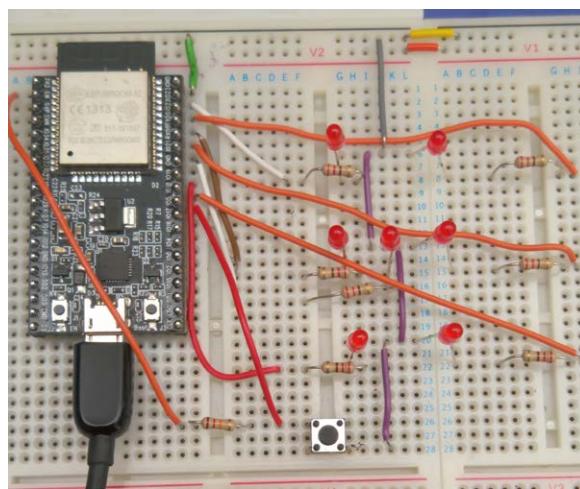


Figure 5.35 The project constructed on a breadboard

5.6.6 PDL of the Project

The PDL of the project is shown in Figure 5.36. The relationship between the required number and the LEDs to be turned ON is shown in Table 5.1. For example, to display number 1, only LED D1 must be turned ON. Similarly, to display number 3, LEDs D2, D4, D6 must be turned ON and so on.

Required Number	LEDs to be Turned ON
1	D4
2	D2, D6
3	D2, D4, D6
4	D1, D3, D5, D7
5	D1, D3, D4, D5, D7
6	D1, D2, D3, D5, D6, D7

Table 5.1 Dice numbers and LEDs to be turned ON

```

BEGIN
    Assign Button to GPIO5
    Assign LEDs to the GPIOs
    Configure all LEDs as outputs
    Configure Button as input
DO FOREVER
    Turn OFF all LEDs
    Wait until the Button is pressed
    Generate a random number between 1 and 6
    Turn ON the appropriate LED to display the number
    Wait for 3 seconds
ENDDO
END

```

Figure 5.36 PDL of the project

5.6.7 Program Listing

The program listing is shown in Figure 5.37 (program: **DICE**).

```

/*****
 *          LED DICE
 *          =====
 *
 * In this program 7 LEDs are connected in to the ESP32 Devkitc
 * to simulate the faces of a dice. In addition, a push-button
 * switch is used. When the switch is pressed, a random number
 * is generated between 1 and 6 and the LEDs are turned ON to
 * indicate the number as if it is a real dice. The LEDs display
 * the number for 3 seconds and then turn OFF to indicate that
 * the system is ready so that the user can press the button
 * again if desired.
 *
 * The connections of the LEDs and the push-button switch are
 * as follows:
 *
 * D1: GPIO23, D2: GPIO1, D3: GPIO19, D4: GPIO21, D5: GPIO22
 * D6: GPIO3,   D7: GPIO18

```

```
* Push button switch: GPIO5
*
*
* Program: DICE
* Date    : July, 2017
*****/
#define Button 5
#define D1 23
#define D5 22
#define D2 1
#define D6 3
#define D4 21
#define D3 19
#define D7 18
int LEDs[] = {23, 22, 1, 3, 21, 19, 18};

//
// Configure the LEDs as outputs, and the Button as input
//
void setup()
{
    for(int i = 0; i < 7; i++)
    {
        pinMode(LEDs[i], OUTPUT);
    }
    pinMode(Button, INPUT);
    randomSeed(10);

}

//
// Turn all LEDs OFF
//
void ALL_OFF()
{
    for(int i = 0; i < 7; i++)
    {
        digitalWrite(LEDs[i], LOW);
    }
}

//


// Turn OFF all LEDS so that the system is ready to accept a
// new button press. Wait until the Button is pressed. Then,
// generate a random number between 1 and 6. Turn ON the
```

```
// appropriate LEDs to display the number as a real dice face.  
// The program displays the number for 3 seconds and then all  
// the LEDs are turned OFF, ready for the next button press.  
//  
void loop()  
{  
    ALL_OFF();                                // Turn OFF all LEDs  
    while(digitalRead(Button) == 1);           // Wait for Button pressed  
    int dice = random(1, 7);                  // Generate a random number  
  
    switch (dice)  
    {  
        case 1:                                // Number 1  
            digitalWrite(D4, HIGH);             // Turn ON D4  
            break;  
        case 2:                                // Number 2  
            digitalWrite(D2, HIGH);             // Turn ON D2,D6  
            digitalWrite(D6, HIGH);  
            break;  
        case 3:                                // Number 3  
            digitalWrite(D2, HIGH);             // Turn ON D2,D4,D6  
            digitalWrite(D4, HIGH);  
            digitalWrite(D6, HIGH);  
            break;  
        case 4:                                // Number 4  
            digitalWrite(D1, HIGH);              // Turn ON D1,D3,D5,D7  
            digitalWrite(D3, HIGH);  
            digitalWrite(D5, HIGH);  
            digitalWrite(D7, HIGH);  
            break;  
        case 5:                                // Number 5  
            digitalWrite(D1, HIGH);              // Turn ON D1,D3,D4,D5,D7  
            digitalWrite(D3, HIGH);  
            digitalWrite(D4, HIGH);  
            digitalWrite(D5, HIGH);  
            digitalWrite(D7, HIGH);  
            break;  
        case 6:                                // Number 6  
            digitalWrite(D1, HIGH);              // Turn ON D1,D2,D3,D5,D6,D7  
            digitalWrite(D2, HIGH);  
            digitalWrite(D3, HIGH);  
            digitalWrite(D5, HIGH);  
            digitalWrite(D6, HIGH);  
            digitalWrite(D7, HIGH);  
            break;  
    }  
}
```

```

    delay(3000); // Display for 3 seconds
}

```

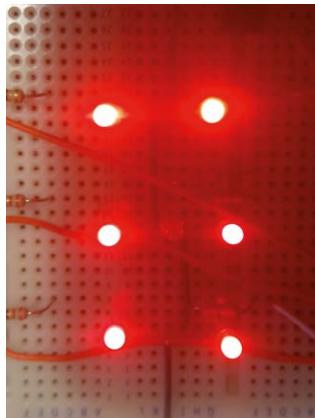
Figure 5.37 Program listing

5.6.8 Program Description

At the beginning of the program **Button** is assigned to GPIO port 5. Similarly, LEDs **D1, D2, D3, D4, D5, D6** are assigned to GPIO ports 23, 1, 19, 21, 22, 3 and 18 respectively. All the LEDs are configured as outputs and the Button is configured as an input.

The main program is executed in an endless loop. Inside this loop all the LEDs are turned OFF initially. The program then waits until the Button is pressed. When the button is pressed a random number is generated between 1 and 6. A **switch** statement is then used to turn ON the appropriate LEDs depending upon the generated number. The program displays the generated number for 3 seconds. After this time the above process is repeated.

Figure 5.38 shows an example where number 6 is displayed.

*Figure 5.38 Displaying number 6*

5.7 PROJECT 6 – Logic Probe

5.7.1 Description

This project is a simple logic probe. A logic probe is used to indicate the logic status of an unknown digital signal. In a typical application a test lead (probe) is used to detect the unknown signal and two different colour LEDs are used to indicate the logic status. If, for example, the signal is logic 0, then the RED colour LED is turned ON. If, on the other hand, the signal is logic 1, then the GREEN LED is turned ON.

5.7.2 The Aim

The aim of this project is to show how a logic probe can be designed to test the logic state of an applied logic signal.

5.7.3 Block diagram:

Figure 5.39 shows the block diagram of the project. The RED and GREEN LEDs are connected to GPIO ports 23 and 22 respectively. The logic signal to be tested is applied to GPIO port 1.

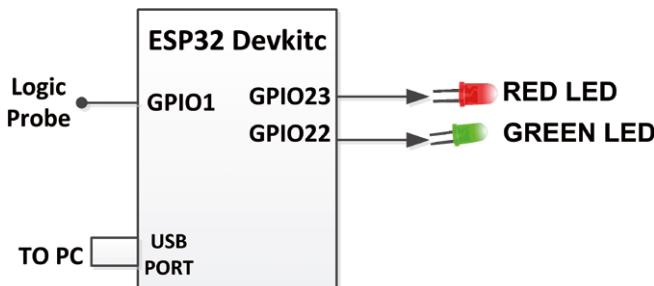


Figure 5.39 Block diagram of the project

5.7.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 5.40. The LEDs are connected to GPIO pins through 330 ohm current limiting resistors. The logic signal to be tested is directly applied to GPIO port 1.

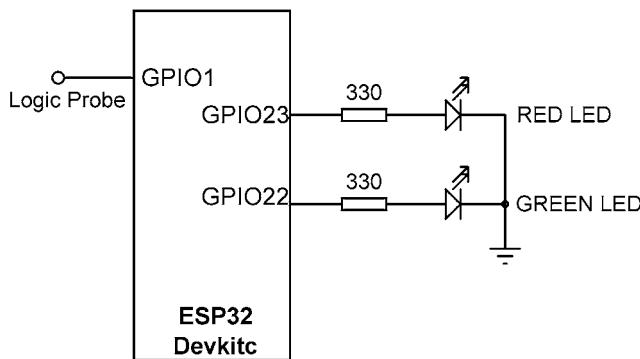


Figure 5.40 Circuit diagram of the project

5.7.5 Construction

The ESP32 DevKitC board is mounted on a breadboard with the two LEDs as shown in Figure 5.41.

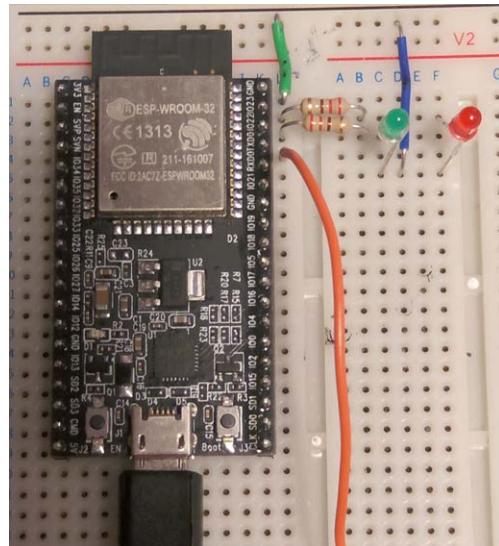


Figure 5.41 The project constructed on a breadboard

5.7.6 PDL of the Project

The PDL of the project is shown in Figure 5.42. If the external logic state is 0 then the RED LED is turned ON and the GREEN LED is turned OFF. If on the other hand the external logic state is 1 then the GREEN LED is turned ON and the RED LED is turned OFF.

```

BEGIN
    Assign RED to GPIO port 23
    Assign GREEN to GPIO port 22
    Assign Probe to GPIO port 1
    Configure GPIO ports 22,23 as outputs
    Configure GPIO port 1 as input
DO FOREVER
    IF Probe is logic 0 THEN
        Turn ON RED LED
        Turn Off GREEN LED
    ELSE
        Turn ON GREEN LED
        Turn OFF RED LED
    ENDIF
ENDDO
END

```

Figure 5.42 PDL of the project

5.7.7 Program Listing

The program listing is shown in Figure 5.43 (program: **Probe**).

```
*****  
*          LOGIC PROBE  
*          ======  
*  
* This is a logic probe program. Two LEDs, one RED and one  
* GREEN are connected to GPIO ports 23 and 22 respectively.  
* The program determines the state of the logic signal applied  
* to GPIO port 5. If the applied logic level is 0 then the  
* RED LED is turned ON. If on the other hand the applied  
* logic level is 1 then the GREEN LED is turned ON.  
*  
*  
* Program: Probe  
* Date   : July, 2017  
*****/  
  
#define Probe 1  
#define RED 23  
#define GREEN 22  
  
//  
// Configure the LEDs as outputs, and the Probe as input  
//  
void setup()  
{  
    pinMode(RED, OUTPUT);  
    pinMode(GREEN, OUTPUT);  
    pinMode(Probe, INPUT);  
}  
  
//  
// Examine the state of the applied external logic signal. If  
// the state is 0, then turn ON the RED LED. if on the other  
// hand the external applied logic state is 1, then turn ON the  
// GREEN LED  
//  
void loop()  
{  
    if(digitalRead(Probe) == 0)           // Logic state is 0  
    {  
        digitalWrite(RED, HIGH);         // RED ON  
        digitalWrite(GREEN, LOW);        // GREEN OFF  
    }  
    else                                // Logic state 1
```

```

    {
        digitalWrite(GREEN, HIGH);           // GREEN ON
        digitalWrite(RED, LOW);            // RED OFF
    }
}

```

Figure 5.43 Program listing

5.7.8 Program Description

At the beginning of the program, **Probe** is assigned to GPIO port 1, **RED LED** is assigned to GPIO port 23 and the **GREEN LED** is assigned to GPIO port 22. The LED ports are then configured as outputs and the Probe is configured as an input.

The remainder of the program is executed in an endless loop. Inside this loop, the state of the external logic signal is read and if it is logic 0 then the RED LED is turned ON, otherwise the GREEN LED is turned ON. This process is repeated continuously.

5.7.9 Modified Program

The logic probe program shown in Figure 5.43 has a problem that one of the LEDs is always ON, even if the probe is not connected to any external digital signal (e.g. the high-impedance state). We can develop the project further so that the high-impedance state can also be detected and none of the LEDs are turned ON when there is no applied logic level.

Figure 5.44 shows the modified circuit diagram. Notice here that a transistor (BC108 or any other npn transistor) is used at the front end of the circuit. The operation of the circuit is as follows:

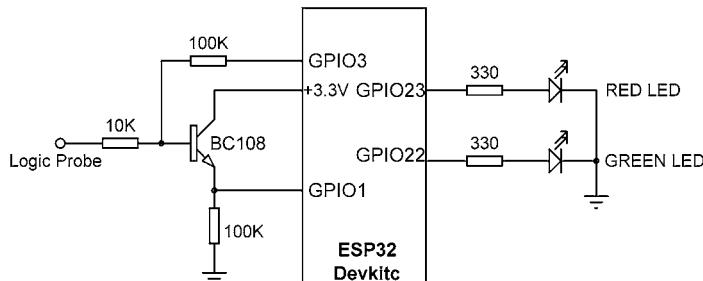


Figure 5.44 Modified circuit diagram

The transistor is configured as an emitter-follower stage with the base connected to GPIO port 3, configured as an output port. The external logic signal is applied to the base of the transistor through a 10K resistor. The emitter of the transistor is connected to GPIO port 1, which is configured as an input port. GPIO port 3 applies logic levels to the base of the transistor and then pin GPIO port 1 determines the state of the external signal as shown in Table 5.2. For example, if after setting **port 3 = 1**, we detect that **port 1 = 1** and also after setting **port 3 = 0** we again detect that **port 1 = 1**, then the probe must be at logic 1.

Probe State	Output from GPIO3	Detected at GPIO1
At high impedance	1 0	1 0
Probe at logic 1	1 0	1 1
Probe at logic 0	1 0	0 0

Table 5.2 Applied and detected logic levels

Figure 5.45 shows the modified program listing (program: **Probe2**). At the beginning of the program **Probe** is assigned to **GPIO1**, **RED** and **GREEN** assigned to GPIO23 and GPIO22 respectively, and **CHECK** assigned to GPIO3. The LEDs and CHECK are then configured as digital outputs and Probe configured as digital input.

The remainder of the program executes in an endless loop and implements the logic given in Table 5.2 in order to turn ON/OFF the RED and GREEN LEDs.

```
*****
*          LOGIC PROBE
*          =====
*
* This is a logic probe program. Two LEDs, one RED and one
* GREEN are connected to GPIO ports 23 and 22 respectively.
* The program determines the state of the logic signal applied
* to GPIO port 5. If the applied logic level is 0 then the
* RED LED is turned ON. If on the other hand the applied
* logic level is 1 then the GREEN LED is turned ON.
*
* This modified program detects the high impedance state when
* there is no logic level applied at the input
*
*
* Program: Probe2
* Date   : July, 2017
*****
#define Probe 1
#define RED 23
#define GREEN 22
#define CHECK 3

// Configure the LEDs as outputs, and the Probe as input
//
void setup()
{
```

```
pinMode(RED, OUTPUT);
pinMode(GREEN, OUTPUT);
pinMode(CHECK, OUTPUT);
pinMode(Probe, INPUT);
}

// 
// Examine the state of the applied external logic signal. If
// there is nothing connected at the input then assume high
// impedance state and turn OFF both LEDs. If on the other
// hand the state is 0, then turn ON the RED LED. if on the
// other hand the external applied logic state is 1, then
// turn ON the GREEN LED
//
void loop()
{
    digitalWrite(CHECK, HIGH);           // Set CHECK = 1
    delay(1);
    if(digitalRead(Probe) == 1)
    {
        digitalWrite(CHECK, LOW);
        delay(1);
        if(digitalRead(Probe) == 0)
        {
            digitalWrite(RED, LOW);      // High impedance state
            digitalWrite(GREEN, LOW);
        }
        else
        {
            digitalWrite(RED, LOW);
            digitalWrite(GREEN, HIGH);
        }
    }
    else
    {
        digitalWrite(CHECK, LOW);
        delay(1);
        if(digitalRead(Probe) == 0)
        {
            digitalWrite(GREEN, LOW);
            digitalWrite(RED, HIGH);
        }
    }
}
```

Figure 5.45 Modified program

The modified circuit built on a breadboard is shown in Figure 5.46.

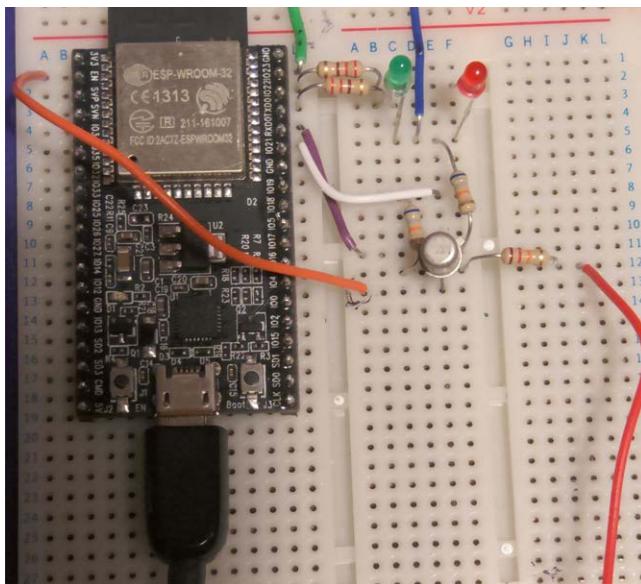


Figure 5.46 Modified circuit built on a breadboard

5.8 PROJECT 7 – 7 - Segment LED Display Counter

5.8.1 Description

This is a simple project and it describes the design of a 7-segment LED based counter which counts from 0 to 9 continuously with one second delay between each count.

7-segment displays are used frequently in electronic circuits to show numeric or alphanumeric values. A 7-segment display basically consists of 7 LEDs (see Figure 5.47) connected such that numbers from 0 to 9 and some letters can be displayed. Segments are identified by letters from a to g and Figure 5.48 shows the segment names of a typical 7-segment LED display.

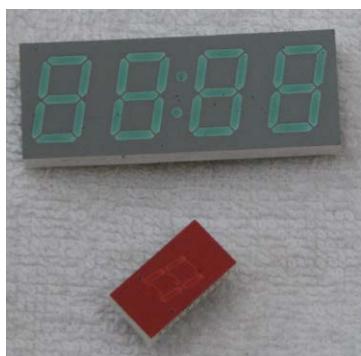


Figure 5.47 Some 7-segment displays

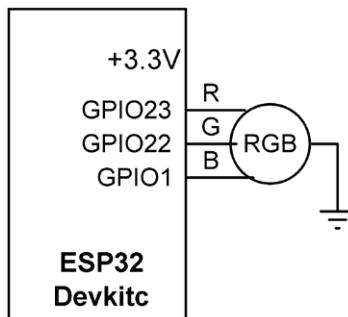


Figure 5.48 Segments of a 7-segment display

Figure 5.49 shows how numbers from 0 to 9 can be obtained by turning ON/OFF different segments of the display.

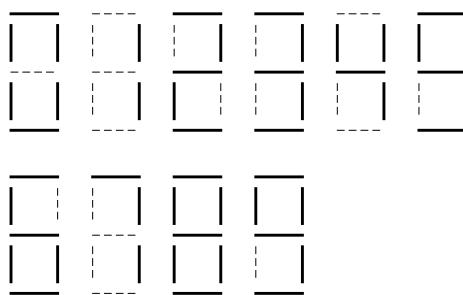


Figure 5.49 Displaying numbers 0 – 9

7-segment displays are available as **common cathode** or **common anode**. In common cathode configuration (see Figure 5.50) the cathodes of all the segment LEDs are connected together to ground. The individual segments are then turned ON by applying a logic 1 to the required segment via current limiting resistors.

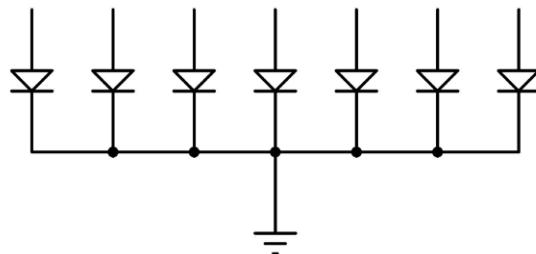


Figure 5.50 Common cathode 7-segment display

In a common anode configuration, the anode terminals of all the LEDs are connected together as shown in Figure 5.51. This common point is then normally connected to the supply voltage. A segment is turned ON by connecting its cathode terminal to logic 0 via a current limiting resistor.

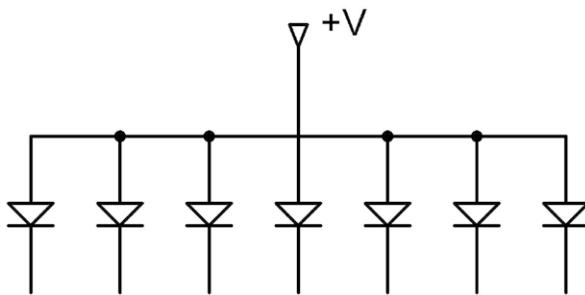


Figure 5.51 Common anode 7-segment display

In this project, a **SMA42056** model common cathode 7-segment display is used. This is a 14.20 mm (0.56 inch) large display that can be viewed from a distance up to 7 metres. The LED forward voltage is about 2 V. The display has 10 pins and it also has a segment LED for the decimal point. Table 5.3 shows the pin configuration of this display. The pin numbering starts from the bottom left corner of the display with pin 1 as shown in Figure 5.52. The bottom right corner is pin 5, and the top left corner is pin 10.

Pin number	Segment
1	E
2	D
3	common cathode
4	C
5	decimal point
6	B
7	A
8	common cathode
9	F
10	G

Table 5.3 SMA42056 7-segment LED pin configuration

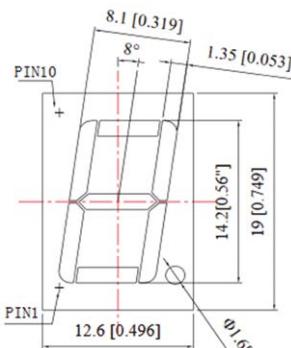


Figure 5.52 Pin numbering of the SMA42056 display

5.8.2 The Aim

The aim of this project is to show how a 7-segment LED can be interfaced and used in an ESP32 DevKitC project.

5.8.3 Block diagram

The block diagram of the project is shown in Figure 5.53.

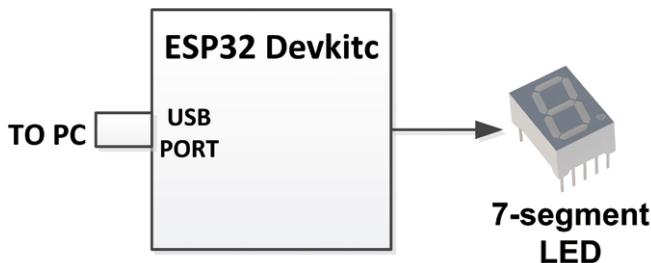


Figure 5.53 Block diagram of the project

5.8.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 5.54. The segments of the LED are connected to GPIO pins through 330 ohm current limiting resistors. The connection between the display segments and the GPIO pins are as follows:

Segment	GPIO pin
a	23
b	22
c	1
d	3
e	21
f	19
g	18

Before driving the display we have to know the relationship between the numbers to be displayed and the GPIO pins that should be HIGH. This is shown below:

Number to be Displayed	GPIO pins to be HIGH
0	23,22,1,3,21,19
1	22,1
2	23,22,18,21,3
3	23,22,18,1,3
4	19,18,22,1
5	23,19,18,1,3
6	19,18,1,3,21
7	23,22,1
8	23,22,1,3,21,19,18
9	23,22,19,18,1

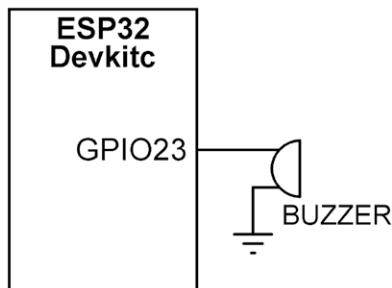


Figure 5.54 Circuit diagram of the project

5.8.5 Construction

The ESP32 DevKitC board is mounted on a breadboard with the 7-segment LED and the current limiting resistors as shown in Figure 5.55.

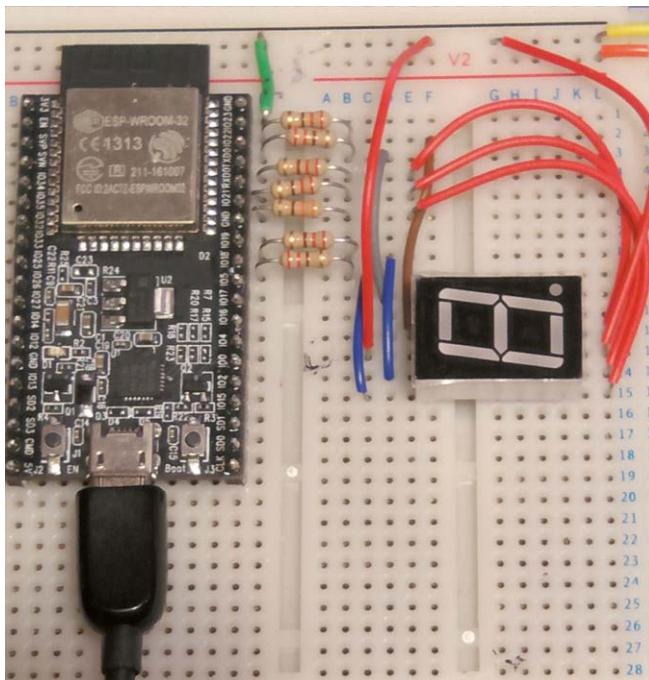


Figure 5.55 The project constructed on a breadboard

5.8.6 PDL of the Project

The PDL of the project is shown in Figure 5.56. Numbers from 0 to 9 are displayed on the 7-segment LED with one second display between each count.

BEGIN

 Declare array Num and store segments to be turned ON for every number

 Declare array Strt and store starting index of each number within Num

 Declare array LEDs and store all the GPIO port numbers used

```

Set Count to 0
Configure all used GPIO ports as outputs
DO FOREVER
    Display Count
    Wait one second
    Increment Count
    Turn OFF all segments
    IF Count = 10 THEN
        Count = 0
    ENDIF
ENDDO
END

```

Figure 5.56 PDL of the project

5.8.7 Program Listing

The program listing is shown in Figure 5.57 (program: **SevenSegment**).

```

/*****
*          7-SEGMENT LED COUNTER
=====
*
* In this project a 7-segment LED is connected to the ESP32
* DevKitC. The program counts up from 0 to 9 with one second
* intervals. The connections between the GPIO pins and the
* 7-segment LED segments are as follows:
*
* Segment      GPIO pin
*   a           23
*   b           22
*   c           1
*   d           3
*   e           21
*   f           19
*   g           18
*
*
* Program: SevenSegment
* Date   : July, 2017
*****
int Num[] = {6, 23, 22, 1, 3, 21, 19,           //0
             2, 22, 1,                   //1
             5, 23, 22, 18, 21, 3,       //2
             5, 23, 22, 18, 1, 3,       //3
             4, 19, 18, 22, 1,          //4
             5, 23, 19, 18, 1, 3,       //5

```

```
    5, 19, 18, 1, 3, 21,                                //6
    3, 23, 22, 1,                                     //7
    7, 23, 22, 1, 3, 21, 19, 18,                         //8
    5, 23, 22, 19, 18, 1};                                //9
int Strt[] = {0, 7, 10, 16, 22, 27, 33, 39, 43, 51};
int LEDs[] = {23, 22, 1, 3, 21, 19, 18};
int Count = 0;

//
// Configure the LEDs as outputs, and also turn OFF all segments
//
void setup()
{
    for(int i = 0; i < 7; i++)
    {
        pinMode(LEDs[i], OUTPUT);
        ALL_OFF();
    }
}

//
// Turn OFF all segments
//
void ALL_OFF()
{
    for(int i = 0; i < 8; i++)
    {
        digitalWrite(LEDs[i], LOW);
    }
}

//
// This function display a given number N on the 7-segment display
//
void Display(int N)
{
    int i, k;
    k = Strt[N];                                         // Starting index
    i = Num[k];                                           // Length
    for(int m = k + 1; m < k + 1 + i; m++)
    {
        digitalWrite(Num[m], HIGH);
    }
}

//
// Increment variable Count by 1 and display on the 7-segment
// display every second
```

```

//  

//  

void loop()  

{  

    Display(Count);  

    delay(1000);  

    ALL_OFF();  

    Count++;  

    if(Count == 10)Count = 0;  

}

```

Figure 5.57 Program listing

5.8.8 Program Description

At the beginning of the program three arrays are declared: **Num**, **Strt**, and **LEDs**. Array **Num** stores the segments to be turned ON in terms of the GPIO port numbers where the rows correspond to the numbers to be displayed and the first number in every row is the number of segments to be turned ON for the given number. For example, the row corresponding to number 3 is:

```
5, 23, 22, 18, 1, 3,           //3
```

The first number 5 here is the number of segments to be turned ON and these segments are connected to GPIO ports 23, 22, 18, 1 and 3.

The second array **Strt** stores the starting index number within array **Num** of every number to be displayed. For example, number 3 starting index is 16 in array **Num** where the 16th element of array **Num** is the number of segments to be turned ON as described above.

Array **LEDs** stores the GPIO port addresses of all the LEDs and this array is used to configure all the ports as outputs and also to turn OFF all the segments when required.

Function **Display** receives the number to be displayed as its argument. This function extracts the segments to be turned ON and then turns them ON using the digitalWrite statement.

The main program is executed in an endless loop. Inside this loop, the current value of variable **Count** is displayed. After one second delay, all the segments are turned OFF ready for the next number. **Count** is incremented by one and it is reset to 0 when it reaches 10.

5.8.9 Modified Program

The program given in Figure 5.57 can be simplified and made more user-friendly if we group the GPIO ports into an 8-bit byte and then access all the port bits by sending a byte to this port. This technique was shown in Figure 4.22 back in Chapter 4.

The modified program listing is shown in Figure 5.58 (program: **SevenSegment2**). Here, the GPIO port bits are grouped as follows:

MSB								LSB
X	23	22	1	3	21	19	18	

X, here, is not used. The relationship between the numbers to be displayed and the number to be sent to the grouped port can be derived as follows (X is taken as 0):

Number	Port Bits to be Turned ON								Number to be Sent (in Hex)
	X	23	22	1	3	21	19	18	
0	x	1	1	1	1	1	1	0	7E
1	x	0	1	1	0	0	0	0	30
2	x	1	1	0	1	1	0	1	6D
3	x	1	1	1	1	0	0	1	79
4	x	0	1	1	0	0	1	1	33
5	x	1	0	1	1	0	1	1	5B
6	x	0	0	1	1	1	1	1	1F
7	x	1	1	1	0	0	0	0	70
8	x	1	1	1	1	1	1	1	7F
9	x	1	1	1	0	0	1	1	73

```
*****
*          7-SEGMENT LED COUNTER
*          =====
*
* In this project a 7-segment LED is connected to the ESP32
* DevKitC. The program counts up from 0 to 9 with one second
* intervals. The connections between the GPIO pins and the
* 7-segment LED segments are as follows:
*
* Segment      GPIO pin
*   a          23
*   b          22
*   c          1
*   d          3
*   e          21
*   f          19
*   g          18
*
*
* Program: SevenSegment2
* Date    : July, 2017
*****
int Num[] = {0x7E, 0x30, 0x6D, 0x79, 0x33, 0x5B, 0x1F, 0x70, 0x7F, 0x73};
int LEDs[] = {23, 22, 1, 3, 21, 19, 18};
int Ports[] = {0, 23, 22, 1, 3, 21, 19, 18};
int Count = 0;
```

```
//  
// Configure the LEDs as outputs  
//  
void setup()  
{  
    for(int i = 0; i < 7; i++)  
    {  
        pinMode(LEDs[i], OUTPUT);  
    }  
}  
  
//  
// Turn ON the appropriate LED  
//  
void Display(int No, unsigned char L)  
{  
    unsigned char j, m, i;  
    m = L - 1;  
    for(i = 0; i < L; i++)  
    {  
        j = pow(2, m);  
        digitalWrite(Ports[i], (No & j));  
        m--;  
    }  
}  
  
//  
// Increment variable Count by 1 and display on the 7-segment  
// display every second  
//  
//  
void loop()  
{  
    Display(Num[Count], 8);  
    delay(1000);  
    Count++;  
    if(Count == 10) Count = 0;  
}
```

Figure 5.58 Modified program

5.9 PROJECT 8 – Clap ON – Clap OFF

5.9.1 Description

In this project a sound sensor module (called the Small Microphone Module) and an LED are used. The LED changes state (i.e. is toggled) when sound is detected by the sound sensor module, such as clapping hands, where if the LED is OFF, it turns ON when we clap hands near the microphone. Clapping hands again turns the LED back OFF. In this project an LED is used, but there is no reason why the LED cannot be replaced by a relay and for example room lights or garage door connected to the relay.

5.9.2 The Aim

The aim of this project is to show how a sound sensor module can be connected to the ESP32 DevKitC and how it can be used to control an external device such as an LED, relay, or any other logic operated device.

5.9.3 Block diagram

In this project the Small Microphone Module (available from Elektor) is used (see Figure 5.59). This is a 4-pin module having the following pin definitions:

A0:	analog output
G:	supply ground
+:	power supply
D0:	digital output

The analog output is not used in this project. A small potentiometer is used to adjust the triggering point (i.e. the sensitivity) of the digital output such that the output is set to be normally logic LOW and goes to logic HIGH when sound is detected by the microphone.

The block diagram of the project is shown in Figure 5.60 where the GPIO pins 23 and 22 are used for the sound sensor module and the LED respectively.

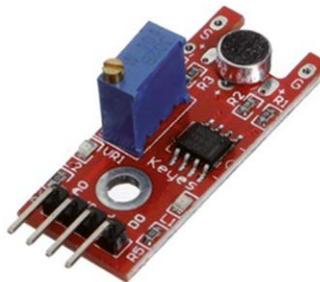


Figure 5.59 Small Microphone Module

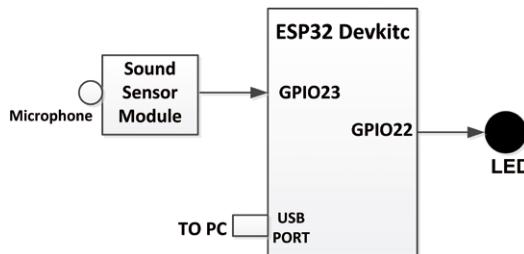


Figure 5.60 Block diagram of the project

5.9.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 5.61. The digital output D0 of the sound sensor module is connected to GPIO port 23 of the ESP32 DevKitC. The LED is connected to GPIO port 22 through a 330 ohm current limiting resistor.

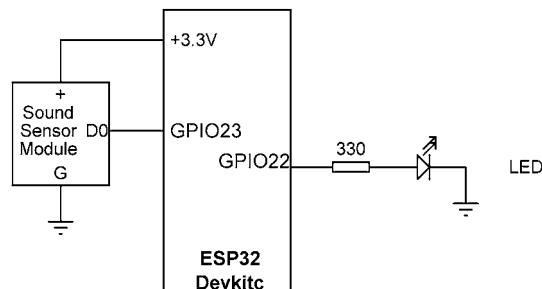


Figure 5.61 Circuit diagram of the project

5.9.5 Construction

The ESP32 DevKitC board is mounted on a breadboard with the sound sensor module and the LED connected to it as shown in Figure 5.62.

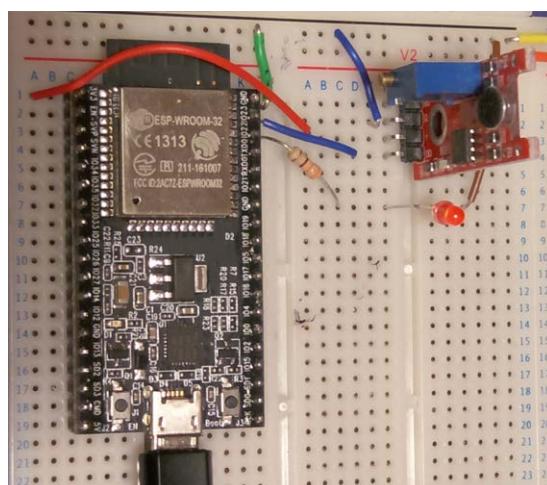


Figure 5.62 The project constructed on a breadboard

5.9.6 PDL of the Project

The PDL of the project is shown in Figure 5.63.

```
BEGIN
    Assign GPIO port 22 to the LED
    Assign GPIO port 23 to the sound sensor module
    Configure GPIO22 as output
    Configure GPIO23 as input
    DO FOREVER
        IF output of sound sensor module = 1
            Toggle the LED
            Wait 500 ms
        ENDIF
    ENDO
END
```

Figure 5.63 PDL of the project

5.9.7 Program Listing

The program listing is shown in Figure 5.64 (program: **Sound**).

```
/*********************************************
*
*          CLAP ON - CLAP OFF
*
*=====
*
* In this project a sound sensor module (called the Small
* Microphone Module) is connected to GPIO port 23 of the
* ESP32 Devkitc. In addition, an LED is connected to GPIO
* port 22. The LED is normally OFF and is turned ON when
* sound is detected by the sensor (e.g. by clapping hands).
* The LED is toggled every time sound is detected by the
* sensor.
*
*****************************************/
#define LED 22
#define SoundSensor 23
int state = 0;

//
// Configure GPIO ports LED as output and SoundSensor as input
//
void setup ()
{
    pinMode (LED, OUTPUT);
    pinMode(SoundSensor, INPUT);
    digitalWrite(LED, LOW);
```

```

}

// 
// Toggle the LED when sound is detected by the sensor
//
void loop ()
{
    if(digitalRead(SoundSensor) == 1)
    {
        digitalWrite(LED, (state) ? HIGH : LOW);
        state = !state;
        delay(500);
    }
}

```

Figure 5.64 Program listing

5.9.8 Program Description

At the beginning of the program GPIO 23 and 22 are assigned to the LED and the sound sensor module respectively. The LED port is configured as output and the sound sensor port is configured as input.

The main program runs in an endless loop. Inside this loop, the output of the sound sensor is read. If the output is at logic 1 (i.e. sound is detected) then the state of the LED is toggled. This process is repeated after 500 milliseconds.

5.10 PROJECT 9 – LCD "Hello from ESP32"

5.10.1 Description

This is a simple LCD project. The project displays the text **Hello from ESP32** on the first row of a 16x2 character LCD.

5.10.2 The Aim

The aim of this project is to show how a character LCD can be interfaced and used in an ESP32 DevKitC project. An I2C compatible 16 character by 2 row LCD is used in this project.

5.10.3 Block diagram

The LCD used in this project is based on the I2C (or IIC) interface. I2C is a multi-slave, multi-master, single-ended serial bus used to attach low-speed peripheral devices to microcontrollers. The bus consists of only two wires called SDA and SCL where SDA is the data line and SCL is the clock line and up to 1008 slave devices can be supported on the bus. Both lines must be pulled up to the supply voltage by suitable resistors. The clock signal is always generated by the bus master. The devices on the I2C bus can communicate at 100 kHz or 400 kHz.

GPIO ports 21 and 22 are available for use by the I2C bus interface on the ESP32 DevKitC, where port 21 is the data line (SDA) and port 22 is the clock line (SCL).

Figure 5.65 shows the front and back of the I2C based LCD. Notice that the LCD has a small board mounted at its back to control the I2C interface. The LCD contrast is adjusted through the small potentiometer mounted on this board. A jumper is provided on this board to disable the backlight if required. The block diagram of the project is shown in Figure 5.66, where the GPIO pins 21 and 22 are used for the SDA and SCL of the LCD respectively.



Figure 5.65 I2C compatible LCD (front and back views)

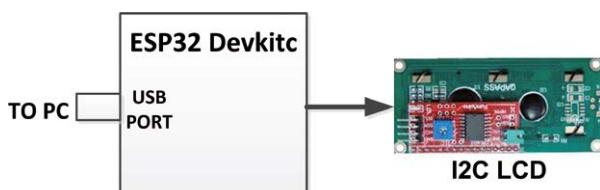


Figure 5.66 Block diagram of the project

5.10.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 5.67. The interface between the I2C LCD and the microcontroller is via 4 pins. Notice that the VCC pin must be connected to +5 V supply and not to +3.3 V supply. The +5 V supply is the bottom pin at the left hand side of the ESP32 DevKitC:

- GND: power ground
- VCC: +5 V power supply
- SDA: I2C data line
- SCL: I2C clock line

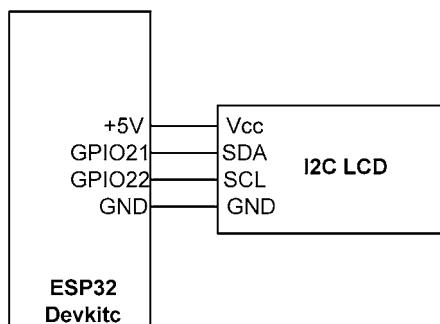


Figure 5.67 Circuit diagram of the project

5.10.5 Construction

The ESP32 DevKitC board is mounted on a breadboard and jumper wires are used to make connection to the I2C LCD as shown in Figure 5.68.

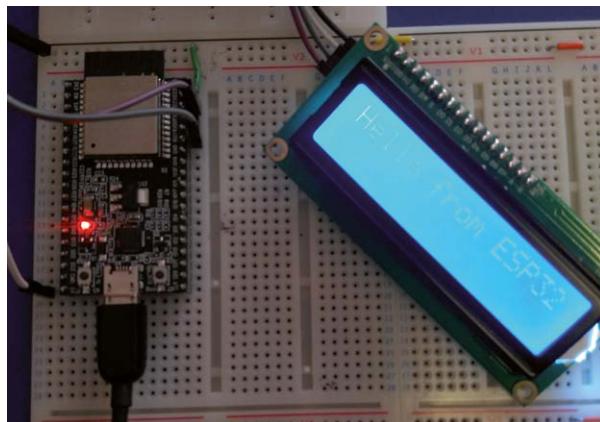


Figure 5.68 The project constructed on a breadboard

5.10.6 PDL of the Project

Before programming the ESP32 for the I2C LCD, it is necessary to download and include the I2C LCD library in our Arduino IDE folder. The steps to do this are given below:

- Create a folder named LiquidCrystal_I2C under the folder named libraries in your Arduino IDE folder (see Figure 5.69).

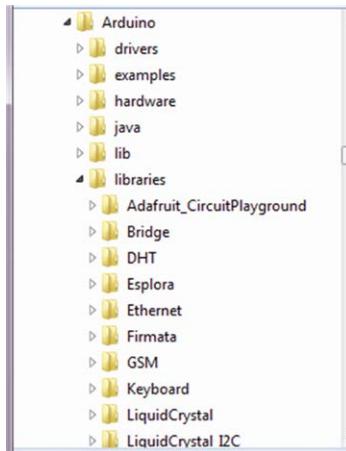
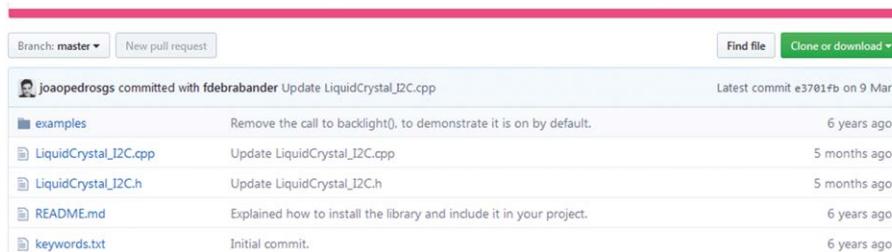


Figure 5.69 Create folder LiquidCrystal_I2C

- Go to the following link to download the I2C LCD library:

<https://github.com/fdebrabander/Arduino-LiquidCrystal-I2C-library>

- Click the button **Clone or download** (see Figure 5.70) and copy all the files to the **LiquidCrystal_I2C** folder as shown in Figure 5.71.



A screenshot of a GitHub repository page for the 'LiquidCrystal_I2C' folder. The repository has 1 commit and 6 files. The files are:

File	Description	Last Commit
examples	Remove the call to backlight(). to demonstrate it is on by default.	6 years ago
LiquidCrystal_I2C.cpp	Update LiquidCrystal_I2C.cpp	5 months ago
LiquidCrystal_I2C.h	Update LiquidCrystal_I2C.h	5 months ago
README.md	Explained how to install the library and include it in your project.	6 years ago
keywords.txt	Initial commit.	6 years ago

Figure 5.70 Download the I2C library files

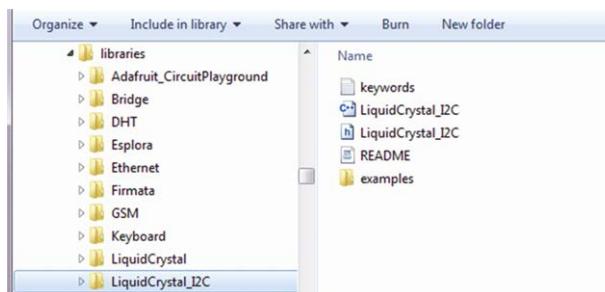


Figure 5.71 Files in the LiquidCrystal_I2C folder

Start the Arduino IDE. Go to **File -> Examples** and you should see **LiquidCrystal_I2C** examples in the drop down menu if the library has been installed correctly.

The PDL of the project is shown in Figure 5.72.

```

BEGIN
Include libraries Wire and LiquidCrystal_I2C
Set I2C LCD address and configuration
Initialize I2C LCD
Turn ON backlight
Display text Hello from ESP32
END

```

Figure 5.72 PDL of the project

5.10.7 Program Listing

The program listing is shown in Figure 5.73 (program: **LCD**).

```

*****
*                               LCD TEXT
*                               =====
*
```

```

* This program displays text "Hello From ESP32" on an I2C LCD.
* The LCD is connected to SDA and SCL port pins of the ESP32
* DevKitC.
*
* File:    LCD
* Date:   July 2017
* Author: Dogan Ibrahim
***** */
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

//
// Set the LCD address to 0x27 and the configuration to
// 16 chars and 2 rows display
//
LiquidCrystal_I2C lcd(0x27, 16, 2);      // LCD address 0x27

void setup()
{
    lcd.begin();                      // Initialize LCD
    lcd.backlight();                  // Turn ON backlight
    lcd.print("Hello from ESP32");    // Display Text
}

void loop()
{
    // No code here
}

```

Figure 5.73 Program listing

5.10.8 Program Description

At the beginning of the program libraries **Wire** (I2C library) and **LiquidCrystal_I2C** are included in the program. Then the address of the LCD (0x27) and its configuration (16 columns by 2 rows) are defined in the I2C LCD library. The program then initialises the I2C LCD library, turns ON the backlight, and displays the message **Hello from ESP32**.

5.11 PROJECT 10 – LCD Event Counter

5.11.1 Description

This is an LCD event counter project. The project counts external events occurring on one of the GPIO ports and displays the results on an LCD. In this project an event is said to occur if the state of GPIO port 23 changes from logic 1 to logic 0. Here, a push-button switch is used to simulate the occurrence of external events.

5.11.2 The Aim

The aim of this project is to show how a character LCD can be interfaced and used in an ESP32 DevKitC project and how numbers can be displayed on the LCD. An I2C compatible 16 character by 2 row LCD is used here, as in the previous project.

5.11.3 Block diagram

The block diagram of the project is shown in Figure 5.74 where the SDA and SCL pins of the I2C LCD are connected to the ESP32 DevKitC. In addition, a push-button switch is connected to DevKitC through a 10K pull-up resistor to simulate the occurrence of external events.

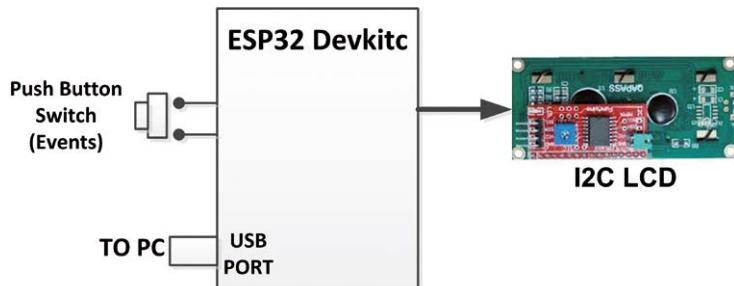


Figure 5.74 Block diagram of the project

5.11.4 Diagram

The circuit diagram of the project is shown in Figure 5.75. The SDA and CL pins of the I2C LCD are connected to GPIO ports 21 and 22 respectively. The push-button switch is connected to GPIO port 23.

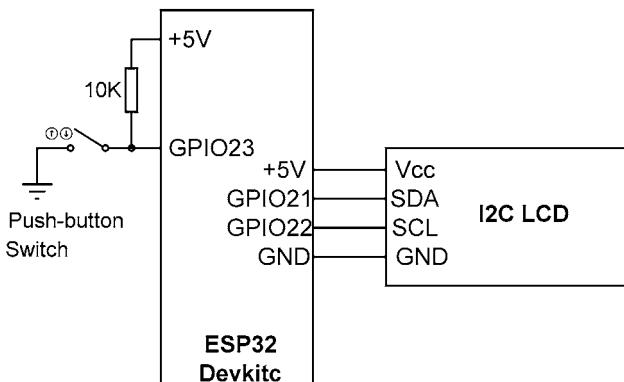


Figure 5.75 Circuit diagram of the project

5.11.5 Construction

The ESP32 DevKitC board is mounted on a breadboard with the push-button switch. The I2C LCD is connected to DevKitC through jumper wires as shown in Figure 5.76.

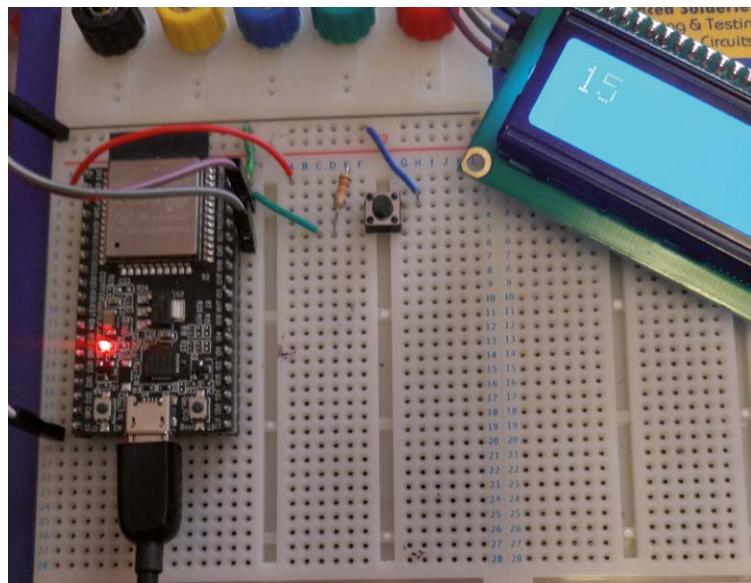


Figure 5.76 The project constructed on a breadboard

5.11.6 PDL of the Project

The PDL of the project is shown in Figure 5.77.

```

BEGIN
  Include libraries Wire and LiquidCrystal_I2C
  Assign GPIO port 23 to EVENTS
  Clear variable Count to 0
  Configure port 23 as input
  Set I2C LCD address and configuration
  Initialize I2C LCD
  Turn ON backlight
  Display text Event Counter
DO FOREVER
  Wait until an event occurs
  Contact debounce
  Clear LCD
  Increment Count
  Convert Count into string
  Display Count on LCD
ENDDO
END

```

Figure 5.77 PDL of the project

5.11.7 Program Listing

The program listing is shown in Figure 5.78 (program: **LCDCounter**).

```
/*****
 *          LCD EVENT COUNTER
 *          =====
 *
 * This is an event counter program with an LCD. Events are
 * assumed to occur on GPIO port 23 where an event is the HIGH
 * to LOW transition of this pin. In this project events are
 * simulated using a push-button switch connected to GPIO23 and
 * pulled high. An I2C LCD is connected to the ESP32 Devkitc.
 *
 * File:    LCDCounter
 * Date:   July 2017
 * Author: Dogan Ibrahim
 *****/
#define EVENTS 23
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
int Count = 0;
char Res[4];
// Set the LCD address to 0x27 and the configuration to
// 16 chars and 2 rows display. Also configure port 23 as
// an input
//
LiquidCrystal_I2C lcd(0x27, 16, 2);      // LCD address 0x27

void setup()
{
    pinMode(EVENTS, INPUT);
    lcd.begin();                      // Initialize LCD
    lcd.backlight();                  // Turn ON backlight
    lcd.print("Event Counter");       // Display Text
}

void loop()
{
    while(digitalRead(EVENTS) == 1);    // Wait until event
    delay(150);                       // Contact debounce
    lcd.clear();                      // Clear LCD
    Count++;                          // Increment Count
    sprintf(Res, 4, "%d", Count);     // Convert to string
    lcd.print(Res);                   // Display on LCD
}
```

Figure 5.78 Program listing

5.11.8 Program Description

At the beginning of the program **EVENTS** is assigned to GPIO port 23 and I2C and LCD libraries are included in the program. Variable **Count** is cleared to zero. In the setup routine port 23 is configured as input, LCD is initialised, and the text message Event Counter is displayed on the LCD.

The remainder of the program runs in an endless loop where the program waits until an event occurs. A small delay is then introduced to the program to wait until the switch contacts stop bouncing (called **contact debouncing**). The program then clears the LCD, increments **Count** by one, converts it into a string using function **snprintf** and displays the value of **Count** on the LCD.

Notice that we could have also used the command **lcd.print(Count)** to display the value of variable **Count** before converting it into a string.

5.12 PROJECT 11 – LCD Command

5.12.1 Description

In this project we shall be looking at the available LCD commands and display text in various formats.

5.12.2 The Aim

The aim of this project is to give a list of the available LCD commands and see how these commands can be used in programs.

5.12.3 Block diagram

The block diagram of the project is as in Figure 5.66.

5.12.4 Circuit Diagram

The circuit diagram of the project is as in Figure 5.67.

5.12.5 Construction

The ESP32 DevKitC board is mounted on a breadboard as in Figure 5.68.

5.12.6 LCD Commands

The I2C LCD library supports the following commands:

<code>clear()</code>	clear the LCD and position cursor to first position
<code>home()</code>	home the cursor to first position
<code>setCursor(x, y)</code>	set cursor to column x, row y (index starts from 0)
<code>print()</code>	display on LCD
<code>display()</code>	show characters on the display (default)
<code>noDisplay()</code>	do not show any characters on the display
<code>blink()</code>	start blinking the cursor
<code>noBlink()</code>	stop blinking the cursor
<code>cursor()</code>	display the cursor indicator

noCursor()	do not show the cursor indicator
scrollDisplayLeft()	scroll the display left by one character position
scrollDisplayRight()	scroll the display right by one character position
leftToRight()	text to flow to the right from the cursor, as if the display is left-justified (default)
rightToLeft()	text to flow to the left from the cursor, as if the display is right-justified.
noBacklight()	disable backlight
backlight()	enable backlight
getBacklight()	get backlight value
autoscroll()	moves all the text one space to the left each time a letter is added (default)
noAutoscroll()	disabled auto scrolling

In this project the following LCD display actions are performed:

Turn backlight ON
Clear LCD
Go to column 7, row 0
Display text "At 7,0"
Go to column 7, row 1
Display text "At 7,1"
Wait 1 second
Clear LCD
Disable cursor
Display text "No cursor"
Wait 1 second
Enable cursor
Set to scroll display right to left
Wait 1 second
Clear LCD
Set cursor to column 10, row 0
Display text "HELLO"
Set back to scroll display left to right
Set Count to 100
Clear LCD
Display Count

5.12.7 Program Listing

The program listing is shown in Figure 5.79 (program: **LCDCommands**).

```
*****  
*          LCD COMMANDS  
*          ======  
*  
* This program uses various LCD commands with the aim of making
```

```
* the programmer aware of the available commands.  
*  
* File: LCDCommands  
* Date: July 2017  
* Author: Dogan Ibrahim  
*****  
#include <Wire.h>  
#include <LiquidCrystal_I2C.h>  
int Count = 100;  
  
//  
// Set the LCD address to 0x27 and the configuration to  
// 16 chars and 2 rows display  
//  
LiquidCrystal_I2C lcd(0x27, 16, 2); // LCD address 0x27  
  
void setup()  
{  
    lcd.begin(); // Initialize LCD  
    lcd.backlight(); // Turn ON backlight  
    lcd.clear(); // Clear LCD  
    lcd.setCursor(7, 0); // column 7, row 0  
    lcd.print("At 7,0"); // Display "At 7,0"  
    lcd.setCursor(7, 1); // Column 7, row 1  
    lcd.print("At 7,1"); // Display "At 7,1"  
    delay(1000); // Wait 1 second  
    lcd.clear(); // Clear LCD  
    lcd.noCursor(); // Disable cursor  
    lcd.print("No cursor"); // Display "No cursor"  
    delay(1000); // Wait 1 second  
    lcd.cursor(); // Enable cursor  
    lcd.scrollDisplayRight(); // Scroll right  
    delay(1000); // Wait 1 second  
    lcd.clear();  
    lcd.setCursor(10,0);  
    lcd.rightToLeft(); // Right to left  
    lcd.print("Hello"); // Display "Hello"  
    lcd.leftToRight(); // Left to right  
    delay(1000); // Wait 1 second  
    Count = 100; // Count = 100  
    lcd.clear(); // Clear LCD  
    lcd.print(Count); // Display Count  
}  
  
void loop()
```

```
{  
    // No code  
}
```

Figure 5.79 Program listing

5.12.8 Program Description

Figures 5.80 to 5.84 show the various displays when program in Figure 5.79 is run.



Figure 5.80 LCD display 1



Figure 5.81 LCD display 2



Figure 5.82 LCD display 3



Figure 5.83 LCD display 4

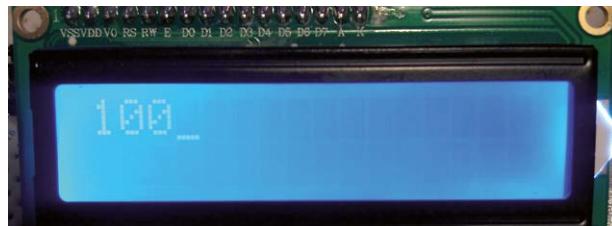


Figure 5.84 LCD display 5

5.13 PROJECT 12 – External Interrupts

5.13.1 Description

In this project an LED is connected to GPIO pin 23. Also a push-button switch is connected to GPIO pin 22. The LED normally flashes every 500 ms. Pressing the button generates an external interrupt. If the LED is flashing when the interrupt is generated then it stops flashing. If on the other hand the LED is already stopped, then it starts to flash when the button is pressed.

5.13.2 The Aim

The aim of this project is to show how external interrupts can be configured in ESP32 DevKitC when programmed using the Arduino IDE.

5.13.3 Block diagram

The block diagram of the project is shown in Figure 5.85.

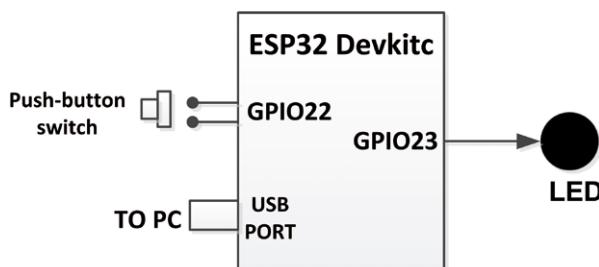


Figure 5.85 Block diagram of the project

5.13.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 5.86 where the LED is connected to GPIO port 23 through a 330 ohm resistor, and the push-button switch is connected to GPIO port 22. Notice that port 22 is pulled HIGH internally by the program.

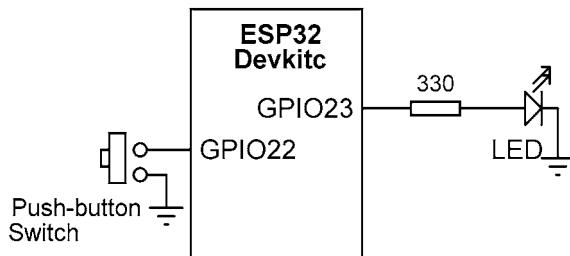


Figure 5.86 Circuit diagram of the project

5.13.5 Construction

The ESP32 DevKitC, the LED, and the push-button switch are mounted on a breadboard as in Figure 5.87.

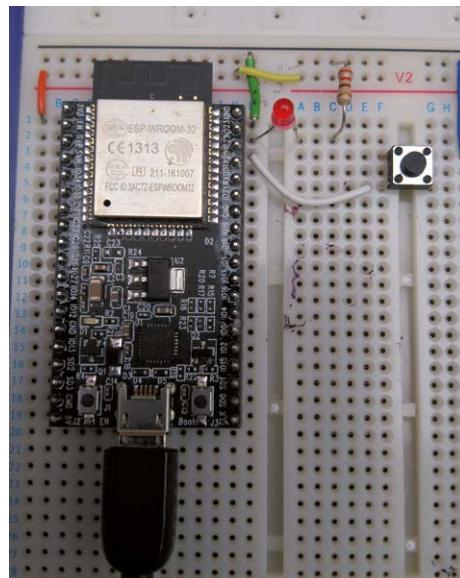


Figure 5.87 The project constructed on a breadboard

5.13.6 PDL of the Project

The PDL of the project is shown in Figure 5.88.

```

BEGIN
  Assign LED to GPIO port 23
  Assign IntPin to GPIO port 22
  Configure LED port as output
  Configure IntPin as input
  Flag = 1
DO FOREVER
  IF Flag = 1 THEN
    Flash the LED
  
```

```

ELSE
    Stop flashing
ENDIF
ENDDO
END

Interrupt Service Routine: LEDControl
BEGIN
    Toggle Flag
END

```

Figure 5.88 PDL of the project

5.13.7 Program Listing

The program listing is shown in Figure 5.89 (program: **ExtInt**).

```

/*****
*          EXTERNAL INTERRUPT
=====
*
* This is an example of using an external interrupt on the
* ESP32 DevKitC. In this example an LED is connected to GPIO
* port 23 through a 330 ohm resistor. In addition, a push-button
* switch is connected to GPIO pin 22 and this button is then
* configured as an interrupt pin such that pressing the button
* generates an external interrupt in the program. Normally the
* LED flashes every 500 ms. If the button is pressed when the
* LED is flashing then the flashing stops. If on the other hand
* the flashing is stopped, pressing the button will re-start
* the LED to flash.
*
*
* File:  ExtInt
* Date:  July 2017
* Author: Dogan Ibrahim
*****
#define LED 23                                // LED on GPIO 23
#define IntPin 22                               // Interrupt pin
int Flag = 1;                                 // LED flag

//
// Configure LED port as output, interrupt pin as input, and
// PULL-UP this pin so that its state is at normally logic HIGH.
// Also, Attach the interrupt pin IntPin to function LEDControl,
// and accept interrupts on the FALLING edge (logic HIGH to LOW)

```

```
// of the interrupt pin
//
void setup()
{
    pinMode(LED, OUTPUT);           // LED is output
    pinMode(IntPin, INPUT_PULLUP);   // Interrupt pin
    attachInterrupt(digitalPinToInterruption(IntPin),LEDControl,FALLING);
}

//
// Flash the LED every 500 ms. The flashing is stopped and
// re-started by the interrupt pin. When interrupt occurs
// by pressing the IntPin, the state of variable Flag is
// changed (if 0, it becomes 1; and if 1 it becomes 0)
//
void LEDControl()
{
    if(Flag == 0)
        Flag = 1;
    else Flag = 0;
}

//
// Flash the LED every second unless topped by the interrupt
//
void loop()
{
    if(Flag == 1)
    {
        digitalWrite(LED, LOW);          // LED OFF
        delay(500);                    // 500 ms delay
        digitalWrite(LED, HIGH);         // LED ON
        delay(500);                    // 500 ms delay
    }
    else
    {
        digitalWrite(LED, LOW);          // LED OFF
    }
}
```

Figure 5.89 Program listing

5.13.8 Program Description

At the beginning of the program **LED** is assigned to GPIO port 23 and the interrupt pin **Int-Pin** is assigned to GPIO pin 22. Inside the setup routine the LED is configured as output and the interrupt pin is configured as INPUT_PULLUP so that the pin is normally at logic HIGH.

Then, the external interrupt pin is attached to function **LEDControl**, which is the interrupt service routine in this example. Thus, when button **IntPin** is pressed an external interrupt will be generated and the program will automatically jump to function **LEDControl**. Inside the main program the LED flashes every 500 ms when variable **Flag** is 1. This variable is toggled inside function **LEDControl**. Thus, when the button is pressed variable **Flag** changes its value to 0 and the flashing stops. Pressing the button again will change **Flag** back to 1 and hence the flashing will start again.

Notice that four external interrupt modes are supported:

RISING: The interrupt is activated on the rising edge (LOW to HIGH) transition of the interrupt pin.

FALLING: The interrupt is activated on the FALLING edge (HIGH to LOW) of the interrupt pin.

CHANGE: The interrupt is activated when the state of the interrupt pin changes (HIGH to LOW or LOW to HIGH).

LOW: The interrupt is activated when the interrupt pin is LOW.

5.14 PROJECT 13 – Timer Interrupts

5.14.1 Description

In this project an LED is connected to GPIO pin 23 as in the previous project. The project uses a timer interrupt to flash the LED at every 500 ms.

5.14.2 The Aim

The aim of this project is to show how the timer interrupts of the ESP32 DevKitC can be programmed using the Arduino IDE.

5.14.3 Block diagram

The block diagram of the project is as in Figure 4.1.

5.14.4 Circuit Diagram

The circuit diagram of the project is as in Figure 4.3.

5.14.5 Construction

The ESP32 DevKitC and the LED are mounted on a breadboard as in Figure 4.4.

5.14.6 PDL of the Project

The PDL of the project is shown in Figure 5.90. The ESP32 processor contains two hardware timer groups, each containing two general-purpose hardware timers. All the timers are 64-bit generic timers based on 16-bit pre-scalers and have 64-bit auto-reload features, capable of counting up or down.

BEGIN

 Assign LED to GPIO port 23

 Configure LED port as output

```
Configure timer 0 with pre-scaler 80 and counting up  
Attach the timer interrupt to function LEDControl  
Set timer alarm to interrupt every 500 ms  
END
```

```
Timer interrupt service routine (LEDControl):  
BEGIN  
    Toggle the LED  
END
```

Figure 5.90 PDL of the project

5.14.7 Program Listing

The program listing is shown in Figure 5.91 (program: **TimerInt**).

```
/*****  
*          TIMER INTERRUPT  
*          ======  
*  
* This is an example of using a timer interrupt on the  
* ESP32 Devkitc. In this example an LED is connected to GPIO  
* port 23. The LED is flashed every 500 ms inside the interrupt  
* service routine.  
*  
*  
* File:  TimerInt  
* Date:  July 2017  
* Author: Dogan Ibrahim  
*****/  
#define LED 23                                // LED on GPIO 23  
int state = 0;  
  
hw_timer_t * timer = NULL;                      // Create a hardware timer  
  
//  
// Configure LED port as output. We are using timer 0 (1st timer)  
// in timerBegin with the prescaler set to 80 and counting mode  
// is up (set to true).  
//  
// The clock frequency is 80 MHz, thus, by using a pre-scaler  
// value of 80, we have 1 MHz, or the timer interrupt rate will  
// be every microsecond. The interrupt service routine is called  
// LEDControl. We are setting to call function LEDControl when  
// the timer expires and a timer interrupt is generated. Because  
// the edge mode is set to true in timerAttachInterrupt function,  
// interrupts will be accepted on edge of the clock.
```

```

// In timerAlarmWrite function the timer alarm value is set to
// 500 ms and auto repeat is set to be true so that the interrupt
// service routine (LEDControl) will be called every 500 ms (
// remember that 1 second is equal to 1000000 microseconds).
//
//
void setup()
{
    pinMode(LED, OUTPUT); // LED is output
    timer = timerBegin(0, 80, true); // Timer every microsecond
    timerAttachInterrupt(timer, &LEDControl, true);
    timerAlarmWrite(timer, 500000, true);
    timerAlarmEnable(timer);
}

//
// This is the timer interrupt service routine. Flash the LED
// every 500 ms
//
void LEDControl()
{
    if(state == 0)
    {
        state = 1;
        digitalWrite(LED, LOW); // LED OFF
    }
    else
    {
        state = 0;
        digitalWrite(LED, HIGH); // LED ON
    }
}

void loop()
{
    // no code
}

```

Figure 5.91 Program listing

5.14.8 Program Description

At the beginning of the program **LED** is assigned to GPIO port 23. Inside the setup routine, the LED is configured as output. Timer 0 is set with a pre-scaler value of 80 and in up-counting mode. Since the clock frequency is 80 MHz, with a pre-scaler value of 80 the

interrupt timing is one microsecond. Function **LEDControl** is attached to the timer interrupt so that whenever the timer overflows and generates an interrupt, the program will jump to function **LEDControl** automatically. Timer alarm is set to 500 ms (500000 microseconds) and in auto repeat mode so that the timer interrupt repeats.

Function **LEDControl** basically toggles the LED such that if it is OFF it is turned ON, and vice versa.

5.15 PROJECT 14 – Using the Touch Sensitive Inputs – Touch Based LED Control

5.15.1 Description

In this project an LED is connected to GPIO pin 23 of the ESP32 as in the previous project. In addition, a piece of wire is connected to one of the touch sensor inputs of the ESP32. The LED is turned ON when one touches the wire.

5.15.2 The Aim

The aim of this project is to show how a touch sensor input of the ESP32 can be used in a simple project.

5.15.3 Block diagram

The block diagram of the project is shown in Figure 5.92.

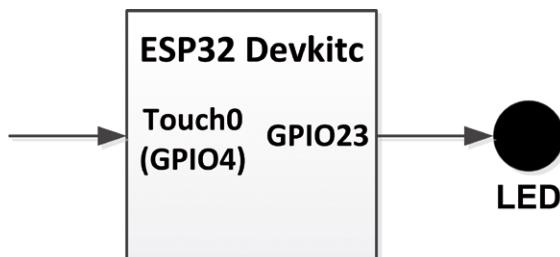


Figure 5.92 Block diagram of the project

5.15.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 5.93 with the LED connected to GPIO pin 23 through a current limiting resistor and a short piece of wire is connected to GPIO pin 4 of the ESP32 which is the touch sensitive input Touch0 (or T0).

The ESP32 processor has 10 capacitive touch sensitive input pins that respond when touched by hand. These pins can be used in projects in which external devices may be required to be controlled by touching the pins. (We could use buttons instead of the touch sensitive inputs, but buttons have limited lifetimes and also force has to be applied to activate them.) The following pins are the touch sensitive pins of the ESP32 DevKitC. Notice that Touch1 (T1) is not available as a touch sensitive input since this pin is pulled HIGH and the BOOT switch is connected to this pin:

Touch Sensitive Input	GPIO Pin
Touch0 (T0)	4
Touch1 (T1)	0
Touch2 (T2)	2
Touch3 (T3)	15
Touch4 (T4)	13
Touch5 (T5)	12
Touch6 (T6)	14
Touch7 (T7)	27
Touch8 (T8)	32
Touch9 (T9)	33

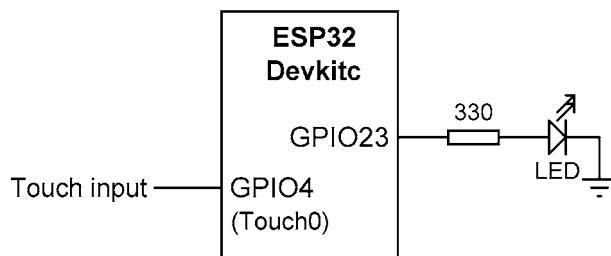


Figure 5.93 Circuit diagram of the project

5.15.5 Construction

The ESP32 DevKitC and the LED are mounted on a breadboard as shown in Figure 5.94. A short piece of wire is connected to touch sensitive input GPIO pin 4 (Touch0 or T0).

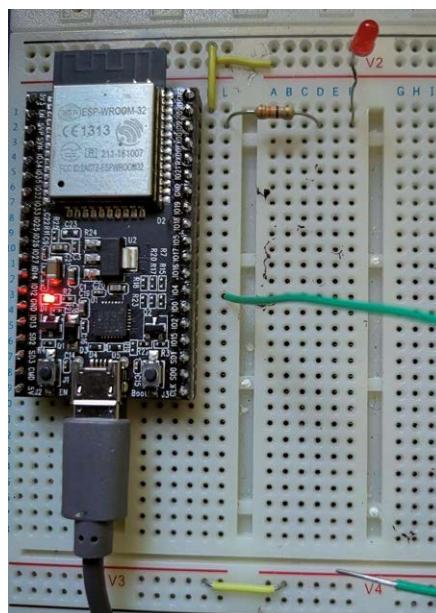


Figure 5.94 The project constructed on a breadboard

5.15.6 PDL of the Project

The PDL of the project is shown in Figure 5.95.

```
BEGIN
    Configure GPIO port 23 as output
    Turn OFF LED
    DO FOREVER
        IF touch sensor input T0 is touched THEN
            Turn ON LED
        ELSE
            Turn OFF LED
        ENDIF
    ENDDO
END
```

Figure 5.95 PDL of the project

5.15.7 Program Description

The **touchRead** function is used to read the value returned from a touch sensitive input. This function returns an analog value which varies with the input capacitance. The capacitance and hence the returned value changes depending on the touching capacitance, which in turn depends on the length and type of the cable connected to the touch sensitive pin. It is best initially to display the returned values on for example a PC screen (using the **Putty** terminal emulator program) as the input pin is touched. The main program can then be written based on the returned values.

The following program code will display the returned values as the piece of wire connected to the touch sensitive input is touched lightly with fingers, and then when the wire is squeezed between the fingers.

```
#define TouchPin T0
int TouchValue;

void setup()
{
    Serial.begin(9600);
}

void loop()
{
    TouchValue = touchRead(T0);
    Serial.println(TouchValue);
    delay(1000);
}
```

The values displayed on the PC screen are shown in Figure 5.96. The returned values were as follows:

Condition	Returned Value
Away from the input wire	54
Fingers close to the input wire (not touching)	53
Touching the input wire lightly with fingers	10 - 16
Squeezing the input wire between fingers	9

```
54
54
54
54
54
54
54
54
54
54
54
54
54
54
53
53
16
13
13
10
10
9
9
9
9
9
9
54
```

Figure 5.96 Values returned by function `touchRead`

In this project we shall assume that a value lower than 50 indicates that the wire is touched. The program listing (program: **TouchLED**) of the project is shown in Figure 5.97. At the beginning of the program, **TouchPin** is defined as touch sensitive input **T0**, integer variable **TouchValue** is created, and **LED** is assigned to GPIO port 23. Inside the **setup** routine the LED is configured as an OUTPUT and it is turned OFF. Inside the main program loop the value returned by the touch sensitive input **T0** is read into variable **TouchValue**. If the returned value is less than 50 it is assumed that the wire is touched and the LED is turned ON, otherwise the LED is turned OFF.

```
*****
*          TOUCH SENSOR LED CONTROL
*          =====
*
* In this program an LED is connected to the ESP32 DevKitC
* GPIO port 23. Additionally, a piece of wire is connected
* to touch sensitive input GPIO 4 (TOUCH0, or T0). The LED
* turns ON when we touch the wire.
*
* Program: TouchLED
* Date   : October, 2018
*****
```

```
#define TouchPin T0
int TouchValue;

#define ON HIGH
#define OFF LOW
#define LED 23           // LED on port GPIO23

void setup()
{
    pinMode(LED, OUTPUT);      // Configure as OUTPUT
    digitalWrite(LED, OFF);    // LED OFF at beginning
}

void loop()
{
    TouchValue = touchRead(T0); // Read touch sensitive input
    if(TouchValue < 50)        // If touched...
        digitalWrite(LED, ON); // Turn ON LED
    else                      // Otherwise...
        digitalWrite(LED, OFF); // Turn OFF LED
}
```

Figure 5.97 Program listing

5.15.8 Using Touch Interrupts

The program given in Figure 5.97 can be modified to use touch interrupts. Function **touchAttachInterrupt** can be called to generate an interrupt when we touch a touch sensitive input pin. The format of this function is as follows:

```
touchAttachInterrupt(Tn, ISR, threshold);
```

Tn, here, is the touch sensitive input name (T0 to T9), **ISR** is the function to be called when the pin is touched to, and **threshold** defines the sensitivity of the touch input. Normally the threshold is set to 40, and higher values give greater sensitivities. In the following example code, touch sensitive input **T2** is used with sensitivity set to **40**, where function **MyTouch** will be called when we touch input **T2**:

```
touchAttachInterrupt(T2, MyTouch, 40);
.....
.....
void MyTouch()
{
    Serial.println("Touched to input T2");
}
```

5.16 PROJECT 15 – Using the Touch Sensitive Inputs – Changing the LED Brightness

5.16.1 Description

In this project an LED is connected to GPIO pin 23 and a piece of wire is connected to touch sensitive input T0 of the ESP32 as in the previous project. A PWM waveform is sent to the LED whose duty cycle changes as we touch (or squeeze) the wire. As a result, the brightness of the LED changes accordingly.

5.16.2 The Aim

The aim of this project is to show how a touch sensor input can be used to change the duty cycle of a PWM waveform and hence the brightness of an LED.

5.16.3 Block diagram

The block diagram of the project is as in Figure 5.92.

5.16.4 Circuit Diagram

The circuit diagram of the project is as in Figure 5.93.

5.16.5 PDL of the Project

The PDL of the project is shown in Figure 5.98.

```

BEGIN
    Configure GPIO port 23 as output
    Configure PWM channel 0
DO FOREVER
    Read touch sensitive input value
    Change this value to PWM duty cycle (between 0 and 255)
    Send the PWM waveform to the LED
ENDDO
END

```

Figure 5.98 PDL of the project

5.16.6 Program Description

In this project a PWM waveform is sent to GPIO port 23, to which the LED is connected. The duty cycle of this waveform is changed (the duty cycle is an integer number between 0 and 255) as the touch sensitive wire is touched or squeezed between the fingers.

The program listing (program: **TouchLED2**) is shown in Figure 5.99. At the beginning of the program the PWM parameters for channel 0 are defined and the PWM channel is configured inside the setup routine. The resolution of the PWM is set to 8 and its frequency to 1000 Hz. Inside the main program loop the program reads the value returned by function **touchRead**. As was shown in the previous project, the returned value is around 54 when not touching the wire, and it goes down to 9 when we squeeze the wire. The value returned by function **touchRead** is multiplied by 4 and taken away from 200 and the resulting

value is used to set the duty cycle of the PWM waveform. Therefore, when the wire is not touched, the duty cycle is around $255 - 5*54$, which is less than zero and the LED is turned OFF. When we touch the wire the duty cycle becomes around $255 - 5*12 = 195$ which turns the LED ON. By squeezing the wire the duty cycle increases, thus making the LED brighter. Different multiplier values can be used for different touch sensitivities.

```
/*****
 *          TOUCH SENSOR LED BRIGHTNESS CONTROL
 *          =====
 *
 * In this program an LED is connected to the ESP32 Devkitc
 * GPIO port 23. Additionally, a piece of wire is connected
 * to touch sensitive input GPIO 4 (TOUCH0, or T0). The LED
 * brightness is changed by touching or squeezing the wire.
 *
 * Program: TouchLED2
 * Date   : October, 2018
 *****/
#define TouchPin T0
int TouchValue;

// 
// PWM parameters
//
int pwmDuty;
int pwmresolution = 8;
int pwmchannel = 0;
int pwmfreq = 1000;

#define ON HIGH
#define OFF LOW
#define LED 23           // LED on port GPIO23

//
// Configure LED as output, configure the PWM channel 0
// Attach to the LED port
//
void setup()
{
    pinMode(LED, OUTPUT);
    ledcSetup(pwmchannel, pwmfreq, pwmresolution);
    ledcAttachPin(LED, pwmchannel);
}

void loop()
```

```

{
    TouchValue = touchRead(T0);           // Get touch value
    pwmDuty = 255 - 5*TouchValue;
    if(pwmDuty < 0)pwmDuty = 0;          // If < 0
    ledcWrite(pwmchannel, pwmDuty);      // Change duty cycle
}

```

Figure 5.99 Program listing

5.17 PROJECT 16 – Using Multiple Touch Sensitive Inputs – Electronic Organ

5.17.1 Description

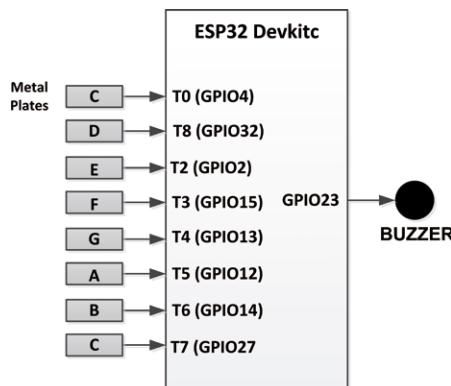
In this project 8 touch sensitive inputs of the ESP32 processor are used to make an electronic organ for one octave. The musical notes are generated by touching small metal plates connected to the touch sensitive inputs via wires. A buzzer is used to generate the required musical notes. PWM waveforms are sent to the buzzer at the correct musical note frequencies.

5.17.2 The Aim

The aim of this project is to show how multiple touch sensitive inputs can be used in a project.

5.17.3 Block diagram

The block diagram of the project is shown in Figure 5.100.

*Figure 5.100 Block diagram of the project.*

5.17.4 Circuit Diagram

The circuit diagram of the project is as in Figure 5.101. A small buzzer is connected to GPIO pin 23 of the DevKitC development board. The following touch sensitive pins are used to represent the musical tones:

Touch Sensitive Input	GPIO Pin	Musical Note
Touch0 (T0)	4	C
Touch8 (T8)	32	D
Touch2 (T2)	2	E
Touch3 (T3)	15	F
Touch4 (T4)	13	G
Touch5 (T5)	12	A
Touch6 (T6)	14	B
Touch7 (T7)	27	C

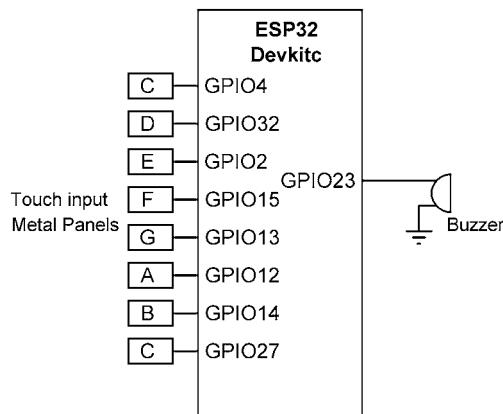


Figure 5.101 Circuit diagram of the project

5.17.5 PDL of the Project

The PDL of the project is shown in Figure 5.102.

```

BEGIN
  Store musical note frequencies in an array
  Configure PWM channel 0 duty cycle to 50%
  Configure Buzzer as output
DO FOREVER
  IF touch sensitive input T0 is touched to THEN
    Generate musical note C
  ELSE IF touch sensitive input T8 is touched to THEN
    Generate musical note D
  ELSE IF touch sensitive input T2 is touched to THEN
    Generate musical note E
  ELSE IF touch sensitive input T3 is touched to THEN
    Generate musical note F
  ELSE IF touch sensitive input T4 is touched to THEN
    Generate musical note G
  ELSE IF touch sensitive input T5 is touched to THEN
    Generate musical note A
  ELSE IF touch sensitive input T6 is touched to THEN

```

```

        Generate musical note B
ELSE IF touch sensitive input T7 is touched to THEN
        Generate musical note C
ELSE
        Generate tone with 0 frequency (not audible)
ENDIF
ENDO
END

```

Figure 5.102 PDL of the project

5.17.6 Program Description

The frequencies of the musical notes for one octave are as follows. In this program integer approximations of these frequencies are used:

Notes	C	D	E	F	G	A	B	C
Hz	261.63	293.66	329.63	349.23	392	440	493.88	523.26

In this project, the PWM waveform is generated at GPIO port 23, to which the buzzer is connected. The duty cycle of this waveform is set to 50%, but its frequency is changed depending on which touch sensitive metal panel the user touches.

Figure 5.103 shows the program listing (program: **TouchOrgan**). At the beginning of the program the frequencies of the musical notes for one octave are stored in an array called **Notes**. Then, the PWM parameters are defined where the duty cycle is set to 50%. Inside the setup routine the Buzzer is configured as an output and the PWM waveform is directed to the port that the Buzzer is connected to. The main program runs in an endless loop. Inside this loop the 8 touch sensitive inputs are checked to see if any of these inputs are touched to and if so the frequency of the PWM waveform is changed to the requested musical note frequency. If none of the inputs are touched to then the PWM frequency is set to 0 which silences the Buzzer. Notice here that the **touchRead** threshold point is taken as 30 so that the inputs are not very sensitive.

```

/*****
*          TOUCH SENSOR ELECTRONIC ORGAN
*          =====
*
* In this program a Buzzer is connected to the ESP32 DevKitC
* GPIO port 23. Additionally, 8 small metal panels are
* connected to 8 touch sensitive inputs using wires. The musical
* notes for one octave can be played by touching the metal plates
*
* Program: TouchOrgan
* Date   : October, 2018
*****/

```

```
//  
// Assign musical notes to touch sensitive pins  
//  
#define C1 0  
#define D 1  
#define E 2  
#define F 3  
#define G 4  
#define A 5  
#define B 6  
#define C2 7  
int Notes[] = {262,294,330,349,392,440,494,523};  
int TouchValue;  
  
//  
// PWM parameters  
//  
int pwmDuty = 127; // 50% duty cycle  
int pwmresolution = 8; // 8 bit resolution  
int pwmchannel = 0; // PWM channel 0  
int pwmfreq = 1000; // intila frequency  
  
#define ON HIGH  
#define OFF LOW  
#define BUZZER 23 // BUZZER on port GPIO23  
  
//  
// Configure BUZZER as output, configure the PWM channel 0  
// Attach to the BUZZER port, set the duty cycle to 50%  
//  
void setup()  
{  
    pinMode(BUZZER, OUTPUT);  
    ledcSetup(pwmchannel, pwmfreq, pwmresolution);  
    ledcWrite(pwmchannel, pwmDuty);  
    ledcAttachPin(BUZZER, pwmchannel);  
}  
  
//  
// Check the touch sensitive inputs and generate a musical note  
// depending on the touch sensitive input pin touched to  
//  
void loop()  
{  
    if(touchRead(T0) < 30)
```

```

        ledcWriteTone(pwmchannel, Notes[C1]);
    else if(touchRead(T8) < 30)
        ledcWriteTone(pwmchannel, Notes[D]);
    else if(touchRead(T2) < 30)
        ledcWriteTone(pwmchannel, Notes[E]);
    else if(touchRead(T3) < 30)
        ledcWriteTone(pwmchannel, Notes[F]);
    else if(touchRead(T4) < 30)
        ledcWriteTone(pwmchannel, Notes[G]);
    else if(touchRead(T5) < 30)
        ledcWriteTone(pwmchannel, Notes[A]);
    else if(touchRead(T6) < 30)
        ledcWriteTone(pwmchannel, Notes[B]);
    else if(touchRead(T7) < 30)
        ledcWriteTone(pwmchannel, Notes[C2]);
    else
        ledcWriteTone(pwmchannel, 0);
}

```

Figure 5.103 Program listing

5.17.7 Suggestions For Additional Work

Modify the project hardware by replacing the buzzer with a small speaker and use an audio amplifier to amplify the sound level.

5.18 PROJECT 17 – Using the SPI Bus – Digital to Analog Converter (DAC)

5.18.1 Description

The SPI bus is one of the commonly used protocols to connect sensors and many other devices to microcontrollers. The SPI bus is a master-slave-type bus protocol. In this protocol, one device (the microcontroller) is designated as the master, and one or more other devices (usually sensors) are designated as slaves. In a minimum bus configuration there is one master and only one slave. The master establishes communication with the slaves and controls all activity on the bus.

Figure 5.104 shows an SPI bus example with one master and three slaves. The SPI bus uses three signals: clock (SCK), data in (SDI), and data out (SDO). SDO of the master is connected to the SDIs of the slaves, and SDOs of the slaves are connected to the SDI of the master. The master generates the SCK signals to enable data to be transferred on the bus. In every clock pulse one bit of data is moved from master to slave, or from slave to master. The communication is only between a master and a slave, and the slaves cannot communicate with each other. It is important to note that only one slave can be active at a time since there is no mechanism to identify the slaves. Thus, slave devices have enable lines (e.g. CS) which are normally controlled by the master. A typical communication between a master and several slaves is as follows:

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

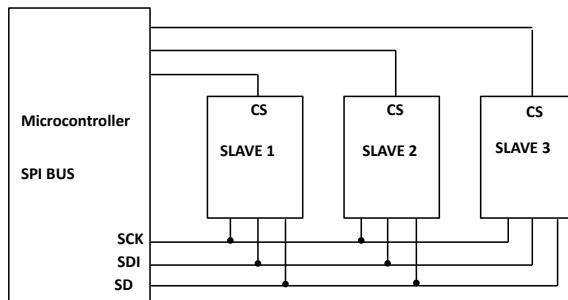


Figure 5.104 SPI bus with one master and 3 slaves

The SPI signal names are also called MISO (Master in, Slave out), and MOSI (Master out, Slave in). Clock signal SCK is also called SCLK and the CS is also called SSEL.

Before using the SPI interface we have to define the following SPI parameters: speed, data order, and data mode. Users should refer to the device data sheets before setting these parameters. These parameters are defined in the function called **SPISettings**, in the following order:

```
SPISettings(speed, data order, data mode)
```

The speed is the data transfer speed in Hz and it depends on the peripheral SPI device used. The data order defines whether the MSB bit or the LSB bit will be sent out first. By default the MSB bit is sent out first. Valid values are **MSBFIRST** or **LSBFIRST**. Data mode defines the clock phase and polarity. There are four ways that the SPI clock can be sampled: Mode0, Mode1, Mode2, and Mode3. Assuming that CPOL is the clock polarity and CPHA is the clock phase the modes can be explained as follows (see Figure 5.105):

Mode0: This is the default mode. Here the clock is normally LOW (CPOL = 0) and data is sampled when the clock goes from LOW to HIGH (i.e. on leading edge of the clock) where CPHA = 0.

Mode1: In this mode the clock is LOW (CPOL = 0) and data is sampled when the clock goes from HIGH to LOW (i.e. on trailing edge of the clock) where CPHA = 1.

Mode2: In this mode the clock is normally HIGH (CPOL = 1) and data is sampled when the clock goes from HIGH to LOW (i.e. on the leading edge of the clock) where CPHA = 0.

Mode3: Here, the clock is normally HIGH (CPOL = 1) and data is sampled when the clock goes from LOW to HIGH (i.e. on trailing edge of the clock) where CPHA = 1.

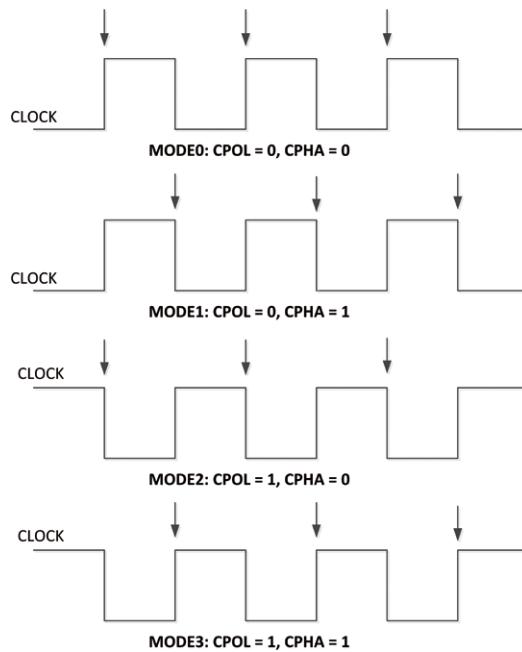


Figure 5.105 SPI clock phase and polarity

In this project an SPI bus based DAC chip is connected to the ESP32 DevKitC development board. The project generates square wave output with a voltage of 1 V, duty cycle of 50%, and frequency of 1 kHz (period of 1 ms).

5.18.2 The Aim

The aim of this project is to show how the SPI bus can be used in a project.

5.18.3 Block diagram

The block diagram of the project is shown in Figure 5.106.

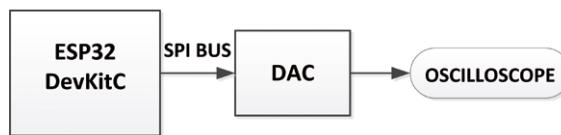


Figure 5.106 Block diagram of the project.

5.18.4 Circuit Diagram

In this project the MCP4921 type SPI bus compatible DAC chip is used. MCP4921 is a 12-bit serial DAC manufactured by Microchip Inc., with the following basic specifications:

- 12-bit resolution
- Up to 20 MHz clock rate (SPI)
- Fast settling time of 4.5 μ s
- Unity or 2x gain output
- External Vref input
- 2.7 V to 5.5 V operation
- Extended temperature range (-40 °C to +125 °C)
- 8-pin DIL package

Figure 5.107 shows the pin layout of the MCP4921. The pin definitions are:

VDD, AVSS:	power supply and ground
CS:	chip select (LOW to enable the chip)
SCK,SDI:	SPI bus clock and data in
VOUTA:	analog output
VREFA:	reference input voltage
LDAC:	DAC input latch (transfers the input data to the DAC registers. Normally tied to ground so that CS controls the data transfer).

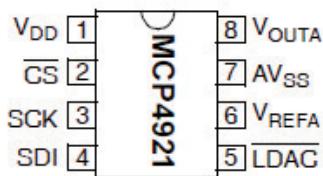


Figure 5.107 Pin layout of MCP4921 DAC

The SPI port on the ESP32 DevKitC has one SPI bus port with the following GPIO pins:

GPIO Pin	SPI Bus Signal
GPIO23	SPI MOSI
GPIO19	SPI MISO
GPIO18	SPI SCK
GPIO5	SPI SS

Figure 5.108 shows the circuit diagram of the project. The SDI, SCK and CS inputs of the MCP4921 are connected to GPIO pins SPI MOSI, SPI SCK, and GPIO22. The reference voltage Vref and the supply voltage Vdd are connected to a +3.3 V pin of DevKitC. Pins LDAC and Avss are connected to ground. The output of the DAC is connected to a PC based oscilloscope in order to record the generated waveform.

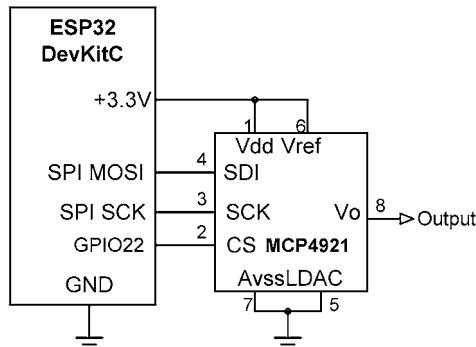


Figure 5.108 Circuit diagram of the project

5.18.5 The Construction

The project was constructed on a breadboard as shown in Figure 5.109 and connections were made to DevKitC using jumper wires.

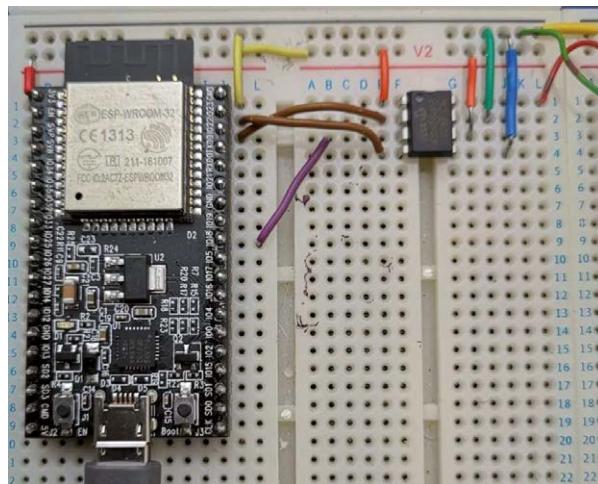


Figure 5.109 Project constructed on a breadboard

5.18.6 The PDL

The PDL of the project is shown in Figure 5.110.

BEGIN/MAIN

Configure the SPI bus to MSB first and Mode 0

Set CS as output

Disable CS

Configure SPI

DO FOREVER

Call SendData to send 1V to DAC

Wait 0.5ms

Call SendData to send 0V to DAC

```
    Wait 0.5ms
ENDDO
END/MAIN

BEGIN/SendData
    Enable CS
    Extract the HIGH byte
    Send HIGH byte to DAC with Gain = 1
    Extract LOW byte
    Send LOW byte to DAC
    Disable CS
END/SendData
```

Figure 5.110 PDL of the project

5.18.7 Program Listing

The program listing is shown in Figure 5.111 (program: **SPI**).

```
*****  
*          SPI BUS DIGITAL TO ANALOG CONVERTER  
*          ======  
  
* This program uses the SPI bus DAC chip MCP4921. This is a 12-bit  
* DAC. The program outputs a square waveform with an amplitude of 1V  
* and a frequency of 1kHz (period 1ms) through the DAC.  
  
*  
* Program: SPI  
* Date   : October, 2018  
* Author : Dogan Ibrahim  
*****  
  
#include "SPI.h"  
#define CS 22  
#define ENABLE LOW  
#define DISABLE HIGH  
  
SPISettings MySettings(2000000, MSBFIRST, SPI_MODE0);  
  
void setup()  
{  
    pinMode(CS, OUTPUT);           // CS is output  
    digitalWrite(CS, DISABLE);     // Disable CS  
    SPI.begin();                  // Begin SPI  
    SPI.beginTransaction(MySettings); // Configure SPI  
}
```

```

//  

// Send Data to DAC through the SPI bus  

//  

void SendData(unsigned int MyData)  

{  

    unsigned char xdata;  
  

    digitalWrite(CS, ENABLE);           // Enable CS  

    xdata = highByte(MyData);          // Get HIGH byte  

    xdata = 0b00001111 & xdata;         // Extract D8-D11  

    xdata = 0b00110000 | xdata;         // Set G,SHDN  

    SPI.transfer(xdata);              // Send to DAC  
  

    xdata = lowByte(MyData);           // Get LOW byte  

    SPI.transfer(xdata);              // Send to DAC  

    digitalWrite(CS, DISABLE);         // Disable CS  

}  
  

//  

// Main program loop. Call function SendData to send data to the  

// DAC to generate a square waveform with frequency of 1kHz and  

// duty cycle of 50%  

//  

void loop()  

{  

    float V;  

    unsigned int OutData;  
  

    V = 4095.0 / 3.3;                // 1V output  

    OutData = (unsigned int)V;         // Convert to int  

    SendData(OutData);               // Send 1 for 0.5ms  

    delayMicroseconds(500);           // Wait 0.5ms  

    OutData = 0;                      // 0V output  

    SendData(OutData);               // Send 0 for 0.5ms  

    delayMicroseconds(500);           // Wait 0.5ms  

    //SPI.endTransaction();           // Not needed here  

}

```

Figure 5.111 Program listing

5.18.8 Program Description

Before writing the program it is worthwhile to briefly look at the operation of the MCP4921. MCP4921 is a 12-bit DAC with the output voltage given by:

$$V_o = G \times (V_{ref} \times D_n) / 2^n$$

Where G is the output gain (can be 1 or 2), Dn is the DAC input code, and n is 12. The gain is normally set to 1 and therefore, with a +3.3 V reference voltage, the output voltage is given by:

$$V_o = (3.3 \times D_n) / 4096$$

For example, if the input code is **0000 0000 0001** then the output voltage will be 0.805 mV. Similarly, if the input code is **0000 0000 0010** then the output voltage will be 1.61 mV. The output data of MCP4921 is presented in two bytes. As shown in Figure 5.112, the lower byte is the data bits D0 to D7. Four lower nibble of the upper byte is the data bits D7 to D11. The bits in the higher nibble of the higher byte are used to configure the device as follows:

A/B: cleared to 0 to write to the DAC.

BUF: this is the input buffer (0: data is unbuffered, 1: data is buffered). In most applications the input data is set to be unbuffered.

G: This bit controls the output gain (0: Gain is 2, 1: Gain is 1).

SHDN: This is the shutdown bit (1: normal operation, 0: shutdown the device).

In normal operations the upper nibble of the higher byte is set to **0011**.

HIGH BYTE								LOW BYTE							
A/ B	BU	G	SHD N	D1 1	D1 0	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

Figure 5.112 Output bytes of MCP4921

The output pin of the MCP4921 was connected to a PC based oscilloscope and the waveform is shown in Figure 5.113. It is clear from this figure that the frequency and amplitude of the generated square waveform are 1 kHz and 1 V respectively.

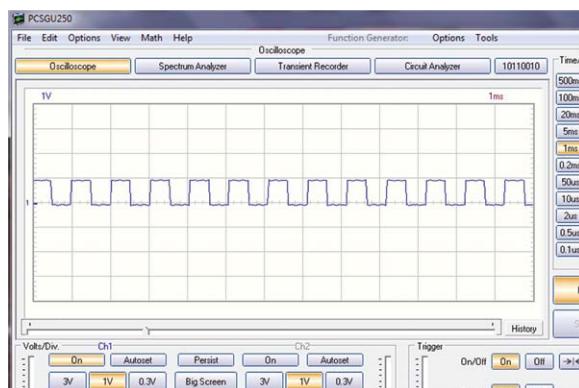


Figure 5.113 Output waveform on oscilloscope

Notice that the MCP4921 operates with the MSB bit received/sent first and in mode SPI_MODE0. The clock frequency is set to 2000000 Hz in this project.

5.19 PROJECT 18 – Using the UARTs

5.19.1 Description

UART has traditionally been used for many years to establish serial communication between two or more devices. In this project a random number is generated every 3 seconds between 1 and 255 and is sent to a UART port. Another UART port reads the generated number and stores it in a variable. This number is then displayed on the PC screen using the serial USB interface of the ESP32 DevKitC development kit.

5.19.2 The Aim

The aim of this project is to show how multiple UARTs can be used in an ESP32 DevKitC project.

5.19.3 Block diagram

The block diagram of the project is shown in Figure 5.114. Two UART ports named as Ser1 and Ser2 and the USB serial port are used in the project.

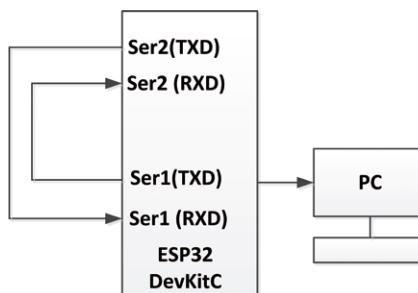


Figure 5.114 Block diagram of the project.

5.19.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 5.115. In UART based communications two pins are normally used where the TXD and RXD of one device are connected to the RXD and TXD of the other device respectively. In addition to the TXD and RXD pins, other pins such as the RTS (request to send) and CTS (clear to send) are available that can be used for synchronisation purposes. In most applications however it is enough just to use the TXD and RXD pins. Both communicating devices must be configured to have the same following parameters set:

- Baud rate (usually 9600 or 115200)
- Number of data bits (usually 8)
- Parity (usually NONE)
- Stop bits (usually 1)

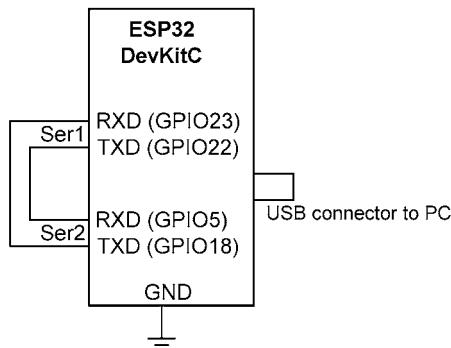


Figure 5.115 Circuit diagram of the project

The ESP32 DevKitC have 3 UART pins as follows:

RXD (U0)	GPIO3
TXD (U0)	GPIO1
RXD (U1)	GPIO9
TXD (U1)	GPIO10
RXD (U2)	GPIO16
TXD (U2)	GPIO17

U0 is used by the USB to PC serial communication port. In some devices U2 is reserved for the on-board flash and may not be available for serial communication. In general most other pins can be configured for UART communication. In this project the following pins are used as UART ports:

GPIO pin	Name	Configuration
GPIO23	Ser1	RXD
GPIO22	Ser1	TXD
GPIO5	Ser2	RXD
GPIO18	Ser2	TXD

The UART consisting of pins GPIO23 and GPIO22 is named as **Ser1** and the UART consisting of pins GPIO5 and GPIO18 is named as **Ser2**. The TXD and RXD pins of **Ser1** are connected to RXD and TXD pins of **Ser2** respectively.

5.19.5 The PDL

The PDL of the project is shown in Figure 5.116.

```

BEGIN
    Define the two UART pins
    Configure the two UART ports
DO FOREVER
    Generate a random number between 1 and 255
    Send the number to UART port Ser1
  
```

```

IF data is available at UART port Ser2 THEN
    Read the data
    Convert data into ASCII
    Display the data on PC screen
ENDIF
ENDDO
END

```

Figure 5.116 PDL of the project

5.19.6 Program Listing

The program listing is shown in Figure 5.117 (program: **UART**).

```

/*****
*           UART COMMUNICATIONS
* =====
*
* This program demonstrates how multiple UARTs on the ESP32 DevKitC
* development board can be used. A random number is generated between
* 1 and 255 every 3 seconds. This number is sent to a UART port where
* another UART port reads the number and stores in a variable. The
* number is then displayed on the PC screen.
*
* Program: UART
* Date   : October, 2018
* Author : Dogan Ibrahim
*/
#include "HardwareSerial.h"

#define RXD1 23                      // First UART RX
#define TXD1 22                      // First UART TX
#define RXD2 5                       // Second UART RX
#define TXD2 18                      // Second UART TX

HardwareSerial Ser1(1);
HardwareSerial Ser2(2);

void setup()
{
    Serial.begin(9600);             // USB port
    Ser1.begin(9600, SERIAL_8N1, RXD1,TXD1); // First UART
    Ser2.begin(9600, SERIAL_8N1, RXD2,TXD2); // Second UART
}

//
// This is the main program loop. Generate a random number and send
// it to a UART port. Read the same number from another UART port.

```

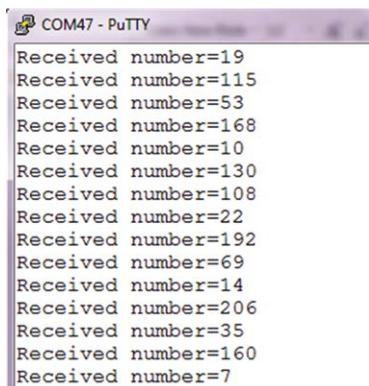
```
// Finally, display the number on the PC screen
//
void loop()
{
    char buffer[10];
    unsigned int RandomNumber = random(0, 255);      // Generate number
    Ser1.write(RandomNumber);                         // Write to UART
    while(Ser2.available() > 0)                      // If data available
    {
        unsigned int dat = Ser2.read();                // Read data
        itoa(dat, buffer, 10);                        // Convert to ASCII
        Serial.print("Received number=");
        Serial.println(buffer);                       // Display number
        delay(3000);                                 // 3 seconds delay
    }
}
```

Figure 5.117 Program listing

5.19.7 Program Description

At the beginning of the program the two UART pins used in the project are defined. The setup routine configures the UARTs to operate at 9600 baud. Inside the main program loop a random number between 1 and 255 is generated. This number is sent to UART **Ser1**. The status of UART **Ser2** is then checked to see if any data is available and if so the data is converted into ASCII by calling function **itoa** and is stored in character array called **buffer**. This data is then displayed on the PC screen. The main program loop is repeated after 3 seconds delay.

Figure 5.118 shows an example display from of the program.

A screenshot of the Putty terminal window titled "COM47 - PuTTY". The window displays a series of text lines representing the output of the program. Each line starts with the prefix "Received number=" followed by a three-digit decimal number. The numbers listed are: 19, 115, 53, 168, 10, 130, 108, 22, 192, 69, 14, 206, 35, 160, and 7. The text is black on a white background with a standard monospaced font.

```
Received number=19
Received number=115
Received number=53
Received number=168
Received number=10
Received number=130
Received number=108
Received number=22
Received number=192
Received number=69
Received number=14
Received number=206
Received number=35
Received number=160
Received number=7
```

Figure 5.118 Example display on PC screen

5.20 PROJECT 19 – Writing Data to the Flash Memory

5.20.1 Description

Data can be saved in the non-volatile flash memory of the ESP32 processor so that the data remains there even after reset or when power is removed. (Notice that ESP32 has no EEPROM memory but we can save data in the non-volatile flash memory as if it is an EEPROM.) This is especially useful to save data such as configuration data, username, password, last value of a variable (e.g. the highest or the lowest temperature) and so on. The Arduino IDE EEPROM library is used to read and write the flash memory of ESP32. Up to 512 bytes of flash memory can be used by this library. This means that up to 512 characters can be stored permanently in the flash memory.

In this project an analog temperature sensor chip is connected to the ESP32 DevKitC. The program runs for 60 seconds and reads the ambient temperature every second. The maximum temperature during this period is stored in the flash memory in the first two bytes. Only integer temperature is considered in this project where the data is stored in ASCII format. i.e. a two digit temperature value (e.g. 25) is stored as two digit ASCII numeric data (e.g. as "2" and "5").

5.20.2 The Aim

The aim of this project is to show how data can be written to the ESP32 DevKitC flash memory.

5.20.3 Block diagram

The block diagram of the project is as in Figure 5.1, in section 5.2.

5.20.4 Circuit Diagram

The circuit diagram of the project is as in Figure 5.3. The analog temperature sensor TMP36 is connected to GPIO4 of DevKitC.

5.20.5 The PDL

The PDL of the project is shown in Figure 5.119.

```

BEGIN
  Define EEPROM size
  Define TMP36 port number
  Initialize the EEPROM
  DO 60 times
    Read temperature
    Convert into degrees centigrade
    Calculate the maximum temperature so far
    Wait one second
  ENDDO
  Save the maximum temperature in the first 2 locations of EEPROM
  Stop
END

```

Figure 5.119 PDL of the project

5.20.6 Program Listing

The program listing is shown in Figure 5.120 (program: **FLASH**).

```
*****  
*          WRITE TO FLASH MEMORY  
*          =====  
  
* In this program the ambient temperature is read using an analog  
* temperature sensor chip. The temperature is read every second  
* for 60 seconds and the maximum temperature is stored in the  
* flash memory in the first two bytes of the memory in integer  
* format as two digits  
  
* Program: FLASH  
* Date   : October, 2018  
* Author : Dogan Ibrahim  
*****/  
  
#include "EEPROM.h"  
#define EEPROM_SIZE 2           // No of bytes to access  
#define TMP36 4                // TMP36 on GPIO4  
  
void setup()  
{  
    EEPROM.begin(EEPROM_SIZE);  
}  
  
void loop()  
{  
    char buffer[5];           // Buffer to store temp  
    int T, Temp, MaxT = 0;  
    float mV, Temperature;  
  
    for(int k = 0; k < 60; k++) // Do for 60 seconds  
    {  
        Temp = analogRead(TMP36); // Read temperature  
        mV = Temp * 3300.0 / 4096.0; // in mV  
        Temperature = (mV - 500.0) / 10.0; // Temperature  
        T = (int)Temperature; // In integer  
        if(T > MaxT)MaxT = T; // Max is MaxT  
        delay(1000); // Wait 1 second  
    }  
  
    itoa(MaxT, buffer, 10); // Convert to ASCII  
    EEPROM.write(0, buffer[0]); // Write in location 0  
    EEPROM.write(1, buffer[1]); // Write in location 1
```

```

    EEPROM.commit();           // Save to flash
    while(1);                // End of program
}

```

Figure 5.120 Program listing

5.20.7 Program Description

The following EEPROM functions are available:

EEPROM.write(addr, data)	- write data byte to EEPROM location addr;
EEPROM.read(addr)	- read byte from EEPROM location addr;
EEPROM.put(addr, data)	- write any data byte to EEPROM location addr;
EEPROM.get(addr, data)	- read any data from EEPROM location addr;
EEPROM.update(addr, data)	- write to EEPROM if the new data differs from existing data;
EEPROM.commit()	- save the written data.

Normally data is written or read as a byte, having a value of 0 to 255. Using the functions **EEPROM.put** and **EEPROM.get** we can write any type of data, such as integer, floating point, structure, etc. Notice that the program returns 255 if we try to read from an EEPROM location that data was not saved to before.

At the beginning of the program the EEPROM header file **EEPROM.h** is included in the program and the number of bytes to be accessed is defined in **EEPROM_SIZE** as two bytes. Inside the setup routine the EEPROM is initialised. The remainder of the program runs in a loop which is iterated every second and runs for 60 seconds. Inside this loop the temperature is read from TMP36 as an analog value and is converted into degrees Celsius and stored in variable **Temperature**. This reading is then converted into integer and is stored in a variable called **T**. The program then checks if this reading is greater than the existing maximum reading and if so makes this reading to be the maximum temperature reading so far. At the end of the loop the reading is converted into ASCII format and is stored in character array called **buffer**, where **buffer[0]** and **buffer[1]** hold the MSD and LSD of the maximum readings respectively. These two values are then written to the first two locations of the flash memory and they are saved by calling function **EEPROM.commit**. The program then terminates by waiting in a loop.

5.21 PROJECT 20 – Reading Data from the Flash Memory

5.21.1 Description

In this project the maximum temperature value stored in the flash memory in the previous project is read and displayed on the PC screen.

5.21.2 The Aim

The aim of this project is to show how data can be read from the ESP32 DevKitC flash memory.

5.21.3 Block diagram

The block diagram of the project is as in Figure 5.1, in section 5.2.

5.21.4 Circuit Diagram

The circuit diagram of the project is as in Figure 5.3. Analog temperature sensor TMP36 is connected to GPIO4 of DevKitC.

5.21.5 The PDL

The PDL of the project is shown in Figure 5.121.

```
BEGIN
    Define No of bytes to be read
    Initialize EEPROM
    Read MSD digit
    Read LSD digit
    Display heading "Maximum Temperature = "
    Display MSD and LSD
    Stop
END
```

Figure 5.121 PDL of the project

5.21.6 Program Listing

The program listing is shown in Figure 5.122 (program: **FlashRead**).

```
*****
*          READ FROM FLASH MEMORY
*          =====
*
* In the previous program the ambient temperature was stored as
* an integer number in the first two locations of the flash memory.
* This program reads this data and displays the maximum temperature
* on the PC screen.
*
* Program: FlashRead
* Date   : October, 2018
* Author : Dogan Ibrahim
*****
#include "EEPROM.h"
#define EEPROM_SIZE 2                                // No of bytes to access

void setup()
{
    Serial.begin(9600);
    EEPROM.begin(EEPROM_SIZE);
}
```

```

void loop()
{
    char MSD, LSD;

    MSD = EEPROM.read(0);           // Read MSD
    LSD = EEPROM.read(1);           // Read LSD
    Serial.println("");             // New line
    Serial.print("Maximum Temperature = "); // Display heading
    Serial.print(MSD);              // Display MSD
    Serial.println(LSD);            // Display LSD

    while(1);                     // Stop
}

```

Figure 5.122 Program listing

5.21.7 Program Description

In this program function **EEPROM.read** is used to read the MSD (address 0) and LSD (address 1) data from the flash memory. The data read is displayed on the PC screen in the following format:

```
Maximum Temperature = nn
```

Figure 5.123 shows an example display from the program.

Figure 5.123 Example display on PC screen

5.22 PROJECT 21 – Writing Floating Point Data to the Flash Memory

5.22.1 Description

This project is similar to Project 19, but here the maximum temperature is stored as floating point data in the flash memory.

5.22.2 The Aim

The aim of this project is to show how floating point data can be written to the ESP32 DevKitC flash memory.

5.22.3 Block diagram

The block diagram of the project is as in Figure 5.1, in section 5.2.

5.22.4 Circuit Diagram

The circuit diagram of the project is as in Figure 5.3.

5.22.5 Program Listing

The program listing is shown in Figure 5.124 (program: **FLASH2**).

```
*****  
*          WRITE FLOATING POINT DATA TO FLASH MEMORY  
*          ======  
*  
* In this program the maximum temperature is stored as a floating  
* point number in the flash memory  
*  
* Program: FLASH2  
* Date   : October, 2018  
* Author : Dogan Ibrahim  
*****/  
  
#include "EEPROM.h"  
  
#define EEPROM_SIZE 4           // No of bytes to access  
#define TMP36 4                // TMP36 on GPIO4  
  
void setup()  
{  
    EEPROM.begin(EEPROM_SIZE);  
}  
  
void loop()  
{  
    int Temp;  
    float mV, CTemp, MaxT = 0.0;  
  
    for(int k = 0; k < 60; k++)           // Do for 60 seconds  
    {  
        Temp = analogRead(TMP36);         // Read temperature  
        mV = Temp * 3300.0 / 4096.0;       // in mV  
        CTemp = (mV - 500.0) / 10.0;        // Temperature  
        if(CTemp > MaxT)MaxT = CTemp;      // Max is MaxT  
        delay(1000);                      // Wait 1 second  
    }  
  
    EEPROM.put(0, CTemp);                // Write in location 0  
    EEPROM.commit();                   // Save to flash  
    while(1);                          // End of program  
}
```

Figure 5.124 Program listing

5.22.6 Program Description

In this program the maximum temperature is read as a floating point number and is stored in the variable **CTemp**. This value is then saved in the flash memory using the function **EEPROM.put**.

5.23 PROJECT 22 – Reading Floating Point Data from the Flash Memory

5.23.1 Description

This project is similar to Project 20, but here the temperature is read as floating point data from the flash memory.

5.23.2 The Aim

The aim of this project is to show how floating point data can be read from the ESP32 DevKitC flash memory.

5.23.3 Block diagram

The block diagram of the project is as in Figure 5.1, in section 5.2.

5.23.4 Circuit Diagram

The circuit diagram of the project is as in Figure 5.3.

5.23.5 Program Listing

The program listing is shown in Figure 5.125 (program: **ReadFlash2**).

```
/*
 *          READ FLOATING POINT DATA FROM FLASH MEMORY
 *=====
 *
 * In this program the maximum temperature is read as a floating
 * point number from the flash memory
 *
 * Program: ReadFlash2
 * Date   : October, 2018
 * Author : Dogan Ibrahim
 */
#include "EEPROM.h"
#define EEPROM_SIZE 4           // No of bytes to access

void setup()
{
    Serial.begin(9600);
    EEPROM.begin(EEPROM_SIZE);
}

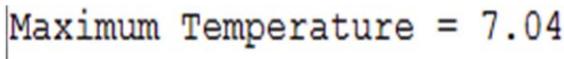
void loop()
```

```
{  
    float Temp;  
  
    EEPROM.get(0, Temp);           // Read temperature  
    Serial.println("");           // New line  
    Serial.print("Maximum Temperature = "); // Display heading  
    Serial.println(Temp);         // Display temperature  
    while(1);                   // End of program  
}
```

Figure 5.125 Program listing

5.23.6 Program Description

In this program the temperature is read from the flash memory and stored in variable **Temp** as a floating point number using function **EEPROM.get**. The temperature is then displayed on the PC screen. A typical display is shown in Figure 5.126.



Maximum Temperature = 7.04

Figure 5.126 A typical display

5.24 Summary

In this chapter we have developed a number of simple projects using the ESP32 DevKitC. In the next chapter we shall be looking at the development of intermediate level projects which are more complex than the projects we have developed so far.

Chapter 6 • Intermediate projects using the Arduino IDE and the ESP32 DevKitC

6.1 Overview

In the last chapter we have seen the development of some simple projects using the ESP32 DevKitC. In this chapter we shall be developing more complex projects using the Arduino IDE as the development software.

As with the previous projects, the title, description, aim, block diagram and so on of all the projects will be given.

6.2 PROJECT 1 – ON-OFF Temperature Control

6.2.1 Description

This is an ON-OFF temperature control project. A temperature sensor measures the temperature of the place (e.g. a room) where the temperature is to be controlled and compares it with a set temperature. A heater is connected to the ESP32 DevKitC through a relay. If the measured temperature is below the set temperature then the heater is turned ON. If, on the other hand, the measured temperature is above the set temperature then the heater is turned OFF. An I2C LCD is used to show both the set temperature and the measured temperature. In addition, a push-button switch (SetPoint) is connected to the DevKitC and the required temperature is set using this push-button switch and the LCD. Another push-button switch (START) is used to start the control algorithm. The temperature can be set between 15 °C and 30 °C. Every time the push-button switch is pressed the set point temperature is increased by 1 °C and this is shown on the LCD.

6.2.2 The Aim

The aim of this project is to show how an ON-OFF temperature control system can be designed using the ESP32 DevKitC.

6.2.3 Block diagram:

Figure 6.1 shows the block diagram of the project. The temperature is measured using the DHT11 sensor chip which can measure both the temperature and the humidity (see Project 5.3). The temperature control range is between 15 °C and 30 °C. The push-button switch SetPoint is used to set the required temperature so that every time the button is pressed the set temperature goes up by 1 °C (returns back to 15 °C after 30 °C). The LCD helps to see the settings while the SetPoint button is pressed. Another button called START starts the control algorithm. The heater is turned ON and OFF by the relay. The LCD shows both the measured and the required temperature values every second. In addition the status of the relay is shown as either ON or OFF.

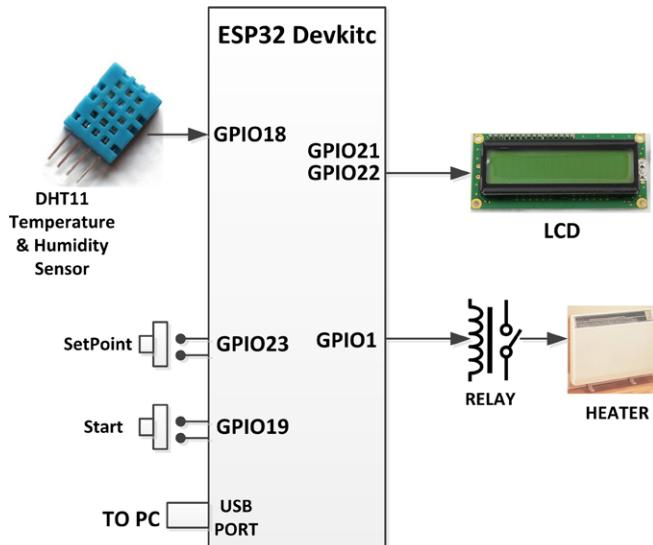


Figure 6.1 Block diagram of the project

6.2.4 Circuit Diagram

The circuit diagram of the system is shown in Figure 6.2. The push-button switch SetPoint is connected to GPIO port 23. The relay is connected to GPIO port 1. The SDA and SCL pins of the I2C LCD are connected to GPIO ports 21 and 22 respectively. The DHT11 temperature sensor chip is connected to GPIO port 18. The START button is connected to GPIO port 19. In this project the Relay from their Elektor Sensor Kit is used (see Figure 6.3). The relay has 3 pins: + (power supply), S (control input), and - (power supply ground). Also, the 3-pin DHT11 sensor from the same kit is used (see Figure 6.4) to measure the temperature. The DHT11 also has 3 pins: + (power supply), S (data output), and - (power supply ground).

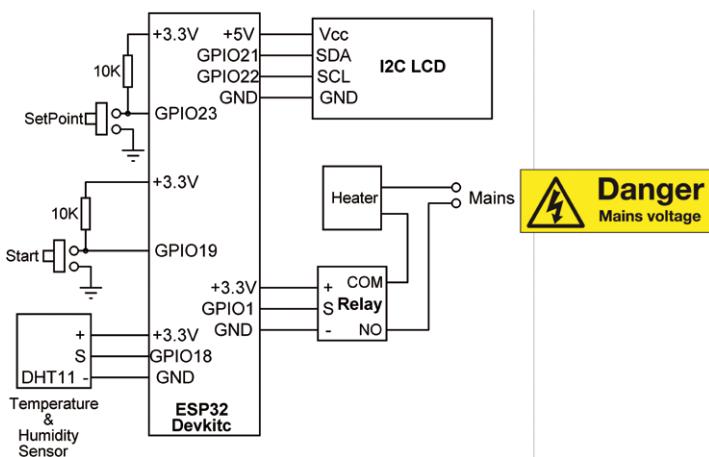


Figure 6.2 Circuit diagram of the project



Figure 6.3 The relay

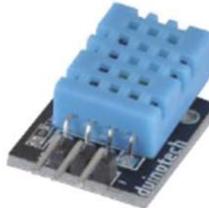


Figure 6.4 The DHT11

6.2.5 Construction

The ESP32 DevKitC board is mounted on a breadboard together with the push-button switches and the DHT11 temperature sensor chip as shown in Figure 6.5. The I2C LCD is connected to the DevKitC using jumper wires.

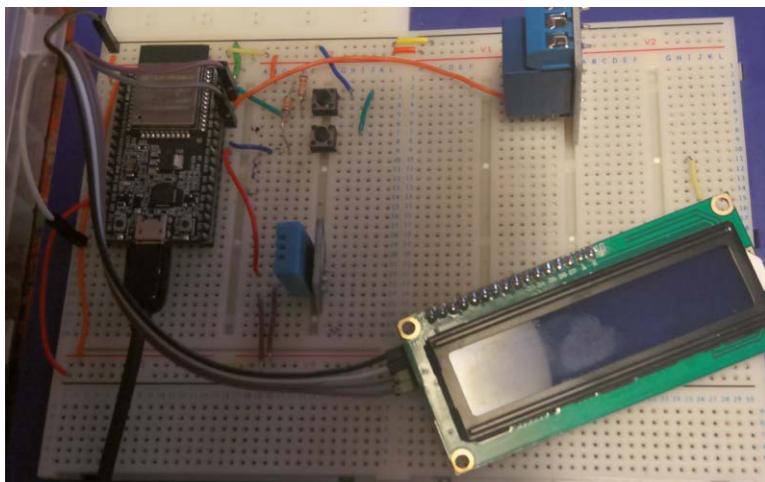


Figure 6.5 Project on a breadboard

6.2.6 PDL of the Project

The PDL of the project is shown in Figure 6.6. As described earlier, if the measured temperature is below the set temperature then the heater is turned ON. If, on the other hand, the measured temperature is above the set temperature, then the heater is turned OFF. The button SetPoint is used to set the required temperature, while the button START starts the control algorithm.

```
BEGIN
    Assign SetPoint to GPIO port 23
    Assign RELAY to GPIO port 1
    Assign START to GPIO port 19
    Include I2C and DHT11 libraries
    Configure SetPoint and START as input ports
    Configure RELAY as output port
    Turn OFF the relay to start with
    DO FOREVER
        Read the required temperature setting
        Read the temperature from DHT11
        Display both the measured and the required temperatures
        IF required temperature > measured temperature THEN
            Turn ON relay
            Display text ON
        ELSE
            Turn OFF relay
            Display OFF
        ENDIF
        Wait 1 second
    ENDDO
END
```

Figure 6.6 PDL of the project

6.2.7 Program Listing

The program listing of the project is shown in Figure 6.7 (program: **ONOFF**).

```
*****
*          ON-OFF TEMPERATURE CONTROL
*=====
*
* This is an ON-OFF temperature control project. A temperature
* sensor measures the temperature of the place (e.g. a room)
* where the temperature is to be controlled and compares it
* with a set temperature. A heater is connected to the ESP32
* Devkitc through a relay. If the measured temperature is below
* the set temperature then the heater is turned ON. If on the
* other hand the measured temperature is above the set point
* temperature then the heater is turned OFF. An I2C LCD is used
* to show both the set temperature and the measured temperature.
* In addition, a push-button switch is connected to the Devkitc
* and the required temperature is set using this push-button
* switch and the LCD. The temperature can be set between 15°C
* and 30°C. Every time the push-button switch is pressed the set
* point temperature is increased by one Degree Centigrade and
* this is shown on the LCD. The temperature control process
```

```
* starts when another push-button switch is pressed.  
*  
* File:    ONOFF  
* Date:    July 2017  
* Author:  Dogan Ibrahim  
*****  
#define SetPoint 23                                // SetPoint button  
#define RELAY 1                                     // RELAY  
#define START 19                                    // START button  
  
//  
// I2C LCD libraries  
//  
#include <Wire.h>  
#include <LiquidCrystal_I2C.h>  
//  
// DHT11 libraries  
//  
#include <Adafruit_Sensor.h>  
#include "DHT.h"  
#define DHT11_PIN 18  
#define DHTTYPE DHT11  
//  
// Temperature variables  
//  
int MinTemperature = 15;                            // Min setting  
int MaxTemperature = 30;                            // Max setting  
int RequiredTemperature;  
int First = 1;  
  
DHT dht(DHT11_PIN, DHTTYPE);  
  
//  
// Set the LCD address to 0x27 and the configuration to  
// 16 chars and 2 rows display. Also, initialize the LCD  
// and the DHT11 sensor chip, turn the RELAY OFF to start  
// with and clear the LCD  
//  
LiquidCrystal_I2C lcd(0x27, 16, 2);                // LCD address 0x27  
  
void setup()  
{  
    pinMode(RELAY, OUTPUT);                         // Configure output  
    pinMode(SetPoint, INPUT);                       // Configure input  
    pinMode(START, INPUT);                          // Configure input  
    digitalWrite(RELAY, LOW);                      // RELAY OFF to start with  
    dht.begin();                                  // Initialize DHT11
```

```
lcd.begin();                                // Initialize LCD
lcd.backlight();                            // Turn ON backlight
}

// This function sets the required temperature. Pressing the
// SetPoint button increases the required temperature value by
// 1. When the required setting is reached, you should press
// the START button to exit from this function. The required
// temperature can be between 15 and 30 degrees Centigrade
//
int SetTemperature()
{
    int ExitFlag = 0;
    lcd.setCursor(0, 0);                      // Go to col 0, row 0
    lcd.print("Req. Temperature");           // Display "Req. Temperature"
    lcd.setCursor(0, 1);                      // Go to col 0, row 1
    RequiredTemperature = MinTemperature;
    lcd.print(RequiredTemperature);          // Display req temp

    while(ExitFlag == 0)                     // Do until START pressed
    {
        if(digitalRead(SetPoint) == 0)
        {
            RequiredTemperature++;
            if(RequiredTemperature > MaxTemperature)
            {
                RequiredTemperature = MinTemperature;
            }
            lcd.setCursor(0, 1);
            lcd.print(RequiredTemperature);
            delay(150);
        }
        else if(digitalRead(START) == 0)
        {
            ExitFlag = 1;
        }
    }
    lcd.clear();
    return RequiredTemperature;
}

// Main program. At the beginning the required temperature is read
// from the user. The program then implements the ON-OFF temp
// algorithm to control the temperature. Both the measured and
```

```

// the required temperature values are displayed on the LCD. If
// the required temperature is greater than the measured one
// then the relay is turned ON, otherwise the relay is turned
// OFF. Texts ON or OFF are also displayed at the second row of
// the LCD to show whether or not the relay is ON or OFF
//
void loop()
{
    if(First == 1)                                // If First time
    {
        RequiredTemperature = SetTemperature();
        First = 0;
    }
    int Measured = dht.readTemperature();           // Read the temperature
    lcd.setCursor(0,0);                            // Go to col 0, row 0
    lcd.print("Measured=");                        // Display "Measured"
    lcd.print(Measured);
    lcd.print(char(223));                          // Display degree sign
    lcd.setCursor(0,1);                            // Go to col 0, row 1
    lcd.print("Required=");                        // Display "Required"
    lcd.print(RequiredTemperature);
    lcd.print(char(223));                          // Display degree sign
    if(RequiredTemperature > Measured)
    {
        digitalWrite(RELAY, HIGH);                  // Turn RELAY ON
        lcd.print(" ON ");                         // Display "ON"
    }
    else
    {
        digitalWrite(RELAY, LOW);                  // Turn RELAY OFF
        lcd.print(" OFF");                        // Display "OFF"
    }
    delay(1000);                                 // Wait 1 second
}

```

Figure 6.7 Program listing

6.2.8 Program Description

At the beginning of the program, SetPoint, RELAY, and START are assigned to GPIO ports 23, 1 and 19, respectively. Then the I2C and the DHT11 libraries are included in the program. The required temperature range is set between **MinTemperature** and **MaxTemperature**, which are initialised to 15 and 30 respectively.

Inside the setup routine, RELAY is configured as an output, and SetPoint and START buttons are configured as inputs. The relay is turned OFF at the beginning of the program, the DHT11 and I2C LCD libraries are initialised and the LCD is cleared.

The remaining program runs in an endless loop. At the beginning of the loop, function **SetTemperature** is called once to set the required temperature. Here, variable **RequiredTemperature** is initialised to **MinTemperature** and the state of button SetPoint is checked, and every time the button is pressed variable RequiredTemperature is incremented by 1 (the limits are 15 to 30). The function exits and the control algorithm starts when the START button is pressed (see Figure 6.8).



Figure 6.8 Setting the required temperature

Inside the main program the temperature is read from the DHT11 chip and both the measured temperature and the required temperature are displayed. If the required temperature is greater than the measured temperature then the relay is turned ON (and hence the heater connected to the relay) and the text ON is displayed at the second row of the LCD (see Figure 6.9). If, on the other hand, the required temperature is less than the measured temperature, then the relay is turned OFF and the text OFF is displayed at the second row of the LCD. The above process is repeated forever after one second delay.



Figure 6.9 The relay is ON

Notice: You must be careful and be aware of the possibility of electric shock if you intend to use mains electricity in this project. You should seek professional advice before using the mains power in any electronic circuit.

6.3 PROJECT 2 – Generating Waveforms – Sawtooth Waveform

6.3.1 Description

This project shows how a Sawtooth waveform can be generated using the ESP32 DevKitC.

6.3.2 The Aim

The aim of this project is to show how the DAC on the ESP32 DevKitC can be used to generate a Sawtooth waveform.

6.3.3 Block diagram:

Figure 6.10 shows the block diagram of the project. Here, the ESP32 processor generates the required Sawtooth waveform as a digital signal, and then, the DAC outputs this signal through the GPIO port reserved for the DAC.

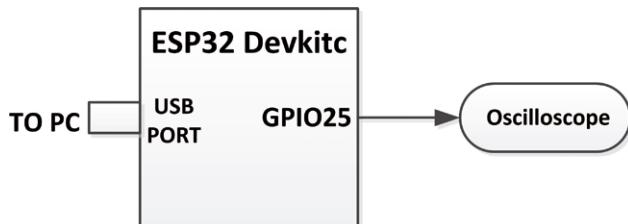


Figure 6.10 Block diagram of the project

Basically, two methods are used for waveform generation:

- The processor calculates the waveform points in real-time and sends them to the DAC.
- The waveform points are stored in a look-up table. The processor reads these points from the table and sends them to the DAC. (This method is used to generate any arbitrary waveform, or to generate higher frequency waveforms.) The rate at which the waveform points are sent to the DAC determines the frequency of the waveform.

6.3.4 The DAC

A DAC converts digital signals to analog. Although a DAC can be an external component, most microcontrollers have built-in DACs. For example, the ESP32 processor has two 8-bit DACs. In external DACs, the digital input can either be in serial or in parallel form. In external parallel converters, the width of the digital input is equal to the width of the converter. For example, 10-bit DACs have 10 input bits. External serial DACs generally use the SPI or the I2C bus, and a clock and a data line are used to send the data to be converted to the DAC. Parallel converters provide much faster conversion times, but they are housed in larger packages.

DACs are manufactured as either unipolar or bipolar as far as the output voltages are concerned. Unipolar converters can only output positive voltages, whereas bipolar converters can output both positive and negative voltages.

The relationship between the digital input-output and the voltage reference of a DAC is given by:

$$V_o = \frac{DV_{ref}}{2^n}$$

V_o , here, is the output voltage, V_{ref} is the reference voltage, and n is the width of the converter. For example, in an 8-bit converter with a +3.3 V reference voltage,

$$V_o = \frac{3.3D}{2^8} = 12.89D \text{ mV}$$

Thus, for example, if the input digital value is 00000001, the analog output voltage will be 12.89 mV, if the input digital value is 00000010, the analog output voltage will be 25.78 mV, and so on.

In this project we will be generating a Sawtooth waveform with the following specifications:
Output voltage: 0 to +3.3 V

Frequency: 100 Hz (period = 10 ms)
Step size: 0.1 ms

6.3.5 Circuit Diagram

The circuit diagram of the system is shown in Figure 6.11. The ESP32 processor has two built-in 8-bit DACs on GPIO ports 25 and 26. In this project we shall be using the one at GPIO port 25.

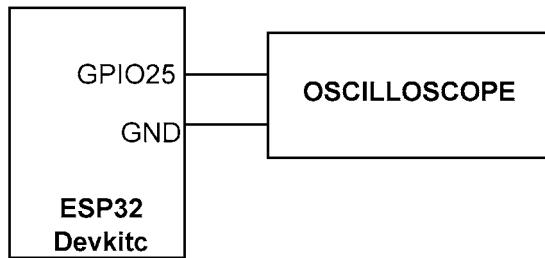


Figure 6.11 Circuit diagram of the project]

6.3.6 Construction

The ESP32 DevKitC board is mounted on a breadboard. The analog output voltage was displayed and captured using the PC based oscilloscope PSCGU250.

6.3.7 PDL of the Project

The PDL of the project is shown in Figure 6.12. Since there are 11 steps (0 to 10) in the waveform and the required frequency is 100 Hz, i.e. period of 10 ms, the duration of each step should be $10,000 / 11 = 909 \mu\text{s}$. The delayMicroseconds function can be used to generate the required delay in the program.

```
BEGIN
    Assign Sawtooth to GPIO port 25
    Configure port 25 as analog
DO FOREVER
    Send a step to the DAC
    Wait 909 microseconds
ENDDO
END
```

Figure 6.12 PDL of the project

6.3.8 Program Listing

The program listing of the project is shown in Figure 6.13 (program: **Sawtooth**).

```
/*
 *          SAWTOOTH WAVEFORM GENERATION
 *
 * In this project a Sawtooth waveform is generated through the
 * DAC at GPIO port 25 of the ESP32 Devkitc.
 *
 * The specifications of the generated waveform are:
 * Output voltage: 0 to +3.3V
 * Frequency: 100 Hz (period 10 ms)
 * Step size: 0.1 ms
 *
 * File:    Sawtooth
 * Date:    July 2017
 * Author:  Dogan Ibrahim
 */
#define Sawtooth 25
int DAC_Value;

void setup()
{
    pinMode(Sawtooth, ANALOG);           // Configure as analog
}
```

```

void loop()
{
    for(float i = 0; i <= 1; i=i+0.1)
    {
        DAC_Value = i * 255;
        dacWrite(Sawtooth, DAC_Value);           // Send to DAC
        delayMicroseconds(909);                  // 909 us delay
    }
}

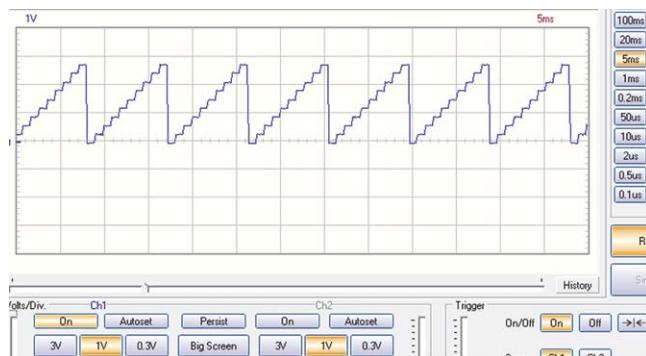
```

Figure 6.13 Program listing

6.3.9 Program Description

At the beginning of the program Sawtooth is assigned to GPIO port 25 and it is configured as an analog port. Inside the main program 11 steps are continuously sent to the DAC with 909 μ s delay between each step. Function dacWrite is used to send the steps to the DAC. This function has two arguments: DAC port number, and the digital value to be converted into analog.

Figure 6.14 shows the generated Sawtooth waveform, captured on the PSCGU250 PC based oscilloscope.

*Figure 6.14 Generated Sawtooth waveform*

6.4 PROJECT 3 – Generating Waveforms – Triangle Waveform

6.4.1 Description

This project shows how a Triangle waveform can be generated using the ESP32 DevKitC.

6.4.2 The Aim

The aim of this project is to show how the DAC on the ESP32 DevKitC can be used to generate a Triangle waveform.

6.4.3 Block diagram:

The block diagram is same as in Figure 6.10.

In this project we will be generating a Triangle waveform with the following specifications:
 Output voltage: 0 to +3.3 V

Frequency: 100 Hz (period = 10 ms)
 Step size: 0.1 ms

6.4.4 Circuit Diagram

The circuit diagram of the system is same as in Figure 6.11.

6.4.5 PDL of the Project

The PDL of the project is shown in Figure 6.15. Since the required period is 10 ms, the rising and falling parts of the waveform will each be 454 μ s. Again, the delayMicroseconds function will be used to generate the required delay in the program.

```
BEGIN
  Assign Triangle to GPIO port 25
  Configure port 25 as analog
DO FOREVER
  Send a step to the DAC
  Wait 454 microseconds
ENDDO
END
```

Figure 6.15 PDL of the project

6.4.6 Program Listing

The program listing of the project is shown in Figure 6.16 (program Triangle).

```
/*********************************************
*
*          TRIANGLE WAVEFORM GENERATION
*
*=====
*
* In this project a Triangle waveform is generated through the
* DAC at GPIO port 25 of the ESP32 Devkitc.
*
* The specifications of the generated waveform are:
* Output voltage: 0 to +3.3V
* Frequency: 100 Hz (period 10 ms)
* Step size: 0.1 ms
*
* File:    Triangle
* Date:    July 2017
* Author:  Dogan Ibrahim
*/
#define Triangle 25
int DAC_Value;
```

```

float Sample = 0.0, Inc = 0.1;

void setup()
{
    pinMode(Triangle, ANALOG);           // Configure as analog
}

void loop()
{
    DAC_Value = Sample * 255;
    dacWrite(Triangle, DAC_Value);        // Send to DAC
    Sample = Sample + Inc;
    if(Sample > 1.0 || Sample < 0.0)
    {
        Inc = -Inc;
        Sample = Sample + Inc;
    }
    delayMicroseconds(454);               // 909 us delay
}

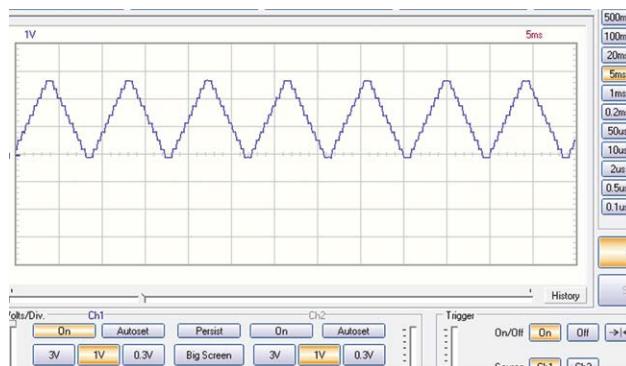
```

Figure 6.16 Program listing

6.4.7 Program Description

At the beginning of the program, Triangle is assigned to GPIO port 25 and it is configured as an analog port. Inside the main program samples are sent to the DAC with 454 μ s delay between each sample. Function dacWrite is used to send the samples to the DAC.

Figure 6.17 shows the generated Triangle waveform, captured on the PSCGU250 PC based oscilloscope.

*Figure 6.17 Generated Triangle waveform*

6.5 PROJECT 4 – Port Expander

6.5.1 Description

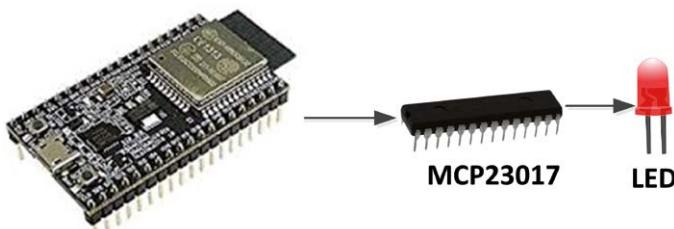
In this project a Port Expander chip (MCP23017) is used to give additional 16 I/O ports to the ESP32 DevKitC. An LED is connected to port pin GPA0 (pin 21) of MCP23017 for testing the hardware and the software where the LED is flashed ON and OFF every second. A 330 ohm current limiting resistor is used in series with the LED.

6.5.2 The Aim

The aim of this project is to show how the I2C commands can be used with the ESP32 DevKitC and the Arduino IDE, and also how the Port Expander chip MCP23017 can be programmed.

6.5.3 Block diagram:

Figure 6.18 shows the block diagram of the project where the LED is connected to one of the ports and 15 more I/O ports are available from the MCP23017.



ESP32 Devkitc

Figure 6.18 Block diagram of the project

6.5.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 6.19. MCP23017 is connected to ESP32 DevKitC through the I2C bus, where the SDA and the SCL lines of both devices are connected to each other and 10K pull-up resistors are used on these lines. The LED is connected to port pin GPA0 of the MCP23017 chip. Address select bits of the MCP23017 are all connected to ground so that the device slave address is set to 0x20.

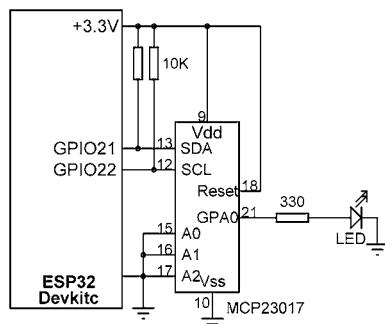


Figure 6.19 Circuit diagram of the project

6.5.5 The MCP23017

The MCP23017 is a 28 pin chip with the following features. Figure 6.20 shows a picture of the chip. The pin configuration is shown in Figure 6.21.

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

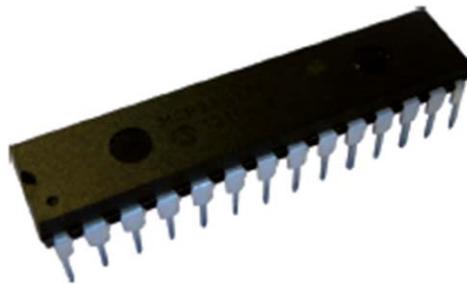


Figure 6.20 The MCP23017 port expander chip

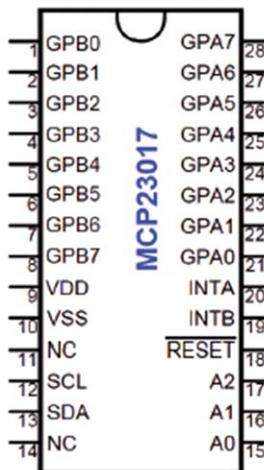


Figure 6.21 Pin configuration of the MCP23017

The pin descriptions are given in Table 6.1.

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

Table 6.1 MCP23017 pin descriptions

The MCP23017 is addressed using pins A0 to A2. Table 6.2 shows the address selection. In this project the address pins are connected to ground, thus the address of the chip is 0x20.

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

Table 6.2 Address selection of the MCP23017

The MCP23017 chip has 8 internal registers that can be configured for its operation. The device can either be operated in 16-bit mode or in two 8-bit mode by configuring bit IO-CON.BANK. On power-up this bit is cleared which chooses the two 8-bit mode by default. The I/O direction of the port pins are controlled with registers IODIRA (at address 0x00) and IODIRB (at address 0x01). Clearing a bit to 0 in these registers make the corresponding port pin(s) output(s). Similarly, setting a bit to 1 in these registers make the corresponding port pin(s) input(s). GPIOA and GPIOB register addresses are 0x12 and 0x13 respectively. This is shown in Figure 6.22.

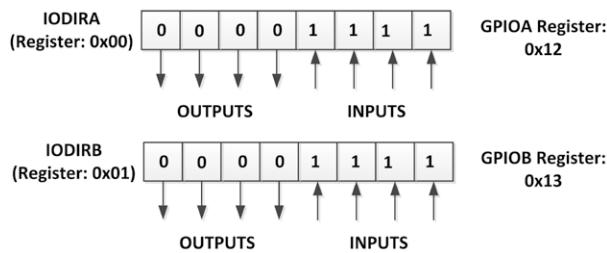


Figure 6.22 Configuring the I/O ports

More information on the MCP23017 chip can be obtained from the data sheet:

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

6.5.6 Construction

ESP32 DevKitC is mounted on a breadboard together with the MCP23017 Port Expander chip and the LED as shown in Figure 6.23.

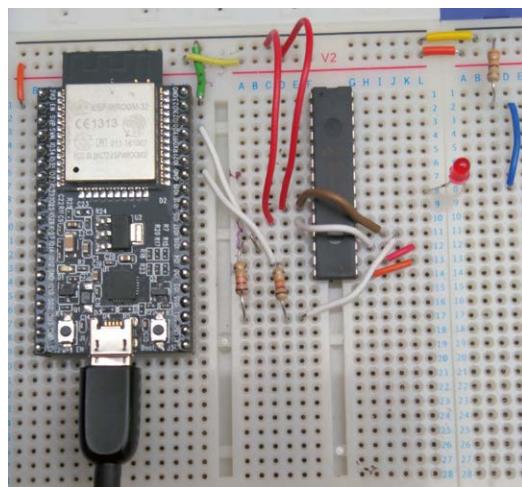


Figure 6.23 Project on a breadboard

6.5.7 PDL of the Project

The PDL of the project is shown in Figure 6.24.

```

BEGIN
  Include I2C library
  Define slave address and register addresses
  Initialize I2C bus
  Configure GPIOA as output
DO FOREVER
  Turn OFF LED (Send 0)
  Wait 1 second

```

```

        Turn ON LED (Send 1)
        Wait 1 second
ENDDO
END
```

Figure 6.24 PDL of the project

6.5.8 Program Listing

The program listing of the project is shown in Figure 6.25 (program: **Expander**).

```

***** PORT EXPANDER *****
*
* In this project an MCP23017 Port Expander chip is used to
* add 16 more I/O ports to the ESP32 Devkitc. This chip is
* interfaced to a processor via the I2C bus (SDA and SCL lines).
* 10K pull-up resistors are used at these lines. In order to
* test the hardware and teh software, an LED is connected to
* port GPA0 (pin 21) of the MCP23017 chip. The program flashes
* the LED every second.
*
*
* File:   Expander
* Date:   July 2017
* Author: Dogan Ibrahim
*****/
#include <Wire.h>                                // Include I2C library
const byte slave_addr=0x20;                        // MCP23017 I2C address
const byte IODIRA=0x00;                            // IODIRA address
const byte GPIOA=0x12;                            // GPIOA address

//
// This function configures the MCP23017 chip such that Port pin
// GPA0 is an output pin. PortDir is the address of the port
// direction register (IODIRA or IODIRB), and Data is the
// direction data such that 0 sets the corresponding port pin to
// output, and 1 sets the corresponding port pin to input
//
void Configure(char PortDir, char Data)
{
    Wire.beginTransmission(slave_addr);
    Wire.write(PortDir);
    Wire.write(Data);
    Wire.endTransmission();
}
```

```
//  
// This function sends data to the specified port. The port can be  
// GPIOA or GPIOB and data is a byte  
//  
void Send(char Port, char Data)  
{  
    Wire.beginTransmission(slave_addr);  
    Wire.write(Port);  
    Wire.write(Data);  
    Wire.endTransmission();  
}  
  
//  
// Initialize the I2C bus, configure port GPIOA as an output port  
//  
void setup()  
{  
    Wire.begin();  
    Configure(IODIRA, 0xFE);  
}  
  
//  
// Main program initializes the I2C, configures the GPIOA port  
// and flashes the LED connected to port pin GPA0 every second  
//  
void loop()  
{  
    Send(GPIOA, 0); // LED OFF  
    delay(1000); // Wait 1 second  
    Send(GPIOA, 1); // LED ON  
    delay(1000); // Wait 1 second  
}
```

Figure 6.25 Program listing

6.5.9 Program Description

Sending data to an I2C slave device is a 5 step process as described below:

- Begin transmission.
- Send device address.
- Send register address.
- Send data to the register.
- End transmission.

At the beginning of the program the I2C library is included in the program, slave address is set to 0x20, and the IODIRA and GPIOA addresses are set to 0x00 and 0x12 respectively. Inside the setup routine the I2C is initialised and the MCP23017 is configured so that GPIOA pin 0 is an output port. Notice that the hexadecimal number 0xFE (binary: 1111 1110) is sent to the register so that bit 0 is output, and all the other bits are inputs. Inside the main program function **Send** is called to turn the LED ON and OFF. One second delay is inserted between each output.

6.6 PROJECT 5 – Mini Electronic Organ

6.6.1 Description

In this project a mini electronic organ is designed using the ESP32 DevKitC together with a 4 x 4 keypad with 16-keys to produce musical tones in one octave.

6.6.2 The Aim

The aim of this project is to show how a keypad can be used in an ESP32 DevKitC project and also how a mini electronic organ can be designed using this keypad.

6.6.3 Block diagram:

Figure 6.26 shows the block diagram of the project where the keypad and a passive buzzer are connected to the ESP32 DevKitC.

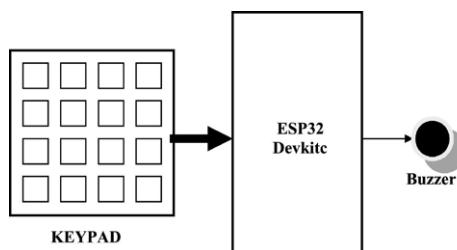


Figure 6.26 Block diagram of the project

6.6.4 Circuit Diagram

Figure 6.27 shows the keypad used in this design. It basically has 16 keys, labelled from S1 to S16 as follows:

Row 1:	S1 to S4
Row 2:	S5 to S8
Row 3:	S9 to S12
Row 4:	S13 to S16

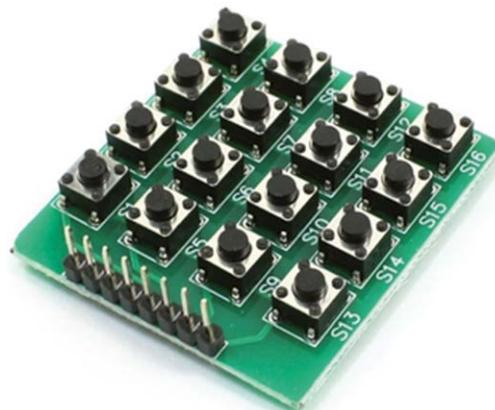


Figure 6.27 Keypad used in the design

The circuit diagram of the keypad is shown in Figure 6.28. Rows are numbered from top to bottom as 1 to 4. Similarly, columns are numbered from left to right as 1 to 4.

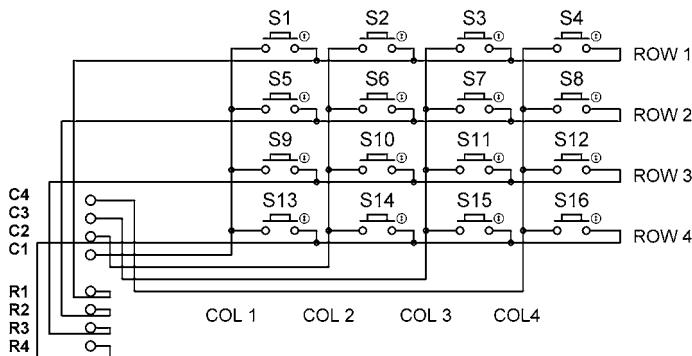


Figure 6.28 Circuit diagram of the keypad

The circuit diagram of the project is shown in Figure 6.29. The rows and columns of the keypad are connected as follows (pin 1 is the top pin when the pins are on the left of the keypad). See Figure 4.2 for the ESP32 DevKitC pin assignments.

Keypad Row	Keypad Pin	ESP32 GPIO Port
C4	1	23
C3	2	22
C2	3	21
C1	4	19
R1	5	18
R2	6	5
R3	7	17
R4	8	16

The passive buzzer is connected to GPIO port 4.

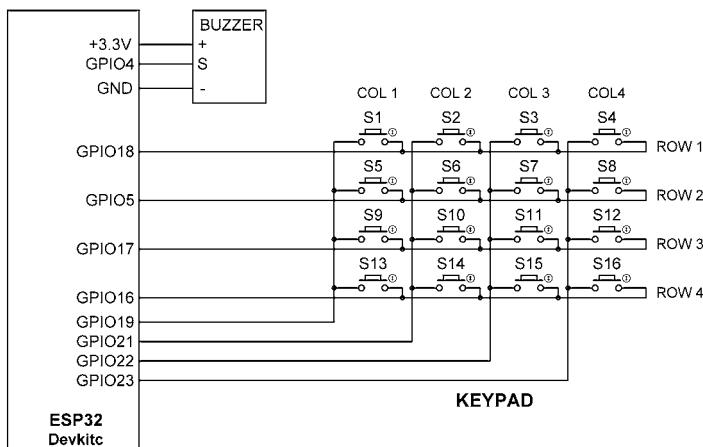


Figure 6.29 Circuit diagram of the project

The musical tones are configured on the keypad as follows:

C4	D4	E4	F4
G4	A4	B4	C5
C4#	D4#	F4#	G4#
A4#			

The frequencies of the notes are:

Notes	C4	C4#	D4	D4#	E4	F4	F4#	G4	G4#	A4	A4#	B4	C5
Hz	261.63	277.18	293.66	311.13	329.63	349.23	370	392	415.3	440	466.16	493.88	523.25

6.6.5 Construction

ESP32 DevKitC is mounted on a breadboard together with the keypad and the passive buzzer as shown in Figure 6.30.

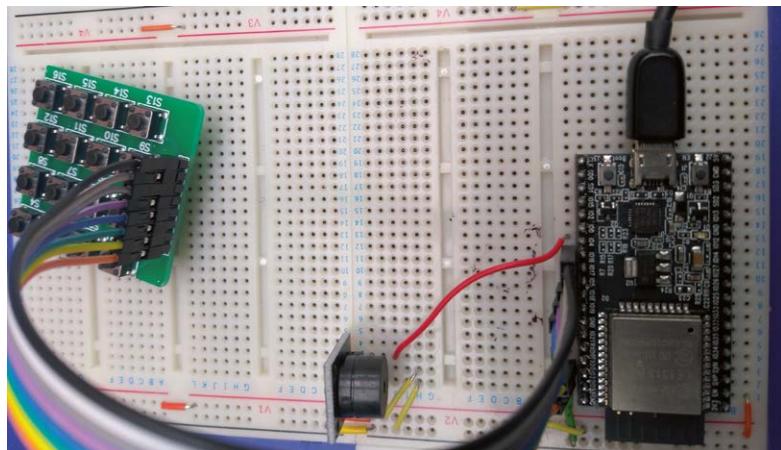


Figure 6.30 Project on a breadboard

6.6.6 PDL of the Project

The keypad library should be loaded into the Arduino IDE before the keypad functions can be used. To load this library, follow the steps given below:

- Start Arduino IDE.
- Click **Sketch -> Include Library -> Manage Libraries**.
- Search for **Keypad**.
- Click on the text and then click **Install** to install the library (see Figure 6.31).
- Close the library.

After installing the Keypad library, you should include the header file **Keypad.h** in your programs.

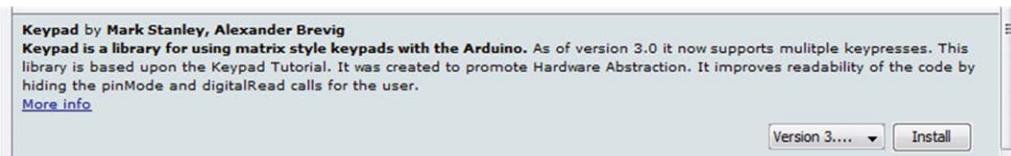


Figure 6.31 Install the Keypad library

The PDL of the project is shown in Figure 6.32.

```
BEGIN
Include Keypad.h in the program
Assign BUZZER to GPIO 4
Define musical tones for an octave
Define PWM parameters
Define keypad parameters
Define the interface between the keypad and the GPIO ports
DO FOREVER
```

```

IF a key is pressed THEN
    Find the numeric value of the key
    Find the required tone frequency
    Send tone to the buzzer for 250 ms
ENDIF
ENDDO
END

```

Figure 6.32 PDL of the project

6.6.7 Program Listing

Before writing the complete program it is worthwhile to construct the hardware and then write a program that will test the keypad interface by displaying the key numbers on the Serial Monitor of the Arduino IDE as the keys are pressed one by one. This program code is shown in Figure 6.33 (program: **TestKeypad**). Run the program and press keys on the keypad, you should see the key numbers (1 to 9 or a to g) displayed on the monitor as shown in Figure 6.34.

```

BEGIN
    Include Keypad.h in the program
    Assign BUZZER to GPIO 4
    Define musical tones for an octave
    Define PWM parameters
    Define keypad parameters
    Define the interface between the keypad and the GPIO ports
DO FOREVER
    IF a key is pressed THEN
        Find the numeric value of the key
        Find the required tone frequency
        Send tone to the buzzer for 250 ms
    ENDIF
ENDDO
END

```

Figure 6.33 Keypad test program

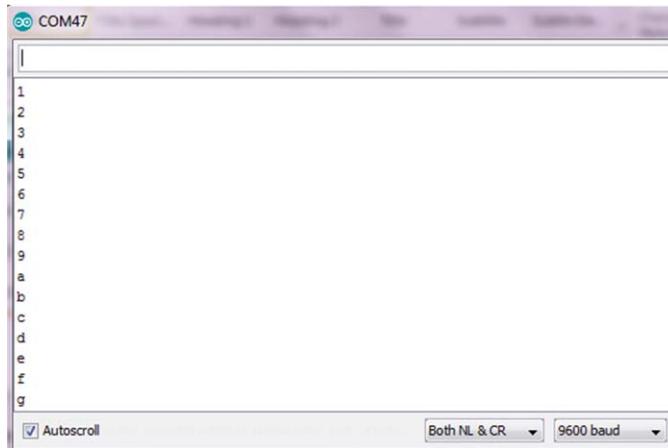


Figure 6.34 Key presses displayed on the Serial Monitor

The complete program listing of the project is shown in Figure 6.35 (program Organ).

```
*****  
* MINI ELECTRONIC ORGAN  
* =====  
*  
* In this project a 4x4 keypad is connected to the ESP32 DevKitC  
* boards. In addition, a passive buzzer is connected to GPIO  
* port 17. The program configures the keypad so that the  
* musical notes for one octave can be played by using the  
* keypad.  
*  
* The keypad is configured as follows (A to G are the musical  
* notes):  
* C4 D4 E4 F4  
* G4 A4 B4 C5  
* C4# D4# F4# G4#  
* A4#  
*  
*  
* File: Organ  
* Date: July 2017  
* Author: Dogan Ibrahim  
*****/  
#include "Keypad.h" // Include Keypad library  
#define BUZZER 4  
int Notes[] = {0,262,294,330,349,392,440,494,524,277,311,370,415,466};  
  
//  
// Define PWM parameters
```

```
//  
int duty = 127; // Duty cycle (50%)  
int resolution = 8; // 8 bit resolution  
int channel = 0; // Channel 0  
int freq = 1000; // To start with  
  
//  
// Define Keypad parameters  
//  
const byte Rows = 4; // Number of rows  
const byte Cols = 4; // Number of columns  
  
//  
// Define the keymap on the keypad  
//  
char keys[Rows][Cols] =  
{  
    {'1', '2', '3', '4'},  
    {'5', '6', '7', '8'},  
    {'9', 'a', 'b', 'c'},  
    {'d', 'e', 'f', 'g'}  
};  
  
//  
// Define the Keypad connections for the rows and columns  
//  
byte rowPins[Rows] = {18, 5, 17, 16}; // Rows 1 to 4  
byte colPins[Cols] = {19, 21, 22, 23}; // Cols 1 to 4  
  
//  
// Initialize the Keypad library with the row and columns defs.  
// Here, we have to specify the key map name (keys), row pin  
// assignments to GPIO ports (rowPins), and the column pin  
// assignments to GPIO ports (colPins)  
//  
Keypad kpd=Keypad(makeKeymap(keys),rowPins,colPins,Rows,Cols);  
  
//  
// Configure BUZZER as an output port. Also, configure the PWM  
//  
void setup()  
{  
    pinMode(BUZZER, OUTPUT);  
    ledcSetup(channel, freq, resolution);  
    ledcAttachPin(BUZZER, channel);  
}
```

```
//  
// Inside the main program, look for a key press. When a key is  
// pressed, get its code, convert to a number, find the frequency  
// corresponding to this number, and then activate the buzzer by  
// sending it the required frequency. The buzzer sounds for 250 ms  
// where after this time the buzzer stops.  
//  
void loop()  
{  
    char keypressed = kpd.getKey();           // Look for key press  
    if(keypressed != NO_KEY)                  // If a key is pressed  
    {  
        if(keypressed >= 'a')                // Convert to a number  
            keypressed = keypressed - 'a' + 10;  
        else keypressed -= '0';  
  
        freq = Notes[keypressed];           // Get the frequency  
        ledcWriteTone(channel, freq);         // Start buzzer  
        delay(250);                        // Sound for 250 ms  
        ledcWrite(channel, 0);               // Stop buzzer  
    }  
}
```

Figure 6.35 Program listing

6.6.8 Program Description

At the beginning of the program, header file **Keypad.h** is included in the program, BUZZER is assigned to GPIO port 4, and the frequencies of musical notes for one octave are stored in array **Notes** (the first element of the array is not used). Then the PWM parameters for generating musical tones are defined where the duty cycle is set to 50%, resolution to 8 bits, the frequency to 1000 Hz temporarily, and PWM channel 0 is selected.

The keypad is configured having 4 rows and 4 columns. Character array **keys** define the characters to be received by the program when a key is pressed on the keypad. For example, pressing the key at the top left hand of the keypad (S1) will send the character '1' to the program. The interface between the GPIO ports and the keypad are defined in array **rowPins** for rows 1 to 4, and **colPins** for columns 1 to 4 respectively. For example, in the following statement, row 1 is connected to GPIO port 18, row 2 to GPIO port 5 and so on.

```
byte rowPins[Rows] = {18, 5, 17, 16};
```

Inside the main program we look for a key press by using the statement **kpd.getKey**. When a key is pressed remember that what is received by the program is a character between '1' and '9' or 'a' and 'g'. This character is converted into a numeric value between 1 and 13 and used as an index to array **Notes** to get the required tone frequency. This tone

is then sent to the buzzer, which is set to sound for 250 ms. After this time the duty cycle is set to 0, which stops the buzzer sounding.

6.7 PROJECT 6 – Calculator with Keypad and LCD

6.7.1 Description

In this project a simple integer calculator is designed. The calculator can add, subtract, multiply, and divide integer numbers and show the results on the LCD. The operation of the calculator is as follows: when power is applied to the system, the LCD displays text **CAL-CULATOR** for 2 seconds. Then the text **No1:** is displayed in first row of the LCD and the user is expected to type the first number and then press the **ENTER** key. Then text **No2:** is displayed in the second row of the LCD where the user enters the second number and the user presses the **ENTER** key. After this the required operation key should be pressed. The result will be displayed on the LCD for 5 seconds and then the LCD will be cleared, ready for the next calculation. The example below shows how numbers 12 and 20 can be added (characters entered by the user are in bold for clarity):

No1: **15** **ENTER**

No2: **20** **ENTER**

Op: **+**

Res = 35

In this project the keyboard is labelled as follows:

1	2	3	4
5	6	7	8
9	0		ENTER
+	-	X	/

The key between 0 and ENTER is not used in this project.

6.7.2 The Aim

The aim of this project is to show how a keypad and an I2C LCD can be used together in a project, and also how a simple calculator can be designed.

6.7.3 Block diagram:

Figure 6.36 shows the block diagram of the project where the keypad and the I2C LCD are connected to the ESP32 DevKitC.

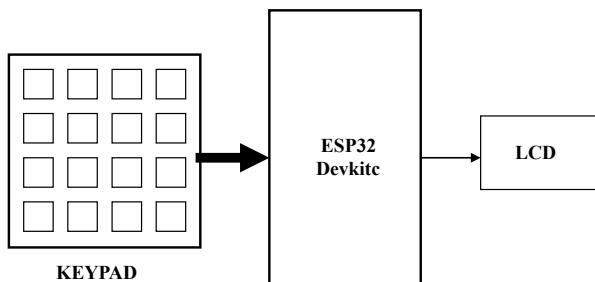


Figure 6.36 Block diagram of the project

6.7.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 6.37. The rows and columns of the keypad are connected as follows (pin 1 is the top pin when the pins are on the left of the keypad). See Figure 4.2 for the ESP32 DevKitC pin assignments. Notice that the connections here are different to the ones in the previous project since pins 21 and 22 are used by the I2C LCD module.

Keypad Row	Keypad Pin	ESP32 GPIO Port
C4	5	23
C3	6	19
C2	7	18
C1	8	5
R1	4	17
R2	3	16
R3	2	4
R4	1	0

The I2C LCD pins SDA and SCL are connected to GPIO pins 21 and 22 respectively.

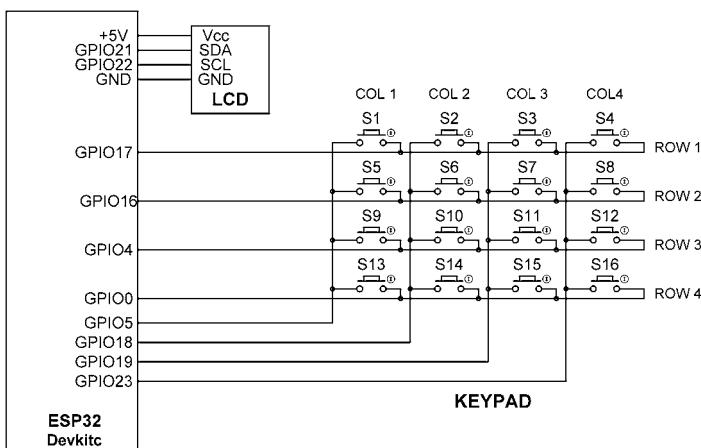


Figure 6.37 Circuit diagram of the project

6.7.5 Construction

ESP32 DevKitC is mounted on a breadboard together with the keypad and the I2C LCD as shown in Figure 6.38.

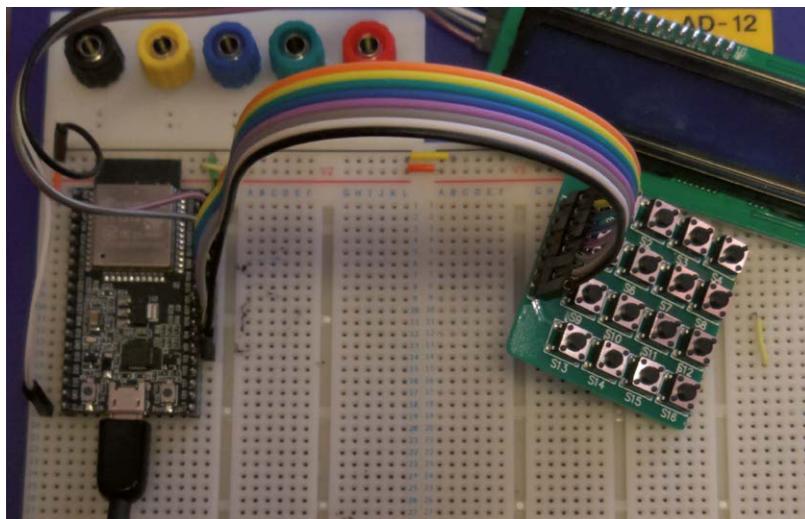


Figure 6.38 Project on a breadboard

6.7.6 PDL of the Project

As in the previous project, make sure that the keypad library has already been loaded to the Arduino IDE. The PDL of the project is shown in Figure 6.39.

```

BEGIN
Include the Keypad and the I2C libraries
Define the keypad and the I2C LCD connections
Initialize the LCD
Initialize the Keypad
Display heading CALCULATOR
DO FOREVER
  Display No1:
  Read the first number and save in Op1
  Display No2:
  Read the second number and store in Op2
  Display Op:
  Read the required operation
  Perform the required operation
  Display the result on the LCD
ENDDO
END

```

Figure 6.39 PDL of the project

6.7.7 Program Listing

Before writing the complete program it is worthwhile to construct the hardware and then write a program that will test the keypad interface by displaying the key numbers on the Serial Monitor of the Arduino IDE as the keys are pressed one by one. This program code is shown in Figure 6.40 (program: **TestKeypad2**). Run the program and press keys on the keypad, you should see the key numbers (1 to 9 or a to g) displayed on the Serial Monitor.

```
/*****  
*          CALCULATOR WITH KEYPAD AND LCD  
*          ======  
*  
* In this project a 4x4 keypad is connected to the ESP32 Devkitc  
* boards. In addition, an I2C LCD is connected. The program is a  
* simple integer calculator that can perform the arithmetic  
* operations of addition, subtraction, multiplication and  
* division.  
*  
* The keypad is configured as follows (the key between 0 and  
* ENTER is not used:  
*  
* 1  2  3  4  
* 5  6  7  8  
* 9  0      ENTER  
* +  -  X  /  
*  
*  
* File:  Testkeypad2  
* Date:  July 2017  
* Author: Dogan Ibrahim  
*****/  
#include "Keypad.h"                                // Include Keypad library  
//  
// Define Keypad parameters  
//  
const byte Rows = 4;                                // Number of rows  
const byte Cols = 4;                                // Number of columns  
//  
// Define the keymap on the keypad  
//  
char keys[Rows][Cols] =  
{  
    {'1', '2', '3', '4'},  
    {'5', '6', '7', '8'},  
    {'9', '0', ' ', 'E'},  
    {'+', '-', 'X', '/'}  
};
```

```

// Define the Keypad connections for the rows and columns
// byte rowPins[Rows] = {17, 16, 4, 0};           // Rows 1 to 4
// byte colPins[Cols] = {5, 18, 19, 23};          // Cols 1 to 4
// Initialize the Keypad library with the row and columns defs.
// Here, we have to specify the key map name (keys), row pin
// assignments to GPIO ports (rowPins), and the column pin
// assignments to GPIO ports (colPins)
// Keypad kpd=Keypad(makeKeymap(keys),rowPins,colPins,Rows,Cols);

void setup()
{
    Serial.begin(9600);
}

void loop()
{
    char keypressed = kpd.getKey();                // Look for key press
    if(keypressed != NO_KEY)                      // If a key is pressed
    {
        Serial.print(keypressed);                 // Display on Serial Monitor
    }
}

```

Figure 6.40 Keypad test program

The complete program listing of the project is shown in Figure 6.41 (program Calculator).

```

*****
*          CALCULATOR WITH KEYPAD AND LCD
*          =====
*
* In this project a 4x4 keypad is connected to the ESP32 Devkitc
* boards. In addition, an I2C LCD is connected. The program is a
* simple integer calculator that can perform the arithmetic
* operations of addition, subtraction, multiplication and
* division.
*
* The keypad is configured as follows (the key between 0 and
* ENTER is not used:
*
* 1 2 3 4

```

```
* 5 6 7 8
* 9 0      ENTER
* + - X /
*
*
* File:   Calculator
* Date:   July 2017
* Author: Dogan Ibrahim
******/
```

```
#include "Keypad.h"                      // Include Keypad library
#include <Wire.h>                         // Include I2C library
#include <LiquidCrystal_I2C.h>
```

```
//
// Keypad non-numeric keys
//
#define Enter 'E'
#define Plus '+'
#define Minus '-'
#define Multiply '*'
#define Divide '/'
```

```
//
// Set the LCD address to 0x27 and the configuration to 16 chars
// by 2 rows display
//
LiquidCrystal_I2C lcd(0x27, 16, 2);
```

```
//
// Define Keypad parameters
//
const byte Rows = 4;                      // Number of rows
const byte Cols = 4;                      // Number of columns
//
// Define the keymap on the keypad
//
char keys[Rows][Cols] =
{
    {'1', '2', '3', '4'},
    {'5', '6', '7', '8'},
    {'9', '0', ' ', 'E'},
    {'+', '-', '*', '/'}
};
//
// Define the Keypad connections for the rows and columns
//
```

```
byte rowPins[Rows] = {17, 16, 4, 0};           // Rows 1 to 4
byte colPins[Cols] = {5, 18, 19, 23};          // Cols 1 to 4
//
// Initialize the Keypad library with the row and columns defs.
// Here, we have to specify the key map name (keys), row pin
// assignments to GPIO ports (rowPins), and the column pin
// assignments to GPIO ports (colPins)
//
Keypad kpd=Keypad(makeKeymap(keys),rowPins,colPins,Rows,Cols);

char MyKey;
long Calc, Op1, Op2;

//
// Initialize the I2C LCD, turn ON backlight, clear LCD, display
// heading (CALCULATOR), wait 2 seconds and then clear LCD
//
void setup()
{
    lcd.begin();
    lcd.backlight();
    lcd.clear();
    lcd.print("CALCULATOR");                  // Display heading
    delay(2000);
    lcd.clear();
}

//
// Inside the main program, read the first number, the second
// number, and the required operation. The program displays the
// result for 5 seconds and after this time the process is
// repeated
//
void loop()
{
    Op1 = 0;
    Op2 = 0;
    MyKey = 0;

    lcd.setCursor(0,0);                      // Col 0, row 0
    lcd.print("No1: ");                     // Display No1:
    while(1)
    {
        do
            MyKey = kpd.getKey();
        while(!MyKey);                      // Wait until key pressed
```

```
    if(MyKey == Enter)break;           // ENTER key
    lcd.print(MyKey);                // Echo the key
    Op1 = 10*Op1 + MyKey - '0';     // Entered number so far
}

lcd.setCursor(0,1);                // Col0, row 1
lcd.print("No2: ");
while(1)
{
    do
        MyKey = kpd.getKey();
    while(!MyKey);                  // Wait until key pressed
    if(MyKey == Enter)break;
    lcd.print(MyKey);                // Echo the key
    Op2 = 10*Op2 + MyKey - '0';     // Entered number so far
}

lcd.clear();                      // Col 0, row 0
lcd.setCursor(0,0);                // Display Op:
do
    MyKey = kpd.getKey();
while(!MyKey);
lcd.clear();
lcd.setCursor(0,0);
lcd.print("Op: ");                // Display "Res="
// Calculate the result and display on the LCD
//
switch(MyKey)
{
    case Plus:
        Calc = Op1 + Op2;
        break;
    case Minus:
        Calc = Op1 - Op2;
        break;
    case Multiply:
        Calc = Op1 * Op2;
        break;
    case Divide:
        Calc = Op1 / Op2;
        break;
}
lcd.print(Calc);                  // Display result
delay(5000);                     // Wait 5 seconds
```

```

    lcd.clear();
}

```

Figure 6.41 Program listing

6.7.8 Program Description

At the beginning of the program, header files **Keypad.h**, **Wire.h** and **LiquidCrystal_i2C.h** are included. Then the non-numeric keypad keys are defined. The LCD is configured with address 0x27, having 16 characters and 2 rows.

The keypad is configured having 4 rows and 4 columns. Character array **keys** define the characters to be received by the program when a key is pressed on the keypad. For example, pressing the key at the top left hand of the keypad (S1) will send character '1' to the program. The interface between the GPIO ports and the keypad are defined in array **rowPins** for rows 1 to 4, and **colPins** for columns 1 to 4, respectively. For example, in the following statement, row 1 is connected to GPIO port 17, row 2 to GPIO port 16, and so on.

```
byte rowPins[Rows] = {17, 16, 4, 0};
```

Inside the setup routine text CALCULATOR is displayed for 2 seconds. The remainder of the program code is executed forever inside the main program loop. Here, **No1:** is displayed and the user is expected to enter the first integer number. The number read is stored in variable **Op1**. Similarly, the second integer number is read and stored in variable **Op2**. The program then prompts the user to enter the required operation after displaying text **Op:**. The result is calculated using a **switch** statement and is displayed on the LCD in the form **Res=nn** for 5 seconds. After this time the LCD is cleared and the process repeats.

Figure 6.42 shows a typical display where it is required to multiply numbers 12 and 2. The result is displayed in Figure 6.43 as **Res=24**.

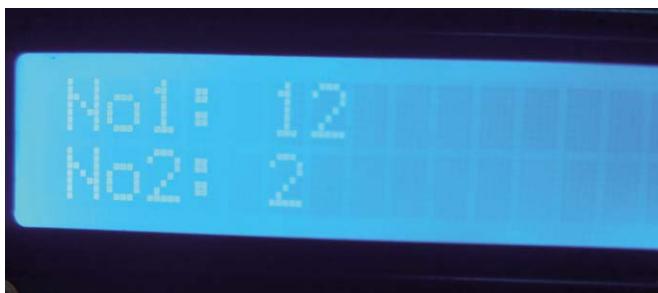
*Figure 6.42 Enter the numbers to be calculated*



Figure 6.43 Result of the multiplication

6.8 PROJECT 7 – High-Low Game

6.8.1 Description

This project uses a 4 x 4 keypad and an LCD as in the previous project to implement the classical High-Low game. For those of you who do not know how to play the game, here are the rules for this version of the game:

- The computer will generate a secret random number between 1 and 100.
- The top row of the LCD will display "Guess Now..".
- The player will try to guess what the number is by entering a number on the keypad and then pressing the ENTER key.
- If the guessed number is higher than the secret number the bottom row of the LCD will display "HIGH – Try Again".
- If the guessed number is lower than the secret number the bottom row of the LCD will display "LOW – Try Again".
- If the player guesses the number then the bottom row will display "Well Done..".
- The program waits for 5 seconds and the game restarts automatically.

In this project the keyboard is labelled as follows:

1	2	3	4
5	6	7	8
9	0	NU	ENTER
NU	NU	NU	NU

NU, here, are the unused keys in this project.

6.8.2 The Aim

The aim of this project is to show how a simple game can be created using a keypad and an LCD.

6.8.3 Block diagram:

The block diagram is same as in Figure 6.36.

6.8.4 Circuit Diagram

The circuit diagram of the project is as in Figure 6.37. The keypad and the LCD are connected to the ESP32 DevKitC as in the previous project.

6.8.5 Construction

ESP32 DevKitC is mounted on a breadboard together with the keypad and the I2C LCD as shown in Figure 6.38.

6.8.6 PDL of the Project

The PDL of the project is shown in Figure 6.44.

```

BEGIN
    Include the Keypad and the I2C libraries
    Define the keypad and the I2C LCD connections
    Initialize the LCD
    Initialize the Keypad
    Display heading HIGH-LOW GAME
    NewNumber = 1
DO FOREVER
    IF NewNumber = 1 THEN
        Generate a random number between 1 and 100
        NewNumber = 0
    ENDIF
    Display message "Guess Now"
    Read a number
    Calculate the difference between the generated and guessed number
    IF the difference > 0 THEN
        Display "HIGH - TRY AGAIN"
    ELSE IF the difference < 0 THEN
        Display "LOW -TRY AGAIN"
    ELSE IF the difference = 0 THEN
        Display "Well done"
        Wait 5 seconds
        NewNumber = 1
    ENDIF
ENDDO
END

```

Figure 6.44 PDL of the project

6.8.7 Program Listing

The program listing of the project is shown in Figure 6.45 (program: **HighLow**).

```
*****  
*          HIGH-LOW GAME  
*          ======  
*  
* In this project a 4x4 keypad is connected to the ESP32 Devkitc  
* boars. In addition, an I2C LCD is connected as in the previous  
* project. This project is the classical High-Low game. The  
* processor generates a random number between 1 and 100. The  
* user is required to guess this number. If the number entered  
* is smaller or greater than teh generated number then a  
* message is displayed to help the user guess the correct number.  
* The game continues until the user finds the generated number  
*  
* The keypad is configured as follows (NU keys are not used):  
*  
* 1 2 3 4  
* 5 6 7 8  
* 9 0 NU ENTER  
* NU NU NU NU  
*  
*  
* File:  HighLow  
* Date:  July 2017  
* Author: Dogan Ibrahim  
*****/  
#include "Keypad.h"                      // Include Keypad library  
#include <Wire.h>                         // Include I2C library  
#include <LiquidCrystal_I2C.h>  
//  
// Keypad non-numeric keys  
//  
#define Enter 'E'  
  
//  
// Set the LCD address to 0x27 and the configuration to 16 chars  
// by 2 rows display  
//  
LiquidCrystal_I2C lcd(0x27, 16, 2);  
  
//  
// Define Keypad parameters  
//  
const byte Rows = 4;                      // Number of rows
```

```
const byte Cols = 4;                                // Number of columns
//
// Define the keymap on the keypad
//
char keys[Rows][Cols] =
{
    {'1', '2', '3', '4'},
    {'5', '6', '7', '8'},
    {'9', '0', ' ', 'E'},
    {' ', ' ', ' ', ' '}
};

//
// Define the Keypad connections for the rows and columns
//
byte rowPins[Rows] = {17, 16, 4, 0};                // Rows 1 to 4
byte colPins[Cols] = {5, 18, 19, 23};                // Cols 1 to 4
//
// Initialize the Keypad library with the row and columns defs.
// Here, we have to specify the key map name (keys), row pin
// assignments to GPIO ports (rowPins), and the column pin
// assignments to GPIO ports (colPins)
//
Keypad kpd=Keypad(makeKeymap(keys),rowPins,colPins,Rows,Cols);

long MyGuess;
char MyKey, NewGame = 1;
int Guess, Diff;

//
// Initialize the I2C LCD, turn ON backlight, clear LCD, display
// heading (HIGH-LOW GAME), wait 2 seconds and then clear LCD
//
void setup()
{
    lcd.begin();
    lcd.backlight();
    lcd.clear();
    lcd.print("HIGH-LOW GAME");                      // Display heading
    delay(2000);
    lcd.clear();
    randomSeed(10);
}

//
// Inside the main program, generate a random number between 1
// and 100. The user attempts to guess this number and sees
```

```
// messages such as HIGH or LOW to help him in making a choice
// for the next guess
//
void loop()
{
    if(NewGame == 1)
    {
        lcd.clear();
        lcd.print("Guess Now..."); 
        Guess = random(1, 101);           // Generate between 1-100
        NewGame = 0;
    }
    MyGuess = 0;
    MyKey = 0;

    lcd.setCursor(0,1);                // Col 0, row 0
    while(1)
    {
        do
            MyKey = kpd.getKey();
        while(!MyKey);                  // Wait until key pressed
        if(MyKey == Enter)break;         // ENTER key
        lcd.print(MyKey);               // Echo the key
        MyGuess = 10*MyGuess + MyKey - '0'; // Entered number so far
    }

    Diff = MyGuess - Guess;           // Difference
    if(Diff > 0)                    // High Guess ?
    {
        lcd.setCursor(0,1);
        lcd.print("HIGH - TRY AGAIN"); // Display HIGH
        delay(2000);
        lcd.setCursor(0,1);
        lcd.print("          ");
    }
    else if(Diff < 0)
    {
        lcd.setCursor(0,1);
        lcd.print("LOW - TRY AGAIN"); // Display LOW
        delay(2000);
        lcd.setCursor(0,1);
        lcd.print("          ");
    }
    else if(Diff == 0)
    {
        lcd.setCursor(0,1);
```

```

        lcd.print("Well Done...");           // Correct
        delay(5000);                      // Wait 5 seconds
        NewGame = 1;                      // New game flag set
    }
}

```

Figure 6.45 Program listing

6.8.8 Program Description

At the beginning of the program, header files **Keypad.h**, **Wire.h** and **LiquidCrystal_i2C.h** are included. Then the non-numeric keypad keys are defined. The LCD is configured with address 0x27, having 16 characters and 2 rows.

Inside the main program a random integer number is generated between 1 and 100 and stored in variable **Guess**. Then, the number entered by the user is stored in variable **My-Guess**. The difference between the generated number and the guessed number is stored in variable **Diff**. If **Diff** is greater than zero, i.e. the guessed number is greater than the generated one, the message **HIGH – TRY AGAIN** is displayed and the user is prompted to try again. If **Diff** is less than zero, i.e. the guessed number is smaller than the generated one, the message **LOW – TRY AGAIN** is displayed and the user is prompted to try again. If on the other hand **Diff** is equal to zero then the guessed number is found and message **Well Done...** is displayed. The game repeats with a new number after 5 seconds.

Figure 6.46, 6.47 and 6.48 show sample displays from the program.



Figure 6.46 Sample display – guessed number is lower



Figure 6.47 Sample display – guessed number is higher



Figure 6.48 Sample display – correct guess

6.9 PROJECT 8 – Learning the Times Table

6.9.1 Description

Most children learn the times tables in primary school before the age of 12. In this program the times table is displayed on the LCD for a given number, in the following format (assuming the times table for 5 is required):

```
1 X 5 = 5
2 X 5 = 10
3 X 5 = 15
.....
.....
12 X 5 = 60
```

This project uses a 4 x 4 keypad and an LCD as in the previous project. Each output is displayed for one second and the table runs from 1 to 12.

In this project the keyboard is labelled as follows:

1	2	3	4
5	6	7	8
9	0	NU	ENTER
NU	NU	NU	NU

NU, here, are the unused keys in this project.

6.9.2 The Aim

The aim of this project is to teach the times table to children in a fun way.

6.9.3 Block diagram:

The block diagram is same as in Figure 6.36.

6.9.4 Circuit Diagram

The circuit diagram of the project is as in Figure 6.37. The keypad and the LCD are connected to the ESP32 DevKitC as in the previous project.

6.9.5 Construction

ESP32 DevKitC is mounted on a breadboard together with the keypad and the I2C LCD as shown in Figure 6.38.

6.9.6 PDL of the Project

The PDL of the project is shown in Figure 6.49.

```

BEGIN
    Include the Keypad and the I2C libraries
    Define the keypad and the I2C LCD connections
    Initialize the LCD
    Initialize the Keypad
    Display heading TIMES TABLE
    NewTable = 1
DO FOREVER
    IF NewTable = 1 THEN
        Prompt for the required number
        NewTable = 0
    ENDIF
    FOR i=1 to 12
        Res = i * required number
        Display i, "X", required number, "=", Res
        Wait 1 second
    ENDFOR
    NewTable = 1
ENDDO
END

```

Figure 6.49 PDL of the project

6.9.7 Program Listing

The program listing of the project is shown in Figure 6.50 (program: **TimesTable**).

```

/*****
*          LEARNING THE TIMES TABLE
=====
*
* In this project a 4x4 keypad is connected to the ESP32 Devkitc
* boars. In addition, an I2C LCD is connected as in the previous
* project. This project displays the times table for a chosen
* number. Each output is displayed for one second.
*
* The keypad is configured as follows (NU keys are not used):
*
* 1 2 3 4
* 5 6 7 8
* 9 0 NU ENTER
* NU NU NU NU
*
*/

```

```
* File:    TimesTable
* Date:    July 2017
* Author: Dogan Ibrahim
*****
#include "Keypad.h"                                // Include Keypad library
#include <Wire.h>                                  // Include I2C library
#include <LiquidCrystal_I2C.h>
//
// Keypad non-numeric keys
//
#define Enter 'E'

//
// Set the LCD address to 0x27 and the configuration to 16 chars
// by 2 rows display
//
LiquidCrystal_I2C lcd(0x27, 16, 2);

//
// Define Keypad parameters
//
const byte Rows = 4;                               // Number of rows
const byte Cols = 4;                               // Number of columns
//
// Define the keymap on the keypad
//
char keys[Rows][Cols] =
{
    {'1', '2', '3', '4'},
    {'5', '6', '7', '8'},
    {'9', '0', ' ', 'E'},
    {' ', ' ', ' ', ' '}
};
//
// Define the Keypad connections for the rows and columns
//
byte rowPins[Rows] = {17, 16, 4, 0};                // Rows 1 to 4
byte colPins[Cols] = {5, 18, 19, 23};                // Cols 1 to 4
//
// Initialize the Keypad library with the row and columns defs.
// Here, we have to specify the key map name (keys), row pin
// assignments to GPIO ports (rowPins), and the column pin
// assignments to GPIO ports (colPins)
//
Keypad kpd=Keypad(makeKeymap(keys),rowPins,colPins,Rows,Cols);
```

```
int Res, MyChoice, NewTable = 1;
char MyKey;

// Initialize the I2C LCD, turn ON backlight, clear LCD, display
// heading (TIMES TABLE) for 2 seconds
//
void setup()
{
    lcd.begin();
    lcd.backlight();
    lcd.clear();
    lcd.print("TIMES TABLE");           // Display heading
    delay(2000);                      // Wait 2 seconds
}

// Inside the main program, read the number whose times table is
// required and then display the times table for this number
//
void loop()
{
    if(NewTable == 1)                // If new table
    {
        lcd.clear();
        lcd.print("Which Number ?"); // No for the table
        NewTable = 0;
    }

    MyKey = 0;
    MyChoice = 0;

    lcd.setCursor(0,1);             // Col 0, row 1
    while(1)
    {
        do
            MyKey = kpd.getKey();
            while(!MyKey);          // Wait until key pressed
            if(MyKey == Enter)break; // ENTER key
            lcd.print(MyKey);       // Echo the key
            MyChoice = 10*MyChoice + MyKey - '0'; // Entered number so far
        }

        lcd.clear();
        lcd.print("Timetable for ");
        lcd.print(MyChoice,10);     // Display heading
        // Display no
    }
}
```

```
for(int i = 1; i <=12; i++)           // Do for 1 to 12
{
    lcd.setCursor(0, 1);             // Goto col0, row 1
    lcd.print("      ");
    lcd.setCursor(0,1);
    Res = i * MyChoice;            // Calculate the result
    lcd.print(i,DEC);              // Display the times table
    lcd.print(" X ");
    lcd.print(MyChoice, 10);
    lcd.print(" = ");
    lcd.print(Res, 10);
    delay(1000);                  // Wait 1 second
}
lcd.clear();
NewTable = 1;                      // Set new table flag
}
```

Figure 6.50 Program listing

6.9.8 Program Description

At the beginning of the program header files **Keypad.h**, **Wire.h** and **LiquidCrystal_i2C.h** are included in the program as in the previous keypad and LCD projects. Then the non-numeric keypad keys are defined. The LCD is configured with address 0x27, having 16 characters and 2 rows. In the setup routine LCD is initialised, backlight turned ON and the text TIMES TABLE is displayed for 2 seconds.

Inside the main program the user is requested to enter the number of which the times table is to be displayed and this number is stored in variable **MyChoice**. Then, a **for** loop is set up that runs from 1 to 12 and displays the times table for the required number. When the times table is complete the LCD is cleared and the new times table flag **NewTable** is set to 1 so that a new table can be displayed.

Figure 6.51 shows a sample display from the program.



Figure 6.51 Sample display

6.10 PROJECT 9 – Learning Basic Mathematics

6.10.1 Description

This project aims to teach basic mathematics (multiplication here) to children. Two integer random numbers are generated between 1 and 100. These numbers are multiplied but the result is not shown for 30 seconds. The user is expected to find the answer and compare it with the one displayed by the program after 30 seconds.

A typical run of the program is as follows:

```
Multiplication
26 X 5 = (wait 30 seconds)

26 X 5 = 130
```

6.10.2 The Aim

The aim of this project is to teach basic mathematics (multiplication in this case) to children.

6.10.3 Block diagram:

The block diagram is same as in Figure 5.65.

6.10.4 Circuit Diagram

The circuit diagram of the project is as in Figure 5.66.

6.10.5 Construction

ESP32 DevKitC is mounted on a breadboard together with the I2C LCD as shown in Figure 5.67.

6.10.6 PDL of the Project

The PDL of the project is shown in Figure 6.52.

```
BEGIN
    Include the I2C library
    Define the I2C LCD connections
    Initialize the LCD
    Display heading BASIC MATHS
DO FOREVER
    Generate two integer random numbers between 1 and 100
    Multiply these two numbers
    Display the numbers but not the result of the multiplication
    Wait 30 seconds
    Display the result
ENDDO
END
```

Figure 6.52 PDL of the project

6.10.7 Program Listing

The program listing of the project is shown in Figure 6.53 (program: **Maths**).

```
*****  
*          BASIC MATHEMATICS  
*          =====  
*  
* In this project an I2C LCD is connected to the ESP32 Devkitc.  
* The program generates two integer random numbers between 1 and  
* 100 and multiplies these numbers. The result is not shown for  
* 30 seconds where the program waits so that the user can  
* calculate the result. After 30 seconds the result is displayed  
* so that the user can check with his/her own result.  
*  
* The I2C LCD is conencted to the ESP332 Devkitc as in the  
* previous projects, i.e. using the SDA and SCL I2C lines  
*  
* File: Maths  
* Date: July 2017  
* Author: Dogan Ibrahim  
*****  
#include <Wire.h>                                // Include I2C library  
#include <LiquidCrystal_I2C.h>  
  
//  
// Set the LCD address to 0x27 and the configuration to 16 chars  
// by 2 rows display  
//  
LiquidCrystal_I2C lcd(0x27, 16, 2);  
  
int FirstNumber, SecondNumber, Res;  
  
//  
// Initialize the I2C LCD, turn ON backlight, clear LCD, display  
// heading (MATHS) for 2 seconds  
//  
void setup()  
{  
    lcd.begin();                                     // Initialize LCD  
    lcd.backlight();                                 // Backlight ON  
    lcd.clear();                                    // Clear LCD  
    lcd.print("BASIC MATHS");                      // Display heading  
    delay(2000);                                    // Wait 2 seconds  
    randomSeed(10);  
}
```

```

// Inside the main program, read the number whose times table is
// required and then display the times table for this number
//
void loop()
{
    lcd.clear();
    lcd.print("Multiplication"); // Display heading
    FirstNumber = random(1, 101); // First number
    SecondNumber = random(1, 101); // Second number
    Res = FirstNumber * SecondNumber; // Result
    lcd.setCursor(0, 1); // Go to col 0, row 1
    lcd.print("      ");
    lcd.setCursor(0,1);
    lcd.print(FirstNumber,DEC); // Display the question
    lcd.print(" X ");
    lcd.print(SecondNumber, 10);
    lcd.print(" = ");
    delay(30000); // Wait 30 seconds
    lcd.print(Res, 10);
    delay(2000); // Wait 2 seconds
}

```

Figure 6.53 Program listing

6.10.8 Program Description

At the beginning of the program header files **Wire.h** and **LiquidCrystal_I2C.h** are included in the program as in the previous LCD based projects. In the setup routine the LCD is initialised, backlight is turned ON, and heading BASIC MATHS is displayed on the LCD.

Inside the main program two integer random numbers called **FirstNumber** and **SecondNumber** are generated and their product is stored in variable **Res**. The program displays the numbers in the second row of the LCD but does not display the result for 30 seconds. During this time the user is expected to calculate the answer. After 30 seconds the result is displayed so that the user can compare it with their own answer.

6.10.9 Suggestions

Although only multiplication is used in this project, the program can be modified to include all four operations of addition, subtraction, multiplication, and division.

Figure 6.54 shows a typical display.

*Figure 6.54 A typical display*

6.11 PROJECT 10 - Keypad Door Lock

6.11.1 Description

This is an electronic door lock project using a keypad and an LCD. A 4 digit secret number (can be changed) is stored in the system. When power is applied to the system, the message **Enter code:** is displayed. The user will have to enter the correct code in order for the lock to be opened. In this project a relay is activated when the correct code is entered. It is assumed that an electrical lock mechanism is connected to the relay. If the correct code is entered then the message **Door Opened** will be displayed and the relay will be activated. If a wrong code is entered then the message **Wrong code** will be displayed. For security reasons, one minute delay is introduced into the system after 3 consecutive failed attempts and the message **Failed to open** will be displayed. The secret code is chosen as **1234** in this example.

6.11.2 The Aim

The aim of this project is to show how an electronic door lock system can be designed using the ESP32 DevKitC, a keypad, an LCD, and a relay.

6.11.3 Block diagram:

The block diagram of the project is shown in Figure 6.55.

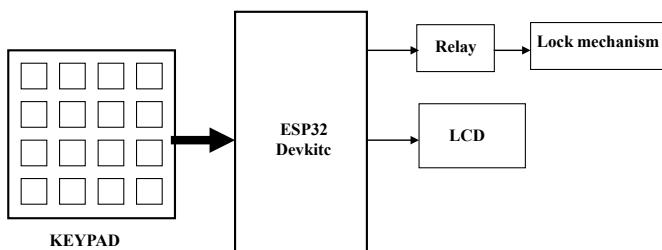


Figure 6.55 Block diagram of the project

6.11.4 Circuit Diagram

The circuit diagram of the project is as in Figure 6.56. The keypad and the LCD are connected to ESP32 DevKitC as in Project 8. Pin S of the relay (see Figure 6.3) is connected to GPIO port 2.

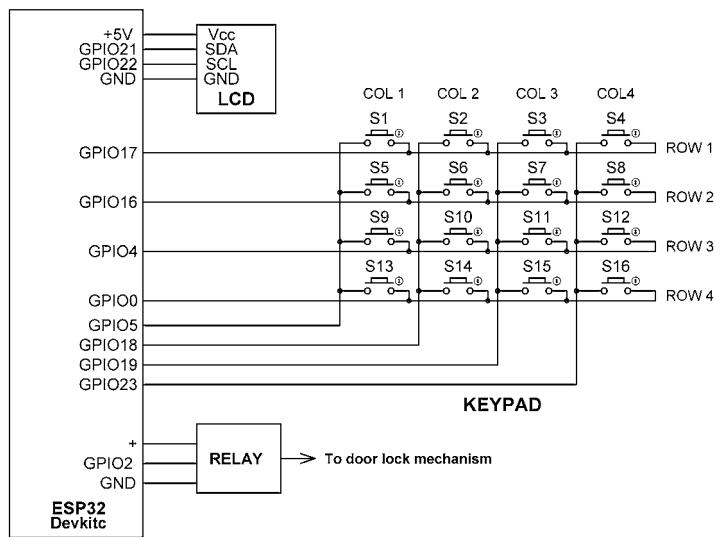


Figure 6.56 Circuit diagram of the project

6.11.5 Construction

ESP32 DevKitC is mounted on a breadboard together with the keypad, LCD, and the relay as shown in Figure 6.57.

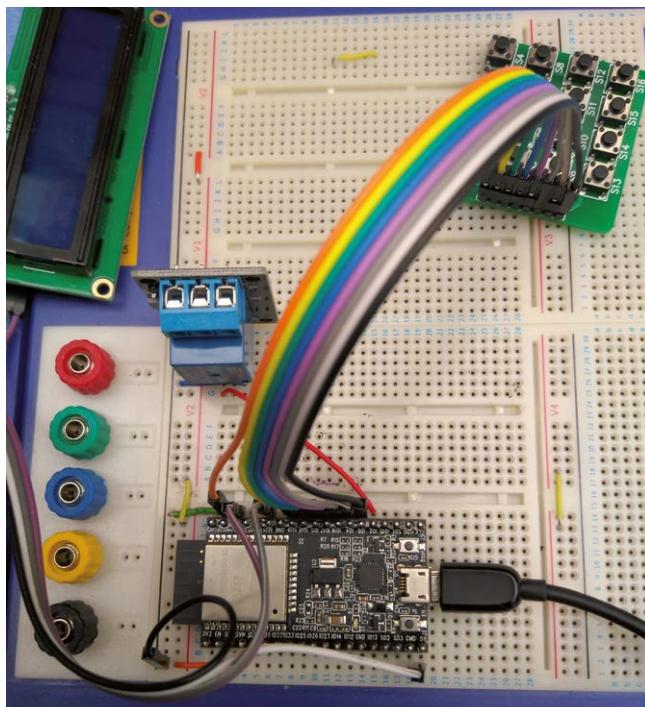


Figure 6.57 Project on a breadboard

6.11.6 PDL of the Project

The PDL of the project is shown in Figure 6.58.

```
BEGIN
    Include the Keypad and the I2C libraries
    Define the keypad and the I2C LCD connections
    Define the secret number
    Initialize the LCD
    Initialize the Keypad
    Clear attempt number
DO FOREVER
    Display Enter code:
    Read user code
    IF correct code entered THEN
        Activate the relay
        Wait 10 seconds
        Deactivate the relay
        Clear attempt number
    ELSE IF wrong code entered THEN
        IF 3rd attempt THEN
            Display failure message
            Wait for 1 minute
            Clear attempt number
        ELSE
            Increment attempt number
            Display wrong code message
            Wait 2 seconds
        ENDIF
    ENDIF
ENDDO
END
```

Figure 6.58 PDL of the project

6.11.7 Program Listing

The program listing of the project is shown in Figure 6.59 (program: **Lock**).

```
/*****
 *          KEYPAD LOCK
 *          =====
 *
 * This is an electronic lock project using a keypad, LCD, and
 * a buzzer. A 4 digit secret number is stored in memory. The
 * lock mechanism is electrically operated and is connected to
 * a relay. The user must enter the correct code in order to
 * activate the relay and open the door. If wrong code is entered
```

```
* on three consequent attempts then 30 second delay is introduced
* to the system for security.
*
* The relay is connected to GPIO port 2 of the ESP32 Devkitc.
*
* The keypad is configured as follows (NU keys are not used):
*
* 1 2 3 4
* 5 6 7 8
* 9 0 NU ENTER
* NU NU NU NU
*
* File: Lock
* Date: July 2017
* Author: Dogan Ibrahim
*****
#define SecretNumber 1234                                // Secret number
#define RELAY 2                                         // Relay at port 0
#include "Keypad.h"                                    // Include Keypad library
#include <Wire.h>                                       // Include I2C library
#include <LiquidCrystal_I2C.h>

//
// Keypad non-numeric keys
//
#define Enter 'E'

//
// Set the LCD address to 0x27 and the configuration to 16 chars
// by 2 rows display
//
LiquidCrystal_I2C lcd(0x27, 16, 2);

//
// Define Keypad parameters
//
const byte Rows = 4;                                  // Number of rows
const byte Cols = 4;                                  // Number of columns

//
// Define the keymap on the keypad
//
char keys[Rows][Cols] =
{
    {'1', '2', '3', '4'},
    {'5', '6', '7', '8'},
```

```
{'9', '0', ' ', 'E'},  
{' ', ' ', ' ', ' '}  
};  
  
//  
// Define the Keypad connections for the rows and columns  
//  
byte rowPins[Rows] = {17, 16, 4, 0}; // Rows 1 to 4  
byte colPins[Cols] = {5, 18, 19, 23}; // Cols 1 to 4  
  
//  
// Initialize the Keypad library with the row and columns defs.  
// Here, we have to specify the key map name (keys), row pin  
// assignments to GPIO ports (rowPins), and the column pin  
// assignments to GPIO ports (colPins)  
//  
Keypad kpd=Keypad(makeKeymap(keys),rowPins,colPins,Rows,Cols);  
  
int Res, MyNumber, Attempts = 0;  
char MyKey;  
  
//  
// Initialize the I2C LCD, turn ON backlight, clear LCD  
//  
void setup()  
{  
    pinMode(RELAY, OUTPUT); // Relay is output  
    digitalWrite(RELAY, LOW); // Relay off  
    lcd.begin(); // Initialize LCD  
    lcd.backlight(); // Backlight ON  
    lcd.clear(); // Clear LCD  
}  
  
//  
// Inside the main program, prompt for the code to be entered.  
// If the code is correct then open the door (activate the relay).  
// If teh code is wrong, display message to say that it is wrong.  
// After three failures stop the system for 1 minute for security.  
//  
void loop()  
{  
    lcd.clear();  
    lcd.print("Enter code:"); // Prompt for the code  
  
    MyKey = 0;
```

```
MyNumber = 0;

lcd.setCursor(0,1);                                // Col 0, row 1
while(1)                                         // Get user code
{
    do
        MyKey = kpd.getKey();
    while(!MyKey);                                // Wait until key pressed
    if(MyKey == Enter)break;                      // ENTER key
    lcd.print(MyKey);                            // Echo the key
    MyNumber = 10*MyNumber + MyKey - '0';          // Entered number so far
}

if(MyNumber == SecretNumber)                      // Correct code ?
{
    Attempts = 0;
    digitalWrite(RELAY, HIGH);                   // Open door
    lcd.clear();
    lcd.print("Door Opened");                  // Display opened
    delay(10000);                           // Wait 10 seconds
    digitalWrite(RELAY, LOW);                  // Close door
}
else
{
    Attempts++;
    if(Attempts == 3)                         // 3 failures ?
    {
        Attempts = 0;                        // Clear attempts
        lcd.setCursor(0,1);                  // Go to col 0, row 1
        lcd.print("Failed to open");        // Display Wrong code
        delay(60000);                      // Wait 1 minute
    }
    else
    {
        lcd.setCursor(0,1);                  // Goto col 0, row 1
        lcd.print("Wrong code...");        // Display wrong code
        delay(2000);                       // Wait 2 seconds
        lcd.clear();                        // Clear LCD
    }
}
```

Figure 6.59 Program listing

6.11.8 Program Description

At the beginning of the program header files **Keypad.h**, **Wire.h** and **LiquidCrystal_i2C.h** are included in the program as in the previous keypad and LCD based projects. In the setup routine the relay is configured as an output port and it is turned OFF to start with, the LCD is initialised, backlight is turned ON, and it is cleared. Variable **SecretNumber** stores the secret code (**1234** in this example, but it can be changed if desired).

Inside the main program the message **Enter code:** is displayed and the user is prompted to enter the secret code, where the user entered code is entered in variable **MyNumber**. If correct number is entered then the relay is activated to open the door. If the wrong code is entered the user has three attempts. After three attempts the system is blocked for one minute. This is done to increase the security by discouraging the user from keeping on trying different numbers. Figure 6.60 shows a sample display from the project.



Figure 6.60 Sample display

6.11.9 Suggestions

Try to modify the program by increasing the secret number to 6 digits. Also, add a buzzer to the system such that the buzzer sounds when a wrong number is entered.

6.12 PROJECT 11 – Using SD Cards – Writing to a Card

6.12.1 Description

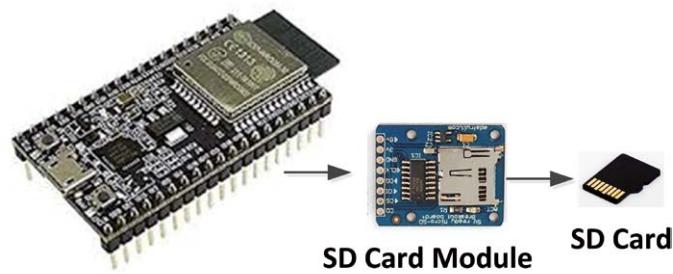
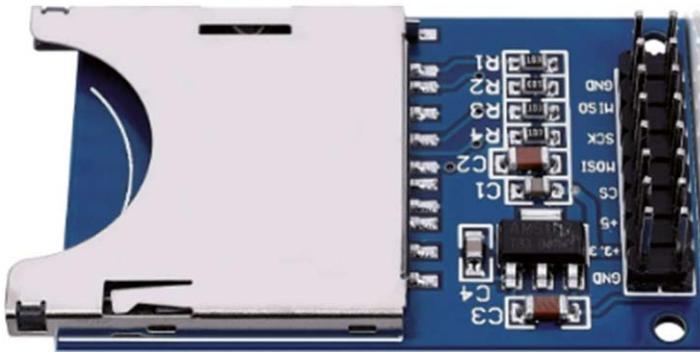
In this project an SD card module is connected to the ESP32 DevKitC. The Program generates integer random numbers every second and writes them to the SD card for 10 seconds (i.e. 10 numbers are written to the file) with a heading. The data is written to a file called TEST.TXT under directory NUM.

6.12.2 The Aim

The aim of this project is to show how an SD card can be interfaced to the ESP32 DevKitC and how data can be written to the SD card.

6.12.3 Block diagram:

The block diagram of the project is shown in Figure 6.61. In this project a +3.3 V compatible standard size SD card module is used as shown in Figure 6.62.

**ESP32 DevKitC***Figure 6.61 Block diagram of the project**Figure 6.62 SD card module used in the project*

6.12.4 Circuit Diagram

A standard SD card is used in this project. As shown in Figure 6.63, these cards can be controlled using the SPI bus interface and they have 9 pins with the following pin layout:

Pin number	Description
1	CS (SS)
2	MOSI
3	Ground
4	Voltage
5	Clock (SCK)
6	Ground
7	MISO
8	Unused
9	Unused

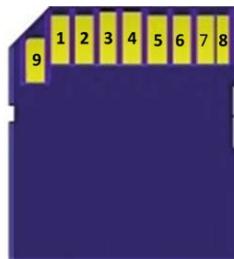


Figure 6.63 Standard SD card

In this project an SD card module is used as shown in Figure 6.62. This module has the following pin layout:

GND
3+3.3 V
+5 V
CS
MOSI
SCK
MISO
GND

The circuit diagram of the project is shown in Figure 6.64. SD card module is connected to DevKitC through the standard SPI interface. The connections between the SD card module and the DevKitC GPIO ports are as follows:

SD card Board Pin	DevKitC Pin
GND	GND
3.3	+3.3 V
SCK	GPIO18 (SPI SCK)
MISO	GPIO19 (SPI MISO)
MOSI	GPIO23 (SPI MOSI)
CS	GPIO5 (SPI SS)

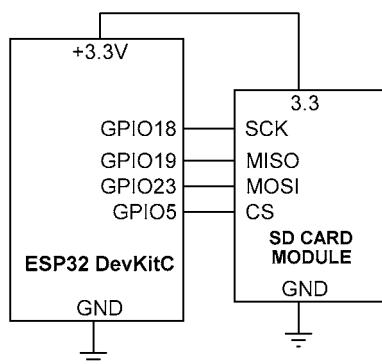


Figure 6.64 Circuit diagram of the project

6.12.5 Construction

ESP32 DevKitC is mounted on a breadboard together with the SD card module as shown in Figure 6.65. Jumper wires are used to connect the SD card module to the DevKitC.

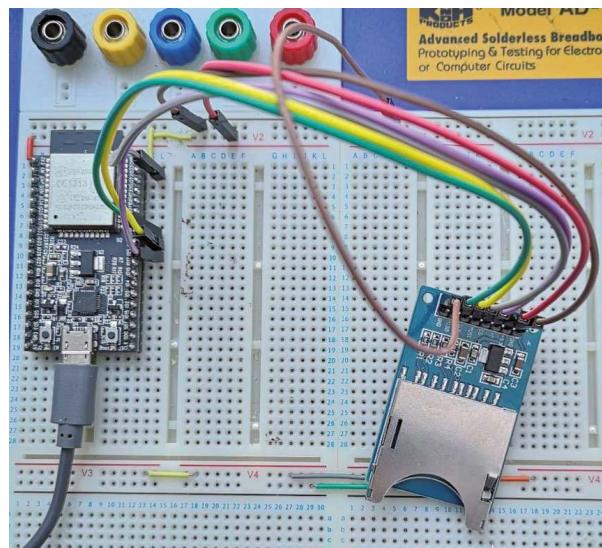


Figure 6.65 Project on a breadboard

6.12.6 PDL of the Project

The PDL of the project is shown in Figure 6.66.

```

BEGIN
  Define SD card CS pin
  Initialize SD card library
  Enable the serial USB communication to the PC
  Create directory NUM on SD card
  Create file TEST.TXT under directory NUM
  Display message on the PC screen
  Write heading to the file
DO 10 times
  Generate a random number
  Save the generated number in the file
  Wait one second
ENDDO
  Display message on the PC screen
  Close the file
END

```

Figure 6.66 PDL of the project

6.12.7 Program Listing

The program listing of the project is shown in Figure 6.67 (program: **SD-CARD**).

```
*****  
*          WRITE TO SD CARD  
*          =====  
*  
* In this program a standard SD card module is connected to the  
* ESP32 DevKitC. The program generates random numbers every second  
* and stored them on the SD card over 10 seconds (i.e. 10 numbers)  
* are stored on the SD card. The data is stored in file TEST.TXT  
* under directory called NUM  
*  
* Program: SD-CARD  
* Date   : October, 2018  
* Author : Dogan Ibrahim  
*****/  
#include "SD.h"  
#define SD_CS 5                                // SD card CS pin  
  
void setup()  
{  
    Serial.begin(9600);  
    Serial.println("");                            // New line  
    if(!SD.begin(SD_CS))                         // Initialize SD card  
    {  
        Serial.println("Initialization failed");  
        while(1);  
    }  
    else  
        Serial.println("Initialization done");  
}  
  
//  
// This is the main program. A file called TEST.TXT is created in  
// directory NUM on the SD card. The program stores 10 integer random  
// integer numbers in this file on the SD card  
//  
void loop()  
{  
    File MyFile;  
    unsigned int RandomNumber;  
    char k;  
  
    if(SD.cardType() == CARD_NONE)
```

```

{
    Serial.println("SD card is not found");
    while(1);
}

SD.mkdir("/NUM");                                // Create directory
NUM
MyFile = SD.open("/NUM/TEST.TXT", FILE_WRITE);    // Open file TEST.TXT
if(MyFile)                                       // If file opened
{
    Serial.println("Writing to SD card");          // Write heading
    MyFile.println("Random Numbers");               // Write heading
    MyFile.println("=====");                      // Write heading

    for(k = 0; k < 10; k++)                       // Do 10 times
    {
        RandomNumber = random(0, 255);            // Generate number
        MyFile.println(RandomNumber);              // Save the number
        delay(1000);                            // Wait one second
    }

    MyFile.close();                               // Close the file
    Serial.println("Data written");              // Display message
}
else
    Serial.println("Error in opening file");      // Display message
while(1);                                         // Stop
}

```

Figure 6.67 Program listing

6.12.8 Program Description

The SD card should be formatted before it is used with the ESP32. The steps are as follows:

- Make sure that your SD card is not write protected.
- Insert the SD card in your computer (you may need an SD card adapter).
- Go to **My Computer**.
- Right click on SD card and select **Format**.
- Select **FAT32** and press **Start** to format the card.

At the beginning of the program header file **SD.h** is included in the program and SD card chip select pin (**SD_CS**) is assigned to GPIO pin 5. Inside the setup routine the SD card library is initialised. Notice that the chip select pin must be specified as the argument of library function **SD.begin**. The serial PC USB interface (**Serial.begin**) is also enabled so that messages can be displayed on the PC screen while the program is running.

Inside the main program loop the program checks if there is an SD card in the card module. The program then creates a directory called **NUM** on the SD card using library function **SD.mkdir**. Function **SD.open** creates a new file or opens an existing file. This function has two arguments: the name of the file to be created or opened, and the mode. The mode can either be **FILE_WRITE** or **FILE_READ**. When the **FILE_WRITE** mode is used data is added to the end of the file if the file exists, otherwise a new file is created. Mode **FILE_READ** opens an existing file in read mode. Full path must be given to the file including its directory name. In this program the full path to the file was: **/NUM/TEST.TXT**. Variable **MyFile** is set true if a new file has been created or an existing file is opened. Message **Random Numbers** with an underline is written to the SD card as the heading. A **for** loop is then setup which iterates 10 times. Inside this loop integer random numbers are generated and are stored in the newly created file on the SD card. The program then waits for one second. The file is closed at the end of the **for** loop. Closing the file physically saves the data on the SD card. Notice that the function **SD.flush()** can also be used to save the data physically on the SD card.

Figure 6.68 shows the SD card file structure after the program is run. The data displayed on the PC screen while the program runs (this data is also stored on the SD card) is shown in Figure 6.69.

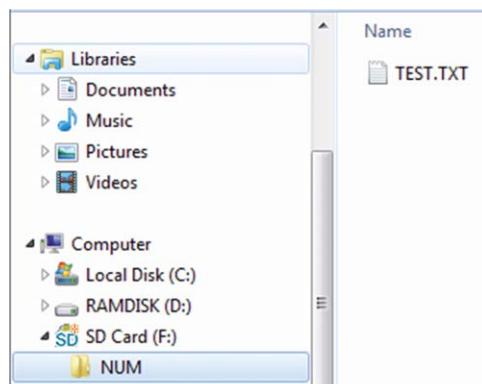


Figure 6.68 SD card file structure

```
Random Numbers
=====
14
165
173
3
39
125
153
38
25
102
```

Figure 6.69 Data displayed on the PC screen

Some of the commonly used SD card library functions are (see website: <https://www.arduino.cc/en/Reference/SD> for full explanation of each function and for additional functions) :

begin():	initialize the SD card library
exists():	check if a file exists on the SD card
mkdir():	create a new directory on the SD card
remove():	delete a file from the SD card
rmdir():	delete a directory on the SD card (the directory must be empty)
name():	returns the filename
available():	check if there is any data available to read from a file
close():	close a file. Closing a file physically saves data on the card
flush():	physically save data on the card
size():	get the size of the file
read():	read from the file
write():	write to the file

6.13 PROJECT 12 – Using SD Cards – Reading from a Card

6.13.1 Description

In this project an SD card module is connected to the ESP32 DevKitC as in the previous project. The program opens file **TEST.TXT** under directory **NUM**, created in the previous project, and displays the contents of this file on the PC screen.

6.13.2 The Aim

The aim of this project is to show how an SD card can be interfaced to the ESP32 DevKitC and how data can be read from the SD card.

6.13.3 Block diagram:

The block diagram of the project is as shown in Figure 6.61.

6.13.4 Circuit Diagram

The circuit diagram of the project is as shown in Figure 6.64.

6.13.6 PDL of the Project

The PDL of the project is shown in Figure 6.70.

```

BEGIN
  Define SD card CS pin
  Initialize SD card library
  Enable USB PC serial port
  Open file TEST.TX in directory NIUM
  DO WHILE there are data in file
    Read data from file
    Display data read on PC screen
  ENDDO
  Close file

```

END*Figure 6.70 PDL of the project*

6.13.7 Program Listing

The program listing of the project is shown in Figure 6.71 (program: **SD-READ**).

```
/*****
*          READ FROM SD CARD
*          =====
*
* In this program a standard SD card module is connected to the
* ESP32 DevKitC. The program opens file TEST.TXT in directory NUM
* on the SD card and then reads and displays the contents (random
* numbers) of this file on the PC screen.
*
* Program: SD-READ
* Date   : October, 2018
* Author : Dogan Ibrahim
*****
#include "SD.h"
#define SD_CS 5                                // SD card CS pin

void setup()
{
    Serial.begin(9600);
    Serial.println("");                         // New line
    if(!SD.begin(SD_CS))                      // Initialize library
    {
        Serial.println("Initialization failed");
        while(1);
    }
    else
        Serial.println("Initialization done");
}

//
// This is the main program. File TEST.TXT is opened in directory
// NUM of the SD card and its contents are displayed on the PC screen
//
void loop()
{
    File MyFile;

    if(SD.cardType() == CARD_NONE)
    {
```

```

        Serial.println("SD card is not found");
        while(1);
    }

MyFile = SD.open("/NUM/TEST.TXT", FILE_READ);           // Open file TEST.TXT
if(MyFile)                                              // If file opened
{
    Serial.println("File opened");                      // Display message
    while(MyFile.available())                           // While there is data
    {
        Serial.write(MyFile.read());                  // Write file contents
    }
    MyFile.close();                                    // Close the file
}
else
{
    Serial.println("Error in opening file");          // Display message
    while(1);                                         // Stop
}

```

Figure 6.71 Program listing

6.13.8 Program Description

In this program the file **TEST.TXT** in directory **NUM** on the SD card is opened in read mode (**FILE_READ**). Then a **while** loop is formed to read all the data from the file and display it on the PC screen. Figure 6.72 shows the output of the program.

```

Initialization done
File opened
Random Numbers
=====
14
165
173
3
39
125
153
38
25
102

```

Figure 6.72 Output of the program

6.14 PROJECT 13 – Infrared Receiver (IR)

6.14.1 Description

Infrared (Ir) communication is usually used in applications such as in television remote control systems and overhead projectors. This is an easy and inexpensive form of short distance communication. Ir light is very similar to visible light, except that it has longer wavelength and therefore it is not detectable to the human eye. Commonly, 38 kHz mod-

ulation technique is used in Ir communication (although other frequencies are also used) where encoded data is sent from the transmitter to the receiver.

In this project an infrared receiver module is connected to DevKitC. Additionally, a relay is connected to one of the GPIO ports. An infrared keypad transmitter is used to control the relay so that pressing a button on the keypad activates the relay. Similarly, pressing another button deactivates the relay.

6.14.2 The Aim

The aim of this project is to show how an infrared receiver can be interfaced to the ESP32 DevKitC and how a relay can be controlled by receiving commands from an infrared transmitter.

6.14.3 Block diagram:

The block diagram of the project is as shown in Figure 6.73.

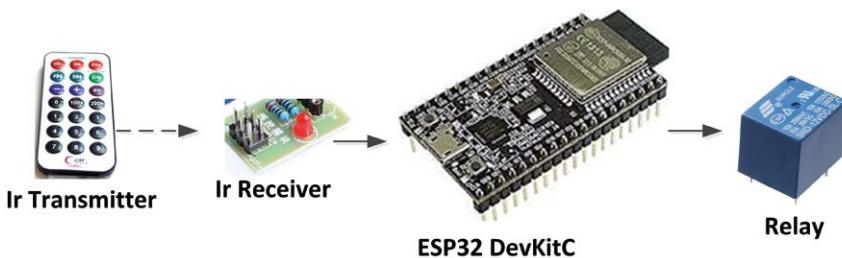


Figure 6.73 Block diagram of the project

6.14.4 Circuit Diagram

In this project the HX1838 infrared transmitter-receiver module is used. This module has the following specifications:

- High sensitivity
- Operating voltage +3.3 V to 5 V
- Digital output
- VCC external 3.3 V-5 V voltage (can be directly connected with the a 5 V microcontroller and 3.3 V microcontroller)
- 10K pull-up resistor on IN pin

The transmitter is a keypad has 21 keys (see Figure 6.74). The receiver module consists of an Ir receiver chip, two resistors and an LED and it has 3 pins: Vcc, Gnd, and IN. Figure 6.75 shows the circuit diagram of the project. Vcc and Gnd pins are connected to the +3.3 V and Gnd pins of the ESP32 respectively. IN pin is connected to GPIO pin 15 of the ESP32 DevKitC. A small relay is connected to GPIO pin 5 of the development board and is activated when the pin is at logic HIGH.



Figure 6.74 Ir transmitter-receiver module

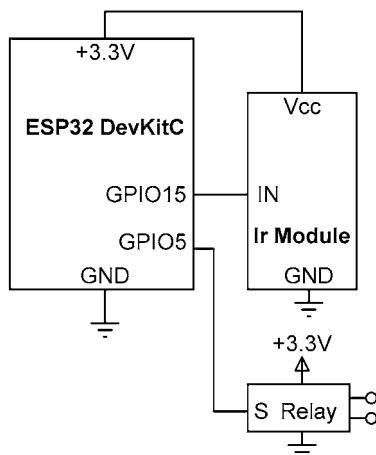


Figure 6.75 Circuit diagram of the project

6.14.5 PDL of the Project

The PDL of the project is shown in Figure 6.76.

```

BEGIN
    Define Ir receiver module and Relay connections
    Configure relay as output and de-activate it
    Enable the Ir receiver module
DO FOREVER
    Receive key-code from Ir transmitter keypad
    IF key code is 16753245 THEN
        Activate the relay
    ELSE IF key code is 16720605 THEN
        De-activate the relay
    ENDIF
ENDDO
END
    
```

Figure 6.76 PDL of the project

6.14.6 Program Listing

In this program we will be using the **IRremote** library by **Ken Shirriff**. The steps to load the library are given below:

- Click the following link to download the IRremote zip library.
<https://randomnerdtutorials.com/arduino-ir-remote-control/>
- Unzip the **.zip** folder to get the folder **IRremote-master**.
- Rename the folder to IRremote.
- Copy the folder to Arduino IDE **libraries** folder.
- Open the Arduino IDE and copy the program given in this project.

Before developing the main program we shall be writing a program to display the key codes returned by the library when the keys on the infrared transmitter keypad are pressed. The program given in Figure 6.77 (program: **IR**) displays the key codes received by the receiver module when a key is pressed on the transmitter keypad.

```
/*********************  
*          INFRARED TRANSMITTER KEY CODES  
*          ======  
*  
* In this program an infrared receiver module is connected to the  
* DevKitC. The program displays the key codes received by the  
* module when a key is pressed on the transmitter keypad  
*  
* Program: IR  
* Date   : October, 2018  
* Author : Dogan Ibrahim  
*****/  
#include "IRremote.h"  
int IN_Pin = 15;                                // IN connected to GPIO15  
  
IRrecv Receiver(IN_Pin);  
decode_results results;  
  
void setup()  
{  
    Serial.begin(9600);  
    Receiver.enableIRIn();                         // Enable the receiver  
}  
  
void loop()
```

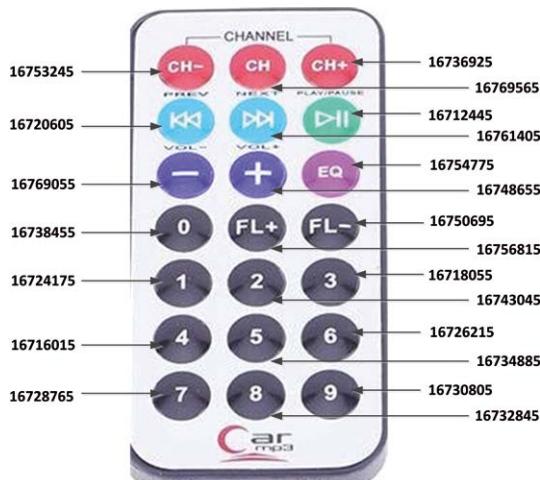
```

{
    if(Receiver.decode(&results))           // Decode teh results
    {
        Serial.println(results.value);      // Display in hexadecimal
        Receiver.resume();                 // Next
    }
    delay(200);
}

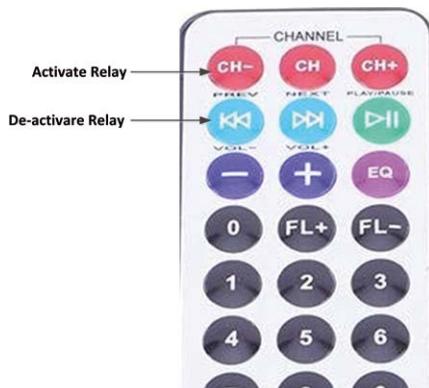
```

Figure 6.77 Program to display the key codes

Figure 6.78 shows the key codes for each key pressed on the transmitter keypad.

*Figure 6.78 The key codes*

In this project pressing the top left button on the transmitter keypad activates the relay. Similarly pressing the second button at the top left de-activates the relay (see Figure 6.79). These buttons have the key codes 16753245 and 16720605 respectively.

*Figure 6.79 Buttons to activate and de-activate the relay*

The main program of the project is shown in Figure 6.80 (program: **IR-RELAY**). At the beginning of the program header file IRremote.h is included in the program, Ir receiver module and relay connections to the DevKitC are defined. Inside the setup routine the relay is configured as an output and is de-activated at the beginning of the program, and the Ir module is enabled. The remainder of the program runs in an endless loop waiting to receive key codes from the Ir transmitter keypad. If the key code 16753245 is received then the relay is activated. If, on the other hand, key code 16720605 is received, then the relay is de-activated.

```
*****  
*          INFRARED RELAY CONTROL  
*          ======  
  
* In this program an infrared receiver module is connected to the  
* DevKitC. Additionally a relay is connected to GPIO port 5. The  
* relay is activated and de-activated by pressing buttons on the  
* transmitter keypad  
  
* Program: IR-RELAY  
* Date   : October, 2018  
* Author : Dogan Ibrahim  
*****/  
  
#include "IRremote.h"  
int IN_Pin = 15;                                // IN connected to GPIO15  
#define Relay 5                                    // Relay on port 5  
#define ON HIGH  
#define OFF LOW  
  
IRrecv Receiver(IN_Pin);  
decode_results results;  
  
void setup()  
{  
    pinMode(Relay, OUTPUT);                         // Relay is output  
    digitalWrite(Relay, LOW);                      // Relay OFF at start  
    Receiver.enableIRIn();                         // Enable the receiver  
}  
  
//  
// This is the main program loop. Read the received Ir code and then  
// control the relay accordingly. Pressing the top left red button  
// activates the relay, and pressing the second button from the top  
// left de-activates the relay  
//  
void loop()  
{
```

```

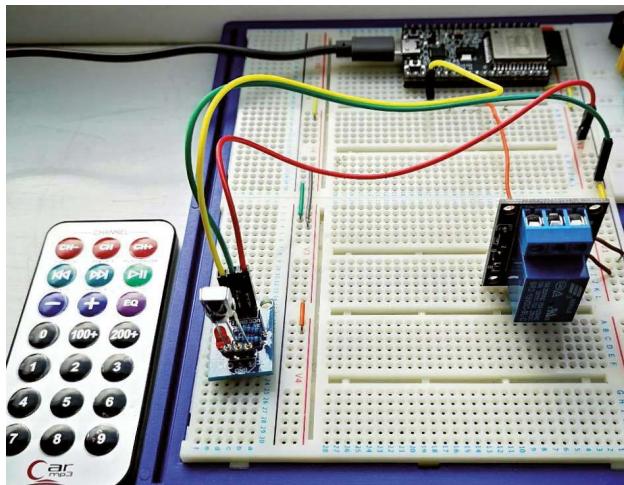
        if(Receiver.decode(&results))           // Decode the results
        {
            if(results.value == 16753245)      // If ON request
                digitalWrite(Relay, ON);       // Activate relay
            else if(results.value == 16720605) // If OFF request
                digitalWrite(Relay, OFF);     // De-activate relay

            Receiver.resume();               // Next
        }
        delay(200);
    }
}

```

Figure 6.80 Program listing

Figure 6.81 shows the project built on a breadboard.

*Figure 6.81 Project built on a breadboard*

6.14.7 Modifications

The program given in Figure 6.81 can be modified and be made more user-friendly if a **switch** statement is used to select the key codes as shown below. The advantage of the following code is that all the key codes can easily be defined with their actions.

```

switch(results.value)
{
    case 16753245:
        digitalWrite(Relay, ON);
        break;
    case 16720605:
        digitalWrite(Relay, OFF);
        break;
    case 16769055
        .....
}

```

```
        break;
case 16738455:
.....
    break;
.....
.....
}
```

Another modification could be to assign names to the keys on the transmitter keypad and then use these names in the program as shown below. Assuming the top 3 keys at the top are labelled from left to right as A B C:

```
#define A 16753245
#define B 16769565
#define C 16736925
.....
.....
switch(results.value)
{
    case A:
        digitalWrite(Relay, ON);
        break;
    case B:
        digitalWrite(Relay, OFF);
        break;
.....
.....
}
```

6.15 Low Power Operation

There are many applications such as in IoT and also in portable applications where batteries are used. The capacity of batteries is limited and we need to save power for extended operations. The ESP32 chip is a fantastic chip with great features. In addition to its processing power and many on-chip modules, it offers low power mode which conserves power and enables the chip to be used for a very long time. It has been reported that the chip can run in low power mode with AA batteries for over several years. Of course this is only achievable with very little use of the Wi-Fi and the Bluetooth modules since these modules consume large amounts of power.

Before looking at the low power modes of the ESP32 chip, it is worthwhile to review the basic on-chip modules of the chip. The ESP32 chip consists of the following:

- Two processing cores: CPU and ULP
- RAM and ROM memory
- Wi-Fi and Bluetooth modules
- Hardware acceleration module

- The RTC module
- Peripheral modules

The RTC module is of interest during low power operation of chip. This module consists of the following:

- Phasor Measurement unit (PMU)
- 32-bit ULP processor
- 8KB of RAM memory

The CPU, Wi-Fi, and the Bluetooth modules require large amounts of power and they should be disabled to conserve power. As described below briefly, the ESP32 chip can be configured in 5 modes as far as the power usage is concerned:

Active mode: This is the normal working mode of the chip, in which all modules of the chip are active. The chip consumes the maximum amount of power in this mode.

Modem Sleep mode: In this mode the Wi-Fi and the Bluetooth modules are OFF; other modules operate normally. This mode is used when it is not required to transmit or receive data via the Wi-Fi or Bluetooth communications.

Light Sleep mode: In this mode the Wi-Fi and the Bluetooth are OFF, and the CPU is in gated pause state.

Deep Sleep mode: This is the most commonly used mode when it is required to conserve power. In this mode the CPU, Wi-Fi, Bluetooth, and the peripheral modules are all OFF. The RTC module and the ULP processor are in ON state (the ULP processor can be shut if desired). In this mode the CPU can be woken up by external or internal timer events to resume operation.

Hibernation mode: In this mode all modules of the chip are in OFF state.

Even more power can be conserved if the ULP processor is programmed correctly in deep sleep mode. This however requires assembly language programming which is beyond the scope of this book. Interested readers can refer to an excellent article by **Tam Hanna**, entitled **ESP32 Low Power - Programming the ULP coprocessor**, which appeared in the March 7 April 2018 issue of the **Elektor** magazine (pages 32-38).

In this section we are only interested in the Deep Sleep mode. Interested readers can obtain further details on power saving modes from the following document:

Sleep Modes – ESP-IDF Programming Guide v3.2-beta1-15-g12b2268 documentation

6.15.1 Deep Sleep Mode

It is reported by the manufacturers that the ESP32 chip consumes around 0.15 mA of current in deep sleep mode. We shall see later in this section the actual measured power consumption.

When the chip is in deep sleep mode, it can be woken up by one of the following 5 trigger signals or events:

- External signal using RTC_IO
- External signal using RTC_CNTL
- Timer
- Touchpad
- ULP program

Remember that in deep sleep mode the CPU core, main memory, Wi-Fi module, Bluetooth module, and the peripherals are all disabled and so everything stored in the main memory are lost. The only module that works is the RTC module and its memory. Therefore, if we want to save data to survive the deep sleep mode then this data must be saved in the RTC memory. Some example projects are given in the next sections to show how the deep sleep mode can be used.

6.16 PROJECT 14 – Using Timer to Wake Up

6.16.1 Description

In this project an LED is connected to one of the GPIO ports of ESP32 DevKitC through a current limiting resistor. ESP32 is put into deep sleep mode for 15 seconds and when it wakes up a counter in the RTC memory is incremented by one. The LED is then flashed at a rate of one second where the number of flashes depends on the value of this variable. Therefore, every time the ESP32 wakes up it flashes one more time.

6.16.2 The Aim

The aim of this project is to show how the ESP32 can be put into deep sleep mode in order to conserve power, and then how it can be woken up by the timer.

6.16.3 Block diagram:

The block diagram of the project is as in Figure 4.1.

6.16.4 Circuit Diagram

The circuit diagram of the project is as in Figure 4.3 where the LED is connected to GPIO port 23 through a 330 ohm current limiting resistor.

6.16.5 PDL of the Project

The PDL of the project is shown in Figure 6.82.

```

BEGIN
    Define variable BootCount in RTC memory
    Define LED at port 23
    Configure LED as output
    Set BootCount to 0
    Increment BootCount
    DO FOR BootCount
        Turn ON LED
        Wait one second
        Turn OFF LED
        Wait for one second
    ENDDO
    Configure Deep Sleep wake up time as 15 seconds
    Start Deep Sleep mode
END

```

Figure 6.82 PDL of the project

6.16.6 Program Listing

The program listing of the project is shown in Figure 6.83 (program: **DSLEEP**).

```

/*****
*          DEEP SLEEP WITH LED
* =====
*
* In this program an LED is connected to GPIO port 23 of the ESP32
* DevKitC through a 330 ohm resistor. The processor is put
* into deep sleep mode for 15 seconds and is woken up by the timer.
* The LED then flashes at a rate of one second where the
* number of flashes depends on the number of times the processor
* is woken up. For example, the LED flashes once initially, 2 times
* after 15 seconds, 3 times after 30 seconds, 4 times after 45
* seconds and so on.
*
* Program: DSLEEP
* Date   : November, 2018
* Author : Dogan Ibrahim
*****
#define us_TO_Seconds 1000000           // us to seconds
#define TIME_TO_SLEEP 15                // 15 seconds
RTC_DATA_ATTR int BootCount = 0;       // BootCount

#define LED 23                          // LED on port 23

void setup()
{

```

```
pinMode(LED, OUTPUT);                                // LED is output
BootCount++;                                         // Increment count

for(int k = 0; k < BootCount; k++)
{
    digitalWrite(LED, HIGH);                         // LED ON
    delay(1000);                                    // Wait one second
    digitalWrite(LED, LOW);                          // LED OFF
    delay(1000);                                    // wait one second
}

// Now put the processor into Deep Sleep mode with timer wake up
// time of 15 seconds, and start the Deep Sleep
//
esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * us_TO_Seconds);
esp_deep_sleep_start();
}

// We never execute the following code
//
void loop()
{

}
```

Figure 6.83 Program listing

6.16.7 Program Description

Function **esp_sleep_enable_timer_wakeup** has one argument which specifies the time to wake up the processor in microseconds. At the beginning of the program the conversion factor from microseconds to seconds is defined as 1,000,000 in **us_TO_Seconds**. Time to sleep is defined as 15 seconds in **TIME_TO_SLEEP**. Variable **BootCount** is defined in the RTC module memory (using **RTC_DATA_ATTR**) and is initialised to zero. Also, the LED is assigned to GPIO port 23. The remainder of the program runs entirely in the setup routine. Here, the LED is configured as an output and variable **BootCount** is incremented by one. The LED is then flashed at a rate of one second where variable **BootCount** specifies how many times it will flash. Timer wake up from deep sleep is then set to 15 seconds and the processor is put into deep sleep using function **esp_deep_sleep_start**. The processor will now go into deep sleep and wake up after 15 seconds. On wake up the setup routine is executed where variable **BootCount** is incremented and the LED is flashed accordingly. The processor is put back into deep sleep mode and this process is repeated. Notice that the **loop** function is never executed.

An ammeter was placed in series with the USB power cable (by breaking up the cable) and the following current measurements were taken during the program execution:

Current consumption with the LED continuously OFF: 12mA
 Current consumption with the LED continuously ON: 20mA
 Current consumption when the processor is in Deep Sleep: 0.01mA

In some applications we may want to disable just the Bluetooth module. This can be done by including the following header files and the program statements:

```
#include "esp_bt.h"
#include "esp_bt_main.h"

esp_bt_controller_disable();
esp_bluetooth_disable();
```

Similarly, the Wi-Fi module can be disabled by the following program statements:

```
#include "esp_wifi.h"
esp_wifi_stop();
```

Additionally, all of the RTC peripherals can be turned OFF by the following program statements:

```
#include "esp_deep_sleep.h"
esp_deep_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH,ESP_PD_OPTION_OFF);
```

6.16.8 Wake Up Reason

Sometimes we may want to know the reason for the wake up. The reason for wake up is one of the cases described in Section 6.15.1. This can be obtained by calling to function **esp_sleep_get_wakeup_cause** which returns an integer between 1 and 5 where 1: RTC_IO, 2: RTC_CNTL, 3: Timer, 4: Touchpad, 5: ULP program. Program **SREASON** in Figure 6.84 shows how the wake up reason can be displayed on the PC screen. Here a **switch** statement is used and messages are displayed depending on the reason for the wake up. A variable of type **esp_sleep_wakeup_cause_t** must be defined before calling the function to return the reason for wake up. In this example the text **Wakeup cause: Timer** is displayed on the PC screen.

```
/*
 *          DEEP SLEEP AND DISPLAY THE REASON FOR WAKE UP
 */
*
* This program is same as program DSLEEP but here the reason for wake
* up is displayed on the PC screen
*
* Program: SREASON
* Date   : November, 2018
* Author : Dogan Ibrahim
*/

```

```
#define us_TO_Seconds 1000000          // us to seconds
#define TIME_TO_SLEEP 15                // 15 seconds
RTC_DATA_ATTR int BootCount = 0;       // BootCount

#define LED 23                         // LED on port 23

void Display_Wake_Up_Reason()
{
    esp_sleep_wakeup_cause_t Reason_For_Wakeup;

    Reason_For_Wakeup = esp_sleep_get_wakeup_cause();
    switch(Reason_For_Wakeup)
    {
        case 1:
            Serial.println("Wakeup cause: RTC_IO");
            break;
        case 2:
            Serial.println("Wakeup cause: RTC_CNTL");
            break;
        case 3:
            Serial.println("Wakeup cause: Timer");
            break;
        case 4:
            Serial.println("Wakeup cause: Touchpad");
            break;
        case 5:
            Serial.println("Wakeup cause: ULP program");
    }
}

void setup()
{
    Serial.begin(9600);
    pinMode(LED, OUTPUT);             // LED is output
    BootCount++;                     // Increment count
    Display_Wake_Up_Reason();

    for(int k = 0; k < BootCount; k++)
    {
        digitalWrite(LED, HIGH);      // LED ON
        delay(1000);                // Wait one second
        digitalWrite(LED, LOW);       // LED OFF
        delay(1000);                // wait one second
    }
}

// Now put the processor into Deep Sleep mode with timer wake up
```

```

// time of 15 seconds, and start the Deep Sleep
//
esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * us_TO_Seconds);
esp_deep_sleep_start();
}

//
// We never execute the following code
//
void loop()
{
}

```

Figure 6.84 Displaying the reason for wake up

6.17 PROJECT 15 – Using Touchpad to Wake Up

6.17.1 Description

In this project an LED is connected to one of the GPIO ports of ESP32 DevKitC through a current limiting resistor as in the previous example. ESP32 is put into deep sleep mode and is woken up when touch is detected on touchpad input T0, i.e. GPIO 4 (see Section 5.15 on how the touchpad works). The LED is then flashed at a rate of one second where the number of flashes depends on the value of this variable. Therefore, every time the ESP32 wakes up it flashes one more time.

6.17.2 The Aim

The aim of this project is to show how the ESP32 can be put into deep sleep mode in order to conserve power, and then how it can be woken up by a touchpad input pin.

6.17.3 Block diagram:

The block diagram of the project is as in Figure 4.1.

6.17.4 Circuit Diagram

The circuit diagram of the project is as in Figure 4.3 where the LED is connected to GPIO port 23 through a 330 ohm current limiting resistor. Additionally, a wire is connected to GPIO pin 4 to be used as the touchpad input.

6.17.5 PDL of the Project

The PDL of the project is shown in Figure 6.85.

```

BEGIN
Define variable BootCount in RTC memory
Define LED at port 23
Configure LED as output
Set BootCount to 0
Increment BootCount

```

```
DO FOR BootCount
    Turn ON LED
    Wait one second
    Turn OFF LED
    Wait for one second
ENDDO
Attach T0 (GPIO pin 4) to interrupt callback
Configure Deep Sleep with touchpad
Start Deep Sleep mode
END
```

Figure 6.85 PDL of the project

6.17.6 Program Listing

The program listing of the project is shown in Figure 6.86 (program: **STOUCH**).

```
/*********************************************
*          DEEP SLEEP AND WAKE UP BY TOUCH
* =====
*
* This program is same as program DSLEEP but here touchpad GPIO input
* 4 is used to wake up the processor. On wake up, variable BootCount
* is incremented by one and the LED flashes where the number of flashes
* is equal to the value of BootCount.
*
* Program: STOUCH
* Date   : November, 2018
* Author : Dogan Ibrahim
*****
RTC_DATA_ATTR int BootCount = 0;           // BootCount

#define LED 23                                // LED on port 23
#define Threshold 50                           // Touch sensitivity

void Callback()
{
    //not used
}

void setup()
{
    pinMode(LED, OUTPUT);                    // LED is output
    BootCount++;                            // Increment count

    for(int k = 0; k < BootCount; k++)
    {
```

```

        digitalWrite(LED, HIGH);           // LED ON
        delay(1000);                     // Wait one second
        digitalWrite(LED, LOW);          // LED OFF
        delay(1000);                     // wait one second
    }
}

// Now put the processor into Deep Sleep mode and wake up by touching
// to GPIO pin 4 (touchpad T0)
//
touchAttachInterrupt(T0,Callback,Threshold);
esp_sleep_enable_touchpad_wakeup();
esp_deep_sleep_start();
}

// We never execute the following code
//
void loop()
{
}

```

Figure 6.86 Program listing

6.17.7 Program Description

At the beginning of the program variable **BootCount** is initialised to 0 using the RTC memory. The LED is then assigned to GPIO port 23 and the touch sensitivity is set to 50 (higher values give more sensitivity). Inside the setup routine variable **BootCount** is incremented by one and the LED is flashed based on this count. Interrupt is then attached to Touchpad T0 (GPIO pin 4) with function **Callback**. Touchpad wake up is set by calling function **esp_sleep_enable_touchpad_wakeup()**. Calling function **esp_deep_sleep_start** puts the processor into deep sleep mode. The processor wakes up when we touch the GPIO pin 4 and the **setup** routine is then executed where variable **BootCount** is incremented by one and the LED flashes based on this count. Notice that we can call function **esp_sleep_get_touchpad_wakeup_status** to find out which touchpad input caused the wake up, as shown in the following program code:

```

touch_pad_t PinTouched;
.....
PinTouched = esp_sleep_get_touchpad_wakeup_status();
switch(PinTouched)
{
    case 0:
        Serial.println("T0, GPIO 4 is touched");
        break;
    case 1:
        Serial.println("T1, GPIO 0 is touched");
}

```

```
        break;  
.....  
.....  
case 9:  
    Serial.println("T9, GPIO 32 is touched");  
    break;  
}
```

6.18 PROJECT 16 – Using External Trigger to Wake Up

6.18.1 Description

In this project an LED is connected to one of the GPIO ports of ESP32 DevKitC through a current limiting resistor as in the previous two examples. ESP32 is put into deep sleep mode and is woken up when GPIO pin 4 is pulled LOW. The LED is then flashed at a rate of one second where the number of flashes depends on the value of this variable. Therefore, every time the ESP32 wakes up it flashes one more time.

Notice that when using external trigger to wake up from deep sleep only the RTC input-output pins can be used. These are: 0, 2, 4, 12, 13, 14, 15, 25, 26, 27, 32, 33, 34, 35, 36, 37, 38, 39.

6.18.2 The Aim

The aim of this project is to show how the ESP32 can be put into deep sleep mode in order to conserve power, and then how it can be woken up by an external trigger (GPIO pin 4 going LOW in this example).

6.18.3 Block diagram:

The block diagram of the project is as in Figure 6.87.



Figure 6.87 Block diagram of the project

6.18.4 Circuit Diagram

The circuit diagram of the project is shown in Figure 6.88. An LED is connected to GPIO port 23 through a 330 ohm current limiting resistor. Additionally, a push button switch is connected to GPIO pin 4 through a 10K resistor. This pin is normally at logic LOW and goes to logic HIGH when the button is pressed.

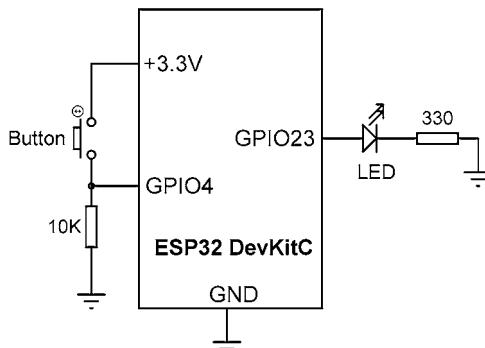


Figure 6.88 Circuit diagram of the project

6.18.5 PDL of the Project

The PDL of the project is shown in Figure 6.89.

```

BEGIN
    Initiaize variable BootCount in RTC memory
    Assign GPIO port 23 to LED
    Increment BootCount
    DO FOR BootCount
        Turn ON LED
        Wait one second
        Turn OFF LED
        Wait one second
    ENDDO
    Configure Deep Sleep with external trigger
    Start Deep Sleep
END

```

Figure 6.89 PDL of the project

6.18.6 Program Listing

The program listing (program: **TSLEEP**) is shown in Figure 6.90.

```

/*****
*          DEEP SLEEP AND WAKE UP BY EXTERNAL TRIGGER
* =====
*
* In this program an LED is conencted to GPIO pin 23 as in the
* previous two examples. Additionally, a push button switch is
* connected to GPIO pin 4 where this pin is normally at logic LOW.
* Pushing the button forces pin 4 to logic HIGH and as a result
* the processor wakes up
*
* Program: TSLEEP
* Date   : November, 2018
*/

```

```
* Author : Dogan Ibrahim
*****
RTC_DATA_ATTR int BootCount = 0;                                // BootCount

#define LED 23                                         // LED on port 23

void setup()
{
    pinMode(LED, OUTPUT);                            // LED is output
    BootCount++;                                    // Increment count

    for(int k = 0; k < BootCount; k++)
    {
        digitalWrite(LED, HIGH);                     // LED ON
        delay(1000);                               // Wait one second
        digitalWrite(LED, LOW);                     // LED OFF
        delay(1000);                               // wait one second
    }
}

// Now put the processor into Deep Sleep mode and wake up by pushing
// button at GPIO pin 4. GPIO 4 is normally at logic LOW and pushing
// the button forces the pin to logic HIGH
//
esp_sleep_enable_ext0_wakeup(GPIO_NUM_4, 1);
esp_deep_sleep_start();
}

// We never execute the following code
//
void loop()
{

}
```

Figure 6.90 Program listing

6.18.7 Program Description

At the beginning of the program variable **BootCount** is initialised to 0 in the RTC memory, and the LED is assigned to GPIO port 23. Inside the setup routine the processor is put into deep sleep mode by calling function **esp_sleep_enable_ext0_wakeup(GPIO_NUM_4, 1)** where GPIO pin 4 is specified as the wake up pin, and the processor is configured to wake up when the pin goes to logic HIGH. The processor is then put into deep sleep mode by calling function **esp_deep_sleep_start** as before. When the button is pressed the processor will wake up and execute the setup routine. Inside this routine variable **BootCount** will be incremented by one and the LED will flash depending on the count.

Notice that there are two types of external triggers: **ext0** and **ext1**. **ext0** uses the RTC peripherals and therefore requires these peripherals to be kept powered during deep sleep. Because RTC peripherals are enabled in this mode, internal pull-up or pull-down resistors can be used. These resistors can be configured by calling functions **rtc_gpio_pullup_en** or **rtc_gpio_pulldown_en**.

ext1 uses the RTC controller and therefore does not need the peripherals to be powered on. The wake up source is implemented by the RTC controller and therefore the RTC peripherals and the RTC memory can be powered down to conserve more power, but the internal pull-up or pull-down resistors will be disabled. The selected pin can be used as a trigger by specifying the following options in function **esp_sleep_enable_ext1_wakeup**:

- Wake up if the selected pin is HIGH: **ESP_EXT1_WAKEUP_ANY_HIGH**.
- Wake up if all the selected pins are LOW: **ESP_EXT1_WAKEUP_ALL_LOW**.

In order to use the internal pull-up or pull-down resistors, we have to keep power to the RTC peripherals during sleep. This can be done using the following function:

```
esp-sleep-pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_ON)
```

Internal pull-ups can be disabled by calling function:

```
gpio_pullup_dis(GPIO_NUM)
```

or, internal pull-downs can be enabled by calling function:

```
gpio_pulldown_en(GPIO_NUM)
```

In ext1 mode the following function is used to enable deep sleep (assuming wake up is requested when any pin is HIGH):

```
esp_sleep_enable_ext1_wakeup(Bitmask, ESP_EXT1_WAKEUP_ANY_HIGH)
```

where **Bitmask** is defined as the power of the pin with base 2. e.g. if GPIO pin 4 is used then **Bitmask** will be 0x10 (i.e. 2^4). Similarly, for GPIO 32, **Bitmask** will be 0x100000000 (i.e. 232).

Figure 6.91 shows the program listing (program: **TSLEEP-2**) using the ext1 mode of external trigger.

```
/*
 *          DEEP SLEEP AND WAKE UP BY EXTERNAL TRIGGER
 *          =====
 *
 * In this program ext1 mode of RTC_IO wake up is used
 *
 * Program: TSLEEP-2
 * Date   : November, 2018
 * Author : Dogan Ibrahim
 *****/
RTC_DATA_ATTR int BootCount = 0;           // BootCount
#define Bitmask 0x10                         // GPIO pin 4

#define LED 23                                // LED on port 23

void setup()
{
    pinMode(LED, OUTPUT);                  // LED is output
    BootCount++;                           // Increment count

    for(int k = 0; k < BootCount; k++)
    {
        digitalWrite(LED, HIGH);           // LED ON
        delay(1000);                     // Wait one second
        digitalWrite(LED, LOW);          // LED OFF
        delay(1000);                     // wait one second
    }
    //
    // Now put the processor into Deep Sleep mode and wake up by pushing
    // button at GPIO pin 4. GPIO 4 is normally at logic LOW and pushing
    // the button forces the pin to logic HIGH
    //
    esp_sleep_enable_ext1_wakeup(Bitmask, ESP_EXT1_WAKEUP_ANY_HIGH);
    esp_deep_sleep_start();
}

//
// We never execute the following code
//
void loop()
{
```

Figure 6.91 Using ext1 mode of external trigger

6.19 Summary

In this chapter we have looked at the design of more complex projects using the ESP32 DevKitC with the Arduino IDE programming environment. In the next chapter we shall be looking at the design of network-based applications again using the Arduino IDE environment for the ESP32 DevKitC.

Chapter 7 • ESP32 DevKitC network programming using the Arduino IDE

7.1 Overview

In last chapter we have seen the development of more complex ESP32 Devkit projects using the Arduino IDE as the programming environment.

Wi-Fi capability is one of the powerful features and advantages of the ESP32 processor. In this chapter we shall be looking at the Wi-Fi network applications of the ESP32 DevKitC, again using the Arduino IDE environment.

7.2 Scanning the Surrounding Wi-Fi Networks

In some applications we may want to scan the surrounding environment in order to find out about the nearby Wi-Fi networks. Function **WiFi.scanNetworks()** is used to scan the surrounding networks. If any networks are found, this function returns the following parameters:

- Total number of networks found
- Network names (SSID)
- Network signal strengths (RSSI)
- MAC addresses (BSSIDstr)
- Encryption types

Figure 7.1 shows a program (program: SCAN) that scans the surrounding Wi-Fi networks and returns the various network parameters.

```
*****  
*           SCAN WI-FI NETWORK  
*           ======  
*  
*   This program does a scan and displays the surrounding Wi-Fi  
*   networks. The following parameters of the found networks  
*   are displayed: Network name (SSID), signal strength (RSSI),  
*   and MAC address (BSSIDstr).  
*  
*   File:   SCAN  
*   Author: Dogan Ibrahim  
*   Date:   November, 2018  
*  
*****/  
#include "WiFi.h"  
  
//  
// Set mode to station, scan the surrounding Wi-Fi networks and  
// display their details on the Serial Monitor  
//
```

```
void setup()
{
    Serial.begin(9600);
    WiFi.mode(WIFI_STA); // Mode station

    int N = WiFi.scanNetworks(); // Scan networks
    if(N > 0) // If networks found
    {
        Serial.println("");
        Serial.print(N);
        Serial.println(" Networks found");
        for (int i = 0; i < N; ++i)
        {
            Serial.print(i + 1);
            Serial.print(": ");
            Serial.print("Network Name: ");
            Serial.print(WiFi.SSID(i));
            Serial.print(" ");
            Serial.print("Signal Strength: ");
            Serial.print(WiFi.RSSI(i));
            Serial.print(" ");
            Serial.print("MAC Address: ");
            Serial.println(WiFi.BSSIDstr(i));
            delay(50);
        }
    }
}

void loop()
{
    // no code
}
```

Figure 7.1 Wi-Fi network scan program

In the above program the returned data is displayed on the Serial Monitor. A sample output is shown in Figure 7.2 (only part of the output is shown here).

```

17 Networks found
1: Network Name: Chromecast7872.b Signal Strength: -43 MAC Address: FA:8F:CA:56:8D:3F
2: Network Name: BTWifi-with-FON Signal Strength: -51 MAC Address: CA:91:F9:66:5A:B1
3: Network Name: BTHomeSpot-XNH Signal Strength: -56 MAC Address: 00:07:26:C7:AC:20
4: Network Name: BTWifi-X Signal Strength: -60 MAC Address: EA:91:F9:66:5A:B1
5: Network Name: HP-Print-19-LaserJet 200 color Signal Strength: -66 MAC Address: 9C:D2
6: Network Name: BTOpenzone-B Signal Strength: -69 MAC Address: 12:81:D8:78:27:BA
7: Network Name: BTWiFi Signal Strength: -74 MAC Address: 02:81:D8:78:27:BA
8: Network Name: BTHub3-MFK8 Signal Strength: -75 MAC Address: 00:81:D8:78:27:BA
9: Network Name: BTHomeSpot-XNH Signal Strength: -76 MAC Address: 00:07:26:C7:AC:24
10: Network Name: DIRECT-CC-HP OfficeJet Pro 6960 Signal Strength: -81 MAC Address: 30:1
11: Network Name: BTHub4-GMP9 Signal Strength: -83 MAC Address: 44:E9:DD:5D:1D:90
12: Network Name: BTWifi-with-FON Signal Strength: -84 MAC Address: 44:E9:DD:5D:1D:93
13: Network Name: VM4929888 Signal Strength: -89 MAC Address: 84:16:F9:65:FF:00

```

Figure 7.2 Sample output from the program

7.3 Connecting to a Wi-Fi Network

Connecting to a Wi-Fi network is easy. We have to specify the network name and the password. Then, calling function WiFi.begin with the specified network name and password as the arguments should connect the ESP32 to the required network. It is always a good idea to check Wi-Fi status to make sure that the connection has been successful. The constant WL_CONNECTED can be used for checking the Wi-Fi status for a connection.

Figure 7.3 shows a program (program: CONNECT) that connects to the local Wi-Fi network with the name BTHomeSpot-XNH. In this program the connection status is displayed on the Serial Monitor as shown in Figure 7.4.

```

/*****
 *          CONNECT TO A WI-FI NETWORK
 * =====
 *
 * This program connect to the local Wi-Fi network with the
 * name BTHomeSpot-XNH. The conenction status is displayed on
 * the Serial Monitor.
 *
 *
 * File:    CONNECT
 * Author:  Dogan Ibrahim
 * Date:   November, 2018
 *
 *****/
#include "WiFi.h"
const char* ssid = "BTHomeSpot-XNH";
const char* password = "49315abseb";

// Set mode to station, scan the surrounding Wi-Fi networks and
// display their details on the Serial Monitor
//
```

```

void setup()
{
    Serial.begin(9600);
    Serial.println("");
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
    {
        Serial.println("Attempting to connect...");
        delay(1000);
    }
    Serial.println("Connected to Wi-Fi network");
}

void loop()
{
    // no code
}

```

Figure 7.3 Connecting to a Wi-Fi network

```

Attempting to connect...
Attempting to connect...
Attempting to connect...
Attempting to connect...
Connected to Wi-Fi network

```

Figure 7.4 Displaying the connection status

The local IP address and the MAC address can easily be obtained after a connection is made to a Wi-Fi network. Function WiFi.localIP() returns the local IP address. Similarly, function WiFi.macAddress() returns the MAC address as shown in the program (CONNECT2) in Figure 7.5. Figure 7.6 shows a sample output from the program.

```

*****
*           CONNECT TO A WI-FI NETWORK
*           =====
*
*   This program connects to the local Wi-Fi network with the
*   name BTHomeSpot-XNH. The connection status is displayed on
*   the Serial Monitor. In addition, the local IP address and the
*   MAC address are displayed. The program disconnects from the
*   network before stopping
*
*
*   File:   CONNECT2
*   Author: Dogan Ibrahim
*   Date:   November, 2018

```

```
*  
*****  
#include "WiFi.h"  
const char* ssid = "BTHomeSpot-XNH";  
const char* password = "49341abseb";  
  
//  
// Set mode to station, scan the surrounding Wi-Fi networks and  
// display their details on the Serial Monitor  
//  
void setup()  
{  
    Serial.begin(9600);  
    Serial.println("");  
    WiFi.begin(ssid, password);  
    while(WiFi.status() != WL_CONNECTED)  
    {  
        Serial.println("Attempting to connect...");  
        delay(1000);  
    }  
    Serial.println("Connected to Wi-Fi network");  
//  
// Display the local IP address and the MAC address, and then  
// disconnect from the Wi-Fi  
//  
    Serial.print("IP address: ");  
    Serial.println(WiFi.localIP());  
    Serial.print("Mac address: ");  
    Serial.println(WiFi.macAddress());  
    WiFi.disconnect(true);  
}  
  
void loop()  
{  
    // no code  
}
```

Figure 7.5 Program to return the IP address and the MAC address

```
Attempting to connect...  
Attempting to connect...  
Attempting to connect...  
Connected to Wi-Fi network  
IP address: 192.168.1.156  
Mac address: 30:AE:A4:05:5B:E0
```

Figure 7.6 Sample output]

7.4 HTTP GET Requests

The HTTP (HyperText Transfer Protocol) is designed to enable communication between client servers on a network. There are several operations that can be performed with the HTTP, such as GET, PUT, LOAD, DELETE, HEAD, TRACE etc. The GET method is used to retrieve information from a web server where the URL of the server is specified. The information is only retrieved and the contents of the server are not changed.

Figure 7.7 shows a program (program: HTTPGet) that retrieves information from the website <http://httpbin.org/ip>. This website returns the local IP address of the caller. At the beginning of the program the Wi-Fi and the HTTP libraries are included in the program. The program then connects to the local WiFi in the setup routine. Inside the main program, the object `HTTPClient` is declared. Then the URL of the website is specified using the `http.begin` statement. The return code of the HTTP GET request is stored in variable `HttpRetCode`. Returned data is stored in variable `Contents` and is then displayed on the Serial Monitor together with the HTTP return code.

```
/*
 *          HTTP GET EXAMPLE
 *=====
 *
 *  This is an example of using the HTTP GET service to retrieve
 *  information from a web site. In this example the web site:
 *  http://httpbin.org/ip is used as an example, where this URL
 *  returns the local IP address. The retrieved information is
 *  displayed on the Serial Monitor together with the HTTP return
 *  code
 *
 *  File:    HTTPGet
 *  Author: Dogan Ibrahim
 *  Date:   November, 2018
 *
 */
#include "WiFi.h"
#include <HTTPClient.h>

const char* ssid = "BTHomeSpot-XNH";
const char* password = "49345abseb";

//
// Set mode to station, and connect to teh local Wi-Fi
//
void setup()
{
    Serial.begin(9600);
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
```

```
{  
    Serial.println("Attempting to connect...");  
    delay(1000);  
}  
Serial.println("Connected to local Wi-Fi");  
  
}  
  
//  
// Retrieve information from site:  
// http://httpbin.org/ip and display on the Serial Monitor  
//  
void loop()  
{  
    HTTPClient http;  
    http.begin("http://httpbin.org/ip");           // Specify site  
    int HttpRetCode = http.GET();                  // HTTP GET request  
  
    if(HttpRetCode > 0)                          // If data returned  
    {  
        Serial.println("Received data...");        // Display the data  
        String Contents = http.getString();  
        Serial.println(HttpRetCode);                // Display the code  
        Serial.println(Contents);                   // Display the contents  
        http.end();                                // End the connection  
    }  
}
```

Figure 7.7 HTTP Get Request program

Figure 7.8 shows the output on the Serial Monitor. Notice that in this example the return code is 200 which corresponds to success. Some of the other important return codes are:

- 200: Success
- 202: The request has been accepted but the processing is not complete yet
- 206: The request must include a range header field to indicate the desired range
- 301: The requested resource has been assigned a new permanent URL and any future references should use the returned URL
- 404: No matching URL
- 408: Request timeout
- 409: There is a conflict with the resource
- 414: The URL is too long
- 500: Internal server error
- 505: The HTTP version is not supported

```

Attempting to connect...
Attempting to connect...
Connected to local Wi-Fi
Received data...
200
{
  "origin": "109.159.164.148"
}

Received data...
200
{
  "origin": "109.159.164.148"
}

```

Figure 7.8 Output from the program

7.5 Using the Socket Library

Socket library is mainly used in network-based communications. UDP and TCP/IP are the two most commonly used protocols to send and receive data over a network. TCP/IP is a reliable protocol which includes handshaking and thus guarantees the delivery of packets to the required destination. UDP on the other had is not so reliable, but is a fast protocol.

Table 7.1 shows a comparison of the UDP and TCP/IP type communications.

TCP/IP	UDP
Connection oriented protocol	Connectionless protocol
Slow	Fast
Highly reliable data transmission	Not so reliable
Packets arranged in order	No ordering of packets
20 byte header size	8 byte header size
Error checking and re-transmission	No error checking
Data received acknowledgement	No acknowledgement
Used in HTTP, HTTPS, FTP etc	Used in DNS, DHCP, TFTP etc

Table 7.1 Comparison of UDP and TCP/IP

UDP and TCP/IP protocol based programs are server-client based where one node sends data and the other node receives it and vice-versa. Data is transferred through ports where the server and the clients must use the same port numbers.

In the next sections we shall be looking at how UDP and TCP/IP protocol based programs can be written using for the ESP32 DevKitC, using the Arduino IDE as the program development environment.

7.5.1 UDP Programs

UDP is a connectionless protocol and thus there is no need to make a connection to the destination node before a packet can be sent to it. The communication between a server and a client are basically as follows:

Server

- Define the destination node IP address and port number.
- Construct the data packet.
- Create a socket.
- Bind the socket to the local port.
- Receive data from the client.
- Send data to the client.
- Close the connection.

Client

- Define the destination node IP address and port number.
- Create a socket.
- Send data to the server.
- Receive data from the server.
- Close the connection.

Notice that both the server and the client can send and receive data packets from each other. An example is given below to show how data can be sent and received by a UDP program.

Example 7.1

Write a UDP program to receive a text message from a UDP client and display the message on the Serial Monitor. Then, send the message Hello From the Server to the client. Assume the following IP addresses and port numbers for the server and the client:

Server (ESP32 DevKitC) IP address:	192.168.1.156
Client (PC) IP address:	192.168.1.71

In this example, the ESP32 DevKitC is the UDP Server and the PC is the UDP Client. The local port number is set to 5000.

Solution 7.1

Figure 7.9 shows the test system setup with the IP addresses and the port numbers. Notice that we need another program to test our program. Instead of writing another ESP32 program, we can use a UDP packet program available for the PC that can send and receive UDP and TCP/IP packets. One such program is the Packet Sender (available free of charge at the link: <https://packetsender.com/download>).

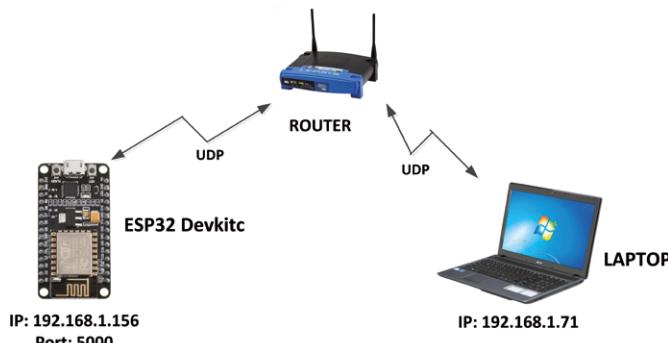


Figure 7.9 Test system setup

We will now use the **Packet Sender** program to send and receive UDP packets from our ESP32 DevKitC program.

To use the program, download and extract it to a folder, then double click on file **PacketSender** to start the program. Give a name to your test (e.g. Test packet), enter the message to be sent to our DevKitC server program (e.g. **Hello from the client**), enter the destination (DevKitC here) IP address, enter the port number, select UDP as the protocol (see Figure 7.10).

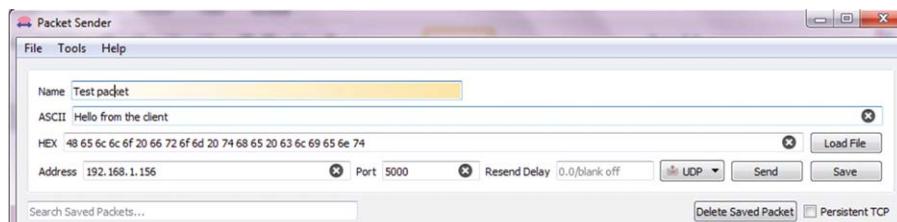


Figure 7.10 Configure the PacketSender program

The ESSP32 DevKitC UDP program is shown in Figure 7.11 (program: **UDP**). At the beginning of the program libraries WiFi and WiFiUDP are included in the program, the network name and the password of the local Wi-Fi router are specified, and the local port number and the remote IP address are specified. Inside the setup routine connection is made to the local network and the program is set to listen for a connection.

Inside the main program the received packet is parsed and the parameters such as the packet length, remote IP address, and remote port number are extracted and displayed on the Serial Monitor. The program then reads and displays the contents of the received data packet (**Hello from the client**). Then the **udp.printf** is used to send the message **Hello from the server** to the client. The program then closes the connection.

```
/*****
 *          UDP SERVER EXAMPLE
 *          =====
 *
 * This is an UDP server example. The program sends the message
 * "Hello from the Server" to an UDP client. The IP addresses
 * and port numbers of the server and the client are:
 *
 * Server (ESP32 Devkitc) IP address: 192.168.1.156 Port: 5000
 * Client (PC) IP address:           192.168.1.71
 *
 *
 * File:    UDP
 * Author:  Dogan Ibrahim
 * Date:   November, 2018
 *
 *****/
#include "WiFi.h"
#include <WiFiUDP.h>

const char* ssid = "BTHomeSpot-XNH";
const char* password = "49345abseb";
const int UDPPort = 5000;
const char* DestUDPAddress = "192.168.1.71";
char Buffer[80];
WiFiUDP udp;

// Set mode to station, and connect to the local Wi-Fi. Display
// the local IP address
//
void setup()
{
    Serial.begin(9600);
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
    {
        Serial.println("Attempting to connect...");
        delay(1000);
    }
    Serial.println("Connected to local Wi-Fi");
    Serial.print("Local IP address is: ");
    Serial.println(WiFi.localIP());
    udp.begin(UDPPort);           // listen to incoming packets
}
```

```
//  
// Parse the received packet and get its details, such as the  
// packet size, remote IP address, and remote port number. Then  
// display the contents of the received packet which is stored  
// in variable Buffer  
//  
void loop()  
{  
    udp.beginPacket(DestUDPEndPoint, UDPPort);  
    int packetSize=udp.parsePacket();  
    if(packetSize)  
    {  
        Serial.print("Received packet size is: ");  
        Serial.print(packetSize);  
        Serial.println(" characters");  
        Serial.print("Packet received from: ");  
        Serial.println(udp.remoteIP());  
        Serial.print("Port address is: ");  
        Serial.println(udp.remotePort());  
    }  
    int len = udp.read(Buffer, 80);  
    if(len > 0)  
    {  
        Serial.println(Buffer);  
    }  
    udp.printf("Hello from the Server");  
    delay(2000);  
    udp.endPacket();  
}
```

Figure 7.11 Program listing

Clicking the **Send** button on the **Packet Sender** program will send the packet to the ESP32 DevKitC. Figure 7.12 shows the data received and displayed on the Serial Monitor by the ESP32 DevKitC. Figure 7.13 shows the **Packet Sender** program with the UDP packet received and sent to the ESP32 DevKitC.

```
Attempting to connect...  
Attempting to connect...  
Connected to local Wi-Fi  
Local IP address is: 192.168.1.156  
Received packet size is: 21 characters  
Packet received from: 192.168.1.71  
Port address is: 55184  
Hello from the client
```

Figure 7.12 Serial Monitor display

							<input checked="" type="checkbox"/> Log Traffic	Save Log	Save Traffic Packet	Copy to Clipboard
Time	From IP	From Port	To IP	To Port	Method	Error	ASCII	Hex		
1 12:09:29.905 pm	192.168.1.156	5000	You	55184	UDP		Hello from the Server	48	65	6C 6C 6F 20 66 72 6F 6D 20 74 68
2 12:09:27.423 pm	You	55184	192.168.1.156	5000	UDP		Hello from the client	48	65	6c 6c 6f 20 66 72 6f 6d 20 74 68 61
3 12:08:50.215 pm	You	55184	192.168.1.156	5000	UDP		Hello from the client	48	65	6c 6c 6f 20 66 72 6f 6d 20 74 68 61

Figure 7.13 Packet Sender display

7.5.2 TCP/IP Programs

TCP/IP is a connection based reliable protocol. An example is given in this section to show how this protocol can be used for data exchange using the ESP32 DevKitC.

Example 7.2

Write a TCP/IP program to receive a text message from a TCP/IP client and display the message on the Serial Monitor. Then, send the message **Hello From the Server** to the client.

Assume the following IP addresses and port numbers for the server and the client:

Server (ESP32 DevKitC) IP address:	192.168.1.156
Client (PC) IP address:	192.168.1.71

In this example, the ESP32 DevKitC is the TCP/IP Server and the PC is the TCP/IP Client. The local port number is set to 5000.

Solution 7.2

The test system setup is as in Figure 7.9, but here TCP/IP is used instead of UDP. Figure 7.14 shows the program listing (program: **TCP**). Again, the packet sender is used to test the program, but the protocol is set to TCP and a full-stop is inserted at the end of the message, i.e. the message is "**Hello from the client.**". At the beginning of the program the server port address is defined. Inside the setup routine connection is made to the local Wi-Fi router, the local IP address is displayed, and the server is set to listen for client connections. Inside the main program a packet is read from the client until the full-stop character is detected. Then a message is sent to the client.

```
/*
 *          *****TCP/IP SERVER EXAMPLE*****
 *          =====
 *
 *  This is a TCP/IP server example. The program sends the message
 * "Hello from the Server" to a TCP/IP client. The IP addresses
 * and port numbers of the server and the client are:
 *
 *  Server (ESP32 Devkitc) IP address: 192.168.1.156 Port: 5000
 *  Client (PC) IP address:           192.168.1.71
 *
 *  File:    TCP

```

```
* Author: Dogan Ibrahim
* Date: November, 2018
*
***** */
#include "WiFi.h"

const char* ssid = "BTHomeSpot-XNH";
const char* password = "49345abseb";
const int TCPPort = 5000;
char Buffer[80];
WiFiServer server(TCPPort);

//
// Set mode to station, and connect to the local Wi-Fi. Display
// the local IP address, and begin listening for clients
//
void setup()
{
    Serial.begin(9600);
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
    {
        Serial.println("Attempting to connect...");
        delay(1000);
    }
    Serial.println("Connected to local Wi-Fi");
    Serial.print("Local IP address is: ");
    Serial.println(WiFi.localIP());
    server.begin();                                // Begin listening for
clients
}

//
// Read the packet from the client until character "." detected,
// and then send the message "Hello from teh Server" to the
// client
//
void loop()
{
    WiFiClient client = server.available();          // A client connected?
    while(client.connected() && !client.available())
    {   delay(1);
    }
    String line=client.readStringUntil('.');
    Serial.println(line);
```

```
    client.printf("Hello from the Server");
    client.stop();

}
```

Figure 7.14 Program listing

Clicking the **Send** button on the **Packet Sender** program will send the packet to the ESP32 DevKitC. Figure 7.15 shows the data received and displayed on the Serial Monitor by the ESP32 DevKitC. Figure 7.16 shows the **Packet Sender** program with the TCP packet received and sent to the ESP32 DevKitC.

```
Attempting to connect...
Connected to local Wi-Fi
Local IP address is: 192.168.1.156

Hello from the client
```

Figure 7.15 Serial Monitor display

Time	From IP	From Port	To IP	To Port	Method	Error	ASCII	Hex
1 3:35:47.052 pm	192.168.1.156	5000	You	14211	TCP			
2 3:35:46.923 pm	192.168.1.156	5000	You	14211	TCP	Hello from the Server	48 65 6C 6F 20 66 72 6F 6D 20 74 68 6	48 65 6c 6f 20 66 72 6f 6d 20 74 68 65
3 3:35:46.808 pm	You	14211	192.168.1.156	5000	TCP	Hello from the client.	48 65 6c 6f 20 66 72 6F 6d 20 74 68 65	48 65 6c 6f 20 66 72 6f 6d 20 74 68 65

Figure 7.16 Packet Sender display

7.6 Getting the Time from NTP Client

In some applications we may want to timestamp a reading. For example, after reading the ambient temperature and humidity readings at regular intervals we may want to store the data with timestamping.

In this program we will read the time from the NTP Client and display it every second on the PC screen. You should install the NTP Client library (by Fabrice Weinberg, V3.1.0) as follows before compiling the program:

- Start your Arduino IDE.
- Click **Sketch** and then **Install Library** and then **Manage Libraries**.
- Search for **NTP CLIENT** and click to Install **NTP Client by Fabrice Weinberg**.
- Make sure that the folder **NTPClient** is included in directory **Libraries** under your **Arduino** top directory.

The program listing (program: **NTPTIME**) is shown in Figure 7.17. At the beginning of the program the header files **WiFi.h**, **NTPClient.h** and **WiFiUdp.h** are included in the program. Inside the **setup** routine the program connects to the local Wi-Fi and also to the NTP Client. The remainder of the program runs continuously inside the program loop. Time is received by calling function **getFormattedTime** and is stored in variable **TimeNow**, and is displayed on the PC screen. Figure 7.18 shows an output from the program.

```
/*********************************************
*                      CONNECT TO NTC CLIENT
*                      =====
*
* This program connects to the local NTP Client and reads and
* displays the time in the following format: hh:mm:ss
*
* File:    NTPTIME
* Author: Dogan Ibrahim
* Date:   November, 2018
*****
#include "WiFi.h"
#include "NTPClient.h"
#include "WiFiUdp.h"

WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP);                                // NTP Client

const char* ssid = "BTHomeSpot-XNH";                          // Wi-Fi SSID
const char* password = "49145abseb";                          // Wi-Fi Password

String TimeNow;

//
// Connect to local Wi-Fi router and to the NTP Client
//
void setup()
{
    Serial.begin(9600);
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
    {
        delay(1000);
    }
    Serial.println("");
    Serial.println("Connected to Wi-Fi network");
    timeClient.begin();
}
```

```
void loop()
{
    timeClient.update();
    TimeNow = timeClient.getFormattedTime();           // Get time
    Serial.println(TimeNow);                          // Display time

    delay(1000);                                     // Wait 1 sec
}
```

Figure 7.17 Program listing

```
Connected to Wi-Fi network
19:37:15
19:37:16
19:37:17
19:37:18
19:37:19
19:37:20
19:37:21
19:37:22
19:37:23
```

Figure 7.18 Output from the program

7.7 Summary

In this chapter we have seen network programming of the ESP32 DevKitC development board using the Arduino IDE as the programming environment. In the next few chapters we shall be designing projects using the DevKitC in the Arduino IDE environment.

Chapter 8 • Project – The temperature and humidity on the cloud

8.1 Overview

In the last chapter we have seen how to write network programs for the ESP32 DevKitC development board.

In this chapter we will be developing an interesting network based application. Here, the system will get the ambient temperature and humidity using a sensor and will then store this data on the cloud every 30 seconds so that it can be accessed from anywhere.

8.2 The Block Diagram

The block diagram of the system is shown in Figure 8.1. In this project a DHT11 temperature and humidity sensor chip is used (see Chapter 5.3).

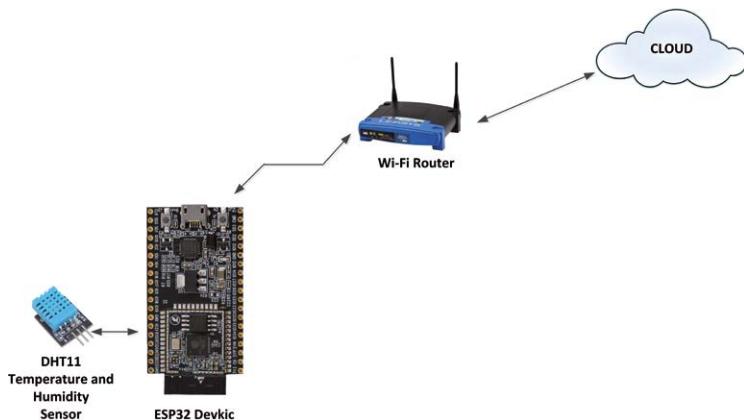


Figure 8.1 Block diagram of the project

The circuit diagram of the project is as in Figure 5.12. The data output of DHT11 is connected to GPIO port 23 of the ESP32 DevKitC development board. The circuit is built on a breadboard as shown in Figure 5.13.

8.3 The Cloud

There are several cloud services that can be used to store data (for example SparkFun, Thingspeak, Cloudino, Bluemix etc). In this project the Thingspeak is used. This is a free cloud service where sensor data can be stored and retrieved using simple HTTP requests. Before using the Thingspeak we have to create an account from their website and then log in to this account. Create a new account by entering your email address and choose a password by opening the following link:

https://thingspeak.com/users/sign_up

You should get an email to verify and activate your account. After this, click **Continue** and you should get a successful sign-up notice as shown in Figure 8.2 and you should agree to the conditions.

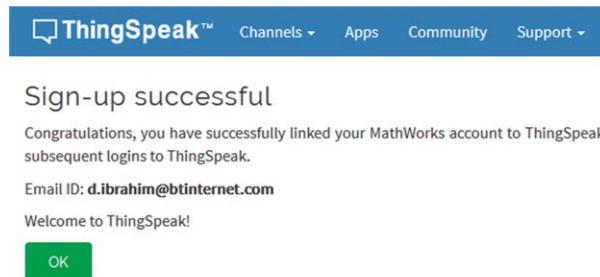


Figure 8.2 Successful sign-up to Thingspeak

Then, create a New Channel by clicking on **New Channel**. Fill in the form as shown in Figure 8.3. Give the name **MyWeather** to the application and create two channels called **Temperature** and **Humidity**, tick to make the channels public.

The screenshot shows the "New Channel" creation interface. At the top is a blue header bar with the ThingSpeak logo and navigation links: Channels ▾, Apps, Community, and Support. The main area is titled "New Channel". It contains fields for "Name" (set to "MyWeather"), "Description" (set to "Get the ambient temperature and humidity data"), and three "Field" sections. "Field 1" is set to "Temperature" with a checked checkbox. "Field 2" is set to "Humidity" with a checked checkbox. "Field 3" is empty with an unchecked checkbox.

Figure 8.3 Create a New Channel (only part of the form shown)

Click **Save Channel** at the bottom of the form. Your channel is now ready to be used with your data. You will now see tabs with the following names. You can click at these tabs and see the contents to make corrections if necessary:

- **Private View:** This tab displays private information about your channel that only you can see.
- **Public View:** If your channel is public, use this tab to display selected fields and channel visualisations.
- **Channel Settings:** This tab shows all the channel options you set at creation. You can edit, clear, or delete the channel from this tab.
- **API Keys:** This tab displays your channel API keys. Use the keys to read from and write to your channel.
- **Data Import/Export:** This tab enables you to import and export channel data.

You should click the **API Keys** tab and save your **Write API** and **Read API** keys and the **Channel ID** in a safe place. The API Key and the Channel Number in this project were:

API Key = QAAHT5XIK3ZB8736
Channel Number = 265433

8.4 Program Listing

At the time of writing this book, Arduino IDE supported the Thingspeak library. Using this library makes the programming very easy. But unfortunately, the support was for only for the Arduino hardware family and the ESP8266 processor. There was no library support for the ESP32 processor. Because of this, we have to program from the first principles, where the data is sent to the cloud using TCP/IP data packets.

Figure 8.4 shows the complete program listing of the project (program: **CLOUD**). At the beginning of the program, library headers for the Wi_Fi and the DHT11 sensor are included. Local Wi-Fi name and password are specified, the API key and Thingspeak host name are defined.

Function **Connect_WiFi** makes connection to the local Wi-fi router. Function **TempHum** reads the ambient temperature and humidity and stores in variables **T** and **H** respectively. Inside the setup routine the DHT11 is initialised, function **Connect_WiFi** is called and a connection is made to the HTTP via port 80.

The main program runs in an endless loop in which the code is executed every 30 seconds. Here, the ambient temperature and humidity are read by calling to function **TempHum**, and then the following code is used to send an HTTP GET request to the Thingspeak website:

```
client.print("GET /update?api_key=QAAHT5XIK3ZB8736&field1=");
client.print(String(T));
client.print("&field2=");
client.print(String(H));
client.print(" HTTP/1.0\r\nHost: api.thingspeak.com\r\n\r\n");
```

The API key and the latest readings of the temperature and humidity are converted into strings and are included in the HTTP GET request. Notice that **field1** and **field2** expect the temperature and the humidity values to be supplied respectively. The actual string is:

```
GET /update?api_key=QAAHT5XIK3ZB8736&field1=String(T)&field2=String(H)
HTTP/1.0\r\nHost: api.thingspeak.com\r\n\r\n
```

The data is plotted automatically by the Thingspeak website. The following link can be used to see the plotted data:

https://api.thingspeak.com/channels/YOUR_CHANNEL_ID

In this project, for example the channel ID is 265433. Therefore, the link to the data is:

<https://api.thingspeak.com/channels/265433>

```
/*
 *          ***** TEMPERATURE AND HUMIDITY ON THE CLOUD *****
 *          =====
 *
 * In this project a DHT11 type temperature and humidity sensor
 * chip is connected to GPIO port 23 of the ESP32 Devkitc. The
 * program reads the ambient temperature and humidity and sends
 * this data to the cloud every 60 seconds where it can be
 * accessed from anywhere on Earth. In addition, change of the
 * temperature and the humidity can be plotted using the cloud
 * services.
 *
 * The program uses the Thingspeak cloud service. The ESP32
 * must be connected to the Wi-Fi network before we can send
 * data to the cloud
 *
 *
 * File:    CLOUD
 * Author:  Dogan Ibrahim
 * Date:   November, 2018
 *
 *****/
#include "WiFi.h"
//
// DHT11 sensor library includes
//
#include <Adafruit_Sensor.h>
#include "DHT.h"
#define DHT11_PIN 23
#define DHTTYPE DHT11

//
// Local Wi-Fi name and password
//
const char* ssid = "BTHomeSpot-XNH";
const char* password = "49341abseb";

//
// Thingspeak API key, channel number, and host name
//
unsigned long myChannelNumber = 265433;
String APIKEY = "QAAHT5XIK3ZB8736";
```

```
const char* host = "api.thingspeak.com";  
  
int T, H;  
DHT dht(DHT11_PIN, DHTTYPE);  
WiFiClient client;  
  
//  
// This function connects the ESP32 Devkitc to the local Wi-Fi  
// network. The network name and passwors are specifed earlier  
void Connect_WiFi()  
{  
    WiFi.begin(ssid, password);  
    while(WiFi.status() != WL_CONNECTED)  
    {  
        delay(1000);  
    }  
}  
  
//  
// This function reads the ambient temperature and humidity and  
// stores in variables T and H respectively  
//  
void TempHum()  
{  
    H = dht.readHumidity(); // Read humidity  
    T = dht.readTemperature(); // Read temperature  
}  
  
//  
// Initialize DHT11, connect to local Wi-Fi, connect to HTTP  
//  
void setup()  
{  
    dht.begin();  
    Connect_WiFi();  
    client.connect(host, 80);  
}  
  
//  
// This is the main program. Read the ambient temperature and  
// humidity and send the data to Thingspeak every 30 seconds
```

```
//  
void loop()  
{  
    client.connect(host,80);  
    TempHum();  
    client.print("GET /update?api_key=QAAHT5XIK3ZB8736&field1=");  
    client.print(String(T));  
    client.print("&field2=");  
    client.print(String(H));  
    client.print(" HTTP/1.0\r\nHost: api.thingspeak.com\r\n\r\n");  
    delay(30000);  
}
```

Figure 8.4 Program listing of the project

A sample output plotted by the Thingspeak is shown in Figure 8.5.



Figure 8.5 A sample output from Thingspeak

At the time of writing this book it was not possible to configure the graphs for example to change the vertical or the horizontal axes, the starting date of the graphs etc. A separate graph with the required starting date (and time) can, however, be plotted using the following web link where the channel number and the starting date are specified (see the Thingspeak website for more details):

<HTTP://pi.thingspeak.com/channels/265433/charts/1?start=2017-07-31>

It is also an option to export the graph data into a spreadsheet program like Excel and then plot a graph of the data. This option gives more control over the configuration of the graph type, axes, labels, etc. For example, we can export the data to Excel by clicking Data Export and selecting file type as CSV.

8.5 Summary

In this chapter we have developed an interesting application which gets the ambient temperature and humidity using the DHT11 sensor chip, and then sends this data to the cloud so that it can be accessed at anytime and anywhere on Earth.

In the next chapter we will be developing a Web Server application in which two LEDs are connected to the ESP32 DevKitC development board and they are controlled remotely from a web application.

Chapter 9 • Remote Web Based Control – Web Server

9.1 Overview

In the last chapter we have seen the design and development of an interesting Wi-Fi application which gets the ambient temperature and humidity and stores on the cloud.

In this chapter we will be developing another interesting application. Here, the system will control remotely two LEDs connected to an ESP32 DevKitC development board using an HTTP Web Server application. Although in this example two LEDs are used, relays can generally be connected to DevKitC and any type of electrical equipment can be controlled remotely.

9.2 The Block Diagram

The block diagram of the system is shown in Figure 9.1. Figure 9.2 shows the circuit diagram of the project. LED0 and LED1 are connected to GPIO ports 23 (LED0) and 22 (LED1) through 330 ohm current limiting resistors. The LEDs are turned ON when the port output is at logic 1 (high) and turned OFF when the port output is at logic 0 (low).

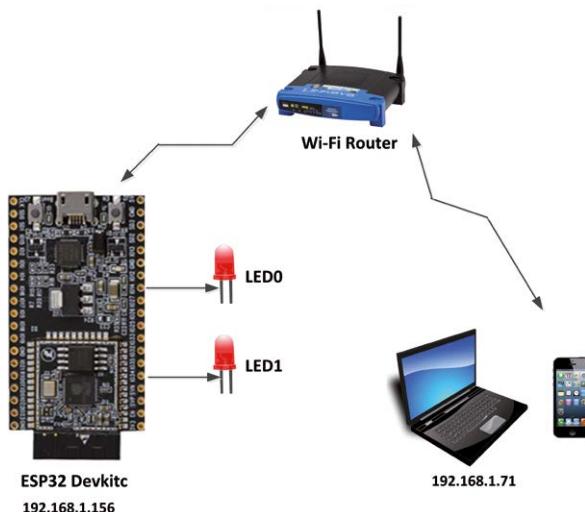


Figure 9.1 Block diagram of the system

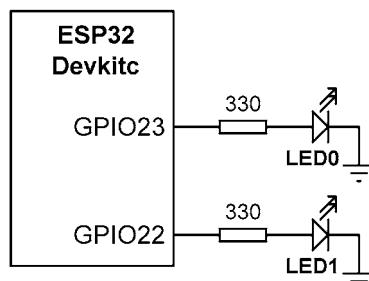


Figure 9.2 Circuit diagram of the system

9.3 HTTP Web Server/Client

A web server is a program that uses HTTP (Hypertext Transfer Protocol) to serve web pages to users in response of their requests. These requests are forwarded by HTTP clients. By using the HTTP server/client pair we can control any device connected to a web server processor over the web.

Figure 9.3 shows the structure of a web server/client setup. In this figure, the ESP32 DevKitC is the web server and the PC (or a laptop, tablet or a mobile phone) is the web client. The device to be controlled is connected to the web server processor. In this example we have two LEDs connected to the ESP32 DevKitC development board. The operation of the system is as follows:

- The web server is in the listen mode, listening for requests from the web client.
- The web client makes a request to the web server by sending an HTTP request.
- In response, the web server sends an HTTP code to the web client which is activated by the web browser by the user on the web client and is shown as a form on the web client screen.
- The user sends a command (e.g. ticks a button on the web client form to turn ON an LED) and this sends a code to the web server so that the web server can carry out the required operation.

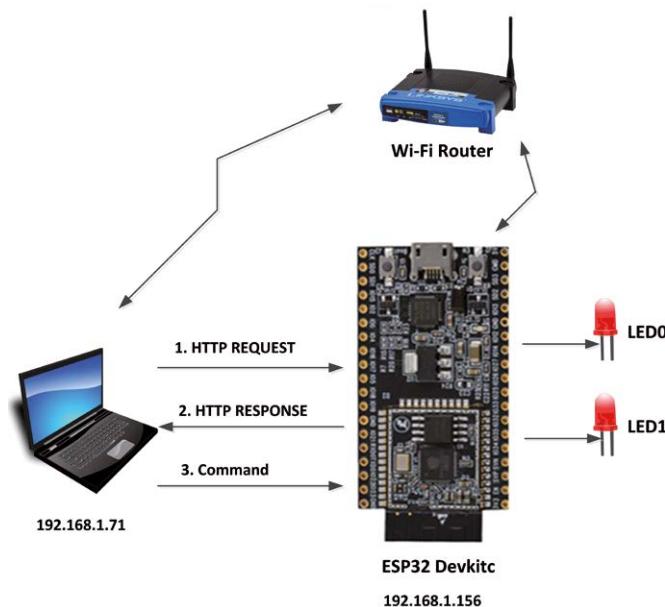


Figure 9.3 Web server/client structure

In this program a laptop with IP address 192.168.1.71 is used as the web client and the ESP32 DevKitC with the IP address 192.168.1.156 is used as the web server.

9.4 ESP32 DevKitC Program Listing

Figure 9.4 shows the complete program (program: **WEB SERVER**). At the beginning of the program, the Wi-Fi library is included in the program, LED0 and LED1 are assigned to GPIO ports 23 and 22 respectively, and the local Wi-Fi name and password are defined.

The program then defines the HTML code that is to be sent to the web client when a request comes from it. This code is stored in variable **html**. When executed by the web browser (e.g. Internet Explorer) on the web client, the display in Figure 9.5 is shown on the client screen.

A heading is displayed at the centre of the screen and buttons are displayed at the left hand side. Two pairs of buttons are displayed: one pair to turn ON/OFF LED0 and another pair to turn ON/OFF LED1. The ON buttons are green, and the OFF buttons are red.

The web server program sends the html file (statement **client.print(html)**) to the client and waits for a command from the client. When a button is clicked on the client, a command is sent to the web server in addition to some other data. Here, we are only interested in the actual command sent. The following commands are sent to the web server for each button pressed:

Button Pressed	Command sent to web server
LED0 ON	/?LED0=ON
LED0 OFF	/?LED0=OFF
LED1 ON	/?LED1=ON
LED1 OFF	/?LED1=OFF

The web server program on the ESP32 DevKitC makes a TCP connection and reads the command from the web client as a string (**String request = client.readStringUntil('\r')**). The program then searches the received command to see if any of the above commands are present in the received data. If the sub-string is not found then a -1 is returned. For example, consider the following statement:

```
if(request.indexOf("LED0=ON") != -1)digitalWrite(LED0, HIGH);
```

Here, sub-string **LED0=ON** is searched in string **request**. A -1 is returned if the sub-string **LED0=ON** does not exist in string **request**. The program looks for a match for all the 4 commands and if a command is found then the corresponding LED is turned ON or OFF accordingly:

```
if(request.indexOf("LED0=ON") != -1)digitalWrite(LED0, HIGH);
if(request.indexOf("LED0=OFF") != -1)digitalWrite(LED0, LOW);
if(request.indexOf("LED1=ON") != -1)digitalWrite(LED1, HIGH);
if(request.indexOf("LED1=OFF") != -1)digitalWrite(LED1, LOW);
```

ESP32 Devkitc LED ON/OFF

Web Server Example with 2 LEDs



Figure 9.5 The screen when the HTML code is executed by the client

In this project, the IP address of the web server was 192.168.1.156. The procedure to test the system is as follows:

- Compile, upload and run the program on the ESP32 DevKitC.
- Open your web browser on your PC and enter the web site address 192.168.1.156.
- The form shown in Figure 9.5 will be displayed on your PC screen.
- Click a button, e.g. LED0 ON, to turn ON LED0.

Figure 9.6 and Figure 9.7 show the commands sent to the web server when LED0 ON button is clicked, and also when LED0 OFF button is clicked.

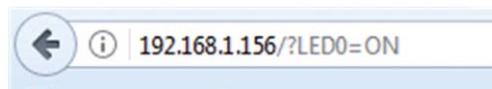


Figure 9.6 Clicking LED0 ON button

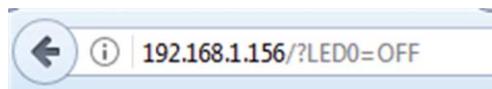


Figure 9.7 Clicking LED0 OFF button

```
/*
 *      WEB SERVER TO CONTROL 2 LEDs
 *      =====
 *
 * This is a simple web server program. 2 LEDs are connected to
 * GPIO ports 23 and 22 of the ESP32 devkitc board. The project
 * controls these LEDs (turns ON or OFF) from a web server
 * application. For example, the LEDs can be controlled from any
 * device that is on the web, for example, a PC, tablet, mobile
 * phone etc. when activated, a form will appear on the device
 * with buttons and clicking these buttons will control the
 * LEDs. The LEDs are named LED0 and LED1, connected to GPIO
 * ports 23 and 22 respectively.
 *
 *
 * File:    WEBSERVER
 * Author:  Dogan Ibrahim
```

```
* Date: November, 2018
*
***** /



#include "WiFi.h"
#define LED0 23
#define LED1 22

// Local Wi-Fi name and password
//
const char* ssid = "BTHomeSpot-XNH";
const char* password = "49345abseb";
WiFiServer server(80);

// The HTML code. This code will display two buttons on user's
// device which can be clicked to control the LEDs
//
String html ="<!DOCTYPE html> \
<html> \
<body> \
<center><h1>ESP32 Devkitc LED ON/OFF</h1></center> \
<center><h2>Web Server Example with 2 LEDs</h2></center> \
<form> \
<button name=\"LED0\" button style=\"color:green\" value=\"ON\" \
type=\"submit\">LED0 ON</button> \
<button name=\"LED0\" button style=\"color:red\" value=\"OFF\" \
type=\"submit\">LED0 OFF</button><br><br> \
<button name=\"LED1\" button style=\"color:green\" value=\"ON\" \
type=\"submit\">LED1 ON</button> \
<button name=\"LED1\" button style=\"color:red\" value=\"OFF\" \
type=\"submit\">LED1 OFF</button> \
</form> \
</body> \
</html>";

// This function connects the ESP32 DevKitC to the local Wi-Fi
// network. The network name and passwors are as specified earlier
void Connect_WiFi()
{
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
    {
        delay(1000);
    }
}
```

```
}

// Configure LEDs as outputs, turn OFF the LEDs to start with,
// Connect to the local Wi-Fi, start the server
//
void setup()
{
    pinMode(LED0, OUTPUT);
    pinMode(LED1, OUTPUT);
    digitalWrite(LED0, LOW);
    digitalWrite(LED1, LOW);
    Connect_WiFi();
    server.begin();
}

// This is teh main program loop. Inside the main program we
// check for the client connection and send the HTML file so
// that it is displayed on user's device. The user clicks the
// buttons to control the LEDs.
//
void loop()
{
    WiFiClient client=server.available();
    String request = client.readStringUntil('\r');
    client.flush();

    // We control the LEDs depending upon the key click. Variable
    // request holds the request and we search this string to see
    // which LED should be turned ON/OFF. The contents of request
    // is of the form (for example, to turn OFF LED0):
    // "/?LED0=OFF", or similarly, to turn LED 1: "/?LED1=ON"
    //

    if(request.indexOf("LED0=ON") != -1)digitalWrite(LED0, HIGH);
    if(request.indexOf("LED0=OFF") != -1)digitalWrite(LED0, LOW);
    if(request.indexOf("LED1=ON") != -1)digitalWrite(LED1, HIGH);
    if(request.indexOf("LED1=OFF") != -1)digitalWrite(LED1, LOW);

    //
    // The HTML page to be displayed on user's device
    //
    client.print(html);
}
```

Figure 9.4 Program listing of the project

9.5 Accessing Web Server From Anywhere

In our project the web server can only be accessed locally using the local Wi-Fi router. In this section we will see how to access our web server from anywhere in the world, so that, for example, appliances in our house can be controlled and monitored remotely from anywhere in the world.

We will be using a free service called **ngrok** in order to create a tunnel to our ESP32 so that we can access it from anywhere in the world. First of all we have to change our default port number from 80 to something else, e.g. 8888, since for some reason ngrok does not work with port number 80.

The steps to install and use **ngrok** are as follows:

- Go to web site <https://ngrok.com>
- Enter your details as shown in Figure 9.8 to create an account.

The screenshot shows the 'Join ngrok' registration page. It has fields for 'Your Name' (Full Name), 'Your Email' (Email address), 'Confirm Email' (Email address), and 'Password'. Below these is a reCAPTCHA field with a checkbox labeled 'I'm not a robot' and a reCAPTCHA logo. At the bottom is a large orange 'Create an account' button.

Figure 9.8 Enter your details

- Click **Auth** link at the left hand side to get your unique Tunnel token as shown in Figure 9.9. In this example author's tunnel token is: **gquidoMhDuJeXnLqVGH-qv_7jm25SyUTtbYpnQzTk3Eb**



Figure 9.9 Tunnel token

- Click the **Download** tab at the top of the screen and install **ngrok**

- Unzip file to a folder and run the **ngrok.exe** application. You should see a window as in Figure 9.10.

```
Administrator: C:\Users\Dogan\AppData\Local\Temp\Temp1_ngrok-stable-windows-amd64.zip\...
ngrok tcp 22 # tunnel arbitrary TCP traffic to port 22
ngrok tls -hostname=foo.com 443 # TLS traffic for foo.com to port 443
ngrok start foo bar baz # start tunnels from the configuration file

VERSION:
2.2.8

AUTHOR:
inconshreveable - <alan@ngrok.com>

COMMANDS:
auth_token    save auth_token to configuration file
credits       prints author and licensing information
http          start an HTTP tunnel
start         start tunnels by name from the configuration file
tcp           start a TCP tunnel
tls           start a TLS tunnel
update        update ngrok to the latest version
version       print the version string
help          shows a list of commands or help for one command

ngrok is a command line application, try typing 'ngrok.exe http 80'
at this terminal prompt to expose port 80.
C:\Windows\system32>
```

Figure 9.10 Running ngrok

- Enter the following command by replacing the code in < > with your own IP address and **ngrok** Token:

```
ngrok tcp <ESP-IP-ADDRESS>:8888 --authToken <Your ngrok Token>
```

For our project in this chapter the command should be:

```
ngrok tcp 192.168.1.156:8888 --authToken gqundoMhDuJeXnLqVGHqv_7jm2SSyUTtbYpnQzTk3Eb
```

The command should look as shown in Figure 9.11 on the PC terminal.

```
C:\ngrok>ngrok tcp 192.168.1.156:8888 --authToken gqundoMhDuJeXnLqVGHqv_7jm2SSyUTtbYpnQzTk3Eb
```

Figure 9.11 Command on the PC terminal

- You should now see that your tunnel is online and a Forwarding URL will be given as shown in Figure 9.12. In this example the Forwarding URL is: **tcp://0.tcp.ngrok.io:17386**

Session Status	online												
Account	Dogan Ibrahim (Plan: Free)												
Version	2.2.8												
Region	United States (us)												
Web Interface	http://127.0.0.1:4040												
Forwarding	tcp://0.tcp.ngrok.io:17386 -> 192.168.1.156:8888												
Connections	<table border="1"> <thead> <tr> <th>ttl</th> <th>open</th> <th>rt1</th> <th>rt5</th> <th>p50</th> <th>p90</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0.00</td> <td>0.00</td> <td>0.00</td> <td>0.00</td> </tr> </tbody> </table>	ttl	open	rt1	rt5	p50	p90	0	0	0.00	0.00	0.00	0.00
ttl	open	rt1	rt5	p50	p90								
0	0	0.00	0.00	0.00	0.00								

Figure 9.12 The Forwarding URL

In order to access your web server anywhere in the world you should enter your unique URL in a browser, which in this example is:

<http://0.tcp.ngrok.io:17386>

Notice that your PC must be up and running **ngrok** while you access your web server. You will be asked to enter your username and password in order to open your web server.

9.6 Summary

In this chapter we have seen an interesting web server application in which two LEDs are controlled remotely from the web. Although in this project LEDs are used, in general relays can be used and any electrical equipment can be controlled remotely. The chapter has also explained how to access the web server from anywhere in the world using the free service ngrok.

In the next chapter we will see how to control equipment remotely by sending UDP packets from a mobile phone.

Chapter 10 • Remote control using mobile phone

10.1 Overview

In the last chapter we have seen the design and development of an interesting Wi-Fi application which controls two LEDs from a web server application. Although in that simple example LEDs are used, there is no reason why relays cannot be connected instead of the LEDs to control any kind of electrical equipment remotely.

In this chapter we will be developing another interesting application. Here, the system will control remotely two LEDs connected to an ESP32 DevKitC development board using a mobile phone. The idea here is that the DevKitC will be configured as an UDP client. Then, UDP packets will be sent from a mobile phone to control these LEDs. As mentioned earlier, these LEDs can be replaced by relays and the project can be used to control any kind of electrical equipment connected to the ESP32 DevKitC.

Valid commands are as follows (any other commands are ignored by the program):

0=ON	Turn ON LED0
0=OFF	Turn OFF LED0
1=ON	Turn ON LED1
1=OFF	Turn OFF LED1

10.2 The Block Diagram

The block diagram of the project is shown in Figure 10.1. Figure 9.2 shows the circuit diagram of the project. LED0 and LED1 are connected to GPIO ports 23 (LED0) and 22 (LED1) through 330 ohm current limiting resistors as in the project in Chapter 9. The LEDs are turned ON when the port output is at logic 1 (high), and turned OFF when the port output is at logic 0 (low).

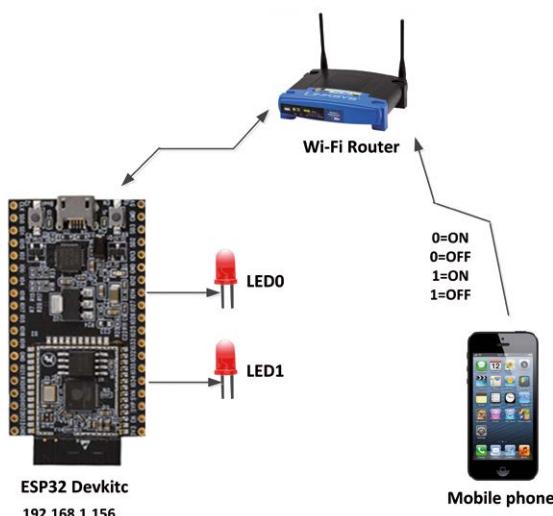


Figure 10.1 Block diagram of the system

10.3 Mobile Phone Application

In this project an Android based mobile phone is used to send the UDP commands. Although one could develop a mobile application to send the UDP commands, there are many freely available UDP apps in the Play Store that can be used to send and/or receive UDP packets. The one used in this project is called UDP RECEIVE and SEND by Wezzi Studios (version 4.0). This application can send and receive UDP packets from any other device running the UDP protocol. The screen is in two parts: the upper Sender part and the lower Receiver part (see Figure 10.2). Before sending a packet the user must enter the destination IP address, port number, and the message to be sent. Clicking the SEND UDP MESSAGE button will send the packet to its destination. In the Receiver part the local IP address is given. The user must specify the required port number. Received packets will then be displayed on the screen.

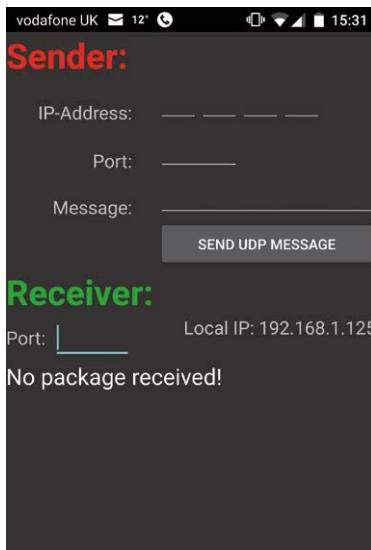


Figure 10.2 Android UDP send/receive apps

In this program, the IP address of the ESP32 DevKitC was 192.168.1.156.

10.4 ESP32 DevKitC Program Listing

Figure 10.3 shows the complete program (program: UDPControl). At the beginning of the program, the Wi-Fi library headers are included in the program, LED0 and LED1 are assigned to GPIO ports 23 and 22 respectively, local port number is defined, and the local Wi-Fi name and password are defined. Function Connect_WiFi connects to the local Wi-Fi network.

Inside the setup routine the LED ports are configured as outputs and both LEDs are turned OFF to start with. Then function Connect_WiFi is called to connect to local Wi-Fi and UDP is started on the local port.

The remainder of the program is executed in an endless loop. Inside this loop the program waits to receive a packet (command) from the UDP server (mobile phone here). The received packet is stored in character array Packet and the program checks the command and if this is a valid command, turns ON/OFF the LEDs as required.

The following code is used to decode the received command:

```

if(Packet[1] == '=')
{
    if(Packet[0] == '0')
    {
        if(Packet[2] == '0' && Packet[3] == 'N')
        {
            digitalWrite(LED0, HIGH);
        }
        else if(Packet[2] == '0' && Packet[3] == 'F' && Packet[4] == 'F')
        {
            digitalWrite(LED0, LOW);
        }
    }
    else if(Packet[0] == '1')
    {
        if(Packet[2] == '0' && Packet[3] == 'N')
        {
            digitalWrite(LED1, HIGH);
        }
        else if(Packet[2] == '0' && Packet[3] == 'F' && Packet[4] == 'F')
        {
            digitalWrite(LED1, LOW);
        }
    }
}

/*****
*          UDP BASED CONTROL FROM MOBILE PHONE
*          =====
*
* This is an UDP based control program where 2 LEDs are connected
* to GPIO ports 23 and 22 of the ESP32 devkitc board, and named
* as LED0 and LED1. Commands are sent from an Android type
* mobile phone to turn the LEDs ON/OFF. The ESP32 DevKitC is
* configured as an UDP client in this application. The format of
* the commands are as follows:
*
* 0=ON   turn ON LED0
* 0=OFF  turn OFF LED0
* 1=ON   turn ON LED1

```

```
* 1=OFF  turn OFF LED1
*
*
*
* File: UDPControl
* Author: Dogan Ibrahim
* Date: November, 2018
*
*****
#include "WiFi.h"
#include <WiFiUdp.h>

//
// LED assignments
//
#define LED0 23
#define LED1 22
WiFiUDP udp;

//
// Use local port 5000
//
const int Port = 5000;
char Packet[80];

//
// Local Wi-Fi name and password
//
const char* ssid = "BTHomeSpot-XNH";
const char* password = "49345abseb";

//
// This function connects the ESP32 Devkitc to the local Wi-Fi
// network. The network name and passwors are as specified earlier
void Connect_WiFi()
{
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
    {
        delay(1000);
    }
}

//
// Configure LEDs as outputs, turn OFF the LEDs to start with,
// Connect to the local Wi-Fi. Also, UDP is started in local port
```

```
//  
void setup()  
{  
    pinMode(LED0, OUTPUT);  
    pinMode(LED1, OUTPUT);  
    digitalWrite(LED0, LOW);  
    digitalWrite(LED1, LOW);  
    Connect_WiFi();  
    udp.begin(Port);  
}  
  
//  
// This is the main program loop. Inside the main program we read  
// UDP packets and then control the LEDs as requested. The format  
// of the control commands are:  
//  
// 0=ON or 0=OFF for LED0  
// 1=ON or 1=OFF for LED1  
//  
// Any other commands are simply ignored by the program  
//  
void loop()  
{  
    int PacketSize = udp.parsePacket();  
    if(PacketSize)  
    {  
        udp.read(Packet, PacketSize);  
  
        if(Packet[1]=='=')  
        {  
            if(Packet[0]=='0')  
            {  
                if(Packet[2]==‘0’ && Packet[3]==‘N’)  
                {  
                    digitalWrite(LED0, HIGH);  
                }  
                else if(Packet[2]==‘0’ && Packet[3]==‘F’ && Packet[4]==‘F’)  
                {  
                    digitalWrite(LED0, LOW);  
                }  
            }  
            else if(Packet[0]==‘1’)  
            {  
                if(Packet[2]==‘0’ && Packet[3]==‘N’)  
                {
```

```
        digitalWrite(LED1, HIGH);
    }
    else if(Packet[2]=='0' && Packet[3]=='F' && Packet[4]=='F')
    {
        digitalWrite(LED1, LOW);
    }
}
}
}
```

Figure 10.3 Program listing

As an example, Figure 10.4 shows the command sent from the mobile phone application to turn on LED1 (command: 1=ON).

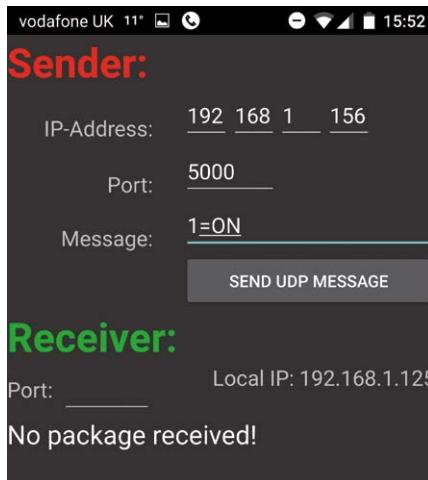


Figure 10.4 Command to turn ON LED1

10.5 Summary

In this chapter we have seen an interesting web server application where two LEDs are controlled remotely from a mobile phone using UDP packets. Although in this project LEDs are used, in general relays can be used and any electrical equipment can be controlled remotely.

In the next chapter we will see how to send the temperature and humidity readings to a mobile phone, again using UDP packets.

Chapter 11 • Send temperature and humidity to a mobile phone

11.1 Overview

In the last chapter we have seen the design and development of an interesting Wi-Fi application which controls two LEDs from an Android mobile phone.

In this chapter we will be developing another interesting application. Here, we will read the ambient temperature and humidity and then send this data to a mobile phone whenever a request is made. In this program data is exchanged using UDP packets.

The data will be sent in the following format, where T and H are the temperature and humidity readings:

$$T = nn \quad H = nn$$

A request for data is made by the mobile phone when it sends the character S to the ESP32 DevKitC.

This project uses two way UDP communications to receive and send data.

11.2 The Block Diagram

The block diagram of the project is shown in Figure 11.1. Figure 5.12 shows the circuit diagram of the project where the DHT11 output (pin S) is connected to GPIO port 23 of the ESP32 DevKitC.

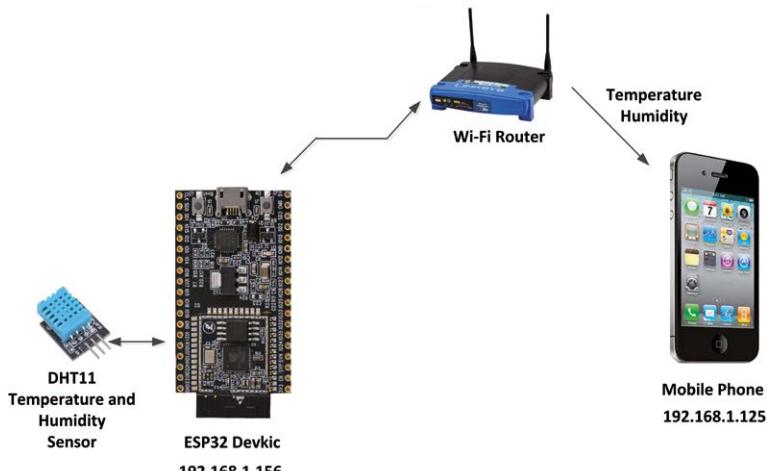


Figure 11.1 Block diagram of the system

11.3 Mobile Phone Application

In this project an Android based mobile phone is used to receive the UDP packers from the ESP32 DevKitC. As in the project in Chapter 10, Android apps UDP RECEIVE and SEND by Wezzi Studios (version 4.0) is used in this project.

The ESP32 DevKitC and the mobile phone IP addresses are 192.168.1.156 and 192.168.1.125 respectively. Notice that the IP address of the mobile phone can be seen on the screen when the UDP apps is started.

11.4 ESP32 DevKitC Program Listing

Figure 11.2 shows the complete program (program: UDPMonitor). At the beginning of the program, the Wi-Fi and the DHT11 library headers are included in the program, local port number is defined, and the local Wi-Fi name and password are defined. Function Connect_WiFi connects to the local Wi-Fi network. Function TempHum reads the ambient temperature and humidity and stores in variables T and H respectively.

Inside the setup routine function Connect_WiFi is called to connect to local Wi-Fi and UDP and DHT11 are started.

The remainder of the program is executed in an endless loop, in which the program waits to receive a packet (command to return the temperature and humidity readings) from the UDP server (mobile phone here). The request is identified by character S and when this character is received, the temperature and humidity are read and a packet is formed using character array Packet. This array stores the temperature and humidity readings in the required format. The packet is then sent to the mobile phone over port 5000.

Character array Packet is formed as follows:

```
Packet[0] = 'T';           // Display T
Packet[1] = '=';           // Display =
int msd = T/10;           // Extract high digit
int lsd = T - msd*10;     // Extract low digit
Packet[2] = msd + '0';    // Convert msd to ASCII character
Packet[3] = lsd + '0';    // Convert lsd to ASCII character
Packet[4] = ' ';
Packet[5] = ' ';
Packet[6] = 'H';           // Display H
Packet[7] = '=';           // Display =
msd = H/10;               // Extract high digit
lsd = H - msd*10;         // Extract low digit
Packet[8] = msd + '0';    // Convert msd to ASCII character
Packet[9] = lsd + '0';    // Convert lsd to ASCII character
Packet[10] = 0;            // NULL terminator

/*********************************************
*          SEND TEMPERATURE AND HUMIDITY TO A MOBILE PHONE
*          =====
*
* This is an UDP based monitoring program where a DHT11 type
* temperattrue and humidity sensor chip is connected to GPIO
```

```
* port 23 of the ESP32 Devkitc. The program reads the ambient
* temperattrue and humidity and sends the readings to an Android
* mobile phone using UDP packets. The data on the mobile phone
* is displayed using an UDP apps, available on the Play Store.
* The data is sent to the mobile phone whenever it is requested.
*
* A request for data is made to teh ESp32 Devkitc when character
* S is sent to the ESp32 Devkit over the UDP link. The ESP32
* Devkitc sends the temperature and humidiry data to the device
* that made the request. Port 5000 is used for the communication
*
* The IP addresses of the ESP32 Devkitc and mobile phone are:
* 192.168.1.56 and 192.168.1.125 respectively.
*
* File:    UDPMonitor
* Author: Dogan Ibrahim
* Date:    November, 2018
*
*****
#include "WiFi.h"
#include <WiFiUdp.h>
//
// DHT11 sensor library includes
//
#include <Adafruit_Sensor.h>
#include "DHT.h"
#define DHT11_PIN 23
#define DHTTYPE DHT11
DHT dht(DHT11_PIN, DHTTYPE);

WiFiUDP udp;

//
// Use local port 5000
//
const int Port = 5000;
char Packet[20];

//
// Local Wi-Fi name and password
//
const char* ssid = "BTHomeSpot-XNH";
const char* password = "49345abseb";
int T, H;

//
```

```
// Read the ambient temperature and humidity and store in variables
// T and H respectively
//
void TempHum()
{
    H = dht.readHumidity();                                // Read humidity
    T = dht.readTemperature();                            // Read temperature
}

//
// This function connects the ESP32 Devkitc to the local Wi-Fi
// network. The network name and passwors are as specifed earlier
void Connect_WiFi()
{
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
    {
        delay(1000);
    }
}

//
// Connect to the local Wi-Fi. Also, UDP is started in local
// port and DHT11 is started
//
void setup()
{
    Connect_WiFi();
    udp.begin(Port);
    dht.begin();
}

//
// This is the main program loop. Inside the main program we read
// the temperatrue and humidity an send them as a packet to the
// mobile phone when a request is made by the mobile phone.
// A request is made when character S is sent by the mobile
// phone.
//
// The packet format is:
//
//  T=nn H=nn
//
```

```
void loop()
{
    int PacketSize = udp.parsePacket();
    if(PacketSize)
    {
        udp.read(Packet, PacketSize);
        if(Packet[0] == 'S')
        {
            TempHum();
            Packet[0] = 'T';
            Packet[1] = '=';
            int msd = T/10;
            int lsd = T - msd*10;
            Packet[2] = msd + '0';
            Packet[3] = lsd + '0';
            Packet[4] = ' ';
            Packet[5] = ' ';
            Packet[6] = 'H';
            Packet[7] = '=';
            msd = H/10;
            lsd = H - msd*10;
            Packet[8] = msd + '0';
            Packet[9] = lsd + '0';
            Packet[10] = 0;
            udp.beginPacket(udp.remoteIP(), Port);
            udp.print(Packet);
            udp.endPacket();
        }
    }
}
```

Figure 11.2 Program listing

As an example, Figure 11.3 shows the request S sent from the mobile phone application. The returned temperature and humidity data are shown in the lower part of the screen.

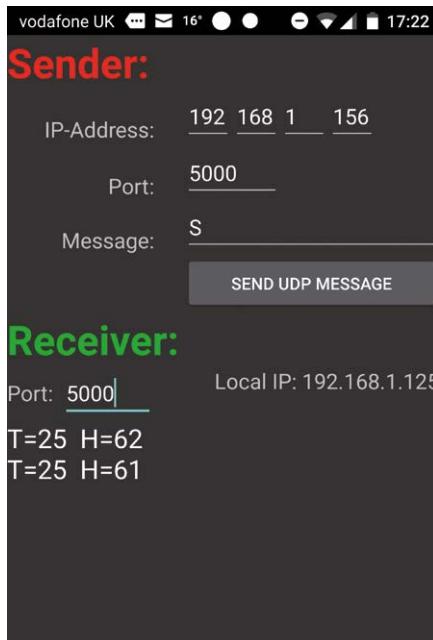


Figure 11.3 Sending request S from mobile phone

11.5 Summary

In this chapter we have seen an interesting web server application where the ambient temperature and humidity readings are sent from the ESP32 DevKitC to an Android mobile phone using UDP packets. This project shows how two-way communication can be established using UDP packets.

In the next chapter we will be installing the microPython software onto our ESP32 DevKitC and see how microPython programs can be developed and uploaded to the DevKitC.

Chapter 12 • Web server with relay

12.1 Overview

In the previous chapter we have seen how to read and send the ambient temperature and humidity readings to a mobile phone.

In this chapter we shall be looking at the design of a web server project in which a 4-channel relay board is connected to the ESP32 DevkitC and is controlled through a web server interface. Using this project we can connect high voltage devices such as a microwave, dish washer or washing machine to the relay outputs and control them over a web server interface from our PC or mobile phone.

12.2 The Aim

The aim of this project is to show how a 4-channel relay board can be connected to the ESP32 DevKitC and also how it can be controlled over a web server interface.

12.3 The Block Diagram

The block diagram of the project is shown in Figure 12.1 where a 4-channel relay board is directly connected to the ESP32 DevkitC GPIO pins.

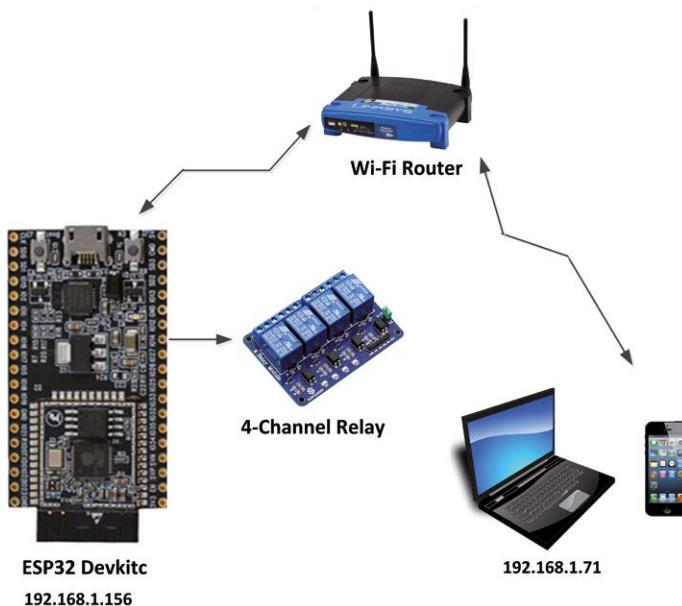


Figure 12.1 Block diagram of the project

12.4 Circuit Diagram

In this project a 4-channel relay board (see Figure 12.2) from Elegoo (www.elegoo.com) is used. This is an opto-coupled relay board with 4 inputs; one for each channel. The relay inputs are at the bottom right hand side of the board while the relay outputs are located at the top. The middle position of each relay is the common point, the connection to its left

is the normally closed (NC) contact, while the connection to the right is the normally open (NO) contact. The relay contacts support AC250V at 10A and DC30V 10A. IN1, IN2, IN3 and IN4 are the **active LOW** inputs, which means that a relay is activated when a logic LOW signal is applied to its input pin. Relay contacts are normally closed (NC). Activating the relay changes the active contacts such that the common pin and NC pin become the two relay contacts and at the same time the LED at the input circuit of the relay board corresponding to the activated relay is turned ON. The VCC can be connected to either +3.3 V or to +5 V. Jumper JD is used to select the voltage for the relay. **Because the current drawn by a relay can be in excess of 80 mA, you must remove this jumper and connect an external power supply (e.g. +5 V) to pin JD-VCC.**

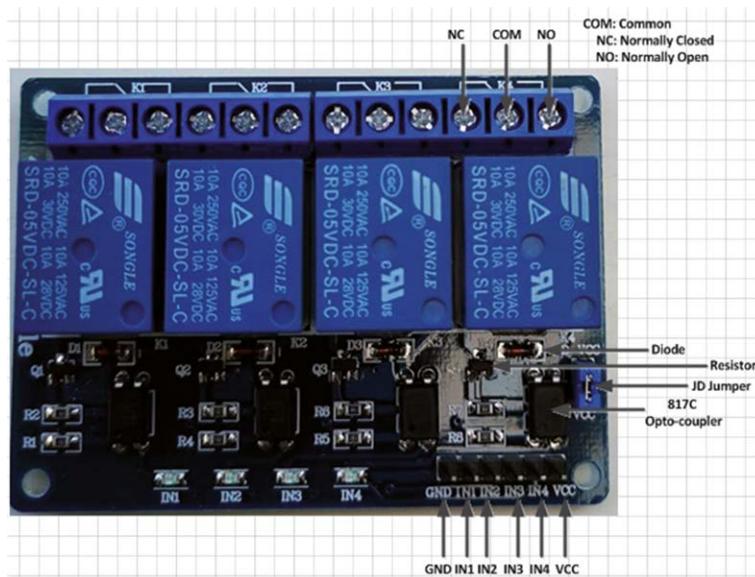


Figure 12.2 4-channel relay board

Figure 12.3 shows the circuit diagram of the project. IN1, IN2, IN3 and IN4 inputs of the relay board are connected to GPIO pins 5, 17, 16, and 4 respectively. Also, GND and +3.3 V pins of the development board are connected to GND and VCC pins of the relay board. You must make sure that jumper JD is removed from the board. Connect an external +5 V power supply to the JD-VCC pin of the relay board.

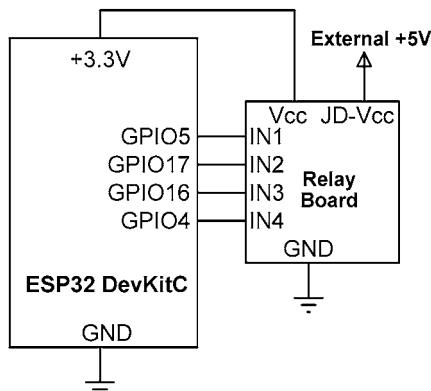


Figure 12.3 Circuit diagram of the project

12.5 The Construction

As shown in Figure 12.4, the ESP32 DevkitC was mounted on a breadboard and connections to the relay board were made using jumper wires.

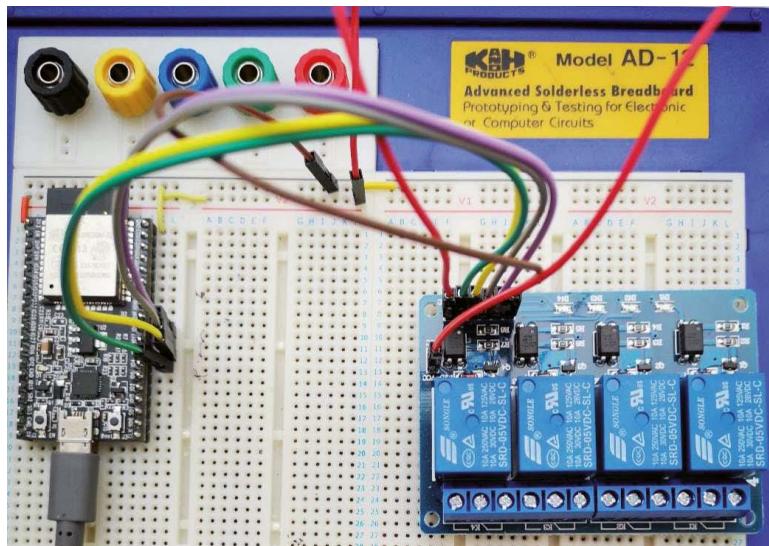


Figure 12.4 Project constructed on a breadboard

12.6 Program Listing

The program listing (program: **WebRelay**) is shown in Figure 12.5.

```
*****  
*          WEB SERVER TO CONTROL 2 LEDs  
*          =====  
*  
* This is a web server program. A 4-channel relay board is  
* connected to the ESP32 DevKitC.2 The project controls the relays  
* from any device that is on the web, for example a PC, tablet, or  
* mobile phone etc. When the IP address of ESP32 is entered on the  
* browser the user will be presented with buttons to control the  
* relays.  
*  
* File: WebRelay  
* Author: Dogan Ibrahim  
* Date: November, 2018  
*  
*****  
  
#include "WiFi.h"  
byte RelayArray[] = {5, 17, 16, 4};                                // Relay interface  
  
//  
// Local Wi-Fi name and password  
//  
const char* ssid = "BTHomeSpot-XNH";                                // SSID  
const char* password = "49345abseb";                                // Password  
WiFiServer server(80);  
  
//  
// The HTML code. This code will display 4 buttons on user's  
// device which can be clicked to control the 4 Relays ON or OFF  
//  
String html ="<!DOCTYPE html> \  
<html> \  
<body> \  
<h1>ESP32 DevKitC RELAY CONTROL</h1>\n<h2>Web Server Example with 4-Channel Relays</h2> \  
<form> \  
<button name=\"RELAY1\" button style=\"background:green\", value=\"ON\" type=\"submit\"><b>RELAY1 ON</b></button> \  
<button name=\"RELAY1\" button style=\"background:red\", value=\"OFF\" type=\"submit\"><b>RELAY1 OFF</b></button><br><br> \  
<button name=\"RELAY2\" button style=\"background:green\", value=\"ON\" type=\"submit\"><b>RELAY2 ON</b></button> \  
<button name=\"RELAY2\" button style=\"background:red\", value=\"OFF\"
```

```
type=\"submit\"><b>RELAY2 OFF</b></button><br><br> \
<button name=\"RELAY3\" button style=\"background:green\", value=\"ON\" type=\"submit\"><b>RELAY3 ON</b></button> \
<button name=\"RELAY3\" button style=\"background:red\", value=\"OFF\" type=\"submit\"><b>RELAY3 OFF</b></button><br><br> \
<button name=\"RELAY4\" button style=\"background:green\", value=\"ON\" type=\"submit\"><b>RELAY4 ON</b></button> \
<button name=\"RELAY4\" button style=\"background:red\", value=\"OFF\" type=\"submit\"><b>RELAY4 OFF</b></button> \
</form> \
</body> \
</html>";

// 
// This function connects the ESP32 Devkitc to the local Wi-Fi
// network. The network name and passwors are as specified earlier
//
void Connect_WiFi()
{
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED)
    {
        delay(1000);
    }
}

// 
// Configure Relays as outputs, turn OFF the Relays to start with,
// Connect to the local Wi-Fi, start the server. Notice that a relay
// is turned OFF when logic HIGH is applied to its input
//
void setup()
{
    for(int k = 0; k < 4; k++)
    {
        pinMode(RelayArray[k], OUTPUT);
        digitalWrite(RelayArray[k], HIGH);
    }
    Connect_WiFi();
    server.begin();
}

//
```

```
// This is the main program loop. Inside the main program we
// check for the client connection and send the HTML file so
// that it is displayed on user's device. The user clicks the
// buttons to control the Relays.
//
void loop()
{
    WiFiClient client=server.available();
    String request = client.readStringUntil('\r');
    client.flush();
    //
    // We control the Relays depending upon the key click. Variable
    // request holds the request and we search this string to see
    // which Relay should be turned ON/OFF. The contents of request
    // is of the form (for example, to turn OFF Relay1):
    // "/?RELAY1=OFF", or similarly, to activate Relay1: "/?RELAY1=ON"
    //
    if(request.indexOf("RELAY1=ON") != -1)digitalWrite(RelayArray[0],LOW);
    if(request.indexOf("RELAY1=OFF") != -1)digitalWrite(RelayArray[0],HIGH);
    if(request.indexOf("RELAY2=ON") != -1)digitalWrite(RelayArray[1],LOW);
    if(request.indexOf("RELAY2=OFF") != -1)digitalWrite(RelayArray[1],HIGH);
    if(request.indexOf("RELAY3=ON") != -1)digitalWrite(RelayArray[2],LOW);
    if(request.indexOf("RELAY3=OFF") != -1)digitalWrite(RelayArray[2],HIGH);
    if(request.indexOf("RELAY4=ON") != -1)digitalWrite(RelayArray[3],LOW);
    if(request.indexOf("RELAY4=OFF") != -1)digitalWrite(RelayArray[3],HIGH);
    //
    // The HTML page to be displayed on user's device
    //
    client.print(html);
}
```

Figure 12.5 Program listing

12.7 Program Description

At the beginning of the program array **RelayArray** defines the pins used to connect the ESP32 to the relay board. The SSID name and password of the local Wi-Fi router are then specified. String **HTML** displays 4 buttons corresponding to the 4 relays when activated from a web browser. The user can click the ON or OFF buttons to activate or deactivate a relay. Inside the setup routine the relay inputs are configured as outputs and are deactivated as their default states at the beginning of the program. The program then connects to the Wi-Fi router and starts the server.

The web server program sends the html file (statement **client.print(html)**) to the client and waits for a command from the client. When a button is clicked on the client, a command is sent to the web server in addition to some other data. Here, we are only interested in the actual command sent. The following commands are sent to the web server for each button pressed:

Button Pressed	Command sent to web server
RELAY1 ON	/?RELAY1=ON
RELAY1OFF	/?RELAY1=OFF
RELAY2 ON	/?RELAY2=ON
RELAY2 OFF	/?RELAY2=OFF
RELAY3 ON	/?RELAY3=ON
RELAY3OFF	/?RELAY3=OFF
RELAY4 ON	/?RELAY4=ON
RELAY4 OFF	/?RELAY4=OFF

The web server program on the ESP32 DevKitC makes a TCP connection and reads the command from the web client as a string (**String request = client.readStringUntil('\r')**). The program then searches the received command to see if any of the above commands are present in the received data. If the sub-string is not found then a -1 is returned. For example, consider the following statement:

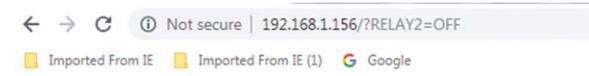
```
if(request.indexOf("RELAY1=ON") != -1) digitalWrite(RelayArray[0], LOW);
```

Here, sub-string **RELAY1=ON** is searched in string **request**. A -1 is returned if the sub-string **RELAY1=ON** does not exists in string **request**. The program looks for a match for all the eight commands and if a command is found then the corresponding Relay is turned ON or OFF accordingly:

```
if(request.indexOf("RELAY1=ON") != -1) digitalWrite(RelayArray[0], LOW);
if(request.indexOf("RELAY1=OFF") != -1) digitalWrite(RelayArray[0], HIGH);
if(request.indexOf("RELAY2=ON") != -1) digitalWrite(RelayArray[1], LOW);
if(request.indexOf("RELAY2=OFF") != -1) digitalWrite(RelayArray[1], HIGH);
if(request.indexOf("RELAY3=ON") != -1) digitalWrite(RelayArray[2], LOW);
if(request.indexOf("RELAY3=OFF") != -1) digitalWrite(RelayArray[2], HIGH);
if(request.indexOf("RELAY4=ON") != -1) digitalWrite(RelayArray[3], LOW);
if(request.indexOf("RELAY4=OFF") != -1) digitalWrite(RelayArray[3], HIGH);
```

In this project, the IP address of the web server was 192.168.1.156. The procedure to test the system is as follows:

- Compile, upload and run the program on the ESP32 DevKitC.
- Open your web browser on your PC and enter the website address 192.168.1.156.
- The form shown in Figure 12.6 will be displayed by the web browser e.g. on your PC screen. The ON buttons are displayed in green and the OFF button in red.
- Click a button, e.g. RELAY1 ON, to turn ON Relay1.



ESP32 DevKitC RELAY CONTROL

Web Server Example with 4-Channel Relays

RELAY1 ON	RELAY1 OFF
RELAY2 ON	RELAY2 OFF
RELAY3 ON	RELAY3 OFF
RELAY4 ON	RELAY4 OFF

Figure 12.6 Form displayed by the web browser

Figure 12.7 show the command sent to the web server when RELAY2 OFF button is clicked.

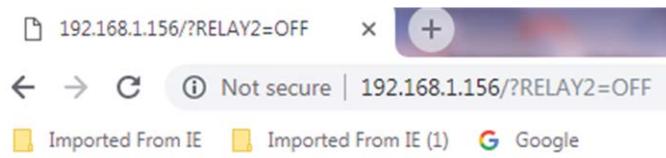


Figure 12.7 Clicking RELAY2 OFF button

12.8 Summary

In this chapter we have seen the development of a web server project with a 4-channel relay. The output contacts of the relay can be connected to high voltage devices such as a microwave, dish washer, washing machine, microwave, etc., so that they can be controlled remotely over a web server interface from a PC or a mobile phone.

In the next project we shall be looking at the development of projects using the Bluetooth module of the ESP32 DevKitC.

Chapter 13 • ESP32 DevKitC Bluetooth programming

13.1 Overview

In the previous chapter we have seen an example web server program controlling a 4-channel relay board.

In this chapter we will be seeing how to program the ESP32 DevKitC for Bluetooth communications by developing projects. Before going into the details of the projects it is worthwhile to look at the features of the Bluetooth briefly.

ESP32 supports the classic Bluetooth as well as the latest Bluetooth Low Energy (BLE) 4.2, thus enabling communication with both old and new Bluetooth phones, tablets, and PCs.

Table 13.1 shows a comparison of the classical Bluetooth and the Bluetooth BLE. Classical Bluetooth has the following features:

- Multi connections
- Up to 3 Mb/s data rate
- Up to 79 channels
- Continuous data streaming
- Device discovery
- Asynchronous data communication
- Master/slave Switch
- Adaptive frequency hopping
- Authentication and encryption
- Secure pairing
- Sniff mode
- Point-to-point network topology

Bluetooth BLE has the following features:

- Multiple connections
- Up to 2 Mb/s data rate
- Up to 40 channels
- Short burst data transmission
- Scanning
- Asynchronous data communication
- Data length extension
- Connection parameter update
- Point-to-point, broadcast, and mesh network topologies

	Bluetooth Low Energy (LE)	Bluetooth Basic Rate/ Enhanced Data Rate (BR/EDR)
Optimized For...	Short burst data transmission	Continuous data streaming
Frequency Band	2.4GHz ISM Band (2.402 – 2.480 GHz Utilized)	2.4GHz ISM Band (2.402 – 2.480 GHz Utilized)
Channels	40 channels with 2 MHz spacing (3 advertising channels/37 data channels)	79 channels with 1 MHz spacing
Channel Usage	Frequency-Hopping Spread Spectrum (FHSS)	Frequency-Hopping Spread Spectrum (FHSS)
Modulation	GFSK	GFSK, π/4 DQPSK, 8DPSK
Power Consumption	~0.0tx to 0.5x of reference (depending on use case)	1 (reference value)
Data Rate	LE 2M PHY: 2 Mb/s LE 1M PHY: 1 Mb/s LE Coded PHY (S=2): 500 Kb/s LE Coded PHY (S=8): 125 Kb/s	EDR PHY (8DPSK): 3 Mb/s EDR PHY (π/4 DQPSK): 2 Mb/s BR PHY (GFSK): 1 Mb/s
Max Tx Power*	Class 1: 100 mW (+20 dBm) Class 1.5: 10 mW (+10 dBm) Class 2: 2.5 mW (+4 dBm) Class 3: 1 mW (0 dBm)	Class 1: 100 mW (+20 dBm) Class 2: 2.5 mW (+4 dBm) Class 3: 1 mW (0 dBm)
Network Topologies	Point-to-Point (including piconet) Broadcast Mesh	Point-to-Point (including piconet)

Table 13.1 Comparison of classical Bluetooth and Bluetooth BLE (Source: Bluetooth.com)

13.2 Bluetooth BLE

Bluetooth Low Energy (BLE) has mainly been developed for battery-operated short distance communications and it consumes less power than the classical Bluetooth. BLE is normally in sleep mode except when a connection is initiated. As a result of this feature, it is suitable for applications that require exchanging small amounts of data at regular intervals, such as in sending/receiving the weather data, in healthcare, security, in home automation, in IoT applications etc.

In Bluetooth BLE applications we have a server and a client. In applications that we are interested in here, ESP32 acts as the server and the PC or a mobile phone act as clients, although this can be changed if required.

The server-client communication is normally done using a point-to-point protocol where the data exchange takes place as follows:

- The server makes itself known to nearby Bluetooth BLE client devices.
- The client devices scan their surroundings and when they find the server that they are looking for they establish connection to the server.
- The client device listens for incoming data from the server.

Other modes of Bluetooth BLE communication are Broadcast, and Mesh. In Broadcast mode the server broadcasts data to a number of connected client devices. In Mesh mode all the BLE devices are connected to each other and they can exchange data. In this chapter we will be developing projects using the point-to-point protocol.

13.2.1 Bluetooth BLE Software Model

Bluetooth BLE devices use the Generic Attributes Profile (GATT) which defines the way that two BLE devices can exchange data. GATT is a hierarchical data structure. As shown in Figure 13.1 GATT hierarchy consists of the Profile, Service, and Characteristics. At the top level we have the Profile which consists of one or more Services. A Service consists of one or more Characteristics and they can make references to other services. A Characteristic contains the actual data and consists of Properties, Values, and Descriptors.

Every Service, Characteristic, and Descriptor in a Profile have unique 16 byte (128-bits) Universally Unique Identifiers (UUID) which identifies a particular service provided by a BLE device. The Bluetooth Special Interest Group (<https://www.bluetooth.com/specifications/gatt/services>) gives lists of shortened UUIDs. For example, the **Battery Service** (used in portable battery operated BLE devices to indicate the current battery level) has the Uniform Type Identifier: **org.bluetooth.service.battery_service** and its assigned number is **0x180F**. Using unique identifiers, any BLE device can find out the battery level, regardless of the manufacturer. Looking at the **Battery Service** we can see that **Battery Level** is a characteristic of this service and its UUID is 0x2A19 (see website: **org.bluetooth.characteristic.battery_level**). The **Characteristic Descriptor** and **Characteristic Presentation Format** are the descriptors of this service and their UUIDs are 0x2902 and 0x2904 respectively. As an example, Figure 13.2 shows the GATT data structure for the **Battery Service**. If your application requires its own UUID, you can generate it from the following website:

<https://www.uuidgenerator.net/>

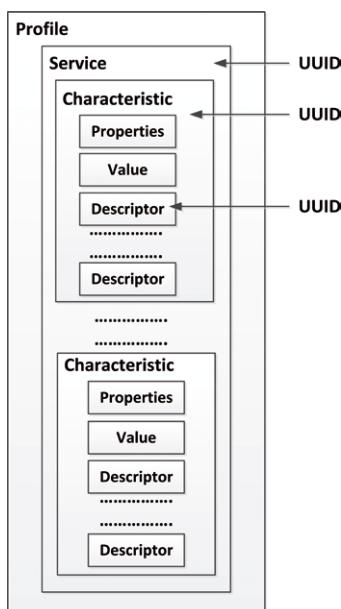


Figure 13.1 The GATT hierarchy



Figure 13.2 GATT data structure for the Battery Service

13.3 PROJECT 1 – Sending Data to Mobile Phone Using Bluetooth BLE

13.3.1 Description

In this project a counter is incremented by one every 3 seconds and its value is sent to a Bluetooth BLE compatible Android mobile phone where it is displayed with an app.

13.3.2 Program Listing

The program listing is shown in Figure 13.3 (program: **BLUECNT**).

```

/*****
 *          SEND DATA TO A BLUETOOTH BLE DEVICE
 * =====
 *
 * In this program a counter is incremented by one in the main
 * program loop every 3 seconds. The counter value is sent to a
 * Bluetooth BLE compatible device, e.g. to a mobile phone.
 *
 * Notice: This program is based on Neil Kolban example for IDF:
 * https://github.com/nkolban/esp32-snippets/blob/master/
 * cpp_utils/tests/BLE%20Tests/SampleNotify.cpp, Ported to Arduino
 * ESP32 by Evandro Copercini
 *
 * File:     BLUECNT
 * Author:   Dogan Ibrahim
 * Date:    November, 2018
 *****/
// Import the necessary BLE libraries
//
#include "BLEDevice.h"
#include "BLEServer.h"
#include "BLEUtils.h"

//
```

```
// Define SERVICE and CHARACTERISTIC UUIDs
//
#define SERVICE_UUID      "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"
BLECharacteristic *pCharacteristic;           // Pointer to BLE Char
bool DeviceConnected = false;                  // BLE connection status
int Count = 0;                                // Initialize Count
char strcnt[10];                             // Array to store Count

//
// Define the callback class with the DeviceConnected status such
// that the status is set true when a connection is established
//
class MyServerCallbacks: public BLEServerCallbacks
{
    void onConnect(BLEServer* pServer)
    {
        DeviceConnected = true;
    }

    void onDisconnect(BLEServer* pServer)
    {
        DeviceConnected = false;
    }
};

void setup()
{
    // Define our BLE name as ESP_CNT (can be changed if required)
    BLEDevice::init("ESP_CNT");

    // Create BLE Server
    BLEServer *pServer = BLEDevice::createServer();

    // Server callback function
    pServer->setCallbacks(new MyServerCallbacks());

    // Create BLE Service
    BLEService *pService = pServer->createService(SERVICE_UUID);

    // Create BLE Characteristic
    pCharacteristic = pService->createCharacteristic(
        CHARACTERISTIC_UUID,
        BLECharacteristic::PROPERTY_WRITE |
```

```
        BLECharacteristic::PROPERTY_READ |  
        BLECharacteristic::PROPERTY_NOTIFY);  
  
    // Start the BLE service  
    pService->start();  
  
    // Start advertising  
    pServer->getAdvertising()->start();  
}  
  
//  
// This is the main program loop. If the Bluetooth BLE compatible  
// device is connected then increment Count and send its value to  
// the connected device. This loop is repeated every 3 seconds  
//  
void loop()  
{  
    if(DeviceConnected)                      // If connected...  
    {  
        Count++;                          // Increment Count  
        itoa(Count,strcnt,10);            // Convert to string  
        pCharacteristic->setValue(strcnt); // Set its value  
        pCharacteristic->notify();       // Notify it  
    }  
    delay(3000);                          // Wait 3 seconds  
}
```

Figure 13.3 Program listing

13.3.3 Program PDL

Figure 13.4 shows the operation of the program as a PDL.

```
BEGIN  
    Define Service and Characteristic UUIDs  
    Initialize Count to 0  
    Create a BLE server  
    Define a BLE callback function  
    Create a BLE service  
    Create the BLE Characteristics and its Descriptor  
    Start the BLE service  
    Advertise the BLE  
DO FOREVER  
    IF connected to a BLE device THEN  
        Increment Count  
        Convert Count into a string  
        Set value to this string  
        Notify the connected BLE
```

```

ENDIF
ENDDO
END

```

Figure 13.4 Program PDL

13.3.4 Program Description

At the beginning of the program the header files used by the BLE are included in the program. The **Service** UUID and **Characteristic** UUID are then defined (you can generate new UUIDs as described in Section 13.2.1). Variable **Count** is initialised to 0. Inside the **setup** routine the BLE name is set to **ESP_CNT**, BLE is created, the server callback function is set, and the BLE service is created. The **BLE Characteristic** is configured by specifying its UUID, and the properties are set to **NOTIFY**, **READ** and **WRITE**. Lastly the BLE service is started and is then advertised.

The remainder of the program runs inside the main program loop. Here, the program checks to see if a Bluetooth BLE device is connected, and if so, the **Count** is incremented by one, it is converted into a string, and is stored in character array **strcnt** using function **itoa**. **strcnt** is then set as a **value** and then sent as a notification to the connected BLE device.

13.3.5 Testing the Program

The program can be tested easily using an Android mobile phone. There are several free of charge apps that can be used to test the program, such as the **nRF Connect for Mobile** (by Nordic Semiconductor ASA), **Bluetooth Monitor** (by Tis Veugen), and so on. Here, the Bluetooth Monitor is used.

Figure 13.5 shows the installation screen of the **Bluetooth Monitor**. The steps to test the program are given below:

- Compile and load the program to the ESP32.
- Open the **Bluetooth Monitor** on your Arduino.
- Click **Scan devices** on your mobile phone.
- You should see **ESP_CNT** displayed. Click on it.
- You should see a screen showing that the device is connected.
- Click on **Unknown service** with the UUID: 4fafc201-1fb5-459e-8fcc-c5c9c331914b.
- Click on **Unknown characteristic** (in blue colour). You should see the data send by your ESP32 displayed at the upper section of the screen as shown in Figure 13.6 (**61** in this example). The data is displayed both in decimal and in hexa-decimal formats. Keep clicking on the **Unknown Characteristic** to see the data sent by the ESP32.



Rate this app

Figure 13.5 Bluetooth Monitor

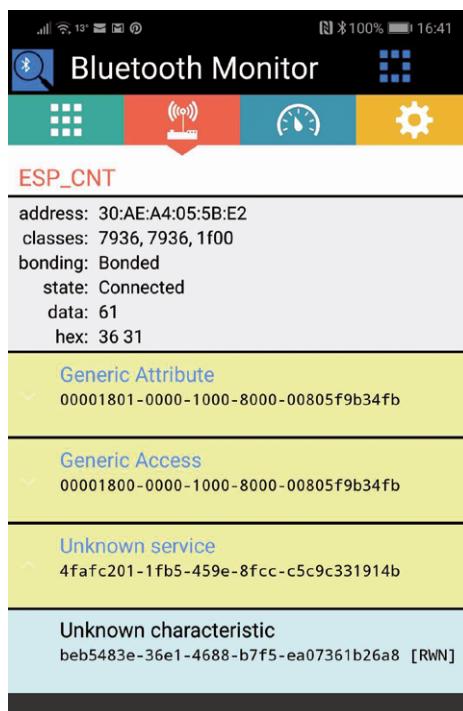


Figure 13.6 Displaying the data sent by the ESP32

13.4 PROJECT 2 – 4-Channel Relay Control Using Bluetooth BLE

13.4.1 Description

In this project a 4-channel relay board is connected to the ESP32 DevKitC and the relay is controlled by sending commands from a Bluetooth BLE apps running on an Arduino mobile phone. The following commands can be sent from the mobile phone:

1=ON	Activate Relay1
1=OFF	De-activate Relay1
2=ON	Activate Relay2
2=OFF	De-activate Relay2
3=ON	Activate Relay3
3=OFF	De-activate Relay3
4=ON	Activate Relay4
4=OFF	De-activate Relay4

13.4.2 The Aim

The aim of this project is to show how a 4-channel relay board can be connected to the ESP32 DevKitC and also how it can be controlled using the Bluetooth BLE communication with a mobile phone.

13.4.3 The Block Diagram

The block diagram of the project is shown in Figure 13.7 where a 4-channel relay board is directly connected to the ESP32 DevkitC GPIO pins.

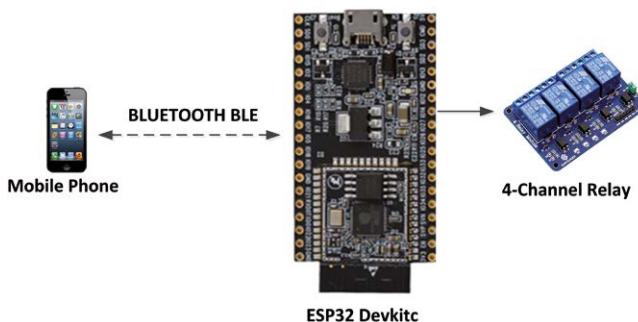


Figure 13.7 Block diagram of the project

13.4.4 Circuit Diagram

The circuit diagram of the project is as in Figure 12.3 where the 4-channel relay inputs IN1, IN2, IN3 and IN4 are connected to GPIO pins 5, 17, 16 and 4, respectively.

13.4.5 Program PDL

Figure 13.8 shows the operation of the program as a PDL.

```
BEGIN
    Define Service and Characteristic UUUIDs
    Initialize RelayArray as outputs
    De-activate all 4 relays
    Create a BLE server
    Define a BLE callback function
    Create a BLE service
    Create the BLE Characteristics and its Descriptor
    Start the BLE service
    Advertise the BLE

DO FOREVER
    IF command received THEN
        IF the command is n=ON THEN
            Activate Relay n
        ELSE IF command is n=OFF THEN
            De-activate Relay n
        ENDIF
    ENDIF
ENDDO
END
```

Figure 13.8 Program PDL

13.4.6 Program Listing

The program listing is shown in Figure 13.9 (program: **BLERELAY**).

```
*****
*          BLUETOOTH BLE 4-CHANNEL RELAY CONTROL
* =====
*
* In this program a 4-channel relay board is connected to input pins
* 5,17,16, and 4 of the ESp32 DevKitC. The program receives commands from
* a Bluetttooth BLE compatible device (e.g. mobile phone) and then
* controls the relays as requested.
*
* This program is based on Neil Kolban example for IDF:
* https://github.com/nkolban/esp32-snippets/blob/master/cpp_utils/tests
* /BLE%20Tests/SampleNotify.cpp, Ported to Arduino ESP32 by
* Evandro Copercini
*
* Program: BLERELAY
* Date   : November, 2018
* Author : Dogan Ibrahim
*****
```

```

#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>

byte RelayArray[] = {5, 17, 16, 4}; // Relay interface
BLECharacteristic *pCharacteristic;

// Define Service UUID and Characteristic UUID
//
#define SERVICE_UUID "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"

//
// This routine is called when data is received from the connected
// BLE device. The data is in the following format: n=ON or n=OFF
// where n=1,2,3,4 corresponding to the relay number. RXholds the
// received data
//
class MyCallbacks: public BLECharacteristicCallbacks
{
    void onWrite(BLECharacteristic *pCharacteristic)
    {
        std::string RX = pCharacteristic->getValue();

        if (RX.length() >= 4)
        {
            if(RX[1] == '=' && RX[2] == '0' && RX[3] == 'N')
            {
                for(int i = 0; i < 4; i++)
                {
                    if(RX[0] == i+1+'0')digitalWrite(RelayArray[i], LOW);
                }
            }
            else if(RX[1] == '=' && RX[2] == '0' && RX[3] == 'F' && RX[4] == 'F')
            {
                for(int i = 0; i < 4; i++)
                {
                    if(RX[0] == i+1+'0')digitalWrite(RelayArray[i], HIGH);
                }
            }
        }
    }
};

```

```
//  
// Configure Relays as outputs, turn OFF the Relays to start with.  
// Notice that a relay is turned OFF when logic HIGH is applied to its input  
//  
void setup()  
{  
    for(int k = 0; k < 4; k++)  
    {  
        pinMode(RelayArray[k], OUTPUT);           // Relays are outputs  
        digitalWrite(RelayArray[k], HIGH);         // De-activate relays  
    }  
  
    // Create the BLE Device  
    BLEDevice::init("BLE_RELAY");  
  
    // Create the BLE Server  
    BLEServer *pServer = BLEDevice::createServer();  
  
    // Create the BLE Service  
    BLEService *pService = pServer->createService(SERVICE_UUID);  
  
    BLECharacteristic *pCharacteristic = pService->createCharacteristic(  
        CHARACTERISTIC_UUID,  
        BLECharacteristic::PROPERTY_READ  
    );  
  
    pCharacteristic->setCallbacks(new MyCallbacks());  
  
    // Start the service  
    pService->start();  
  
    // Start advertising  
    pServer->getAdvertising()->start();  
}  
  
void loop()  
{  
}
```

Figure 13.9 Program listing

13.4.7 Program Description

At the beginning of the program, the Bluetooth BLE header files used by the program are included and array **RelayArray** defines the pins used to connect the ESP32 to the relay board. The **Service** UUID and **Characteristic** UUID are then defined. Inside the **setup** routine the relay inputs are configured as outputs and the relays are deactivated at the beginning of the program. Then our Bluetooth BLE device is given the name **BLE_RELAY**, BLE server is created, BLE service is created, and the BLE Characteristics are defined. The BLE service is started and advertised. The program then waits to receive data from a connected BLE device. When data is received the program executes the code inside the **MyCallbacks** routine. Here, the correct relay is controlled based on the received data. For example, receiving data **1=ON** activates Relay1 and so on.

13.4.8 Testing the Program

In this project the Arduino app **nRF Connect for Mobile** (by Nordic Semiconductor ASA) is used to send data to the ESP32 DevKitC over a Bluetooth BLE interface. This app is available free of charge in Google Play as shown in Figure 13.10.

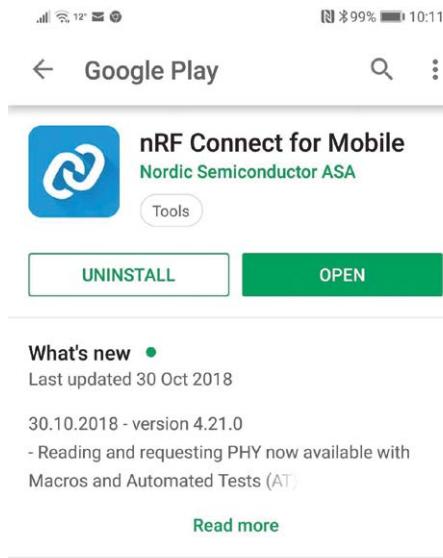


Figure 13.10 nRF Connect for Mobile

The steps to test the program are given below:

- Install apps **nRF Connect for Mobile** on your Android mobile phone.
- Compile and upload your program to the ESP32 DevKitC.
- Start the apps on your mobile phone. You should see ESP32 DevKitC displayed as device **BLE_RELAY**, as shown in Figure 13.11. Click **CONNECT** to connect to your device. You should see the CONNECTED message displayed.

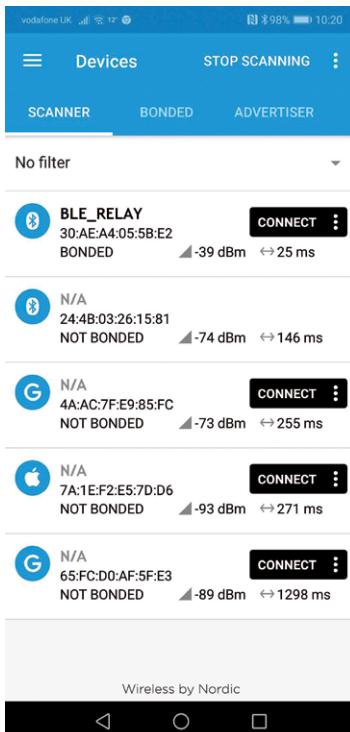


Figure 13.11 Device BLE_RELAY listed

- Click on Unknown Service and then click on the up arrow to send data to the ESP32.
- Click next to BYTE to select TEXT.
- As an example, enter **1=ON** to activate Relay1 (see Figure 13.12).

```
/*
 *          SERIAL COMMUNICATION OVER BLUETOOTH
 *          =====
 *
 * In this program the message Hello from DevKitC is sent over
 * Bluetooth serial link every 3 seconds
 *
 * File:    SERIALBLUE
 * Author: Dogan Ibrahim
 * Date:   November, 2018
 *
 *****/

```

```
#include"BluetoothSerial.h"
```

```

BluetoothSerial BT;

void setup()
{
    BT.begin("MyESP");
}

void loop()
{
    BT.println("Hello from ESP32 DevKitC");
    delay(3000);
}

```

Figure 13.12 Testing the program

13.5 PROJECT 3 – Serial Communication Over Bluetooth Classic

13.5.1 Description

In this project data is sent using serial communication over Bluetooth classic. The message **Hello from ESP32 DevKitC** is sent every three seconds.

13.5.2 The Aim

The aim of this project is to show how data can be sent via serial communication operating over Bluetooth classic.

13.5.3 Program Listing

The program listing is shown in Figure 13.13 (program: **SERIALBLUE**).

```

*****
*           SERIAL COMMUNICATION OVER BLUETOOTH
*           =====
*
* In this program the message Hello from DevKitC is sent over
* Bluetooth serial link every 3 seconds
*
* File:   SERIALBLUE
* Author: Dogan Ibrahim
* Date:   November, 2018
*
*****
#include"BluetoothSerial.h"
BluetoothSerial BT;

void setup()
{
    BT.begin("MyESP");
}

```

```
}

void loop()
{
    BT.println("Hello from ESP32 DevKitC");
    delay(3000);
}
```

Figure 13.13 Program listing

13.5.4 Program Description

At the beginning of the program the header file **BluetoothSerial.h** is included in the program. Make sure that you have the latest version of the ESP32 software for Arduino since the Bluetooth Serial library has been added recently to the ESP32 Arduino IDE. Inside the **setup** routine the Bluetooth device is initialised with the name **MyESP**. The remainder of the program runs in an endless loop where the message **Hello from ESP32 DevKitC** is sent every 3 seconds.

Notice that we can use function **BT.read()** to read data from the serial port. Function **BT.available()** can be used to check if data is available in the input buffer before attempting to read.

13.5.5 Testing the Program on the PC

The program can be tested on the PC as follows:

- Compile and upload the program to your ESP32 DevKitC.
- Open Bluetooth on your PC and add Bluetooth device MyESP.
- Open Bluetooth settings on your PC and click to display the COM Ports (see Figure 13.14).

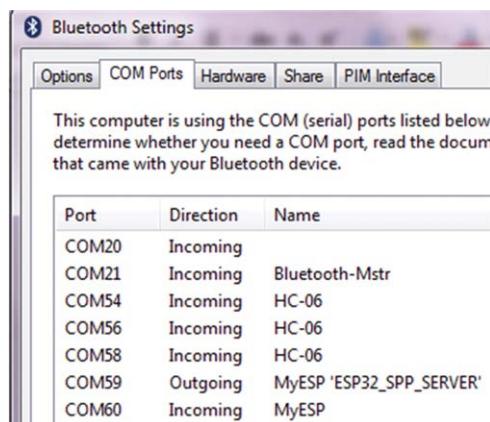


Figure 13.14 Display the COM ports

- Notice the **Outgoing** COM port number for our device (COM59 in our example).
- Start the **Putty** terminal emulation program, select **Serial**, Serial line: **COM59**, Speed: **9600**. Alternatively open the Serial Monitor with the correct COM port number.
- You should see the message **Hello from ESP32 DevKitC** displayed every 3 seconds (see Figure 13.15).

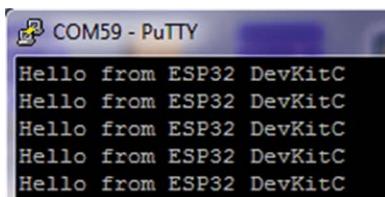


Figure 13.15 Messages displayed on the Putty terminal

13.5.6 Testing the Program on Mobile Phone

The program can be tested on an Android mobile phone as follows:

- Add a Bluetooth classic app to your Android mobile phone. In this project the free of charge app called **Serial Bluetooth Terminal** (by Kai Morich) shown in Figure 13.16 is used.

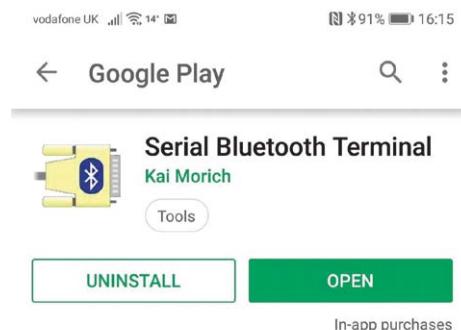


Figure 13.16 Serial Bluetooth Terminal

- Connect your ESP32 DevKitC to a power supply (e.g. PC USB port or a portable power supply).

- Open the Bluetooth folder on your mobile phone and pair with **MyESP** if requested. Delete any older pairings if not needed.
- Install and open the **Serial Bluetooth Terminal** on your mobile phone.
- Click **Devices** from the menu and then click on device **MyESP**. You should see the **Connected** message displayed.
- You should now see the message **Hello from ESP32 DevKitC** displayed on your mobile phone every 3 seconds as shown in Figure 13.17.

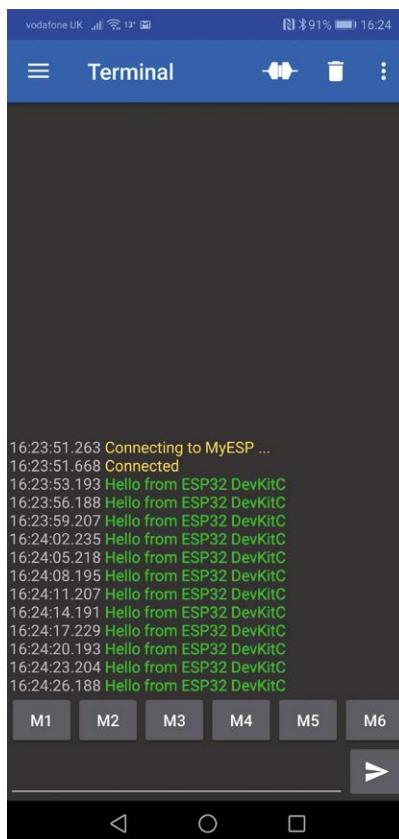


Figure 13.17 Message is displayed on your mobile phone

13.6 PROJECT 4 – 4-Channel Relay Control Using Bluetooth Classic

13.6.1 Description

In this project a 4-channel relay board is connected to the ESP32 DevKitC as in Section 13.4 and the relay is controlled by sending commands from a Bluetooth classic app running on an Arduino mobile phone. The following commands can be sent from the mobile phone (# character is the command terminator):

1=ON#	Activate Relay1
1=OFF#	De-activate Relay1
2=ON#	Activate Relay2
2=OFF#	De-activate Relay2
3=ON#	Activate Relay3
3=OFF#	De-activate Relay3
4=ON#	Activate Relay4
4=OFF#	De-activate Relay4

13.6.2 The Aim

The aim of this project is to show how a 4-channel relay board can be connected to the ESP32 DevKitC and also how it can be controlled using the Bluetooth classic communication with a mobile phone.

13.6.3 The Block Diagram

The block diagram of the project is as in Figure 13.7

13.6.4 Circuit Diagram

The circuit diagram of the project is as in Figure 12.3, where the 4-channel relay inputs IN1, IN2, IN3 and IN4 are connected to GPIO pins 5, 17, 16 and 4 respectively.

13.6.5 Program Listing

The program listing is shown in Figure 13.18 (program: **BTRELAY**).

```
*****
*          BLUETOOTH CLASSIC 4-CHANNEL RELAY CONTROL
*          =====
*
* In this program a 4-channel relay board is connected to input pins
* 5,17,16, and 4 of the ESp32 DevKitC. The program receives commands from
* a Bluetooth classic compatible mobile phone and controls the relays
*
* Program: BTRELAY
* Date   : November, 2018
* Author : Dogan Ibrahim
*****
#include"BluetoothSerial.h"
BluetoothSerial BT;

char rcv[10];                                // Receive buffer
byte RelayArray[] = {5, 17, 16, 4};           // Relay interface

//
// This function activates or de-activates the relays depending on
// received command in character array rcv. Command n=ON# activates
// Relayn, command n=OFF# de-activates Relayn. # is the terminator
```

```
//  
void RelayControl()  
{  
    if(rcv[1] == '=' && rcv[2] == '0' && rcv[3] == 'N')  
    {  
        for(int i = 0; i < 4; i++)  
        {  
            if(rcv[0] == i+1+'0')  
            {  
                digitalWrite(RelayArray[i], LOW);  
                BT.printf("Relay%d is activated\n\r", i+1);  
            }  
        }  
    }  
    else if(rcv[1] == '=' && rcv[2] == '0' && rcv[3] == 'F' && rcv[4] == 'F')  
    {  
        for(int i = 0; i < 4; i++)  
        {  
            if(rcv[0] == i+1+'0')  
            {  
                digitalWrite(RelayArray[i], HIGH);  
                BT.printf("Relay%d is de-activated\n\r", i+1);  
            }  
        }  
    }  
}  
  
//  
// Configure Relays as outputs, turn OFF the Relays to start with.  
// Notice that a relay is turned OFF when logic HIGH is applied to its input  
//  
void setup()  
{  
    for(int k = 0; k < 4; k++)  
    {  
        pinMode(RelayArray[k], OUTPUT);           // Relays are outputs  
        digitalWrite(RelayArray[k], HIGH);         // De-activate relays  
    }  
    BT.begin("ESPRelays");                   // Initialize  
}  
  
void loop()  
{  
    int dummy, k = 0, flag = 0;
```

```

while(1)
{
    while(flag == 0)                                // while not terminator
    {
        if(BT.available())                          // If data available
        {
            rcv[k] = BT.read();                    // Read byte
            if(rcv[k] == '#') flag = 1;           // If terminator
            k++;                                 // Increment buffer count
        }
    }
    k = 0;                                         // Zero for next command
    if(flag == 1)
    {
        while(BT.available()) dummy = BT.read(); // Flush receive buffer
        flag = 0;                                // Ready for next command
        RelayControl();                         // Decode command
    }
}

```

Figure 13.18 Program listing

13.6.6 Program Description

At the beginning of the program header file **BluetoothSerial.h** is included in the program, receive buffer **rcv** is defined, and connections between the relay board and ESP32 DevKitC are defined in byte array **RelayArray**. Inside the **setup** routine all the relays are configured as outputs and they are de-activated. The serial Bluetooth is then initialised to the name **ESPRelays**. Inside the main program loop the program reads data from the mobile phone until the terminator character **#** is detected. The input buffer is then flashed so that any remaining characters in the input buffer are deleted. The program then calls function **RelayControl** to activate or deactivate the relays based on the received command in buffer **rcv**. Every valid command is confirmed by sending a message to the mobile phone. For example, if the received command is **1=ON#** then **Relay1** is activated and message **Relay1 is activated** is sent to the mobile phone. Similarly, command **1=OFF#** de-activated **Relay1** and so on for all the other relays. The main program loop then repeats, waiting for a new command from the mobile phone.

13.6.7 Testing the Program on Mobile Phone

The Android app installed in the previous project can also be used in this project.

Connect your ESP32 DevKitC to a power supply (e.g. PC USB port or a portable power supply).

- Open the Bluetooth folder on your mobile phone and pair with **ESPRelay** if requested. Delete any older pairings if not needed.
 - Install and open the **Serial Bluetooth Terminal** on your mobile phone.

- Click **Devices** from the menu and then click on device **ESPRelays**. You should see the **Connected** message displayed.
- Press and hold button M1 to save a command on the mobile phone. Enter the command, e.g. 1=ON# as shown in Figure 13.19 and click the tick at the top right hand side of the screen. Repeat to save other commands in M2, M3, M4 etc.

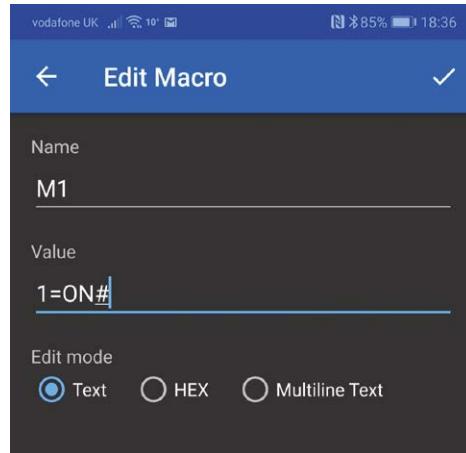


Figure 13.19 Save **1=ON#** command in button M1

- Click button M1 to send command 1=ON# to the ESP32 DevKitC. Notice that a confirmation message is displayed on the mobile phone screen for every command sent.
- Figure 13.20 shows some example commands sent to the ESP32 DevKitC.

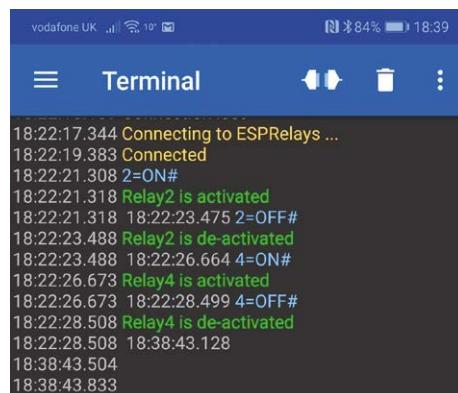


Figure 13.20 Example commands

13.7 PROJECT 5 – Sending Temperature and Humidity to Mobile Phone Using Bluetooth Classic

13.7.1 Description

In this project the ambient temperature and humidity are read from a sensor and are then sent to a mobile phone using Bluetooth classic every 5 seconds.

13.7.2 The Aim

The aim of this project is to show how the temperature and humidity readings can be sent to a mobile phone using Bluetooth classic.

13.7.3 The Block Diagram

The block diagram of the project is shown in Figure 13.21. A DHT11 type temperature and humidity sensor is used in this project.

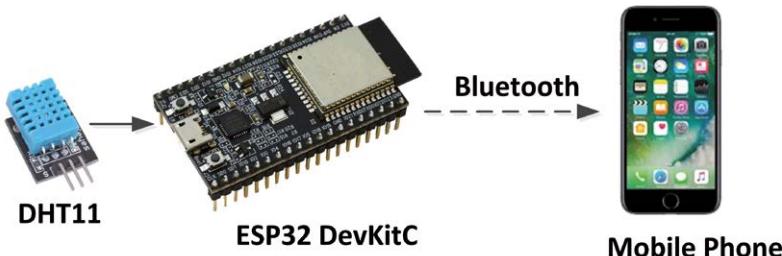


Figure 13.21 Block diagram of the project

13.7.4 Circuit Diagram

The circuit diagram of the project is as in Figure 5.12, where the output of sensor DHT11 is connected to GPIO port 23.

13.7.5 Program Listing

The program listing is shown in Figure 13.22 (program: **BTDHT11**).

```

/****************************************************************************
 * SENDING TEMPERATURE & HUMIDITY TO MOBILE PHONE USING BLUETOOTH CLASSIC
 * =====
 *
 * In this program a DHT11 type temperature and humidity sensor chip
 * is connected to the ESP32 DevKitC. The program sends the temperature
 * and humidity readings to a Bluetooth classic compatible mobile phone
 * every 5 seconds
 *
 * Program: BTDHT11
 * Date   : November, 2018
 * Author : Dogan Ibrahim
 */
#include "Adafruit_Sensor.h"

```

```
#include "DHT.h"
#define DHT11_PIN 23                                     // DHT11 on pin 23
#define DHTTYPE DHT11

#include"BluetoothSerial.h"
BluetoothSerial BT;
DHT dht(DHT11_PIN, DHTTYPE);

// Initialize the DHT11 and BluetoothSerial
//
void setup()
{
    dht.begin();                                         // Initialize DHT11
    BT.begin("ESPDHT11");                                // Initialize
}

void loop()
{
    while(1)
    {
        float humidity = dht.readHumidity();             // Read humidity
        float temperature = dht.readTemperature();         // Read temperature
        BT.printf("Temperature=%5.2f\n\r", temperature);   // Send temperature
        BT.printf("Humidity=%5.2f\n\r", humidity);          // Send humidity
        delay(5000);                                       // Wait 5 seconds
    }
}
```

Figure 13.22 Program listing

13.7.6 Program Description

The DHT11 temperature and humidity sensor chip used in the project (see Section 5.3, Project 2). At the beginning of the program **DHT** and **BluetoothSerial** header files are included in the program (see Section 5.3 on how to install the DHT libraries). Inside the **setup** routine the DHT11 and BluetoothSerial are initialised where the ESP32 is given the name **ESPDHT11**. The main program runs in an endless loop where the temperature and humidity readings are read from the DHT11 sensor chip every 5 seconds and are then sent to the mobile phone over a Bluetooth classic link using **printf** statements.

13.7.7 Testing the Program on a Mobile Phone

- The Android app installed in the previous project can also be used in this project.
- Connect your ESP32 DevKitC to a power supply (e.g. PC USB port or a portable power supply).

- Open the Bluetooth folder on your mobile phone and pair with **ESPDHT11** if requested. Delete any older pairings if not needed.
- Install and open the **Serial Bluetooth Terminal** on your mobile phone.
- Click **Devices** from the menu and then click on device **ESPDHT11**. You should see the **Connected** message displayed.
- Select **Terminal** from the menu in your app. You should see the temperature and humidity displayed every 5 seconds as shown in Figure 13.23.

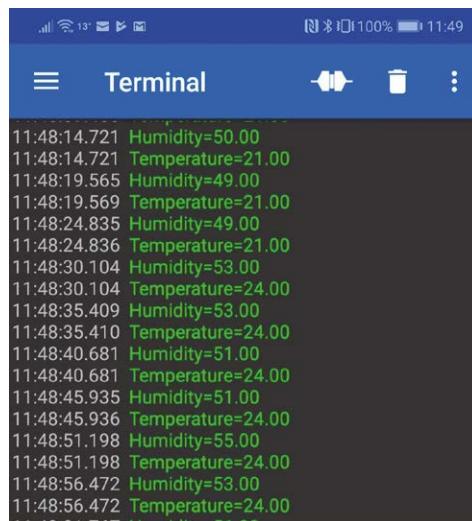


Figure 13.23 Displaying the temperature and humidity

13.8 Summary

The Bluetooth module is one of the nicest features of the ESP32 processor. In this chapter we have developed projects using both the Bluetooth Low Energy (BLE) programming and the Bluetooth classical programming.

In the next chapter we will be installing the microPython to our ESP32 DevKitC and develop some projects using this powerful and popular programming language.

Chapter 14 • Using Micropython with the ESP32 DevKitC

14.1 Overview

Up to now we have been developing programs for our ESP32 DevKitC using the Arduino IDE environment. The ESP32 processor can also be programmed with:

ESP – IDF: This is the native development environment from Espressif Systems, developed for the ESP32 processors. This development environment is based on FreeRTOS and enables the user to access all modules of the ESP32 processor, including the Bluetooth. Although the ESP – IDF is very powerful it is rather difficult to write programs using this environment.

Mongoose OS: This is an open source operating system supporting the ESP32, ESP8266, STM32 etc., developed for the Internet of Things applications

Arduino IDE: We have seen using this development environment with the ESP32 processor

Lua-RTOS: Lua-RTOS is a real-time operating system based on the Lua language which has been designed to run on embedded systems. Lua-RTOS is currently available for the ESP32 processor. Lua-RTOS can be programmed in two ways: using the Lua language directly or using a block-based programming language that translates blocks to Lua.

MicroPython: This is currently one of the most popular programming languages used in most universities throughout the world in introductory programming courses.

In this chapter we will be installing the MicroPython on our ESP32 DevKitC and also develop some simple projects using the MicroPython.

14.2 Installing MicroPython on ESP32 DevKitC

Before installing MicroPython on our ESP32 DevKitC we have to install Python 2.7 to our PC. The steps for this are given below:

- Download and install Python 2.7 on your PC from the following website (see Figure 14.1 and Figure 14.2). The Python files will be loaded into folder C:\Python27

<https://www.python.org/downloads/>



Figure 14.1 Install Python 2.7



Figure 14.2 Python 2.7 being installed

- Make Python accessible from anywhere on the PC by configuring the PATH environment variable. **Click Control Panel -> System** and then click **Advanced system settings** as shown in Figure 14.3.

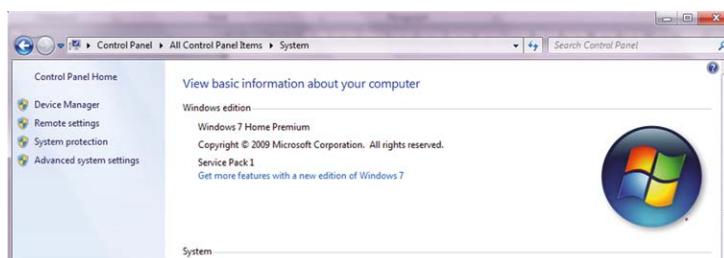


Figure 14.3 Click Advanced system settings

- Click **Advanced -> Environment Variables** as shown in Figure 14.4

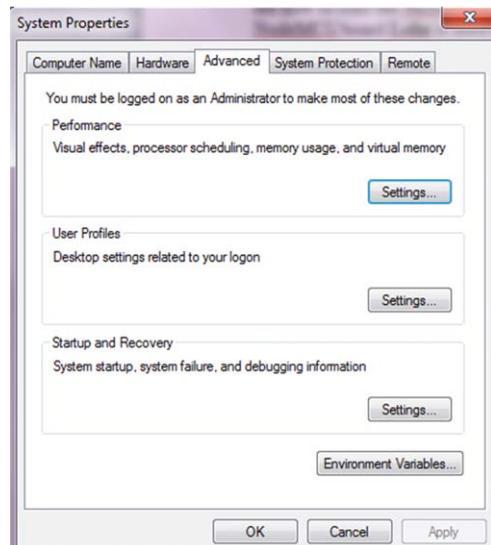


Figure 14.4 Click Environment Variables

- Click **Path** in **System variables** as shown in Figure 14.5

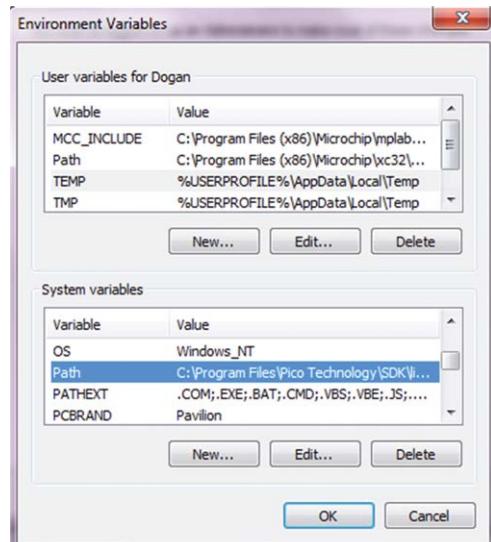


Figure 14.5 Click Path in System variables

- Click **Edit** and add the following statement to the end of the line: ;**Python27** as shown in Figure 14.6. Click OK to save and exit the Environment settings.

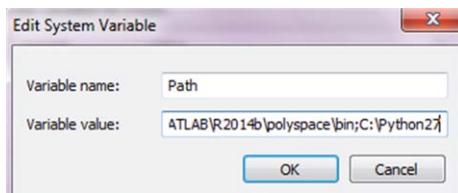


Figure 14.6 Add python27

- Python should now accessible from anywhere on your PC. Start the command line on your PC by clicking **Start** and then enter **cmd**. Type **Python** to start the Python interpreter and you should see a screen similar to the one shown in Figure 14.7.

Figure 14.7 Start Python on your PC

- Enter the statement **print ("Hello there")** and you should see the string **Hello there** displayed on your screen. Exit from Python by entering **Cntl-Z** followed by Enter.
- This shows that the Python has been successfully installed on your PC. Now we have to continue with the steps to install the ESP32 into Arduino IDE.

We also need a software called esptool on our PC so that we can flash the ESP32 with the MicroPython binaries. The steps to install esptool to our PC are given below:

- Start the command line on your PC and go to the root directory by entering **CD** (see Figure 14.8)

Figure 14.8 Go to the root directory

- Enter the following command to install the latest version of the **esptool** on your PC (see Figure 14.9). Note that it must be at least version 2.0 since version 1.x of esptool does not support the ESP32 processor. You should see the message **Successfully installed esptool** at the end of the installation.

```
C:\> pip install esptool
or,
C:\> python -m install esptool

>>> import network
>>> net=network.WLAN(network.STA_IF)
>>> net.isconnected()
False
>>> net.active(True)
I (1524367) wifi: mode : sta (30:ae:a4:05:b8:e0)
I (1524367) wifi: STA_START
True
>>> net.connect("BTHomeSpot-XNH","49345abaeb")
>>> I (1561557) wifi: n:6 0, o:1 0, ap:255 255, sta:6 0, prof:1
I (1562117) wifi: state: init -> auth (b0)
I (1562127) wifi: state: auth -> assoc (0)
I (1562137) wifi: state: assoc -> run (10)
I (1562157) wifi: connected with BTHomeSpot-XNH, channel 6
I (1562157) wifi: event 4
I (1564817) event: ip: 192.168.1.156, mask: 255.255.255.0, gw: 192.168.1.254
I (1564817) wifi: GOT_IP
I (1572137) wifi: pm start, type:0

>>> net.ifconfig()
('192.168.1.156', '255.255.255.0', '192.168.1.254', '192.168.1.254')
>>> 
```

Figure 14.9 Installing the esptool

- It is recommended to erase the entire flash memory of the ESP32 before uploading new firmware. To do this, connect the ESP32 DevKitC to the PC using the mini USB cable. Find out the serial port number where the DevKitC is connected to from the Device Manager, and enter the following command. In this example it is assumed that the DevKitC is connected to serial port COM61. Figure 14.10 shows the command being executed to erase the flash memory. The erasure processes should take about 10 seconds. Make sure that you keep pressing the BOOT button while the erasure process continues.

```
C:\> esptool.py --port COM61 erase_flash
or,
C:\> python c:\python27\lib\site-packages\esptool.py --port COM61
erase_flash

C:\>python c:\python27\lib\site-packages\esptool.py --port COM47 erase_flash
esptool.py v2.0.1
Connecting....._____
Detecting chip type... ESP32
Chip is ESP32DWDQ6 (Revision 0)
Uploading stub...
Running stub...
Stub running...
Erasing flash <this may take a while>...
Chip erase completed successfully in 2.7s
Hard resetting...
```

Figure 14.10 Erase the entire flash memory of the ESP32 DevKitC

- We will now need to copy the latest MicroPython binary file for ESP32 to a folder. At the time of writing this book the latest version was called: esp32-20181112-v1.9.4-683-gd94aa577a.bin. Click on the file and save it to a folder on your PC. This file can be found at the following link. In this example we have copied it to folder C:\ESP32 (see Figure 14.11).
-

www.micropython.org/download

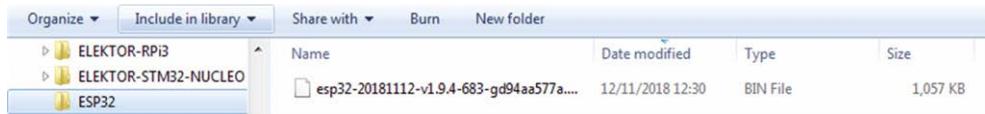


Figure 14.11 Copy the MicroPython binary file to a folder

- We are now ready to upload the MicroPython firmware to our ESP32 DevKitC development board. Make sure that you press the BOOT button while the copy process continues. Also, make sure that the DevKitC is still connected to the USB port and enter the following command (see Figure 14.12). The flash memory size will be detected automatically and the MicroPython firmware will be uploaded to the ESP32 DevKitC:

```
C:\> esptool.py --port COM61 --baud 460800 write_flash -fm dio 0x1000
C:\ESP32\esp32-20170803-v1.9.1-394-g79feb956.bin
```

or,

```
C:\> python c:\python27\lib\site-packages\esptool.py --port
COM61 --baud 460800 write_flash -fm dio 0x1000 C:\ ESP32\esp32-
20170803-v1.9.1- 394-g79feb956.bin
```

```
C:\>python c:\python27\lib\site-packages\esptool.py --port COM61 --baud 460800 w
rite_flash -fm dio 0x1000 c:\esp32\esp32-20181112-v1.9.4-683-gd94aa577a.bin
esptool.py v2.0.1
Connecting...
Detecting chip type... ESP32
Chip is ESP32D0WDQ5 (revision 1)
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 460800
Changed.
Configuring flash size...
Auto-detected Flash size: 4MB
Compressed 1082256 bytes to 684127...
Wrote 1082256 bytes (684127 compressed) at 0x000001000 in 16.7 seconds (effective
519.7 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting...
```

Figure 14.12 Uploading the MicroPython firmware

Note that in Figure 14.12 it is assumed that the flash memory size is 4 MB and this is why parameter dio is specified. This parameter should be qio for 512 kB modules. The size of the flash memory can be found by the following command:

```
C:\> python c:\python27\lib\site-packages\esptool.py --port COM61 flash_id
```

The display in this example is shown in Figure 14.13.

```
C:\>python c:\python27\lib\site-packages\esptool.py --port COM47 flash_id
esptool.py v2.0.1
Connecting.....
Detecting chip type... ESP32
Chip is ESP32D0WDQ6 (revision 0)
Uploading stub...
Running stub...
Stub running...
Manufacturer: c8
Device: 4016
Detected flash size: 4MB
Hard resetting...
```

Figure 14.13 Displaying the flash memory size

- After the uploading is finished, press the RESET button on the ESP32 DevKitC.

14.2.1 Testing the MicroPython Installation

In order to test your MicroPython installation, start the Putty program. Set the connection type to Serial, enter your serial port number (COM61 here) into Serial line, and set the Speed to 115200. Press the RESET button on your ESP32 DevKitC. You should see the MicroPython interpreter screen as in Figure 14.14. Enter the following command to test your installation:

```
print ("Hello from MicroPython")
```

The screenshot shows a Windows-style terminal window titled "COM61 - Putty". The window contains the following text:

```
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:4732
load:0x40078000,len:7496
load:0x40080400,len:5512
entry 0x4008114c
I (398) cpu_start: Pro cpu up.
I (399) cpu_start: Single core mode
I (399) heap_init: Initializing. RAM available for dynamic allocation:
I (402) heap_init: At 3FFAE6E0 len 00001920 (6 Kib): DRAM
I (408) heap_init: At 3FFC4F68 len 0001B098 (108 Kib): DRAM
I (415) heap_init: At 3FFE0440 len 00003BC0 (14 Kib): D/IRAM
I (421) heap_init: At 3FFE4350 len 0001BC80 (111 Kib): D/IRAM
I (427) heap_init: At 40091604 len 0000E9FC (58 Kib): IRAM
I (534) cpu_start: Pro cpu start user code
I (4) cpu_start: Starting scheduler on PRO CPU.
OSErr: [Errno 2] ENOENT
MicroPython v1.9.4-683-gd94aa577a on 2018-11-12; ESP32 module with ESP32
Type "help()" for more information.
>>> print ("Hello from microPython")
Hello from microPython
>>> 
```

Figure 14.14 MicroPython interpreter screen

It is not the intention of this book to teach the MicroPython programming language since there are many books (e.g ESP8266 AND MICROPYTHON from Elektor), tutorials, projects, and application notes on this topic. In the next sections some projects of using MicroPython on the ESP32 DevKitC are given to show how MicroPython can be used in projects.

14.3 Flashing LED

This project is same as Project 1 in Chapter 4.2 where an LED is connected to GPIO port 23 through a 330 ohm current limiting resistor and the LED is flashed every second.

The program is written interactively using the Putty terminal emulator to connect to the ESP32 DevKitC. Figure 14.15 shows the program listing.

```
>>>
>>> from machine import Pin
>>> import time
>>> p23 = Pin(23, Pin.OUT)
>>> while True:
...     p23.value(0)
...     time.sleep(1)
...     p23.value(1)
...     time.sleep(1)
...
...
```

Figure 14.15 MicroPython program listing

14.4 LED With Push-Button Switch

In this project an LED is connected to GPIO port 23 as in the previous project. In addition, a push-button switch is connected to GPIO port 22. The LED is turned ON whenever the button is pressed.

Figure 14.16 shows the circuit diagram of the project. The project listing is shown in Figure 14.17. GPIO port 22 is configured as an input and it is pulled up so that the pin is at normally logic 1 and the LED is OFF. Pressing the button changes the button input state to logic 0 and turns ON the LED.

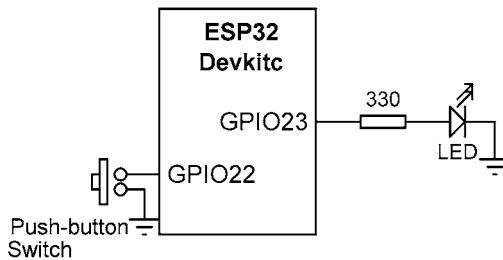


Figure 14.16 Circuit diagram of the project

```
>>>
>>> from machine import Pin
>>> p23 = Pin(23, Pin.OUT)
>>> p22 = Pin(22, Pin.IN, Pin.PULL_UP)
>>> p23.value(0)
>>> while True:
...     if p22.value() == 0:
...         p23.value(1)
...     else:
...         p23.value(0)
...
...
```

Figure 14.17 MicroPython program listing

14.5 Temperature and Humidity

In this project the temperature and humidity are measured using the DHT11 sensor chip and are displayed on the Putty screen every 5 seconds. The block diagram and circuit diagram of the project are as in Figure 5.10 and Figure 5.13 respectively.

Figure 14.18 shows the MicroPython program listing which is entered interactively through Putty. Notice libraries **dht**, **Pin**, and **time** are included at the beginning of the program.

```
>>> import dht
>>> import time
>>> from machine import Pin
>>> d = dht.DHT11(Pin(23))
>>> while True:
...     d.measure()
...     t = d.temperature()
...     h = d.humidity()
...     print("Temperature=%fC, Humidity=%f" % (t, h))
...     time.sleep(5)
...
...
Temperature=23.000000C, Humidity=60.999999
Temperature=23.000000C, Humidity=60.999999
Temperature=23.000000C, Humidity=60.999999
```

Figure 14.18 Program listing

14.6 Connecting to a Wi-Fi

Before communicating with other nodes on a network we first have to establish ourselves as a node on the network and have an IP address assigned to us. The MicroPython library called **network** provides network related functions such as connecting to a network, finding the IP address, and so on. This library must be imported into our MicroPython programs before we can use the network functions.

The **network** library has a class called **WLAN**. Some of the commonly used methods of this class are:

network.WLAN(network.STA_IF): creates a WLAN network object to connect to a Wi-Fi.

network.WLAN(network.AP_IF): creates a WLAN network object to connect to an AP, allowing other Wi-Fi clients to connect.

active(True): activates the network interface.

active(False): deactivates the network interface.

connect("ssid", "password"): connects to the specified Wi-Fi network.

disconnect(): disconnect from the currently connected network.

isconnected(): used to find out if we are connected to a Wi-Fi.

ifconfig(): returns the IP address assigned to our connection.

An example is shown in Figure 14.19 using interactive command mode which establishes connection to a Wi-Fi network by specifying the ssid (in this example BTHomeSpot-XNH) and password (in this example 49345abaeb), and then displays the assigned IP address which happens to be 192.168.1.156.

```

>>> import network
>>> net=network.WLAN(network.STA_IF)
>>> net.isconnected()
False
>>> net.active(True)
I (1524367) wifi: mode : sta (30:ae:a4:05:b5:b0)
I (1524367) wifi: STA_START
True
>>> net.connect("BTHomeSpot-XNH","49345abaeb")
>>> I (1561557) wifi: n:6 0, o:1 0, ap:255 255, sta:6 0, prof:1
I (1562117) wifi: state: init -> auth (b0)
I (1562127) wifi: state: auth -> assoc (0)
I (1562137) wifi: state: assoc -> run (10)
I (1562157) wifi: connected with BTHomeSpot-XNH, channel 6
I (1562157) wifi: event 4
I (1564817) event: ip: 192.168.1.156, mask: 255.255.255.0, gw: 192.168.1.254
I (1564817) wifi: GOT_IP
I (1572137) wifi: pm start, type:0

>>> net.ifconfig()
('192.168.1.156', '255.255.255.0', '192.168.1.254', '192.168.1.254')
>>> █

```

Figure 14.19 Connecting to a Wi-Fi network

The following MicroPython function can be called from a program to connect to a Wi-Fi network. The assigned IP address is displayed at the end of the function:

```

def Connect_WiFi():
    import network
    net = network.WLAN(network.STA_IF)
    if not net.isconnected():
        net.active(True)
        net.connect("ssid", "password")
    print (net.ifconfig())

```

14.7 MicroPython UDP Programs

As described in Chapter 7, UDP is a connectionless protocol and it is commonly used to communicate over a network by sending and receiving data packets.

An example program is given in this section where the temperature and humidity readings are sent to an Android mobile phone as described in Chapter 11, using UDP packets. The operation of the program is as follows:

- The mobile phone requests data by sending character S to the ESP32 DevKitC.
- DevKitC reads the temperature and humidity from chip DHT11 and sends to the mobile phone.
- The above process is repeated forever until stopped.

Figure 14.20 shows the program listing for this example, entered interactively using the Putty terminal emulator. At the beginning of the program the socket library, the DHT11 and Pin libraries are imported into the program. Function **Connect_WiFi** connects to the local Wi-Fi. Function **tempHum** reads the ambient temperature and humidity and returns in variables **T** and **H** respectively. The program then creates an UDP socket and binds to this socket. The remainder of the program is executes inside the **while** loop. Here, the program

waits to receive the request S from the mobile phone. After the request is received, function **tempHum** is called and the temperature and humidity readings are sent to the mobile phone as in Figure 11.3.

```
>>> import socket
I (95754) modsocket: Initializing
>>> import dht
>>> from machine import Pin
>>> def Connect_WiFi():
...     import network
...     net=network.WLAN(network.STA_IF)
...     if not net.isconnected():
...         net.active(True)
...     net.connect("BTHomeSpot-XNH","49345abaeb")
...
...
>>> def tempHum():
...     d=dht.DHT11(Pin(23))
...     d.measure()
...     t=d.temperature()
...     h=d.humidity()
...     return t,h
...
...
>>> Connect_WiFi()
>>> port=5000
>>> UDP=":", port
>>> sock=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
>>> sock.bind(UDP)
>>> while True:
...     data,addr=sock.recvfrom(80)
...     if(data.decode() == 'S'):
...         T,H=tempHum()
...         dat=str(T)+" "+str(H)
...         tup=(addr[0],port)
...         sock.sendto(dat,tup)
...
...
5
5
```

Figure 14.20 Program listing entered interactively

Figure 14.21 shows the program listing as a file with comments.

```
-----#
#           SEND TEMPERATURE AND HUMIDITY TO A MOBILE PHONE
#-----#
#
# In this program a DHT11 sensor chip is connected to GPIO port 23 of the
# ESP32 Devkitc. The MicroPython program reads the ambient temperature and
# humidity from the sensor chip and sends to the mobile phone when
# requested.
#
# A request is made by the mobile phone it sends character S. The data is
# displayed on the mobile phone in the following format:
#
# T=nn H=nn
#
# Author: Dogan Ibrahim
# Date : August 2017
#-----#
---#
#
# Import libraries used in the program
```

```
#  
import socket  
import dht  
from machine import Pin  
  
#  
# Function to connect to Wi-Fi  
#  
def Connect_WiFi():  
    import network  
    net = network.WLAN(network.STA_IF)  
    if not net.isconnected():  
        net.active(True)  
        net.connect("BTHomeSpot-XNH", "49345abaeb")  
  
#  
# Function returns the temperature and humidity  
#  
def tempHum():  
    d = dht.DHT11(Pin(23))  
    d.measure()  
    t = d.temperature()  
    h = d.humidity()  
    return t,h  
  
#  
# Start of main program. Define port number, create a socket, and bind to it  
#  
Connect_WiFi()  
port = 5000  
UDP = ("", port)  
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)  
sock.bind(UDP)  
  
#  
# This part of the program runs in an endless loop, sending the temperature  
# and humidity readings to the mobile phone when it receives a request (S)  
#  
while True:  
    data, addr = sock.recvfrom(80)  
    if(data.decode() == 'S'):  
        T,H = TempHum()  
        dat = str(T) + " " + str(H)  
        tup = (addr[0], port)  
        sock.sendto(dat, tup)
```

Figure 14.21 Program listing with comments

14.8 Storing Temperature and Humidity on the Cloud

In this project the ambient humidity and temperature are read and stored on the cloud. The block diagram and the circuit diagram of the project are as in Figure 8.1 and Figure 5.12 respectively.

The readings are stored on the **Thingspeak** cloud as in Chapter 8. You should create an account with the **Thingspeak** and have an API key and a channel number as described in Chapter 8. In this section only the MicroPython program is given, and the same API key and channel number as in Chapter 8 are used here.

Figure 14.22 shows the MicroPython program entered interactively using the Putty terminal emulator. The program basically makes an HTTP GET request to upload the temperature and the humidity values.

The program listing with comments is shown in Figure 14.23. At the beginning of the program the network libraries, time library, DHT11 library and the Pin library are imported to the program. Function **Connect_WiFi** connects to the local Wi-Fi. Function **TempHum** returns the temperature and the humidity. The main program loop start with the **while** statement. Inside this loop the IP address of the **Thingspeak** website is extracted and a connection is made to this site. Then the temperature and humidity readings are obtained and are included in the **path** statement. The **sock.send** statement sends an HTTP GET request to the **Thingspeak** site and uploads the temperature and humidity values. This process is repeated every 30 seconds.

As in Chapter 8, the link to view the data graphs in the cloud is:

<https://api.thingspeak.com/channels/265433>

```
>>> import network
>>> import socket
>>> import time
>>> import dht
>>> from machine import Pin
>>> APIKEY="QAAHTSXIK32B8736"
>>> def Connect_WiFi():
...     net=network.WLAN(network.STA_IF)
...     net.active(True)
...     net.connect("BTHomeSpot-XNH","49345abaeb")
...
>>> def TempHum():
...     d=dht.DHT11(Pin(23))
...     d.measure()
...     t=d.temperature()
...     h=d.humidity()
...     return t,h
...
>>> while True:
...     Connect_WiFi()
...     sock=socket.socket()
...     addr=socket.getaddrinfo("api.thingspeak.com", 80)[0][-1]
...     sock.connect(addr)
...     t,h=TempHum()
...     host="api.thingspeak.com"
...     path="api key="+APIKEY+"&field1="+str(t)+"&field2="+str(h)
...     sock.send(bytes("GET /update?"+path+" HTTP/1.0\r\nHost: "+host+"\r\n\r\n", "utf8"))
...     sock.close()
...     time.sleep(30)
...
95
95
95
```

Figure 14.22 MicroPython program entered interactively

```
#-----
#          TEMPERATURE AND HUMIDITY ON THE CLOUD
# =====
#
# In this project temperature and humidity sensor is connected to
# GPIO port 23 of the ESP32 Devkitc board. The project reads the
# temperature and humidity and sends to the cloud where it can be accessed
# from anywhere. In addition, change of the temperature and the humidity
# can be plotted in the cloud.
#
# The DHT11 sensor chip is used to get the ambient temperature and the
# humidity.
#
# The program uses the Thingspeak cloud service. The ESP32 Devkitc must be
# conencted to the Wi-Fi router before we can send data to the cloud.
#
#
# Date      : May, 2017
# Programmer: Dogan Ibrahim
#-----
----  
import network  
import socket  
import time  
import dht  
from machine import Pin  
  
APIKEY = "QAAHT5XIK3ZB8736"  
  
#  
# This function connects to the Wi-Fi  
#  
def Connect_WiFi():  
    net = network.WLAN(network.STA_IF)  
    if not net.isconnected():  
        net.active(True)  
        net.connect('iBTHomeSpot-XNH', '49345abaeb')  
        print(net.ifconfig())  
  
    #  
# This function returns the temperature and humidity  
#  
def TempHum():  
    d = dht.DHT11(Pin(0))  
    d.measure()
```

```
t = d.temperature()
h = d.humidity()
return t, h

#
# Send data to Thingspeak. This function sends the temperature and
# humidity data to the cloud every 30 seconds
#
while True:
    Connect_WiFi()
    sock = socket.socket()
    addr = socket.getaddrinfo("api.thingspeak.com", 80)[0][-1]
    sock.connect(addr)
    (t, h) = TempHun()
    host = "api.thingspeak.com"
    path = "api_key="+APIKEY+"&field1="+str(t)+"&field2="+str(h)
    sock.send(bytes("GET /update?"+path+" HTTP/1.0\r\nHost: "+host+"\r\n\r\n", "utf8"))
    sock.close()
    time.sleep(30)
```

Figure 14.23 Program listing with comments

14.9 Remote Control Using Mobile Phone (Web Server)

This project is same as the project in Chapter 10 where two LEDs are connected to an ESP32 DevKitC development board and are controlled from a mobile phone. The idea here is that the DevKitC will be configured as an UDP client. Then, UDP packets will be sent from a mobile phone to control these LEDs. As mentioned earlier, these LEDs can be replaced by relays and the project can be used to control any kind of electrical equipment connected to the ESP32 DevKitC.

Valid commands are as follows (any other commands are ignored by the program):

0=ON	Turn ON LED0
0=OFF	Turn OFF LED0
1=ON	Turn ON LED1
1=OFF	Turn OFF LED1

The block diagram and the circuit diagram of this project are as in Figure 10.1 and Figure 9.2 respectively where the LEDs are connected to GPIO ports 23 (LED0) and 22 (LED1) through 330 ohm current limiting resistors. The LEDs are turned ON when the port output is at logic 1 (high), and turned OFF when the port output is at logic 0 (low).

In this project an Android based mobile phone loaded with the UDP apps called **UDP RE-CEIVE** and **SEND** by Wezzi Studios (version 4.0) is used to send commands to the ESP32 DevKitC. Clicking the SEND UDP MESSAGE button will send the packet to its destination.

In the Receiver part the local IP address is given. The user must specify the required port number. Received packets will then be displayed on the screen (see Chapter 10 for more details).

Figure 14.24 shows the complete MicroPython program.

```
#-----
#           WEB SERVER TO CONTROL 2 LEDs
# =====
#
# This is a simple web server program. 2 LEDs are connected to GPIO ports 23
# 22 of the ESP32 Devkitc development board. The project controls these LEDs
# (turns ON or OFF) from a web server application. For example, the LEDs can be
# controlled from any device that is on the web, for example a PC, tablet,
# mobile phone etc. When activated, a form will appear on the device with
# buttons and clicking these button will control the LEDs. The LEDs are named as
# LED0 and LED1, connected to ports 23 and 22 respectively.
#
#
# Date      : May, 2017
# Programmer: Dogan Ibrahim
#-----
----
import network
import socket
from machine import Pin
#
# Configure the pins as outputs
#
LED0 = Pin(23, Pin.OUT)
LED1 = Pin(22, Pin.OUT)
#
# Turn OFF the LEDs to start with
#
LED0.low()
LED1.low()

#
# The HTML code
#
html = """<!DOCTYPE html>
<html>
<body>
<center><h1>ESP32 DECKITC LED ON/OFF</h1></center>
<center><h2>MicroPython Web Server Example with 2 LEDs</h2></center>
<form>
```

```
<button name="LED0" button style="color:green" value="ON" type="submit">LED0  
ON</button>  
<button name="LED0" button style="color:red" value="OFF" type="submit">LED0  
OFF</button><br><br>  
  
<button name="LED1" button style="color:green" value="ON" type="submit">LED1  
ON</button>  
<button name="LED1" button style="color:red" value="OFF" type="submit">LED1  
OFF</button>  
</form>  
</body>  
</html>  
****  
  
#  
# This function connects to the Wi-Fi  
#  
#  
def Connect_WiFi():  
    net = network.WLAN(network.STA_IF)  
    if not net.isconnected():  
        net.active(True)  
        net.connect('iBTHomeSpot-XNH1', 'i49345abaeb1')  
  
#  
#  
#  
Connect_WiFi()  
addr=socket.getaddrinfo("0.0.0.0",80)[0][-1]  
s = socket.socket()  
s.bind(addr)  
s.listen(1)  
while True:  
    conn, address = s.accept()  
    request = conn.recv(1024)  
    request = str(request)  
    LEDON0 = request.find("/?LED0=ON")  
    LEDOFF0 = request.find("/?LED0=OFF")  
    LEDON1 = request.find("/?LED1=ON")  
    LEDOFF1 = request.find("/?LED1=OFF")  
    if LEDON0 != -1:  
        LED0.high()  
    if LEDOFF0 != -1:  
        LED0.low()  
    if LEDON1 != -1:
```

```

        LED1.high()
if LEDOFF1 != -1:
    LED1.low()
resp = html
conn.send(resp)
conn.close()

```

Figure 14.24 Program listing with comments

At the beginning of the program, the network libraries and the Pin library are included in the program, LED0 and LED1 are assigned to GPIO ports 23 and 22 respectively, these ports are configured as outputs, and the LEDs are turned OFF to start with. The program then defines the HTML code that is to be sent to the web client when a request comes from it. This code is stored in variable html.

The web server program sends the html file (statements resp=html and conn.send(resp)) to the client and waits for a command from the client. When a button is clicked on the client, a command is sent to the web server in addition to some other data. Here, we are only interested in the actual command sent. The following commands are sent to the web server for each button pressed:

Button Pressed	Command sent to web server
LED0 ON	/?LED0=ON
LED0 OFF	/?LED0=OFF
LED2 ON	/?LED2=ON
LED2 OFF	/?LED2=OFF

The web server program on the ESP32 DevKitC makes a TCP connection and reads the command from the web client (request = conn.recv(1024)). The program then searches the received command to see if any of the above commands are present in the received data. Remember that find searches a string for a sub-string and returns the starting position of the sub-string in the string. If the sub-string is not found then a -1 is returned. For example, consider the following statement:

```
LEDON0 = request.find("/?LED0=ON")
```

Here, sub-string /?LED0=ON is searched in string request and if found, the starting position of the sub-string in the string is given. A -1 is returned if the sub-string /?LED0=ON does not exist in string request. The program looks for a match for all the 4 commands and if a command is found then the corresponding LED is turned ON or OFF accordingly:

```

LEDON0 = request.find("/?LED0=ON")
LEDOFF0 = request.find("/?LED0=OFF")

```

```
LEDON2 = request.find("/?LED2=ON")
LEDOFF2 = request.find("/?LED2=OFF")
if LEDON0 != -1:           # If ON0 command found
    LED0.high()            # Turn ON LED0
if LEDOFF0 != -1:          # If OFF0 command found
    LED0.low()              # Turn OFF LED0
if LEDON2 != -1:           # If ON2 command found
    LED2.high()             # Turn ON LED2
if LEDOFF2 != -1:          # If OFF2 command found
    LED2.low()              # Turn OFF LED2
```

In this project, the IP address of the web server was 192.168.1.156. The procedure to test the system is as follows:

- Run the MicroPython program either interactively or from a file.
- Open your web browser on your PC and enter the website address 192.168.1.156.
- The control HTML form will be displayed on your PC screen.
- Click a button (e.g. LED0 ON) to turn ON (or OFF) an LED (e.g. LED0).

14.10 Loading MicroPython Programs to the ESP32 DevKitC

Up to now we have been running our MicroPython programs interactively where the program is entered via the terminal emulator Putty. There are many applications for which we may want to upload our program to the ESP32 DevKitC and run automatically at boot time or after a RESET. In this section we will see how this can be done. An example project is given here where a small servo motor is used and the motor is rotated 180 degrees in each direction with 3 seconds delay between each rotation. The aim here is to show that the motor will be controlled automatically after the ESP32 DevKitC is restarted.

14.10.1 Using the ampy

ampy (Adafruit MicroPython Tool) is a software utility that can be used to serially interact with an ESP32 board loaded with the MicroPython. Ampi is a simple command line based tool that can be used to perform the following operations on our MicroPython board:

- List contents of a directory on the MicroPython board (command: ls).
- Create a folder on the MicroPython board (command: mkdir).
- Retrieve a file from the MicroPython board (command: get).
- Send a file from the PC to the MicroPython board (command: put).
- Remove a folder and its contents from the MicroPython board.
- Run a script on the PC and print its output (command: run).
- Perform reset of the board (command: reset).

ampy should be installed on the PC before it can be used. This requires Python 2.7.x or 3.x to be already installed on your PC. In this section we are assuming that Python 2.7 has

already been installed on your PC.

To install ampy, enter the following command from the command line on your PC (see Figure 14.25):

```
C:\> cd python27\scripts  
C:\> Python27\Scripts> pip install adafruit-ampy
```

```
C:\>cd Python27\scripts  
C:\Python27\Scripts>pip install adafruit-ampy  
Collecting adafruit-ampy  
  Downloading adafruit_ampy-1.0.1-py2.py3-none-any.whl  
Collecting click (from adafruit-ampy)  
  Downloading click-6.7-py2.py3-none-any.whl (71kB)  
    100% :#####: 71kB 484kB/s  
Requirement already satisfied: pyserial in c:\python27\lib\site-packages (from adafruit-ampy)  
Installing collected packages: click, adafruit-ampy  
Successfully installed adafruit-ampy-1.0.1 click-6.7  
C:\Python27\Scripts>
```

Figure 14.25 Installing ampy

You can check whether or not ampy has been installed successfully by entering the following help command (see Figure 14.26):

```
C:\Python27\Scripts> ampy --help
```

```
C:\Python27\Scripts>ampy --help  
Usage: ampy [OPTIONS] COMMAND [ARGS]...  
  
      ampy - Adafruit MicroPython Tool  
  
      ampy is a tool to control MicroPython boards over a serial connection.  
      Using ampy you can manipulate files on the board's internal filesystem  
      and even run scripts.  
  
Options:  
  -p, --port PORT  Name of serial port for connected board. Can optionally  
                   specify with AMPY_PORT environment variable. [required]  
  -b, --baud BAUD  Baud rate for the serial connection (default 115200). Can  
                   optionally specify with AMPY_BAUD environment variable.  
  --version         Show the version and exit.  
  --help            Show this message and exit.  
  
Commands:  
  get   Retrieve a file from the board.  
  ls    List contents of a directory on the board.  
  mkdir Create a directory on the board.  
  put   Put a file or folder and its contents on the...  
  reset Perform soft reset/reboot of the board.  
  rm   Remove a file from the board.
```

Figure 14.26 ampy help command

An example of using ampy is given below. Before using these commands you should connect your ESP32 DevKitC development board to the PC and find out the serial port number assigned to your board (COM47 in the examples below).

An example is given in Figure 14.27 which lists the files on the MicroPython board:

```
C:\Python27\Scripts> ampy --port COM47 ls
```

```
C:\Python27\Scripts>ampy --port COM2 ls  
boot.py  
C:\Python27\Scripts>
```

Figure 14.27 Listing files on the board

14.10.2 Creating and Running a Program

Using ampy we can run a MicroPython program on our MicroPython board. The program can be created using our favourite text editor such as the Notepad (or even ESPlorer). An example program created using the Notepad is shown in Figure 14.28. The program is named **sinetable.py** and is saved in sub-folder **C:\Python\Scripts** folder. This program displays the trigonometric sines of angles from 0 degrees to 90 degrees in steps of 5 degrees.

```
#-----  
#  
#      TRIGONOMETRIC SINE TABLE  
#  ======  
#  
# This program tabulates the trigonometric sine of angles from 0  
# to 90 degrees in steps of 5 degrees  
#  
#  
# Program: sinetables.py  
# Date   : August, 2017  
# Author : Dogan Ibrahim  
#-----  
import math  
  
print ("TABLE OF TRIGONOMETRIC SINE")  
print ("=====  
print ("")  
print ("    ANGLE      SINE")  
  
for angle in range (0, 95, 5):  
    r = math.radians(angle)  
    s = math.sin(r)  
    print (" %d      %f" %(angle, s))  
  
print("End of program")
```

Figure 14.28 Program sinetable.py

We can now run our program by entering the following command:

```
C:\Python27\Scripts> ampy --port COM47 run sinetable.py
```

The output of the program is shown in Figure 14.29. Notice that any output from the program will be displayed on the PC screen. By default, the run command waits for the program to finish running on the board before displaying its output. In many microcontroller based applications the program is executed in an endless loop and never finishes. In such applications you must use the **--no-output** option in the run command. Choosing this option tells ampy not to wait for any output and to just start running the program and return. It is also important to realise that any keyboard input cannot be sent to the program.

The **-no-output** option is used as follows:

```
C:\Python27\Scripts> ampy --port COM2 run --no-output sinetable.py
C:\Python27\Scripts> ampy --port COM2 run sinetable.py
=====
TABLE OF TRIGONOMETRIC SINE
=====
    ANGLE      SINE
    0      0.000000
    5      0.087156
   10      0.173648
   15      0.258819
   20      0.342020
   25      0.422618
   30      0.500000
   35      0.573576
   40      0.642787
   45      0.707107
   50      0.766044
   55      0.819152
   60      0.866025
   65      0.906308
   70      0.939692
   75      0.965926
   80      0.984808
   85      0.996195
   90      1.000000
```

Figure 14.29 Output of the program

14.10.3 Running a Program at Boot Time

In some applications we may want to run our program automatically when the ESP32 DevKitC board is powered up. This can be done easily as described below.

When the MicroPython installed board is powered up it automatically executes the firmware file called **boot.py**. This file should not normally be modified unless if you want to customise the boot process. You can use the ampy **get** command to see the contents of the **boot.py** file. When **boot.py** completes it calls to MicroPython program named **main.py** which may or may not be resident on the board. If **main.py** is not resident, we should rename our program to **main.py** and upload it to our board. This way, our program will run automatically as soon as the board boots up. The steps are given below.

- Create the required program (e.g. **sinetable.py** as in Figure 14.27).
- Rename the program to **main.py**.
- Connect the board to your PC.
- Upload the file to the board using the following ampy command:

```
C:\Python27\Scripts> ampy --port COM47 put main.py
```

- Make sure that file main.py has been uploaded. Enter the following ampy command and you should see files boot.py and main.py listed:

```
C:\Python27\Scripts> ampy --port COM47 ls
```

- Start the terminal emulation program, e.g. PUTTY.
- Restart the board and you should see the output displayed on the terminal emulator window.

An example is given here where the small low-cost servo motor Tower Pro SG90 (see Figure 14.29) is controlled such that it rotates fully left and right with 3 seconds between each turn.



Figure 14.30 Tower Pro SG90 servo

The servo has the following pins:

- Black: GND
- Red: +3.3 V
- Brown: Signal

The basic features of this servo are:

- Dimension: 22 mm x 12 mm x 31 mm
- Operating speed: 0.3 s/60 degree
- Weight: 9 g
- Operating voltage: 4.2 – 6 V

Even though the minimum operating voltage of the servo is specified as 4.2 V, it will work at +3.3 V. In this project the signal pin of the servo is connected to GPIO pin 23 and the other pins to the ground and +3.3 V.

The servo is operated by applying a 50 Hz (period = 20 ms) PWM waveform to it. When the pulse width is approximately 2 ms the servo rotates fully in one direction. Similarly, when the pulse width is 1 ms the servo rotates fully in the opposite direction. We can find the required PWM duty cycles for 2 ms and 1 ms pulses as follows:

$$\text{Duty cycle} = (\text{Pulse width} / \text{Period}) \times 100 \%$$

or,

$$\text{Duty cycle} = (\text{Pulse width} * \text{frequency}) \times 100 \%$$

For 2 ms pulse:

$$\text{Duty cycle} = (0.002 * 50) \times 100 \% = 10\%$$

For the 1 ms pulse

$$\text{Duty cycle} = (0.001 * 50) \times 100 \% = 5\%$$

Because the PWM resolution is 10-bits, the full-scale (100%) duty cycle is obtained when 1024 is loaded into the duty cycle. Therefore, 10% and 5% duty cycles correspond to 102 and 51 respectively. In practice, however, it was found that the servo was fully in one direction when the duty cycle was 126%, and then fully in the other direction when the duty cycle was set to 15%.

Figure 14.31 shows the program listing. GPIO pin 23 is configured as a PWM channel, the frequency is set to 50 Hz, and the duty cycle is varied inside a while loop so that the servo rotates in both directions. 3 seconds of delay is inserted between each rotation.

```
#-----
#                               SERVO CONTROL
#-----#
# In this program a small servo motor is conencted to GPIO
# port 23 of the ESP32 Devkitc. Teh program sends PWM pulses
# to the servo to rotate it fully in one direction, and after
# 3 seconds in the opposite direction. This process is repeated
# until stopped manually.
#
#
# File  : main.py
# Author: Dogan Ibrahim
# Date  : August, 2017
#-----#
```

```
import machine
import time
p23 = machine.Pin(23)
pwm23 = machine.PWM(p23)
pwm23.freq(50)      # set frequency to 50 Hz
while True:          # Do Forever
    pwm23.duty(20)   # Turn in one direction
    time.sleep(3)     # Wait 3 seconds
    pwm23.duty(128)  # Turn in opposite direction
    time.sleep(3)     # Wait 3 seconds
```

Figure 14.31 MicroPython Program listing

Create the program using the Notepad and save it with the name **main.py** in folder **C:\Python27\Scripts**. Then copy the program to your ESP32 DevKitC with the following command. Restart your ESP32 DevKitC and you should see the servo rotating as expected.

```
C:\Python27\Scripts> ampy --port COM47 put main.py
```

You can remove the program from the root directory of your ESP32 DevKitC with the following command to stop it running at boot time:

```
C:\Python27\Scripts> ampy --port COM47 rm main.py
```

14.11 Using the Real Time Clock (RTC)

The RTC is useful in applications when we want to know the absolute real time. For example, we may want to read the ambient temperature and then save the data frequently with date and time stamping. Although the ESP32 processor has a built-in RTC it has been reported by many users that it is not accurate for precision applications. Additionally, the RTC data is lost after power is removed from the chip and it is required to have external battery support. The date and time data are also lost when the processor is hard reset. There are accurate RTC modules with built-in batteries such as the DS3231 or similar modules and the author recommends the readers to use such modules if accurate date and time stamping are required in their applications.

In this section we shall be looking at how to use the built-in RTC to load the current date and time and then how to read this data at a later time.

The RTC can be accessed by the following functions in MicroPython command mode:

```
>>> import machine
>>> rtc = machine.RTC()
>>> print(rtc.datetime())
(2000, 1, 1, 5, 0, 17, 769304)
```

The date and time are displayed as an 8-tuple with the following format:

```
(year, month, day, weekday, hours, minutes, seconds, subseconds)
```

Weekday takes values 1 to 7 with Monday being weekday 1.

The date and time can be set using the `datetime` function. For example the following command sets the date to 12 November 2018, and the time to 17:55:30, and the Weekday to 1:

```
>>> rtc.datetime((2018, 11, 12, 1, 17, 55, 30, 0))
```

We can extract the fields of date and time. In the following example the date fields are extracted:

```
>>> import time
>>> now = time.localtime()
>>> year = now[0]
>>> month = now[1]
>>> day = now[2]
```

14.11.1 Timestamping Temperature and Humidity Readings

In this example the ambient temperature and humidity data are read every 10 seconds as described in Section 14.5, where a DHT11 type sensor is connected to GPIO port 23 of the ESP32 DevKitC. These readings are then displayed on the screen with time stamping. Figure 14.32 shows the MicroPython program listing and output from the program. At the beginning of the program the libraries used in the program are imported to the program and the RTC time is set to 12-11-2018 18:13:16. The remainder of the program runs in a loop which is repeated every 10 seconds. Inside this loop the current date and time are read, and the temperature and humidity readings are read and displayed on the screen. Figure 14.33 shows the output of the program.

```
>>> import time
>>> import machine
>>> import dht
>>> from machine import Pin
>>> rtc = machine.RTC()
>>> rtc.datetime((2018, 11, 12, 1, 18, 11, 0, 0))
>>> while True:
...     now = time.localtime()
...     y=now[0]; m=now[1]; d=now[2]; h=now[3]; mn=now[4]; s=now[5]
...     dh = dht.DHT11(Pin(23))
...     dh.measure()
...     T = dh.temperature()
...     H = dh.humidity()
...     print("%d-%d-%d %d:%d:%d T=%fC H=%f" %(d,m,y,h,mn,s,T,H))
...     time.sleep(10)
...
...
...
12-11-2018 18:13:16 T=33.000000C H=20.000000
12-11-2018 18:13:27 T=33.000000C H=15.000000
12-11-2018 18:13:37 T=33.000000C H=15.000000
12-11-2018 18:13:47 T=33.000000C H=16.000000
```

Figure 14.32 Program listing

14.11.2 Summary

In this chapter we have seen how to install the MicroPython interpreter on an ESP32 DevKitC. In addition, several examples and projects are also given to show how a MicroPython program can be developed and run on the ESP32 DevKitC.

Chapter 15 • Using ESP-IDF for programming the ESP32 DevKitC

15.1 Overview

Up to now we have been developing programs for our ESP32 DevKitC using the Arduino IDE and MicroPython. ESP-IDF is the native development environment from Espressif Systems, developed for the ESP32 processors. This development environment is based on FreeRTOS and enables users to access all modules of the ESP32 processor. Although the ESP-IDF is very powerful, it is more difficult to learn to program than Arduino and MicroPython.

In this chapter we shall be looking at how to install the ESP-IDF development environment to a Windows PC. Some examples will be given to show how an ESP-IDF based program can be developed and uploaded to the ESP32 DevKitC.

15.2 Installing the ESP-IDF

The following is needed in order to develop an application using the ESP-IDF (see website: <https://docs.espressif.com/projects/esp-idf/en/latest/get-started/index.html>):

- A Windows, Linux, or Mac operating system (in this chapter we shall be using Windows).
- A Toolchain to build applications for ESP32.
- An API to operate the Toolchain.
- A text editor to write programs in C (Notepad, Eclipse etc).
- The ESP32 processor board (ESP32 DevKitC is used in this chapter) and a USB cable.

The installation consists of the following steps:

- Setting up the Toolchain.
- Getting the ESP-IDF from GitHub.
- Installing a text editor (optional).

The application development process consists of writing the code, compiling and linking the code, uploading the code to the ESP32 processor, and finally testing/debugging the code.

15.2.1 Setting Up the Toolchain

The steps to setup the Toolchain are as follows:

- Download the Toolchain from the following website:

https://dl.espressif.com/dl/esp32_win32_msys2_environment_and_toolchain-20181001.zip

- Unzip the file to **C:** (recommended, but some other location could also be used) and wait until the unzip process finishes. A directory called **msys32** will be created under **C:** with many files in it (see Figure 15.1).

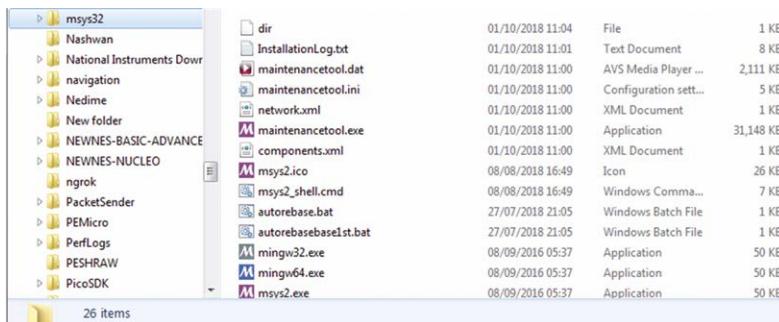


Figure 15.1 Directory msys32

- Run **C:\msys32\mingw32.exe** to open a MSYS2 MINGW32 terminal window. Then create a directory named esp which is the default location to develop ESP32 applications. This can be done as follows (see Figure 15.2):

```
mkdir esp
```

- You can move to the newly create directory by typing **cd esp**.

```

~/esp
Copying skeleton files.
These files are for the users to personalise their msys2 experience.

They will never be overwritten nor automatically updated.

'./.bashrc' -> '/home/Dogan/.bashrc'
'./.bash_logout' -> '/home/Dogan/.bash_logout'
'./.bash_profile' -> '/home/Dogan/.bash_profile'
'./.inputrc' -> '/home/Dogan/.inputrc'
'./profile' -> '/home/Dogan/.profile'
'./vimrc' -> '/home/Dogan/.vimrc'

Dogan@Dogan-HP MINGW32 ~
# mkdir -p ~/esp

Dogan@Dogan-HP MINGW32 ~
# cd ~/esp

Dogan@Dogan-HP MINGW32 ~/esp
# |

```

Figure 15.2 Open a MSYS2 MINGW32 terminal window

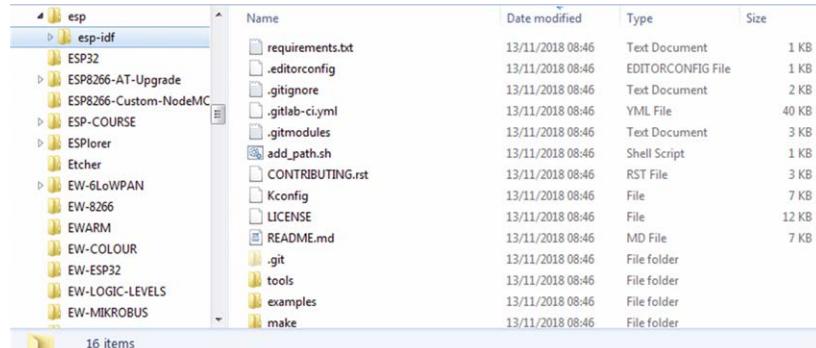
15.2.2 Getting ESP-IDF

The Toolchain includes the programs to compile and build your applications. In addition to these, we need ESP32 specific API and libraries which are provided by the ESP-IDF repository. The steps to get the ESP-IDF are given below (see website: <https://docs.espressif.com/projects/esp-idf/en/latest/get-started/index.html#get-started-get-esp-idf>). Here, we will be installing ESP-IDF into directory esp/esp-idf.

- Go to directory **esp** by entering command **cd esp**.

- **git clone --recursive https://github.com/espressif/esp-idF.git**
- ESP-IDF will be downloaded to directory **esp/esp-idF** (see Figure 15.3).
- Navigate to **msys32** directory and enter the following command:

```
cd c:\msys32
c:/esp/esp-idF/tools/windows/windows_install_prerequisites.sh
```



	Name	Date modified	Type	Size
esp	requirements.txt	13/11/2018 08:46	Text Document	1 KB
esp-idF	.editorconfig	13/11/2018 08:46	EDITORCONFIG File	1 KB
ESP32	.gitignore	13/11/2018 08:46	Text Document	2 KB
ESP8266-AT-Upgrade	.gitlab-ci.yml	13/11/2018 08:46	YML File	40 KB
ESP8266-Custom-NodeMC	.gitmodules	13/11/2018 08:46	Text Document	3 KB
ESP-COURSE	add_path.sh	13/11/2018 08:46	Shell Script	1 KB
ESPplorer	CONTRIBUTING.rst	13/11/2018 08:46	RST File	3 KB
Etcher	Kconfig	13/11/2018 08:46	File	7 KB
EW-6LoWPAN	LICENSE	13/11/2018 08:46	File	12 KB
EW-B26	README.md	13/11/2018 08:46	MD File	7 KB
EWARM	.git	13/11/2018 08:46	File folder	
EW-COLOUR	tools	13/11/2018 08:46	File folder	
EW-ESP32	examples	13/11/2018 08:46	File folder	
EW-LOGIC-LEVELS	make	13/11/2018 08:46	File folder	
EW-MIKROBUS				

Figure 15.3 Folder esp/esp-idF

15.2.3 Path to ESP-IDF

The Toolchain programs use IDF_PATH environment variable to access ESP-IDF. You can set this path on your PC user profile permanently so that it is available each time PC is re-started. Alternatively, you can specify the path before building programs.

Entering the Path Manually

If you do not like to have **IDF_PATH** set up permanently in your user profile, you should enter it manually each time the PC is re-started after opening an MSYS2 window:

```
export IDF_PATH="c:/esp/esp-idF"
```

Adding the Path to User Profile

See the following link to add a permanent path to your user profile:

https://docs.espressif.com/projects/esp-idF/en/latest/get-started/add-idf_path-to-profile.html

15.2.4 Required Python Packages

Some Python packages are required by the ESP-IDF. These can be installed by running the following command (this command assumes you have already installed Python 2.7 on your PC, see Chapter 14 on how to install Python 2.7):

```
cd esp
python2.7 -m pip install --user -r $IDF_PATH/requirements.txt
```

We should now download a project template. Download and extract it from the following link. This will create an example template in the **blink** directory under **esp**. This template can be used in our programs, as we shall see later.**cd esp**

```
git clone https://github.com/espressif/esp-idf-template.git blink
```

We are now ready to develop our ESP-IDF based projects. Some example projects will be given to show the steps to write and upload a program to our ESP32 DevKitC. Some useful examples can be found at the following links:

<https://github.com/espressif/esp-idf/tree/master/examples>

<http://docs.platformio.org/en/latest/platforms/espressif32.html#examples>

There are also many examples in the **examples** directory under **esp/esp-idf**.

15.3 PROJECT 1 – Flashing an LED

15.3.1 Description

This is perhaps the easiest project we can develop using the ESP-IDF. In this project an LED is connected to GPIO port 23 of the ESP32 DevkitC and the LED is flashed every three seconds.

15.3.2 Aim

The aim of this project is to show how a program can be developed and uploaded to the ESP32 DevKitC using the ESP-IDF programming environment.

15.3.3 Programming and Uploading Steps

There are already demo programs in directory **c:\esp\esp-idf\examples** which can be copied if desired. In this project we have created a new file called **main.c** and our new program file is shown in Figure 15.4. This file can be created using, for example, the **Note-pad** editor on your PC/ (You can also install and use **Eclipse**. See the following link on how to install and configure Eclipse for ESP32: <http://icircuit.net/esp-idf-setting-eclipse-esp-32/1867>.) The author created this file with the name **main.c** and saved in directory **c:**. The file is then copied to directory **esp\blink\main** using the following command (make sure that you are in directory **esp\blink**):

```
cp -r c:/main.c main/main.c
```

```
/*-----  
FLASHING LED  
=====
```

In this program an LED is connected to pin 23 of the ESP32 DevKitC and the LED is flashed every 2 seconds

```

Author: Dogan Ibrahim
Date : November 2018
File : main.c
-----*/
#include <stdio
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
#include "sdkconfig.h"

#define LED 23

void LED_Flash(void *pvParameter)
{
    gpio_pad_select_gpio(LED);
    gpio_set_direction(LED, GPIO_MODE_OUTPUT);
    while(1)
    {
        gpio_set_level(LED, 0);
        vTaskDelay(2000 / portTICK_PERIOD_MS);
        gpio_set_level(LED, 1);
        vTaskDelay(2000 / portTICK_PERIOD_MS);
    }
}

void app_main()
{
    xTaskCreate(&LED_Flash, "LED_Flash", configMINIMAL_STACK_SIZE,NULL,5,NULL);
}

```

Figure 15.4 Program listing

Notice that you can edit the file using the **vim** editor if required. Enter **vim main.c**, then type "**i**" (without the quotation) and do the required changes. Then press **ESC**, followed by **:wq** to save and exit from the editor.

Go to directory **/esp/blink**, and connect your ESP32 DevKitC to your PC and start the project configuration utility **menuconfig**. Here, we will have to define the communication details between our ESP32 DevKitC and the PC:

```
make menuconfig
```

You should see a window as in Figure 15.5. Select option **Serial flasher config**.

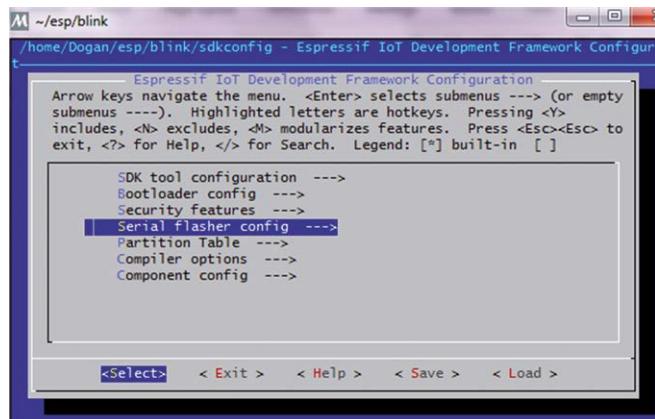


Figure 15.5 Select Serial flasher config

Set the Default serial port to **COM47** as shown in Figure 15.6 (notice that your COM port number may be different. Check the correct number in the Hardware Manager). Click **OK**. Select **Save** at the bottom, and then select **EXIT** to close the window.

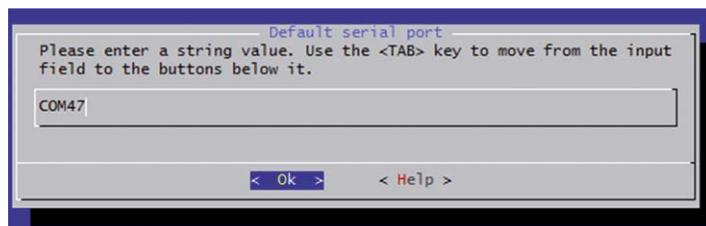


Figure 15.6 Set Default port to COM47

We are now ready to compile and upload our program to the ESP32 DevKitC. Make sure that you are in directory **/esp/blink** and that the ESP32 DevKitC is connected to your PC. Enter the following command:

```
make flash
```

The first time you use these commands it will take a while for all the necessary files to be compiled. Wait until you see the success message as in Figure 15.7. Your program should now start working and the LED will flash every 3 seconds.

```

M ~/esp/blink
Flashing binaries to serial port COM47 (app at offset 0x10000)...
esptool.py v2.6-beta1
Serial port COM47
Connecting.....
Chip is ESP32D0WDQ6 (revision 0)
Features: WiFi, BT, Dual Core, Coding Scheme None
MAC: 30:a8:a4:05:5b:e0
Uploading stub...
Running stub...
Stub running...
Configuring flash size...
Auto-detected Flash size: 4MB
Flash params set to 0x0220
Wrote 32768 bytes at 0x00001000 in 3.1 seconds (85.3 kbit/s)...
Hash of data verified.
Wrote 163840 bytes at 0x00010000 in 15.2 seconds (86.2 kbit/s)...
Hash of data verified.
Wrote 16384 bytes at 0x00008000 in 1.5 seconds (86.6 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...

```

Dogan@Dogan-HP MINGW32 ~/esp/blink

Figure 15.7 Successful compile and upload

We can also use the following commands instead of the **make flash** command:

```

make app    - build just the application
make app-flash - flash just the application (will be re-built if necessary)

```

We can also start the IDF Monitor by issuing the command **make monitor**. The IDF Monitor tool is a serial terminal Python program which can be used for debugging purposes to display and read data from the serial port. More information on IDF Monitor can be obtained from the following site:

<https://docs.espressif.com/projects/esp-idf/en/latest/get-started/idf-monitor.html>

15.3.4 Description of the Program

ESP-IDF programs make use of the FreeRTOS environment which enables complex multi-tasking programs to be developed. At the beginning of the program the header files required by the program are included in the program. The program starts to run from **app_main**. Here, function **xTaskCreate** creates a task. The first argument is the address of the task to execute when the task is scheduled to run, the second argument is the name given to the task, the third parameter is the memory to be allocated to the task. In this example minimum stack size is allocated to the task. Fifth parameter is the task priority, the other parameters can be set to NULL.

LED is assigned to GPIO port 23. The task that is to run is the function **LED_Flash**. **pvParameter** is the pointer that will be used as the parameter for the task being created. LED is configured as an output and then it is set to logic HIGH and LOW in a loop using statements **gpio_set_level**. Function **vTaskDelay** delays a task for a given number of ticks. Using constant **portTICK_PERIOD_MS** enables the delay to be set in milliseconds. Here, the delay is set to 3000 ms (three seconds).

15.3.5 Project Files

A project consists of the following files (e.g. in directory `/esp/blink`):

makefile: This file sets the **PROJECT_NAME** variable. It includes the `$(IDF_PATH)/make/project.mk` makefile

sdkconfig: This is the project configuration file created/updated when **make menuconfig** runs.

main: This is a directory that contains the source code for the project. The default name of the source code is **main.c**

build: This is a directory where the build output is created after the **make** command runs.

CMakeLists.txt: This file includes the name of the main source file (e.g. **main.c**)

15.4 PROJECT 2 – Three LEDs Flashing at Different Rates

15.4.1 Description

In this project three LEDs are connected to GPIO pins 23, 5 and 17 through current limiting resistors. These LEDs are named as NORMAL_LED, FAST_LED, and VERYFAST_LED respectively. This is a multi-tasking project where the LEDs are flashed at the following rates:

NORMAL_LED:	Every two seconds
FAST_LED:	Every second
VERYFAST_LED:	Every 0.5 seconds

15.4.2 Aim

The aim of this project is to show how a simple multi-tasking project can be developed using just three LEDs.

15.4.3 Circuit Diagram

Figure 15.8 shows the circuit diagram of the project.

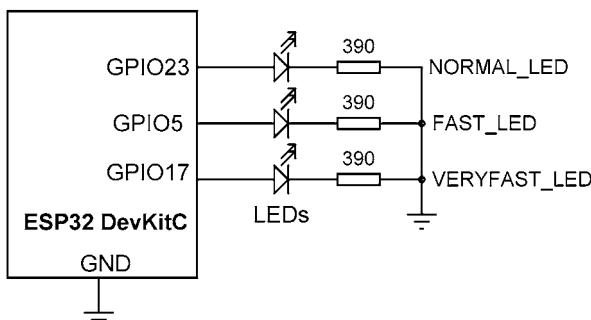


Figure 15.8 Circuit diagram of the project

15.4.4 Programming and Uploading Steps

In this project we will be using the same template file (**blink**) as in the previous project. The program file is named as **main.c** and is shown in Figure 15.9. As in the previous project, the file was created using **Notepad** and was then copied to directory **/esp/blink/main**

```
/*
-----+
      MULTI-TASKING WITH 3 LEDS
-----+
-----+
In this program 3 LEDs are connected to pins 23, 5 and 17 of the ESP32
DevKitC and these LEDs are named as NORMAL_LED, FAST_LED, and VERYFAST_LED
respectively. The LEDs flash at the following rates: NORMAL_LED every 2
seconds, FAST_LED every second, and VERYFAST_LED every 0.5 seconds.

Author: Dogan Ibrahim
Date : November 2018
File : main.c
-----+-----*/
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
#include "sdkconfig.h"

#define NORMAL_LED 23          // NORMAL_LED
#define FAST_LED 5           // FAST_LED
#define VERYFAST_LED 17        // VERYFAST_LED

void NORMAL_LED_Flash(void *pvParameter)
{
    gpio_pad_select_gpio(NORMAL_LED);
    gpio_set_direction(NORMAL_LED, GPIO_MODE_OUTPUT);

    while(1)
    {
        gpio_set_level(NORMAL_LED, 0); // LED OFF
        vTaskDelay(2000 / portTICK_PERIOD_MS); // Wait 2 secs
        gpio_set_level(NORMAL_LED, 1); // LED ON
        vTaskDelay(2000 / portTICK_PERIOD_MS); // Wait 2 secs
    }
}

void FAST_LED_Flash(void *pvParameter)
{
```

```
gpio_pad_select_gpio(FAST_LED);
gpio_set_direction(FAST_LED, GPIO_MODE_OUTPUT);

while(1)
{
    gpio_set_level(FAST_LED, 0); // LED OFF
    vTaskDelay(1000 / portTICK_PERIOD_MS); // Wait 1 sec
    gpio_set_level(FAST_LED, 1); // LED ON
    vTaskDelay(1000 / portTICK_PERIOD_MS); // Wait 1 sec
}

void VERYFAST_LED_Flash(void *pvParameter)
{
    gpio_pad_select_gpio(VERYFAST_LED);
    gpio_set_direction(VERYFAST_LED, GPIO_MODE_OUTPUT);

    while(1)
    {
        gpio_set_level(VERYFAST_LED, 0); // LED OFF
        vTaskDelay(500 / portTICK_PERIOD_MS); // Wait 0.5 secs
        gpio_set_level(VERYFAST_LED, 1); // LED ON
        vTaskDelay(500 / portTICK_PERIOD_MS); // Wait 0.5 secs
    }
}

// Start of main program. The main program starts task LED_FLASH
// 
void app_main()
{
    xTaskCreate(&NORMAL_LED_Flash, "NORMAL_LED_Flash",
               configMINIMAL_STACK_SIZE,NULL,5,NULL);
    xTaskCreate(&FAST_LED_Flash, "FAST_LED_Flash",
               configMINIMAL_STACK_SIZE,NULL,5,NULL);
    xTaskCreate(&VERYFAST_LED_Flash, "VERYFAST_LED_Flash",
               configMINIMAL_STACK_SIZE,NULL,5,NULL);
}
```

Figure 15.9 Program listing

Go to directory **/esp/blink**, and connect your ESP32 DevKitC to your PC.

We are now ready to compile and upload our program to the ESP32 DevKitC. Make sure that the IDF_PATH is set correctly. Enter the following command (note that there is no need to enter the **menuconfig** command again):

```
make flash
```

Wait until you see the success message. Your program should now start working and the 3 LEDs will flash at different rates.

15.4.5 Description of the Program

In this program three tasks are created with the names **NORMAL_LED_Flash**, **FAST_LED_Flash** and **VERYFAST_LED_Flash**. These tasks run continuously in a multi-tasking manner and are started from **app_main** using three **xTaskCreate** functions. In each task the corresponding LED is configured as an output and the LED is flashed ON and OFF using functions **gpio_set_level** with arguments 1 and 0 respectively. The flashing rate is set using functions **vTaskDelay**.

15.5 Summary

In this chapter we have briefly looked at how to install the ESP-IDF environment. We have also developed two simple projects using the ESP-IDF.

The next chapter is about the interesting and useful security features of the ESP32 processor.

Chapter 16 • Security features of ESP32

16.1 Overview

The ESP32 processor has two interesting and useful security features: **Flash Encryption** and **Secure Boot**. The flash encryption and secure boot protect the processor from unwanted access. Flash encryption is separate from Secure Boot and we can for example use flash encryption without secure boot. For the highest security it is recommended to both encrypt the flash and secure the boot process. In this chapter we shall briefly look at both of these features.

16.2 Flash Encryption

The flash is external to the SoC chip and when the device is shipped, any firmware is actually stored on the SPI flash. It is possible that people with sufficient knowledge of the internal architecture of the processor can read and/or modify the contents of the flash where the firmware is stored in plain text form. By encrypting the flash, we can disable its contents to be read or modified.

The ESP32 has a 1024-bit internal eFUSE which is basically a one-time programmable memory and this memory is important in managing the security features. The eFUSE is divided into 4 blocks with 256-bits each block. Blocks 0 and 3 are reserved. The contents of flash are encrypted using AES-256 and Block 1 stores the 256-bit encryption key for flash encryption. Encryption is applied by flashing the ESP32 with plaintext data, and when enabled, the bootloader encrypts the data on first boot. The encryption key is read- and write-protected so that software cannot access it. Similarly, Block 2 stores the encryption keys for secure boot. Once the encryption keys are stored in the eFUSE, we can ensure that no software running on the ESP32 can read or modify these keys. Therefore, these keys are protected from software access, only the hardware can decrypt the contents of flash. The flash controller uses the AES keys stored in the eFUSE to perform the encryption. Any memory-mapped write operations to the flash are first encrypted before being written to flash. Similarly, all memory-mapped read operations are transparently decrypted at run time. Every 32 byte block in flash is encrypted with a unique key derived from the flash encryption key. Once flash encryption is enabled on first boot, the hardware allows a maximum of three subsequent flash updates. During program development, we can use a *pregenerated flash encryption key* to allow re-flashing unlimited number of times.

Flash encryption can be enabled in **make menuconfig** (see Chapter 14), by selecting **YES** for **Enable flash encryption on boot** under Security Features. Secure Boot can also be enabled at the same time under **Secure Boot Configuration** in **make menuconfig**. On first boot, the bootloader sees **FLASH_CRYPT_CNT** eFUSE is set to 0 (which is the factory default) so it generates a flash encryption key using the hardware random number generator. This key is stored in eFUSE. All of the encrypted partitions are then encrypted in-place by the bootloader and the processor should not be interrupted while the first boot encryption takes place, otherwise the flash could be corrupted (this process can take up to several minutes). The **FLASH_CRYPT_CNT** is then set to 1 and this eFUSE limits the number of subsequent re-flashes. When even number of bits (0,2,4,6,8) are set in **FLASH_CRYPT_CNT**, it means that the flash has just been written with plaintext and needs en-

crypting. When odd number of bits (1,3,5,7) are set in **FLASH_CRYPT_CNT** it means that flash encryption is enabled and is functional. After the re-encryption process completes, a "1" is added to **FLASH_CRYPT_CNT** and the eFUSE is re-burned.

Further details on flash encryption can be obtained from the following website:

<https://docs.espressif.com/projects/esp-idf/en/latest/security/flash-encryption.html>

16.3 Secure Boot

Secure Boot ensures that only trusted and known software runs on your device. This feature can be used, for example, to ensure that only your code runs on the device. Secure Boot encryption key is stored in Block 2 of eFUSE. A single eFUSE bit is burned to permanently enable secure boot on the chip. The bootloader generates this key from the internal hardware random number generator. Secure Boot can be enabled from make menuconfig. After a reset, the BootROM uses the secure boot key in eFUSE to verify the Software Bootloader. Once this is verified, then the BootROM executes the Software Bootloader to load the application firmware.

If flash encryption is used without secure boot, it is possible to load unauthorised code using serial re-flashing, and this unauthorised code can then read all encrypted partitions (in decrypted form), thus making flash-encryption ineffective. This can be avoided by write protecting **FLASH_CRYPT_INC** (or by enabling secure boot).

Further details on Secure Boot can be obtained from the following website:

<https://docs.espressif.com/projects/esp-idf/en/latest/security/secure-boot.html>

16.4 Using espefuse.py

espefuse.py can be used to read and write the eFUSE values in the ESP32 processor. **You should take great care using this tool since you can easily permanently damage your ESP32 processor since the eFUSE is one time programmable.**

espefuse.py should be available on your PC if you have already installed V2.0 (or newer) version of esptool.py.

Some examples of using espefuse.py are given in the following sections. Here, it is assumed that your ESP32 DevKitC is connected to serial port COM47. The default baud rate is 115200.

16.4.1 eFUSE Summary

The following command displays a summary of the eFUSES (assuming espefuse.py is not in the path).

```
c:\> cd \msys32\esptool  
c:\msys32\esptool> python espefuse.py -port COM47 summary
```

Figure 16.1 shows the data displayed.

```
C:\nexus32\esptool>python esptool.py --port COM47 summary
esptool.py v2.6-beta1
Connecting.....[REDACTED]
EFUSE_NAME          Description = [Meaningful Value] [Readable/Writeable] <Hex Value>
Security fuses:
FLASH_CRYPT_CNT    Flash encryption mode counter           = 0 R/W <0x0>
FLASH_CRYPT_CONFIG  Flash encryption config (key tweak bits) = 0 R/W <0x0>
CONSOLE_DEBUG_DISABLE Disable ROM BASIC interpreter fallback = 0 R/W <0x0>
BBS_DONE_B          Secure boot enabled for bootloader      = 0 R/W <0x0>
BBS_ABSTRACT_L      Secure boot abstract 1 locked          = 0 R/W <0x0>
JTAC_DISABLE        Disable JTAC                         = 0 R/W <0x0>
DISABLE_DL_ENCRYPT  Disable flash encryption in UART bootloader = 0 R/W <0x0>
DISABLE_DL_DECRYPT  Disable flash decryption in UART bootloader = 0 R/W <0x0>
DISABLE_DL_CACHE    Disable flash cache in UART bootloader   = 0 R/W <0x0>
BLK1                Flash encryption key
= 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 R/W
BLK2                Secure boot key
= 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 R/W
BLK3                Variable Block 3
= 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 R/W

Efuse fuses:
UR_DIS              Efuse write disable mask           = 0 R/W <0x0>
RD_DIS              Efuse read disablemask          = 0 R/W <0x0>
CODING_SCHEME       Efuse variable block length scheme = 0 R/W <0x0>
KEY_STATUS          Usage of efuse block 3 (reserved)     = 0 R/W <0x0>

Config fuses:
XPD_SDIO_FORCE      Ignore MTDI pin (GPIO12) for UDD_SDIO on reset = 0 R/W <0x0>
XPD_SDIO_REG        If XPD_SDIO_FORCE, enable UDD_SDIO reg on reset = 0 R/W <0x0>
XPD_SDIO_TIEH       If XPD_SDIO_FORCE & XPD_SDIO_REG, 1=3.3V 0=1.8V = 0 R/W <0x0>
SPI_PAD_CONFIG_CLK  Override SD_CLK pad (GPIO16/SPICLK) = 0 R/W <0x0>
SPI_PAD_CONFIG_Q   Override SD_DATA_0 pad (GPIO17/SPIQ) = 0 R/W <0x0>
SPI_PAD_CONFIG_D   Override SD_DATA_1 pad (GPIO18/SPID) = 0 R/W <0x0>
SPI_PAD_CONFIG_ID  Override SD_DATA_2 pad (GPIO19/SPIHD) = 0 R/W <0x0>
SPI_PAD_CONFIG_CS0 Override SD_CMD pad (GPIO11/SPICS0) = 0 R/W <0x0>
DISABLE_SDIO_HOST  Disable SDIO host                  = 0 R/W <0x0>

Identity fuses:
MAC                 Factory MAC Address
= 30:ae:a4:85:5b:e0 (CRC 53 OK) R/W
CHIP_VER_REV1       Silicon Revision 1             = 0 R/W <0x0>
CHIP_VERSION        Reserved for future chip versions = 0 R/W <0x0>
CHIP_PACKAGE        Chip package identifier         = 0 R/W <0x0>

Calibration fuses:
BLK3_PART_RESERVED BLOCK3 partially served for ADC calibration data = 0 R/W <0x0>
ADC_UREF            Voltage reference calibration     = 1100 R/W <0x0>

Flash voltage <UDD_SDIO> determined by GPIO12 on reset (High for 1.8V, Low/NC for 3.3V).
C:\nexus32\esptool>
```

Figure 16.1 eFUSE summary

Notice that the eFUSES are set to 0 since they are unburned. There are 32 byte blocks of eFUSES where BLK1 and BLK2 hold the flash encryption and secure boot keys respectively.

FLASH_CRYPT_CNT is an 8-bit field which controls flash encryption. Encryption is enabled or disabled depending on the number of bits in the eFUSE which are set to 1. If an even number of bits (0,2,4,6,8) are set then flash encryption is disabled and any encrypted data cannot be decrypted. If an odd number (1,3,5,7) bits are set then transparent reading of encrypted flash is enabled. If all bits are set (eFUSE value 0xFF) then transparent reading of encrypted flash is disabled and any encrypted data is inaccessible.

By default, the flash encryption process will burn eFUSES **DISABLE_DL_ENCRYPT**, **DISABLE_DL_DECRYPT** and **DISABLE_DL_CACHE** on first boot, where:

DISABLE_DL_ENCRYPT disables the flash encryption operation when running in UART bootloader mode.

DISABLE_DL_DECRYPT disables transparent flash decryption when running in UART bootloader mode

DISABLE_DL_CACHE disables the entire MMU flash cache when running in UART bootloader mode.

Notice that the display also shows the chip ID (MAC address), chip version etc. Users should refer to the ESP32 Technical Manual for further information.

Further information on other espefuse.py commands are available at the following websites:

<https://github.com/espressif/esptool/wiki/espefuse>

<https://github.com/espressif/esptool>

16.4.2 Write Protecting FLASH_CRYPT_CNT

The following command can be used to write protect FLASH_CRYPT_CNT (PORT is the serial port number, e.g COM47):

```
python espefuse.py --port PORT write_protect_efuse FLASH_CRYPT_CNT
```

16.5 Summary

Security and encryption are important issues when we develop applications software that we wish to market. In this chapter we had briefly looked at the basic features of the ESP32 security and encryption processes. Interested readers should refer to the given websites for further detailed information on these topics.

APPENDIX A

List of Components used in the book

A list of all the components used in the book is given in this Appendix:

- 1 x ESP32 DevKitC
- 8 x LEDs (RED)
- 1 x LED (GREEN)
- 8 x 330 ohm resistors
- 2 x push-button
- 1 x Buzzer
- 1 x RGB LED
- 1 x 4-channel Relay Board
- 1 x SD card module
- 1 x Infrared Receiver module
- 1 x TMP36 temperature sensor chip
- 1 x DHT11 temperature and humidity chip
- 1 x MCP4921
- 1 x MCP23017
- 1 x LDR
- 1 x BC108 (or any other PNP) transistor
- 1 x 7 segment LED
- 1 x Small Microphone Module
- 1 x I2C LCD
- 1 x SG90 servo
- 1 x 4x4 Keypad
- 8 x Female-Male jumpers
- 4 x Male-Male jumpers
- 1 x Small breadboard

APPENDIX B

ESP32 DevKitC features used in projects in the book

Feature	Project	Section
Random number generator	5	4.6
	10	4.11
PWM	9	4.10
Built-in Hall sensor	12	4.13
Built-in temperature sensor	13	4.14
Chip ID	14	4.15
ADC	1	5.2
	3	5.4
	4	5.5
	8	5.9
I2C	9	5.10
	10	5.11
	11	5.12
	1	6.2
	4	6.5
	7	6.8
	9	6.10
	10	6.11
External interrupts	12	5.13
Timer interrupts	13	5.14
Touch sensitive inputs	14	5.15
	15	5.16
	16	5.17
SPI	17	5.18
UART	18	5.19
Writing to Flash	19	5.20
	21	5.22
Reading from Flash	20	5.21
	22	5.23
DAC	2	6.3
	3	6.4
SD card	11	6.12
	12	6.13
Infrared	13	6.14
Deep Sleep	14	6.16
	15	6.17
	16	6.18
Wi-Fi		7.3
		7.4
		7.5
		8

		9
		10
		11
		12
Bluetooth BLE	1	13.3
	2	13.4
Bluetooth Classic	3	13.5
	4	13.6
	5	13.7
Python		14
ESP-IDF	1	15.3
	2	15.4
Security		16
Encryption		16

59 Experiments with Arduino IDE and Python

The Complete ESP32 Projects Guide



Prof. Dr. Dogan Ibrahim
has a BSc. in electronic engineering, an MSc. in automatic control engineering, and a PhD in digital signal processing. He has worked in many industrial organizations before he returned to academic life. Prof Ibrahim is the author of over 60 technical books and over 200 technical articles on microcontrollers, microprocessors, and related fields. He is a Chartered electrical engineer and a Fellow of the Institution of Engineering Technology.

ISBN 978-1-907920-75-2



Elektor International Media BV
www.elektor.com

The main aim of this book is to teach the Arduino IDE and MicroPython programming languages in ESP32 based projects, using the highly popular ESP32 DevKitC development board. Many simple, basic, and intermediate level projects are provided in the book using the Arduino IDE with ESP32 DevKitC. All projects have been tested and work. Block diagrams, circuit diagrams, and complete program listings of all projects are given with explanations. In addition, several projects are provided for programming the ESP32 DevKitC using MicroPython.

The projects provided in this book are designed to teach the following features of the ESP32 processor:

- GPIOs
 - Touch sensors
 - External interrupts
 - Timer interrupts
 - I2C and I2S
 - SPI
 - PWM
 - ADC
 - DAC
 - UART
 - Hall sensor
 - Temperature sensor
 - Infrared controller
 - Reading and writing to SD card
 - Reading and writing to flash memory
 - RTC timer
 - Chip ID
 - Security and encryption
 - Wi-Fi and network programming
 - Bluetooth BLE programming
 - Communication mobile devices
 - Low power design
 - ESP-IDF programming

The projects have been organized with increasing levels of difficulty. Readers are encouraged to tackle the projects in the order given. A specially prepared hardware kit is available from Elektor. With the help of this hardware, it should be easy and fun to build the projects in this book.



LEARN ➤ DESIGN ➤ SHARE