

SERIES IN SENSORS

A Hands-On Course in Sensors Using the Arduino and Raspberry Pi



Volker Ziemann



CRC Press
Taylor & Francis Group

A Hands-On Course in Sensors Using the Arduino and Raspberry Pi

Series in Sensors

Series Editors: Barry Jones and Haiying Huang

Other recent books in the series:

Resistive, Capacitive, Inductive, and Magnetic Sensor Technologies

Winnicy Y. Du

Semiconductor X-Ray Detectors

B. G. Lowe and R. A. Sareen

Portable Biosensing of Food Toxicants and Environmental Pollutants

Edited by Dimitrios P. Nikolelis, Theodoros Varzakas, Arzum Erdem,
and Georgia-Paraskevi Nikoleli

Optochemical Nanosensors

Edited by Andrea Cusano, Francisco J. Arregui, Michele Giordano,
and Antonello Cutolo

Electrical Impedance: Principles, Measurement, and Applications

Luca Callegaro

Biosensors and Molecular Technologies for Cancer Diagnostics

Keith E. Herold and Avraham Rasooly

Compound Semiconductor Radiation Detectors

Alan Owens

Metal Oxide Nanostructures as Gas Sensing Devices

G. Eranna

Nanosensors: Physical, Chemical, and Biological

Vinod Kumar Khanna

Handbook of Magnetic Measurements

S. Tumanski

Structural Sensing, Health Monitoring, and Performance Evaluation

D. Huston

Chromatic Monitoring of Complex Conditions

Edited by G. R. Jones, A. G. Deakin, and J. W. Spencer

Principles of Electrical Measurement

S. Tumanski

Novel Sensors and Sensing

Roger G. Jackson

Hall Effect Devices, Second Edition

R. S. Popovic

A Hands-On Course in Sensors Using the Arduino and Raspberry Pi

Volker Ziemann



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2018 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works

Printed on acid-free paper
Version Date: 20180129

International Standard Book Number-13: 978-0-8153-9360-3 (Paperback)
International Standard Book Number-13: 978-0-8153-9359-7 (Hardback)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

Names: Ziemann, Volker (Associate professor of physics), author.
Title: A hands-on course in sensors using the Arduino and Raspberry Pi /
Volker Ziemann.
Other titles: Sensors series.
Description: Boca Raton, FL : CRC Press, Taylor & Francis Group, [2018] |
Series: Series in sensors
Identifiers: LCCN 2017045669 | ISBN 9780815393597 (hardback ; alk. paper) |
ISBN 9780815393603 (pbk. ; alk. paper) | ISBN 9781351188319 (e-book) |
ISBN 9781351188296 (e-book) | ISBN 9781351188302 (ebook) | ISBN
9781351188289 (ebook)
Subjects: LCSH: Detectors. | Raspberry Pi (Computer) | Arduino (Programmable
controller) | Microcontrollers.
Classification: LCC TK7872.D48 Z54 2018 | DDC 681/.20285464--dc23
LC record available at <https://lcn.loc.gov/2017045669>

Visit the eResource: <https://www.crcpress.com/9780815393603>

Visit the Taylor & Francis Web site at
<http://www.taylorandfrancis.com>

and the CRC Press Web site at
<http://www.crcpress.com>

Contents

Preface	xv
Acknowledgments	xvii
CHAPTER 1 ■ Introduction	1
CHAPTER 2 ■ Sensors	5
2.1 ANALOG SENSORS	5
2.1.1 Resistance-based sensors	5
2.1.2 Voltage-based sensors	10
2.1.3 Current-based sensors	14
2.2 SIGNAL CONDITIONING	16
2.2.1 Voltage divider	16
2.2.2 Amplifiers	17
2.2.3 Filters	21
2.2.4 Analog-to-digital conversion	23
2.2.5 Supply voltage	26
2.3 DIGITAL SENSORS	28
2.3.1 Buttons and switches	29
2.3.2 On/off devices	30
2.3.3 I2C devices	32
2.3.4 SPI devices	35
2.3.5 RS-232 devices	36
2.3.6 Other sensors	36
CHAPTER 3 ■ Actuators	39
3.1 SWITCHES	39
3.1.1 Light-emitting diodes and optocouplers	39
3.1.2 Large currents	41
3.2 MOTORS	43
3.2.1 DC motors	44
3.2.2 Servomotors and model-servos	46
3.2.3 Stepper motors	47

3.3	ANALOG VOLTAGES	51
3.4	OTHER ACTUATORS	52
CHAPTER 4 ■ Microcontroller: Arduino		55
4.1	HARDWARE	55
4.1.1	Arduino UNO	55
4.1.2	ESP8266 and NodeMCU	56
4.2	GETTING STARTED	57
4.3	HELLO WORLD, BLINK	58
4.4	INTERFACING SENSORS	60
4.4.1	Button	60
4.4.2	Analog input	61
4.4.3	I2C	65
4.4.4	SPI	77
4.4.5	Other protocols	80
4.5	INTERFACING ACTUATORS	85
4.5.1	Switching devices	85
4.5.2	DC motors	87
4.5.3	Servos	91
4.5.4	Stepper motors	92
4.5.5	Analog voltages	101
4.5.6	Human attention actuators	102
4.6	COMMUNICATION TO HOST	103
4.6.1	RS-232 and USB	103
4.6.2	Bluetooth	104
4.6.3	WiFi	105
4.6.4	Other communication	110
CHAPTER 5 ■ Host Computer: Raspberry Pi		113
5.1	HARDWARE	113
5.2	GETTING STARTED	113
5.3	INSTALLING AND USING NEW SOFTWARE	117
5.4	RASPI AS A ROUTER	121
5.5	COMMUNICATION WITH THE ARDUINO	123
5.5.1	Arduino IDE	123
5.5.2	From the command line	124
5.5.3	Python	125
5.5.4	Octave	129
5.6	DATA STORAGE	132
5.6.1	Flatfile	132
5.6.2	MySQL	134

5.6.3 RRDtool	138
5.7 ONLINE PRESENTATION	141
CHAPTER 6 ■ Control System: EPICS	147
6.1 INSTALLATION	147
6.2 COMMUNICATING WITH EPICS	149
6.3 ASYN AND STREAM LIBRARIES	150
6.4 WRITING AN IOC	151
6.5 STARTING THE IOC AT BOOT TIME	154
CHAPTER 7 ■ Messaging System: MQTT	157
7.1 BROKER	158
7.2 NODEMCU CLIENTS	159
7.3 GATEWAY TO EPICS	161
CHAPTER 8 ■ Example: Weather Station with Distributed Sensors	167
CHAPTER 9 ■ Example: Geophones	177
CHAPTER 10 ■ Example: Monitor for the Color of Water	185
CHAPTER 11 ■ Example: Capacitance Measurement	191
CHAPTER 12 ■ Example: Profile of a Laser Beam	197
CHAPTER 13 ■ Example: Fire-Seeking Robot	205
CHAPTER 14 ■ Presenting and Writing	223
14.1 PREPARING A PRESENTATION	223
14.2 PREPARING A REPORT	225
14.3 PRESENTING DATA	226
14.4 GOOD ENGLISH	227
14.5 POSTSCRIPTUM	228
APPENDIX A ■ Basic Circuit Theory	229
APPENDIX B ■ Least-Squares Fit and Error Propagation	231
Bibliography	235
Index	237



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

List of Figures

1.1	The outline of the book.	2
2.1	Image of a light-dependent resistor (LDR) on the left and how to connect it in a plain voltage-divider configuration (center) and in a Wheatstone bridge (right).	5
2.2	The band-level scheme and the schematics for the circuit with the LDR in a voltage divider. The upper graph illustrates dark conditions and the lower graph shows conditions where the LDR is exposed to light. Note that the vertical axis by convention shows the energy of electrons. This causes the positive pole of the battery to have the most negative energy. See the text for a discussion.	6
2.3	NTC resistors are doped semiconductors that have donor levels just below the conduction band. Increasing the temperature increases the kinetic energy of the electrons that allows them to occasionally jump into the conduction band, where they contribute to the conductivity of the material.	8
2.4	A linear potentiometer (left) and a circuit illustrating the electric connections (right).	9
2.5	A joystick is shown on the left and a fluid-level resistive sensor on the right.	10
2.6	A strain gauge.	10
2.7	Schematic of an MQ-x gas detector (left) and a sensor mounted on a small breakout board (right).	11
2.8	Image of an LM35 temperature sensor (left) and how to connect it (right).	11
2.9	On the left we show a schematic of a thermocouple on the top and a thermopile on the bottom. On the right we show an image of an MLX90614 contact-free thermometer.	12
2.10	Schematic of a Hall sensor (left) and the A1324 sensor (right).	12
2.11	The operational principle of an ADXL accelerometer.	13
2.12	An SM-24 geophone with a diameter of about 30 mm.	14
2.13	Energy-band diagram (left) and circuit (right) of a reverse-biased pin diode in photoconductive mode.	14
2.14	Image of a BPW34 pin diode on the left and two phototransistors on the right; an IR-sensitive BPX38 and an SFH3310, sensitive in the visible part of the spectrum.	15
2.15	On the left we have a voltage divider to reduce the input voltage of 0-5 V to 0-3.3 V. The right circuit shows the use of clamping diodes to protect the input of the microcontroller to lie between ground and 5 V.	16

2.16	A bare operational amplifier shown on the left and shown wired as a line buffer on the right. We omitted the supply wires on the latter schematic.	17
2.17	A non-inverting (left) and inverting (right) amplifier.	18
2.18	A difference amplifier (left) and the same circuit with adjustable V_1 (right) that is used to subtract the baseline.	19
2.19	Instrumentation amplifier (left) and logarithmic amplifier (right).	19
2.20	Amplifying a weak signal and shifting it to mid-range between the power rails.	20
2.21	A simple low-pass (left) and high-pass (right) filter.	21
2.22	A simple band-pass (left) and band-stop (right) filter.	22
2.23	An active non-inverting (left) and inverting (right) low-pass filter.	22
2.24	The operating principle of a 3-bit flash ADC.	23
2.25	The operating principle of a successive approximation ADC.	24
2.26	The operating principle of a delta-sigma ADC.	24
2.27	Sampling a signal in different Nyquist zones. The dashed line displays a frequency of $0.15f_s$ and the solid lines show signals with frequency $(1 - 0.15)f_s$ on the left and $(1 + 0.15)f_s$ on the right. Note that at the times when the signal is sampled (indicated by boxes), the signals are indistinguishable.	25
2.28	The dashed signals s_1 and s_2 are the images aliased into the base band of the original signals S_1 and S_2 in higher Nyquist zones.	26
2.29	Schematics of very simple power supply circuits.	27
2.30	Variable-voltage (left) and fixed-voltage (right) regulator circuits.	28
2.31	Connecting a switch or button with a pull-up resistor.	29
2.32	Level shifter circuitry using a n-type MOSFET. The source of the MOSFET is connected to the 3.3 V logic and the drain to the 5 V logic.	30
2.33	Schematic view of a PIR sensor (left) and the hardware (right).	31
2.34	HR-SR04 distance sensor.	31
2.35	Illustration of the operational principle of a barometric pressure sensor.	32
2.36	A BMP180 barometric pressure sensor and an HYT-221 humidity sensor.	33
2.37	The operational principle of one gyroscope in the MPU-6050.	34
2.38	An MPU-6050 accelerometer on a breadboard.	35
2.39	A GPS receiver on the left and a DHT11 humidity sensor on a breadboard on the right.	36
2.40	A Shinyei PPD42NS particle sensor (left) and a GP2Y1010AU0F dust sensor (right).	37
3.1	A close-up of a light-emitting diode is shown on the left. In the center is the schematic of connecting an LED and the same circuit on a breadboard.	40
3.2	The physics of an LED.	41
3.3	The terminals of an NPN transistor (left), and using an NPN transistor as switch (right).	42
3.4	Two NPN transistors connected to form a Darlington pair (left) and a ULN2003 Darlington array (right).	42

3.5	Schematic illustrating the functionality of a relay (left) and an image of a relay (right).	43
3.6	Basic operation principle of a DC motor. The magnetic north pole of the stator is to the left and the south pole to the right of the coil. Current is supplied to the brushes where it enters the commutator and passes through the rotor-coil, where the Lorentz force causes a force on the wire, moving it upwards. After half a turn, the commutator has rotated and reverses the polarity such that the current on the left side of the coil points in the same direction as before.	44
3.7	Schematic illustrating the functionality of an H bridge.	45
3.8	A small model-servo.	46
3.9	The timing of the control signal for the servo.	47
3.10	Schematics of a stepper motor. Note that the coils are denoted by upper-case letters and the corresponding terminals with lower-case letters.	48
3.11	Wiring of the coils in a bipolar stepper motor (left) where the respective center terminals are often connected. In a unipolar motor (right) the center terminals are exposed.	50
3.12	Connection for a unipolar stepper motor.	51
3.13	Operational principle of a digital-to-analog converter.	51
3.14	Sketch of a butterfly valve.	53
4.1	An Arduino UNO.	56
4.2	The ESP-01 on the left and a NodeMCU on the right.	56
4.3	The Arduino IDE.	57
4.4	The “Blink” sketch.	59
4.5	Interfacing a button to the Arduino.	60
4.6	Interfacing a potentiometer (left) or an LM35 temperature sensor (right) to the Arduino.	62
4.7	Connecting the MLX90614 contact-free thermometer to an Arduino UNO.	66
4.8	Connecting the HYT-221 humidity sensor to the Arduino (left) and using the serial plotter to display the humidity and temperature measurements (right).	67
4.9	Connecting the MCP23017 IO-extender to the UNO.	74
4.10	Connecting the MCP3304 ADC to the NodeMCU.	77
4.11	Connecting the DS18B20 sensor to the Arduino UNO.	82
4.12	Oscilloscope trace of a pulse-width modulated signal on pin D9 of an Arduino UNO with values of 88 (left) and 220 (right).	85
4.13	Using pulse-width modulation and a transistor to adjust the brightness of an LED.	87
4.14	Controlling the speed of a motor with pulse width modulating the base of a TIP-120 Darlington transistor. See the text for a discussion of the diodes.	88
4.15	The pin assignment of the L293D H-bridge driver.	88
4.16	Controlling speed and direction of a DC motor.	89
4.17	Connecting a model-servo to the Arduino UNO.	91

4.18	Connecting a unipolar stepper motor to the UNO with a ULN2003 Darlington driver.	93
4.19	Connecting a bipolar stepper motor to the UNO with an L293D H-bridge driver.	94
4.20	Connecting a bipolar stepper motor to the UNO with a DRV8825 stepper motor driver with microstep capability.	98
4.21	Adjusting the maximum current on the DRV8825 breadboard.	99
4.22	Connecting an MCP4921 12-bit DAC to the Arduino UNO.	101
4.23	Connecting a speaker and a piezo buzzer to the Arduino.	103
4.24	Front and back side of the HC-06 Bluetooth dongle and the connection to the Arduino.	104
4.25	The NodeMCU with an LM35 temperature sensor.	106
4.26	The communication with a web server showing the HTTP header.	108
5.1	A Raspberry Pi board.	114
5.2	The Accessory menu behind the button with the raspberry shows a number of installed programs such as the Terminal (command) window.	115
5.3	The nano text-editor with available commands listed on the bottom.	118
5.4	The Synaptic Package Manager.	119
5.5	Telnet session connected wirelessly to the NodeMCU.	124
5.6	Using Python on the Raspi to communicate with the query_response sketch running on the Arduino UNO.	125
5.7	Simple ASCII graphics from querying the UNO repeatedly.	126
5.8	Plot from the temperature logger.	131
5.9	Graphics from rrdtool .	140
5.10	Web page from the Raspi after installation of the Apache2 web server.	141
5.11	The first self-made web page.	142
5.12	User pi 's home page that displays the continuously (but slowly) updated temperature.	144
6.1	Using the EPICS commands caget and caput .	150
8.1	The weather-node circuit with the NodeMCU on the right and the LM35, HYT221, and BMP180 sensors towards the left.	168
8.2	The weather station web page served by the Raspi.	173
9.1	The amplifier to interface an SM-24 geophone to the NodeMCU.	178
9.2	The prototype with the SM-24 geophone connected to the amplifier board and the NodeMCU.	180
9.3	The raw time series from the geophone and the corresponding spectrum.	181
10.1	The setup to measure the color-dependent absorption (left) and the reflection from a surface (right).	185
10.2	The setup to measure the water color with an Arduino UNO.	186

10.3	The color sensor with the RGB-LED on the left and the phototransistor on the right.	188
11.1	The setup to measure the capacitance.	192
11.2	The waveform of the voltage on the capacitor as a function of time, while a $2.2\,\mu\text{F}$ Tantal capacitor discharges through a $33\,\text{k}\Omega$ resistor on a linear scale (above) and on a logarithmic scale (below).	195
12.1	The schematic setup to measure the beam size of a laser pointer.	197
12.2	The chassis from the CD-ROM drive with the laser mounted on the top right and the photoresistor on the bottom right. The black obstacle is mounted on the carriage that can be moved via the spindle on the top by a small stepper motor that is located below the laser.	198
12.3	The schematics of the circuit. The stepper motor of the frame is connected to points labelled PA, . . . , PD.	199
12.4	The raw sensor value as a function of the position of the obstacle and the derived laser beam profile, which shows a moderate asymmetry.	203
13.1	The chassis of the robot with two breadboards. The smaller one can be turned by operating a model-servo. Mounted on the larger breadboard are a NodeMCU on the right and a second chip on the left, a bare ATmega328 that was initially tested and later replaced by an Arduino NANO.	206
13.2	Simplified setup of the remote controller on a breadboard.	207
13.3	The schematic of the remote controller.	207
13.4	The electronics circuit of the robot (color version available online).	212
13.5	The schematic of the robot electronics.	213
13.6	The operational robot from the back (left) and from the front (right).	221
A.1	Two impedances connected in series (left) and in parallel (right).	230



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Preface

Some years ago three young students inquired about a moderately complex project to earn some credits. I happily agreed to supervise them and assigned moderately difficult tasks, namely, to build from scratch a data acquisition system for slow signals. I suggested to connect some sensors to an Arduino microcontroller and then write a program for the Arduino to interface the measurement values to the control system we use in our lab.

The students were very dedicated and a real joy to work with. They had the Arduino under control within a few hours and had the first sensors reporting their measurement values after the first day. Then they worked out a protocol that is compatible with our EPICS-based control system, and after discussions with our control systems experts and even more debugging, eventually the students had a prototype system working. After cleaning up their project, they had to give a presentation and write a report to earn their well-deserved credits.

I soon realized that there is a clear progression of the information generated by a sensor. The information bubbles upwards through a sequence of microcontrollers and computers that provide data-handling, storage, and online presentation to a seminar presentation, and eventually ends up in a report. Understanding the path the measurement data take appeared like a useful concept to communicate to students. Moreover, I wanted the students to understand the details of the signal chain and how it *really* works. Therefore, I used the hands-on approach with programming the Arduino that serves as communication glue between the sensor and the control system. This proved beneficial for the students' understanding and was appreciated by them. The abstract concepts thus led to a very concrete realization. In the final stages of the project I coached the students on how to prepare a presentation for a seminar according to some simple guidelines, and eventually put the oral presentation into writing for a report to hand in and receive their credits.

This book is inspired by these students and their projects, but goes a step further and adds a number of additional topics such as signal conditioning, controlling actuators such as switches and motors, as well as control system setup, data storage, and networking. Please note that I cover only basic examples that are boiled down to the bare essentials in order to illustrate the main concepts and to get started quickly. Anyway, the concepts covered should come in handy when working with real-world data-acquisition tasks. I basically follow *Mrs. Robinson's guideline* of "help you learn to help yourself" (remember the Simon and Garfunkel song?) and try to fill the toolbox with practical know how. This know how should enable the reader to help herself and pick up datasheets and manuals to adapt the basics from this book to realize far more advanced projects.

User Guide

The main theme of the book is *From Sensor to Report*, and that should be the guiding principle of using it in the classroom, either in a student laboratory or as the basis for individual projects.

For a student laboratory I suggest installing the software with some of the more arcane instructions before starting the lab. This comprises turning the Raspi into a router (Sec-

tion 5.4), installing the MySQL database (Section 5.6.2), and installing EPICS (Section 6.1). The students should focus on the sensors and use the above systems as a background infrastructure. They should, on the other hand, understand the basic operation of the sensors, learn how to interface them to a microcontroller, and move the information to the next level on a different computer. This requires them to write network code, fill an SQL database, prepare the protocol files for EPICS, or present data on a web server. In the lab a knowledgeable supervisor, a “tutor”, should be available to answer questions and guide the students. Using solderless breadboards in the lab enables the students to quickly arrive at a working system on which to base further experiments and try out new ideas.

A suitable scope for student projects, suitable for a single or a group of two students, is to connect a small number of sensors to an Arduino. Then they should be given a target system where they can publish the data. This can be a database, EPICS, MQTT, or a web page. After a prototype system is working, the students should present their system in a seminar and prepare a report.

All code and the corresponding images of the circuits on a breadboard, prepared with Fritzing [1], are available on this book’s web site at <https://www.crcpress.com/9780815393603>.

Acknowledgments

This book only materialized because my students, Adam, Måns, and Frida, asked about “some project” and then completed it with such enthusiasm. I gratefully acknowledge their contributions and input.

I gratefully thank my colleagues Roger Ruber and Mattias Klintenberg. They read and commented on parts of the manuscript and provided essential feedback. All remaining errors are of course my responsibility.

I thank Camilla Thulin, Uppsala University, for her help with the photography.

This book is only possible thanks to the open source community that created the Arduino and Raspberry Pi ecosystems and the large number of people who answer questions on Internet forums.

I acknowledge the creators and maintainers of the Fritzing software. I relied on it to prepare many drawings to illustrate the wiring of circuits.

I am indebted to my editor Francesca McGowan and to Rebecca Davies at Taylor & Francis for competently guiding me through the intricacies of writing and publishing a book.

Last, but not least, I acknowledge my family for putting up with me during the writing and editing period, when I was more often than not absent minded and showed a distinct lack of response to other matters.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Introduction

What is the path that the electrical signal from a sensor takes to end up in a report? We address this question because collecting sensor data, processing them, and deriving some understanding from the data plays an important role in many circumstances. One example is a utility company that gathers information about electricity, heating, and water in order to prepare statements for their customers and to estimate demand for their product in the future. Smart homes are another example; they measure temperatures or detect the presence of beverages in the refrigerator to adjust the thermostat in the first case, or to prepare a report for us to pick up some milk or beer on the way home in the second case. Quite generally, many Internet of Things (IoT) technologies share a common base with the topic of the book, but even large experimental collaborations such as the ATLAS [2] or CMS [3] that operate the huge detectors at the Large Hadron Collider (LHC) [4] at CERN [5] get their data from sensors that are buried deep inside the detectors. They sense currents from drift chambers where charged particles cause a discharge between wires at different potentials, or they cause electrical signals from semiconductor detectors, where they create electron-hole pairs that induce a current. Other examples are Hall sensors, to measure magnetic fields, and humidity sensors or barometric pressure sensors to detect variations of ambient conditions. All these sensors produce electrical signals that often need to be amplified or otherwise conditioned. This stage involves operational amplifiers and various filters to improve the signal-to-noise ratio. Once properly processed, the analog signals are passed on to analog-to-digital converters (ADC), where they are converted to a digital representation that is subsequently handled by computers. Often some of the computing power is located close to the sensor and is provided by microcontrollers that collect signals from nearby sensors and convert them to the underlying physical quantities, formatted to have a standardized output format. Thus they act as “communication glue” between the specific interface to the sensor and a more generic interface to a host computer that is usually located further away. The latter is the other end of the communication channel from the microcontroller and provides data storage and presentation, and sometimes also shows recent data for on-line monitoring. The host computer may run generic control-system software to provide a further abstraction layer towards higher-level software. Examples we discuss are MQTT [6], which is popular with IoT projects, and the EPICS control system [7], commonly found in scientific laboratories.

In this book we will build a system that contains all the ingredients also found in large scientific or industrial installations. In a sense it is a simplified model of a large installation, yet containing all the hardware and logical building blocks. In particular, we use the Arduino microcontroller [8] as the local intelligence to control switches and motors to move the sensors around and enable reading them out and translating the signals to a format that allows communication with a host computer using a standardized protocol.

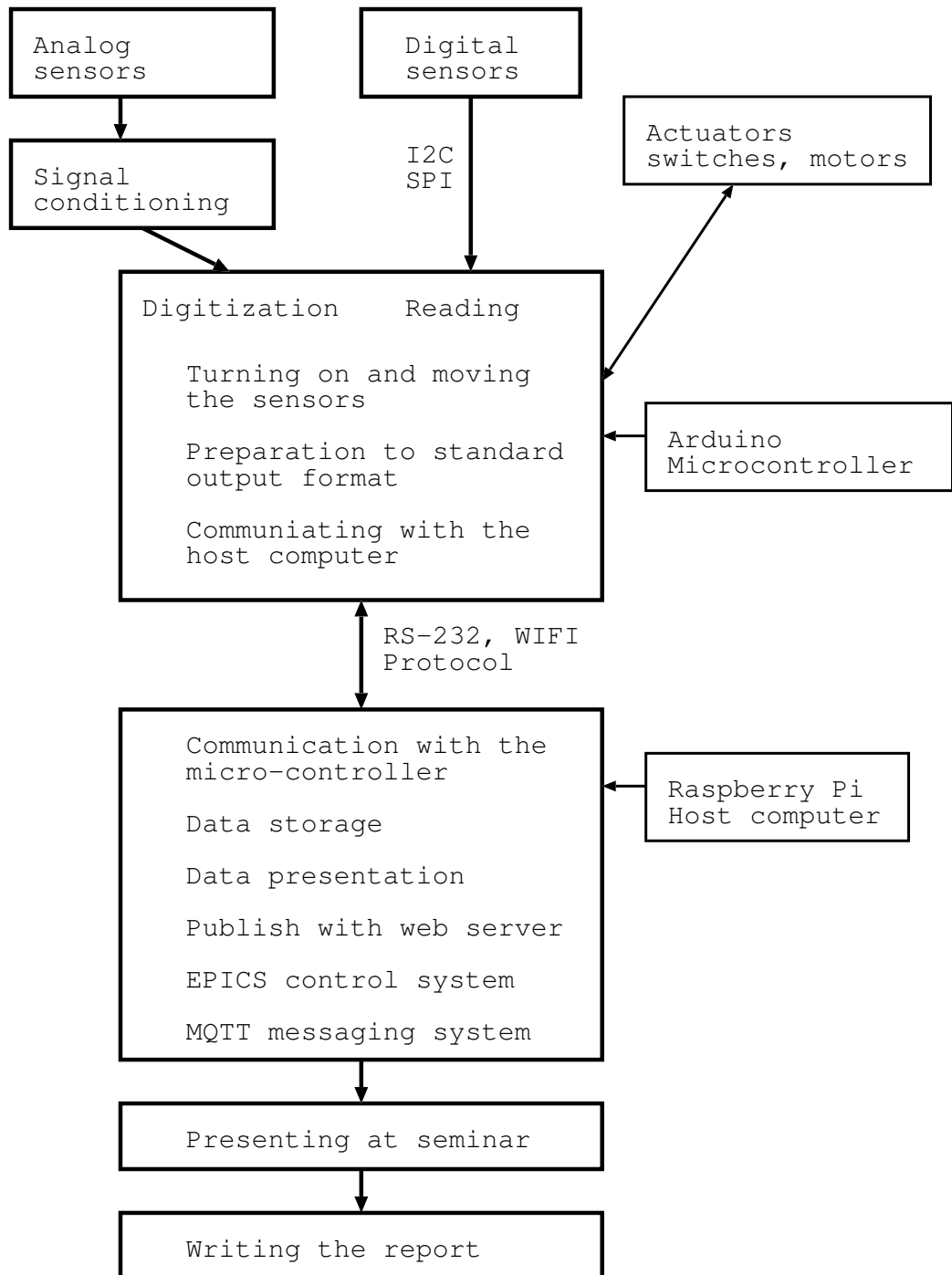


Figure 1.1 The outline of the book.

This comprises the hardware channels, which can be USB, RS-232, Bluetooth, or WiFi as well as the logical protocol, such as a simple query-response protocol. As host computer we use a Raspberry Pi [9] which, in its most recent incarnation, features four processing cores and runs a standard Linux system with a huge base of available software, including web servers, the MATLABTM-clone Octave, and even MathematicaTM, all without license charges.

The remainder of the book is organized as laid out in Figure 1.1. We first discuss a number of analog sensors and signal-conditioning methods, followed by a number of sensors that already provide their measurements in digital form, and the buses and protocols used. Next we discuss actuators. They switch things on and off, even those requiring large currents, and we learn how to control different types of motors that are sometimes needed as part of the measurement process. We then describe how to interface the actuators and the sensors with the Arduino, either by digitizing the signals or by using the appropriate bus-interface. We go on to describe a program structure that permits the Arduino to support a simple query-response protocol in order to serve as a slave to a host computer. Next, we configure a Raspberry Pi as a standardized host computer. It will provide data storage in databases, and present the data in graphical form either using Octave running locally on the Raspi or by publishing our measurement data with a web server, also running on the Raspi. We continue the discussion by installing the EPICS control system software and turn the Raspi into a full-blown control-system server that can join any other EPICS installation in a transparent way. We go on to discuss the MQTT message-passing system, which plays an important role in IoT applications. Having assembled all the parts, we consider examples in which we build a weather station with distributed sensors, and systems to record ground vibrations, monitor the color of water, and measure the capacitance of a capacitor. In two more advanced examples, we build a system to measure the width of the beam of a laser pointer, and a remote-controlled robot that also senses flames autonomously, moves to the fire, and sounds an alarm. We conclude with presenting guidelines about how to prepare a seminar presentation based on the examples and how to write a readable publication describing our data acquisition system using sensors, actuators, Arduino, and Raspberry Pi.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Sensors

A *sensor* is a device that converts a physical quantity to an electrical signal [10, 11, 12, 13], and therefore either provides a voltage or a current, or causes a change of its resistance. More generally, the impedance of the sensor, which also comprises capacitive and inductive sensors, may change. We are thus faced with the task to measure either of these electrical quantities.

Below we discuss examples of the different types of sensors. The examples only show a selection of those available on the market, and searching the Internet for the physical quantity one wants to measure jointly with the keyword “sensor” will give an idea of what is available. Once the sensor is identified, careful reading of the datasheet to learn about how to interface the sensor is mandatory.

2.1 ANALOG SENSORS

We start by considering resistance-based sensors.

2.1.1 Resistance-based sensors

The first resistance-based sensor we look at is a *light-dependent resistor* or LDR, shown on the left of Figure 2.1. It changes its resistance depending on the exposure to light. The range of variation depends on the device and typically ranges from a few $100\ \Omega$ to $M\Omega$. The

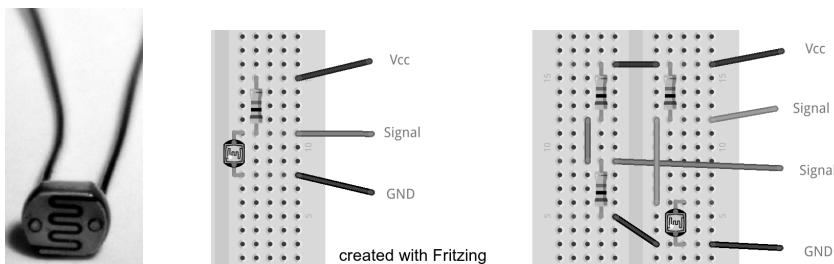


Figure 2.1 Image of a light-dependent resistor (LDR) on the left and how to connect it in a plain voltage-divider configuration (center) and in a Wheatstone bridge (right).

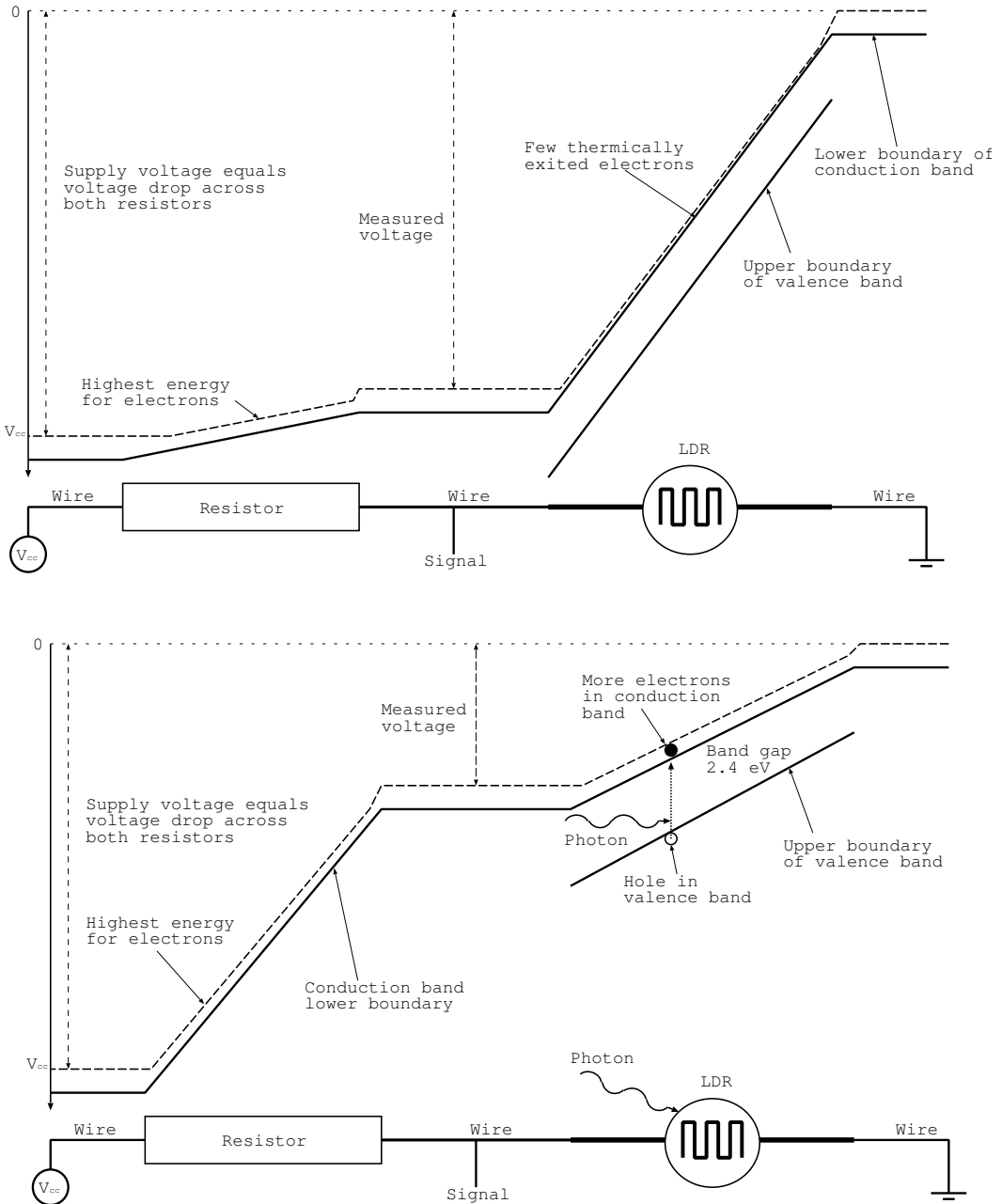


Figure 2.2 The band-level scheme and the schematics for the circuit with the LDR in a voltage divider. The upper graph illustrates dark conditions and the lower graph shows conditions where the LDR is exposed to light. Note that the vertical axis by convention shows the energy of electrons. This causes the positive pole of the battery to have the most negative energy. See the text for a discussion.

operating principle of this and many other sensors is based on the availability of electrons in the conduction band of a material. In good (wires) or bad (resistors) conductors, electrons partially fill the available states in the conduction band up to some energy, the Fermi level, whereas in insulators the Fermi level is located in between the completely filled valence band and the conduction band [14]. Therefore, no electrons are available in the conduction band. Furthermore, the energy-difference between the upper boundary of the valence band and the lower boundary of the conduction band, the *bandgap*, is large, while for semiconductors it is on the order of electron-volt (eV).

In photoresistors the base material is often CdS, a semiconductor with a bandgap of about 2.4 eV. This energy equals that of photons of green light with a wavelength of about 500 nm. Therefore, green photons can elevate electrons from the valence to the conduction band and thus create electron–hole pairs. These now freely moving charge carriers conduct electric current and therefore increase the conductivity of the material. Figure 2.2 illustrates this in more detail. The upper figure shows a simplified band level scheme under dark conditions. The bold lines show the lower boundary of the conduction band and the upper boundary of the valence band. The dashed line shows the highest energy-states that electrons occupy. For metals and resistors, this is close to the Fermi level, but in a semiconductor, like CdS, the Fermi level lies between valence and conduction band. Yet, at room temperature, there are a few thermally excited electrons in the conduction band of CdS. We visualize this by the close proximity of the dashed line to the lower conduction-band boundary. In metals there are plenty of electrons in the conduction band, and the conductivity is high. In the resistor there are fewer electrons in the conduction band, or their mobility is impeded in other ways such that there is a shift in the Fermi level across the resistor. In dark conditions there are only very few thermally excited electrons in the conduction band of the LDR, and the conductivity is very low. Consequently, there is a large voltage-drop across the LDR and the measured voltage, which is the difference of Fermi levels between the measurement points. The measured voltage is therefore close to the full voltage delivered by the battery. If, on the other hand, the LDR is illuminated, the photons lift electrons from the valence band into the conduction band. This increases the conductivity and only a small voltage is dropped across the LDR. This consequently reduces the measured voltage, as shown in the lower graph of Figure 2.2. Note that we do not discuss the details of the interfaces between the different parts because it is beyond the scope of this book.

In the middle of Figure 2.1 we show how to connect a photoresistor in series with a resistor R_0 (here 10 k Ω) to create a voltage divider between the supply voltage V_{cc} and ground. From the discussion of voltage dividers and a short refresher of basic circuit theory in appendix A or [15, 16], the voltage V_s on the signal terminal is then given by $V_s = V_{cc}R_{LDR}/(R_0 + R_{LDR})$. Thus, the illumination of the LDR changes its resistance R_{LDR} and the signal voltage varies correspondingly. Note that we have to select the resistor R_0 in the middle of the range of R_{LDR} . This causes the voltage we measure to be around one half of the supply voltage. Therefore, we also need to use a volt meter in that voltage range. Very small variations of the light intensity are then difficult to resolve and may need to be amplified. Using a Wheatstone bridge, where we compare the voltages in two resistor dividers, as shown on the right of Figure 2.1, helps to alleviate this problem. We expand on the use of Wheatstone bridges in Section 2.2.

Other resistance-based sensors are *resistance-based temperature detectors* (RTD) such as the PT100 temperature sensor. It is a calibrated platinum-based sensor with a resistance of exactly 100 Ω at 0 °C. It is based on the fact that the resistance of a very pure metal is determined only by scattering of electrons in the conduction band with phonons, which are vibrations of the ions that make up the crystal lattice of the metal. Moreover, higher

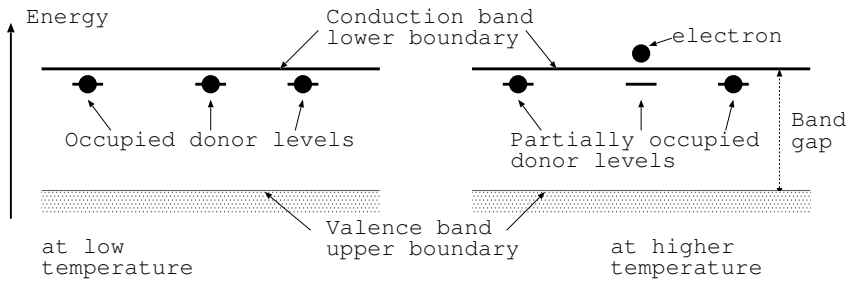


Figure 2.3 NTC resistors are doped semiconductors that have donor levels just below the conduction band. Increasing the temperature increases the kinetic energy of the electrons that allows them to occasionally jump into the conduction band, where they contribute to the conductivity of the material.

temperatures cause stronger vibrations of the lattice, with correspondingly higher resistance. Intuitively one might think of the crystal ions at higher temperature to oscillate with larger amplitudes, creating a larger target for the electrons to scatter, thus impeding their motion. Since this is an intrinsic property of the material, calibration measurements of resistance as a function of temperature are universally valid for all sensors of the same metal, provided the metal is very pure and free of imperfections. Commercial sensors are often made of platinum wire wound on a ceramic support body. The PT100 sensors are connected to a calibrated current source and the voltage drop across the sensor is measured with a volt meter, just as any other resistance measurement.

Thermistors are resistors that have their temperature dependence deliberately made large. In *positive temperature calibration* (PTC) devices, the resistance increases with temperature, and in *negative temperature calibration* (NTC) devices, it decreases. PTCs are mostly used as protection devices that switch the resistance from a low- to a high-resistance state if a certain temperature is exceeded. They are based on polycrystalline materials that change their dielectric constant at a certain temperature, the Curie temperature, by a large amount. Above the Curie temperature, the state of the magnetic dipoles is disordered and the dielectric constant is small. This causes the formation of large potential barriers between the crystal grains, which leads to a high resistance. Below the Curie temperature the molecular dipoles are aligned, the dielectric constant is large, and the resistance is low. A typical application of the PTC thermistor is a self-regulating heater, in which the heater also warms up the thermistor, which increases the resistance and limits the current to the heater until an equilibrium is found. PTCs can also be used to detect whether a threshold temperature is exceeded.

The converse thermistors are NTCs, which decrease their resistance with increasing temperature. They are often used for temperature sensing and are based on a doped semiconducting material that has occupied impurity donor levels below the conduction band, as shown in Figure 2.3. Increasing the temperature thermally excites these electrons to jump into the conduction band, thus increasing the conductivity. This effect is much larger than the reduction of the resistance due to the ions oscillating and impeding the motion of the electrons, which was responsible for the temperature dependence in the PT100 sensor. Both NTC and PTC thermistors are sensed by connecting them to a constant-current source and measuring the voltage drop across the thermistor.

A number of *position sensors* are based on potentiometers. A potentiometer is a variable resistor where a slider moves up and down a resistance and shortens the distance of one end point to the slider, thereby reducing the resistance between two terminals. The distance between the slider and the other end point lengthens, causing the resistance between the slider and the other terminal to increase correspondingly. On the left-hand side of Figure 2.4 we show a potentiometer with three connectors; the two end points are connected to dark wires and the one controlled by the slider is connected to a lighter-colored wire. The schematic view on the right of Figure 2.4 explains the functionality; the slider controls a variable mid-point of a voltage divider and the output voltage interpolates from 0 to 5 V in this case. A variation of the potentiometer is a *joystick*, which is based on two orthogonally mounted potentiometers, controlled by a small stick. We simply need to measure the voltage on the output with respect to ground in order to determine the position of the slider or the stick. An image of a joystick is shown on the left of Figure 2.5.

Also, *fluid levels* of liquids with a small conductivity can be determined with a resistor whose resistance is varied by reducing the resistance between conducting stripes shown in Figure 2.5. The sensor is connected just as a potentiometer with the liquid level acting as the slider.

Applying an external force to a material will cause a change its equilibrium shape, and therefore *strains* the material. An example is a stretched wire that will get longer and thinner if pulled. Consider the resistance $R = \rho L/A$ of the wire, given in terms of resistivity ρ , length L , and cross section A , and consider its change. We see that increasing L and decreasing A increases the resistance R . Thus, the simple wire converts its deformation to a small change in the resistivity, and therefore serves as a *force-sensitive resistor* that is often used as one branch in a Wheatstone bridge. The small effect of only changing the geometry of the wire can be greatly enhanced by using heavily doped semiconductors instead. The latter also change their resistivity ρ by a large amount when strained. Alternatively, they are based on polymer thick-film technology. An example is the strain gauge shown in Figure 2.6.

The MQ-2 and other MQ-x are *gas sensors* sensitive to different types of gas, depending on their type specification. They are based on a semiconductor substrate with a thin surface layer of polycrystalline SnO_2 tin oxide that is either sputtered or deposited by evaporation. The specificity to various gases depends on the temperature of the active area, which is adjusted by a heater, by the method of deposition, or by small additions of other materials, such as palladium, gold, or platinum. The variation of the resistance depends on the grain boundaries of the polycrystalline SnO_2 and on the insulating oxide layer between the grains

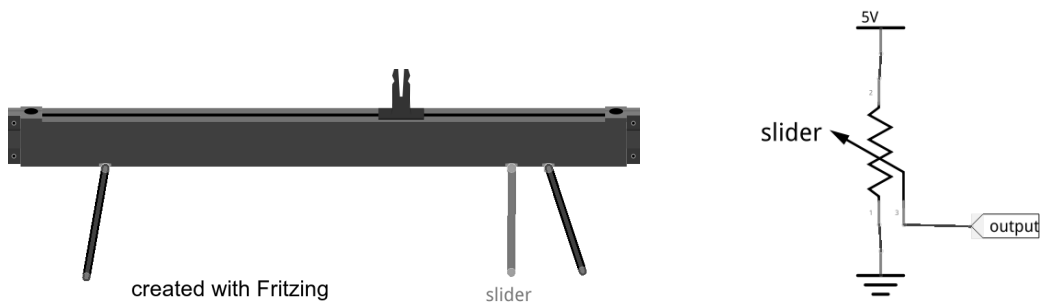


Figure 2.4 A linear potentiometer (left) and a circuit illustrating the electric connections (right).

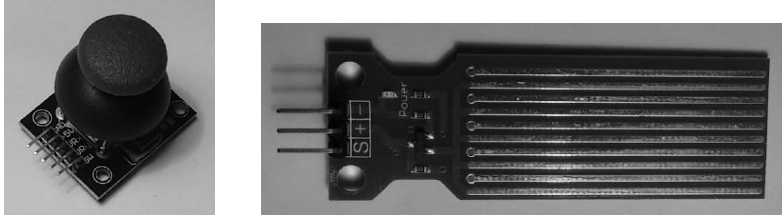


Figure 2.5 A joystick is shown on the left and a fluid-level resistive sensor on the right.

that is affected by the adsorbed gases. On the left of Figure 2.7 we illustrate the working principle. The heater is located under the SnO_2 active layer and powered by passing a current through it. The resistance, which depends on the concentration of the specific gas, can be measured between the terminals labeled 1 and 2. A device mounted on a small breakout board is shown on the right of Figure 2.7. The sensing area is located under the metallic hat that protects against the heated area, and potential explosive reactions on the hot surface should a “wrong” gas ignite.

After this short selection of resistance-based sensors we progress to discuss sensors that report a voltage directly.

2.1.2 Voltage-based sensors

An example of a sensor that directly produces a voltage at its output pin is the LM35 *temperature sensor*, which is a silicon-bandgap temperature sensor. The operating principle is based on passing known currents I_n with $n = 1, 2$ with current densities j_n across the base-emitter junctions of two bipolar transistors, and comparing their respective voltage drops $V_{BE,n}$. The voltage difference is proportional to the temperature. This is easily understandable by inverting the current–voltage curve for the diode of the base-emitter junction

$$j_n = A(T) \left[e^{(eV_{BE,n} - E_g)/kT} - 1 \right]$$

where $E_g = 1.2\text{V}$ is the bandgap energy of silicon, k is the Boltzmann constant, and T the absolute temperature in Kelvin. $A(T)$ is a device-specific constant with moderate temperature dependence. Assuming that both transistors are located on the same substrate and have the same temperature, we solve for two current densities j_1 and j_2 and obtain for the voltage difference $\Delta V_{BE} = V_{BE,1} - V_{BE,2} = kT/e \ln(j_1/j_2)$. In the LM35, the base-

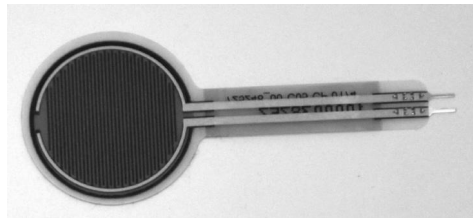


Figure 2.6 A strain gauge.

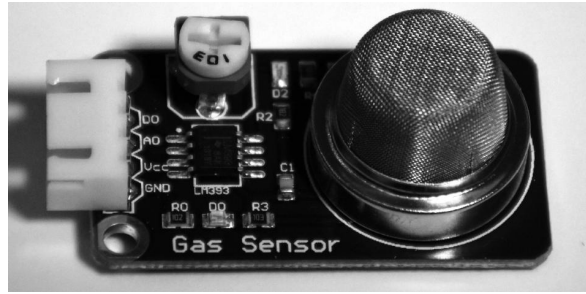
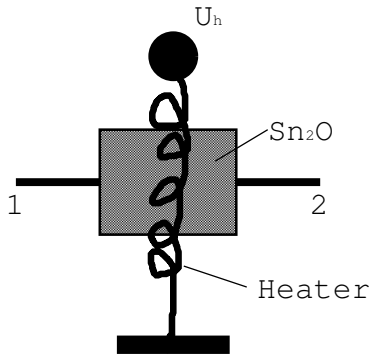


Figure 2.7 Schematic of an MQ-x gas detector (left) and a sensor mounted on a small breakout board (right).

emitter diodes of the two transistors have different areas such that the ratio of the areas determines the current densities, provided that the same macroscopic current passes through the two transistors. There are operational amplifiers on the same substrate to provide signal conditioning such that the LM35 produces an output voltage V_s that is related to the temperature T by $V_s = T/100$. Here V_s is measured in volts and the temperature in degrees Celsius, such that a temperature of 23°C results in a voltage of 0.23 V . The LM35 has three pins; one is connected to ground, one to the supply voltage, and the third one carries the voltage V_s that is proportional to the temperature. Note the polarity for connecting the LM35 in Figure 2.8. The flat surface is pointing to the wires on the left-hand side.

Thermocouples are temperature sensors that are based on the effects of temperature and temperature gradient on conductors made of different materials. Directly at the junction of the conductors, the Peltier effect causes a current that depends on the temperature. This happens at the points labeled by their respective temperatures T_1 and T_2 on the top left in Figure 2.9. On the wire segments a temperature gradient causes an additional current to flow, the Thomson effect. And finally, joining the two junctions and the wires causes a current to circulate, provided the loop is closed. This is called the Seebeck effect. If the loop is open, as shown at the top left of Figure 2.9, a voltage U develops at the end terminals as a consequence of the Peltier, Thomson, and Seebeck effects. In practice, one junction, say at T_1 , is held at known and constant temperature, for example, by immersing the

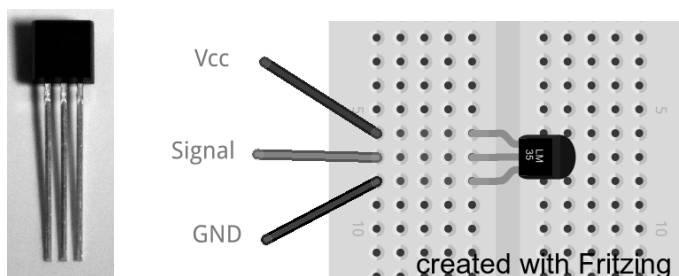


Figure 2.8 Image of an LM35 temperature sensor (left) and how to connect it (right).

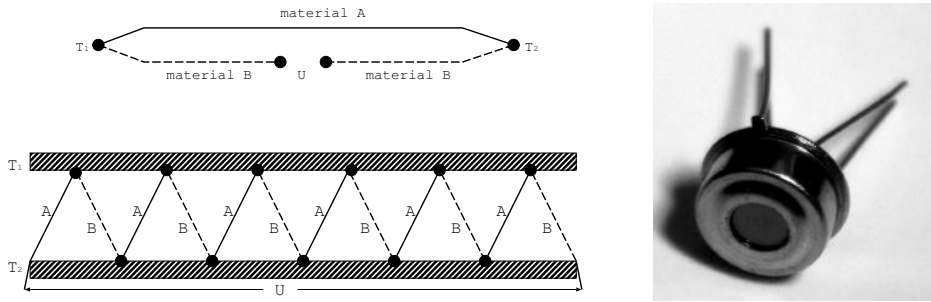


Figure 2.9 On the left we show a schematic of a thermocouple on the top and a thermopile on the bottom. On the right we show an image of an MLX90614 contact-free thermometer.

junction in ice water. Then the voltage U is related to the temperature difference $T_2 - T_1$ of the sensing end at T_2 and the reference temperature T_1 . The magnitude of the voltage generated depends on the combination of metals and is typically on the order of $50 \mu\text{V}/^\circ\text{C}$.

In a *thermopile* a number of wire segments of materials A and B are connected in series, as shown on the bottom left in Figure 2.9. This increases the sensitivity of the device to temperature differences. Thermopiles are often found in devices sensing heat and infrared radiation, such as thermal imaging devices or contact-free thermometers. The sensor used in the latter is shown on the right of Figure 2.9.

Some crystals and ceramics react to external stresses by producing a *piezoelectric* voltage between opposite sides of the material, as a consequence of rearranging charges within their crystal structure. The resulting voltages reach several kV and can be used to produce sparks in ignition circuits or in old-fashioned vinyl record players. There, a “crystal”-stylus is squeezed in the grooves of the record and the generated voltages are amplified and made audible as sound. In scientific applications, piezoelectric sensors are used to measure pressure or forces.

The speed of angular motion is easily sensed by a DC electrical motor that is operated backwards as a generator. Instead of applying a voltage to turn the axis of the motor, turning the axis induces an induction voltage in the motor coils that is proportional to the angular velocity. Attached to a propeller that is turned by either a flowing liquid or a gas, such a device can measure *flow rates*.

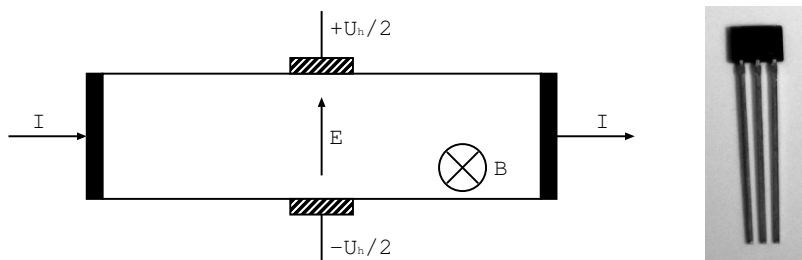


Figure 2.10 Schematic of a Hall sensor (left) and the A1324 sensor (right).

Hall sensors produce a voltage that is proportional to the *magnetic induction* B . Their mode of operation is explained in Figure 2.10 and is based on passing a known current I through a semiconductor. In the presence of a magnetic field, the Lorentz force deflects the charge carriers—electrons and holes—towards perpendicularly mounted electrodes (shaded). This creates a potential difference (a voltage) between the electrodes, which causes a transverse electric field that counteracts the deflection from the Lorentz force such that the following charge carriers can move towards the exit electrode undeflected. In equilibrium, the voltage difference between the upper and lower electrodes is proportional to the magnetic induction B and can be measured with a voltmeter. The A1324 Hall sensor, shown on the right in Figure 2.10, has signal-conditioning circuitry on board and only needs three pins for ground, supply voltage, and output voltage. The latter is proportional to the magnetic field, with a sensitivity of 50 mV/mT centered at 2.5 V when no field is present.

The ADXL335 is a three-axis integrated *acceleration* sensor based on micromachined structures on a silicon substrate where an inertial mass is suspended by springs [12]. The inertial mass is part of an assembly of capacitors driven by an AC voltage that is used to measure the imbalance of a capacitive voltage divider. As opposed to our simplified model with only one capacitor doublet, the real device uses a large number of interleaved capacitor doublets in order to increase the sensitivity. Figure 2.11 illustrates the principle of operation for a single direction. On the left in Figure 2.11 there is an AC-voltage generator that drives the light-grey capacitor plates. The dark-grey inertial mass is placed halfway between the driven plates, and in the absence of acceleration, the capacitances between the two light grey plates and the inertial mass are equal. An acceleration introduces an imbalance in the capacitances that affects the voltage level on the inertial mass. Comparing the phase and amplitude of that signal with that of the driving AC signal yields direction and magnitude of the acceleration. After some signal processing, it is then made available as U_{acc} on one of the output pins of the ADXL335 in the range from 0 to 3 V, such that the voltage is proportional to the acceleration in the range from -3g to +3g and is updated at a rate of about 100 times per second.

The SM-24 is another type of accelerometer called a *geophone*, and is shown in Figure 2.12. It is based on a coil connected to the housing by springs embedded in a magnetic field generated by permanent magnets that are attached to the housing. If the housing moves, the coil remains stationary due to its inertia, and a voltage is induced in the coil, which is proportional to the velocity and can be measured. The sensitivity is 28.8 V/(m/s) and the device operates in the range of 10-240 Hz.

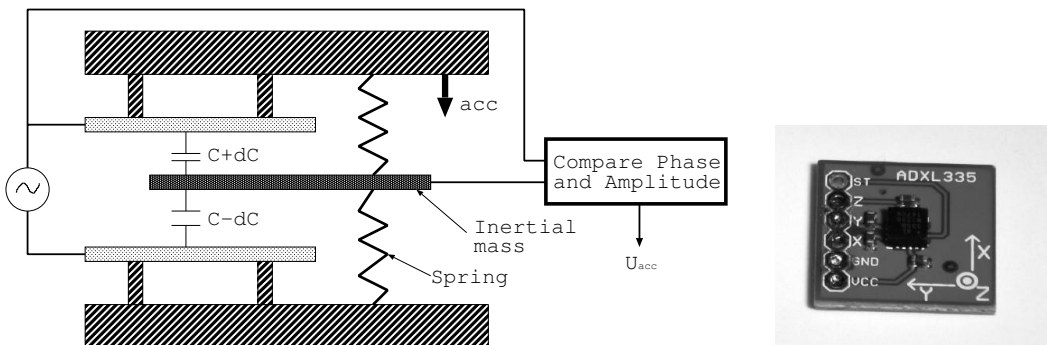


Figure 2.11 The operational principle of an ADXL accelerometer.



Figure 2.12 An SM-24 geophone with a diameter of about 30 mm.

Microphones convert sound to electrical signals and can be classified as sensors. Two major classes are on the market. *Dynamic microphones* operate similarly to the geophones. A coil, attached to a membrane, is excited by sound, moves in a magnetic field, and induces an induction voltage in the coil that is amplified and measured. In *electret microphones*, the membrane constitutes one electrode of a capacitor. If it moves, the capacitance changes, and the amount of charge stored on the capacitor is pushed on and off the capacitor and creates a current that is amplified and measured.

The electret microphone serves as a nice example with which to turn to current-based sensors.

2.1.3 Current-based sensors

The BPW34 is a *pin diode* that generates a current of up to 100 nA depending on the irradiance of up to mW/cm^2 . Pin diodes are similar to conventional diodes and consist of a semiconductor with a *pn* junction, which is created by doping semiconductor material, often silicon based, with a material that has either five valence electrons, in which case it becomes *n*-doped, or three valence electrons, in which case it becomes *p*-doped. Pin diodes, however,

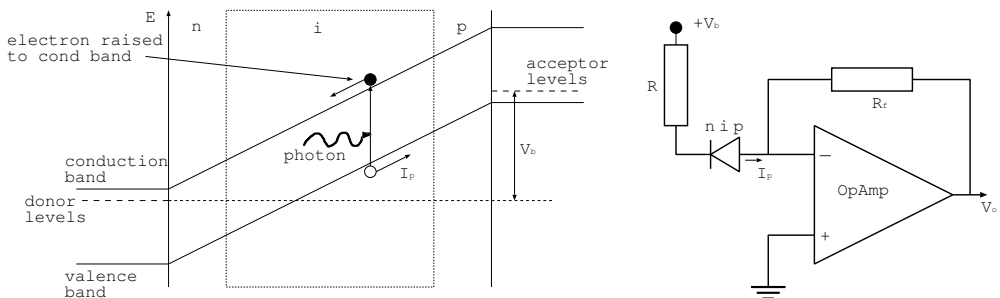


Figure 2.13 Energy-band diagram (left) and circuit (right) of a reverse-biased pin diode in photoconductive mode.

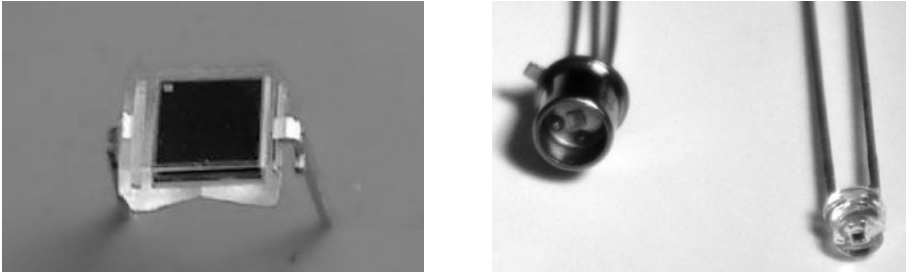


Figure 2.14 Image of a BPW34 pin diode on the left and two phototransistors on the right; an IR-sensitive BPX38 and an SFH3310, sensitive in the visible part of the spectrum.

have an additional layer of un-doped, intrinsically ('*i*') conducting silicon in order to increase the target area for the photons and provide a chance to produce additional charge carriers. One operational mode of the pin diode, called photoconductive, is illustrated on the left-hand side of Figure 2.13, which shows a simplified energy-band diagram of a reverse-biased diode. Note that by convention, the upwards energy axis corresponds to the potential energy of *electrons* that is lowest at the most positive voltage and that is found on the left-hand side. In the figure the cathode (*n* side) is therefore at higher voltage than the anode (*p* side) and results in all charge carriers being pulled out of the intermediate zone; electrons to the left and holes to the right. This results in the diode blocking any current flow. The extra layer of un-doped silicon provides extra potential charge carriers that act as targets for photons, having energy higher than the band-gap. These photons create electron-hole pairs by lifting electrons from the valence band into the conduction band, as indicated in Figure 2.13. The applied voltage, which is more positive on the left-hand side, causes the electrons to move to the left and the holes move to the right. Combined, this constitutes a current I_p . We mention in passing that ionizing radiation, such as high-energy photons and gamma rays as well as charged particles with high energies, create electron-hole pairs. This makes pin-diodes suitable as radiation detectors. The circuit on the right-hand side of Figure 2.13 shows an operational amplifier that converts the current I_p flowing towards its negative input port into a voltage $V_o = -R_f I_p$ on its output port. We will cover operational amplifiers in more detail in the coming sections. A BPW34 pin diode is shown on the left of Figure 2.14.

Phototransistors such as the BPX38 or SFH3310, both shown on the right of Figure 2.14, are similar to normal transistors, but their base-collector diode is a reverse-biased photo-diode, similar to the one described in the previous paragraph. It causes a current to flow as a consequence of impinging photons. The base-emitter diode is already forward biased and will ensure that the collector-emitter connection becomes conducting. Moreover, often there is a lens to increase the number of photons impinging onto the base terminal with the photosensitive area. Phototransistors sensitive to special spectral ranges such as infrared radiation, by suitably choosing their band gap, can be used as flame detectors.

The sensors in imaging applications such as cameras are *charge-coupled devices*, or CCDs, which are similar to a pin diode that is attached to a small capacitor, one for each pixel of the camera. Exposure to light transfers a small charge to the capacitor. The often large number of pixels are read out sequentially by transferring the charge from one capacitor to the one closer to the external readout port. A bucket chain to transfer water that is emptied

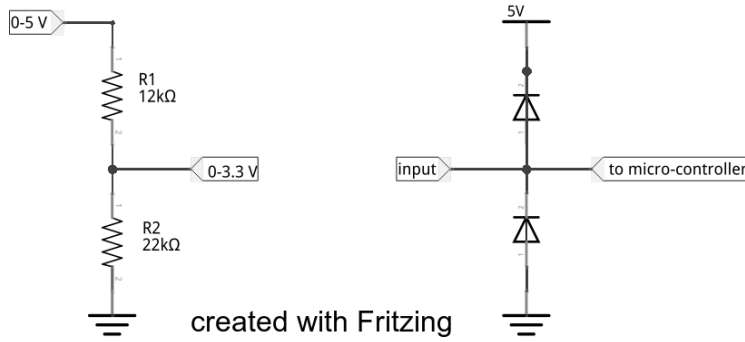


Figure 2.15 On the left we have a voltage divider to reduce the input voltage of 0-5 V to 0-3.3 V. The right circuit shows the use of clamping diodes to protect the input of the microcontroller to lie between ground and 5 V.

at the end point comes to mind. In a CCD, once the charge reaches the output port, it is passed through a resistor, where it creates a voltage drop that can be measured.

Solar cells operate in a similar fashion to photodiodes in photovoltaic mode where they provide a voltage to a load. They are, however, optimized to absorb as large a part of the spectrum as possible, and also to have a large absorbing area, in order to maximize the electric power available to the load.

After this brief overview of different analog sensors, we need to address how to prepare the signals such that they can be easily interfaced. This preparatory stage is referred to as signal conditioning.

2.2 SIGNAL CONDITIONING

The analog signals from the sensors can be too high or too low. They can be too noisy or otherwise inadequate to directly feed to an analog-to-digital converter (ADC) or a microcontroller. In many cases some signal conditioning is needed, and we will discuss some common methods in the following sections.

2.2.1 Voltage divider

In case the input voltage of the sensor exceeds the input range of what the microcontroller can handle, we have to reduce the voltage by a *voltage divider*, which consists of two fixed-value resistors. A typical example is to reduce the range from 0-5 V to 0-3.3 V, which is accomplished by a combining two resistors with a ratio of $R_1/(R_1 + R_2) = 3.3/5 = 0.66$. A close example is shown in Figure 2.15, with a 12 and a 22 kΩ resistor. The ratio is $22/34 = 0.65$, which is close to the desired ratio. Other combinations with larger and smaller values will work as well, as long as the ratio is correct. The sum of both resistors should not be too small because that will draw a larger current from the 0-5 V voltage source, and, depending on the internal resistance of the source, may affect the measured values.

In order to improve the measurement sensitivity, resistance-based sensors are often wired in a Wheatstone-bridge configuration. An example was shown on the right of Figure 2.1. Here the voltage divider on the left side of the breadboard provides half the supply voltage

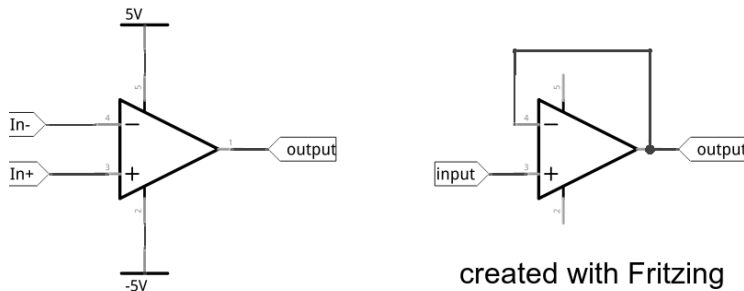


Figure 2.16 A bare operational amplifier shown on the left and shown wired as a line buffer on the right. We omitted the supply wires on the latter schematic.

at its center tap (the lower-signal wire) because the two resistors are equal. The voltage divider on the right-hand side is the same one, we encountered previously, with the upper-signal wire connected to the point between the upper resistor and the LDR. Normally one would choose the resistance of the right resistor to be in the middle of the range of interest of the LDR, such that voltage-difference between the signal wires is close to zero, indicating mid-range. In this way, depending on the light exposure, the voltage difference between the wires varies around zero and the sign will tell us whether the exposure is lower or higher than the expected mid-range value. Since we now deal with voltages that vary around zero, it is easier to amplify that voltage in order to increase the sensitivity. For example, when using a plain voltmeter, we can use a smaller voltage range.

In case we use a piezo-based sensor, the generated voltages can be much higher than is acceptable in the following circuit such as a microcontroller. In such a situation, *clamping diodes*, as shown on the right of Figure 2.15, are used. If the input voltage is between 0 and 5 V, the diodes are blocking, and the signal is passed on to the microcontroller. If, on the other hand, the input voltage exceeds 5 V plus the forward voltage drop, the diode starts conducting and shorts the input to the upper power rail of 5 V. If the input voltage is below 0 V, the lower diode starts conducting and shorts the input to ground. In either case, the voltage delivered to the microcontroller is limited to 0-5 V plus or minus the forward diode voltage drop. Many integrated circuits including microcontrollers have built-in clamping diodes. The Raspberry Pi, however, is a notable exception.

2.2.2 Amplifiers

Very small electrical signals usually need to be amplified to reach levels adequate for further processing. The standard device to achieve this is an *operational amplifier* or op-amp, shown on the left of Figure 2.16. There are two input ports on the left, one labeled “plus”, one labeled “minus”, and one output port. The latter delivers a voltage that depends on the difference between the two input ports. In an ideal op-amp, the amplification factor is infinite and we usually use some feedback mechanism to obtain a deterministic behavior, as discussed below. Normally, op-amps require both positive and negative supply voltage, even though sometimes it is possible to tie the negative supply rail to ground, in which case only unipolar signals can be amplified.

Before discussing different circuits, we need to describe three basic principles that characterize op-amps.

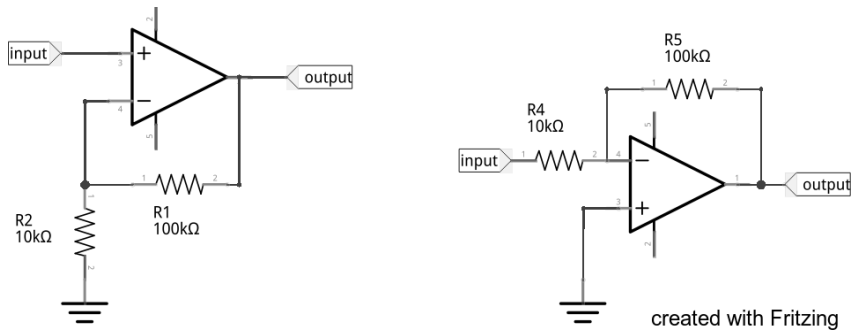


Figure 2.17 A non-inverting (left) and inverting (right) amplifier.

The input impedance of the input ports is “infinite”, which means that no current flows into the op-amp and it will not load the upstream circuitry.

The op-amp tries to reduce the difference between the input ports $V_+ - V_-$ to zero.

The amplification without feedback is quasi-infinite, and the output has a very low impedance and can provide high output currents.

These simple rules will help us to design and understand the following circuits, but consult [15] and [16] for a more extensive discussion.

We start by considering a *line buffer*, which is typically used as an impedance converter that transforms the output of a sensor with a high impedance to a low-impedance signal that is less susceptible to noise. We show the circuit on the right of Figure 2.16, where the output of the op-amp is fed back onto the negative input port. By using the second op-amp rule, we see that the op-amp tries to make the positive and negative input ports equal, but negative is tied to the output, which forces the output to follow the positive input. Moreover, by the first rule, the input impedance is high and no current (or at least very small current) is drawn from the sensor, while the output is low impedance and can provide a high current. Note that we omitted the connections to the supply rails on the right of Figure 2.16, and will do so henceforth in order not to clutter the schematics.

The line buffer is essentially an amplifier with unit amplification, but if we require a higher degree of amplification, we need to add two additional resistors to the circuit, and arrive at the left of Figure 2.17. The two resistors R_1 and R_2 constitute a voltage divider that forces the negative input voltage to be $V_- = V_{out}R_2/(R_1 + R_2)$. But, at the same time, the op-amp forces $V_- = V_+$, which, after solving for V_{out} , leads to $V_{out} = V_+(R_1 + R_2)/R_2$, where $(R_1 + R_2)/R_2 = 1 + R_1/R_2$ is the amplification factor. Since the output voltage V_{out} has the same sign as the input voltage V_+ on the positive input port, this configuration is called a non-inverting amplifier...

...which hints at the existence of an inverting amplifier, for which we show the schematics on the right of Figure 2.17. To calculate the amplification factor, we note that the input current I_{in} only flows through the input resistor R_4 and the feedback resistor R_5 because the input impedance of the op-amp is essentially infinite and no current flows into the input ports. But the equality of the current in the input and the feedback resistor implies $I = (V_{in} - V_-)/R_4 = (V_- - V_{out})/R_5$. Moreover, we observe that the positive input port is grounded, which forces the negative input port to be on ground potential as well. This implies $V_- = 0$, and the relation between input and output voltage becomes

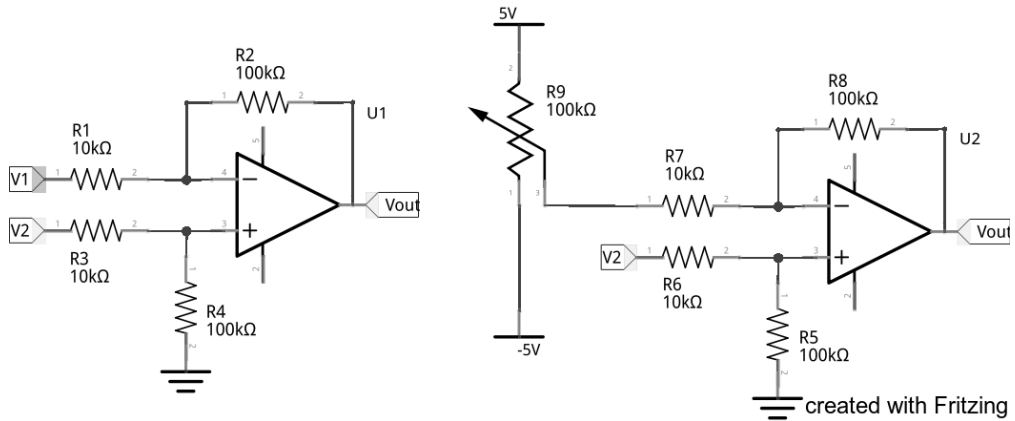


Figure 2.18 A difference amplifier (left) and the same circuit with adjustable V_1 (right) that is used to subtract the baseline.

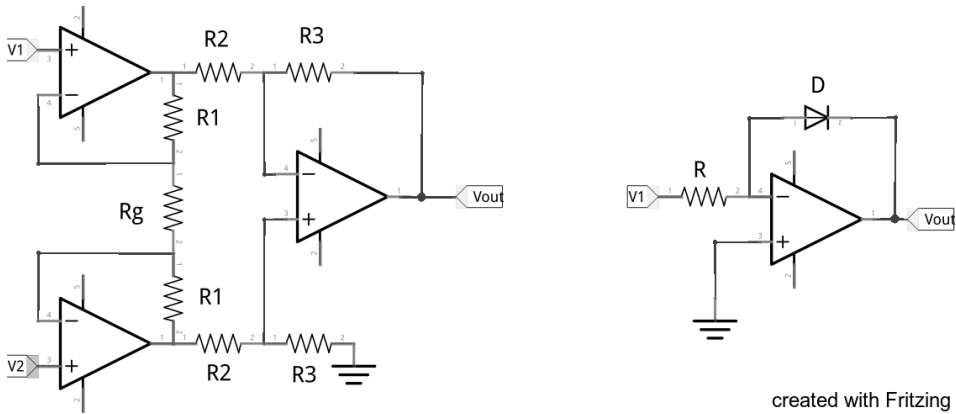


Figure 2.19 Instrumentation amplifier (left) and logarithmic amplifier (right).

$V_{out} = -V_{in}R_5/R_4$, where the negative sign indicates that the amplifier is inverting. Note that we can add several input resistors R_4 in parallel with one end connected to the negative input port. This allows us to add the currents passing through the parallel copies of R_4 , and we obtain a summing amplifier. The resistor values were simply chosen to be in a reasonable range. They need to be determined adequately for each application.

Sometimes the signal one wants to measure changes around a non-zero baseline. Examples are the Hall sensor A1324 from the previous section, where zero magnetic field produces 2.5 V and the magnetic field added or subtracted from that value depending on its polarity. In order to increase the resolution, we want to amplify not the signal, but the difference of the signal to the baseline. In other words, we need a circuit to subtract the baseline and amplify the difference. A differential amplifier as shown in Figure 2.18 accomplishes this feat, provided that $R_2 = R_4$ and $R_1 = R_2$. The output voltage V_{out} in that case is given by $V_{out} = (V_2 - V_1)R_2/R_1$. Adding a potentiometer that adjusts V_1 between the positive and negative supply rail subtracts it as baseline voltage.

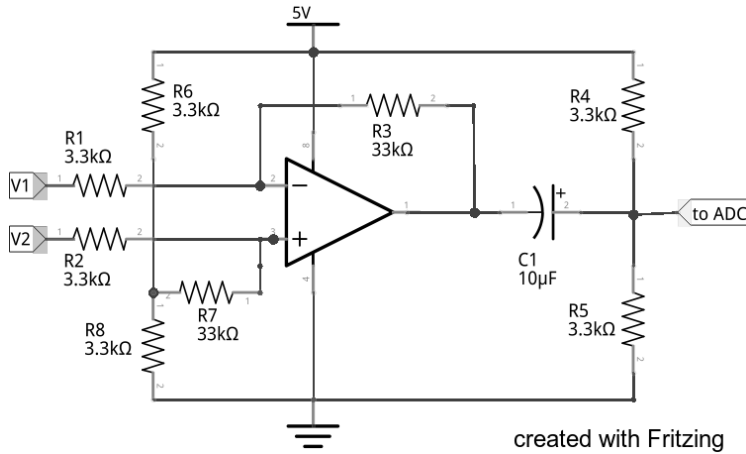


Figure 2.20 Amplifying a weak signal and shifting it to mid-range between the power rails.

A circuit similar to the one in Figure 2.18 but with line buffers on the inputs is called an *instrumentation amplifier*, and an example is shown on the left of Figure 2.19. Normally it is not necessary to build instrumentation amplifiers from discrete components. There are ready-made circuits available such as the AD620 or INA131.

Often, sensors produce small voltages that vary around zero, but the analog-to-digital converter (ADC) requires an input range of 0 to 5 V. We thus face the problem of amplifying a bipolar signal and changing the baseline level to mid range of the ADC. We show a circuit that amplifies by approximately a factor of 10 in Figure 2.20. The amplification is mainly determined by the ratio of the feedback resistor R_3 to the input resistors R_1 and R_2 . The voltage divider of R_6 and R_8 provides the mid-range offset voltage. The average level of the output voltage crucially depends on the tolerances of the resistors, and in order to place the level safely in mid range, we use the capacitor C_1 to first remove the DC level of the output signal before adjusting it properly to mid-range, with the voltage divider consisting of R_4 and R_5 .

In case we need to amplify input signals that vary over a huge range of values, a *logarithmic amplifier* such as the one on the right of Figure 2.19 is a useful circuit. It can be shown that the relation between input and output voltage is $V_{out} = -V_t \ln(V_1/I_s R)$, where V_t is the thermal voltage and I_s the saturation current of the diode. Swapping the diode and the resistor results in an exponential amplifier.

A close relative to the operational amplifier is the *comparator*. It can be visualized as an op-amp with very large, even infinite, amplification, whose output port saturates at the power rails. If the voltage at the positive input terminal of a comparator is larger than that on the negative input terminal, the output voltage is very close to the positive supply voltage. On the other hand, if the voltage on the positive input terminal is lower than that on the negative, the output is close to ground potential. In this way it translates the input voltages to a binary digital state. A comparator may therefore be considered as a 1-bit analog-to-digital converter, and we will see in a later section how it is used to extend the number of bits of the conversion. Some comparators have the threshold when switching from low to high output configured to be slightly higher than the threshold switching from the high output state back to the low one. This small hysteresis prevents the output from

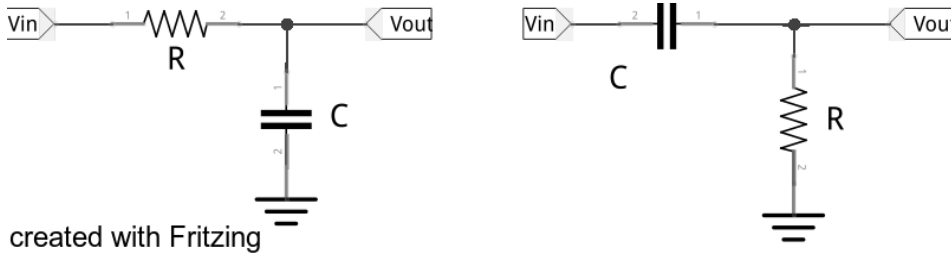


Figure 2.21 A simple low-pass (left) and high-pass (right) filter.

switching back and forth uncontrollably, should the voltages on the two input terminal be very close.

After the basics of signal amplification, we will now address the question of how to reduce noise in the circuits and decrease the sensitivity of a circuit in an undesired frequency range. This is the realm of filters.

2.2.3 Filters

The task of filters is to remove certain frequencies from an electrical signal such as all high frequencies, in which case the filter is called a low-pass filter. An example is a low-pass filter that removes “hissing” in audio-signals. The converse filter is a high-pass filter that removes low frequencies. An example is a anti-rumble filter found in old vinyl record players. If we know that the desired signal contains only a certain range of frequencies and we wish to remove all others, we use a band-pass filter. An example is the IF filter found in radios that are based on the super-heterodyne principle. And finally there are filters that remove only frequencies in a narrow band. They are called band-stop or notch filters. An example is a filter that removes the omnipresent 50 Hz or 60 Hz hum coming from the power grid.

We first consider a *low-pass filter*, which in the simplest incarnation is a frequency-dependent voltage divider made of a resistor with resistance R and a capacitor with capacitance C , as shown on the left of Figure 2.21. For a refresher of basic concepts circuit theory, such as impedance, please consult appendix A. In our circuit, the capacitor has an impedance $1/i\omega C$, which gets smaller with increasing frequency $\omega = 2\pi f$. Intuitively, the higher frequencies are shorted to ground. If we build a voltage divider as shown on the left of Figure 2.21, the output voltage V_{out} is given by $V_{out} = (V_{in}/i\omega C)/(R + 1/i\omega C) = V_{in}/(1 + i\omega RC)$ and we see that it is attenuated with increasing frequency ω and, conversely, the low frequencies are unaffected, hence the name low-pass filter. The frequency where the signal amplitude is attenuated by a factor $\sqrt{2}$ is given by $\omega_c = 1/RC$, and the imaginary unit indicates that there is a phase shift between input and output voltage that depends on the frequency. Equivalently, a low-pass filter can be constructed with an inductor with impedance $i\omega L$ and resistor R , but in many operational situations the inductances have values that are difficult to find; therefore filters are usually constructed from resistors and capacitors. The frequency dependence of the filter asymptotically behaves as the first inverse power of ω , and the filter is called a single-pole filter. Cascading two such filters results in two-pole low-pass filters that exhibit a steeper frequency-dependence of $1/\omega^2$.

Swapping the resistor and capacitor in the low-pass filter results in the *high-pass* filter shown on the right of Figure 2.21, which has a frequency dependence of $i\omega RC/(1 + i\omega RC)$

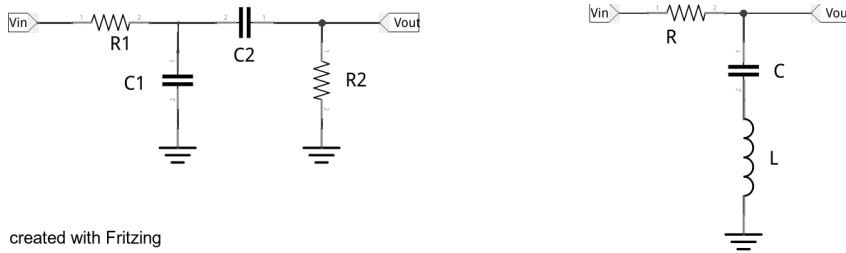


Figure 2.22 A simple band-pass (left) and band-stop (right) filter.

that attenuates low frequencies due to the factor ω in the numerator and approaches unity as $\omega \gg \omega_c = 1/RC$.

If we want to filter out everything except a small range of frequencies, we need a *band-pass* filter that we can most easily construct by a combination of a low- and a high-pass filter, as shown on the left of Figure 2.22. Here we have to ensure that the cutoff-frequency $1/R_1C_1$ of the initial low-pass filter is higher than that of the high-pass filter, given by $1/R_2C_2$.

In case we need to reject a certain perturbing frequency, we implement a *band-stop* or notch-filter, which is shown on the right of Figure 2.22. The combination of inductance L and capacitance C has a resonance at the frequency $\omega_c^2 = 1/LC$ where their series resistance vanishes, and a signal close to this frequency is shorted to ground and not passed on to the output.

The filters discussed so far are passive filters that only depend on resistors, capacitors, and inductors, and will only be able to attenuate unwanted frequencies. Sometimes, however, we need to combine amplification of a signal with filtering. The simplest solution is placing an operational amplifier immediately after the filter. Alternatively, one may include a capacitor in the feedback branch of the amplifier, as shown in Figure 2.23.

There is a huge amount of expertise in designing filters, both passive and active, documented in the literature. For a practical overview, see [17].

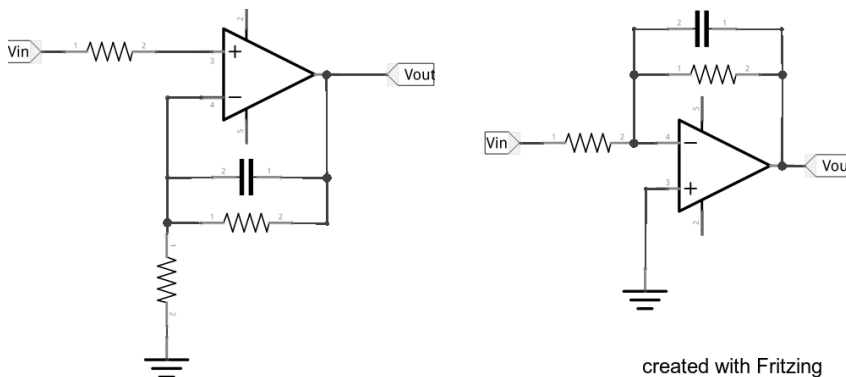


Figure 2.23 An active non-inverting (left) and inverting (right) low-pass filter.

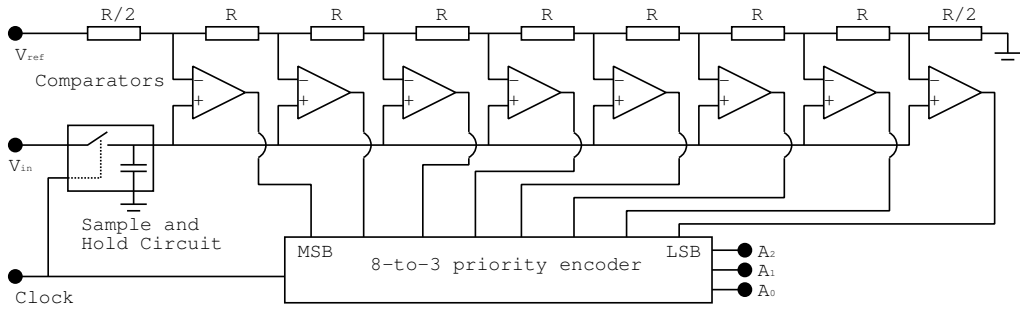


Figure 2.24 The operating principle of a 3-bit flash ADC.

2.2.4 Analog-to-digital conversion

Normally we want to convert analog signals to a digital format in order to process them on a computer. The device that achieves this feat is an analog-to-digital converter (ADC), which can be conceptually understood as a very rapidly measuring voltmeter. It consists of a sample-and-hold circuit that holds the voltage constant for a short time while it is measured and digitized. Several digitization methods are employed, and we will discuss a few different types.

In a *flash ADC* the voltmeter is based on large number of comparators that compare the voltage against a sequence of voltages derived from a resistor ladder. An additional circuit encodes the output from the comparators into a binary representation. Since all comparators operate in parallel, flash ADCs have the highest conversion rate, in excess of 10^9 samples per second. The high conversion rate comes at a price, though, because for a resolution of n bits, 2^n comparators are required. We illustrate the operating principle in Figure 2.24 for a 3-bit flash ADC. On the left there are inputs for the reference voltage V_{ref} , the voltage to be converted V_{in} , and a clock signal. First, the input voltage is held constant for the duration of the conversion in a sample-and-hold circuit, which consists of a switch and a small holding capacitor. The voltages to compare to are produced by a resistive voltage divider, shown on the top. The voltages from the divider are routed to the negative input terminal of the comparators. All of the latter then compare the sampled-and-held input voltage, simultaneously. The outputs of the comparators are connected to the inputs of a priority encoder that translates this into a 3-bit binary representation at the output pins A_0 , A_1 , and A_2 . The conversion process and synchronous operation of sampling and encoding is coordinated with an externally supplied clock signal. Flash ADCs are normally available with resolution of 8 bits or less, and are relatively expensive. To sample at lower rates, we can use less expensive ADCs such as those discussed in the following paragraph.

One of the very common types of ADCs, to sample a larger number of bits at a lower rate and lower price are *successive approximation ADCs*. They replace the resistor ladder and the large number of comparators with a single comparator and a digital-to-analog converter (DAC) to dynamically adjust the reference voltage to compare to. We discuss the operational principle of DACs in Section 3.13, but here we just mention that they are circuits that translate a digital word with n bits to an analog voltage, and they can do this very quickly. They play a central role in the operation of a successive approximation ADC whose operating mechanism is illustrated in Figure 2.25, where the input voltage is held constant during the conversion in a sample-and-hold circuit. Then the voltage is passed to a comparator whose negative input terminal is determined by the output voltage of the DAC.

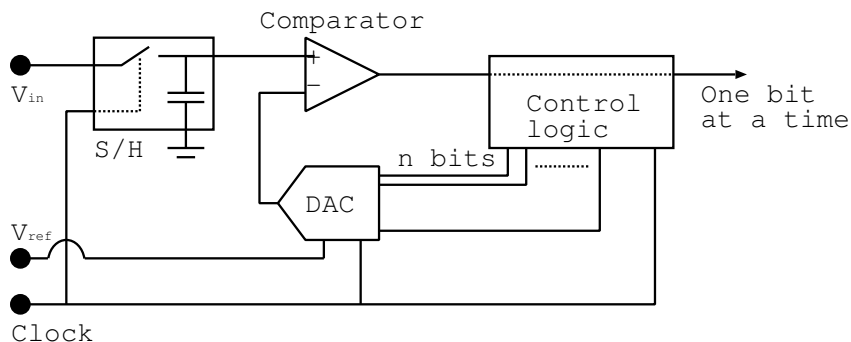


Figure 2.25 The operating principle of a successive approximation ADC.

In the first comparison, the n -bit input word is set to the $0x1000\dots$ which results in half the reference voltage V_{ref} , where we prepend ‘0x’ to identify the hexadecimal representation of a number. If the input voltage V_{in} is smaller, the output of the comparator is at ground level, or logically a low state. The first, most significant bit (MSB), is therefore “0”. The control logic then sets the DAC to $0x0100\dots$ and compares again on the next clock cycle. If V_{in} is larger than $(1/4)V_{ref}$, the comparator output will go to high-voltage level, and the next bit to pass to the output is therefore “1,” and the DAC receives the input $0x0110\dots$ to compare the next step in the bisection sequence. If this is repeated n times the ADC clocks out all bits of the conversion from MSB to least significant bit (LSB). We see that in this case we trade conversion speed for a simpler hardware. Most ADCs we use later in this book are of the successive approximation type. We note in passing that often a multiplexer switch is placed before the sample-and-hold circuitry, which allows us to select different input voltages with a single ADC shared among the different channels. Only one channel can be converted at the highest conversion rate.

It is possible to find a compromise between conversion speed and hardware complexity by combining, for example, two 4-bit flash ADCs, one additional DAC, and advanced control logic to a so-called pipelined flash ADC. They can sustain a continuous conversion-rate determined by the 4-bit ADC. Even though each conversion requires two clock cycles, the first ADC can already start converting the next sample while the second ADC still works

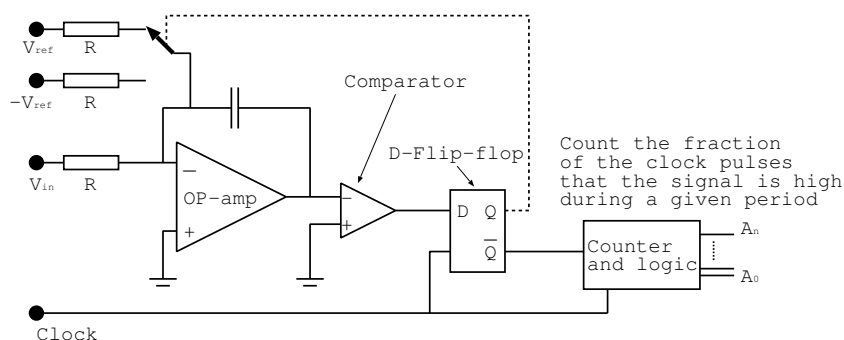


Figure 2.26 The operating principle of a delta-sigma ADC.

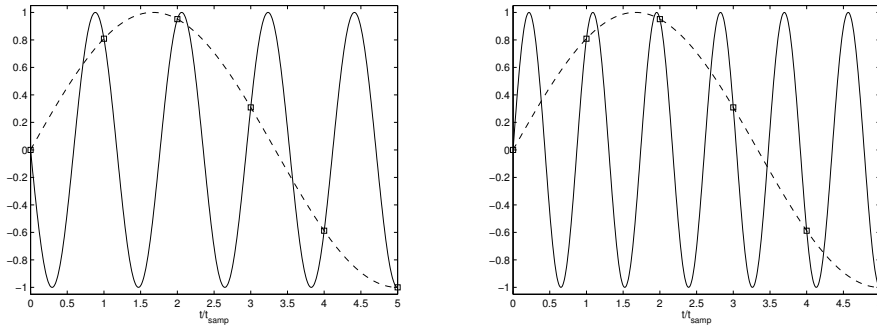


Figure 2.27 Sampling a signal in different Nyquist zones. The dashed line displays a frequency of $0.15f_s$ and the solid lines show signals with frequency $(1 - 0.15)f_s$ on the left and $(1 + 0.15)f_s$ on the right. Note that at the times when the signal is sampled (indicated by boxes), the signals are indistinguishable.

on the four least significant bits of the previous sample. Most high-speed ADCs for radio-frequency applications with more than 8 bits use this method.

Instead of a very high conversion rate, it is often desirable to achieve a higher resolution; in other words, more bits, albeit at a low conversion rate. Devices that fulfill this requirement are *delta-sigma ADCs* whose simplified operating principle is illustrated in Figure 2.26. It is based on adding quantized current pulses to the negative input terminal of the op-amp such that its output is forced to zero voltage. Note that this terminal is a virtual ground, because the positive input terminal is grounded, and operational amplifiers always strive to make their input voltages equal. The op-amp is configured as an integrator and sums the currents over time. Forcing the accumulated charge on the capacitor to zero is achieved by the feedback, shown as a dashed line from the non-inverting output of the flip-flop to the switch that either adds a positive or negative current of magnitude V_{ref}/R into the inverting input terminal of the op-amp. The purpose of the flip-flop is to produce well-defined time-steps, equal to the clock-frequency, for the injected current pulses. All we have to do in the end is to count the number of clock cycles when the output is high and divide by the total number of elapsed cycles. This results in a digitized word with the number of bits given by the time we chose to average. And that time can be quite long. Assume that we use a clock of 10 MHz and sample for 0.1 s such that 10^6 clock pulses happen, which results in a resolution of 20 bits because $10^6 \approx 2^{20}$. Delta-sigma ADCs derive their name because the small quantized difference (delta) current are summed (sigma), together with the current from the input voltage V_{in} , in the integrator. The high resolution makes delta-sigma ADC a good choice to measure the often small voltages in Wheatstone bridges. We need to keep in mind, however, that this increased resolution comes at the price of a rather low conversion rate; often only a few tens of conversions per second are possible.

Despite having a large number of bits, ADCs add noise to the measurements, because they cannot measure voltage differences smaller than that corresponding to the least significant bit. Thus, they introduce a *quantization error*, which could be reduced by choosing an ADC with a larger number of bits. But even then the quantization causes a slightly inaccurate representation of the signal amplitudes.

Sampling the signal at discrete instances in time causes a second potential mis-

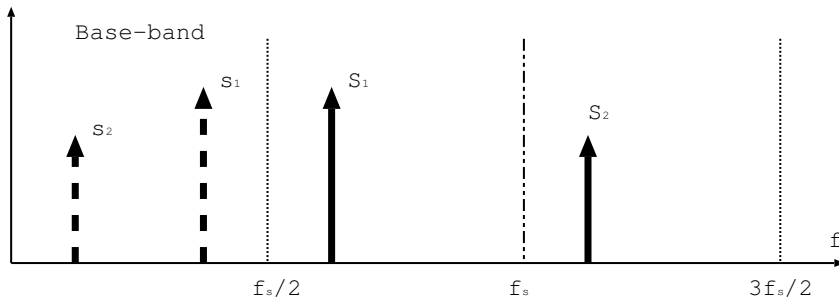


Figure 2.28 The dashed signals s_1 and s_2 are the images aliased into the base band of the original signals S_1 and S_2 in higher Nyquist zones.

representation, because signals of frequencies f higher than half the sampling frequency $f_s/2$, also called the Nyquist frequency, cannot be distinguished from signals with frequency $nf_s - f$ or $nf_s + f$. The origin of this ambiguity is illustrated in Figure 2.27. On the left we display a sinusoidal signal with frequency $f = 0.15f_s$ (dashed) and $f = (1 - 0.15)f_s$ (solid), and on the right the solid line is a sine with frequency $f = (1 + 0.15)f_s$. We see that at the sampling instances, indicated by the squares, the curves have the same values, which makes them indistinguishable if only sampled at rate f_s . In the frequency domain, shown in Figure 2.28, we find that a signal S_1 that lies between the Nyquist boundary and $f_s/2$ and f_s is recorded by the ADC, sampling at rate f_s at frequency s_1 , which is S_1 mirrored at the Nyquist boundary. The signal S_2 that lies above f_2 is observed as signal s_2 . The zone between zero and $f_s/2$ is commonly called the base band or first Nyquist zone, and the appearance of higher-frequency signals in the base band is called *aliasing*. Since the original frequency of the aliased signals is unknown, they are often considered to be noise. A simple way to prevent aliasing is to use analog low-pass filters with cutoff frequencies below the Nyquist frequency before passing the signals to the ADC. Filters such as those discussed in Section 2.2.3 are often sufficient.

Note that ADCs are often built into microcontrollers and sensors with digital interfaces, such as those discussed in later sections, and usually have an ADC built in. After this treatment of the ADC, the workhorse of digital data-acquisition systems, we need to look at the task of providing power to our circuits.

2.2.5 Supply voltage

Of course, our sensors and also the microcontroller require electric power to operate, and that is normally supplied by a power source. A common type of power supplies use a transformer to step down the 220 V or 120 V AC voltage from the wall to a commonly used voltage range of around 5 to 20 V. This depends on the rating and winding ratio of primary to secondary winding of the transformer. Since most electronics circuits require DC voltages, we need to rectify the AC voltage from the secondary winding of the transformer. The simplest way to do this is to use a single rectifying diode with adequate power rating, as shown on the left of Figure 2.29. The diode only lets voltages pass in one direction, and effectively cuts off the negative half-cycle of the AC voltage. The ratio of the voltages before and after the diode are shown on the plot below. The output in that case is very bumpy, and can be smoothed somewhat by adding a large electrolytic capacitor with a few 100s of

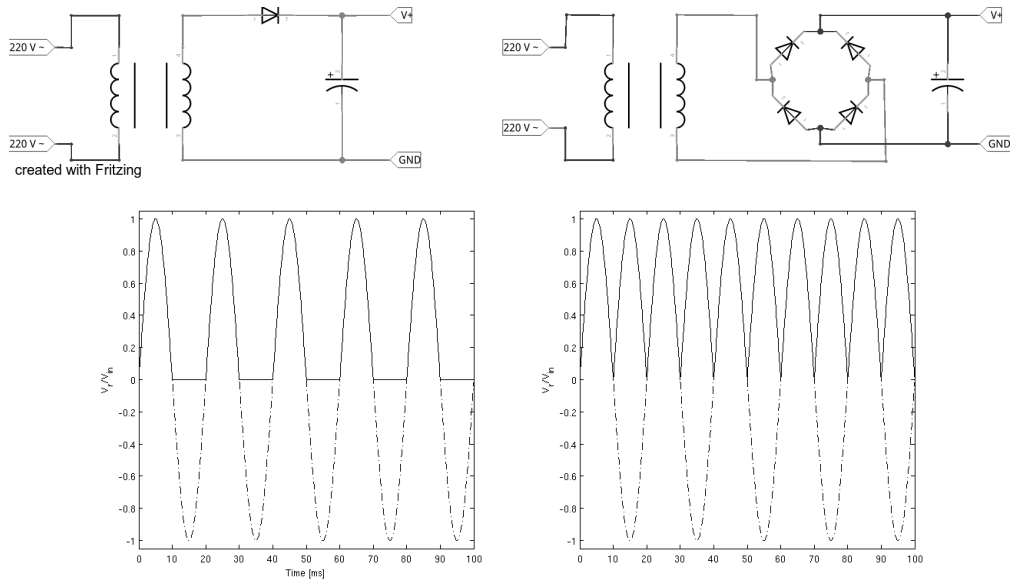


Figure 2.29 Schematics of very simple power supply circuits.

μF . It is charged during the positive half-cycle and is discharged during the negative, where it supplies charge to the powered circuit. The more current the load draws, the bigger the capacitor needs to be. Note also that the power in the negative half-cycle is dissipated in the diodes, which heats them up and may make a heat sink for the diodes necessary.

A better solution is to use a bridge-rectifier circuit made of four diodes with adequate power rating, as shown on the right side of Figure 2.29. The diodes are placed in such a way that during the positive half-cycle, one set of diodes conducts, and during the other half-cycle, the other set of diodes conducts. Essentially the circuit works by transforming the sine of the AC to the absolute value of the sine, as can be seen on the plot below the circuit where we show the ratio of the voltages before and after the bridge rectifier. We still need the smoothing capacitor, but for the same capacitance, we obtain much less ripple on the output voltage. Note that many of the very common wall-plug power supplies contain a circuit similar to the one shown on the right of Figure 2.29.

But the ripple of the output voltage can be reduced further by using a linear voltage regulator based on a temperature-stabilized voltage source with a bandgap voltage reference. In such a circuit, the known temperature gradient of the output voltage from a configuration with two transistors of different size is balanced by the opposite temperature gradient of suitably chosen resistors. We discussed a similar configuration in Section 2.1.2 when discussing the LM35 temperature sensor. The result is a very stable output voltage on the order of 1.25 V, near the bandgap of silicon. The commercially available linear voltage regulators usually have three pins for input IN, output OUT, and adjustment ADJ, and they have internal circuitry that forces the voltage on the output pin to the bandgap reference voltage of 1.25 V. It is straightforward to use an external voltage divider with suitably chosen ratio to select any output voltage up to some maximum current specified in the datasheet. On the left of Figure 2.30, we show how a voltage divider with two resistors is used to set the desired output voltage. The larger capacitors have capacitances of a few μF

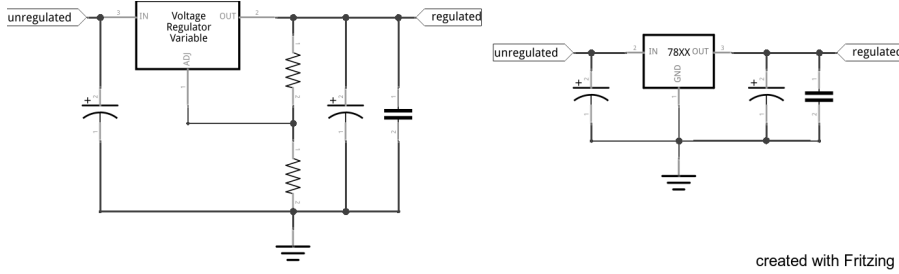


Figure 2.30 Variable-voltage (left) and fixed-voltage (right) regulator circuits.

and guarantee stable operation of the circuit. The smaller capacitors of typically 100 nF are used to absorb any high frequency glitches from the load. A standard example of such a variable voltage regulator is the LM317.

For many standard operating voltages, such as 3.3, 5, 9, and 12 V, ready-made linear voltage regulators are available. They do not require the voltage divider, and most have the numbers 7803, 7805, 7809, or 7812 in their name. The last two digits specify the output voltage. The same series with 79xx exist for negative output voltages. These small circuits are very handy for obtaining well-regulated output voltages. Also, if a power supply already gives 5 V, but 3.3 V are required for a sub-circuit, we use a 7803 or similar. Normally the voltage regulators require about 2 V higher input voltage than their specified output voltage, unless we choose a special *low-dropout voltage* (LDO) regulator, which only requires about 0.5 V overhead. The 3.3 V regulator MCP1700 is a member of this category.

Of course, we can also power an electronic circuit from batteries or rechargeable batteries. Of special interest are lithium-ion or lithium-polymer rechargeable, because they have the ability to provide very high currents to a circuit, which is especially important for devices that require large currents, such as WLAN circuits or motors. The lithium-ion cells provide voltages in multiples of nominally 3.7 V, and require special dis- and recharging circuitry, because their high intrinsic energy density requires special care to prevent under- and over-charging, as well as physically damaging the battery.

For electronic circuits that are used near a computer, we can use power drawn from the USB port. Many RS-232-to-serial converter circuits provide up to about 500 mA, which is often sufficient for small circuits. To make circuits very portable in the field, we can also use solar cells and so-called supercapacitors as temporary power source. In the laboratory, we have, of course, bench power supplies with adjustable output voltage, and current-limiting circuitry which prevents excessive currents that might damage our electronics.

After discussing analog sensors, analog-to-digital-converters, and power supplies, we now turn to sensors that directly produce digital signals.

2.3 DIGITAL SENSORS

We now address sensors that do not require an external ADC, but report their measurement values directly to the microcontroller in digital form. Some sensors already have the ADC built in, and others do not need one. We start with the latter, of which the most prominent examples are buttons and switches.

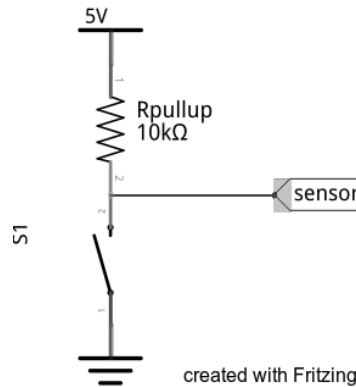


Figure 2.31 Connecting a switch or button with a pull-up resistor.

2.3.1 Buttons and switches

The simplest digital sensor is certainly a *switch* that is either closed or open, or a *button* where the open or close state is only activated temporarily. We use these terms interchangeably. Our task is to sense their state in a reliable way, and this is normally done with a pull-up resistor that is connected to the supply voltage in the way shown in Figure 2.31. In this way the sensing pin on the microcontroller can reliably detect the supply voltage. Only if the switch S1 is pressed does the voltage on the pin drop to zero or ground level. Only a small current, determined by the magnitude of the resistor, flows when the switch is closed. The actual value of the resistor is uncritical, but values around 10 to 30 kΩ are usually reasonable. Without the pull-up resistor, the voltage potential on the pin is undefined when the switch is open, and determined by stray capacitances in the system. So, the recommendation is to always use a pull-up resistor when sensing the state of a switch. Note that swapping the position of the resistor and the switch, the resistor functions as a pull-down resistor, and the sensing level is zero unless the switch is closed.

We need to point out that mechanical switches have the undesirable characteristic of bouncing. Closing the switch is often accompanied by a fast on-off sequence. Fast microcontrollers or other computers act so rapidly that they are easily fooled by sensing multiple switch-closures instead of a single one. Sometimes a small time-delay is introduced after the first switch closure is detected. In this way, only a single closing event is accounted for, and others during the short timeout period are ignored. Optionally, an analog debouncer, which is a simple low-pass filter, as discussed in Section 2.2.3, may be used.

There are a number of other sensors that act just like a switch and can be sensed in the same way. An example is a *reed switch*, which is sensitive to magnetic fields and closes a switch if some field level is exceeded. A further example is a *tilt switch*, which senses the position of a small conducting sphere that horizontally rolls back and forth, and closes a contact when it bumps into one of the extreme positions.

A variation on the theme of switches are *rotary encoders*, which use two switches, A and B, that open and close periodically as an axis is turned. The direction of the rotation can be detected, because each switch produces a regular pattern of on and off, but the timing of the two switches is shifted by 1/4 of the period length. Essentially, one switch produces a cosine-like pattern and the other a sine-like pattern. From knowing both, we can determine

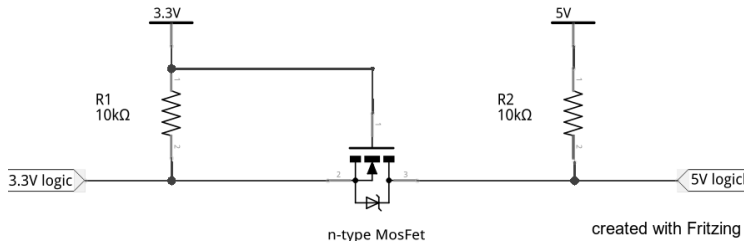


Figure 2.32 Level shifter circuitry using a n-type MOSFET. The source of the MOSFET is connected to the 3.3 V logic and the drain to the 5 V logic.

the direction of rotation by observing whether switch A leads switch B or vice versa. But the fundamental sensing process is based on the same mechanism as shown in Figure 2.31.

2.3.2 On/off devices

A number of sensors provide a *voltage level* to inform the microcontroller about their state or change of state. They can be thought of as a switch with a built-in pull-up resistor and can be sensed in the same way.

A problem can occur if the operating voltage level of the sensor and the microcontroller do not agree. Nowadays many sensors operate on levels of 2.5 to 3.3 V, and microcontrollers on levels from 2.5 to 5 V. Sensing higher external voltages such as those used in cars (12 V) or industrial control applications (24 or 48 V) requires some adjustment of the voltage level, to prevent damaging either the sensor or the microcontroller. There are level-changing chips available, such as the 74LVC245, but in many cases a simple voltage divider with two resistors is adequate. Yet, it only works if there is signal flowing from the high to the low-voltage side. In case a bidirectional signal flow is necessary, such as on the data line of the I2C bus, the solution shown in Figure 2.32, based on an n-type MOSFET transistor, is easy to implement. In the first case, when both logic signals are high, the MOSFET is not conducting, because the voltage difference between gate and source is close to zero. In the second case, if the 3.3 V logic is controlling and the signal is pulled low, the difference between gate and source is positive and the MOSFET conducts, such that even the 5 V logic level is pulled low. In the third case, when the 5 V logic is controlling and the 5 V logic signal is pulled low, the built-in diode (visible in the schematics) conducts, and causes the voltage of the source to drop to about 0.7 V. At this point the gate source voltage drop is sufficiently large to fully cause the MOSFET to conduct, which also pulls the 3.3 V logic level low.

But let us return to the sensors. A prominent device that reports its state through a changing voltage level is a *PIR proximity sensor*, shown on the right of Figure 2.33. It senses the change in the infrared radiation level, which announces the presence of living beings. The sensors are based on collecting the incident infrared radiation with a Fresnel lens, which is the dome visible in Figure 2.33, on a pyroelectric sensor. The lens is often made of polyethylene, a material chosen for its low absorption of IR radiation. One side of the sensor is consequently warmed up and expands, which causes buckling of the piezo- or pyroelectric material, often a polymer film. Two effects contribute to the voltage between the upper and lower plates. First, the buckling strains the material and causes a piezoelectric voltage. Second, heat flows from the hot to the cold side and adds a pyroelectric voltage.

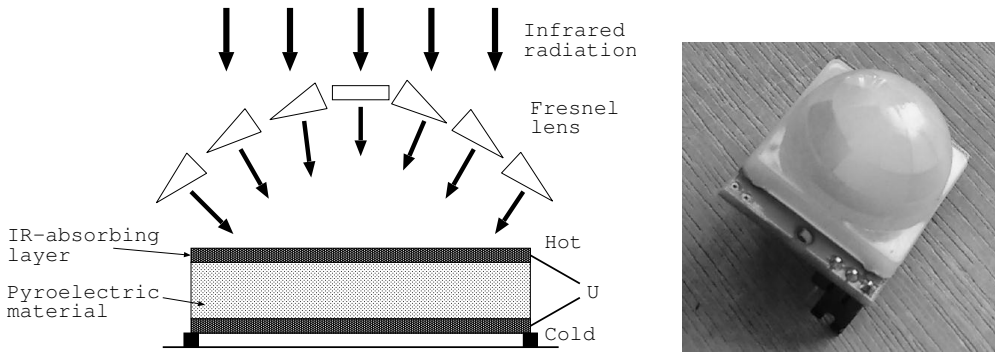


Figure 2.33 Schematic view of a PIR sensor (left) and the hardware (right).

The schematic setup of the device is illustrated on the left of Figure 2.33. The tiny voltage that is generated is subsequently amplified and exposed to the surrounding electronics on an output pin that goes from the low to the high voltage level once it is triggered by the presence of a person, and stays there for a programmable (usually by a small potentiometer) amount of time.

Several sensors report their measurement value encoded as a voltage pulse, with the duration of the pulse proportional to the value. We therefore need to measure the duration of that pulse with adequate accuracy. One device that employs this mode is an HC-SR04 *distance sensor*, shown in Figure 2.34, which operates like a sonar. It emits a short ultrasonic (40 kHz) sound burst and records the duration until the echo arrives, which is the round-trip time Δt of the sound burst. The distance L is given by this duration, and the speed of sound (approximately $v = 340$ m/s) by $L = \Delta t/2v$, or, in convenient units, $L[\text{cm}] = 0.017\Delta t[\mu\text{s}]$. A pin on the device goes high when the pulse is emitted, and returns to low once the echo arrives or some specified timeout expires. Somewhat more advanced models using the same method, but having a larger range, are the LV-EZx sensors. They support other modes of reporting the distance as well, such as an analog voltage proportional to the duration, or direct reporting as RS-232 signals, but more on that below.

Another sensor, often found in modern consumer electronics, detects the *infrared sig-*

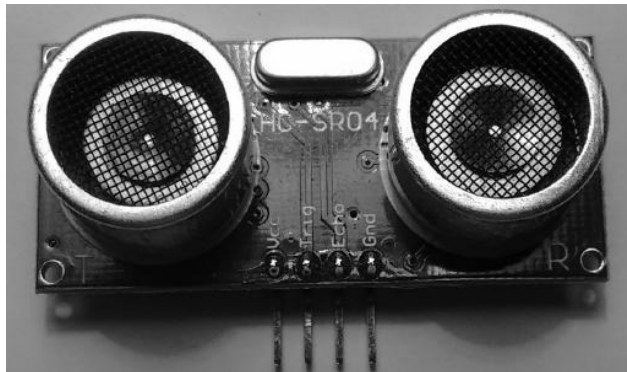


Figure 2.34 HC-SR04 distance sensor.

nals from the remote control. It is not strictly a scientific sensor, but still an interesting device that works remarkably reliably, because it rejects disturbing environmental effects and changes the channel on the TV only when a button on the remote is pressed. Typically they operate at a wavelength of 940 nm, which matches the light-emitting diode on the remote, and they have a built-in optical band-pass filter that lets only that wavelength pass. Then the signals are modulated by 38 kHz carrier frequency, which is demodulated on the sensor, such that only the base-band signal is reported on the output pin.

Now we turn to sensors that directly report their measurement values in digital form, and start by discussing I2C devices.

2.3.3 I2C devices

A large number of sensors have some logic built in and support a high-level communication protocol. An example is the *I2C* protocol operating on the I2C bus. The physical connection to devices supporting I2C only needs four wires: Ground, supply voltage Vcc, clock SCL, and data SDA. The latter two require a pull-up resistor, which is often already included in the microcontroller that also serves as the I2C busmaster to orchestrate the communication. It configures the sensor, initiates a measurement, and then reads data from the sensor. Physically, the communication is based on a synchronous serial protocol, where the data line is sampled every time the clock line changes from a high level to a low level. The protocol is standardized and we will not go into details, but mention that the I2C devices and also I2C sensors have a number of registers internal that can be written to in order to configure the sensor, or read from in order to retrieve sensor data. The communication is entirely based on exchanging digital signals, and, as mentioned before, coordinated by the busmaster which normally is a microcontroller. Several devices can share the same SDA and SCL lines, because each device has its own address and responds only to those messages intended for it by specifying the device address.

The BMP085, as well as the later versions BMP180 and BMP280, are examples of *barometric pressure* sensors. They are based on measuring the strain caused by the deformation of a membrane that separates an evacuated test-volume and the outside air pressure [12] with a piezoresistive strain gauge, as illustrated in Figure 2.35. The entire assembly is directly built into a silicon substrate, and the strain gauge is created by suitable doping of a small section. A temperature sensor is included because the semiconductor-based strain gauge is temperature dependent. Further signal conditioning and processing circuitry is in-

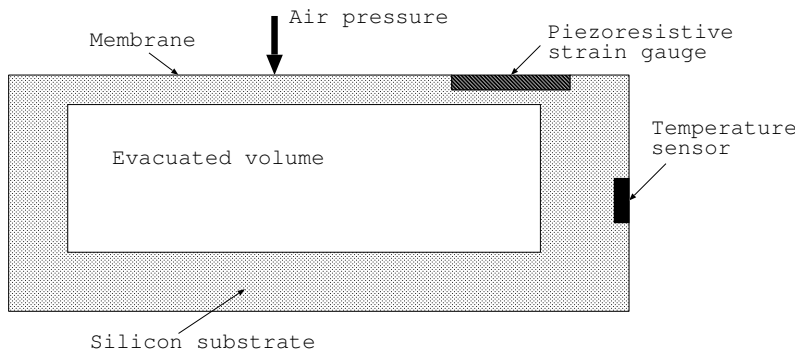


Figure 2.35 Illustration of the operational principle of a barometric pressure sensor.

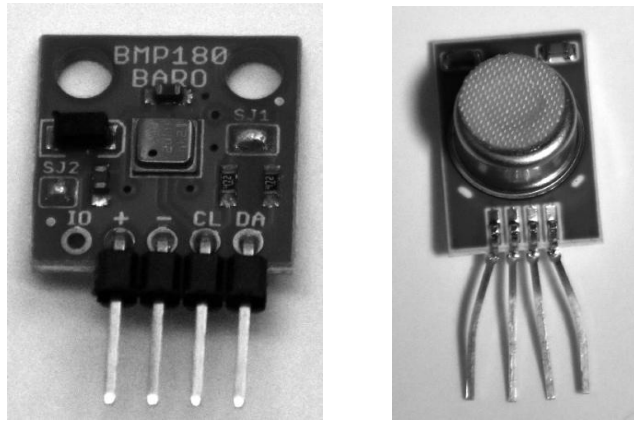


Figure 2.36 A BMP180 barometric pressure sensor and an HYT-221 humidity sensor.

cluded in the assembly as well, such that the device communicates with a microcontroller via an I2C-bus on address 0x76 or 0x77. The measuring range is from 300 to 1100 hPa (mbar) with a relative accuracy of ± 0.12 hPa and an absolute accuracy of about 1 hPa. The sensor also reports the temperature with a resolution of 0.1 °C or better, depending on which part number is selected. A sensor mounted on a small breadboard is shown left in Figure 2.36.

The dependence of the reported pressure on the signal from the primary sensor, the strain gauge, is rather intricate and may vary from one device to another due to manufacturing tolerances. Each device, like many other sensors as well, therefore needs to be *calibrated* by exposing it to known conditions, here the pressure, and recording device-specific constants that allow us to accurately determine the pressure from the primary measurements. In the case of the BMP180, a built-in ADC reports a value related to the resistance of the strain gauge measured in a bridge circuit. The calibration constants, determined during the manufacturing and calibration process, are stored in memory on the chip. The datasheet describes a detailed procedure to obtain the pressure based on the value reported from the ADC and the calibration constants. We come back to this topic in Section 4.4.3 when we connect the BMP180 pressure sensor to a microcontroller.

The HYT221 and HYT939 are sensors that measure the *relative humidity* in the range from 0 to 100 %RH with a resolution of about 0.02 %RH. The operational principle of the measurement is based on a capacitor with a dielectric, made of a polymer as the sensing medium. The polymer is highly hygroscopic and easily absorbs water, which changes the relative dielectric constant ϵ_r by a large amount, because ϵ_r of the dry material is much smaller than that of water, which is about 80. Correspondingly, the capacitance C , being proportional to ϵ_r , changes by a large amount as well. This change of the capacitance is determined in a Wheatstone bridge with capacitors in two branches. For the calculation of the relative humidity, knowledge of the temperature is required, and provided on board by a temperature sensor. On board, the raw data are compensated for a number of non-linearities, post-processed, and made available in registers that are accessible via an I2C interface. The I2C address of the sensor is hard-wired to 0x28 and a device is shown on the right in Figure 2.36.

The HMC5883 is a three-axis magnetic field sensor that is based on the change of the resistance due to the *magnetoresistive* effect. The active material is a long meandering strip

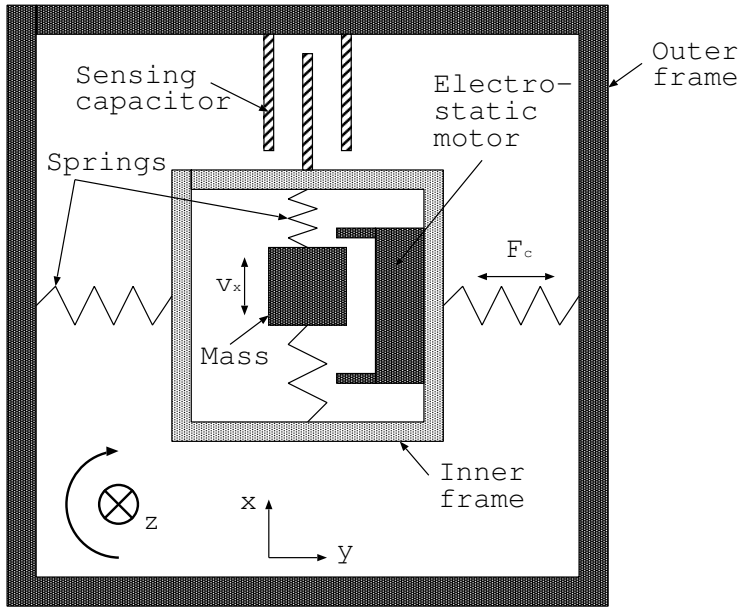


Figure 2.37 The operational principle of one gyroscope in the MPU-6050.

of a nickel-iron alloy on a nonconducting substrate. The change of resistance is due to the change of the spin-orbit coupling of valence electrons in the material. This affects the ease with which the conduction electrons propagate through the material and consequently the resistance. The active material is part of a Wheatstone bridge whose output voltage is conditioned and amplified before being digitized with an on-board ADC and made available on the I2C bus. The HMC5883L measures the field along three axes with a 12-bit (4192 steps) resolution in the range of ± 8 gauss or $\pm 800 \mu\text{T}$. It is typically used in compass applications for mobile phones. The noise floor of the measurements is on the order of $0.2 \mu\text{T}$ and up to 160 measurements/s can be taken.

The MPU-6050 is a three-axis motion-detection chip consisting of an *accelerometer* and *gyroscope* to measure acceleration and angular velocities in three spatial dimensions. The accelerometer is based on the same sensing principle as the ADXL335 discussed previously and illustrated in Figure 2.11, where a spring-suspended inertial mass changes its position when accelerated and varies capacitances that are measured in a bridge circuit. In the MPU-6050, however, digitizer and post-processing digital circuitry are added to provide the measurement data in digital form. The operational principle of the *gyroscope* [12] on the MPU-6050 is illustrated in Figure 2.37, where a small mass, suspended by springs on an inner frame, is forced to oscillate in the x direction by a micromachined electrostatic motor. The mass therefore has a velocity component v_x in the same direction. If the entire assembly rotates around the z axis, which points into the paper, a Coriolis force, given by the crossproduct of the velocity vector and the rotation vector, will point along the y axis. And this force moves the inner frame in the y direction, against the force of the springs that connect the inner to the outer frame. This motion subsequently produces an imbalance in the sensing capacitors, very much like what happens in the accelerometer, and is sensed in much the same way as well. Also, here the analog voltages are digitized on the chip, further processed, and made available on an I2C bus. The performance of the

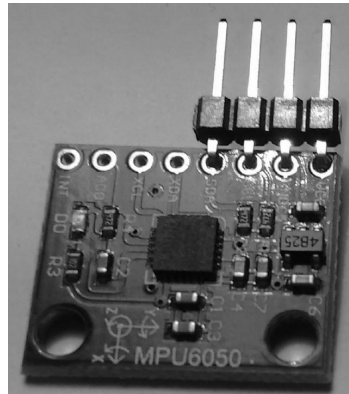


Figure 2.38 An MPU-6050 accelerometer on a breadboard.

device is rather impressive. It measures the acceleration in various ranges between $\pm 2\text{ g}$ and $\pm 16\text{ g}$ with 16-bit resolution and a rate of up to 1 kHz. The rotation speed can be measured between $\pm 250^\circ/\text{s}$ and $\pm 2000^\circ/\text{s}$ with a resolution of about 10 to $100^\circ/\text{s}$ and a rate of up to 8 kHz. The I2C address of the device is 0xA0 or 0xA1, depending on the state of IO-pin AD0.

An enhanced version of the previous device is an MPU-9250 sensor that combines the functionality of accelerometer and gyroscope with that of a three-axis AK8963 Hall sensor, all internally aligned. The magnetic sensor has its own I2C address 0x0C, but shares the same I2C pins with the accelerometer and gyroscope.

The sensor chips are usually very small and difficult to work with, but luckily there are so-called breakout boards available that route the pins of the sensors to normally spaced (2.5 mm spacing) pins that can be attached, for example, to solderless breadboards. Figure 2.38 shows an MPU-6050 mounted on a breakout board.

2.3.4 SPI devices

The SPI interface is a synchronous serial communication bus, similar to the I2C bus, but it can operate at much higher speed and is therefore often used for devices that require the continuous transfer of large amounts of data, such as displays or audio equipment. SPI communication requires one master on the bus, a role normally taken by a microcontroller. The sensors are typically slave devices. They need at least six wires to connect: ground and supply voltage, the clock CLK, one line to send information from the master to the slave (MOSI, for master-out slave-in), one line for the reverse direction (MISO for master-in slave-out), and a chip-select line CS to identify the currently active slave. CLK, MISO, and MOSI lines can be shared among many slaves, but each slave requires its individual CS line.

Some of the devices in the I2C section support this interface as well, and the interface can normally be selected by setting a pin on the device high or low. Details can be found in the datasheet.

In a later project, we will connect external analog-to-digital converters to a microcontroller via SPI communication, in particular, the MCP3304, which has 8 single-ended 12-bit input channels but can also be configured to use two input channels as differential inputs. This circuit detects whether one or the other input is larger and thereby provides an additional sign bit. Thus, it provides one extra bit to obtain a 13-bit resolution. At the same time,

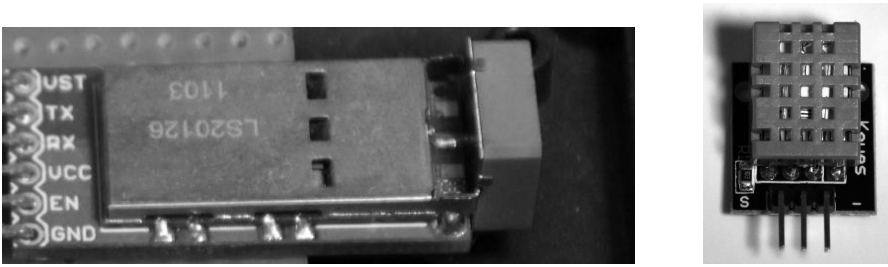


Figure 2.39 A GPS receiver on the left and a DHT11 humidity sensor on a breadboard on the right.

the input range is extended to plus or minus times the supply voltage, despite operating from a unipolar power supply. This chip or its sibling, the MCP3208, are strong candidates to expand the number and resolution of analog input channels for many microcontrollers.

2.3.5 RS-232 devices

Several devices report their measurement values by sending them via the asynchronous RS-232 protocol. Originally, the physical medium for the communication channel used a current loop, but nowadays most sensors operate on 3.3 or 5 V levels. The communication happens point-to-point between two partners who have agreed on a communication speed, which is often 9600 baud or 115200 baud. Three wires are required at the minimum, one for ground potential, one, labeled TX for transmitting from device A to device B, and another one, labeled RX, for transmitting in the reverse direction. To establish communication, cables connect the TX pin on one device to RX on the other and vice-versa. Naturally, ground pins need to be connected as well.

One device that supports RS-232 communication is the LV-EZx distance sensor we already discussed earlier. It can be configured to send the distance measured as an ASCII string that can be read in a terminal program.

Sensors that query the *global positioning system* (GPS) use a small patch antenna on the sensor to pick up signals from a number of satellites placed in geostationary orbits, which broadcast their position and timing information with high precision. On-board electronics that normally comprise a microcontroller use triangulation in order to determine the position of the sensor with high accuracy and convert that information to an ASCII string containing the position in a standardized format, called *NMEA*. The string is written, typically once per second, to an RS-232 serial line, where it is straightforward to read and decode.

2.3.6 Other sensors

Apart from the standardized protocols, there exist a number of communication standards that device vendors come up with. Examples of such devices are the *relative humidity sensor*, DHT22, and the DHT11; the latter is shown on the right in Figure 2.39. In order to determine the relative humidity, the temperature must be known and it is determined with a thermistor. To measure the humidity they use a capacitive humidity sensor. The operational principle is based on using a small capacitor with an exposed dielectric that has a large affinity to attract water, which changes the relative dielectric constant ϵ_r and the capacitance by a large amount. This is determined in a Wheatstone bridge with capacitors

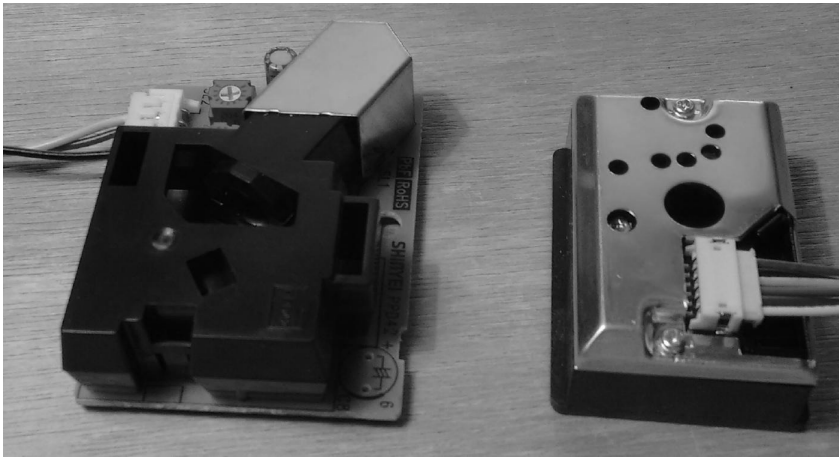


Figure 2.40 A Shinyei PPD42NS particle sensor (left) and a GP2Y1010AU0F dust sensor (right).

in two branches. Internally, the DHT sensors have this, and further circuitry such as an analog-to-digital converter on board to calculate the relative humidity. They provide the data using a non-standard, though documented, digital interface that we discuss further in Section 4.4.5, where we cover connecting a DHT11 to a microcontroller.

The DS18B20 temperature sensor, as was the LM35 discussed earlier, is based on a bandgap temperature sensor. Here, however, ancillary digital electronics and signal processing circuitry is added on the chip such that the measurement value is postprocessed and made available using the so-called Dallas 1-wire bus protocol. The 1-wire protocol uses only ground and a single additional wire to transmit power and information to and from the device.

The air quality can be characterized by the density of microscopic particles suspended in air. In Figure 2.40 we show two such sensors. On the left we see a PPD42NS particle sensor. Inside this detector a resistor heats the air, which causes the air with the suspended dust particles to rise and pass through the light emitted by an infrared diode. There the dust particles scatter the light onto a phototransistor, which pulls an output pin to low potential. After signal conditioning and amplification, a cleaned-up signal is available. It is low when particles scatter light, and high otherwise. The device is calibrated such that the ratio of time at low signal to total time can be translated to particles per liter. The GP2Y1010AU0F, shown on the right in Figure 2.40, works in a similar way. It also detects light scattered off of dust particles, but it periodically turns the infrared diode on and integrates the signal from the phototransistor and one has to sample the output value 0.28 ms after the LED was turned on. The difference between the signal with LED on versus off provides reasonable rejection of ambient light. The performance of both dust sensors can be improved if we place them in the airstream created by a fan, which we need to turn on and off.

And that brings us to actuators, devices such as switches or motors that cause some change in external conditions. Sometimes they are part of the measurement process, such as the fan mentioned in the previous paragraph, or we need to move the sensor to where we want to measure, which typically requires motors.

QUESTIONS AND PROJECT IDEAS

1. Where is the Fermi level in Figure 2.3?
2. Discuss different ways of measuring the capacitance of a capacitor.
3. Discuss different ways of measuring the inductance.
4. Research the possibility of how to use a pin diode as a detector for *ionizing radiation*.
5. Like pin diodes, LEDs generate a small current when illuminated. Use this effect to build a *photometer* from a number of LEDs with different colors to sense different parts of the optical spectrum.
6. Discuss a system to *measure the rotation speed* of a wheel with a small magnet and a reed switch. Can you design it such a way that you can determine both speed and direction of the rotation?
7. Discuss methods to determine the *wind direction*.
8. Discuss how to measure the amount *rain during a day*.
9. Research how to measure the *concentration of sugar* (dextrose) in water by observing the polarization change as a function of amount of sugar dissolved. Which of the described sensors can be used?
10. Build a discrete 3-bit flash ADC from a resistor ladder and external components, such as the LF198 sample-and-hold circuit, two LM339 quad-comparators, and a 74HC147 8-to-3 priority encoder.

Actuators

Even though sensors are the main topic of this book, sometimes we need to turn devices on and off, or we need to move a sensor very accurately, with much higher precision than we can achieve by hand. In other cases, the sensor is not at the location where we need to measure some quantity. In such situations we need an actuator to move it in a controlled way. Here *actuator* is the generic term for a device that controls external parameters. Examples are motors, valves, or switches, and we start the discussion with the latter.

3.1 SWITCHES

Turning an electric signal on and off is very easily done by toggling an output pin of a microcontroller, as we shall discuss in quite some detail in the next chapter. A typical microcontroller can provide rather limited currents on the order of a few mA. This is normally sufficient to control a single light-emitting diode, an LED, which typically draws less than 20 mA.

3.1.1 Light-emitting diodes and optocouplers

An LED, shown on the left in Figure 3.1, is similar to a conventional diode and consists of a semiconductor with a *pn* junction. If it is forward biased, with the *n*-terminal connected to the lower potential, the charge carriers, electrons and holes are pulled into the space-charge zone, provided the voltage is higher than the voltage drop of the diode. This situation is opposite to the situation encountered in Figure 2.13, where the diode was reverse biased. The simplified band-level scheme and the corresponding circuit for the forward-biased LED is shown in Figure 3.2. We only show the lower boundary of the conduction band, and the upper boundary of the valence band inside the diode. Here both electrons and holes are pulled into the space-charge zone and have a chance to recombine. The energy that is released by the electrons dropping from the conduction band to the valence band is emitted as a photon. Note that the forward voltage drop of the LED is about the size of the bandgap and the resistor is needed to limit the current. LEDs emitting different colors have different bandgaps, and correspondingly, different forward voltage drops, where shorter wavelengths corresponds to larger bandgaps.

As already seen in Figures 3.1 and 3.2; LEDs connect to a circuit with two wires; the short one is the cathode and needs to be connected to ground, or the more negative potential. The cathode is normally indicated by a flattened face of the hemispherical housing. The other wire is the anode and is connected to the more positive potential, provided we want the LED to light up. But simply connecting the LED to the power rails will likely destroy

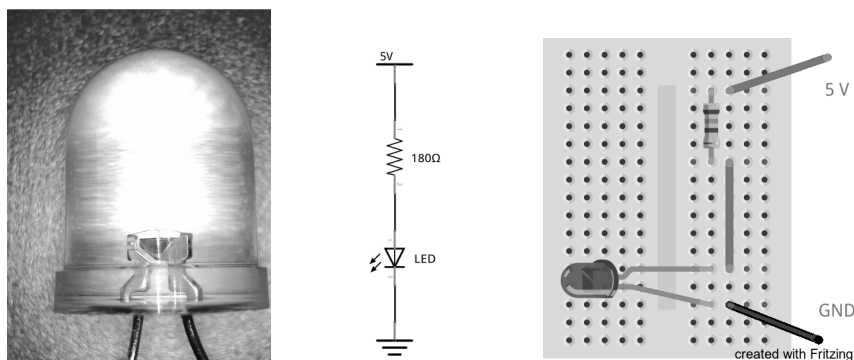


Figure 3.1 A close-up of a light-emitting diode is shown on the left. In the center is the schematic of connecting an LED and the same circuit on a breadboard.

the diode. We need to connect a resistor in series with the diode, as shown in the center and right on Figure 3.1. The voltage drop V_d across most LEDs is between 1.5 and 3 V, where the lower values apply to red and infrared LED, and the higher value to blue and ultraviolet LEDs. Moreover, the typical operating current of the diode must be limited to some value below $I = 20\text{ mA}$. The resistor thus needs to be specified according to $R > (V_c - V_d)/I$, where V_c is the supply voltage. A red LED thus works nicely on $V_c = 5\text{ V}$ with $R = 180\ \Omega$ or $220\ \Omega$, or even larger values where I picked a larger resistor with the commonly available values. Normally the resistance value is not very critical; if chosen too large, the LED is less bright.

In order to dynamically vary the brightness, changing the resistor value is rather inconvenient. The better way to achieve this is by rapidly turning the diode on and off at a rate much faster than the human eye can resolve, typically at a kHz rate. Changing the duty cycle of the on- versus off-time proportionally changes the brightness of the LED. The added benefit of this method, called *pulse-width modulation*, is that the dissipated energy is reduced by the same ratio.

If an LED or any other pulse-width modulated device needs to be galvanically separated from the control electronics, we use an optocoupler. This is a small integrated circuit that consists of an LED and a phototransistor. As discussed in Section 2.1.3, a phototransistor behaves like a normal transistor with a high current amplification, but instead of passing a current into the base terminal of the transistor, the LED illuminates the region with depleted charges in the transistor and creates a large number of electron-hole pairs. Thus, the transistor becomes conducting and switches on a device on the “other” side of the circuit, without being electrically connected. This implies that only the on- or off-state or digital signals are communicated across the optocoupler. Typically they are used if devices located at different electrical potentials need to be switched on or off, or to prevent electric perturbations from the “other” side. We note in passing that the MIDI communication between musical instruments uses optocouplers at their respective inputs. This removes electrical disturbances and protects the musicians from electrocution.

So far we only switched LEDs on and off, but we will look at how to control large currents and loads.

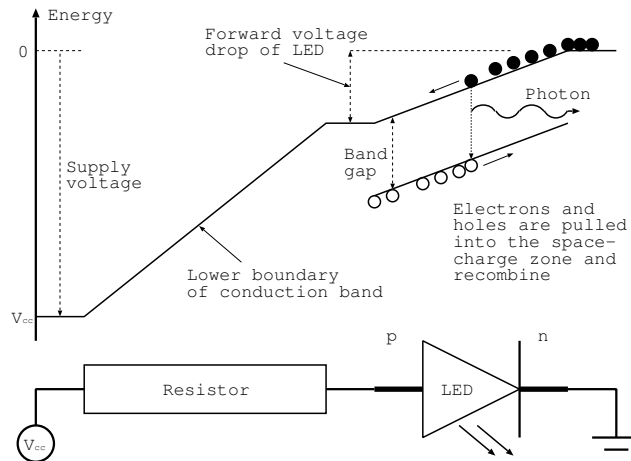


Figure 3.2 The physics of an LED.

3.1.2 Large currents

Switching large currents and voltages requires us to “amplify” the small current that the microcontroller provides, and transistors of various flavors do just that. Namely, they amplify the current flowing into their base terminal by a factor β , which is specified in the datasheet of the transistor to a value that flows across the collector-emitter terminals. Refer to the left side of Figure 3.3 for the naming and location of the respective terminals of an NPN-transistor. If the amplification β is large enough, moderate base-currents cause the output current to saturate the transistor such that it behaves like a switch that turns on the connection from collector to emitter.

In the example shown in Figure 3.3, the $1\text{ k}\Omega$ resistor on the base limits the input current across the base-emitter link to 5 mA , provided the control voltage is 5 V . If the transistor has an amplification of 100, which is typical for many small signal transistors, it can switch up to 500 mA , which is more than is actually needed in this case. The actual current flowing through the collector-emitter link is determined by the 12 V supply voltage to the LED and the 680Ω resistor limiting the current. Note that using the transistor also decouples the voltage level of the controlling circuitry and the supply voltage, here 12 V , of the consumer, the LED. This way of operating a transistor is called *open collector*, because we can think of the collector terminal as a generic connection point for an (almost) arbitrary load that is connected to its own supply voltage. If the emitter-collector link is in the nonconducting state, no current flows, and the load is turned off. If a positive current flows into the base terminal, the emitter-collector link becomes conducting, the lower terminal of the load is shorted to ground, and a current flows from the load’s power supply through the load and turns it on.

Since switching is very common, small switching currents are often desirable. This requires a transistor with a large amplification. One way to achieve this is to connect two transistors as a Darlington pair, as shown on the left in Figure 3.4. Such a pair has the current amplification of approximately the product of the two individual transistors, and is often used in switching applications, which accounts for the availability of integrated circuits that pack seven or eight Darlington pairs into a single package. An example, the ULN2003, is shown on the right in Figure 3.4. The seven input terminals on the left-hand side of the

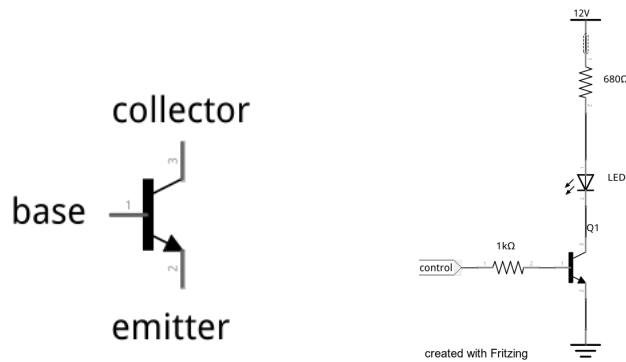


Figure 3.3 The terminals of an NPN transistor (left), and using an NPN transistor as switch (right).

package are the base terminals of the Darlington pairs, and the output terminals on the right-hand side are the corresponding open collector terminals. The ground connection is located at the lower left, and the external positive power supply voltage is connected to the terminal labeled COM. This specific chip can switch up to seven times 500 mA, and the supply voltage for the load can be up to 40 V. The datasheets provide a lot more detailed information.

In case we need to switch very high voltages of up to 1 kV or very large currents, we use MOSFETs. They require almost no current to flow into the gate in order to switch. Only the capacitance between the terminals needs to be charged, and this creates an electric field that pulls charge carriers into the depletion zone of the MOSFET, which causes it to conduct.

Using different flavors of transistors makes switching unipolar voltages very convenient, but if we need to switch AC household voltages on or off, such as lamps or the wake-up radio, we need a relay. Relays consist of a small electromagnet that mechanically closes or opens a contact. This achieves a high degree of separation between the controlling circuitry and the load circuit. The two sides only communicate via the magnetic field that toggles

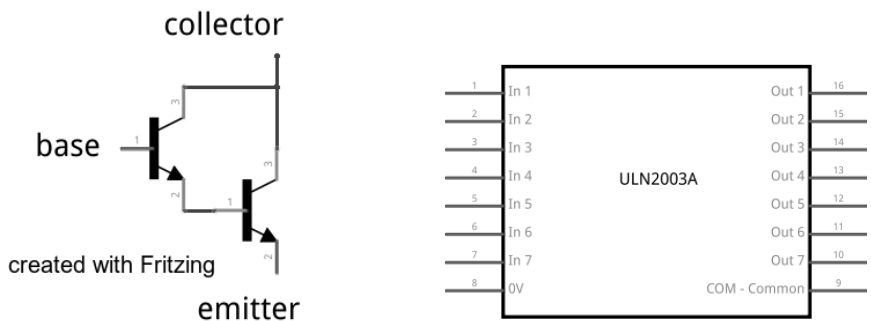


Figure 3.4 Two NPN transistors connected to form a Darlington pair (left) and a ULN2003 Darlington array (right).

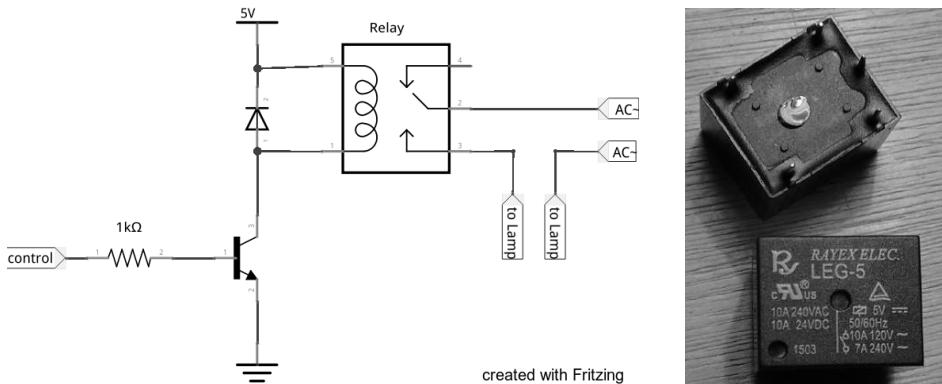


Figure 3.5 Schematic illustrating the functionality of a relay (left) and an image of a relay (right).

the switch, depending on whether current flows through the coil or not. A schematic image is shown on the left in Figure 3.5. There we see the coil with its two terminals on the left, and the switch on the right-hand side that toggles between the two terminals. Normally we need to prepend a transistor to switch the coil on and off because this requires a larger current than the microcontroller provides. Moreover, we need to pay attention, because the coil is an inductive load, and turning it off causes a large induction voltage that may damage the transistor or other parts of the circuit. This can easily be prevented with a *flyback diode*, also shown in Figure 3.5, bypassing the coil in the relay in the normally non-conducting polarity with cathode pointing towards the positive supply voltage. Any voltage in the reverse direction passes through the diode, rather than the transistor. Care needs to be taken with respect to power rating of the diode, and the speed. Normally Schottky diodes, which are particularly fast-switching, are used.

The terminals on the high-power side can handle both AC and DC voltages. The relay is very similar to a normal switch in the way that it establishes an electrical contact between two terminals. In Figure 3.5, it closes or opens the contact between the AC supply and a lamp, but the lamp can be replaced by any other device that needs to be turned on and off.

Please note that the relay in the image can actually handle up to 240 V AC, but we need to stress that this is a potentially lethal voltage. This should only be handled by suitably trained and qualified personnel. If in doubt, do not try it yourself!

Now we have several means at our disposal to switch devices and actuators on and off. Some of the most prominent and useful actuators are motors, and that is what we discuss next.

3.2 MOTORS

Motors in general translate chemical or electrical energy into mechanical energy, normally by rotating an axle in order to provide torque. Here, and in subsequent sections, we address electrical motors only. They come in a wide variety of types and can be powered by both AC and DC voltages. One of the most common types, and the one we will use, is a conventional DC motor with a commutator.

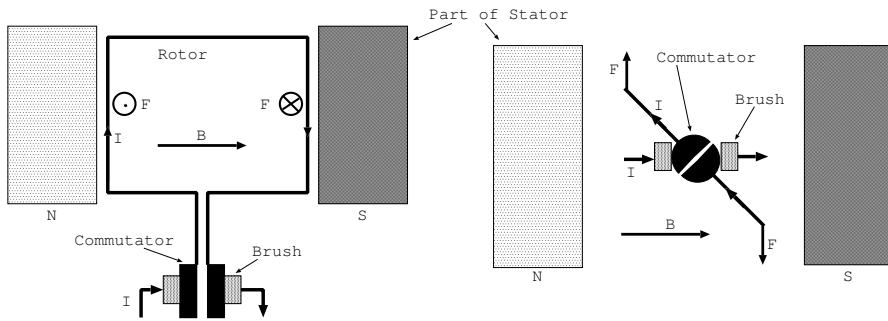


Figure 3.6 Basic operation principle of a DC motor. The magnetic north pole of the stator is to the left and the south pole to the right of the coil. Current is supplied to the brushes where it enters the commutator and passes through the rotor-coil, where the Lorentz force causes a force on the wire, moving it upwards. After half a turn, the commutator has rotated and reverses the polarity such that the current on the left side of the coil points in the same direction as before.

3.2.1 DC motors

The operating principle of a DC motor is most easily explained with the help of Figure 3.6. It is based on static magnetic field that is stationary in space, the *stator*, and one or more coils, electrically excited by an external voltage source. They are forced to rotate by reversing the polarity of the exciting voltage synchronously with the rotation. The rotating part is called the *rotor* and the synchronous switch is the *commutator*. The torque on the rotor is provided by the Lorentz force that the electrons of the electric current experience in the external magnetic field. After half a turn, the external magnetic field has the “wrong” polarity to continue forcing the coil to rotate, and would brake the rotation. This is prevented by reversing the polarity of the current flowing at just that point in time which allows the rotation to continue. The polarity reversal is effected by the commutator, which is shown as the two D-shaped objects that rotate between externally fixed electrodes, called *brushes*. Moreover, reversing the supply voltage causes the motor to rotate in the opposite direction.

The sliding contact between the brushes and the commutator sometimes causes sparking at the moment of polarity reversal, because the brushes briefly connect the two D-shaped objects and thereby short circuit the supply voltage. The sparks then cause electrical disturbances that may be partially alleviated by connecting the brushes with a small, say 100 nF, capacitor. This also reduces the creation of ozone with its distinct smell, which often accompanies the operation of DC motors. In commercially available motors these effects are minimized, and other motor designs are used that circumvent these deficiencies. For example, in some *brushless DC motors* the magnetic field of the rotor is provided by permanent magnets whose position is continuously monitored with a Hall sensor. This information is used in an electronic control unit to periodically excite the current through the coils of the stator in order to maintain the rotation. The complexity of the control unit places these motors outside the scope of this book, and we use normal DC motors for our projects despite their shortcomings.

We can adjust the torque and the speed of the motor by varying voltage applied to the coils. A higher voltage makes the motor turn faster, and reasonable voltages are determined by the resistance of the coils. Often, however, it is advantageous to limit the voltages and

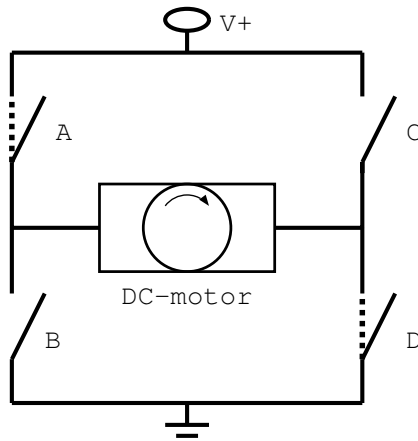


Figure 3.7 Schematic illustrating the functionality of an H bridge.

control the currents that flow through the coils instead. Moreover, since the rotor has a finite inertia due to its mass, we can use pulse-width modulation to control the total power delivered to the coils in order to adjust torque and speed. In the fleeting moments when no current flows, the inertia maintains the rotation.

In case you have a model railway, you may know that in the early days the speed of the trains was regulated by a variable transformer, and it was very difficult to run the trains at very slow speeds with voltages too low to reliably excite the coils. Modern model-railway controllers use pulse-width modulation to regulate the speed. In that way, the full supply voltage and current are always flowing, which reliably excites the coil, just not all the time. Operating the trains at slow speed now works much more reliably.

In this context we have to keep in mind that turning the coils rapidly on and off also rapidly changes the magnetic field, and this causes a voltage, the backwards electromagnetic force, or *back-emf* that opposes the driving voltage. Thus, the faster the motor turns, the more the back-emf reduces the voltage, and thereby also the current flowing through the coils. This results in a reduced torque and for a given coil resistance we can either run at a slower speed or increase the driving voltage in order to maintain the required torque.

As with the model trains, we also want to run the motor both forward and backward. Therefore we need to control the polarity of the supply voltage. Instead of reversing the supply cables by hand, which is not a very convenient option, we use an H bridge whose functionality is easily explained with the help of Figure 3.7. In the center of the figure a DC motor is located, with its supply leads extending to either side. The unipolar supply voltage is connected to the upper terminal labeled V+ and to ground. Depending on the position of the switches A, B, C, and D, the current flows through the motor. In case A and D are engaged, it flows from left to right. This is indicated by the dotted lines and the sense of rotation indicated by the small arrow in the motor. In case B and C are engaged, while A and D are open, the current flows in the opposite direction, causing the motor to turn in the opposite direction as well. Thus, by suitably toggling the four switches, we can adjust the sense of rotation of the motor at will. We only have to ensure that only one of the switches A and B is engaged at a given instance, because otherwise the supply voltage is short circuited.

Of course, we can replace the switches with transistors and arrive at a system that is



Figure 3.8 A small model-servo.

easily controllable by a microcontroller. And the situation is made even simpler, because ready-made integrated circuits that implement H bridges are available. The L293D is one of them, and we will use it in later chapters to control motors. Finally, we add speed control by pulse-width modulation. We either add a switching transistor between the upper terminal and the supply voltage, or we directly modulate the input terminals that control the switches A, B, C, and D.

At this point we can adjust the speed and direction of a motor, which is essential for moving from one place to another, and the velocity at which this happens. On the other hand, if we only want to change the position or angle by a small amount, such as the rudder of a boat, we need means to directly adjust the position, rather than the velocity. Next we discuss two types of systems that do this, servomotors and stepper motors.

3.2.2 Servomotors and model-servos

The term *servo* refers to using the motor in conjunction with a position encoder in a *closed-loop* feedback, or servo, loop [18]. Here the motor speed is continuously adjusted to reduce the difference between the desired position and the actual position, as reported by the encoder. This type of servomotor is often used in industrial applications, and requires an elaborate control system with a PID controller [18] and powerdrive-electronics. Servomotors are used in large industrial robots and applications that require high accuracy of positioning. Matching the parameters of the controller to the desired performance of the control loop in terms of accuracy, speed, and acceptable overshoot requires expert attention, and is beyond the scope of this book.

Instead, we use the modest cousin of the servomotor, the *model-servo*, often simply referred to as a *servo*, shown in Figure 3.8. It is often encountered in radio-controlled cars and planes, but can also appear in robotic applications, such as controlling the position or angle of smaller robotic arms. Servos have a potentiometer mounted on the rotating shaft that is used as an encoder. It provides information about the position of the shaft, which typically has a rotation range of 0 to 180 degrees. The resistance is compared to a voltage derived from an electric signal corresponding to the desired position. The difference signal is amplified and used to turn a DC motor in the direction that minimizes the difference.

This constitutes a simple proportional controller. Normally a small gearbox between the motor axis and the shaft reduces the speed, but increases the precision of control and the torque. Note that the servo connects with three wires to the outside. Two of the wires are ground and supply voltage, nominally 4.8 V. They are colored black and red, respectively. The third wire, yellow or white in many cases, carries the information about the desired position.

The information about the desired position is communicated to the servo as a pulse-width modulated signal, as shown in Figure 3.9. The servo expects a train of pulses with a spacing of 20 ms. Each individual pulse has a duration between 1 and 2 ms, with a 1.5 ms duration specifying the mid-position. Producing such pulse patterns is easily done with a microcontroller, and we will do that in later chapters.

Servomotors and servos use a closed-loop system to achieve a high accuracy of positioning, while stepper motors achieve this without closed-loop feedback.

3.2.3 Stepper motors

Stepper motors change the angular position of a shaft in small discrete steps, such that counting the steps gives the position. In this way, no feedback or servomechanism is needed to achieve high repeatability, and the motor can be operated open loop.

In Figure 3.10, we illustrate how this is accomplished in a permanent-magnet stepper motor. The static part of the motor consists of an iron yoke and four coils, of which two each are operated in series. Here the upper and lower coils, labeled A and B, and the right and left coils, labeled C and D, are two such pairs. On the rotor a moderately large number of permanent magnets are assembled. In the figure we only use six magnets, labeled 1 through 6, with the black end indicating the north pole. They are oriented radially and their polarity alternates.

We assume that initially only coils A and B are excited in such a way that the upper coil behaves as a magnetic north pole and the lower as a south pole. In that case, the magnetic field lines pass through both upper and lower coils, pass through the magnets in the rotor, and return through the surrounding yoke. With this coil excitation the rotor is indeed oriented as shown in Figure 3.10, because the upper coil behaves as a north pole and attracts the south pole of the upper permanent magnet, labeled 1. Consequently, they are as close as possible. The converse is true for the bottom coil and the magnet labeled 4. In order to turn the rotor by 30 degrees in the counterclockwise direction, we have to turn coils A and B off and coils C and D on. Thus, coil C on the right-hand side is a north pole that attracts the permanent magnet labeled 3, which has a south pole pointing outwards. Again, the converse is true for coil D on the left and magnet 6. The rotor will then move until magnet 6 faces coil D. We continue turning the rotor by turning coils C and D off and reversing the polarity of coils A and B, such that the upper coil acts as a south pole that

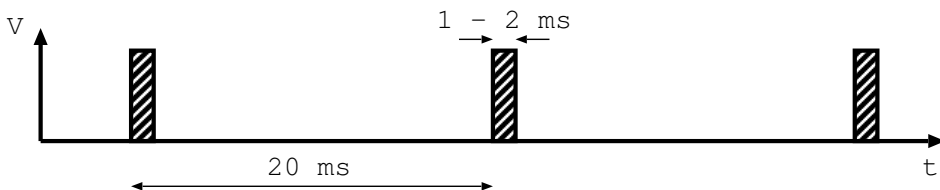


Figure 3.9 The timing of the control signal for the servo.

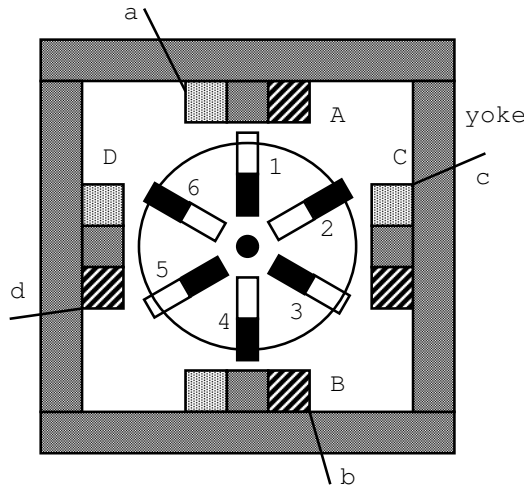


Figure 3.10 Schematics of a stepper motor. Note that the coils are denoted by upper-case letters and the corresponding terminals with lower-case letters.

attracts magnet 2. Continuing in this way, we find that we can turn the motor in a number of steps of 30 degrees each by periodically exciting the coils in the following pattern:

```
terminal a: 1000 1000 1...
           b: 0010 0010 0...
           c: 0100 0100 0...
           d: 0001 0001 0...
```

where time runs from left to right. Here, 1 indicates that the terminal is connected to the positive power supply voltage, and 0 indicates that the coil is connected to ground potential. Stepping through the sequence backwards causes the motor to turn in the other direction. Observing that time slots 1 and 3 show reversed polarity for the coils A and B implies that we need an H bridge to implement the polarity reversal. We will return to this implementation in Section 4.5.

It is instructive to analyze the excitation pattern of the four terminals a through d a little more closely. The first observation is that a sequence of four time slots repeats itself over and over again. Second, we observe that a fundamental pattern is the excitation of terminal a with the sequence 1000. The excitation of terminal b is shifted to the right by two time slots which we may interpret as 180 degrees out of phase if we assume that 360 degrees corresponds to the four time slots. Likewise, terminal c and d are 90 and 270 degrees out of phase with the excitation of terminal a. This interpretation will prove useful later when we write our own stepper motor driver and discuss other modes of operating the motor, such as half- and microstepping modes.

Note that in the excitation pattern only one coil is excited at a time. But we can also operate the motor with two coils excited simultaneously, one coil pulling one permanent magnet and the other coil pushing another magnet. In this way the torque of the motor can be increased by about 40%, albeit at the expense of doubling the required power, because there are two coils excited at the same time. The pattern with increased torque is based

on the fundamental sequence 1100. Applying the appropriate phasing between the four terminals from above, we arrive at the following pattern to excite the terminals:

```
terminal a: 1100 1100 1...
           b: 0011 0011 0...
           c: 0110 0110 0...
           d: 1001 1001 1...
```

where we see that the excitation of terminals **b** through **d** is shifted by 2, 1, and 3 time slots with respect to that of terminal **a**. A special point to note is that exciting two coils simultaneously will cause the positions towards which the permanent magnet will point to lie in between the coils, rather than directly facing the coils. Since we now have the single-coil excitation pattern where the magnets point at the coils and the double-coil excitation pattern where it points in between the coils, we might consider a combination of both patterns...

...and find that by interleaving the steps of the single-coil and double-coil excitation patterns, we obtain a sequence where the step size is halved. This mode of operation is called half-step mode. The pattern in which we have to excite the coils is the following:

```
terminal a: 11100000 11100000 1...
           b: 00001110 00001110 0...
           c: 00111000 00111000 0...
           d: 10000011 10000011 1...
```

where the period length is 8 time slots with a fundamental excitation pattern of 11100000. The excitation sequence of terminal **b** is the fundamental shifted by half the period length, or four time slots, or 180 degrees. Note that here, 360 degrees, correspond to the 8 time slots. And the sequence for terminals **c** and **d** are shifted by two and six time slots, respectively.

Note the general pattern: terminals **a** and **b** are excited by a cosine-like sequence, which is evident by calculating the voltage applied to coils **A** and **B**, which is **a-b**

```
a - b =  1  1  1  0 -1 -1 -1  0
```

which admittedly is a poor approximation of a cosine. Likewise the voltage applied to coils **C** and **D** is **c-d**

```
c - d = -1  0  1  1  1  0 -1 -1
```

and that is an equally poor approximation of a sine. Apparently, the signals applied to the coil pairs roughly follow a cosine and a sine-like excitation. It is easy to see how we can generalize this and use better approximations of the cosine- and sine-like excitations of the coils, which is called *microstepping* the motors.

In microstepping modes, the permanent magnets on the rotor are moved to several intermediate positions in between coils. This finer control comes at the expense of requiring a more advanced driver to control the output current in finer steps than turning it on and off. An example of such a driver is the DRV8825 circuit, which exposes very few control pins for direction and stepping as well as selecting which microstepping mode to use. Internally it generates the appropriate excitation pattern and the currents that are applied to the coils.

In the above example, the stepper motor uses only six permanent magnets, resulting in an angular increment of 30 degrees per step, or 12 steps per revolution, but it is easy to see that we can increase the number of magnets to reduce the step size significantly. Typical step sizes for commercially available motors are 200 or 400 steps per revolution.

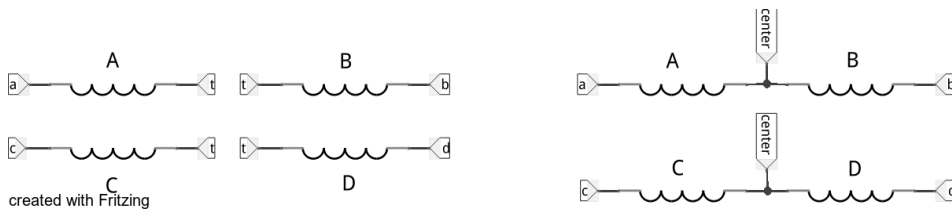


Figure 3.11 Wiring of the coils in a bipolar stepper motor (left) where the respective center terminals are often connected. In a unipolar motor (right) the center terminals are exposed.

Here we need to point out that assigning one phase to coil terminals **a** and **b** and the other to terminals **c** and **d** is only a convention, and some vendors prefer to enumerate the coils and associated terminals sequentially when looking at the motor in Figure 3.10. This amounts to swapping the labels of terminals **b** and **c** and implies that we may have to swap two cables from the driver circuit to the motor in order to make it turn. It is best to check the datasheet for the motor to find out what convention is used for that particular motor.

Furthermore, stepper motors come with different numbers of connecting cables that expose the terminals of the coils. On the left in Figure 3.11, the situation is depicted where all eight terminals are exposed. Sometimes the center taps are internally connected, in which case only four terminals are exposed, and this corresponds to the situation in the discussion earlier in this section. On the right, the center terminals are internally connected and are exposed, in which case the total number of exposed wires is six. If the two central connections are internally joined, five wires are exposed. Normally the exposed wires are color coded and identified in the datasheet, but measuring the resistance between the exposed wires allows us to determine the internal wiring of the motor experimentally.

In the previous stepper motor examples, we always reverse the polarity of the coil excitation, which requires a drive circuit with an H bridge. Stepper motors, however, can also be operated without an H bridge in unipolar mode. In Figure 3.12 we show the connection for a unipolar stepper motor that has center taps of the coils wired together, as discussed in the previous paragraph, and exposes only one terminal that is connected to the positive voltage of the motor power supply. The other terminals of the coils are the collector of an NPN or Darlington transistor and are bypassed with flyback diodes. In a ULN2003 circuit, the diodes are already built in. Placing a positive voltage to one of the base terminals **a**, **b**, **c**, or **d** causes the corresponding transistor to conduct and the coil to be excited.

For unipolar stepper motors, we can use the same excitation pattern we use for bipolar motors. For example, the excitation pattern for the full-step mode with larger torque will also turn a unipolar stepper motor in one way

```
terminal a: 1100 1100 1...
           b: 0011 0011 0...
           c: 0110 0110 0...
           d: 1001 1001 1...
```

Reversing the order causes the motor to turn the other way around. Half- and microstepping modes can be implemented in the same way.

All devices above were based on switching voltages or currents fully on and fully off,

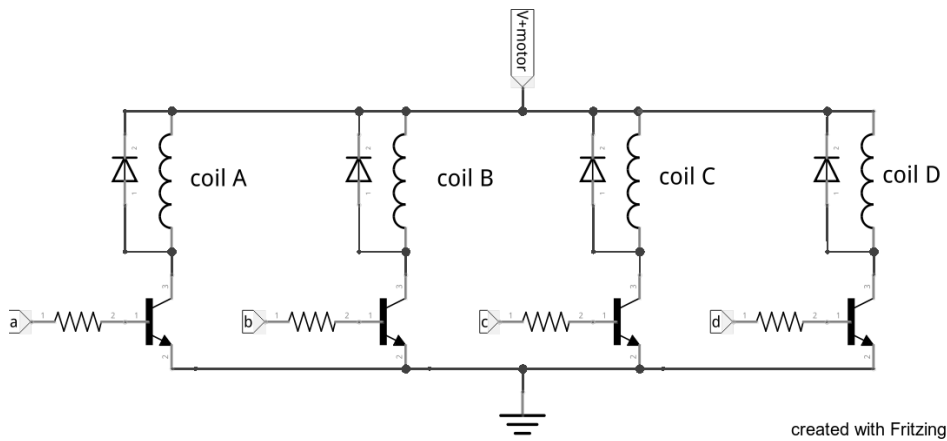


Figure 3.12 Connection for a unipolar stepper motor.

but occasionally, careful adjustment of a control voltage is required; for example, to set the control input of a power supply that determines its output current.

3.3 ANALOG VOLTAGES

A simple way to produce a constant voltage is to low-pass filter a pulse-width modulated output voltage. The filter reduces the on-off variation and results in an average voltage level corresponding to the peak voltage times the duty factor of the pulse-width modulation. Varying the duty factor will produce a time-varying output voltage. There are, however, limitations to this method, mainly due to the quality of the output voltage, which is likely to have some ripple left over from the original modulation.

A more reliable way is to use a *digital-to-analog converter* (DAC), which is a device that converts a digital word with a given length, such as 8 bits or 12 bits, to a discrete voltage with 256 or 4096 intermediate levels, respectively. The operational principle is illustrated in Figure 3.13 for a 4-bit DAC using an $R-2R$ resistor network and an operational amplifier.

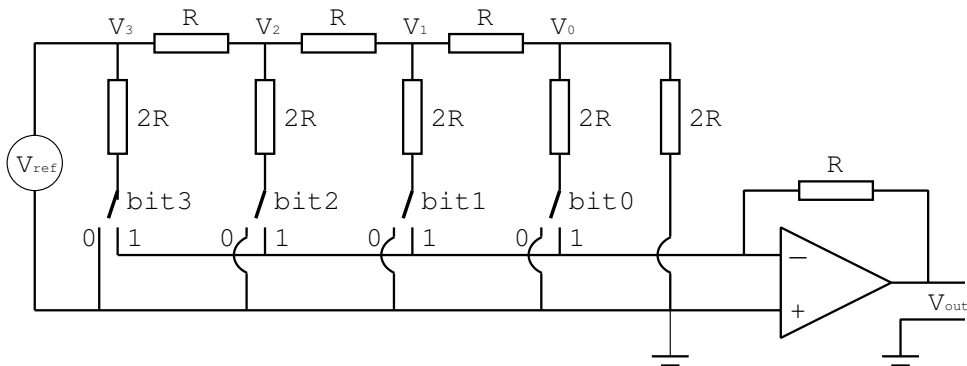


Figure 3.13 Operational principle of a digital-to-analog converter.

The switches labeled `bit3` to `bit0` represent the digital input word, and the inverting input of the operational amplifier sums the currents that are added by toggling the respective switches. The inverting input is a virtual ground because the op-amp forces it to be the same potential as the noninverting input. This implies that the current flowing through all resistors is constant, and independent of the position of the switches; if the switch is in position 0, the current flows to the real ground, and if it is in position 1, it flows into the virtual ground. This implies that the voltages V_3, \dots, V_0 are constant and are given by $V_{k-1} = V_k/2$, which is easy to see for V_1 and V_0 . The two right-most resistors with values $2R$ in parallel combine to a single resistor of value R . But this means that the node with V_0 is sandwiched between two resistors of value R , of which one is connected to a point with voltage V_1 , thus $V_0 = V_1/2$. Since all the voltages V_n are fixed the current flowing into the inverting input of the operational amplifier, provided that bit number k is set, is given by $V_k/2R$, with $V_k = V_{ref}/2^{3-k}$. And all currents coming from the four branches add up, and the operational amplifier converts the current to an output voltage V_{out} .

Most DACs use the $R - 2R$ ladder resistor network, with a current-adding operational amplifier but add a digital front end with semiconductor switches that are controlled from a parallel bus, I2C, or SPI. An example of the latter is the MCP4921, a 12-bit DAC that is controlled via a SPI-compatible interface.

The analog voltages, switches, and motors we discussed so far are likely the most-used actuators, but there are others, and we consider a selection of them in the following section.

3.4 OTHER ACTUATORS

In order to open doors or to move devices between well-defined end stops, *solenoid magnets* are used. They are magnetic coils with a soft-iron core that are switched on and off. They are conceptually not much different from an LED, except that the current required is much higher, and that requires a larger power supply. Moreover, a solenoid is a large inductive load, and turning it off causes a large back-emf that requires a large flyback protection diode, just as we discussed in the case of relays and DC motors in Section 3.1.2.

The flow of liquids and gases can be controlled with pumps and valves. Pumps are usually based on motors driving paddle wheels, propellers, or turbines that move the liquid or gas from a region of low pressure to that of high pressure. The motors are controlled in the same way as discussed before. Flow rates are controlled by valves, which constrict the aperture through which the medium flows. An example is a butterfly valve, shown in Figure 3.14. The aperture can be restricted by the shaded aperture, which has the same diameter as the inner diameter of the pipe. Rotating the shaded aperture then adjusts the flow rate from fully closed, in which case the shaded aperture covers the entire inner diameter of the pipe, to fully open, in which case it stands perpendicular to the flow of the liquid or gas. To change its state we can either use a servo for smaller systems, a stepper motor, or a large servomotor, in case we need to constrict the flow in an oil pipeline.

A common means for controlling large machinery uses hydraulics, based on controlling the flow of hydraulic oil. It uses pumps to move the oil into cylinders that cause a linear travel, or it is pumped through a turbine or propeller-based device to cause rotary motions. In either case the original pressure is provided by pumps, based on electric motors. The position of valves is also controlled by the same devices we discussed earlier in this chapter.

And finally, we may consider humans as part of the larger system, and they need to be actuated. For this purpose we may use various attention-grabbing devices, such as piezoelectric buzzers, sirens, and loudspeakers, as well as optical devices comprising LEDs, neo-pixel

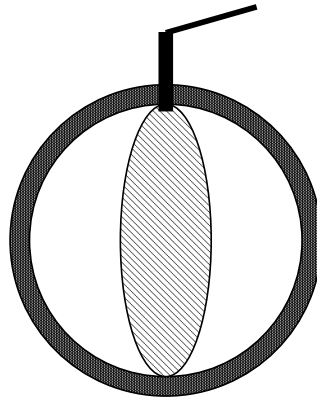


Figure 3.14 Sketch of a butterfly valve.

color LEDs, liquid crystal displays, or small displays such as those found on printers or washing machines that report the current state of the apparatus.

By now we have covered quite a selection of actuators and sensors and can turn things on, move them around, and measure a number of physical quantities. The next task is to interface this variety of sensors and actuators with their multitude of interfaces, and provide a standardized interface to communicate with the external world. This is the task of the microcontroller, and the flavor we will discuss in the following chapter is the Arduino.

QUESTIONS AND PROJECT IDEAS

1. What is the purpose of a fly-back diode? When do you need it?
2. The TV images are badly disturbed as soon as you turn on a DC motor. What is the cause and how can you alleviate the problem?
3. How do you control the speed of a DC motor if you need to run it at low speed?
4. When do you need to use an H bridge driver to control a DC motor?
5. Can you use a Darlington driver, such as the ULN2003, to control a DC motor with pulse-width modulation?
6. When would you use a DC motor, and when is a model-servo the better option? Discuss!
7. Use a multimeter to determine the resistance between the wires of a stepper motor and identify the internal wiring.
8. Discuss the pros and cons of using unipolar versus bipolar stepper motors.
9. Discuss microstepping and draw the currents in the coils as a function of time.
10. We plan to control a device by low-pass filtering a pulse-width modulated signal. How does the ripple of the control voltage depend on the input impedance of the device? How does the capacitor enter the discussion? What happens if we plan to apply fast changes? Discuss!

11. Investigate what a *thyristor* is and where it is used.
12. Investigate what a *triac* is and where it is used.
13. Investigate what an *IGBT* is and where it is used.
14. Design a small crane to lift cargo from a model boat onto a quay. Which actuators and which sensors do you need? Motivate their use.
15. Design a mechanism that wakes you up with almost certainty. Which actuators do you use and why?

Microcontroller: Arduino

The microcontrollers we select for this book come from the Arduino family, because they are supported by an easy-to-use programming environment that allows us to quickly start developing software. There are limitations, but it is a wonderful system with which to start learning about microcontrollers. Here we need to point out that by “Arduino” we mainly mean the development ecosystem, rather than the processor itself, because that has evolved and comprises a number of different hardware incarnations. In this chapter we will discuss a few of them.

4.1 HARDWARE

The original Arduinos are based on Atmel microcontrollers and we mostly discuss the Arduino UNO. Support for a second family of controllers, based on the ESP8266 microcontroller, was recently integrated into the Arduino development environment. These controllers can be programmed in much the same way as UNOs, but have native wireless support built in. But let’s start with the UNOs.

4.1.1 Arduino UNO

An Arduino UNO is shown in Figure 4.1, where the main component is the ATmega328p microcontroller from Atmel (now MicrochipTM), which is the large chip with 28 legs in the image. It is a controller with 8-bit-wide registers, and operates at a clock frequency of 16 MHz. It has 32 kB RAM memory and 1 kB non-volatile EEPROM memory, which can be used to store persistent data that need to survive tuning off and on the supply voltage. There are three timers on board, which are basically counters that count clock cycles and are programmable to perform some action, once a counter reaches some value. The UNO interacts with its environment through 13 digital input–output (IO) pins, of which most can be configured to be either input or output, and have software-configurable pull-up resistors. Several of the pins are configurable to support I2C, SPI, and RS-232 communication. Moreover, there are six analog input pins. They measure voltages of up to the supply voltage of 5 V. An alternative internal reference voltage provides a 1.1 V reference. All digital and analog pins are routed to pin headers that are visible on both sides of the Arduino printed circuit board (PCB) in Figure 4.1. Furthermore, the built-in hardware RS-232 port is connected to an RS-232-to-USB converter that allows communication and programming from a host computer. There is no WiFi, Bluetooth, or Ethernet support on the UNO board, but extension boards, so-called *shields*, are available for mounting directly onto the pin headers.

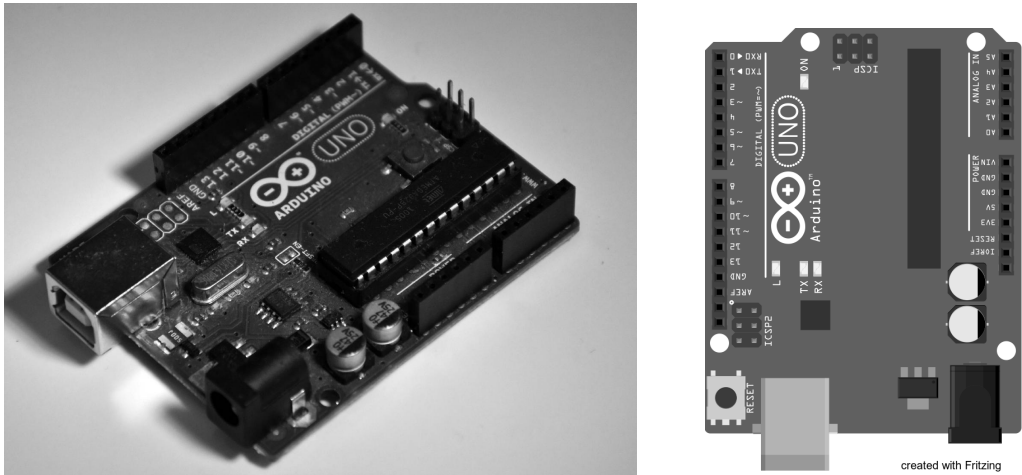


Figure 4.1 An Arduino UNO.

One can describe the Arduino UNO as having the intelligence of a washing machine. It keeps time with the timers, it can sense voltages from, for example, temperature sensors, and it can turn motors or pumps on or off, depending on whether some condition is met. In this way it can also provide the glue logic to interface sensors (analog, I2C, SPI, other) to the host computer, and that is the mode in which we will use the UNO later on. In the next section, however, we address a second microcontroller, the ESP8266.

4.1.2 ESP8266 and NodeMCU

The ESP8266 was released by the Chinese company Espressif in 2014 and quickly became one of the most exciting platforms for Internet of Things (IoT) projects because of its built-in WiFi support, including a small patch antenna. The software development kit (SDK) from Espressif was soon linked to the Arduino software development environment, and today the ESP8266 can be programmed using the Arduino ecosystem as easily as the original Arduinos.

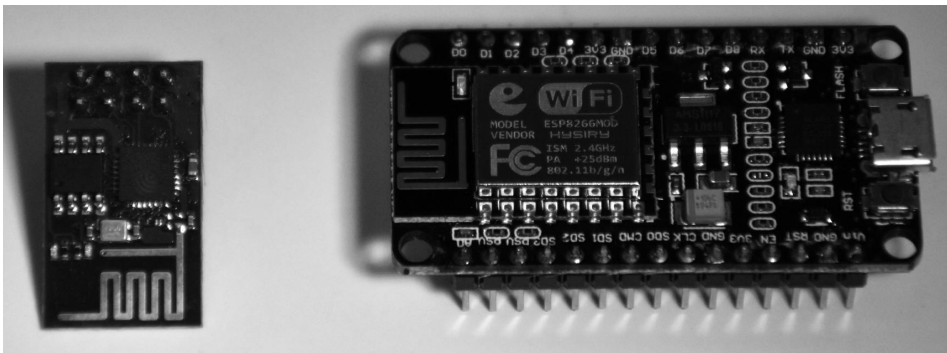


Figure 4.2 The ESP-01 on the left and a NodeMCU on the right.

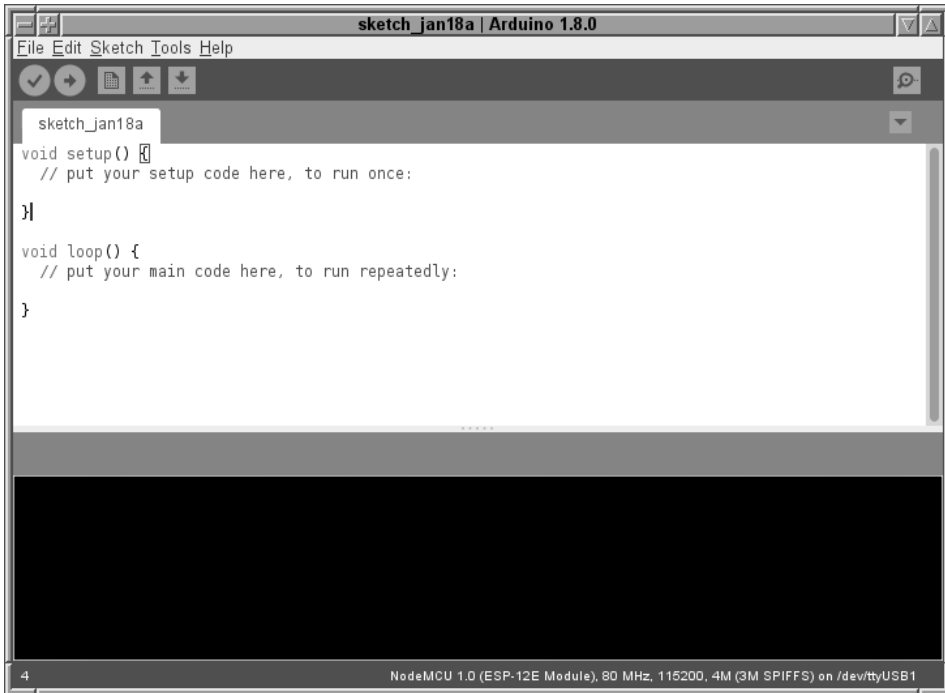


Figure 4.3 The Arduino IDE.

But let us now look at the ESP8266 hardware, which comes in different incarnations. We will focus on the more elaborate NodeMCU board, but also mention the basic ESP-01 in this section. Images of the ESP-01 and NodeMCU hardware are shown at the left and right of Figure 4.2, respectively. The microcontroller on both chips is essentially the same, only the number of internal controller pins that are routed to pins on the circuit boards is different. Internally, the ESP8266 chips feature 32-bit RISC CPUs that normally operate at 80 MHz or 160 MHz, and have 64 kB of RAM for instructions and 96 kB for data. There are 16 general-purpose IO (GPIO) pins that can be configured as input or output pin, as well as a single analog input with a 10-bit ADC. The controller supports I2C and SPI communication as well as RS-232. All this is similar to the Arduino, except it runs faster, and the internal registers are wider (32 instead of 8 bit in the ATmegs), but the really cool feature is the built-in ready-to-use WiFi support. It fulfills the specifications for IEEE 802.11 b/g/n with WPA authentication, which is what most wireless networks use today. So basically, we are able to connect an ESP8266 to any wireless network.

After having introduced the hardware, we need to move on and start programming. This is most easily done using the Arduino IDE (Integrated Development Environment) and we provide a quick-start guide in the following section.

4.2 GETTING STARTED

The Arduino IDE is a software bundle with a size on the order of 100 MB that can be downloaded from

<https://www.arduino.cc>

for the most common computer systems, such as Linux, Mac, or Windows. Basically, we go to the web site, choose the *Download* tab, select our computer type, and download the software package. Then we install the software by following the instructions given on the web site. Once the installation completes, we start the IDE by clicking the icon, or start it from the command line by typing `arduino` followed by `Enter`, which should open the IDE and show us a window similar to the one shown in Figure 4.3. If not, create a “New” *sketch*, which is what Arduino programs are called, by selecting “New” from the “File” menu.

Here we already see the general structure of Arduino programs (or synonymously “sketches”). There is a `setup()` function, which is *executed once*, immediately after power is turned on. In this routine all initialization housekeeping is done, such as defining a pin to be output or input, and configuring the serial line. Once the `setup()` function completes, the `loop()` function is called repetitively, such that once it completes, it is called again and so forth, until power is turned off. Note that the programming language supported by the Arduino IDE is very similar to the C language. There are, however, a number of special extensions to provide access to the specific hardware, such as the ADC.

Now we connect the Arduino UNO to any USB port on the host computer where the Arduino IDE is running, and select Arduino UNO from the **Tools**→**Board** menu. This step tells the IDE for which processor the compiler will generate code, as well as some hardware-specific definitions such as the names of IO pins that we can use when programming. At this point the IDE “knows” what the hardware is, but we still need to tell the IDE to which USB port the UNO is connected. This we do in the **Tools**→**Port** menu, where normally the serial port to which the UNO is connected automatically appears and can be selected. On Linux this often is `/dev/ttyUSB0` or `/dev/ttyACM0`. On a Windows computer it is `COMx` where `x` is some number.

At this point we could start programming the UNO, even though the example program in Figure 4.3 only contains empty functions. But before writing programs for the Arduino, we want to install support files for ESP8266-based microcontrollers. They are easily installed by opening the **File**→**Preferences** menu and adding `http://arduino.esp8266.com/stable/package_esp8266com_index.json` to the “Additional Boards Manager URL” text box and clicking the “OK” button. Then open the **Tools**→**Board** menu and open the “Boards Manager,” and find the “esp8266” platform, select the newest version and click on the “Install” button. Once installation completes, select the ESP platform from the **Tools**→**Board** menu. For the ESP-01 the entry “Generic ESP8266 Module” is a good choice and for the NodeMCU platform it is “NodeMCU 1.0.”

Once the correct platform (UNO, plain ESP8266, or NodeMCU) is selected, we are ready to write programs. In the following we will tacitly assume an UNO is connected, unless stated otherwise. So, now we are ready to write programs (sketches) and download them to the hardware.

4.3 HELLO WORLD, BLINK

The first sketch we consider is the standard for almost any new controller, namely to cause an LED to blink. Examples to interface hardware such as sensors can be found in the **File**→**Examples** menu, where the example to blink an LED is located under the **01.Basics**→**Blink** menu item. Once selected, a new window opens with the example code. Figure 4.4 shows the bare code, without many comment lines that are preceded by `//`. It is reproduced here.

```
void setup() {
    pinMode(LED_BUILTIN, OUTPUT);
```

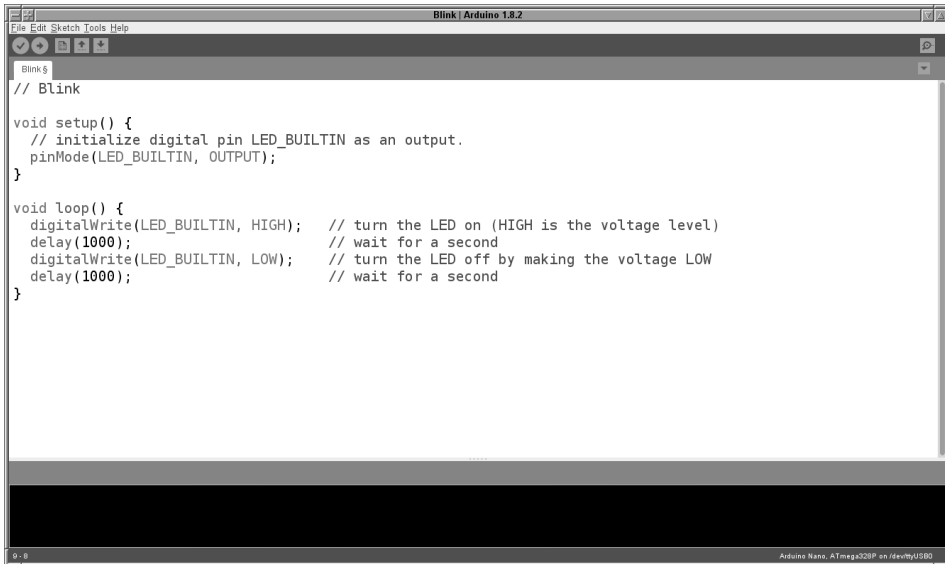


Figure 4.4 The “Blink” sketch.

```

}
void loop() {
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on
  delay(1000);                     // wait for a second
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off
  delay(1000);                     // wait for a second
}

```

Let us step through this sketch one line at a time. In the `setup()` function, we call the `pinMode(Pin,What)` function, which declares that the pin called `LED_BUILTIN` (which is pin 13 on a UNO) will be used as output pin. That is all the initializing we do in the `setup()` function. In the `loop()` we call `digitalWrite(Pin,State)`, which causes the controller to put 5 V onto the specified pin if the requested state is `HIGH` and 0 V if the requested state is `LOW`. The latter happens two lines later. In between the changes to the output pin, we tell the controller to wait for a specified number of milliseconds in the `delay(time_in_ms)` function. So, all the loop function does is turn on the LED on pin 13, wait for 1000 ms or 1 s, turn it off, wait again for 1 s, and then start all over again. Note that all commands are terminated by a semicolon, as is customary in the C-language.

Once we are satisfied with our program, we can compile it and check that the syntax is correct and we did not forget any semicolons. Pressing the checkmark symbol just below the **File** menu entry compiles the program and reports progress in the status window below the program window. If the compilation completes without errors and the UNO board is connected to a USB port, and it is selected in the **Tools**→**Port** menu entry, we can download the program to the UNO by pressing the → button located to the right of the compile button. After a few seconds, a small LED on the UNO board should start to blink once per second. Now we can change, for example, the delay time in the sketch, compile and download again, and observe whether the LED blinks according to the newly specified times.

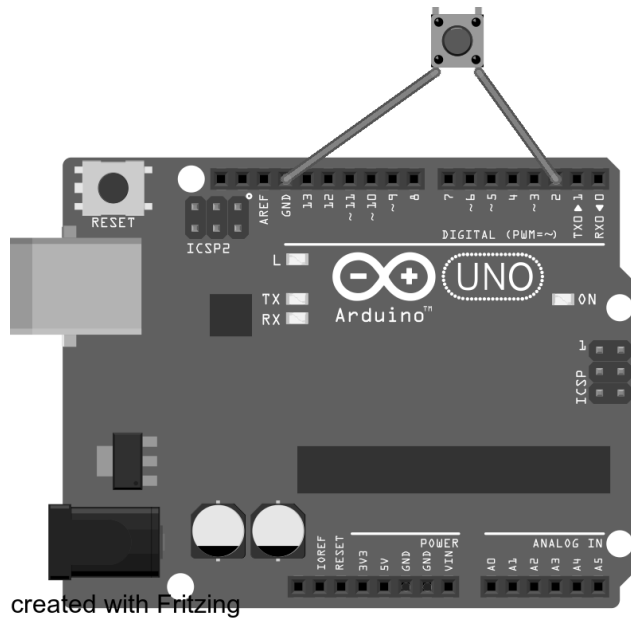


Figure 4.5 Interfacing a button to the Arduino.

Here we only used very few commands, such as `pinMode` or `digitalWrite`, but there are many more, and checking the Reference section on the Arduino web site at <https://www.arduino.cc/en/Reference/HomePage> is very inspirational. Moreover, clicking on the different commands opens a new page, with explanations and examples on how to use the functions. A further source of wisdom is the “Learning” tab from the menu list on the top of <https://www.arduino.cc/>, which offers a large number of tutorials and examples. Keeping these resources in mind will help us to quickly locate information to make things work. Moreover, there are a large number of books on various Arduino projects, and compendia of tips and tricks such as [19].

After the first step to make the microcontroller follow our bidding, we move on and read sensors.

4.4 INTERFACING SENSORS

Here, the main task of the Arduino microcontroller is to read sensors, with their different, often idiosyncratic interfaces, and convert the measured values to a standard representation that is communicated to a host computer. We start with the simplest sensor, a button or a switch.

4.4.1 Button

In Figure 4.5, we show how to connect a button between ground and pin 2 of the Arduino, such that pressing the button will cause pin 2 to read 0 V or LOW. The following listing shows a sketch that causes the LED on pin 13 to light up, if the button is pressed.

```
// button_press, V. Ziemann, 161013
```

```

void setup() {
  pinMode(2,INPUT_PULLUP);    // button, default=HIGH
  pinMode(13,OUTPUT);         // LED
}
void loop() {
  if (digitalRead(2)==LOW) {
    digitalWrite(13,HIGH);
  } else {
    digitalWrite(13,LOW);
  }
  delay(10);
}

```

The program follows the normal scheme of initializing the hardware in the `setup()` function. We first declare that pin 2 is an input pin, and we enable the internal pull-up resistor that causes a well-defined “high” state on the pin if no button is pressed. Pin 13 with the LED is declared as output, so we can turn it on and off from within the sketch. In the `loop()` function, we check the state of the button by calling the `digitalRead(Pin)` function, and test whether pin 2 is LOW. If this is the case (note the double equal sign `==` in the comparison), the LED on pin 13 is turned on by calling `digitalWrite(13,HIGH)`. If pin 2 is found not to be LOW, the LED on pin 13 is turned off with `digitalWrite(13,LOW)`; Note the braces `{` and `}` and their use to define blocks of code in the `if (..) { } else { }` construction. After the `if` statement, a short `delay()` ensures that mechanical button bounces are ignored and that the processor has a little time for its internal affairs. This is not absolutely necessary, but good style.

Sensing the state of switches that indicate open doors or windows follows exactly the same scheme, and we can sense up to 13 different switches (if we do not use the LED on pin 13) and react in some manner.

4.4.2 Analog input

In an earlier chapter, we saw that some sensors require us to measure an analog voltage because they change their resistance. We place them in a voltage divider and then read the voltage change on the central tap, as shown on the left in Figure 2.1. Others, such as the LM35 temperature sensor, directly produce a voltage and in this section we show how to interface these sensors to the Arduino. First, we want to measure a voltage from one of the analog input pins; here we use A0. The left of Figure 4.6 shows how to connect a potentiometer as a variable voltage divider to the Arduino. One end of the potentiometer is connected to ground, the other end to the supply voltage, and the wiper with the variable tap of the potentiometer is connected to pin A0. Turning the axis of the potentiometer causes the voltage on the center tap to vary between ground and the 5 V supply voltage.

The sketch to read the voltage and report it via the serial cable (that is connected to the USB port) to the host computer is shown next.

```

// Analog and serial communication, V. Ziemann, 161130
int inp,val;
void setup() {
  Serial.begin(9600);  // baud rate
}
void loop() {
  if (Serial.available()) {

```

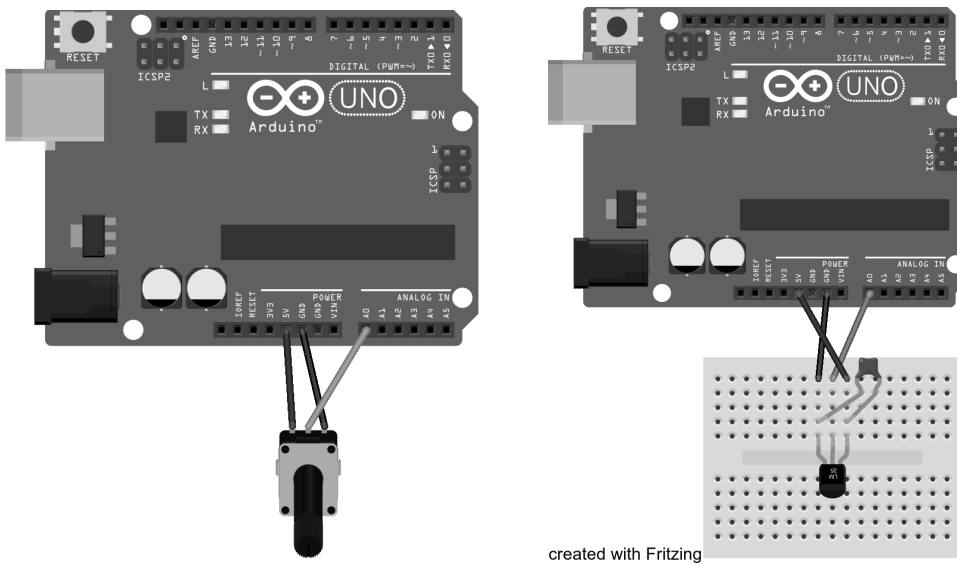



Figure 4.6 Interfacing a potentiometer (left) or an LM35 temperature sensor (right) to the Arduino.

```

    inp=Serial.read();           // read character from serial
    val=analogRead(0);           // read analog
    Serial.print("Value is ");  // and report back
    Serial.println(val);
  }
  delay(50);                    // wait 50 ms
}
```

Since we want to temporarily store integer values, we have to allocate memory for the variables, which is done in the line containing `int inp, val;` as it declares two integer variables named `inp` and `val`. In the `setup()` function, we declare that we want to use the built-in serial (RS-232) hardware port with a speed of 9600 baud, which is about 1000 characters per second. The receive (RX) and transmit (TX) lines of the serial port are connected to pin 0 and pin 1, but also routed to the RS232-USB converter on the UNO board that is connected to the host computer. In the `loop()` function, we first check whether some communication from the host computer has arrived, and if that is the case, we read the character with the `Serial.read();` command, but do nothing further with it. In the next line we read the analog pin A0 using the built-in ADC, and store the value in the integer variable `val`. The value returned from the 10-bit ADC is a number between 0 and 1023 ($= 2^{10} - 1$) and *not* the voltage. Later we will show how to convert this value to a voltage. The result of the measurement is then sent via the serial line by `Serial.print()` and `Serial.println()` commands. The difference between the two is that the former does not send a carriage return character at the end of the message, and the latter does. After the `if() { }` statement at the end of the loop, again a small delay is added. Now we can compile and download the sketch to the Arduino, unless some syntax error crept in. If that is the case, we need to check for missing semicolons or spelling errors, especially the “camelHump-spelling” and capital letters in the middle of the commands. Care is advisable, but the IDE

uses syntax highlighting, which makes such errors reasonably easy to spot. After all errors are corrected, we download the sketch to the hardware.

Once the sketch is running on the Arduino, we want to test it, and for that we need to have access to the other end of the serial communication channel on the host computer. The easiest way is to open the “Serial Monitor” by pressing the icon that looks like a looking glass at the top right in the Arduino IDE. Clicking on it opens a simple terminal program that has an input text box at the top. Enter any character and press “Enter.” If all is well, the Arduino should report with a line **Value is nnnn** in the text box below the entry box. Now turn the potentiometer and enter a character followed by “Enter,” and see how the returned value changes.

So far, so good, but would it not be nice if we could read out more analog ports and even ask for which ones to read. This task is addressed in the following sketch. It is rather similar to the previous one and we only discuss the new features.

```
// Analog and Serial communication, v2, V. Ziemann, 161130
int inp,val;
void setup() {
    Serial.begin(9600); // baud rate
}
void loop() {
    if (Serial.available()) {
        inp=Serial.read(); // read serial line
        if (inp=='s') {
            val=analogRead(0); // read analog
            Serial.print("A0= "); // and report back
            Serial.println(val);
        } else if (inp=='t') {
            val=analogRead(1);
            Serial.print("A1= "); Serial.println(val);
        } else {
            Serial.println("unknown command");
        }
    }
    delay(50); // wait 50 ms
}
```

In this sketch we actually test which character was sent on the serial line, and depending on whether it is an “s” or a “t” the value from analog pin A0 or A1 is returned. The characters sent back to the host via the serial line contain both the pin read and the value. If a character different from “s” or “t” is received by the Arduino, the string “unknown command” is returned via the serial line. In this way, we build a simple query-response protocol to make the Arduino do different things, depending on the character we send.

In the next example we assume that an LM35 temperature sensor is connected to analog pin A0 in the way shown on the right in Figure 4.6. Note that we add a 100 nF capacitor across the positive supply voltage and ground pins of the LM35, to increase the stability of the circuit. Note that it is good practice to add such a decoupling or bypass capacitor close to the power pins of each chip in a circuit. In the following, we often do not show these capacitors on the circuit diagrams in order to make them less cluttered, but the capacitors should be included in the hardware. The software to interface the LM35 is slightly more elaborate than in previous examples. It allows us to send somewhat descriptive multicharacter commands to the Arduino and receive a response that echos the request followed by

the value. The basic idea is that we request a parameter, say A0, by sending the parameter name with a question-mark appended, such that the request looks like this: A0?. The Arduino subsequently replies with A0 <value>. In the following sketch we implement this, but also that the Arduino first reads a full line, in the sense that it waits for a carriage-return and a line-feed character before responding.

```
// query-response, V. Ziemann, 161013
int val;
float temp;
char line[30];
void setup() {
    analogReference(INTERNAL); // 1.1V internal ADC reference
    Serial.begin(9600);
    while(!Serial){;}
}
void loop() {
    if(Serial.available()) {
        Serial.readStringUntil('\n').toCharArray(line,30);
        if(strstr(line,"A0?")==line) {
            val=analogRead(0);
            Serial.print("A0 "); Serial.println(val);
        } else if (strstr(line,"A1?")==line) {
            val=analogRead(1);
            Serial.print("A1 "); Serial.println(val);
        } else if (strstr(line,"T?")==line) {
            temp=1.1*100*analogRead(0)/1023;
            Serial.print("T "); Serial.println(temp);
        } else {
            Serial.println("unknown");
        }
    }
}
```

In the first few lines, we declare the variables, and in particular, a 30-character buffer `line` to hold the multicharacter query. In the `setup()` function, we use the internal 1.1 V reference voltage for the ADC and initialize the Serial communication channel to operate at 9600 baud. The `while(!Serial)` statement waits for the initialization of the Serial library to complete, and is only needed on some newer Arduinos. We include it just for safety. In the `loop()` function, we wait for characters to be available before reading an entire line until the end-of-line character `\n`. The `readStringUntil` returns a `String` but we want to use a character array instead, and need to convert the type in the appended `toCharArray` method. At this point the character array `line` contains the request that was sent from the host computer to the Arduino; for example, A0?. The following construction of `if (strstr(line,"A0?")==line)` and similar commands checks whether `line` starts with the characters A0? by comparing the pointer to the first occurrence of A0? in the array `line[]` to the pointer to the start of `line[]`. The commands that follow are executed provided the result evaluates to true, or else the subsequent comparison in the `else if` sequence is tested. In the example, we test for "A0?", "A1?", and "T?" to reply with the raw values of the analog pins 0 and 1 or with the value from analog pin 0 converted to degrees Celsius. Consult the discussion in Section 2.1.2 regarding the conversion coefficient. If a command arrives, and it is not in the list, the Arduino replies with `unknown command`. The way to

specify the query with an appended question mark and the reply to echo the query with value appended is borrowed from the SCPI command-language that is commonly used to remotely control measurement equipment via GPIB or VXI.

As a final example, we discuss a simple *temperature logger* that again uses the hardware setup as shown on the right side of Figure 4.6, but sends the temperature value continuously to the host computer via the serial line. The sketch is shown here:

```
// ultra-simple LM35 temperature logger, V. Ziemann, 161201
float temp;
void setup() {
  analogReference(INTERNAL); // 1.1V internal ADC reference
  Serial.begin(9600);
  while (!Serial) {} // wait for serial to initialize
}
void loop() {
  temp=1.1*100*analogRead(0)/1023.0;
  Serial.println(temp);
  delay(1000);
}
```

where the `setup()` function equals that from the previous example, but the `loop()` function measures the analog pin and converts the digital word to the temperature, and sends that to the host computer. Then it waits 1000 ms before repeating the same. In the Serial Monitor, that can be started by clicking on the button at the top right in the IDE; we see that the temperature value in Celsius is reported one per second in the lower text box provided the baud rate selected in the selection box at the lower right matches the baud rate specified in the sketch.

In the Arduino IDE, however, there is also a graphical plotting tool available. We select it by choosing the **Tools**→**Serial Plotter** menu item. This opens a window and shows the values received on the serial line in graphic format, provided the baud rate is selected to match the one specified in the sketch. Note that the **Serial Plotter** can show several measurement values simultaneously as traces in different colors. This functionality, built in to the Arduino IDE, may serve as a simple logger for temperatures or other measured values, if a quick solution to record some values as a function of time is needed.

So far, we managed to read analog voltages, and even devised a simple protocol to report the measurement values to the host computer. In the next section, we interface I2C-based sensors to the microcontroller, and use the same query-response protocol to communicate with the host.

4.4.3 I2C

And we continue with another temperature sensor, but a special one that is used by nurses in hospitals. It is based on a contact-free thermometer that takes your temperature by pointing a gadget into your inner ear. An example is the MLX90614 infrared thermometer shown on the right in Figure 2.9. It uses a thermopile to deduce the temperature from the picked-up infrared radiation and it reports the temperature to our Arduino via an I2C interface. Here we point out that the MLX90614 comes in versions for 5 V and 3.3 V supply voltage. In this example we use the 5 V version, labeled AAA. The device has four pins—check the datasheet to find out which pin does what—for ground, supply voltage, and the I2C lines SDA as SCL. The latter two are connected to pin A4 and A5 on the UNO because the I2C data and clock lines are routed to the same output pins as the analog pins A4 and A5. These pins cannot

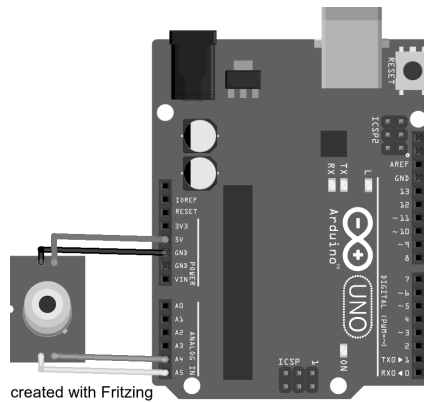


Figure 4.7 Connecting the MLX90614 contact-free thermometer to an Arduino UNO.

be used for analog measurements, in case they are used for I2C connectivity. We show how to connect the sensor in Figure 4.7, and illustrate how to talk to it in the following sketch.

```
// Read MLX90614 IRthermometer, V. Ziemann, 170717
#include <Wire.h>
const int MLX90614=0x5A; // I2C address
float getTemperature(uint8_t addr) { //.....getTemperature
    uint16_t val;
    uint8_t crc;
    Wire.beginTransmission(MLX90614);
    Wire.write(addr); // address
    Wire.endTransmission(false);
    Wire.requestFrom(MLX90614,3);
    val = Wire.read();
    val |= (Wire.read()<<8);
    crc=Wire.read(); // not used
    return -273.15+0.02*(float)val;
}
void setup() { //.....setup
    Serial.begin(9600);
    Wire.begin();
}
void loop() { //.....loop
    float Ta=getTemperature(0x06); // address for ambient temp
    float To=getTemperature(0x07); // address for object1 temp
    Serial.print(Ta); Serial.print("\t"); Serial.println(To);
    delay(1000);
}
```

First we have to include support for the I2C functionality by including the `Wire.h` header file, which also causes the compiler and linker to include the corresponding libraries. After we define the I2C address `0x5A` of the sensor, which we find in the datasheet, we encapsulate the I2C communication in a separate function called `getTemperature()`. It receives the

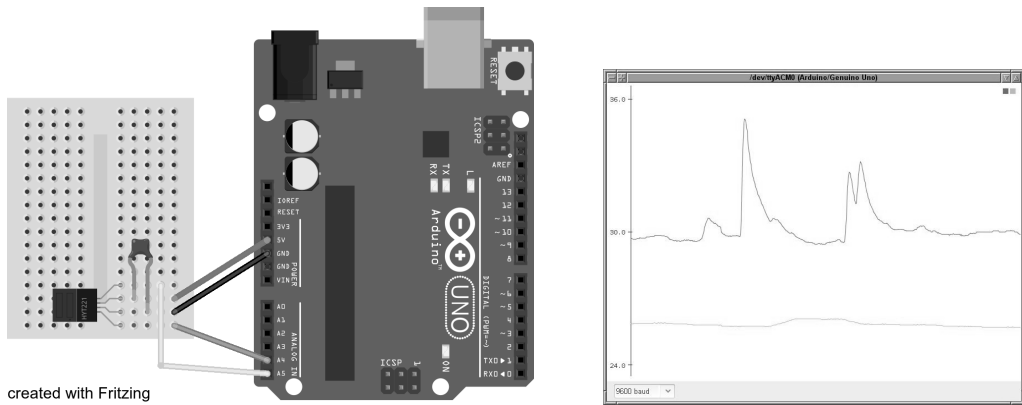


Figure 4.8 Connecting the HYT-221 humidity sensor to the Arduino (left) and using the serial plotter to display the humidity and temperature measurements (right).

register address of the I2C device as input parameter and returns the associate temperature, properly scaled to degrees Celsius. The device also returns a byte that allows us to determine transmission errors, but we do not use that feature in this simple example. In the `setup()` function we only initialize Serial line and I2C communication. In the `loop()` function we use the `getTemperature()` function to retrieve the ambient temperature from address `0x06` and the object temperature from address `0x07`, as described in the datasheet, and print both temperatures formatted with a tabulator to the Serial line. The entire process is then repeated after waiting 1000ms. We can use the Serial monitor or plotter built into the Arduino IDE in order to view or display the results.

Several other sensors, often related to environmental parameters such as humidity or barometric pressure, also report the temperature as a by-product, because it is internally needed to calibrate the reported primary measurement value. For the next sensor, the HYT-221, this is the humidity. Connecting the sensor to the Arduino is illustrated on the left in Figure 4.8. From the datasheet we know that the four pins of the sensor, when looking towards the front of the sensor, are SDA, GND, VDD, and SCL. The pins for GND and VDD are connected to the GND and 5V pins on the Arduino, and SDA to A4 and SCL to A5 in the same way we did for the temperature sensor in the previous example. In the following sketch, we read the HYT sensor, and send the measurement data of humidity and temperature via the serial line to the host computer.

```
// HYT221 humidity sensor, V. Ziemann, 161203
#include <Wire.h>
const int HYT=0x28; // I2C address
void setup() { //.....setup
  Serial.begin(9600); while (!Serial) {}
  Wire.begin();
}
void loop() { //.....loop
  int b1,b2,b3,b4,raw;
  double humidity,temp;
  Wire.beginTransmission(HYT);
  Wire.requestFrom(HYT,4);
```

```

delay(50);
if (Wire.available()==4) {
    b1=Wire.read(); b2=Wire.read(); // humidity rawdata
    b3=Wire.read(); b4=Wire.read(); // temperature rawdata
    Wire.endTransmission(true);
}
raw=(256*b1+b2) & 0x3FFF;          // humidity
humidity=100.0*raw/16384.0;
raw=((256*b3+b4) & 0xFFFC)/4;      // temperature
temp=165.0*raw/16384.0-40.0;
Serial.print(humidity); Serial.print("\t"); Serial.println(temp);
delay(1000);
}

```

First we include the I2C functionality with `Wire.h`, and define a variable `HTY` that contains the I2C address `0x28`. In the `setup()` function, we first initialize the serial communication and then the I2C functionality with the call to the `Wire.begin()` function. In the `loop()` function we first declare a number of auxiliary integer variables and double-precision float variables to hold the measurement data before calling the `Wire.beginTransmission()` function with the address `HTY` as argument, and tell the Arduino to fetch four bytes as response from the sensor in the following line. After a short delay, we check whether the four bytes have arrived, with the `Wire.available()` function, and in case they are available, we read them into the variables `b1`, ..., `b4` before closing the transmission. The first two bytes contain the raw data for the humidity, and the next two for the temperature measurement. In the Arduino sketch, we follow the instructions from the HYT-221 datasheet to convert the raw data to the physical quantities. For the humidity, the datasheet tells us that the first received byte `b1` is the most significant byte of a two-byte word and `b2` is the least significant byte. Therefore, we can reconstruct the word by `256*b1+b2`. But we also have to throw away (mask) the two highest bits, which we do by a logical `and (&)` with a binary value that has ones everywhere except at the two highest bits, which is `& 0x3FFF`. Once we have the raw value, we again follow the datasheet and scale by `100/16384` to obtain the humidity in percent. For the temperature, we follow a similar procedure that is described in the HYT-221 datasheet. Finally, we send the humidity and temperature values to the host computer via the serial line and wait a while before repeating the measurement. On the right of Figure 4.8 we show the data using the Serial-plotter in the Arduino IDE. The upper trace shows the humidity, and the two positive excursions are due to exhaling on to the sensor. The lower trace shows the temperature, which we briefly cause to increase by placing a lamp close to the sensor.

The next environmental sensor—we use it later in a weather station—is the barometric pressure sensor BMP180. Like the previous two sensors, it communicates with the Arduino via the I2C bus, and needs four lines for `SCL`, `SDA`, `GND`, and `VDD`. Note that we can connect this sensor in parallel to the previous sensors using the same four pins on the Arduino because each sensor has its unique address. For this sensor, we must be careful with the supply voltage, because higher voltages than 3.3 V can damage it. Therefore, we use the 3.3 V supply voltage available on the Arduino to power the sensor. The I2C pins are, however, tolerant to higher voltages. As mentioned in Section 2.3.3, each individual sensor is factory calibrated, with calibration constants stored on an EEPROM on the sensor chip. In order to convert raw ADC values, reported from the device, to pressure and temperature in conventional units, we need to follow the instructions in the datasheet. The following

code implements the basic functionality to retrieve the pressure and the temperature from the sensor.

```
// BMP180, V. Ziemann, 170805
#include <Wire.h>
const int BMP=0x77; // I2C address
int16_t ac1,ac2,ac3,b1,b2,mb,mc,md;
uint16_t ac4,ac5,ac6;
void I2Cwrite(uint8_t addr, uint8_t reg, uint8_t val) {
    Wire.beginTransmission(addr); // address
    Wire.write(reg); // register
    Wire.write(val); // write value
    Wire.endTransmission(true);
}
uint8_t I2Cread(uint8_t addr, uint8_t reg) {
    Wire.beginTransmission(addr); // address
    Wire.write(reg); // register
    Wire.endTransmission(false);
    Wire.requestFrom(addr,1); // request 1 byte
    return Wire.read(); // read and return
}
bool bmp180_init() { //.....bmp180_init
    Wire.beginTransmission(BMP); // read EEPROM
    Wire.write(0xAA); // start register address
    Wire.endTransmission(false);
    if (Wire.requestFrom(BMP,22)) {
        ac1=Wire.read()<<8 | Wire.read();
        ac2=Wire.read()<<8 | Wire.read();
        ac3=Wire.read()<<8 | Wire.read();
        ac4=Wire.read()<<8 | Wire.read();
        ac5=Wire.read()<<8 | Wire.read();
        ac6=Wire.read()<<8 | Wire.read();
        b1=Wire.read()<<8 | Wire.read();
        b2=Wire.read()<<8 | Wire.read();
        mb=Wire.read()<<8 | Wire.read();
        mc=Wire.read()<<8 | Wire.read();
        md=Wire.read()<<8 | Wire.read();
        Wire.endTransmission();
        return true;
    }
    return false;
}
float bmp180_getdata(float *T) { //.....bmp180_getdata
    float ut,x1,x2,x3,b3,b4,b5,b6,b7,tt,p;
    uint16_t up;
    I2Cwrite(BMP,0xF4,0x2E); // request temperature first
    delay(5);
    ut=(float)(I2Cread(BMP,0xF6) << 8 | I2Cread(BMP,0xF7));
    x1=(ut-ac6)*ac5/32768.0;
    x2=mc*2048.0/(x1+md);
```



```

    b5=x1+x2;
    *T=(b5+8.0)/160.0;
    I2Cwrite(BMP,0xF4,0x34); // request pressure measurement
    delay(5);                // with oss=0, wait 5 ms, table 8
    up=I2Cread(BMP,0xF6)<<8 | I2Cread(BMP,0xF7);
    b6=b5-4000;
    x1=(b2*(b6*b6/4096.0))/2048.0;
    x2=ac2*b6/2048.0;
    x3=x1+x2;
    b3=(((long)ac1*4+x3))+2)/4;
    x1=ac3*b6/8192;
    x2=(b1*(b6*b6/4096))/65536;
    x3=((x1+x2)+2)/4;
    b4=ac4*(x3+32768)/32768;
    b7=(up-b3)*(50000);
    if (b7< 0x80000000) {p=b7*2/b4;} else {p=(b7/b4)*2;}
    x1=(p/256)*(p/256);
    x1=(x1*3038)/65536;
    x2=(-7357*p)/65536;
    p=p+(x1+x2+3791)/16;
    return 0.01*p;
}

void setup() { //.....setup
    Serial.begin(9600); while (!Serial) {}
    Wire.begin();
    if (!bmp180_init()) Serial.println("No sensor found!");
}

void loop() { //.....loop
    float pressure, temperature;
    pressure=bmp180_getdata(&temperature);
    Serial.print(temperature); Serial.print("\t");
    Serial.println(pressure);
    delay(2000);
}

```

At the top of the sketch we include the I2C functionality, define a constant with the default I2C address of the BMP180 sensor, and declare the variables that will contain the calibration constants. Then we define two convenience functions to read and write a byte from a device at I2C address `addr`, register `reg`, and value `val`. In the `bmp180_setup()` function, we only read the calibration constants from the internal EEPROM. The addresses start at `0xAA` and there are 11 constants with two bytes each to read. We therefore retrieve 22 bytes and immediately assemble the constants by joining two bytes to a 16-bit word, some of them signed and some unsigned, as specified in the datasheet. The initialization function returns `TRUE` if the communication is successful and `FALSE` otherwise. The `bpm180_getdata()` function returns the pressure and the temperature. First, we declare temporary variables, before requesting a temperature measurement by writing `0x2E` to the control register `0xF4`, wait 5 ms for the measurement to complete, and then retrieve the raw reading from the output registers of the ADC, `0xF6`, and `0xF7`, and assemble a 16-bit word. The next four lines with the temporary variables `x1`, `x2`, `b5` are copied straight from the datasheet to convert the raw reading to the temperature in Celsius, which we store in the variable `T`. Next we request

a pressure measurement with low precision by writing 0x34 to the control register 0xF4, wait a short while, and assemble the raw readings from registers 0xF6 and 0xF7 to form the 16-bit word up. The following intermediary calculations are again copied straight from the datasheet, and at the end we return the pressure *p* in Pascal. Finally, we convert to mbar by multiplying the value with 0.01, and return it. In the `setup()` function, we initialize the serial line, the I2C bus, and the sensor by calling `bmp180_init()`, and issue a warning if no sensor is found. In the `loop()` function we only call the `bmp_getdata()` function and display the values before waiting a short while and repeating this indefinitely.

In the previous examples with the humidity sensor and thermometer, we coded the I2C communication by hand, but often there are libraries available that encapsulate the details of the communication and the often-arcaic prescriptions from the datasheets to convert the raw data to physical quantities. For the BMP180 there are several libraries available, and we will use the `SFE_BMP180` library. We install the library by first downloading a zip file with library and examples and unpacking its contents in the `Arduino/libraries` subdirectory. In this case the library resides in the `Arduino/libraries/SFE_BMP180/` directory. After restarting the Arduino IDE, the library and examples of how to use it are ready to use. The example equivalent to the previous one is the following.

```
// BMP-180 barometric pressure sensor, V. Ziemann, 161204
#include <SFE_BMP180.h>
#include <Wire.h>
SFE_BMP180 pressure;
void setup() {
    Serial.begin(9600); while (!Serial) {}
    if (!pressure.begin()) {Serial.println("BMP180 failed!");}
}
void loop() {
    char status;
    double T,P;
    status = pressure.startTemperature(); delay(status);
    status = pressure.getTemperature(T);
    status = pressure.startPressure(3); delay(status);
    status = pressure.getPressure(P,T);
    Serial.print(T,2); Serial.print("\t"); Serial.println(P,2);
    delay(2000);
}
```

This example is much shorter than the previous one, because most of the details of the I2C communication and data conversion from raw to physical quantities are hidden in the library. We include the library by placing the include directive for the header files near the top of the sketch, both for the `SFE_BMP180` library and the I2C library `Wire.h` needed for the communication. In the next line, a variable `pressure` of type `SFE_BMP180` is declared, which is the quantity (an object) that encapsulates the functionality provided by the BMP180 sensor. In the `setup()` function, first the serial and then the communication with the BMP180 sensor is initialized, and an error message is printed if the initialization fails. The `loop()` function is particularly simple. After declaring some variables, the `pressure` sensor is requested to start a temperature measurement, which returns a status that contains the waiting time until the measurement data are available and ready to be read with the `getTemperature` function. Once the temperature is known, we start the pressure measurement, which returns the waiting time as the status until the data can be fetched with the `getPressure` function. Once the temperature and pressure are known, they are

sent via the serial line to the host computer. After a small delay, the whole procedure is repeated. In this very basic example, we only show the basic functionality, and omit error checking of the status. In any program used for serious data acquisition, all available status information should of course be checked.

We can use the next sensor to determine the acceleration and speed of rotation on carousel rides at a fair. Incidentally, the same sensor is used in smart phones to determine whether they are held upright or sideways. This device, the *MPU-6050*, contains separate sensors for acceleration in three dimensions and for angular velocity around three axes. It uses a standard I2C with default address 0x68. In the following sketch, we explain how to interface it to the Arduino.

```
// MPU6050, V. Ziemann, 170726
#include <Wire.h>
const int MPU6050=0x68;
void I2Cwrite(int addr, int reg, int value) { //...I2Cwrite
    Wire.beginTransmission(addr);
    Wire.write(reg);
    Wire.write(value & 0xFF);
    Wire.endTransmission(true);
}
uint8_t I2Cread(int addr, int reg) { //.....I2Cread
    Wire.beginTransmission(addr);
    Wire.write(reg);
    Wire.endTransmission(false);
    Wire.requestFrom(addr,1);
    return Wire.read();
}
void mpu6050_init() { //.....mpu6050_init
    Wire.begin();
    int whoami=I2Cread(MPU6050,0x75);
    if (whoami!=0x68) {
        Serial.println("No MPU6050 found!"); while (1) {yield();}
    }
    I2Cwrite(MPU6050,0x6B,0); // wake up device
    I2Cwrite(MPU6050,0x1B,0); // Gyro config, FS=0
    I2Cwrite(MPU6050,0x1C,0); // Acc config, AFS=0
}
void mpu6050_read_float(int addr, float fdata[7]) { //...read_float
    int16_t intval;
    Wire.beginTransmission(addr);
    Wire.write(0x3B);
    Wire.endTransmission(false);
    Wire.requestFrom(addr,14);
    intval=Wire.read()<<8 | Wire.read(); fdata[0]=intval/16.384; //ax
    intval=Wire.read()<<8 | Wire.read(); fdata[1]=intval/16.384; //ay
    intval=Wire.read()<<8 | Wire.read(); fdata[2]=intval/16.384; //ay
    intval=Wire.read()<<8 | Wire.read(); fdata[3]=intval/340.0+36.53; //T
    intval=Wire.read()<<8 | Wire.read(); fdata[4]=intval/131.0; //gx
    intval=Wire.read()<<8 | Wire.read(); fdata[5]=intval/131.0; //gy
    intval=Wire.read()<<8 | Wire.read(); fdata[6]=intval/131.0; //gz
```

```

    Wire.endTransmission();
}
void setup() { //.....setup
    Serial.begin(115200);
    while (!Serial) {yield();}
    mpu6050_init();
}
void loop() { //.....loop
    float fdata[7];
    mpu6050_read_float(MPU6050,fdata);
    Serial.print(fdata[0],0); Serial.print("\t");
    Serial.print(fdata[1],0); Serial.print("\t");
    Serial.print(fdata[2],0); Serial.print("\t");
    Serial.print(fdata[4],0); Serial.print("\t");
    Serial.print(fdata[5],0); Serial.print("\t");
    Serial.print(fdata[6],0); Serial.print("\t");
    Serial.println(fdata[3]);
    delay(1000);
}

```

Here we first include I2C support by loading the `Wire.h` header file, and define a symbolic name for the I2C-address of the device. Next we define the two convenience functions to write and read a single byte from the device, which we already encountered in a previous example in this section. In the `mpu6050_init()` function, we initialize the I2C library and check for the `whoami` byte at address `0x75`. According to the datasheet, a value of `0x68` indicates that the device is present. Then we need to write to address `0x6B` in order to wake up the device, and then configure the gyro and accelerometer to use the standard ranges of $\pm 250^\circ/\text{s}$ and $\pm 2g$. The `mpu6050_read_float()` function requests 14 consecutive bytes from the device, which encodes the three gyro sensors, three accelerometers, and temperature with two bytes each. Here `0x3B` is the starting address of the range we intend to read. We then convert the corresponding bytes to floating point values. The first three values in the array `fdata[]` are the accelerations in the three spatial directions, converted to units of $10^{-3}g$, where g is the gravitational acceleration on the surface of the earth. The fourth entry is the temperature that is converted from raw ADC readings to degrees Celsius by conversion constants found in the datasheet. The last three entries contain the rate of rotation around three axes, converted to degrees per second. Again the conversion constant is taken from the datasheet. The `setup()` function only initializes the serial line and the MPU6050. The `loop()` function retrieves the data from the device by using the function we defined earlier, and displays the values in a single line. We note in passing that there are libraries available that encapsulate the details of the communication and conversion, and therefore simplify the code significantly, albeit at the expense of reduced transparency and flexibility.

Occasionally the number of input or output pins on the microcontroller is insufficient for a certain project, but luckily there are integrated circuits that help to extend the number of IO pins. One of these is the MCP23017, a device that provides 16 IO pins, each of them freely configurable as an input or output pin. The device has an I2C interface, such that it is equally easy to address as the other examples in this section, and it can even share the same I2C lines, SCL and SDA, with other devices, because it has its own address: `0x20`. Should an address-conflict with another device arise, it can be changed to any value in the range from `0x20` to `0x27` by three address pins that need to be set to either ground or supply

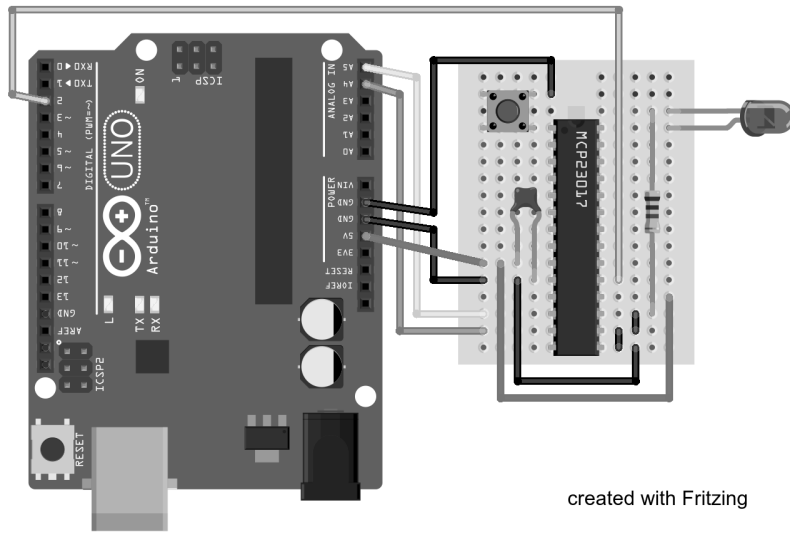


Figure 4.9 Connecting the MCP23017 IO-extender to the UNO.

voltage. This also implies that we can place eight copies of an MCP23017 on the same bus and can use 128 extra IO pins. In the following example, we are less ambitious, and use one chip only. We configure it such that eight pins are output and eight pins are input. The chip has the attractive feature that it can generate an interrupt if an input pin changes its value. We will use that feature, despite having to use one extra pin on the microcontroller, because it relieves the controller of continuously checking whether any input pin on the MCP23017 has changed. The controller only needs to determine which pin has changed, once a change has occurred. In Figure 4.9 we show the circuit with an Arduino UNO interfaced to one MCP23017. On the extender, we connect one pin to an LED that we want to turn on and off. Another pin is connected to a button, and we want to determine whether it is pressed or not.

In the circuit shown in Figure 4.9 we see the UNO on the left and a small solderless breadboard with the extender on the right. Wires connect the breadboard to the 5 V and ground terminals of the UNO, and the I2C lines for SDA and for SCL are connected to pins A4 and A5, respectively, just as before in earlier examples. Moreover, we pull the three address pins at the lower right of the MCP23017 to ground, which causes the device to have the default address 0x20. We also pull the reset pin, the fourth from the bottom on the right-hand side, to 5 V, which is required for stable operation. There are eight IO pins on the top-right part of the chip, labeled A0 to A7, where A7 is connected to the anode of the LED, and eight more IO pins on the top left of the chip. They are labeled B0 to B7 from top to bottom, with B0 connected to the small button on the top left. An additional wire connects a pin called INTB on the MCP23017 to pin D2 on the UNO. We configure the extender in such a way that INTB indicates that one of the pins B0 to B7 has changed. INTB is normally HIGH, but will go LOW once a change occurs, and that triggers an interrupt on the UNO, but more on that later when we discuss the inner workings of the following sketch that runs on the UNO.

```
// MCP23017, V. Ziemann, 170802
#include <Wire.h>
```

```

volatile uint8_t pin_has_changed=0,portB=0,portA=0;
#define MCP23017 0x20
void I2Cwrite(uint8_t addr, uint8_t reg, uint8_t val) {
    Wire.beginTransmission(addr); // addr=MCP23017
    Wire.write(reg);               // register
    Wire.write(val);               // write value
    Wire.endTransmission(true);
}
uint8_t I2Cread(uint8_t addr, uint8_t reg) {
    Wire.beginTransmission(addr); // addr=MCP23017
    Wire.write(reg);               // register
    Wire.endTransmission(false);
    Wire.requestFrom(addr,1);      // request 1 byte
    return Wire.read();            // read and return
}
void action() { //.....action
    pin_has_changed=1;
}
void setup() { //.....setup
    Wire.begin();                // Init I2C bus
    I2Cwrite(MCP23017,0x00,0x00); // IODIRA=all output
    I2Cwrite(MCP23017,0x0D,0xFF); // GPPUB Pullups on
    I2Cwrite(MCP23017,0x03,0xFF); // IPOLB reverse polarity
    I2Cwrite(MCP23017,0x05,0xFF); // GPINTENB, interrupts on
    pinMode(2,INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(2),action,FALLING);
    Serial.begin(9600); while (!Serial) {}
}
void loop() { //.....loop
    if (Serial.available()) {
        char line[30];
        Serial.readStringUntil('\n').toCharArray(line,30);
        if (strstr(line,"A7 ")==line) {
            int val=(int)atof(&line[3]);
            if (val==0) {portA &= B01111111;} else {portA |= B10000000;}
            I2Cwrite(MCP23017,0x012,portA); // write GPIOA
        }
    }
    if (pin_has_changed) {
        portB=I2Cread(MCP23017,0x11); // 0x11=INTCAPB, 0x13=GPIOB
        pin_has_changed=0;
        if (portB & B00000001) {
            Serial.println("Button B0 pressed");
        }
    }
    delay(1);
}

```

In the sketch we first include the `Wire.h` header file for support of I2C communication and a few variables. Here we use `volatile` to indicate that at least one of them can change in an

interrupt handler. Next we re-use the convenience functions to read and write a register on a I2C device. The function `action()` is called once the interrupt is triggered, which happens as a consequence of a pin changing on the MCP23017 that is subsequently signaled via the INTB pin going low. In the interrupt handler we only set a variable to indicate that we need to read the pins, which we later do in the `loop()` function. In the `setup()` function, we initialize I2C communication by calling `Wire.begin()`, and configure the MCP23017. Setting register 0x00 to 0x00 configures all pins of port A (those on the right-hand side with the LED attached to pin A7) as output. We do not need to do that for port B, because we use the pins in their default configuration as input. Next we enable the internal pull-up resistors on all pins of port B, and reverse the polarity with which a change is reported. It is convenient to invert the state, which normally is low if a button is pressed, to pull the input pin low. We prefer, however, to have the state reported as high. Finally we write 0xFF to register 0x05, which enables interrupts to be generated whenever the state of an input pin of port B changes. This completes the configuration of the MCP23017, but we still need to configure the UNO to listen to a change of state on its pin 2. This pin is configured as an input pin with pull-up enabled, and we attach the function `action()` to be executed if pin 2 changes state from high to low; in other words, if it is `FALLING`. We complete the configuration by initializing the serial line. In the `loop()` function, we first handle commands received over the serial line, and test whether the command starts with `A7`. We access the rest of the line by `&line[3]`, because `line` is an array of characters, and using the ampersand indicates the address of the array element number 3. Here we expect an integer, rather than a float value, but it is easy to use the construction `(int) fval` to cast the float value `fval` to an integer. This construction of parsing the rest of the `line` with `atof()` turns out to be a compact and robust way to decipher the numbers, and we will use it repeatedly. The result is placed in the variable `val` and if it is zero, we toggle bit 7 of the variable `portA` off and write the variable to the output register 0x12, which is responsible for the state of the output pins of port A, those on the right-hand side. Note that we use the prepended letter “B” to specify a number in binary representation. In particular, the LED is connected to the uppermost pin, labeled A7. By toggling the appropriate bit, we switch it on and off. After handling the Serial communication, we check whether the variable `pin_has_changed` is set, and if that is the case, we read a register with the present state of port B. We can directly read the present state of port B by reading from the GPIOB register 0x13. But the MCP23017 has a special feature that internally captures the state of port B immediately after the interrupt is triggered, and we retrieve that value by reading from register 0x11, which is called `INTCAPB` in the datasheet. We then check whether bit 0 is set and address that by sending some text to the serial line. Given this template and some reading of the datasheet, it should be straightforward to adapt the software to build a user interface with many buttons and LEDs.

To summarize this section, we point out that we can communicate with the sensors using basic I2C communication functions such as `Wire.read()`. In that case, we have to convert the retrieved raw data to physical quantities ourselves, which requires careful reading of the datasheets. If a library for the particular sensor is available, we can include it in the Arduino IDE by copying it to the `Arduino/libraries` subdirectory and use it after restarting the IDE. In that case, we strongly advise careful inspection of the examples that are normally included.

A second communication bus, similar to I2C, is the SPI bus. A number of sensors, but also other devices, such as analog-to-digital converters, use it, as we shall see in the next section.

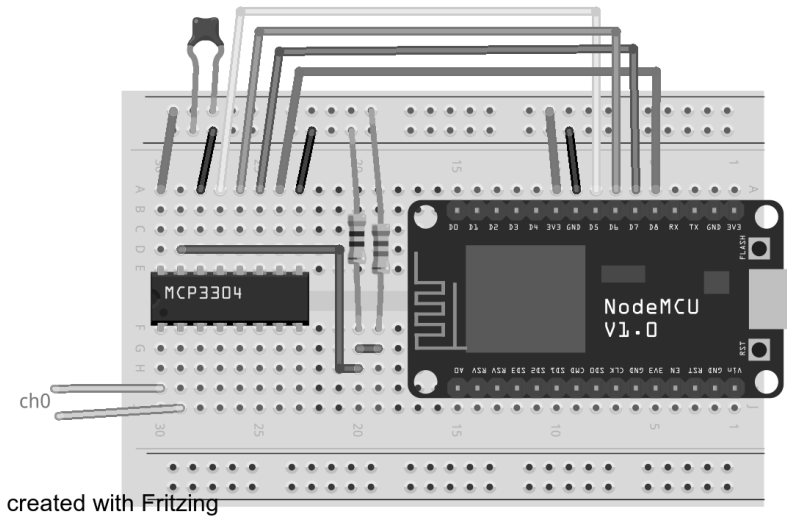


Figure 4.10 Connecting the MCP3304 ADC to the NodeMCU.

4.4.4 SPI

In this section we discuss interfacing the MCP3304 ADC. It supports an SPI interface, and we use it to expand the number of ADC channels of a NodeMCU microcontroller, which otherwise is equipped with a single ADC channel only. In Figure 4.10 we show how to connect the ADC to the NodeMCU on a small solderless breadboard, with the NodeMCU visible on the right and the ADC on the left. From left to right, the eight input pins A0 to A7 of the ADC are on the side facing downwards. A0 and A1 are used together to form differential input channel 0. On the upper side, we connect the supply voltage to the top left (pin 16) of the MCP3304 and ground to pins 9 and 14. The four SPI lines of the chip are connected to pins D5, D6, D7, and D8 on the NodeMCU: SCL connects to D5, MISO to D6, MOSI to D7, and chip-select to D8. We use a voltage divider with an 8.2k Ω and a 68k Ω resistor to lower the reference voltage to 0.4 V. It is connected with a wire to the Aref pin, pin 15 on the MCP3304. This increases the resolution of the ADC to 13 bits between ± 0.4 V.

In the following we discuss two ways to read the ADC values from the MCP3304. Later we program the interface routine using the built-in SPI interface libraries, but first we use a method called bit-banging, which means that we write and read the respective pins in the correct order, quasi *by hand*. The following sketch is adapted from the example code on the Arduino playground.

```
// MCP3304 bit-banged, V. Ziemann, 170726
#define CS 15          // D8, ChipSelect
#define MOSI 13        // D7, MasterOutSlaveIn
#define MISO 12         // D6, MasterInSlaveOut
#define CLK 14         // D5, Clock
void mcp3304_init_bb() { //.....mcp3304_init
    pinMode(CS,OUTPUT);
    pinMode(MOSI,OUTPUT);
    pinMode(MISO,INPUT);
```



```

    pinMode(CLK,OUTPUT);
    digitalWrite(CS,HIGH);
    digitalWrite(MOSI,LOW);
    digitalWrite(CLK,LOW);
}
int mcp3304_read_bb(int channel) { //.....mcp3304_read_bb
    int adcvalue=0, sign=0;
    byte commandbits = B10000000;
    commandbits|=(channel & 0x03) << 4; // 5 config bits, MSB first
    digitalWrite(CS,LOW);    // chip select
    for (int i=7; i>0; i--){ // clock bits to device
        digitalWrite(MOSI,commandbits&1<<i);
        digitalWrite(CLK,HIGH); // including two null bits
        digitalWrite(CLK,LOW);
    }
    sign=digitalRead(MISO);    // first read the sign bit
    digitalWrite(CLK,HIGH);
    digitalWrite(CLK,LOW);
    for (int i=11; i>=0; i--){
        adcvalue+=digitalRead(MISO)<<i;
        digitalWrite(CLK,HIGH);
        digitalWrite(CLK,LOW);
    }
    digitalWrite(CS, HIGH);
    if (sign) {adcvalue = adcvalue-4096; }
    return adcvalue;
}
void setup() { //.....setup
    mcp3304_init_bb();
    Serial.begin(115200);
    while (!Serial) { delay(10);}
}
void loop() { //.....loop
    int val=mcp3304_read_bb(0);
    Serial.print("CH0 = "); Serial.println(val);
    delay(1000);
}

```

In the sketch we first give meaningful names to the pins we use on the NodeMCU, and declare the function `mcp3304_init_bb()` where we collect the code needed to initialize the bit-banged SPI communication. We declare the respective pins to be `INPUT` or `OUTPUT` and initialize their state with a call to `digitalWrite()`. The `mcp3304_read_bb()` function takes the channel number as input and returns the digitized word. In this routine we assume that we use two input pins of the MCP3304 in differential mode. Inside the routine we first declare some variables and initialize the `commandbits` to start with a most significant bit 7 that has the value 1. The next bit 6 is 0, indicating differential mode, and then we add the channel information as bits 5 and 4 and set bit 3 to zero, followed by three more zeros, following the description from the datasheet of how to configure the ADC. Now we can pull the CS line `LOW` to indicate that a transaction is about to start, and then clock the five configuration bits plus two extra bits to allow some time for the conversion by setting the

MOSI pin to the respective value and toggling the CLK pin. Once we complete this we can read the `sign` bit from the MISO line and toggle the CLK pin. Then we repeat to read MISO and toggle CLK 12 times to fill the appropriate bits of the integer `adcvalue`, and finally conclude the transaction by pulling the CS pin HIGH. Before returning the `adcvalue` we ensure that the sign bit is properly folded into the reading. In the `setup()` function we only initialize the serial line and the MCP3304, and in the `loop()` function we read channel 0 from the ADC and display its value on the serial line. The latter is repeated once a second.

Instead of bit banging the pins, we can also utilize the SPI library that comes with the Arduino IDE. The sketch, based on the `<SPI.h>` library, but otherwise equivalent to the previous one, is the following.

```
// MCP3304, V. Ziemann, 170726
#include <SPI.h>
#define CS 15 // D8
int mcp3304_read_adc(int channel) { //...0 to 3.....read_adc
    int adcvalue=0, b1=0, hi=0, lo=0, sign=0;;
    digitalWrite(CS, LOW);
    byte commandbits = B00001000; // Startbit+(diff=0)
    commandbits |= channel & 0x03;
    SPI.transfer(commandbits);
    b1 = SPI.transfer(0x00); // always D0=0
    sign = b1 & B00010000;
    hi = b1 & B00001111;
    lo = SPI.transfer(0x00); // input is don't care
    digitalWrite(CS, HIGH);
    adcvalue = (hi << 8) + lo;
    if (sign) {adcvalue = adcvalue-4096;}
    return adcvalue;
}
void setup() {//.....setup
    pinMode(CS,OUTPUT);
    SPI.begin();
    SPI.setFrequency(2100000);
    SPI.setBitOrder(MSBFIRST);
    SPI.setDataMode(SPI_MODE0);
    Serial.begin(115200);
    while (!Serial) {yield();}
}
void loop() { //.....loop
    int val=mcp3304_read_adc(0);
    Serial.print("CH0 = "); Serial.println(val);
    delay(1000);
}
```

Here we first include the `<SPI.h>` functionality and define the CS pin before defining the `mcp3304_read_adc` function to read a `channel` from the ADC. In this function we first declare a number of variables and pull the CS pin LOW in order to start the transaction. Then we build the `commandbits`; this time they are aligned such that the start bit is bit 3 and the channel information is stored in bits 1 and 0, and we use the `SPI.transfer` function to send it to the ADC. Note that the first bit recognized by the ADC is the first non-zero bit, which is the start bit 3. We then send 0x00 in a second call to `SPI.transfer` and receive

the readings from the MISO pin in `b1`. Since we have a number of idle clock toggles, the sign bit is bit 4 and the next four bits are the four most significant bits of the ADC reading, which we store in the variable `hi`. The next call to `SPI.transfer()` returns the subsequent bits, which are the lower eight bits from the MISO pin, and we store them in the variable `lo`. After pulling the CS pin HIGH to conclude the transaction, we build the ADC word `adcvalue` and add the `sign` information. Finally we return the reading to the calling program. In the `setup()` function we initialize the SPI functionality by calling the `SPI.begin()` function, setting the clock frequency, byte order, and MODE, and then initialize the serial line. The `loop()` function is a straight copy of the previous example, where we only read channel 0 and print the value to the serial line.

This external ADC greatly expands the analog input capabilities of the NodeMCU, which only has a single ADC channel with unipolar 10-bit resolution on board. Now we have four extra channels at 12-bit resolution, including an additional polarity bit. We note in passing that the MCP3304 has a close relative, the MCP3208, which has eight unipolar channels and can be programmed in a similar way. Consult the datasheets for the details. Sensors that support SPI communication can be controlled in a similar way to the ADCs, either by bit banging the respective pins or by using the `SPI.transfer()` function. Again, reading the detailed specifications in the datasheets is mandatory.

So far, we discussed the standard communication channels, RS-232, I2C, and SPI, but there are several others that we briefly address in the next section.

4.4.5 Other protocols

The DHT11 and DHT22/AM2302 humidity sensors use a nonstandard communication protocol, but this is reasonably well explained in the datasheet. The protocol is based on using a single IO-pin, which requires a pull-up resistor of about 4.7 k Ω . On breakout boards, the resistor is likely already mounted. Initially the microcontroller configures the pin as OUTPUT and pulls it to LOW for at least 18 ms, before releasing control to the sensor by reconfiguring the pin to INPUT. The sensor then acknowledges the transaction by sending a pulse of 50 μ s LOW followed by 50 μ s HIGH before clocking 40 data bits to the pin using the following convention. A zerobit is defined by a LOW state for 50 μ s and a subsequent HIGH state for about 27 μ s. A onebit is defined by a 50 μ s LOW state followed by a 70 μ s long HIGH state. The 40 data bits contain the information about the humidity and the temperature. In the following code we implement this protocol.

```
// DHT11, V. Ziemann, 170804
#define DHT 2    // sensor pin
float read_dht11(float *T) {
    bool p[500]= { 0 };
    uint8_t data[5]={0},checksum;
    int ic=0,goes_up=0;
    pinMode(DHT,OUTPUT);
    digitalWrite(DHT,LOW);    // 20 ms low pulse
    delay(20);
    digitalWrite(DHT,HIGH);
    noInterrupts(); // interrupts off for accurate timing
    delayMicroseconds(20); // make sure we start at low
    pinMode(DHT,INPUT);
    for (int i=0;i<500;i++) { // read 5 ms worth of data
        p[i]=digitalRead(DHT);
```

```

    delayMicroseconds(10);    // one every 10 us
}
interrupts();    // interrupts on again
while (p[ic++] == 0); // next HIGH, acknowledge bit
while (p[ic++] == 1); // next LOW, first data bit
for (int i=0;i<5;i++) {      // loop over the
    for (int j=7;j>=0;j--) {  // 40 data bits
        while (p[ic++] == 0) {goes_up=ic;}
        while (p[ic++] == 1) ;
        (ic-goes_up > 4) ? data[i] |= (1<<j) : 0;
    }
}
checksum=((data[0]+data[1]+data[2]+data[3]) & 0xFF);
if (checksum==data[4]) {
    *T=(float) data[2];      // temperature
    return (float) data[0];  // humidity
}
*T=-100;    // only get here if bad reading
return -1;
}

void setup() { //.....setup
    Serial.begin(9600);
    while (!Serial) {}
    pinMode(DHT,INPUT);
    digitalWrite(DHT,HIGH);
    delay(2000);    // wait for DHT to wake up
}

void loop() { //.....loop
    float temperature,humidity;
    humidity=read_dht11(&temperature);
    Serial.print(humidity); Serial.print("\t");
    Serial.println(temperature);
    delay(2000);
}

```

In this sketch we first define the pin to which the DHT11 is connected and the function `read_dht11()` that encapsulates the communication with the sensor. There we first declare a number of variables and then directly progress to configuring the pin to which the sensor is connected as `OUTPUT`, and produce a 20-ms-long `LOW` pulse before pulling the pin `HIGH` and reconfiguring it as `INPUT`. Since the timing of the response from the DHT11 is critical, we briefly turn interrupts off, which disables background processing of serial or `WLAN` activities. Then we read the pin every 10 μ s 500 times for a total duration of 5 ms, which is sufficient to read 40 bits, each having a maximum duration of 120 μ s, into the boolean array `p[]`. Once the acquisition is complete, we turn interrupts on again. Now that we have the entire data set in memory we can decode it. The variable `ic` steps through the received array `p[]` and searches for `LOW` to `HIGH` transitions and `HIGH` to `LOW` transitions. First we take care of the acknowledge pulse before looping over 5 bytes of 8 bits each. For each bit we measure the separation from the time the signal goes up until it goes down again. If the difference of the indices is more than four, which means that the temporal separation is longer than 40 μ s, we set the corresponding bit in the respective `data` byte. Once 40

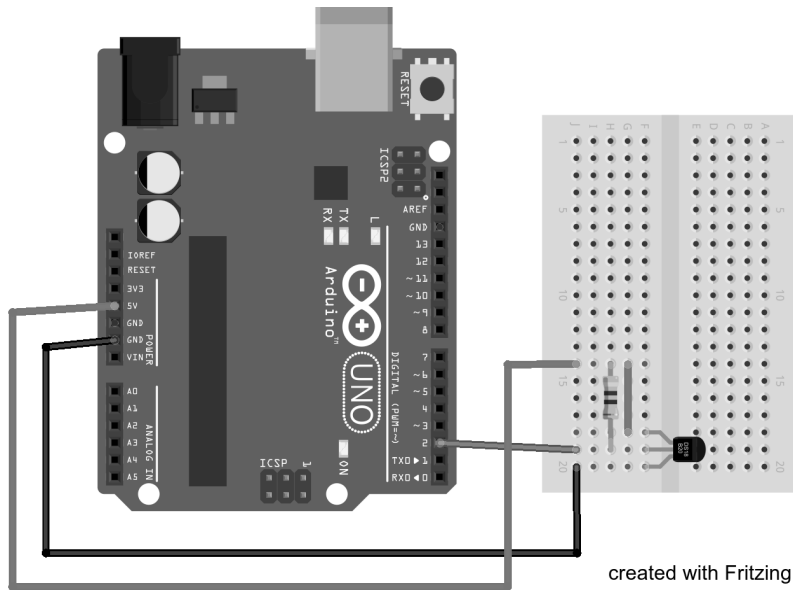


Figure 4.11 Connecting the DS18B20 sensor to the Arduino UNO.

bits are decoded we calculate the checksum, which is the sum of the first four data bytes, truncated to eight bits. If this equals the fifth byte, the transmission is correct and the function returns `data[0]` as the humidity and `data[2]` as the temperature, as specified in the datasheet. If the checksum is incorrect, the function returns values that are obviously wrong to allow checking in the calling program. Note that the DHT22/AM2302 sensors have a slightly different interpretation of the data bytes. For these sensors, the humidity and temperature are instead given by

```
humidity=(0.1*(256*data[0]+data[1]));
*T=(float) (0.1*(256*data[2]+data[3]));
```

In our version of the `read_dht11()` function, we first read the entire time series of 500 points with a spacing of $10\mu\text{s}$ before decoding. This wastes a little memory, but is more robust than waiting for transitions of the pin while decoding, as is done in other implementations of the DHT libraries. Note also that sampling a pin in this way constitutes a simple logic analyzer. This may come handy in other contexts as well.

Another data bus, similar to the one for the DHT sensors, but with additional error handling, is the Dallas 1-wire protocol that is used by the popular DS18B20 temperature sensor, but also by other devices, such as humidity sensors, memory circuits, or autonomous data loggers. We will, however, focus on the DS18B20 and connect it to the Arduino using the schematics shown in Figure 4.11 where we connect ground and supply voltage and the single data pin to pin D2 on the UNO. We also added a $4.7\text{ k}\Omega$ pull-up resistor from the data pin to the supply voltage. This is needed for a bare sensor only. If we use a breakout-board version of the DS18B20, the pull-up can likely be omitted, because it is already mounted on the breakout board. For interfacing to the Arduino, we use ready-made libraries that encapsulate the low-level interaction. Before using the libraries, we need to install the *OneWire* and *DallasTemperature* libraries. An easy way is to use the library manager of the Arduino IDE, which is accessible from the *Sketch*→*Include Library*→*Manage*

Libraries, where we enter the keywords *OneWire* and *DallasTemperature*. This displays a list of supported libraries that can be installed directly from the menu. After installation, we enter the following code, which shows a simple example.

```
// DS18B20 1-wire temperature sensor, V. Ziemann, 170120
#include <OneWire.h>
#include <DallasTemperature.h>
OneWire oneWire(2); //use pin D2
DallasTemperature sensors(&oneWire);
void setup() {
  Serial.begin(9600); while (!Serial) {}
  sensors.begin();
}
void loop() {
  sensors.requestTemperatures();
  float temp=sensors.getTempCByIndex(0);
  Serial.println(temp);
  delay(1000);
}
```

In the sketch, we first include support for the *Onewire* and *DallasTemperature* devices, and initialize the *OneWire* bus using pin number 2, followed by the initialization of the *DallasTemperature* sensors, to use the bus initialized in the previous statement. Note that we can connect several temperature sensors in parallel on the same bus. In the *setup()* function we only initialize serial communication and the sensor. The code in the *loop()* function sends out the query to the connected sensors, which causes them to send back their measurement values. In our example we only connect a single DS18B20 such that we only can read out the temperature in Celsius from the device number zero, and print the result to the serial line. After waiting a second we repeat the procedure.

The HC-SR04 distance sensor, shown in Figure 2.34, determines the distance to the nearest obstacle by emitting a short high-frequency (non-audible) acoustic pulse when its trigger pin receives a short 10- μ s-long trigger pulse. Then it pulls the echo pin to low voltage and only returns it to the high state once the echo is received. We therefore need to connect the sensor with four wires (GND, 5V, Trig, Echo) to the Arduino, and need to measure the duration of the echo pulse. This is accomplished in the following sketch.

```
// HC-SR04 distance logger, V. Ziemann, 161204
const int trig=2, echo=3;
void setup() {
  Serial.begin(9600); while (!Serial) {}
  pinMode(trig,OUTPUT);
  pinMode(echo,INPUT);
}
void loop() {
  float duration,distance;
  digitalWrite(trig,LOW); // make a 10 us trigger pulse
  delayMicroseconds(2);
  digitalWrite(trig,HIGH);
  delayMicroseconds(10);
  digitalWrite(trig,LOW);
  duration=pulseIn(echo,HIGH);      // wait for echo
```

```

    distance=100*duration*340e-6/2;    // in cm
    Serial.println(distance);
    delay(1000);
}

```

In the sketch, we first declare the constants with the pins used in this sketch. The trigger pin of the sensor is connected to pin 2 on the Arduino and the echo pin on the sensor to pin 3. In the `setup()` function, we declare the serial line and whether the respective pins are used as input or output. In the `loop()` function, we first declare the variables for distance and duration and then create a 10- μ s-long trigger pulse by writing `HIGH` and `LOW` to the trigger pin, and wait a short time in between. Once the trigger pulse is dispatched, we use the `pulseIn()` function to wait for the echo pin to return to the `HIGH` state. The `pulseIn()` function returns the elapsed time in microseconds such that 1000 μ s correspond to a distance of about 0.17 m, or 17 cm if we assume a value for speed of sound of 340 m/s and note that the sound has to go back and forth from the sensor to the first obstacle. Then the measured value is sent over the serial line to the host computer, and the whole process is repeated after some delay. We note in passing that we can use the `pulseIn()` function to measure the duration of short pulses in general.

Output pins of a rotary encoder carry a sine-like and a cosine-like signal, respectively. We can determine whether the shaft is turned clockwise or counterclockwise depending on whether the two pins are equal or not while one of them changes on a falling edge, going from a `HIGH` to a `LOW` state. The following code implements this method using interrupts. Using interrupts has the advantage that the state of the pins does not have to be measured continuously and compared, but an `interrupt_handler` is registered with a certain action, in our case on a falling edge on the pin connected to the interrupt. The Arduino has two pins, numbers 2 and 3, that have the interrupt functionality. But let us look at the sketch first.

```

// Rotary encoder, V. Ziemann, 161205
const int pinA=2,pinB=4;
volatile int pos=0;
void setup() {
    Serial.begin(9600); while (!Serial) {}
    pinMode(pinA,INPUT_PULLUP);
    pinMode(pinB,INPUT_PULLUP);
    attachInterrupt(0,interrupt_handler,FALLING);//0=pin2,1=pin3
}
void loop() {
    Serial.println(pos);
    delay(1000);
}
void interrupt_handler() {
    if (digitalRead(pinA)==digitalRead(pinB)){pos++;}else{pos--;}
}

```

The sketch starts by declaring the used pins and a variable `pos` that contains the encoder position. The variable is declared `volatile`, which instructs the compiler that it can change asynchronously within the interrupt routine, and prevents the compiler from optimizing it away, because it does not change in the main program. In the `setup()` function we declare the serial line parameters and the input pins with pull-up resistors enabled. One of the pins *must* be pin 2 or 3, those with interrupt capability. The `attachInterrupt()` function is

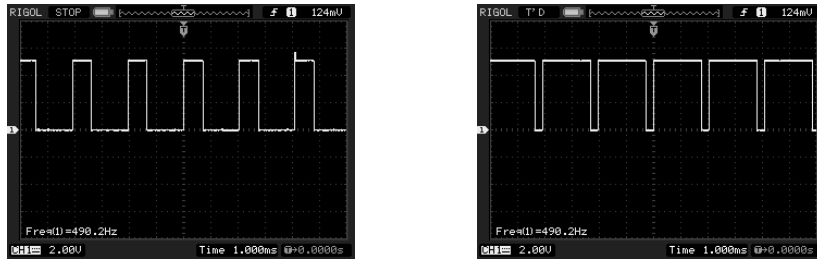


Figure 4.12 Oscilloscope trace of a pulse-width modulated signal on pin D9 of an Arduino UNO with values of 88 (left) and 220 (right).

used to connect a so-called callback function, here `interrupt_handler`. It is defined at the end of the sketch. In our sketch we chose interrupt 0 (connected to pin 2) to trigger the callback function `interrupt_handler`. We also specify that the callback function is called on a `FALLING` edge on the interrupt pin. Instead of hard coding the value 0 as the source of the interrupt, we can use the service function `digitalPinToInterrupt()` that takes the pin number, 2 in our case, as argument, such that the line reads

```
attachInterrupt(digitalPinToInterrupt(2), interrupt_handler, FALLING);
```

instead. Using `digitalPinToInterrupt()` is the preferred way, because it relieves the user of remembering which interrupt number is connected to which pin. The `loop()` function is very simple and only writes the position of the encoder shaft to the serial line once a second. Finally, we define the callback function `interrupt_handler`, which increments the `pos` variable by one step if the two encoder pins are equal, and decrements otherwise. In this way, the sense of rotation of the shaft is taken care of. We need to point out that we had to use separate external pull-up resistors (10 k Ω) to guarantee stable operation, and in the sketch we may encounter a so-called race condition when the variable `pos` changes while it is transmitted on the serial line.

After the discussion of interfacing the sensors, we now move on to investigate how to work with actuators, such as switches, motors, or analog voltages.

4.5 INTERFACING ACTUATORS

In this section we discuss how to interface the actuators discussed in Chapter 3 to the Arduino, and start with the simplest actuator, the LED.

4.5.1 Switching devices

In Section 4.3 we used the `digitalWrite()` function to turn an LED on and off. If we connect a transistor, Darlington driver, or H bridge to an output pin, we can use the same function to handle larger current than the 20 mA the Arduino can drive. Consult the circuits shown in Section 3.1.2 for details.

Dynamically adjusting the brightness of an LED is achieved by pulse-width modulation, and that feature is available on pins D3, D5, D6, D9, and D11 on the Arduino Uno. We use the `analogWrite(pin, value)` function that takes the pin number and a value between 0 and 255 (0 to 1023 on the ESP8266) to adjust the duty cycle of the pulse-width modulated signal from completely off to completely on. The switching frequency is between 500 and

1000 Hz, and in Figure 4.12, we show an oscilloscope trace of the result of `analogWrite()` to pin D9 of a UNO with values 88 and 220, respectively. The frequency of the signal is about 490 Hz, and we see the length of the signal differing from about 1/3 of the time at 5 V on the left to 5 V almost all the time on the right.

In order to be able to manipulate actuators in the same way as the sensors, we use the same query-response protocol to communicate the required actions to the microcontroller. We want to use the convention that `DWx 0` turns digital pin number `x` off and `DWx 1` turns it on, and that sending `AWx nnn` sets pulse-width modulation on pin number `x` to value `nnn`. The code that achieves this is the following:

```
// Switching_and_pwm, V. Ziemann, 170614
char line[30];
void setup() {
    pinMode(2,OUTPUT);
    Serial.begin (9600);
    while (!Serial) {};}
}
void loop() {
    if (Serial.available()) {
        Serial.readStringUntil('\n').toCharArray(line,30);
        if (strstr(line,"DW2 ")==line) {
            int val=(int)atof(&line[4]);
            digitalWrite(2,val);
        } else if (strstr(line,"AW9 ")==line) {
            int val=(int)atof(&line[4]);
            analogWrite(9,val);
        } else {
            Serial.println("unknown");
        }
    }
}
```

where we first declare the variable `line` that contains the characters received on the serial line. In the `setup()` function, we only declare pin 2 as output, and initialize the serial communication, exactly as before. In the `loop()` function, we use the previously seen construction to parse the message received on the serial line and perform some action, depending on the command received. If the command starts with `DW2` we read the characters following the first 4 into the variable `val` and pass `val` to the `digitalWrite()` function. Here we employ the same method of using the `atof()` function to decode the rest of the character array `line[]` that we first used in the sketch to interface the MCP23017 IO extender. We immediately encounter the `atof` construction again when parsing the value received after `AW9`. We then use it to set pin D9 to the desired value with the `analogWrite()` function. In this way we have a simple mechanism to instruct the Arduino to turn pins on and off and adjust those that have pulse width capability to the desired duty factor.

We note that the code above does not contain graceful handling of errors. For example, passing any value `val` received from the Serial line to the `digitalWrite()` function when handling the `DW2` command can result in weird behavior if `val` is not 0 or 1. Even a negative number could be written to the pin, which is meaningless. Replacing that line by the following construction will only write meaningful values to the pin. Implicitly, we use the convention that 0 turns the pin off, and anything else turns it on.

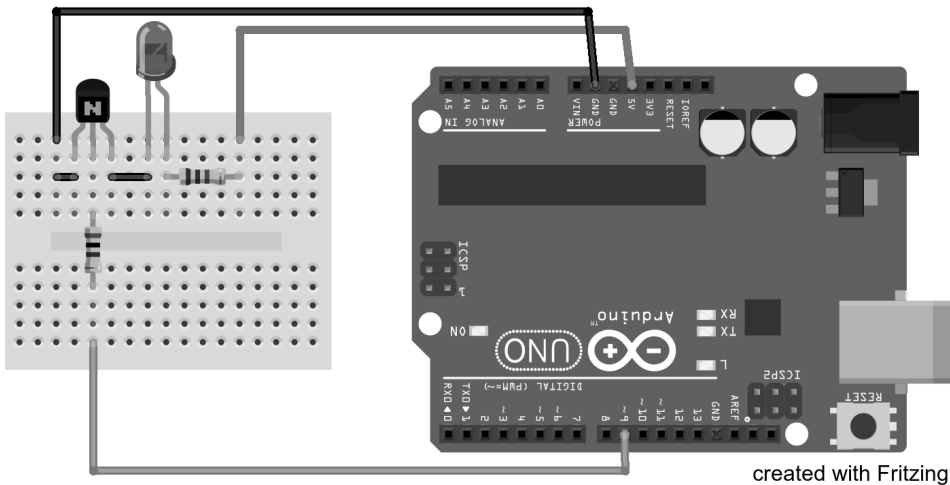


Figure 4.13 Using pulse-width modulation and a transistor to adjust the brightness of an LED.

```
// digitalWrite(2,(int)val);
if ((int)val == 0) {
    digitalWrite(2,LOW);
} else {
    digitalWrite(2,HIGH);
}
```

Of course, other ways of catching errors can be implemented, but that goes beyond the scope of our presentation.

Testing the above program with a UNO is quickly done. In Figure 4.13 we show an implementation on a small breadboard. Two cables connect the power lines to the circuit, where the ground signal connects to the emitter of the NPN transistor. Here we use a BC547, but any small-signal NPN transistor will work. The collector of the transistor is directly connected to the cathode of the LED, and the anode of the LED is connected via a $220\,\Omega$ to the positive supply voltage. The base of the transistor is connected via a $1\,\text{k}\Omega$ resistor to the controlling pin, here pin D9, on the Arduino.

This example covers the basic functionality of turning on and off, as well as controlling the power delivered to a load, the LED. In the next example we use very similar circuitry to control speed and direction of motors.

4.5.2 DC motors

If we only want to control the speed of a very small motor, we can replace the LED and the $220\,\Omega$ resistor in the previous example by the motor. We also need to add flyback diodes to prevent the back-emf from damaging the transistor. For slightly larger motors, we need to use a transistor with a higher power rating, such as a TIP-120 Darlington power transistor. The TIP-120 has a flyback diode from emitter to collector already built in. Note that the performance of larger transistors normally degrades at higher frequency. Inspection of the datasheet, however, shows that this only affects frequencies well above $10\,\text{kHz}$.

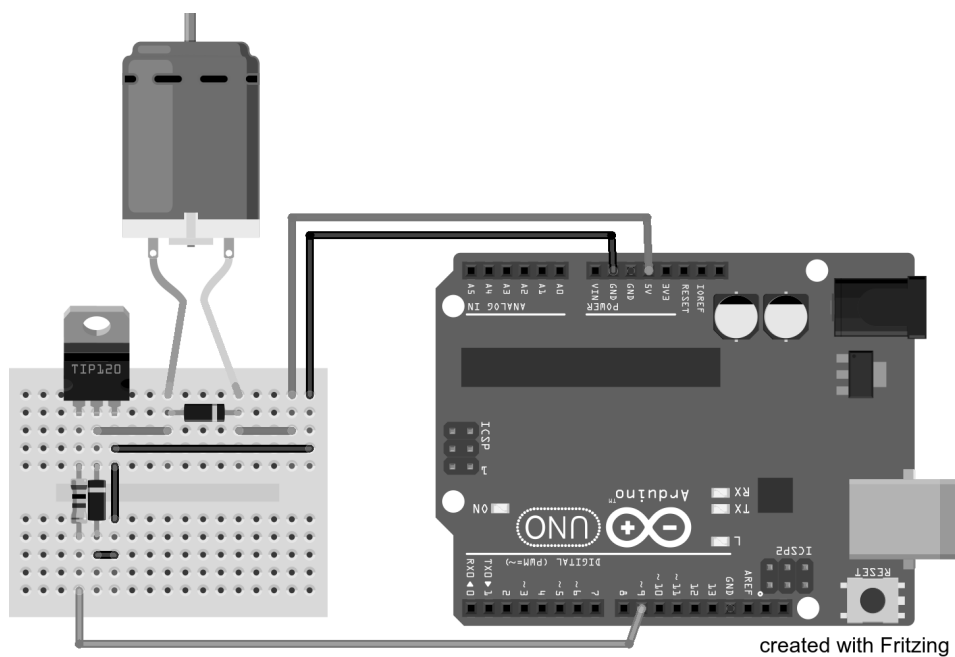


Figure 4.14 Controlling the speed of a motor with pulse width modulating the base of a TIP-120 Darlington transistor. See the text for a discussion of the diodes.

In Figure 4.14 we show the setup with a UNO controlling a small motor. The terminals of the transistor are base, collector, and emitter, from left to right, and the motor is connected between the collector and the supply voltage. The emitter is directly connected to ground, and we include an external flyback diode (horizontally mounted) between the motor leads, with the cathode pointing towards the right. The vertically mounted diode illustrates the connection of the built-in diode. We control the speed of the motor by pulse width modulating the base of the transistor that is connected to pin D9 on the UNO via a 1 kΩ resistor.

Controlling the speed is useful, but often we also need to control the direction of rotation; for example, to drive a vehicle not only forward, but also backwards. The standard way to accomplish this feature is by using an H bridge, as discussed in Chapter 3. Here we use

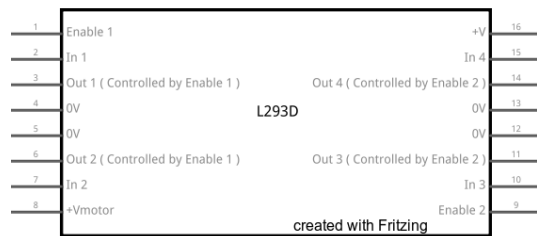


Figure 4.15 The pin assignment of the L293D H-bridge driver.

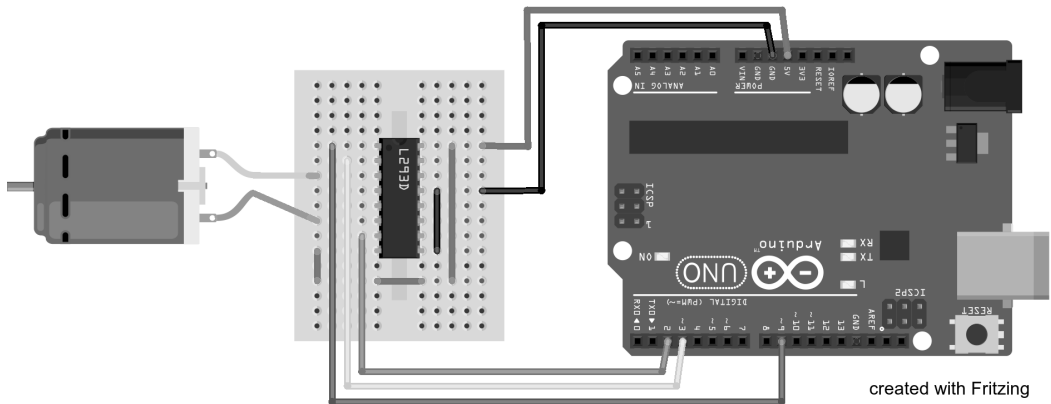


Figure 4.16 Controlling speed and direction of a DC motor.

an L293D H bridge driver, which is called a *quadruple half-bridge driver*, a term that will become clear by considering the pin layout of the 16-pin chip, shown in Figure 4.15. There are ground pins and two supply voltages, one for logic levels and one to supply power to the motor. Then there are four (quadruple) inputs with corresponding outputs. The circuit works in such a way as to translate the logic level on the input pins to the corresponding output pins, which in turn deliver the voltage level from the motor power supply to the motor. Thus, setting input 1 to **HIGH** and input 2 to **LOW** connects one motor lead to the motor supply voltage and the other to ground, which causes the motor to turn in one direction. Setting input 1 to **LOW** and input 2 to **HIGH** will cause the converse, and the motor will turn in the other direction. Two inputs and outputs therefore provide the functionality of a single H bridge. Since there are four inputs and outputs, we can use the L293D to implement two H bridges. Moreover, pin 1 on the L293D can be used to enable the outputs of input 1 and 2. Applying a pulse-width modulated signal to this pin will therefore work as speed control for the motor. Pin 9 provides the same functionality for inputs 3 and 4. Finally, we remark that the character *D* in L293D indicates that flyback protection diodes are already built into the integrated circuit.

With the theoretical background covered we are ready to build a motor controller with an L293D on a breadboard, and control it with an Arduino UNO. We show the circuit in Figure 4.16. The DC motor is shown on the left and the L293D is the only component on the small breadboard. The orientation of the chip is the same as in Figure 4.15, making the functionality easy to understand. The pin numbering goes from top left (pin 1) to bottom left (pin 8), continuing via bottom right (pin 9) to the top right (pin 16), which is the standard for most integrated circuits.

The upper two wires connect ground and the supply voltage to the chip. Here we assume that the supply voltage is the same, and we connect pin 8 of the L293D to the positive logic supply voltage. Moreover, we disable the output on the right-hand side of the chip by pulling pin 9 to ground. Two wires connect digital outputs D2 and D3 on the Arduino to the input of two half-bridges on pins 2 and 7 on the L293D. The corresponding outputs from pins 3 and 6 are connected to the motor cables. Finally, pin 1 on the L293D is connected to the pulse-width modulated output pin D9 on the UNO.

We want control the motor from the Arduino by sending commands via the serial line according to the protocol that **FW nnn** sets the speed in the forward direction and **BW nnn**

sets the speed in the reverse direction. Anything else will stop the motors. This is easy to do with the following sketch.

```
// H bridge DC motor Controller, V. Ziemann, 170614
char line[30];
void setup() {
  pinMode(2,OUTPUT);
  pinMode(3,OUTPUT);
  Serial.begin (9600);
  while (!Serial) {}
}
void loop() {
  if (Serial.available()) {
    Serial.readStringUntil('\n').toCharArray(line,30);
    if (strstr(line,"FW ")==line) {
      digitalWrite(2,LOW);
      digitalWrite(3,LOW);
      digitalWrite(3,HIGH);
      float val=atof(&line[3]);
      analogWrite(9,(int)val);
    } else if (strstr(line,"BW ")==line) {
      digitalWrite(2,LOW);
      digitalWrite(3,LOW);
      digitalWrite(2,HIGH);
      float val=atof(&line[3]);
      analogWrite(9,(int)val);
    } else { // STOP in all other cases
      digitalWrite(2,LOW);
      digitalWrite(3,LOW);
      analogWrite(9,(int)0);
    }
  }
}
```

The `setup()` function only declares the pins used to control the H bridge as outputs and initializes the serial line. In the `loop()` function we employ the standard query-response mechanism. A command on the serial line that starts with **FW** first turns the motor off as a precaution against having both outputs positive, which might short the motor supply voltage. Then pin D3 is pulled high, resulting in the motor turning one way. Finally we parse the rest of the command line and write the value to set the motor speed by pulse-width modulating the enable pin of the motor-driver. Any command received, apart from **FW n** and **BW n**, will disable both outputs and sets the speed to zero. Of course, we can easily implement other commands, such as **STOP**, that will cause the motor to stop.

Even more elaborate commands can be programmed into the controller, such as turning one way at some time for a given time, then stopping and turning backwards for another time. The following code fragment may serve as an example.

```
} else if (strstr(line,"BACKANDFORTH")==line) {
  digitalWrite(2,LOW); // turn all off
  digitalWrite(3,LOW);
  digitalWrite(2,HIGH); // chose one direction
```

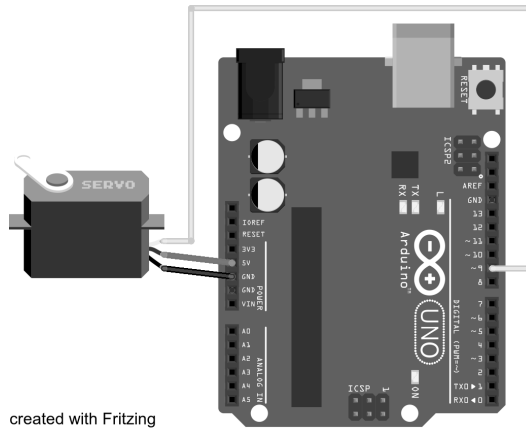


Figure 4.17 Connecting a model-servo to the Arduino UNO.

```

analogWrite(9,255); // full speed
delay(500);
digitalWrite(2,LOW); // stop
delay(1000);
digitalWrite(3,HIGH); // other direction
delay(500);
digitalWrite(3,LOW); // stop
analogWrite(9,0); // stop PWM
} ...

```

Other examples are to move for a short time, do some measurements, using code examples from the previous section, and continue doing the same for a number of steps. Yet another example is moving until a limit switch is engaged, which will cause the motors to turn off, maybe do some measurement, and later return to the parking position at a second limit switch.

Or, what about continuously reading a sensor and stopping once a certain value is achieved? This is actually what a model-servo does. So, let us look at it more closely.

4.5.3 Servos

As we discussed in Chapter 3, servos are used to carefully change the position or orientation of some device, such as the rudder of a boat or the steering system of a radio-controlled car. Servos only need three wires to be connected: ground and the supply voltage as well as one wire that carries the control information according to the timing shown in Figure 3.9. In Figure 4.17 we show how to connect a small servo to an Arduino UNO. If the servo requires more current or other voltages than the UNO provides, one can use a suitable external power supply. The ground cables of the external supply and the UNO need to be connected, and one wire from the servo, which is often red, needs to be connected to the positive terminal of the external supply instead of the 5V power pin on the Arduino. Controlling a servo from the Arduino is rather simple, and we use the following code to do that.

```

// Servo controller, V. Ziemann, 170614
#include <Servo.h>

```

```

Servo myServo;
char line[30];
void setup() {
    myServo.attach(9);    // pin D9
    Serial.begin (9600);
    while (!Serial) {}
}
void loop() {
    if (Serial.available()) {
        Serial.readStringUntil('\n').toCharArray(line,30);
        if (strstr(line,"SERVO ")==line) {
            float val=atof(&line[6]);
            myServo.write((int) val);    // 0 to 180
        }
    }
}

```

The code follows the normal template with opening serial line in the `setup()` function and the query-response construction in the `loop()` function. Servo-specific means the inclusion of the `<Servo.h>` file and the declaration of a `Servo` object that we call `myServo`. In the `setup()` function, we attach the servo functionality to pin D9 on the Arduino and write new values to the servo with the call to `myServo.attach()`. The latter function takes a value between 0 and 180 as argument and moves the arm of the servo to the desired position. Note that `<Servo.h>` is part of the standard Arduino distribution. Note also that we can use any pin on the Arduino to control servos, even multiple servos, but that using the servo functionality disables the pulse-width modulation feature of the `analogWrite()` function on pin D9 and D10 because it uses the same hardware timer.

Using servos in more advanced ways, such as scanning, is also easy to implement. The following code fragment scans back and forth through the entire range

```

} else if (strstr(line,"SCAN")==line) {
    for (int pos=0;pos<180;pos+=1) {
        myServo.write(pos); delay(10); // and do a measurement
    }
    for (int pos=180;pos>=0;pos-=1) {
        myServo.write(pos); delay(10);
    }
}

```

and it is easy to envision including some measurement inside the loops that perform the scanning. Imagine an HR-SR04 distance sensor attached to the servo. Scanning in a semi-circular motion and continuously measuring the distance to the closest object mimics the functionality of radar by using a sonar-like device instead. This could serve as a collision detection system for moving vehicles.

The last type of motors that we consider are stepper motors, and they come next.

4.5.4 Stepper motors

As we discussed before, stepper motors can be very accurately controlled by a given pulse sequence that turns the shaft by a desired number of discrete steps. The coils inside the motors need to be excited in certain patterns, either by a unipolar or bipolar power supply. We start with the unipolar type.

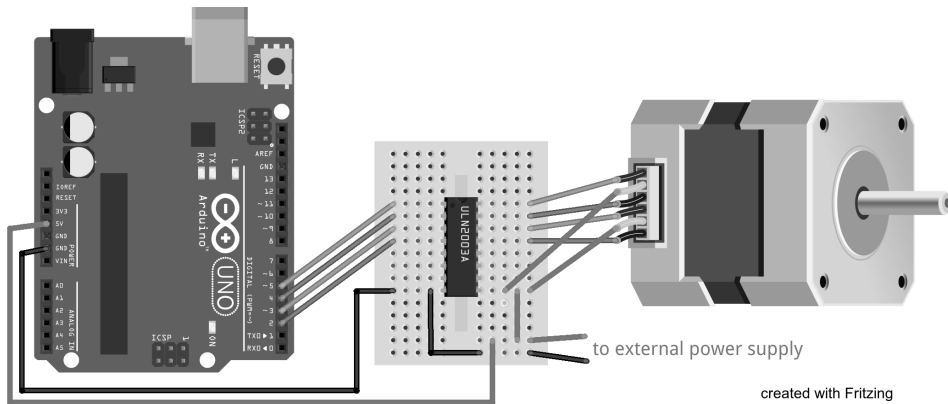


Figure 4.18 Connecting a unipolar stepper motor to the UNO with a ULN2003 Darlington driver.

In Figure 4.18 we show the connection of a unipolar stepper motor with the help of a ULN2003 Darlington driver that is mounted on the small breadboard between the Arduino UNO and the motor. The inner connection of the ULN2003 is very simple. Pins 1 to 7 on the left-hand side are the bases of seven Darlington transistors with resistors built into the chip. The facing pins on the right-hand side are the collectors of the respective transistors. All emitters are wired to pin 8 on the lower left-hand side of the chip, which is also ground. Pin 9 on the lower right-hand side is the terminal for the motor power supply, and is often connected to an external power supply that provides adequate current and voltage to power the motor. In the figure we assume that the stepper motor is small, and we connect pin 9 to the 5 V pin on the Arduino with one wire connected to the left-hand side on the Arduino. Ground is connected with the other wire connected to the left-hand side. The bases of the Darlington transistors are directly connected to pins D2 to D5 of the Arduino and need to be controlled by the program.

The center tap of the motor coils are connected to the positive terminal of the motor power supply with two wires. The other four wires are hooked up to the collectors of the upper four Darlington transistors in the ULN2003. Since the center tap of a coil is connected to positive supply voltage, placing a positive voltage on the base of a Darlington will cause the collector–emitter link to conduct and a current to flow through the coil. In this way we can excite the coils in a suitable pattern to rotate the shaft either clockwise or counterclockwise. The sketch that implements this functionality using the query-response protocol is the following.

```
// Stepper controller, V. Ziemann, 170616
#include <Stepper.h>
Stepper myStepper(200, 2, 4, 3, 5);
char line[30];
void setup() {
    myStepper.setSpeed(60);
    Serial.begin(9600);
    while (!Serial) {}
}
void loop() {
```

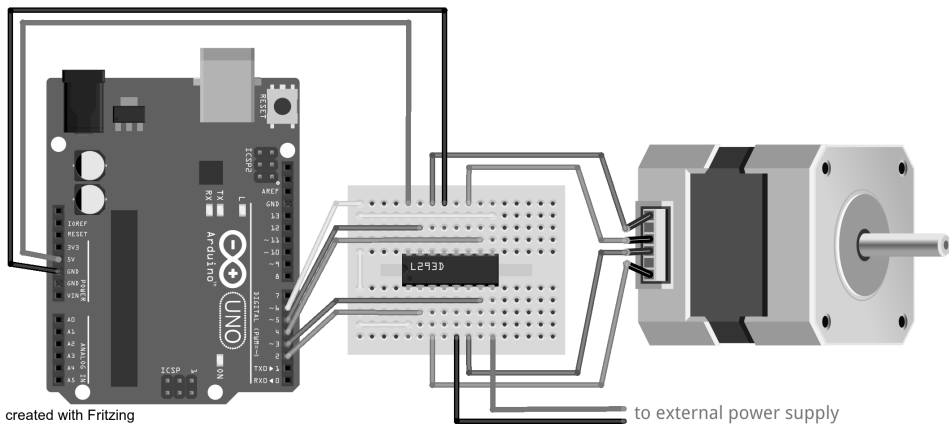



Figure 4.19 Connecting a bipolar stepper motor to the UNO with an L293D H-bridge driver.

```

if (Serial.available()) {
  Serial.readStringUntil('\n').toCharArray(line,30);
  if (strstr(line,"MOVE ")==line) {
    int val=(int)atof(&line[5]);
    myStepper.step(val);
  } else {
    Serial.println("unknown");
  }
}
}

```

Here we first include the `<Stepper.h>` library that comes with the Arduino IDE and declare a `Stepper` object called `myStepper`. The first argument is the number of steps per revolution and the next four integers are the pins to which the four coils are attached. This may differ from motor to motor, and normally this information is provided in the datasheet. Otherwise, a little experimenting with swapping numbers may lead to a moving motor. In the `setup()` function, we use the `setSpeed()` method to declare how fast we want the motor to rotate, and we also initialize the serial line. In the `loop()` function we employ our standard query-response protocol and make the motor respond to the command `MOVE`, and read the characters behind the command as the number of steps to move. This number can be either positive or negative, depending on the desired direction to rotate.

The advantage of using a bipolar stepper motor is increased torque for the same supply voltage, because there is one coil pulling the permanent magnet of the rotor and the other, which has opposite polarity, is pushing. This makes it worthwhile to describe how to drive a bipolar stepper motor.

We connect a bipolar stepper motor in the way shown in Figure 4.19. Here we use an L293D H-bridge driver, the same integrated circuit we use to provide direction control for DC motors. Based on the pin-description for the circuit shown in Figure 4.15, we immediately see that two supply wires from the left-hand side of the UNO connect to the pins for logic supply voltage and ground, and the enable pins are connected with the light-colored wire to pin D6 on the UNO. The wires connected to pins D2, D3, D4, and D5 on the Arduino

are connected to the inputs of the four half-bridge drivers on the L293D. The four leads of the bipolar stepper motor are wired to the four output terminals of the corresponding drivers. In this circuit the terminal for the positive motor supply voltage on pin 8 of the L293D is connected to an external power supply. Since internally all four ground pins of the chip are connected, we can add a wire to connect the ground terminal on the other side of the L293D of the motor to the ground terminal of the power supply.

It is remarkable that we can use the same code we used for the unipolar stepper motor for the bipolar motor as well. The same patterns, sent to the driver input pins, causes the bipolar motor to operate in full step mode. We can therefore use the same Arduino sketch as before to operate the motor, with the small reservation that we may need to swap the assignment of pins in the declaration of `myStepper`.

We need to point out that at least one coil is always excited, and that causes the motor to get warm over time, even when it is not moving. This problem can be alleviated by adding a command to turn off all coils, albeit at the expense of losing some accuracy, because the holding torque from the excited coil is absent. If we can accept the small loss in precision, we can use the pin of the Arduino connected to the yellow wire to turn on and off the outputs of the driver chip. It must, however, be pulled high with `digitalWrite(6,HIGH)` for normal operation.

Using the built-in libraries to control stepper motors is the quickest way to get started, and using well-tested libraries normally leads to robust code, but, at any rate, it is rather instructive to code the step sequence for the stepper motors *by hand* so to speak. And that is what we do in the following example, which implements the same functionality as the previous one, but provides insight into the inner workings of the stepper library. Moreover, the built-in library only implements full-step operation of the motor, where the permanent magnet is directly facing a coil, such as permanent magnet 1 in Figure 3.10, which directly faces coil A. However, by exciting coil pairs simultaneously, we can also fix the permanent magnet halfway between the coils, and move the shaft by half a step. We previously discussed this half-step mode of operation in Section 3.2.3.

For the following exercises, we assume that we have a bipolar stepper motor connected to the Arduino with an L293D H bridge, as shown in Figure 4.19. We first implement the same functionality that the stepper library provides, and later show how to easily add other modes. The full-step mode is implemented in the following code:

```
// Stepper controller by hand, V. Ziemann, 170624
char line[30];
int settle_time=2; // milli-seconds
int stepcounter=0;
const int PA=2;
const int PB=3;
const int PC=4;
const int PD=5;
const int ENABLE=6;
void set_coils(int istep) { //.....full-step mode
    bool patA[]={1,1,0,0}; // or {1,0,0,0}
    int pat_length=4;
    int ii;
    istep=istep % pat_length;
    if (istep < 0) istep+=pat_length;
    digitalWrite(PA,patA[istep]);
    ii=(istep+2) % pat_length;
```

```

    digitalWrite(PB,patA[ii]);
    ii=(istep+3) % pat_length;
    digitalWrite(PC,patA[ii]);
    ii=(istep+1) % pat_length;
    digitalWrite(PD,patA[ii]);
    delay(settle_time);
}
void setup() { //.....setup
    Serial.begin (9600);
    while (!Serial) {}
    pinMode(PA,OUTPUT);
    pinMode(PB,OUTPUT);
    pinMode(PC,OUTPUT);
    pinMode(PD,OUTPUT);
    pinMode(ENABLE,OUTPUT);
    digitalWrite(ENABLE,HIGH);
}
void loop() { //.....loop
    if (Serial.available()) {
        Serial.readStringUntil('\n').toCharArray(line,30);
        if (strstr(line,"MOVE ")==line) {
            int steps=(int)atof(&line[5]);
            if (steps > 0) {
                for (int i=0;i<steps;i++) set_coils(stepcounter++);
            } else {
                for (int i=0;i<abs(steps);i++) set_coils(stepcounter--);
            }
        } else if (strstr(line,"STEPS?")==line) {
            Serial.print("STEPS "); Serial.println(stepcounter);
        } else if (strstr(line,"STEPS ")==line) {
            stepcounter=(int)atof(&line[6]);
        } else if (strstr(line,"DISABLE")==line) {
            digitalWrite(ENABLE,LOW);
        } else if (strstr(line,"ENABLE")==line) {
            digitalWrite(ENABLE,HIGH);
        }
    }
}
}

```

First we declare some global variables, such as `settle_time`, the time to wait between changing coil excitations, and the `stepcounter`, which keeps track of the distance traveled. PA through PD are the pins to which the coil terminals are connected, and `ENABLE` connects to the enable pin of the H-bridge driver and can be used to de-excite all coils to prevent them from overheating. Then the `set_coils()` function, which will excite the coils in the correct pattern, is declared. Inside the function, we first define the excitation pattern `patA` for coil A. In this case it is 1100, the pattern in which two coils are always excited, and and provide the larger torque. Alternatively we can also declare the pattern 1000 which would result in the single-coil excitation pattern. See Section 3.2.3 for a discussion. The variable `pat_length` is the number of different steps, four in this case. Next we calculate the remainder with respect to `pat_length` of the input variable `istep` to determine what

the new state of the pattern excitation is, and write the corresponding entry in the array `patA` to the output that is connected to terminal PA. Next we calculate the entry in `patA` that is shifted by two time slots, also modulo the `pat_length`, and write the corresponding entry to output pin PB. In the same fashion, we set the remaining two output pins, PC and PD, to the entry in `patA` that is shifted by one or three time-slots, respectively. Finally, we wait a short time, given by the variable `settle_time`. This function implements a single step, but in order to move a larger number of steps, we have to call this function repeatedly.

The remainder of the sketch consists as usual for Arduino sketches of a `setup()` function, where we initialize serial communication and declare the output pins to control the motor as OUTPUT. Finally we enable the motor driver, by setting the `ENABLE` pin HIGH. In the `loop()` function we use the often-used construction to read from the serial line, and decode the command with the `strstr()` function. If the command `MOVE` is received, we decode the rest of the line as the integer `steps`, the number of steps we want to move the stepper motor. This number can be positive or negative, depending on the desired direction in which to move. If it is positive, we call the `set_coils()` function the required number of times while incrementing the variable `stepcounter`, in order to keep track of the currently applied step as well as the accumulated number of steps. If `steps` is negative, we call `set_coils()` the necessary number of times while decrementing the variable `stepcounter`. The other commands read and set the `stepcounter` variable with `STEPS?` or `STEPS nnn`, respectively. The commands `ENABLE` and `DISABLE` turn the driver stage of the L293D on and off, which may be convenient to prevent overheating of the coils during long times of idleness. This program moves the motor back and forth. Implementing speed control by adjusting `settle_time` is left as an exercise.

The previous sketch implements full-step mode, but changing it to half-step mode only requires changing the `set_coil()` function to produce the pattern for half stepping, which, according to Section 3.2.3, is given by 11100000. The replacement for `set_coil()` implementing half-step mode is

```
void set_coils(int istep) { //.....half-step mode
    bool patA[]={1,1,1,0,0,0,0,0};
    int pat_length=8;
    int ii;
    istep=istep % pat_length;
    if (istep < 0) istep+=pat_length;
    digitalWrite(PA,patA[istep]);
    ii=(istep+4) % pat_length;
    digitalWrite(PB,patA[ii]);
    ii=(istep+6) % pat_length;
    digitalWrite(PC,patA[ii]);
    ii=(istep+2) % pat_length;
    digitalWrite(PD,patA[ii]);
    delay(settle_time);
}
```

which is very similar to the previously discussed full-step version. The only difference is the array `patA`, which now contains the excitation pattern for the half-step mode. The variable `pat_length` is increased correspondingly to eight, and the steps to excite the other pins are 4, 2, and 6. Remember, this corresponds to 180, 90, and 270 degrees of the eight time-slots. Depending on the sense of wiring for the coils, we may have to swap the cables, or, as an alternative, exchange the step increment in the `set_coils()` function. Now we can full step

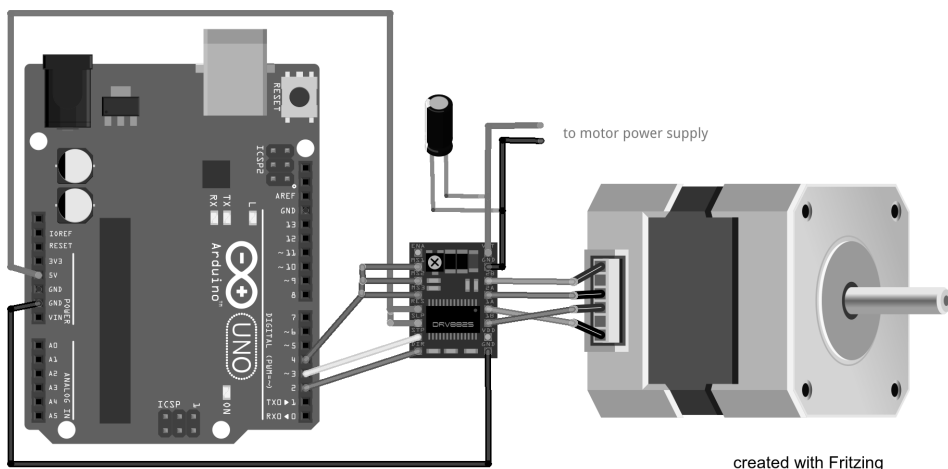


Figure 4.20 Connecting a bipolar stepper motor to the UNO with a DRV8825 stepper motor driver with microstep capability.

and half step the motors, even do so by hand, but for the micro-stepping modes we need additional hardware to control the excitation of the coils more accurately.

A driver circuit that implements these microstepping modes is the DRV8825. Interfacing a bipolar stepper motor with an Arduino using this driver is shown in Figure 4.20. Before connecting the motor to the circuit, we need to match the driver to the motor by adjusting the maximum current by which the coils are excited. So we start from the configuration shown in Figure 4.20, but *without* the four leads to the motor in place. The motor power supply must provide between 8.2 and 40 V, and we add a 100 μ F capacitor to stabilize the supply voltage. We then connect the, normally black, ground lead of a multimeter to digital ground on the motor driver and connect the other, normally red, lead of the multimeter to the tip of a screwdriver. We use the latter to adjust the potentiometer on the top left of the driver breadboard until the voltage shown on the multimeter is half the desired current limit for the motor, as specified in the datasheet of the DRV8825. We illustrate this is Figure 4.21, where we see the multimeter on the bottom left and the DRV8825 placed on a small breadboard with the screwdriver touching the potentiometer on the driver board. The multimeter in this case shows a voltage of 0.56 V, which implies that the current limit for the driver is 1.12 A.

Once we have set the maximum current, we connect the motor to the terminals labeled 1A, 1B and 2A, 2B on the driver circuit. Here we chose to connect the first and third wires on the motor to the B-pins, and the second and fourth to the A-pins, which accounts for the crossed wires on the lower motor coils. Ground and positive voltage of the motor power supply are connected to the two terminals on the top right, and the digital ground is connected to the Arduino ground connector with the black wire. One wire connects UNO pin D2 to the direction pin at the lower left of the DRV8825, and another wire connects D3 to the step input. A wire, connected to pin D4, is used to select the microstepping mode. Several modes such as full-step, half-step, 1/4, 1/8, 1/16, and 1/32 stepping are available. See the datasheet for the details. Here we only implement two modes. If the three mode pins M0, M1, and M2 are pulled to ground, the driver operates in full-step mode. If the three mode pins are pulled to the logic supply voltage, the microstepping mode with 32 microsteps

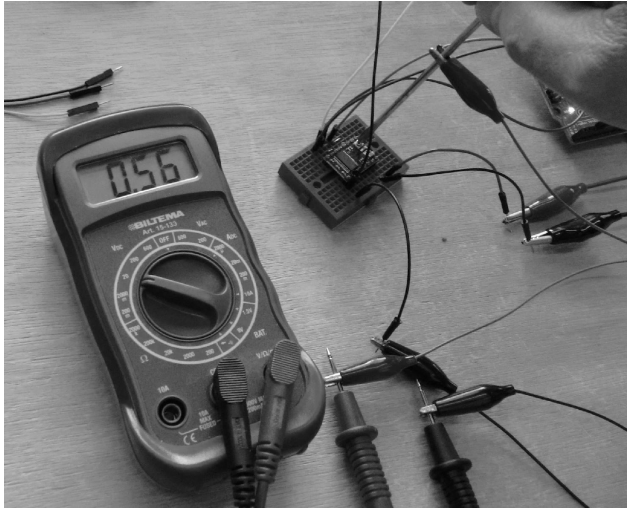


Figure 4.21 Adjusting the maximum current on the DRV8825 breadboard.

is selected. Thus by toggling UNO pin D4 we can switch between full- and micro-stepping mode. The enable pin on the top left of the driver is left unconnected, because it is internally pulled low to enable the driver by default. Moreover, one wire, connected to the 5 V supply of the UNO, pulls the reset and sleep pins of the driver high, thus permanently enabling the driver.

In order to move the motor, we use a sketch for the Arduino UNO that suitably changes the states of the direction, step, and mode pin. This is done with the following code.

```
// Stepper controller with DRV8825, V. Ziemann, 170626
char line[30];
int settle_time=30,stepcounter=0;
const int DIR=2; // direction pin
const int STEP=3; // step pin
const int MODE=4; // mode pin, LOW=FULLSTEP, HIGH=MICROSTEP
void setup() { //.....setup
    Serial.begin (9600);
    while (!Serial) {}
    Serial.println("starting");
    pinMode(DIR,OUTPUT); digitalWrite(DIR,LOW);
    pinMode(STEP,OUTPUT); digitalWrite(STEP,LOW);
    pinMode(MODE,OUTPUT); digitalWrite(MODE,HIGH);
}
void loop() { //.....loop
    if (Serial.available()) {
        Serial.readStringUntil('\n').toCharArray(line,30);
        if (strstr(line,"MOVE ")) {
            int steps=(int)atof(&line[5]);
            if (steps > 0) {
                digitalWrite(DIR,LOW);
                for (int i=0;i<steps;i++) {
```

```

        stepcounter++;
        digitalWrite(STEP,HIGH);
        delayMicroseconds(settle_time);
        digitalWrite(STEP,LOW);
        delayMicroseconds(settle_time);
    }
} else {
    digitalWrite(DIR,HIGH);
    for (int i=0;i<abs(steps);i++) {
        stepcounter--;
        digitalWrite(STEP,HIGH);
        delayMicroseconds(settle_time);
        digitalWrite(STEP,LOW);
        delayMicroseconds(settle_time);
    }
}
} else if (strstr(line,"STEPS?")) {
    Serial.print("STEPS "); Serial.println(stepcounter);
} else if (strstr(line,"STEPS ")) {
    stepcounter=(int)atof(&line[6]);
} else if (strstr(line,"WAIT?")) {
    Serial.print("WAIT "); Serial.println(settle_time);
} else if (strstr(line,"WAIT ")) {
    settle_time=(int)atof(&line[5]);
} else if (strstr(line,"MICROSTEP")) {
    settle_time=30;
    digitalWrite(MODE,HIGH);
} else if (strstr(line,"FULLSTEP")) {
    settle_time=2000;
    digitalWrite(MODE,LOW);
}
}
}
}

```

At the top of the sketch we declare a number of variables, such as the `settle_time` and the `stepcounter`, but also the used pins. In the `setup()` function we initialize serial communication, declare the control pins for the stepper driver as output, and initialize their state. By pulling the mode pin high, we select the microstepping mode. The `loop()` function uses the query-response construction in which the first command deciphers the `MOVE` command and interprets the characters following it as the number of `steps`. Then the code branches, depending on the sign of `steps`. If it is positive, we pull the direction pin `HIGH` and then increment the stepcounter. Then the `STEP` pin is pulled high and low, with a small delay in between. Note that we use the `delayMicroseconds()` function, because the rotor moves very little between the microsteps, and we can afford to reduce the waiting time between steps to achieve smooth motion of the motor. But in any case, the values for the waiting time should be adapted to the actual motor connected. If `steps` is negative, we pull the direction pin `LOW` and decrement the `stepcounter` before toggling the `STEP` pin first `HIGH` and then `LOW`. After handling the `MOVE` command, we add the bookkeeping commands to read and set both `stepcounter` and `settle_time`, and finally, select the operation mode. If the command is `MICROSTEP`, the `settle_time` is set to $30\mu\text{s}$ and the `MODE` pin is pulled

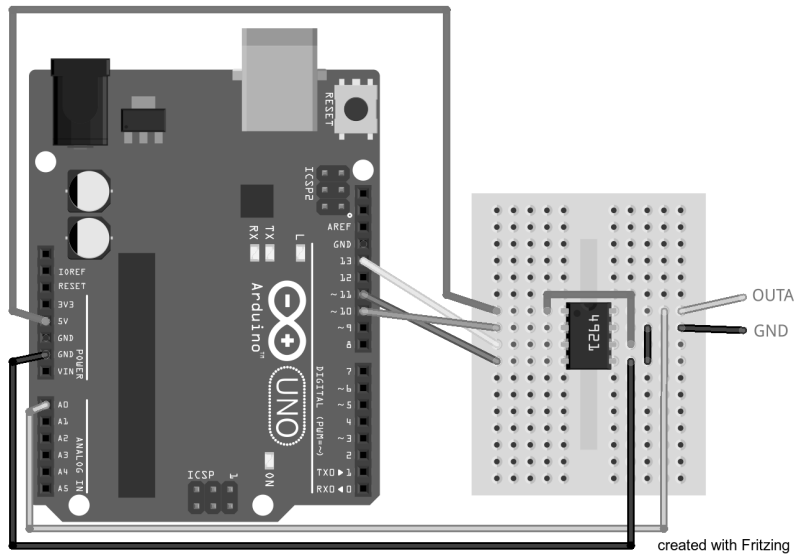


Figure 4.22 Connecting an MCP4921 12-bit DAC to the Arduino UNO.

HIGH. If the command is `FULLSTEP`, we choose a larger `settle.time` and pull the `MODE` pin LOW.

After being able to control various types of motors, we move on to controlling analog voltages.

4.5.5 Analog voltages

In this section we connect an MCP4921 12-bit digital-to-analog converter, which supports SPI-communication, to an Arduino UNO, and discuss a sketch to set the output voltage of the DAC using our query-response protocol. This could, for example, be used to set the control voltage of a power supply or any other device that requires an analog voltage level as input. The circuit that achieves this is shown in Figure 4.22, where we see the UNO on the left and the supply and ground connections to the MCP4921. The bright wire from pin D13 connects the respective clock pins, the wire from pin D11 connects the MOSI pins. It carries the data from the UNO to the DAC. The cable from pin D10 connects the chip select pins. In this case there is no information read back from the DAC to the UNO, and we do not need a MISO wire. As a matter of fact, there is not even a MISO pin on this DAC. On the DAC we connect the reference voltage pin to the 5 V supply voltage and pull the `LDAC` pin low to transfer the new voltage immediately to the output at the end of the SPI transaction. We also add a wire from the output pin of the DAC to analog input A0 of the UNO, which allows us to read back and verify the output voltage. The software running on the UNO is the following.

```
// MCP4921 DAC, V. Ziemann, 170801
#include <SPI.h>
#define CS 10
void setup() { //.....setup
    Serial.begin(9600);
```



```

while (!Serial){;}
pinMode(CS,OUTPUT); digitalWrite(CS,HIGH);
SPI.begin();
SPI.setBitOrder(MSBFIRST);
}
void loop() { //.....loop
char line[30];
if (Serial.available()) {
  Serial.readStringUntil('\n').toCharArray(line,30);
  if (strstr(line,"DAC ")==line) {
    uint16_t val=(int)atof(&line[3]);
    val|=(B0011 << 12); // 0=chA,0=unBuf,1=x1,1=ON
    digitalWrite(CS,LOW);
    SPI.transfer(highByte(val));
    SPI.transfer(lowByte(val));
    digitalWrite(CS,HIGH);
  } else if (strstr(line,"A0?")==line) {
    Serial.print("A0 ");
    Serial.println(analogRead(0)*5.0/1023.0);
  }
}
}
}

```

The sketch starts by including the header file `SPI.h` for the SPI functionality, and declares the chip-select pin `CS`. In the `setup()` function, we initialize serial and SPI communication, as well as declaring the `CS` pin as output, and set its value to the SPI idle-state, which is `HIGH`. In the `loop()` function we use the query-response protocol and respond to the command `DAC n` to set the DAC. First we read the desired 12-bit word from the serial line into the variable `val`, and then add four configuration bits `B0011` to the most significant end, where the first bit corresponds to the channel number, which is zero for the single-channel DAC MSP4921, and the second bit declares that we use unbuffered voltages. With the third bit we choose the internal amplification level to be unity, and the fourth bit enables output. Once the 16 bits in the variable `val` are assembled, we can pull `CS` low to initiate communication and then transfer the 16 bits in two chunks of 8 bits each to the chip with two calls to `SPI.transfer()` before pulling `CS` high again to end the communication and internally transfer the voltage to the output buffer of the DAC. The command `A0?` reads back the voltage, already converted to volts. This simple sketch will allow us to set any device that requires analog control voltages as input, and also read back the voltage to verify correct operation.

Now we have a number of methods to control voltages, switches, and motors, but we will wrap up this section by discussing a few means to attract the attention of a user, which is useful in case of malfunction or another event that requires human intervention.

4.5.6 Human attention actuators

Early in this chapter we discussed an LED as a human attention actuator. An LED obviously addresses the eye. But we also have ears, and can also attract their attention, for instance, in case some unexpected error happens, with an acoustic signal. The simplest example is a small loudspeaker that we can either use as a simple buzzer, or even code with simple

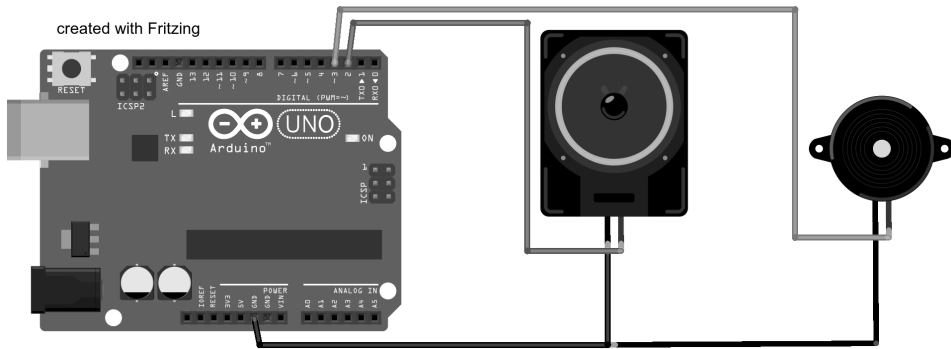


Figure 4.23 Connecting a speaker and a piezo buzzer to the Arduino.

acoustic patterns to identify certain things. For instance, if a moving robot gets stuck in a corner, we may want it to sound like a siren.

We interface a speaker to the UNO by connecting one terminal to ground on the Arduino and the other to a digital output pin, say pin 2, as shown in Figure 4.23. In order to produce a tone of 440 Hz for 1000 ms, we use the Arduino command

```
tone(2,440,1000);
```

placed anywhere in the Arduino sketch. Initialization of the pin as digital output is done automatically. Replacing the 2 by a 3 will sound the piezo buzzer shown in Figure 4.23.

After being able to interface a number of sensors and actuators over the serial line, we need to take a closer look at the options to communicate to the host computer and how to receive the data there.

4.6 COMMUNICATION TO HOST

So far, we used only the serial monitor in the Arduino IDE to communicate with the Arduino, but normally we also want to communicate from other programs, and that is the topic of this section.

4.6.1 RS-232 and USB

The serial monitor in the Arduino IDE uses an RS-232 protocol that is transported over a USB line. The translation happens automatically, and plugging the USB cable into the host computer causes the latter to automatically create a device file (`/dev/ttyUSBx` or `/dev/ttyACMx` on Linux, `COMx` on Windows, and `/dev/tty.usbserial-xxx` on a Mac) that represents the other end of the communication channel from the UNO to the host computer. We can then use any terminal program on the host computer to connect to that device file, but have to keep in mind that we use the same baud rate that we specify in the `setup()` function on the Arduino. To find out the device file to which the Arduino is connected, we can use the Arduino IDE and inspect the **Tools**→**Port** menu, and write the name of the device file down. In my case it was `/dev/ttyACM0`. After we close the Arduino IDE, we can use any terminal-emulator program, such as `putty` on Windows, or the very basic program `screen` on Linux or Mac, to connect to the UNO. In my case the command

```
screen /dev/ttyACM0 9600
```

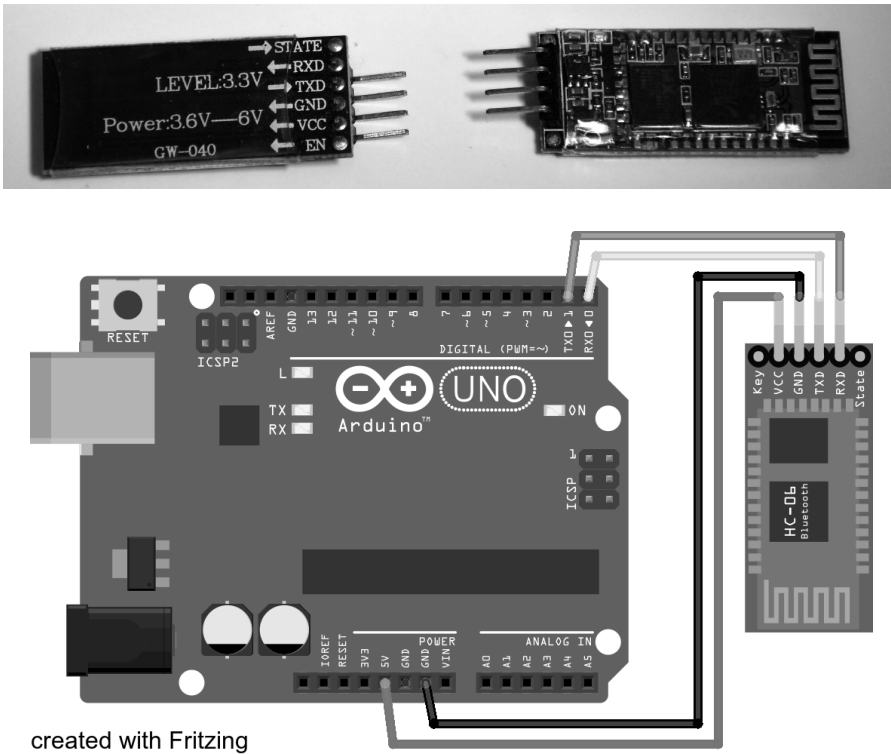


Figure 4.24 Front and back side of the HC-06 Bluetooth dongle and the connection to the Arduino.

does the trick, and we can send commands to the Arduino and receive the response in the terminal window. The `screen` program can be exited by pressing `Ctrl-a-k` and confirming the question of whether one really wants to exit with `y`. The number 9600 is the baud rate used for the communication and must match the number in the `Serial.begin()` statement in the Arduino sketch. Note that 9600 is the default baud rate and could be omitted in the `screen` command.

4.6.2 Bluetooth

Bluetooth functionality can be added by attaching an HC-06 Bluetooth dongle to ground, power, and the Arduino pins 0 and 1. On the top left of Figure 4.24 we see those four pins labeled RXD, TXD, GND, and VCC. The latter two are connected to the respective power supply connections, and RXD on the HC-06 is connected to pin 1, labeled TX on the Arduino, which is illustrated on the bottom of Figure 4.24. On this connection the information flows from the Arduino to the HC-06. The pin labeled TXD on the HC-06 must be connected to pin 0, labeled RX on the Arduino, and the information flows from the HC-06 to the Arduino on this line. This crossed connection from TXD to RX and vice-versa is equivalent to a null-modem cable. After connecting the HC-06 in this way, all communication is sent via the HC-06 and the USB connection in parallel. For reliable operation, the USB link should not be used while Bluetooth is in operation.

On the host computer, we have to *pair* a new Bluetooth device with the host computer. On Windows, this is done in the Bluetooth administration program. On some Linuxes, similar user interfaces exist, but we can always pair Bluetooth devices using a number of command-line programs. First we need to find out whether the host computer has Bluetooth capabilities, by the command `hcitool dev`, which should report at least one device, normally called `hci0`. Then we scan the surroundings for Bluetooth devices with the command `hcitool scan`. If the HC-06 is powered we should see at least one device with a line `xx:xx:xx:xx:xx:xx HC-06` where the six-byte string is the MAC address of the Bluetooth device. To establish pairing, we have to log in as `root` user with the command `su` or `sudo` and call the program `bluetoothctl -a`, which reports the controller and known devices. At the `[bluetooth]#` prompt we initiate a search for new devices with `scan on`, which reports all known devices. The following two commands establish the pairing

```
trust xx:xx:xx:xx:xx:xx
pair xx:xx:xx:xx:xx:xx
```

where `xx:xx:xx:xx:xx:xx` is the MAC address of the HC-06 dongle we want to pair. During the previous actions we are prompted for a PIN number, and unless we have changed the default on the HC-06, we use 1234. Then we exit the `bluetoothctl` program and create the device file for the serial communication by issuing

```
rfcomm bind 0 xx:xx:xx:xx:xx:xx
```

which creates a device file `/dev/rfcomm0` that has the same functionality as the `/dev/ttyACM0` device file we used earlier to communicate with the Arduino. Therefore we can again use the `screen` command to communicate with the Arduino, but using `/dev/rfcomm0` as first argument to the `screen` command instead. Note that the HC-06 is configured to communicate with 9600 baud. This can be changed using AT commands, but we will not discuss this further, and assume that all Bluetooth communication using the HC-06 is done at 9600 baud. Once we are done using the Bluetooth serial link, we should take it down by issuing the command `rfcomm release 0`, which will remove the `/dev/rfcomm0` device file, and we will no longer be able to use it. The bottom line is that we can communicate using Bluetooth in much the same way as using native RS-232 or USB. We only need to pair the device and the host computer once, and then create the device file using the `rfcomm bind` command before using the serial line, and delete it with the `rfcomm release` command once we are finished.

Apart from the channels based on serial communication, the more modern Arduino clones such as the ESP8266 can use normal WLAN-based communication based on network sockets. We will discuss this nifty feature in the next section.

4.6.3 WiFi

For this section, we use a NodeMCU system that is connected to the host computer, with a USB cable for programming, and the device support package as described in Section 4.2 is installed in the Arduino IDE. Our task is to connect the NodeMCU to a local wireless network; we assume it is called `MyHomeNet` and that it can be reached by other computers connected to the same network. All these computers will be able to query the measurement values of sensors connected to the NodeMCU. To better understand the setup, we briefly discuss some general features of computer networks.

Everybody is probably familiar with the fact that computers on the Internet are identified by IP numbers, such as `192.168.10.200`. Because the numbers are difficult to remember, there are also aliases such as `www.cnn.com`, and the translation is done by so-called

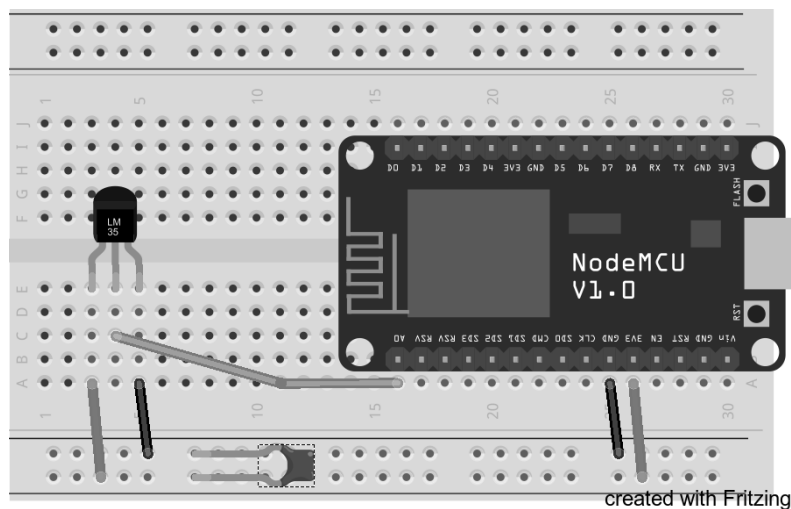


Figure 4.25 The NodeMCU with an LM35 temperature sensor.

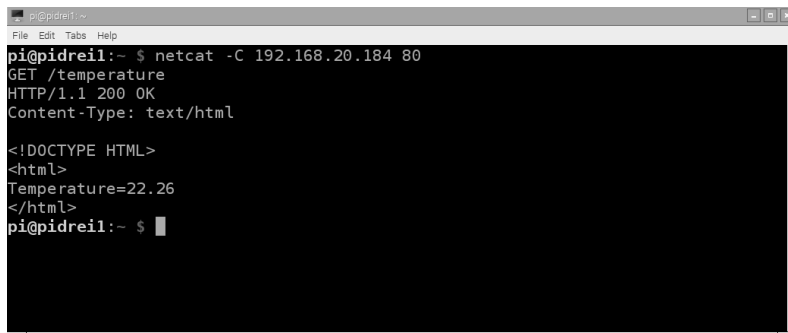
domain name resolution or DNS servers. On our simple network, we assume that we can keep track of the numbers, and all sensors and host computers are connected to the (class-C) network 192.168.10.nn where nn is a number between 2 and 254. The numbers 0 and 255 are reserved for special purposes, and we assume that the router that connects our network to the outside world (“the Internet”) has the IP number 192.168.10.1. Note that the addresses starting with 192.168 are private numbers and can be used by anyone, provided that there is a router that separates this network from the Internet. But we confine all our communication to within the 192.168.10 network, and identify each computer by its IP number. Each computer can potentially provide different services, such as running a web server, a measurement server, a mail server, or a server to allow logging onto the computer with standard protocols such as `telnet`, `ftp`, or `ssh`. The different services that a computer provides are identified by a *port number*. As an analogy, it may help to think of the IP number as a street address of an apartment building and the port numbers as the apartment numbers. The communication with a server that provides a service on a computer therefore requires the specification of the IP number and the port number.

But how does a computer know its own IP number? There are two ways to specify this: by configuring the network setup manually and assigning the IP number explicitly. We use the second way and dynamically acquire the IP number via the *dynamic host configuration protocol*, DHCP, which is much more convenient, provided such functionality is available on a given network. In most networks with a wireless router, the router provides this service and one only has to tell a computer to use DHCP. In that case, the computer sends a request for an IP number at power-up. The DHCP server responds with an IP number that is then assigned to the newly connected computer. We assume that a DHCP server is running on our 192.168.10 network and the NodeMCU is also configured by default to use DHCP.

Wireless networks are normally protected from unauthorized use by encrypting the communication. Most networks use an encryption standard called WPA that requires entering a password to connect to the network. We assume that the “MyHomeNet” WLAN is of that type. Having covered the networking basics, we are ready to connect our Arduino clone, the NodeMCU, to the WLAN.

First we discuss the running of a simple web server on the NodeMCU, that uses the standard `http` protocol to communicate with other computers on the network. The following program shows how to set up a web server that listens on the default `http` port number 80. It provides access to temperature measurements and allows one to control the brightness of an LED. In our example we query the temperature measured by the NodeMCU, by directing a browser such as Mozilla Firefox to the address of the NodeMCU at `http://192.168.10.nn/temperature`, and the NodeMCU returns a web page with the temperature measured by an LM35 temperature sensor. The brightness of the built-in LED is controlled by adding a value to the address after a question mark, such that `http://192.168.10.nn/led?b=1023` sets the brightness to its maximum value. The circuit schematic is shown in Figure 4.25, and the code that brings it to life is the following.

```
// Web server to read temperature and set brightness, V. Ziemann, 170911
#include <ESP8266WiFi.h>
const char* ssid      = "MyHomeNet";
const char* password = ".....";
WiFiServer server(80); // server listens on http port 80
void setup() {
    Serial.begin(115200); delay(10);
    Serial.print("Connecting to "); Serial.println(ssid);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500); Serial.print(".");
    }
    Serial.print("\nWiFi connected and ");
    server.begin();
    Serial.print("server started at ");
    Serial.println(WiFi.localIP());
}
void loop() {
    WiFiClient client = server.available();
    while (client) {
        while(!client.available()){delay(1);}
        String req = client.readStringUntil('\r');
        client.flush();
        client.print("HTTP/1.1 200 OK\r\nContent-Type: text/html");
        client.print("\r\n\r\n<!DOCTYPE HTML>\r\n<html>\r\n");
        if (req.indexOf("/temperature") != -1) { // read sensor
            float temp=100*3.3*analogRead(0)/1023;
            client.print("Temperature="); client.println(temp,2);
        } else if (req.indexOf("/led") != -1) { // brightness of LED
            int i1=req.indexOf("?"); int i2=req.indexOf("HTTP");
            String payload=req.substring(i1+1,i2-1);
            if (i1>0) analogWrite(D4,1023-payload.toInt());
        } else {
            Serial.println("invalid request");
            return;
        }
        client.println("</html>"); delay(1);
        client.stop();
    }
}
```



```

pi@pidreil:~ $ netcat -C 192.168.20.184 80
GET /temperature
HTTP/1.1 200 OK
Content-Type: text/html

<!DOCTYPE HTML>
<html>
Temperature=22.26
</html>
pi@pidreil:~ $

```

Figure 4.26 The communication with a web server showing the HTTP header.

```

    }
}

```

In this example we first include the header file with the `ESP8266WiFi.h` library information, define the access parameters `ssid` and `password` for the wireless network, and instantiate the `server` to listen on the default http-port 80. In the `setup()` function, the serial line is opened to be able to listen to debug information. Then the connection to the wireless network is established with the `WiFi.begin()` function. Inside this function, the log-on to the wireless network and the communication with the DHCP server is handled. Once `WiFi.status` reports that the connection is established, we print information about the connection, such as the acquired IP number, to the serial line, and start the server with the call to the `server.begin()` function. In the `loop()` function a similar construction as before receives a request from a client, stores the request in the variable `req`, returns a standard http header with `Content-type` to the client and then checks whether the request `req` contains the string `/temperature`. If it does, the sketch determines the temperature with a call to the `analogRead()` function, and writes the value back to the client. Note that this time we use the `indexOf` method of a `String` to determine whether a substring is present. If `req` contains `/led`, we determine the position of the question mark and trailing characters `HTTP` and extract everything in between as the `payload`. After conversion to an integer, we use the `payload` to set the brightness of the built-in LED. We can inspect how `req` is constructed and why we have to find the trailing `HTTP` by printing with `Serial.println(req)`. If an unknown request arrives, we are notified on the serial line. Finally, we add the concluding `</html>` tag and close the connection to the client. We point out that in this simple example we do not check the validity of the `payload`, which poses a potential security risk, if the system is reachable from the Internet.

A few words are needed about the cryptic HTTP header that the NodeMCU returns to the calling browser. When a browser, such as Firefox, connects to a web server on the default port 80, it first sends the name of the requested web page, such as `GET /temperature` in the above example, as an HTTP-GET request. Before returning the requested web page, the server sends some meta-information, such as a status code, and what type of information comes next. The status code for a properly understood request is 200, and for a missing page it is 404, a number probably everyone has seen as a response to a typo in the specification of a web page. The type may be an image, a media file such as a video, or HTML-formatted text, which is what we specify in the sketch as `Content-Type`. After an empty line the normal HTML header with the `DOCTYPE` follows, and the information embedded in `<html>`

tags. We will discuss HTML and the structure of web pages in more detail in Section 5.7. Browsers do not render the meta-information, which therefore remains invisible, but we can eavesdrop on the communication with the `netcat` or `telnet` commands (more on those commands later in Section 5.3) by pointing them to port 80 on the NodeMCU at IP address 192.68.20.184 and issuing `GET /temperature` by hand. *All* output from the server subsequently appears in the same window. Figure 4.26 illustrates the exchange.

Setting the brightness of the LED uses the same mechanism that is used to enter credit-card numbers when shopping online. It is based on the HTTP-GET method as well, but adds a *query-string* that follows the question mark to the address. The query-string has the form `name=value`. This mechanism to send information to a web server is often used with HTML forms. It allows the user to enter text or select things from a list and then click a submit button to send it to the server, where it causes some reaction, such as sending the desired purchase—or to set the brightness of an LED.

In a second example, we configure a NodeMCU to run a server listening for connections on port number 1137, and once a connection is established, it waits for commands and then replies appropriately. The hardware is the same as in the previous example and is shown in Figure 4.25. The code that runs on the NodeMCU is the following.

```
// Socket-based measurement server, V. Ziemann, 161211
const char* ssid      = "MyHomeNet";
const char* password = ".....";
const int port = 1137;
#include <ESP8266WiFi.h>
WiFiServer server(port);
void setup() { //.....setup
  Serial.begin(115200);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500); Serial.print(".");
  }
  Serial.println("");
  Serial.print("WiFi connected to "); Serial.println(ssid);
  Serial.print("Server IP address: "); Serial.println(WiFi.localIP());
  server.begin();
  Serial.print("Server started on port "); Serial.println(port);
}
void loop() { //.....loop
  char line[30];
  float volt,temp;
  WiFiClient client = server.available();
  while (client) {
    while(!client.available()) {delay(1);}
    client.readStringUntil('\n').toCharArray(line,30);
    Serial.print("Request: "); Serial.println(line);
    if (strstr(line,"A0?")) {
      volt=3.3*analogRead(0)/1023;
      client.print("A0 "); client.println(volt);
    } else if (strstr(line,"T?")) {
      temp=100*3.3*analogRead(0)/1023;
      client.print("T "); client.println(temp,1);
    }
  }
}
```



```

    } else {
        Serial.println("unknown command, disconnecting");
        client.stop();
    }
    client.flush();
    // client.stop();
}
}

```

In this sketch we first define two character strings, `ssid` and `password`, for the name and passphrase of the wireless network. We also specify the port number, here 1137, in the following line before including the functionality for WiFi support on the ESP8266 device. Once that is included, we define a `WiFiServer` named `server` that listens on the specified port. In the `setup()` function, we start up the serial communication via the USB line that we use for logging, and then connect to the wireless network with the `WiFi.begin()` function. Once the NodeMCU is connected to the wireless network, we start the server with the `server.begin()` command. In the `loop()` function we use the same structure that we used earlier for communication via the serial line, but this time over the socket that is dynamically opened once a client computer connects to our server. The connection to the requesting computer stays open until an unknown command closes the connection. But while the connection is open, values can be requested repeatedly. If we want the connection to close immediately after the reply is sent, we can uncomment the line with `client.stop()`; immediately following the line with `client.flush()`. Note that this server can only handle one request at a time. If one computer is already served and a second computer tries to establish contact, the latter is put on hold until the first computer disconnects. This program on the NodeMCU allows us to query via the custom socket on port 1137 any parameters or measurement values that the NodeMCU has available. How to connect to it from another computer we defer to the next chapter.

Beyond the standard communication protocols there are several others available. We discuss those only briefly in the following section.

4.6.4 Other communication

In the previous sections, we used standard and easy-to-use protocols (RS-232, USB, Bluetooth, WiFi) to communicate with the microcontroller. They have the advantage that they are readily available on many host computers. There are, however, other communication channels, of which we name a few in the following paragraphs. All are supported by Arduino libraries.

One other wireless communication channel is *ZigBEE*, which is related to Bluetooth but uses a simpler protocol overhead and is optimized to consume as little power as possible, to make long-lived battery operation possible. Another option is using inexpensive *433 MHz transceivers*, which can be used to connect slave microcontrollers to their master using a simple wireless link.

MIDI is a protocol developed to interface electronic musical instruments. The physical interface is based on current loops that are galvanically decoupled through optocouplers at the input of each device. Logically the communication is very similar to RS-232 at a baud rate of 31250. Each MIDI message consists of three (or in some cases, two) bytes that have standardized interpretation, such as channel number, tone (corresponding to a specific key on a piano keyboard), and velocity (the intensity with which the key is struck).

It is possible to interface nearby hardware using infrared (IR) light, which is used in TV

remote controls to send information from the remote to the TV to change channel or volume. Incidentally, even the first generation LegoTM Mindstorm microcontrollers communicated in this way. The communication is based on an IR diode using a wavelength of 940 nm and modulating the light at a rate of 38 kHz. Sending bursts of modulated light represents either a LOW or a HIGH signal level and is used to emulate an RS-232-like protocol at low baud rates such as 2400 baud. Another protocol is RC-5, which is commonly used in TV remote controls. Receivers such as the TSOP38438 have optical filters built in and are sensitive only to a narrow band of wavelengths around 940 nm. Moreover, they demodulate the 38 kHz carrier frequency and only deliver a 3.3 or 5 V signal on their output pin, making them very easy to interface.

Note the generic structure: A communication channel is based on the low-level hardware and on a protocol stack that often consists of several layers on top of the hardware implementation. In our examples we use the convention to send a string, terminated with a question mark to signify a request. The reply then consists of the same string followed by a value. This convention defines a simple protocol for the communication. We can easily come up with other protocols, and add features such as checksums to test the integrity of the transmission. Also, I2C and MIDI communication are based on a standardized protocol where a number of bytes is transmitted and each byte signifies some particular information, such as the register to be addressed or the value that is written to the register. Of course, it is mandatory that all participants in a transaction agree on the interpretation of the bits and bytes that are transmitted, otherwise confusion will reign supreme.

So far we always assumed that some unspecified host computer, for example, a desktop computer, is available and serves as the communication partner for the microcontroller. In the next chapter we consider one specific host computer that is widely available, well documented, and inexpensive—the Raspberry Pi.

QUESTIONS AND PROJECT IDEAS

1. Convert the hex number 0x5CA3 to binary representation.
2. The ADC that reads pin A0 on the NodeMCU is not very reliable. For example, connecting A0 to GND will not necessarily give a zero ADC reading. Make an attempt to calibrate it.
3. Can you connect an MCP23017 IO-extender and a BMP180 pressure sensor to the same I2C bus?
4. Under what circumstances is it advantageous to use interrupts?
5. You need up to 128 digital input pins. Use multiple copies of the MCP23017 and explain how to connect them.
6. Inspect the datasheet of the MCP3304 and find out how to configure it to read eight unipolar input voltages. What do you need to change in the software?
7. If you need 128 unipolar analog input signals, how do you connect the MCP3304 to an Arduino? Is it possible at all, and if so, how?
8. Connect a three-color LED to an Arduino and independently control the brightness of the three colors via the serial line. Add features to directly set standard mix-colors such as orange or magenta.

9. Explain why we use (1023-value) to set the brightness in the example with the web server on page 107.
10. What is the difference between *HTTP* and *HTML*?
11. Learn about HTML forms and implement a web page with a user interface to set the brightness of the LED on the NodeMCU.
12. Mount an HC-SR04 distance sensor on top of a model-servo and scan the neighborhood for the distance to obstacles. Produce a beep, if it is closer than 20 cm.
13. Use the `tone()` function to make the Arduino sound like a *siren*.
14. Build a simple *pulse generator* that generates signals with millisecond precision. Discuss how to improve the temporal precision.
15. Build a *magnetic field sensor* with a Hall probe.
16. Build a *compass* using the magnetic field sensor on the MPU9250.
17. Build a device that senses the tilt angle with an MPU6050 or MPU9250.
18. Build a *carousel comparator* and use an MPU6050 to measure the acceleration you experience in a merry-go-round. Use a battery-powered NodeMCU to publish a web page with the data. Enhance the system by measuring and displaying the angular velocity as well. Also display the maximum and minimum values during the past 5 minutes.
19. Build a contact-free *wireless thermometer* with an MLX90614 FIR sensors and a NodeMCU that makes the measured values available on a web page or via a socket-based server.
20. Measure the wind speed with a propeller that breaks the light path in a *slotted optical switch* or between a discrete LED and a phototransistor.
21. Investigate materials for their phosphorescence by briefly flashing an ultraviolet LED and record the response of the material with phototransistors or diodes sensitive to different wavelengths.
22. Investigate how to interface the dust sensors from Section 2.3.6 to the Arduino.
23. Connect an MQ-x gas sensor to the UNO and determine the air quality.
24. Investigate how to interface the GPS sensor from Section 2.3.6 to the Arduino. Consider using the `SoftSerial` library to add additional RS-232 ports to the microcontrollers.
25. Investigate the Arduino NANO. In what way does it differ from the Arduino UNO? Discuss circumstances where you would use a NANO.
26. The ESP-01 does not have a USB interface. Investigate how to program it.

Host Computer: Raspberry Pi

The Raspberry Pi [9] is a small single-board computer that first appeared in 2012, with the intention of providing an inexpensive platform to introduce students and other interested parties to computers in general and to programming in particular.

5.1 HARDWARE

Since its first appearance, the Raspi went through several hardware revisions, and the present incarnation, model 3, appeared in February 2016. This version, shown in Figure 5.1, features a quad-core ARM central-processing unit (CPU) operating at 1.2 GHz and has 1 Gbyte RAM memory on board. This hardware base is sufficiently powerful to run a full-fledged Linux system. An external and replaceable micro-SDHC card (preferably speed class 10) serves as a hard disk to hold the operating system and user files.

But beyond the CPU, the Raspberry Pi sports a video processor that can display videos at full-HD resolution (1920×1080) via the built-in HDMI-connector. Audio output is available either via the HDMI connector or via a 3.5-mm headphone connector. Moreover, there are four USB-2 ports on board to connect peripheral components, such as USB sticks, keyboard, mice, or web cameras. Communication with the outside world is feasible via a built-in wired Ethernet port, and since version 3, the Raspi has had built-in Bluetooth (V4.1) and WiFi (802.11n).

The Raspis are very attractive due to their built-in low-level peripherals. There are 17 general-purpose input-output (GPIO) pins exposed on the board, some of which support I2C, SPI, and UART (RS-232-like) communication. Moreover, a specific audio bus (I2S) is available, as well as a high-speed CSI interface to connect the tailor-made Raspberry Pi camera, and a DSI interface to connect LCD panels. Some of these features can be used to implement the same functionality as on the Arduino, but we will not use that here, using the Raspi as a standardized host computer system instead.

5.2 GETTING STARTED

The CPU and peripherals are adequate for using a Raspi as media center via the *OpenElec* distribution, which runs the *KODI* media center software and turns a dumb TV into a smart TV. There is the *OpenWRT* distribution to turn the Raspi into an Internet router including firewall functionality. Other software packages turn it into a retro-gaming console

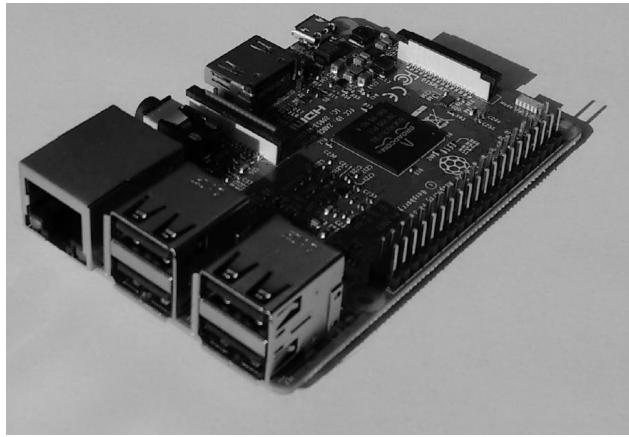


Figure 5.1 A Raspberry Pi board.

or into a network-addressed storage (NAS) server. Apart from these special-purpose uses, we can also install a general-purpose Linux operating system on the SDHC card from which we boot the Raspi. Different flavors of Linux are available, such as Ubuntu, Fedora, or OpenSuse, but the standard system that we use in the remainder of this book is based on the Debian distribution and is called *Raspbian*. In recent years, a large number of books, such as [20, 21, 22], were published with recipes of how to use the Raspi in many circumstances. Use those and others to complement the discussion in this book.

We download the Raspbian operating system from the www.raspberrypi.org web site under the Download link at the top of the page. The download is a rather large, presently 1.4 GB, image file for the full system. Near the top of the download page there are installation instructions for all common desktop operating systems. On a Linux desktop system we have to unzip the image file, which may take some time, and then transfer the image file to the SD card by issuing the command

```
dd bs=4M if=2016-11-25-raspbian-jessie.img of=/dev/sdX
```

as user **root**. We may have to prepend the **dd** (disk-duplicator) command by **sudo** depending on the flavor of Linux on the desktop computer. In the **dd** command, the destination, the output file **/dev/sdX**, must be the device file of the SD card. We can easily determine it by inspecting the system log file after inserting the SD card in the writer. Just before the end of the log file a recently inserted card is typically referenced as **/dev/sdX**, with **X** being **c** or **d** or **e**. We must make sure that you use the correct drive. If we pick the wrong one we may damage our desktop system. If unsure, we carefully follow the instructions in the installation help, which are more extensive than those given here. Once the **dd** command completes, which may take up to 15 minutes, depending on the SD card writer, we insert the SD card into the slot on the Raspi, connect a monitor to the HDMI connector, mouse and keyboard to USB connectors, and apply power via the micro-USB connector to boot the Raspi.

The normal system directly boots into the desktop system on the Raspi, and automatically starts the configuration program **raspi-config**. There we need to do some book-keeping activities, such as expanding the file system to use the entire disk. The image file we download only contains a moderate file system, but we can coax it to use the entire

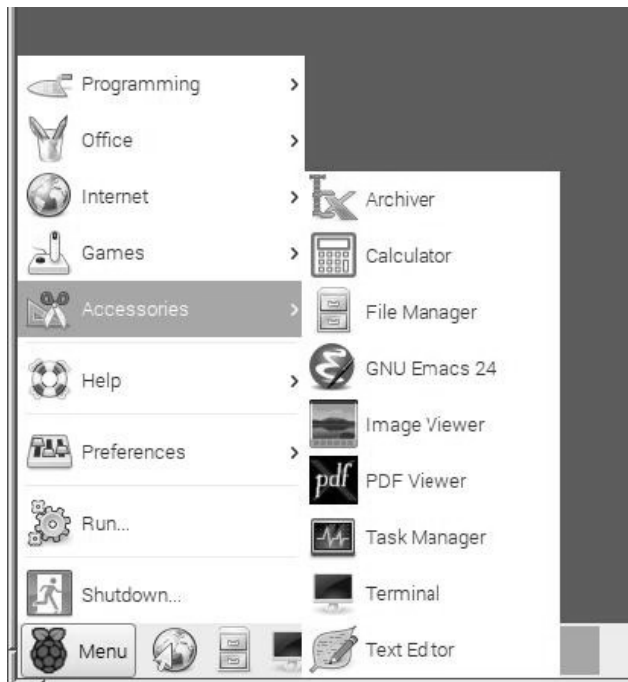


Figure 5.2 The Accessory menu behind the button with the raspberry shows a number of installed programs such as the Terminal (command) window.

space on the SD card, depending on the size of the card, which should be 8, 16, or even 32 GB. Next, we choose the localization option to select the time zone and especially the keyboard layout. Then we change the user password for the default user `pi` from the default password `raspberrypi` to something more unique, and progress to the advanced options to select a suitable `hostname` for the Raspi. Make sure to remember the new password! Then we close the `raspi-config` program and reboot.

Once the Raspi comes up again we are directly logged into the desktop system, without a password being asked. To change this behavior we start a terminal program and type `sudo raspi-config` followed by `Enter` on the command line. Under point 3 (Boot options) we then select an option *without auto-login*, and close the program before rebooting.

At this point we have a running Raspbian Linux system on our Raspi, and it is time to explore it by selecting the Menu button with the raspberry image on it. There we find all available programs, sorted according to groups such as *Programming*, with links to Java, Python, and Mathematica. The group *Office* contains links to the Libreoffice programs for word processing, spreadsheet, and presentation creation. Behind the *Internet* menu item, there are links to the Internet browser and mail client. Moreover, there are menu items for *Games* and *Accessories*. The latter, shown in Figure 5.2 contains a link to the *Terminal* program that provides a console to enter commands. We will use it extensively in later sections, but here are a few basic commands that are useful to know. When we open the terminal window, we are placed in the home directory of user `pi`, namely `/home/pi`, which we can verify by entering the command `pwd` to “print the working directory.” We can list its contents by entering the `ls` command, or its long form `ls -l`, which displays all files

and subdirectories in the present directory. In order to enter a subdirectory we use the `cd <dirname>` command to change the working directory to `<dirname>`. The command `cd ..` takes us back to the directory where we started. If we want to navigate to a directory for which we only know the absolute name that starts with a slash, such as `/usr/local`, we use `cd /usr/local`. We should play around with the programs and test them to familiarize ourselves with the new Raspi system. Either type the commands on the command line or click on the programs in the menus and explore. Finally, there is the *Preferences* menu item with programs to customize the Raspi. In particular, the link *Raspberry Pi Configuration* starts a graphical user interface with the same functionality as the `raspi-config` program we used earlier.

In the menu list, next to the Menu button with the raspberry on it, there are quick-start buttons for several programs, such as an Internet browser, file manager, and Mathematica. Programs can be added to this area by right-clicking on the menu list and selecting *Add/Remove Panel Items*. In the window that appears, we highlight **Application Launch Bar** and click on **Preferences**. The appearing window has a left side with the already present programs in the quick-launch area. On the right is a list of installed applications from which to select. Here we install the *Terminal* program from the *Accessory* group in the Quick launch area because we will use it a lot later on. On the rightmost end of the menu list, widgets for network, audio, and other features are located. We add or remove such widgets from the *Add/Remove Panel Items* window that we launch by right-clicking the menu list and pressing the *Add* button, whence a list of available widgets appears. From it we select whatever we want to add. Finally, we move the entire menu-list to our preferred location on the desktop by right-clicking on it and selecting *Panel Settings* where we choose the location of the menu list. The default location is on the top, but we can also move it to the bottom of the desktop.

In order to log on to the Raspi from our desktop computer, we need to know its IP address, which is found by executing `ifconfig` in a terminal window on the Raspi and finding the point labeled `inet addr` for the interface `eth0`. Once we know it, we log on to the Raspi via the secure shell program `ssh` from our desktop computer by typing `ssh -X pi@192.168.10.nn`, where `192.168.10.nn` is the IP number of the Raspi. The `-X` option allows us to display additional windows on the desktop computer. If this is the first time we connect, a warning message appears that the Raspi is unknown to your desktop and we should answer *yes* that all is well. Then we are presented with a request to enter the password for user `pi` on the Raspi. If we enter that correctly, we get a command prompt from the Raspi at which you can enter commands, such as `sudo raspi-config` or any other. From a Windows computer, you may use any `ssh` client program, such as `putty`, to log on to the Raspi command line. This way of logging on can be simplified significantly by closing the connection, and back on the desktop computer we enter

```
ssh-copy-id pi@192.168.10.nn
```

which presents a prompt to enter the password for user `pi` on the Raspi for the last time, and henceforth we log on to the Raspi from our desktop by typing `ssh -X pi@192.168.10.nn` and are at the command prompt of the Raspi without being asked for a password. Note that this only works from our account on the desktop computer. The `ssh-copy-id` exchanges a secret with the Raspi that is used to authenticate our account on this desktop computer and no other. Thus, logging onto the Raspi is just as safe as logging onto our desktop computer. We could now remove screen, mouse, and keyboard and always use the `ssh` command to log onto the “naked” Raspi, but for convenience we keep the peripheral devices attached a little longer until we have installed software to allow running a virtual screen via the network.

The default system already comes loaded with a good selection of programs, but we can

add more programs, and the process to do so is extremely simple. We discuss this in the next section.

5.3 INSTALLING AND USING NEW SOFTWARE

Before installing new software we first update the current system, which is based on an image file prepared some time ago. To do this we start a terminal program and enter the following two commands:

```
sudo apt-get update
sudo apt-get upgrade
```

where the first command updates the database of installed programs and finds out whether newer versions are available. The second command installs the upgrades after a question to approve its selection. First, the newer versions are downloaded and then installed. This may take quite a while—about 30 minutes if the time since the last update or the creation of the initial installation media is long ago. Note that the system-administration command is called **apt-get**, and we have to run it with superuser privileges by prepending **sudo**. After the upgrade is complete and the program returns to the command line prompt, we have an up-to-date system and are ready to install additional software.

Since we want to test the interface with the Arduino, we need to install the simple terminal emulator program **screen** by issuing the command

```
sudo apt-get install screen
```

and after confirming that some kilo-bytes are downloaded, the program is installed in a few seconds. We assume that we have an Arduino UNO programmed with the query-response sketch from page 64 and connect the USB cable from the UNO to a USB port on the Raspi. We then check the device name of the serial line by issuing **dmesg** in a terminal window, and inspect the output near the end. In my case, one line contains the string **ttyACM0**, but **ttyUSBn** with any number **n** can also occur, depending on the serial-to-USB converter installed on the UNO. Since the Arduino sketch uses 9600 baud for the serial line, we connect to it with

```
screen /dev/ttyACM0 9600
```

on the command line of the Raspi. Then we query the UNO by typing **A0?** followed by **Enter** in the screen terminal-window. The UNO should respond with **A0 nnn** where **nnn** is the value measured by the ADC on the UNO. Congratulations, we have established a communication link from the Raspi to the Arduino UNO. Once we are done with getting data from the UNO, we close the connection by pressing **Ctrl-a-k** in the screen window, confirm the question of whether we really want to quit by **y**, and return back to the command prompt.

The **screen** program and in fact most programs on a Unix or Linux system bring their own help system, called manual pages, which can be accessed from the command line by typing **man <program name>**—for example, **man screen** in the present case. This displays information on how to use the program in the terminal window. In particular, all command line switches are explained. We exit from the **man** program by typing “q” in the window. Using **man** only works if the program name is known, but we can use the command **man -k <keyword>** to search the system for manual pages that have to do with **<keyword>**. The latter command works if we initialize a database once with **sudo mandb**. Check with **man mandb** to see what it really does!

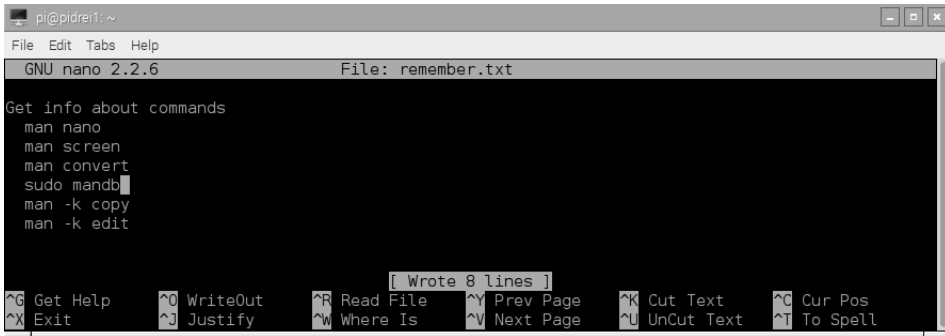


Figure 5.3 The **nano** text-editor with available commands listed on the bottom.

Later on, we will often have to edit configuration files or to write program code. The edited files are usually plain text files, so-called ASCII files, and we create and edit them with an editor program such as **nano**, **vi**, or **emacs**. The first two are already installed on the system, and since **vi** has a steep learning curve, **nano** is a good choice to start, unless we have another preference. We start **nano** from the command line by typing **nano <filename>**, and after pressing **Enter**, the terminal window opens with the contents of the file, if it exists, or an empty file, if it does not exist. Figure 5.3 shows **nano** after we added some text. On the top are the usual menus for *File* and *Edit* operations, and at the bottom of the window the most important commands are listed, especially **Ctrl-x** to exit the program and **Ctrl-g** to open the built-in help system with more information about using **nano**. Here, both small and capital letters work in conjunction with the **Ctrl** key to execute commands, such as **Ctrl-k** and **Ctrl-u** to cut and paste the line with the cursor. Note that once a text file is written to disk, we can also view it using the **less** command or with **cat**. Please check the respective manual pages for additional information.

Any program, library, or other package that is available in the official Raspberry Pi repositories can be installed in the same way by first updating the database with **sudo apt-get update**, which is only necessary once per session and then installing the package with **sudo apt-get install <package-name>**. We use this newly won information to install the MATLAB-clone **octave** [23] by issuing

```
sudo apt-get install octave
```

on the command line. After a lengthy installation that takes a few minutes, depending on the download speed, the program is ready to use. We start it from the command line by the command **octave**, followed by **Enter**. Octave greets the user with its version number and some copyright information before the prompt **octave:1>** appears. There we can enter MATLAB-compatible commands like **A=[1,2;3,4] ; B=inv(A)**, which defines a 2×2 matrix *A*, inverts it, and stores the result in the variable *B*. Later we will also connect to the Arduinos from within octave.

In order to be able to display and convert graphic files from one format to another, the **imagemagick** package is invaluable. We install it by executing

```
sudo apt-get install imagemagick
```

from the command prompt. Once it is installed, we can display almost any graphics file with **display graph.png**, where **graph.png** is just an example of any graphics file. Converting an

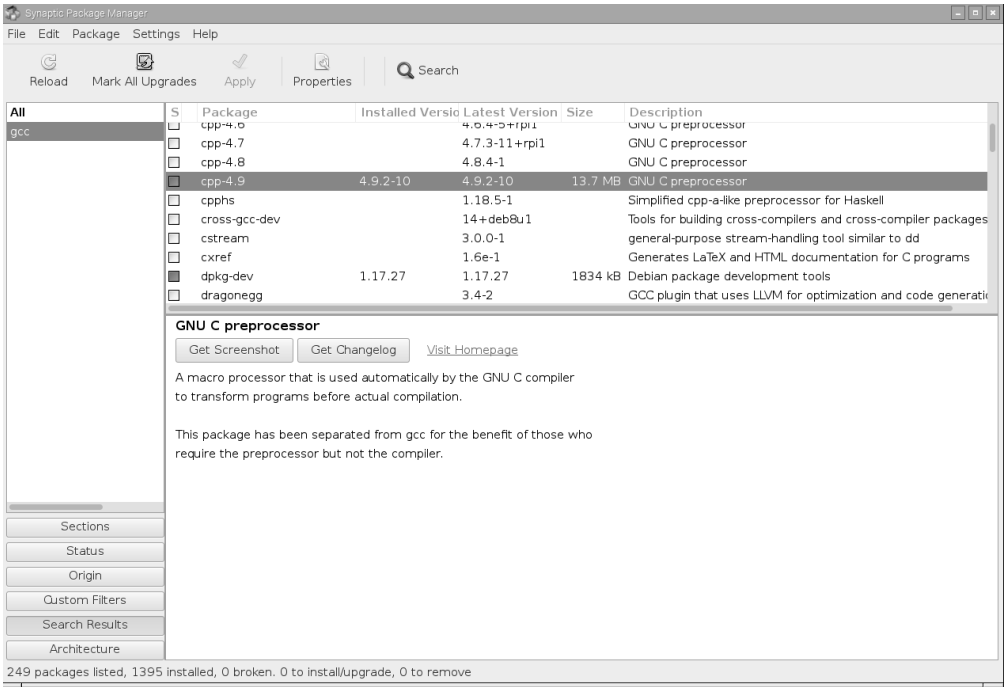


Figure 5.4 The Synaptic Package Manager.

existing file `graph.png` to JPEG format is done by typing `convert graph.png graph.jpg` on the command line. Use `man display` and `man convert` to find out more about what the programs can do.

Installing with `apt-get` is already rather convenient because dependencies to external libraries are automatically resolved, and any missing libraries are installed in order to guarantee a reliably working system. There is, however, an even more convenient way, namely by using the package manager `synaptic`. We install it by issuing

```
sudo apt-get install synaptic
```

on the command line. Once the installation is complete, it appears as *Synaptic Package Manager* in the *Preferences* section on the menu, behind the raspberry logo. Starting it from there first requests the login password before a window appears, shown in Figure 5.4. We use the *Search* button on the top to find programs. Just click on it, enter a keyword and peruse the list, install any interesting program, and try it out.

We use that feature immediately, and install programs to access computers, such as the NodeMCU that provides server capabilities via sockets. For this we need one or both of the `netcat` and `telnet` (client) programs, and install them via `synaptic` by searching for the program name; we pick the required package from the list by right-clicking it and choosing *Mark for Installation*. Once all programs are selected, we can press the *Apply* button near the top left of the `synaptic` user interface. Upon completion of the installation, we are ready to use the programs to connect to the NodeMCU running the socket-based server, which we discuss in detail later. Here we briefly illustrate the capabilities of the `netcat` program by opening an ad-hoc server in one terminal window. Running `netcat -l 11111` starts a server that listens (`-l`) on port 11111. Note that normal users can only use port numbers

above 1023. We connect to that server by running `netcat localhost 11111` from a second terminal window, where `localhost` is the default name for *this computer* and the port number on which the server listens. Now anything written in one terminal automatically appears on the other. In this example, we only connected two running versions of `netcat` on the same computer. One is started as a server, the other as a client, but the same functionality works between any computer with a working network connection. Find out more about `netcat` by consulting its manual page with `man netcat`. Should we wish to remove a previously installed program, we select it in synaptic by right-clicking the program in the list and selecting *Mark for Removal*. Pressing *Apply* promptly removes it.

So far, we have used the Raspi as a standalone computer with screen, keyboard, and mouse directly attached to the Raspi, but that ties up the peripheral hardware. So, now is the time to install software to allow logging onto the Raspi via the network, and receiving a virtual screen of the Raspi desktop on our regular desktop computer. Then we have a Raspi-desktop in a window that behaves just like the one on a regular screen. To achieve this feature, we install the `tightvncserver` on the Raspi with the command

```
sudo apt-get install tightvncserver
```

or, alternatively, via `synaptic`. Once the installation has finished, we run `sudo vncserver` from the command line, answer the questions about passwords, and remember the password.

On the desktop computer from which we want to log onto the Raspi via the network, we install the `vncviewer` software using the package manager of our Linux distribution. During the installation, the `vncpasswd` program should be installed as well, and we need to run it from the command prompt on our desktop computer. It prompts us to enter the same password for the VNC connection that we selected on the Raspi when running the `vncserver` program for the first time. Once that is done, the basic infrastructure to log onto the Raspi using VNC is in place. To simplify the entire process, we create a file named `raspi.sh` on the desktop computer with the following contents

```
#!/bin/bash
RASPI=192.168.10.nn
ssh -X pi@${RASPI} vncserver -geometry 1280x960 &
sleep 10
vncviewer ${RASPI}:5901 -passwd /home/ziemann/.vnc/passwd &
```

and place that file in the `~/bin` directory. This little program remotely logs onto the Raspi with `ssh`, starts the `vncserver` program on the Raspi with the specified desktop geometry (we may use a smaller one such as `1024x768`), then waits a few seconds and starts the `vncviewer` on the desktop computer using the credentials in the `.vnc/passwd` file in the home directory. After a few further seconds the desktop of the Raspi appears in a window on the desktop computer, and we seamlessly move the mouse and keyboard focus to the Raspi desktop, and execute commands on the Raspi. This is a very convenient way to use the Raspi, because it only requires a network connection and no other peripheral devices.

At this point, we have a convenient setup to work on the Raspi, and are basically ready to communicate with all the microcontrollers with attached sensors. But using the hardware for both wired and wireless Ethernet on the Raspi is just too tempting to neglect, and in the next section we briefly describe how to use the wireless interface on the Raspi to span a private WLAN network to which we later connect the sensor nodes.

5.4 RASPI AS A ROUTER

In this setup we assume that the Raspi is connected to our normal network with its wired Ethernet RJ45 port, and we have access to it from our desktop computer to log on using `ssh` or `vncviewer`, as previously discussed. On the Raspi we need to install the following packages

```
sudo apt-get update
sudo apt-get install hostapd dnsmasq
```

where we first update the repositories, and then download and install the `hostapd` and `dnsmasq` package. The former program is responsible for spanning the private wireless network, and turns the Raspi into a WLAN access point. The latter provides IP numbers on the private network via the DHCP protocol, and translates web site names to IP numbers.

Next we need to re-configure our system. In the default configuration there is a background process, a daemon called `dhcpcd`, that tries to obtain IP numbers for every network interface on the Raspi, including the wireless interface called `wlan0`. On the other hand, in order to operate the Raspi as an access point, we need to configure the interface ourselves and therefore need to remove the interface `wlan0` from the control of the daemon by adding the line

```
denyinterfaces wlan0
```

at the end of the file `/etc/dhcpcd.conf` with any text editor, such as `nano` or `emacs`. Now we are ready to configure the interface `wlan0` manually by rewriting the relevant section in the file `/etc/network/interfaces` to look like this:

```
auto wlan0
iface wlan0 inet static
address 192.168.20.1
netmask 255.255.255.0
up /sbin/iptables -A POSTROUTING -t nat -o eth0 -j MASQUERADE
down /sbin/iptables -D POSTROUTING -t nat -o eth0 -j MASQUERADE
```

which defines the IP number of the `wlan0` interface to be 192.168.20.1, and configures the type of network (class C) in the following line with the `netmask` command. The `/sbin/iptables` command enables network address translation between the wireless network on `wlan0` and the wired network on interface `eth0`. In order to allow network packages to pass between the interfaces, which is disabled by default as a security measure, we need to enable *forwarding* by removing the hash `#` from the line

```
net.ipv4.ip_forward=1
```

in the file `/etc/sysctl.conf`. We omit this step if all sensor nodes should only communicate with the Raspi, but no other computer on the local network should be able to surf the outside Internet beyond the Raspi, which might not be needed and poses a potential security risk. This last step completes the basic network setup, and we turn to the configuration of the access point software.

The configuration file for the `hostapd` daemon is `/etc/hostapd/hostapd.conf` and contains the following lines:

```
# /etc/hostapd/hostapd.conf
interface=wlan0
```

```

driver=nl80211
ssid=messnetz
channel=5
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase=zxcvZXCv
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
rsn_pairwise=CCMP
hw_mode=g

```

The file must be made readable for the owner only by executing `sudo chmod 600 /etc/hostapd/hostapd.conf`. It contains the definition of the network name `messnetz` and the details of the encryption, such as the passphrase. The latter we should adapt to your own secret phrase, to prevent others from misusing your wireless network. We need to tell the system where to find the configuration file by entering the line

```
DAEMON_CONF=/etc/hostapd/hostapd.conf
```

near the top of the file `/etc/init.d/hostapd` and then instruct the system to start the `hostapd` daemon at boot time with the command

```
sudo service hostapd start
```

After rebooting the system, we should be able to see a new WLAN named `messnetz` from a computer with a wireless network interface, but before we can connect to it we need to start the `dnsmasq` daemon that provides DHCP services on the `192.168.20.xx` network and distributes IP numbers. The configuration file is called `/etc/dnsmasq.conf` and it should contain the following lines:

```

domain-needed
interface=wlan0
dhcp-range=192.168.20.100,192.168.20.200,12h
listen-address=192.168.20.1

```

We save the originally installed version under a new name because it contains explanations of all parameters and is useful as a reference. For our system, however, we only need those in the above example, and they are almost self-explanatory. Finally, we register the `dnsmasq` daemon to start at boot time with the command

```
sudo service dnsmasq start
```

and reboot for good measure, to ensure that all parts of the system are synchronized. Technically, this is not needed, but it might help, in case some parameter was changed inadvertently during setup.

At this point we have turned the Raspi into a WLAN access point that spans the `messnetz` WLAN. If the IP forwarding is enabled, we can even surf the web from any computer on `messnetz`. But we are mostly interested in communicating with sensor nodes that are connected to `messnetz` and will discuss how to use the Raspi as the spider in the center of a network of sensor nodes connected by serial or WLAN links.

5.5 COMMUNICATION WITH THE ARDUINO

After the basic network infrastructure is set up, we connect the Arduino to the Raspi. Most programs on the Raspi work in a similar way as on desktop computers, which is no surprise, because a modern Raspi is about as powerful as a desktop computer from about a decade ago. We start by communicating via the Arduino IDE first, but later we will also use the Python programming language and octave.

5.5.1 Arduino IDE

We could install the Arduino IDE on the Raspi with `sudo apt-get install arduino`, but the version in the repositories is rather old. It is better to install the most recent version that is available from the <https://www.arduino.cc> web site, even for the Raspi. To do so, we open a browser on the Raspi, direct it to the Arduino download web site, choose the *Linux ARM* version, and download directly to the Raspi's download directory `/home/pi/Downloads/`. At the time of writing the current version was 1.8.0, and we may need to adapt the version number to a later release.

It is customary to install large software packages under the `/usr/local` tree, and we execute the following commands:

```
cd /usr/local
sudo tar xvJf /home/pi/Downloads/arduino-1.8.0-linuxarm.tar.xz
cd /usr/local/bin
sudo ln -s /usr/local/arduino-1.8.0/arduino arduino
```

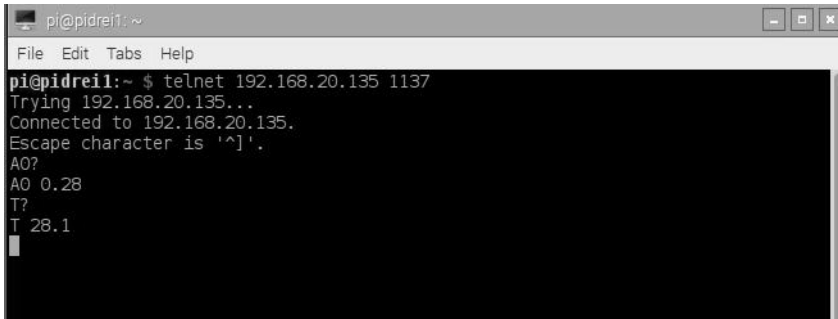
where the `tar` command unpacks the downloaded archive into the subdirectory named `/usr/local/arduino-1.8.0/`, and the `ln -s` creates a so-called soft link for the `arduino` executable in the `/usr/local/bin/` directory. In this way it is automatically found when typing `arduino` at the command line in a terminal window. Optionally, we can create a link in the *Programming menu* by starting the *Main Menu Editor* in the *Preference* section behind the *Menu* button with the image of a raspberry. In the left panel of the *Main Menu Editor* we select *Programming*, and then click on *New Item* on the right-side panel. In the opening window, we enter the name of the new program, *Arduino*, and browse to the `arduino` executable in `/usr/local/bin` directory. Once these steps are completed, there is an *Arduino* entry in the *Programming* section of the main menu. Selecting it opens the Arduino IDE on the Raspi, and we can start programming the Arduino directly from the Raspi. Before using it we add support for ESP8266-based microcontrollers by following the steps outlined at the end of Section 4.2 on page 58. On the Raspi, the procedure is just the same.

Instead of retyping all the programs for the Arduino on the Raspi, we copy the files from our desktop computer to the Raspi with `scp`, which is part of the secure-shell program suite if it is a Unix-based system such as Linux or Mac. On a Windows system we use *WinSCP*. On a Linux Desktop computer we execute

```
scp -r Arduino pi@192.168.10.nn:
```

which copies the Arduino subdirectory and everything in it to the Raspi, including all hardware descriptions from the `Arduino/hardware` directory. Now we need to restart the Arduino IDE, and all our previously prepared sketches are available under the *File*→*Sketchbook* menu in the Arduino IDE on the Raspi. After we connect the Arduino UNO or NodeMCU to an USB port on the Raspi, we are ready to program them from the Raspi.

Of course, we need to select the type of microcontroller and the port to which it is



```

pi@pidreil:~ $ telnet 192.168.20.135 1137
Trying 192.168.20.135...
Connected to 192.168.20.135.
Escape character is '^'.
AO?
AO 0.28
T?
T 28.1

```

Figure 5.5 Telnet session connected wirelessly to the NodeMCU.

connected the same way as on the desktop computer. In order to test that all is working properly, we download the **query-response** sketch to an Arduino UNO and open the *Tools*→*Serial Monitor* in the Arduino IDE. This should allow us to communicate with the UNO via the USB serial line. In case it does not work, ensure that the *Port:* in the *Tools* menu points to the correct serial port with the Arduino UNO and that the UNO is actually selected in the *Board:* menu.

5.5.2 From the command line

At this point we have established the Raspi as a development system for Arduino software, and ensure it communicates via the USB-based serial line with the Arduino IDE. But normally we want to access the Arduino from self-written programs rather than always using the Arduino IDE. The first option is to use the **screen** program or any other terminal program to connect to the UNO. After closing the Arduino IDE, we therefore start the following program on the command line on the Raspi:

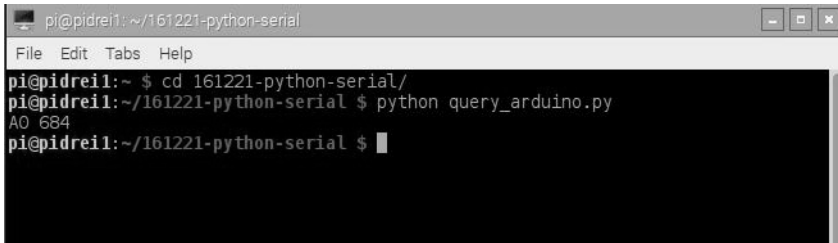
```
screen /dev/ttyACM0 9600
```

just as we did earlier when the UNO was connected to the Desktop computer. Bluetooth communication works exactly the same way as on the desktop computer. Just follow the details described in Section 4.6.2 on page 104.

In the next step, we use a NodeMCU and assume that the Router software from Section 5.4 is installed on the Raspi. We change the variables **ssid** and **password** in the NodeMCU sketches from Section 4.6.3 to those used on **messnetz**. Two lines near the top of the sketches now need to read

```
const char* ssid      = "messnetz";
const char* password = "zxcvZXCv";
```

where we need to make sure to enter the same password as the **wpa_passphrase** in the **hostapd.conf** file. We then connect the NodeMCU to a USB port on the Raspi and download the updated socket-based server to the NodeMCU. We can observe progress of the NodeMCU boot process with the *Serial Monitor* while it connects to the **messnetz** WLAN. After a while it should report the acquired IP number that the **hostapd** on the Raspi dished out. It lies in the 192.168.20.nn network, which corresponds to the IP range we specified in Section 5.4, in the configuration files of **dnsmasq**. In my case the IP number received was 192.168.20.135, but yours may be different. At this point we can connect to NodeMCU via WLAN by issuing the following command in a Raspi terminal window:



```

pi@pidreil1: ~/161221-python-serial
File Edit Tabs Help
pi@pidreil1:~ $ cd 161221-python-serial/
pi@pidreil1:~/161221-python-serial $ python query_arduino.py
A0 684
pi@pidreil1:~/161221-python-serial $

```

Figure 5.6 Using Python on the Raspi to communicate with the `query_response` sketch running on the Arduino UNO.

```
telnet 192.168.20.135 1137
```

where 1137 is the port number that is specified in the sketch that runs on the NodeMCU. The `telnet` program provides a prompt at which we can type commands to send to the NodeMCU. So, typing `A0?` on the telnet prompt should result in the reply `A0 nnn`, also displayed in the telnet window, as shown in Figure 5.5. We close the session by using the command `Ctrl-]` and then type `quit` at the telnet prompt.

Instead of using `telnet`, we can also use `netcat` and type the following command in a terminal window:

```
netcat -C 192.168.20.135 1137
```

where the `-C` option ensures that `CR-LF` characters are sent after each line, which is required by the NodeMCU to recognize the end of a command. Otherwise the communication works in the same way as with `telnet`.

The previous two paragraphs only verify that the communication works properly and that we have our system under control. Now we proceed and use the Python language to achieve the same feat.

5.5.3 Python

Python [24] is a modern programming language that is installed on any Raspi by default. Actually, the *Pi* in Raspberry Pi stands for *Python Interpreter*. Unless you have some experience with Python, I suggest you have a look at some of the tutorials at

<https://wiki.python.org/moin/BeginnersGuide/Programmers>

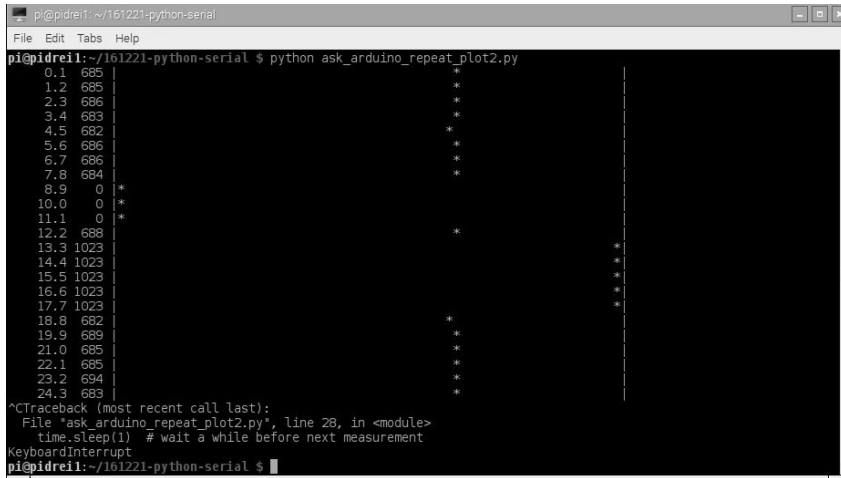
or search the web for detailed explanations of some of the simple, often even self-explanatory, commands we use here. The purpose to access an Arduino or NodeMCU from Python is to show how to write customized programs on the Raspi that work hand-in-hand with sketches that run on the Arduino.

We use the following Python script to query the UNO with the query-response sketch for a single value from analog pin 0.

```

# query_arduino.py
import serial, time
query="A0?\n"
ser=serial.Serial("/dev/ttyACM0",9600,timeout=1)
time.sleep(1)      # wait for serial to be ready

```

```

pi@pidreil1: ~/161221-python-serial
File Edit Tabs Help
pi@pidreil1:~/161221-python-serial $ python ask_arduino_repeat_plot2.py
0.1 685 |
1.2 685 | *
2.3 686 | *
3.4 683 | *
4.5 682 | *
5.6 686 | *
6.7 686 | *
7.8 684 | *
8.9 0 | *
10.0 0 | *
11.1 0 | *
12.2 688 | *
13.3 1023 | *
14.4 1023 | *
15.5 1023 | *
16.6 1023 | *
17.7 1023 | *
18.8 682 | *
19.9 689 | *
21.0 685 | *
22.1 685 | *
23.2 694 | *
24.3 683 | *
*Traceback (most recent call last):
  File "ask_arduino_repeat_plot2.py", line 28, in <module>
    time.sleep(1) # wait a while before next measurement
KeyboardInterrupt
pi@pidreil1:~/161221-python-serial $

```

Figure 5.7 Simple ASCII graphics from querying the UNO repeatedly.

```

ser.write(query)
time.sleep(0.1)
reply=ser.readline()
print(reply.strip())
ser.close()

```

Python is rather lightweight, and we have to import extra functionality such as serial communication by executing

```
sudo apt-get install python-serial
```

at the command prompt and then load the new functionality with the `import` statement. In the above script we import support for handling the serial line and basic time-handling. The latter we need to implement delays in the script. In the second line we define a variable `query` that contains the query string we send to the Arduino. Note that we explicitly add the carriage return `\n` character. Then we open the serial port `ser` on `/dev/ttyACM0` with a baud rate of 9600 and a timeout of 1 second, which prevents unsatisfied read attempts from blocking the program. We wait for one second to allow the operating system to finish opening the serial port, and then submit the query string with the `ser.write` command. Note that we use the method `write` on the serial device `ser`. Then we wait for 0.1 seconds and read characters up to the CR-LF character with the `ser.readline()` function into the variable `reply`. We display the reply with the `print` command after stripping off leading and trailing white-space characters, such as normal spaces or CR-LF characters. Finally, we close the serial line. We run the Python script that we give the name `query_arduino.py` by entering

```
python query_arduino.py
```

on the command line of the Raspi. We show this example in Figure 5.6.

The previous example script shows how to request a single measurement value. We easily expand the script to query the UNO repeatedly, and even provide simple ASCII graphics that show the measurement value as a function of time. This is accomplished by the Python

script below. Example output is shown in Figure 5.7, where we see that we call the program by executing

```
python ask_arduino_repeat_plot2.py
```

and the program then shows the elapsed time since it started, the measurement value, and a graphical representation of the value between the expected minimum and maximum values. This functionality resembles a simplified version of the Serial plotter built into the Arduino IDE. The Python script is the following:

```
# read arduino repeatedly from command line, V Ziemann, 161222
import serial, time, atexit
def cleanup():          # ensure serial line is closed after CTRL-c
    ser.close()
atexit.register(cleanup) # register the cleanup function
query="A0?\n"           # the query string
amin=0                  # minimum expected value
amax=1024               # maximum value
width=70                # number of character used for plot
ser=serial.Serial("/dev/ttyACM0",9600,timeout=1)
l11=len(query)-1        #
time.sleep(1)           # wait for serial to be ready
t0=time.time()          # starting time
while 1:                # repeat forever
    ser.write(query)     # send query
    time.sleep(0.1)      # wait a bit
    reply=ser.readline() # read response
    value=reply[l11:].strip() # make numeric
    k=(width-1)*int(value)/1024 # where to place *
    p='%8.1f %4s |' % (time.time()-t0,value)
    for j in xrange(0,width-1):
        if j==k:
            p+='*'
        else:
            p+=' '
    p+='|'
    print(p)
    time.sleep(1) # wait a while before next measurement
```

First we import support for serial communication, time, and the `atexit` functionality, which allows us to register a function that does some cleaning-up activities when the program terminates. Since we will employ an infinite loop to read the UNO, which we intend to stop asynchronously with Ctrl-C, this ensures that the serial line is properly closed. Note that Python uses indentation instead of brackets to define the scope of functionality, and the function body for the `cleanup()` function is just `ser.close()`, but indented by a few characters. After defining the query string and several variables, the serial line is opened, and we determine the expected length `l11` of the characters preceding the value of the reply, wait a short time, and record the present time in the variable `t0`. The `while 1:` statement initiates a loop that runs forever. Inside the while loop (note the indentation to indicate the scope), we write the query to the UNO, wait a short while, and read the response into the string `reply`. In the following line, we remove the first few characters such as `A0`, and remove

white-space characters with the `strip()` method. The variable `k` scales the measurement value such that it lies within the range specified in the `width` variable, and then we start building the string `p` by first placing the seconds elapsed since `t0` and the measurement value followed by a vertical bar denoting the start of the ASCII graphic. Then we loop over all subsequent places in the string `p` and place a `*` at the location corresponding to the scaled measurement value `k`, and a space otherwise. Finally, we add another vertical bar `|` and print the string `p` to standard output, before waiting one second for the next iteration to start. Running this Python script results in the rudimentary ASCII graphics shown in Figure 5.7.

The previous script works well with any serial line, including a serial port behind which a Bluetooth link is hidden. The only change we need to implement in the previous script is to replace the serial port `/dev/ttyACM0` with the Bluetooth port that is typically called `/dev/rfcomm0` or `/dev/rfcomm1`.

But what about connecting to the NodeMCU running the socket server that listens on port 1137? We only show the most basic network client that sends the query and displays the reply on standard output. More elaborate examples such as the simple ASCII graphics from above can be built quite easily once the basic network communication setup is under control. We illustrate these basics in the following example:

```
import socket, atexit, time
def cleanup():
    sock.send("quit\n")
    sock.close()
atexit.register(cleanup)
sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
sock.connect(("192.168.20.135",1137))
sock.send("T?\n")
time.sleep(0.1)
reply=sock.recv(1000);
print(reply.strip())
```

The structure of the program is very similar to the one for serial communication. First, the necessary functionality is imported, and then we define a `cleanup()` function that executes before the program closes. This is the safest way to ensure that the server closes the connection properly. Inside the `cleanup()` function, we instruct the python script to send the string `quit`, which is unknown to the NodeMCU and causes it to close the connection, before closing the socket on the client side. The function `cleanup()` is then registered in the call to the `atexit.register()` function. In the next line the socket `sock` is created with the specification of a normal TCP/IP socket, before connecting to the server running on the NodeMCU at IP number 192.168.20.135 and port 1137. Then we send the query string `T?` and wait a short while before we receive the reply, and then print it to standard output after stripping white-space characters.

In this section, we cover how to interface the sensor nodes with Python, and even present them as rudimentary ASCII graphics. This might suffice for quick-and-dirty fixes to rapidly observe how some value changes with time, but for more professional-looking results, we turn to octave using it to interface the sensor nodes and prepare much nicer plots for presentations and reports.

5.5.4 Octave

We already installed octave in Section 5.3, but in order to add packages, we need a few octave development tools that we install from the command line by entering

```
sudo apt-get update
sudo apt-get install liboctave-dev
```

which takes a few minutes to complete. For interfacing serial and network devices from within octave, we need to install the `instrument-control` toolbox. Unfortunately, it is not available in the normal repositories such that we can install it with `apt-get`. Instead, we need to use an alternative way to make it accessible. Once the installation of `liboctave-dev` finishes, we type `octave` on the command line and install the `instrument-control` toolbox from within octave by entering

```
pkg install -forge instrument-control
```

at the octave prompt. Specifying the `-forge` option causes octave to download the most recent version of the toolbox from the octave forge at <https://octave.sourceforge.io> and install it on the Raspi. Note that the installation process takes up to 30 minutes and does not report any progress. Just be patient! After completion, we add the following line:

```
pkg load instrument-control
```

to the octave startup file named `.octaverc` in our home directory, which in most circumstances on the Raspi is located in the `/home/pi/` directory. This will cause the toolbox to be loaded every time we start octave. While we edit that file, we may also add the line

```
graphics_toolkit('gnuplot')
```

to the `.octaverc` file to avoid a bug when using the octave `plot` command. But this may not be needed in all circumstances.

Once the installation of the `instrument-control` completes, we are ready to use it to communicate with our sensor nodes. First, we try to communicate with the UNO connected to the USB port of the Raspi, which is accessible as `/dev/ttyACM0` as before. In order to read one measurement from the UNO, we open the serial line from the octave prompt, submit the query, wait for the response, and display the result. The following program achieves this.

```
s=serial("/dev/ttyACM0",9600); % open serial line
sleep(1)                      % wait for this to complete
reply=queryResponse(s,"A0?\n") % send query and receive reply
fclose(s);                    % close serial line
```

It implements the simple query-response communication protocol we used earlier. Here we encapsulate the details of the query-response interaction in a separate function `queryResponse`, because there is no native support in octave to read from the serial device up to a termination character, and we implement that feature in the following function.

```
% send query and return reply up to termination character.
function out=queryResponse(dev,query,term_char)
    if (nargin==2) term_char=10; end % defaults to LF=0x0A
    srl_write(dev,query);            % send query to device
    i=1;
```

```

int_array=uint8(1);
while true                                % loop forever
    val=srl_read(dev,1);                  % and read one byte
    if (val==term_char) break; end        % until term_char appears
    int_array(i)=val;                     % stuff byte in output
    i=i+1;
end
out=char(int_array);                       % convert to characters

```

The three arguments are the serial device `dev`, the `query` string, and an optional termination character, which defaults the line-feed character (hex=0x0A, dec=10). In the function, we first send the query string in the `srl_write()` function call, and after initializing some variables, we repeatedly read one character at a time with the `srl_read()` function and append the character to the `int_array` unless it equals the termination character. Once the termination character is received, the `while true` loop exits, and the `int_array` is converted to characters in the last line. The converted array is returned to the calling program. Needless to say, we can also use any Bluetooth device with a serial line interface, only the device file is `/dev/rfcomm0` instead of the normal serial device files `/dev/ttyACM0` or `/dev/ttyUSB0`.

Reading via WLAN by connecting to the socket on the NodeMCU is done in much the same way as reading from the serial line. Instead of opening a serial line and using the `srl_write` and `srl_read` functions, we open a TCP connection and use the `tcp_write` and `tcp_read` functions. The basic code snippet that accomplishes this is shown here:

```

s=tcp("192.168.20.135",1137);           % open connection
sleep(0.1)                               % wait, not really needed
reply=queryResponseTcp(s,"T?\n")         % send query and get reply
tcp_write(s,"quit\n");                   % close remote socket
tcp_close(s);                             % close local socket

```

which follows the logic from the serial communication example above. The function `queryResponseTcp()` is essentially the same as the above example for the serial line, only the `srl_write` and `srl_read` functions are replaced by the `tcp_write` and `tcp_read` functions, and we therefore do not reproduce the code here. Note also that we need to explicitly close the remote socket on the NodeMCU by sending `quit`.

To illustrate the usefulness of the octave interface to serial line or networked devices, we write a simple temperature logger that connects to the NodeMCU and produces a nice plot of the temperature as a function of time. The following octave program achieves that.

```

% temperature logger, V. Ziemann, 161227
clear all
s=tcp("192.168.20.135",1137);
sleep(0.01)
running=0;
while running<10
    running=running+1;
    reply=queryResponseTcp(s,"T?\n");
    val(running)=str2double(reply(2:end));
    x(running)=now;
    plot(x,val,'*')
    ylim([22,28]);
end

```

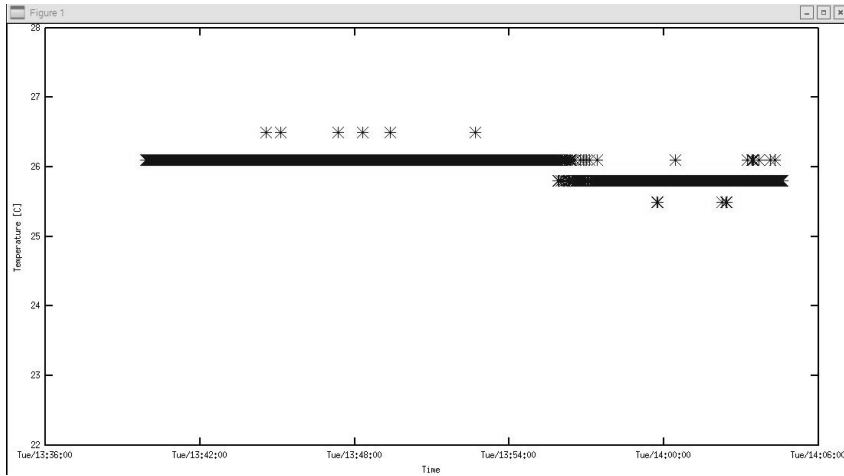


Figure 5.8 Plot from the temperature logger.

```

ylabel('Temperature [C]')
xlabel('Time')
datetick('x','ddd/HH:MM:SS')
sleep(1);
end
tcp_write(s,"quit\n");
tcp_close(s);

```

First, we clear all variables before we establish the TCP connection to the server on the NodeMCU, on port 1137 and IP number 192.168.20.135. Then we wait a short while and initialize a variable `running` to zero. We use this variable as an iterator in the loop and check whether the limit has been reached. In this simple example, we only iterate 10 times. In the loop, we increment the `running` variable and then send the query to the NodeMCU and receive the measurement in the string `reply`. Note that `reply` starts with a T, which we remove in the following line by converting `reply` from position 2 to the end to a `double` variable. The result we copy to the variable `val` at index position `running`. In this way we use the iterator not only to count the loop iterations, but also as an index of where to put the measurement values. In the variable `x` we copy the current time, which is returned by the built-in function `now()`. Then we plot the value versus the time using the `plot()` function, specify the vertical temperature range with the `ylim()` function, and specify the axis labels for vertical and horizontal axes. Finally we use the `datetick()` function to specify the type of tick marks we want. In this case we specify the day of the week and the time in hours, minutes, and seconds. The `datetick()` function uses the special format in which the `now()` function returns the time to extract the desired format of the tick mark. The full set of options is explained in the help text of the `datestr()` function that can be accessed by typing `help datestr` at the octave prompt. Before repeating the measurement, we wait a specified time, one second in the example. Finally, once the desired number of iterations is completed we ensure that the socket is closed on both the server and in the octave client. We show the resulting plot in Figure 5.8, where we run for 1000 iterations instead of just 10.

By now we have managed to record data from sensor nodes via serial and WLAN and display them online, even with reasonably attractive graphics, but for this privilege we need to have octave running while taking data. This may be feasible for a period of a few hours, but is hardly attractive for longer periods. In that case we need a separate repository to store the data, and only create plots when needed. We therefore need to separate the presentation from the data storage process, and this is the topic of the next section.

5.6 DATA STORAGE

In this section, we address several ways to store our measurement data, and the archetypical data storage repository is a *database*. We introduce a number of databases, but in most cases we use octave for the presentation of the data.

5.6.1 Flatfile

The simplest database is certainly a file containing a time stamp and one or more measurement values, possibly even in human-readable form. This is called a flatfile database, and we create one by using the following Python script:

```
# logger with time, V Ziemann, 161228
import serial, time, sys, atexit
def cleanup():
    ser.close()
atexit.register(cleanup)
query="A0?\n"
ser=serial.Serial("/dev/ttyACM0",9600,timeout=1)
time.sleep(1)      # wait for serial to be ready
while 1:
    ser.write(query)
    time.sleep(0.1)
    reply=ser.readline()
    print int(time.time()), reply[3:].strip()
    sys.stdout.flush()
    time.sleep(1)
```

and store it in a file called `ask_repeat.py`. We recognize the same organization as before, when we discussed Python scripts with importing functionality, registering the `cleanup()` function, opening the serial line, and repeatedly sending the query string and receiving the reply. The only difference is that we print the *Unix time*, which is the number of seconds since January 1, 1970, also called the *epoch*, before the measurement value. In the output, we therefore see the following text

```
:
1482954187 680
1482954188 678
1482954189 681
:
```

scrolling by. If we redirect this output into a file using the following line of code:

```
python ask_repeat.py > db.dat
```

that we run in a terminal window on the Raspi, we create a flatfile database `db.dat` with the time stamps and the measurement values. Later we can retrieve and convert the data to a nice plot.

Running the data acquisition program in a dedicated terminal window may be useful for short measurement sessions, but is hardly useful if we want to log, for example, the temperature in a building for an extended period of time, say a few months. In that case it is much preferable to have a dedicated process that wakes up once every few minutes, reads the temperature, and stores the value together with a timestamp in a file before going back to sleep for a few minutes. As it turns out, Unix systems normally have a system that takes care of these repeated tasks. It is called `cron` and it is a background process that wakes up once a minute, checks whether there is a task to do, does it, and sleeps for a minute before checking again.

Let us start by creating a Python script that we want to execute at regular intervals. In this particular case we place it in the `/home/pi/bin` directory and call the file `single_request.py`. It contains the following lines:

```
# single_request.py, V Ziemann, 161229
import serial, time
ser=serial.Serial("/dev/ttyACM0",9600,timeout=1)
time.sleep(1)      # wait for serial to be ready
ser.write("A0?\n")
time.sleep(0.1)
reply=ser.readline()
print int(time.time()), reply[3:].strip()
ser.close()
```

We see that it is rather similar to the previous scripts to read from the Arduino. It just acquires one measurement and prints the value with the timestamp to standard output. In order to make this program accessible for the `cron` software, it is convenient to encapsulate it in a shell script file `/home/pi/bin/readA0.sh`. It contains the following lines:

```
#!/bin/bash
/usr/bin/python /home/pi/bin/single_request.py >> /home/pi/A0.dat
```

We make the file executable with the `chmod` program by executing `chmod +x readA0.sh` in the `/home/pi/bin` directory. The first line instructs the operating system to interpret the following lines using the `bash` shell. The next line starts the `single_request.py` script using the `/usr/bin/python` program, and redirects the output to the file `/home/pi/A0.dat`. Here `>>` implies that new data is appended to an existing file. Note that all file names must be given, including their absolute path. We test the `readA0.sh` script to ensure that it creates the `/home/pi/A0.dat` file or appends a reasonable measurement value with timestamp to the file. Once we are satisfied, we register `readA0.sh` with the `cron` software and edit the configuration file for the `cron` program with the `crontab` program. We execute it from the command prompt by typing

```
crontab -e
```

The first time `crontab -e` is called it asks for an editor. We pick our favorite or follow the suggestion. Once the editor opens with the configuration file, we append the following line at the end of the file:

```
* * * * * /home/pi/bin/readA0.sh
```


and save the file. This automatically registers the `readA0.sh` script to execute once every minute whenever the Raspi is running. We check that the file is registered with the `crontab -l` command. It lists the contents of the `crontab` file for user `pi` provided we are logged on as user `pi`. The meaning of the six columns in the `crontab` file are *minute*, *hour*, *day of month*, *month*, *day of week*, and *program to execute*. Placing asterisks in the first five columns instructs `cron` to execute the program every time it wakes up. The output of the `crontab -l` command has some basic explanations, and more is available by executing `man 5 crontab` at the command prompt. So, now we have a background process that records a measurement once a minute and fills the file `/home/pi/A0.dat` until we remove the entry in the `crontab` file.

Our next task is to produce plots, preferably with the timestamps taken into account adequately. For this task we use `octave`, because it has powerful capabilities to handle timestamps. We extract the data stored in the flatfile database and display it with the following script.

```
# flatfile viewer, V. Ziemann, 161229
d=importdata('/home/pi/A0.dat');
TZ=1; t=719529+(d(:,1)+TZ*3600)/86400.0;
plot(t,d(:,2))
datetick('x','HH:MM:SS')
```

Most of the work is done by the `importdata()` function. It figures out the format of the `A0.dat` file and loads its contents into the variable `d`, which is a matrix with two columns, one with the time information and the other with the measurement values. In the next line we convert the time into the standard format that the `datetick()` function expects. It will properly format the displayed time on the horizontal axis. Note that we use the variable `TZ` to denote the time zone. Since I live in Sweden, the local clock is one hour ahead of UTC standard time, to which the epoch refers. The constant 719529 is the number of days from January 1, 0000 until January 1, 1970, and 86400 is the number of seconds per day. The plot looks similar to the one shown in Figure 5.8.

After storing data in a flatfile database, retrieving it, and displaying the data, we now progress to using a more mature database, MySQL.

5.6.2 MySQL

Among several databases available for the Raspi, we choose *MySQL* [25] because it is widely used and can be accessed reasonably easily from most programming languages. These include Python and, after some archaeology on the Internet, octave as well. We install the MySQL database from the standard repositories with the following command:

```
sudo apt-get install mysql-server
```

The installation of `mysql-server` triggers a request to enter a password for the mysql administrator, which is called mysql-user `root`. So make up your mind and invent a good one. Note that this user `root` is the database administrator and not the superuser for the computer. Just the name `root` happens to be the same.

Once MySQL is installed, we create a database named `dataA0` that will contain roughly the same information as the flatfile in the previous section, namely a timestamp and measurement values. Creation of a new database must be done by mysql-user `root`, thus we first need to log onto the MySQL administration account by executing the following command:

```
mysql -u root -p
```

and entering the password of the mysql-administrator `root` during installation, whence we are greeted by the MySQL prompt `mysql>`. At the prompt, we create the new database with the following command:

```
create database readA0;
```

where we specifically need to point out the necessity to complete every MySQL command by a semicolon. We verify that the database `readA0` exists with the `show databases;` command at the MySQL prompt. Since later we will not always want to work with the database `readA0` as mysql-administrator, we create a database user. To do so, we declare that we want to work with `readA0` by writing

```
use readA0;
```

at the mysql prompt. Now we create the user `me` and then grant privileges to work with the database, with the following sequence of commands:

```
create user 'me'@'localhost' identified by 'pwpw';
grant all privileges on readA0.* to 'me'@'localhost';
```

where `pwpw` is the (too simple, invent a better one) password. Luckily the MySQL syntax is fairly self-explanatory. Note that the granting of privileges can be tailored in a rather detailed fashion into reading, writing, and other privileges, but for our simple example we stick to the simple version. Once the general administration of creating a database and assigning a mysql user is done, we exit from the mysql prompt by typing `quit;`.

But now we log onto MySQL as mysql-user `me` by typing

```
mysql -u me -p
```

at the command prompt and entering the password. Alternatively, the command `mysql -u me -ppwpw` with the password `pwpw` immediately appended after the `-p` will directly log onto the mysql-prompt, but now as the user `me` who only has privileges to work with the `readA0` database. We verify this by executing the `show databases;` command, which shows `readA0` and some other administrative database that we ignore for the time being. In order to create a data structure, called `table` in our database, we select it with `use readA0;` and then enter

```
create table dat (ts timestamp, A0 integer, A1 integer);
```

which creates a table `dat` that stores a time stamp and two values in every row. Other options for things to store are `float` and `BLOB`, the latter being a *binary large object*. The MySQL manual, available at <http://dev.mysql.com/doc/>, holds a wealth of detailed information.

The administrative task to create the table is done at this point, and we could `quit;`, but it is instructive to insert values into the newly created table by executing the following command at the mysql-prompt

```
insert into dat (A0,A1) values (17,4);
```

and verify the contents of the database with the command

```
select * from dat;
```

which prints the contents of the table. We observe that the timestamp variable was automatically filled with the time the values are inserted. The syntax to insert values is, again, rather self-explanatory. We **insert** into a table called **dat** the variables (A0,A1) with values 17 and 4. Reading all values from the table is achieved by the **select** command. The way it is specified with the asterisk displays the entire table **dat**. If we only want to print the timestamp **ts** and value **A1**, we can issue **select ts,A1 from dat;**. These commands to create, insert, and select data from the table are standardized and are called *Structured Query Language* or SQL. Please inspect tutorials and further information about SQL on the Internet. After this exercise we exit the **mysql** program by executing the **quit;** command.

Now we know the basic SQL commands to insert and retrieve data from tables. But rather than typing them from the **mysql** prompt, our next task is to use Python to execute the SQL commands. We add MySQL functionality to Python by installing the **python-mysqldb** package with the command

```
sudo apt-get install python-mysqldb
```

at the command prompt, and are ready to access MySQL databases from within Python. As a first attempt, we read the table **dat** from the database **readA0** we had created earlier. The following Python script achieves just that.

```
# access MySQL database, V. Ziemann, 170101
import MySQLdb
db=MySQLdb.connect("localhost","me","pwpw","readA0")
cur=db.cursor()
cur.execute("select * from dat;")
reply=cur.fetchall()
print(reply)
#for r in reply:                                # loop over entries
#    print r                                     # print each entry
#    print str(r[0]), str(r[1]), str(r[2]) # format nicely
db.close()
```

First the library with MySQL support is imported, and then we connect to the database **readA0** on computer **localhost**, as user **me** with password **pwpw**. The **MySQLdb.connect()** function returns a handle **db** to the database, and executing it corresponds to logging onto the database and executing the **use readA0;** command at the **mysql** prompt. The next line creates a cursor **cur**, which is equivalent to the **mysql** prompt and allows us to enter database insertion or retrieval commands. In the next line we execute the **select** command to display the entire database. We retrieve the output from the command in the variable **reply** with the **fetchall()** call and print the reply before closing the database. Note the structure of first executing a MySQL command and then retrieving the reply with the **fetchall()** function. The reply, however, is a list of entries in an unfamiliar form. It is possible to display each measurement entry on a separate line by using the lines commented out by a hash (#). The code loops over each entry in the list, and in the first commented example, the print command displays one entry at a time. In the second example, the print command uses the **str()** function to convert each entry to a string which makes it human readable. Finally, we close the database with the call to the **db.close()** function.

Our next task is to insert new entries with measurements into the database, and this is accomplished by the following Python script.

```
# serial2mysql.py, V Ziemann, 161229
```

```

import serial, time, MySQLdb
ser=serial.Serial("/dev/ttyACM0",9600,timeout=1)
time.sleep(1)      # wait for serial to be ready
ser.write("A0?\n")
reply=ser.readline()
val0=int(reply[3:].strip())
ser.write("A1?\n")
reply=ser.readline()
val1=int(reply[3:].strip())
ser.close()
db=MySQLdb.connect("localhost","me","pwpw","readA0")
cur=db.cursor()
sql="insert into dat (A0,A1) values (%d,%d);" % (val0,val1)
cur.execute(sql)
db.commit()
db.close()

```

The first part of the script is very similar to earlier scripts. It just queries the Arduino UNO on the serial line and converts the measurement values from analog pin A0 and A1 to integers, which are stored in the variables `val0` and `val1`. In the second part we connect to the database `readA0` and obtain a cursor. We then build the `sql` string with the MySQL insert command and execute it in the next command, before committing the changes to the database and closing it. Note that we could perform several transactions in a row that might leave the database in a bad intermediate state. To prevent this from happening, all insert commands are buffered and all changes are committed to the database simultaneously with the `commit()` function call. In order to automatically record data, we add the line

```
* * * * * /usr/bin/python /home/pi/python/serial2mysql.py
```

to the crontab file using the command `crontab -e` that we already discussed in the previous section on flatfile databases.

Now that we have a process that continuously fills the database, we want to use the stored data in `octave` for postprocessing and to prepare plots for presentations. For this we need a function that allows us to access the database from within `octave`, which turns out to be difficult, because MySQL is not supported by either the Raspberry Pi or `octave` repositories. There is, however, code to bind MySQL to MATLAB, available from <http://www.courant.nyu.edu/~almgren/mysql>, and it turns out that it also works well with `octave`. We download the `mysql.cpp` and `mysql.m` files from the above web site and prepare the `octave` version by running

```

mkoctfile --mex -I/usr/include/mysql -L/usr/lib/mysql\
-lmysqlclient mysql.cpp

```

in a command window on the Raspi, which produces a file named `mysql.mex`. We then copy both `mysql.mex` and `mysql.m` to the directory where it can be found by `octave` at run time; for example, the directory with the script that we are about to write. We start `octave` in that directory and type `help mysql.m` at the `octave` prompt for an explanation of the syntax of how to use the `mysql()` command in `octave`.

In the same directory, we prepare an `octave` script `dbread.m` to read the values from the database and display them.

```
% dbread.m, V. Ziemann, 170101
```

```

mysql('open','localhost','me','pwpw');
mysql('use readA0');
sql='select * from dat;';
[t,a0,a1]=mysql(sql);
mysql('close');
tt=datetime(t,'yyyy-mm-dd HH:MM:SS');
plot(tt,a0,tt,a1)
legend('A0','A1')
datetick('x','ddd/HH:MM')

```

The `dbread.m` script uses the `mysql` mex file from the previous paragraph and opens the database before using the `readA0` database. Note that there must not be a semicolon at the end of this command, for unknown reasons. In the next line the SQL command is written into the string `sql` before we execute it with the `mysql()` function call. It directly returns all variables from the database into variables `t`, `a0`, and `a1`. Note that we do not need the equivalent of Python's `fetchall()` call; it is already built into the `mysql()` function. After retrieving the requested data, we close the database connection and convert the time stamp data that arrives in string format from the database to the `datetime` format, with days since January 1, 0000. We supply the format string to aid the conversion. Finally, we plot the data, provide a legend, and specify the tick marks on the horizontal axis to include the day of the week, hours, and minutes.

In the previous examples we always select all available data from the database. In many circumstances we prefer to restrict the displayed data to a smaller range. Instead of filtering in octave, we instruct the database to only return data from the restricted time window. To achieve this we use the following SQL query string:

```

sql='select * from dat where ts > "2017-01-01 17:30"
    and ts < "2017-01-01 17:55";'

```

instead of the command `select * from dat;` used earlier. Note that the command needs to be written onto a single line. This query string explains the restricted time window in clear text. We require the timestamp `ts` to be larger than some date and less than some other date. The values returned from the command `[t,a0,a1]=mysql(sql);` thus only contain data from within the requested time window.

Hopefully this short introduction to MySQL and how to access it from Python and octave is useful to get you started in case a database is needed in a project. But now we will turn to a second database, one that only stores values over a finite time-horizon and also thins them out the further back in time the data originate. This database is the round-robin database called `rrdtool` that we discuss in the next section.

5.6.3 RRDtool

The `rrdtool` [26] program was initially conceived as a tool for allowing computer-network administrators to present network traffic over different time horizons (last hour, day, week, month, year) in a convenient and flexible way. It generates graphical representations of data that can be shown in a web browser by automatically generating consolidated data, such as average, minimum, or maximum over some period of time. `Rrdtool` is particularly useful to generate plots of the measured data on the fly, and we later use it to display our measurement data, such as temperatures, on a web page. The round-robin database is implemented as a circular buffer that is filled up to the end and then wraps around and starts to overwrite the oldest values at the beginning of the buffer. Using `rrdtool` comprises

three steps: creating the database, filling it with data, and extracting the data. But before delving into examples, we need to install the software with the following command:

```
sudo apt-get install rrdtool
```

after having updated the repositories with `sudo apt-get update` (just a friendly reminder not to forget the update step). Now we are ready to use the software.

In what follows, we assume that all files reside in the subdirectory `/home/pi/rrdtool`. The first step—creating the database—is achieved by the `rrdtool create` command that we enter at the command prompt of the Raspi. In this step we define the frequency of storing data, the type and valid range of data, and the way the stored data should be preprocessed; more on the last point later. The simplest example, namely to create a database `db1.rrd`, is shown in the following example:

```
rrdtool create db1.rrd --step 60 \  
DS:temp:GAUGE:180:-20:100 \  
RRA:AVERAGE:0.5:1:2880
```

where we can also omit the backslash and write the entire command on a single line. The first part of the command creates `db1.rrd` and the database stores values every 60 seconds. The second line defines the data source (DS:), a variable called `temp` that is of type `GAUGE`, which is rrdtool-speak for “measurement value.” We require a valid data point to be uploaded to the database at least every 180 seconds before a value is marked as invalid. The expected range of values for the data points is between -20 and 100, which is reasonable for a temperature reading. The line starting with `RRA` defines the round-robin archive that contains averaged values; the number 0.5 is used internally and should not be changed; the number of data points to be averaged, here 1; and the total number of (averaged) data points that the database should hold. In the example we use 2880, which is the number of minutes in two days. Since we chose to average only one data point, we store all values, rather than actually averaging. The above `rrdtool` command creates the file `db1.rrd` in the directory where the command is executed. Note that we can create several data sources; for example, for temperature, humidity, and barometric pressure, by adding DS: statements. Moreover, note that the `RRA:` line creates one table in the database `db1.rrd`. We can define several more tables with additional `RRA:` statements. For example, adding `RRA:AVERAGE:0.5:30:336` will create a table with data averaged over 30 readings, thus one point every 30 minutes, and will store 336 values, which corresponds to one week, because there are 336 half-hour periods in a week. Other options, instead of `AVERAGE`, are `MIN` and `MAX`, which will store the minimum or maximum in the specified time period, respectively. Please consult `man rrdtool` for more options. Now that we have a database, we can start to fill it with data.

We fill the database with the `rrdtool update` command. The data we store are temperature measurements from an LM35 temperature sensor attached to an Arduino UNO, similar to the way we used it before. The following Python script sends T? to the UNO and receives the temperature data as a string that is similar to T 22.5.

```
# read temperature, V Ziemann, 170102  
import serial, time  
ser=serial.Serial("/dev/ttyACM0",9600,timeout=1)  
time.sleep(1)      # wait for serial to be ready  
ser.write("T?\n")  
reply=ser.readline()  
print reply[2:].strip() # just print temperature
```

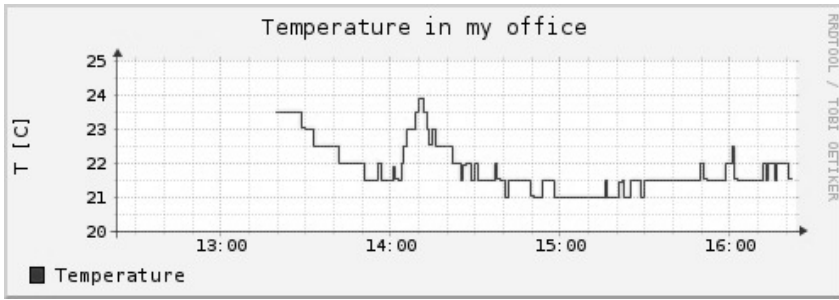


Figure 5.9 Graphics from rrdtool.

```
ser.close()
```

The difference from earlier versions is that we print out the numerical value only. We call this Python program from a shell script file with the name `filldb1.sh`. It contains the following lines:

```
#!/bin/bash
DB=/home/pi/rrdtool/db1.rrd
TEMP=$(/usr/bin/python /home/pi/rrdtool/readtemp.py)
/usr/bin/rrdtool update $DB N:$TEMP
```

and we make it executable by executing `chmod +x filldb1.sh`. In the script, we first define a variable `DB` that contains the absolute path to the database file. It resides in `/home/pi/rrdtool/` as already mentioned above. On the next line we fill the variable `TEMP` with the output of the command between `$(` and `)`, but this is the temperature value that the command `/usr/bin/python /home/pi/rrdtool/readtemp.py` returns. Note that the absolute path is used for both the Python interpreter and the `readtemp.py` script. In the last line we execute the `rrdtool update` command with the database name stored in the variable `DB`, and fill it with the current time stamp, as indicated by `N:` and the temperature value stored in the variable `TEMP`. Executing `filldb1.py` from the command line sends a data point with the current time to the database, but that is rather inconvenient. A better solution is to run `filldb1.py` once a minute, automatically. We therefore update the crontab file by executing `crontab -e` and add the line

```
* * * * * /home/pi/rrdtool/filldb1.sh
```

to the end of the file. This will send a new data point to the database once every minute.

Finally, in the third step, we retrieve the data and generate a graph with the `rrdtool graph` command. Here is an example:

```
rrdtool graph db1.png -s -4h \
-t "Temperature in my office" -v "T [C]" \
DEF:t0=db1.rrd:temp:AVERAGE \
LINE1:t0#FF0000:"Temperature";
```

Note that we also can write the entire command on a single line. It creates a graphics file `db1.png` with start of the time axis 4 hours back, `-s -4h` in time, a title specified after the `-t` option, and the vertical axis label after the `-v` option. In the next line the handle `t0` is defined to refer to the data coming from the `db1.rrd` database and as the `AVERAGE` of the

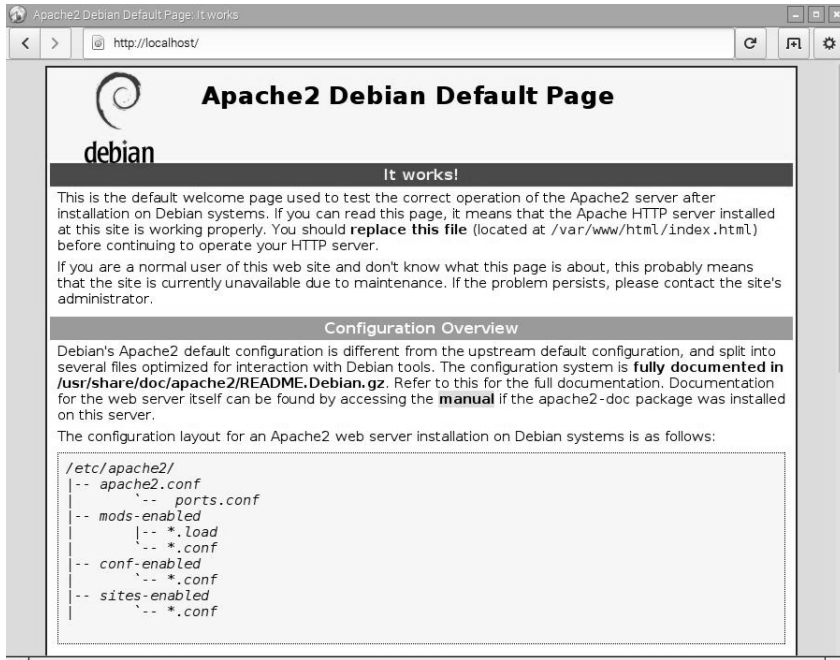


Figure 5.10 Web page from the Raspi after installation of the Apache2 web server.

variable `temp`. Compare this to the `rrdtool create` command, where `temp` was defined in the `DS:` part of the command and `AVERAGE` in the `RRA:` part. Finally, we define a displayed line to use the handle `t0`, specify the color by hexadecimal RGB values (here red: `#FF0000`), and give it the label `Temperature`. In Figure 5.9 we show the resulting graphics produced by the above `rrdtool graph` command. Note that only the last four hours are plotted, but the first part is missing, because the cron job was not running at that time. The data are therefore noted *invalid* in the database, and consequently not printed. The scale is adjusted automatically in the previous examples, but can also be specified explicitly. The command `man rrdtool` and the pointers therein provide a wealth of information on how to fine tune the output.

After being able to measure, store, and retrieve measurement values, we want to present them on a web page for online observation and continuous checking.

5.7 ONLINE PRESENTATION

In this section we describe how to present the measured data on a web page that is published by a web server. It turns out that the Raspi is quite capable of running a web server on the side while doing all the other things such as acquiring and storing data. The basic software needed is a web server, and we chose `apache2` [27], which is installed by executing

```
sudo apt-get install apache2
```

on the command line of the Raspi. After the installation is complete, we use a browser on any computer connected to the network the Raspi is connected to and enter the IP number of the Raspi in the address line. This also works on the browser installed on the Raspi;

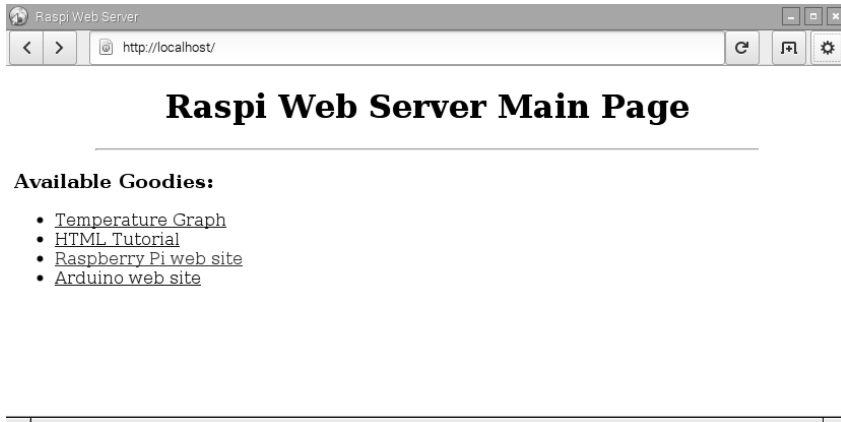


Figure 5.11 The first self-made web page.

enter `localhost` as the web address, and you should be greeted by the web page shown in Figure 5.10. This page instructs us to replace the file named `index.html` in the directory `/var/www/html/` with our own copy. Note that the `apache2` web server is configured to present a file named `index.html` by default, if only the address of the folder is given.

So, we need to prepare a simple web page, which is a specially formatted text file in a format called HTML—an acronym for *Hyper-Text Markup Language*. There are plenty of books on the subject, and tutorials can be found on the web in abundance. Here we only use the most basic features. We navigate to the directory `/var/www/html/` and save `index.html` as a backup with `mv index.html index.html.bak`. Then we start our favorite editor as superuser by prepending `sudo`, enter the following text, and save the contents as `index.html`.

```
<!DOCTYPE HTML>
<HTML>
  <HEAD>
    <TITLE>Raspi Web Server</TITLE>
  </HEAD>
  <BODY>
    <H1 ALIGN=CENTER>Raspi Web Server Main Page</H1>
    <HR SIZE=2 WIDTH=80%>
    <H3>Available Goodies:</H3>
    <UL>
      <LI> <A HREF="/temp/">Temperature Graph</A> </LI>
      <LI> <A HREF="http://www.w3schools.com/html">
        HTML Tutorial</A> </LI>
      <LI> <A HREF="http://www.raspberrypi.org">
        Raspberry Pi web site</A> </LI>
      <LI> <A HREF="http://www.arduino.cc">Arduino web site</A> </LI>
    </UL>
  </BODY>
</HTML>
```

If no typos crept in, by entering the address `http://localhost` on a browser running locally on the Raspi we should see the page shown in Figure 5.11. This is how the web browser renders the contents of the file we just entered and now briefly discuss. At the top of the file we have the declaration of the document type; it is an HTML file. Then we have the opening tag `<HTML>` and a matching closing tag `</HTML>` at the end of the file. Anything between them describes the contents of the web page. Note that HTML tags (almost) always come in pairs: the tag name in angle brackets and a matching closing tag with the same name, but prepended with a slash. The next tags we encounter in the file are `HEAD` and `TITLE`, which describe things that do not show up on the page but appear in the title bar of the web browser. Finally, we reach the `BODY` tags, where we find the description of the web page proper. The `<H1>` tag declares a large headline, and the `ALIGN` directive specifies it to be centered on the web page. There are different levels of header tags, from `H1` to `H6`. The next line defines a horizontal rule with the `<HR>` tag that covers 80 % of the width of the page. Then we add a smaller header with `<H3>` and an unnumbered list between the `` tags, with each list item described by `` tags. The `<A>` tag is called an anchor. It points to other web sites specified in the `HREF` directive. The first list item points to a local directory `temp/` under the directory where the file `index.html` resides. Since this does not yet exist, we need to create it.

In the directory `/var/www/html/`, we create the `temp/` subdirectory and copy a file, also named `index.html`, with the following contents into the newly created subdirectory `/var/www/html/temp/`:

```
<!DOCTYPE HTML>
<HTML>
  <HEAD>
    <TITLE>Raspi Web server</TITLE>
  </HEAD>
  <BODY>
    <H1 ALIGN=CENTER>Temperature</H1>
    <IMG SRC="db1.png" ALT="Temperature in my office">
  </BODY>
</HTML>
```

The file contents follows the same general layout as before, with the `DOCTYPE` declared first, followed by `<HTML>` tags. Next come the `<HEAD>` tags and then the `<BODY>` tags bracketing the displayed contents of the page. Here we also find a header and a new tag `` to direct the web browser to display the image specified in the `SRC` directive. To make this work, we copy the file `db1.png`, the one we created with `rrdtool graph` in the previous section, to the directory `/var/www/html/temp/`. We always need to use `sudo` to edit or copy files to the system areas to which the files under `/var/www/` belong. This we can avoid by enabling private web pages.

The private web pages commonly reside in a subdirectory called `public.html` under the user's home directory. As user `pi` we therefore create it by typing `mkdir/home/pi/public_html`. In order to use it we have to enable the `userdir` module by executing `sudo a2enmod userdir` at the command prompt, and restarting `apache2` with the command `sudo service apache2 restart`, such that the newly enabled module is loaded. Then we copy the files `index.html` and `db1.png` from `/var/www/html/temp/` to `/home/pi/public_html` and are ready to access the same web page as before, but now under the new address `http://localhost/~pi`. Any file we copy to the subdirectory `/home/pi/public_html` is then accessible from a browser at the address `http://localhost/~pi/` with the filename

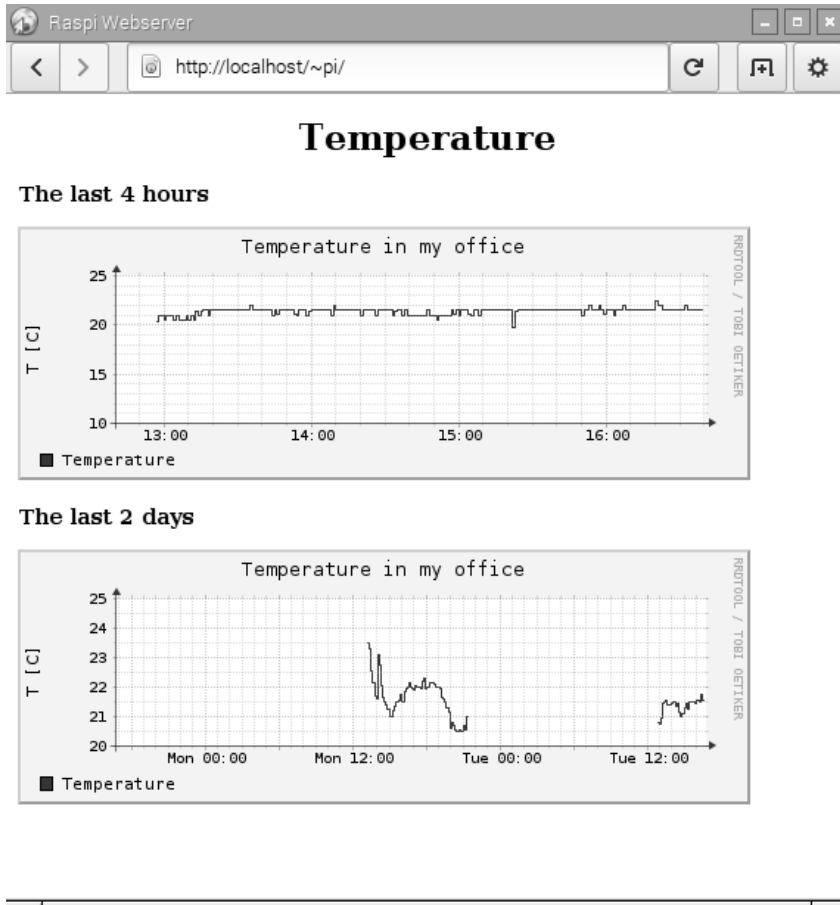


Figure 5.12 User pi's home page that displays the continuously (but slowly) updated temperature.

appended. For example, `http://localhost/~pi/db1.png` only displays the graphics file in the browser window, but nothing else.

The contents on the web page in the previous example is static. We copy files to the `public_html` directory and then they are presented as is. If we want to update, for example, the temperature graph, we have to run `rrdtool graph` again and copy the new copy of the `db1.png` to `public_html`. But this is a task that we easily automatize with the help of a cron job. We prepare a file with the following contents:

```
#!/bin/bash
DB=/home/pi/rrdtool/db1.rrd
/usr/bin/rrdtool graph /home/pi/public_html/db1.png -s -4h \
    -t "Temperature in my office" -v "T [C]" \
    DEF:t0=$DB:temp:AVERAGE \
    LINE1:t0#FF0000:"Temperature";
/usr/bin/rrdtool graph /home/pi/public_html/db2.png -s -2d \
    -t "Temperature in my office" -v "T [C]" \
```

```
DEF:t0=$DB:temp:AVERAGE \
LINE1:t0#FF0000:"Temperature";
```

and name it `makegraph.sh`. Then we place the file in the `/home/pi/rrdtool/` subdirectory and make it executable with `chmod +x makegraph.sh`. In the file, we first define the database to use and then have two almost identical copies of the `rrdtool graph` command from the previous section. But in the first case, we create `db1.png` for data from the last 4 hours, and in the second case we create `db2.png` for data from the last two days (`-s -2d`). Note that the graphics files `db1.png` and `db2.png` are specified to reside in the `public_html` directory, where they are accessible to the web server. Finally, we make the `makegraph.sh` script execute every 10 minutes by adding the following line to our crontab file with the command `crontab -e`

```
*/10 * * * * /home/pi/rrdtool/makegraph.sh > /dev/null
```

where `*/10` in the first column means *every 10 minutes* and the `> /dev/null` at the end means to suppress any output from the `makegraph.sh` command. So now we can update the to-be-displayed content, but we still need to coax the web browser to actually re-read that content at some interval. For this purpose the `<META http-equiv=..>` tag exists, and we include it between the `HEAD` tags, as shown in the following updated version of the `public_html/index.html` file

```
<!DOCTYPE HTML>
<HTML>
  <HEAD>
    <TITLE>Raspi Web server</TITLE>
    <META http-equiv="refresh" content="300">
  </HEAD>
  <BODY>
    <H1 ALIGN=CENTER>Temperature</H1>
    <H3>The last 4 hours</H3>
    <IMG SRC="db1.png" ALT="Temperature over 4 hours">
    <H3>The last 2 days</H3>
    <IMG SRC="db2.png" ALT="Temperature over 2 days">
  </BODY>
</HTML>
```

which instructs the browser to reload the page every 300 seconds, as indicated by the `META` tag. At the same time, we added the second image that displays the temperature over the last two days. A screen shot of the web page is shown in Figure 5.12. We can change the update frequency in both the crontab file and in the `META` tag if we are impatient. They do not need to match.

At this point we can obtain, store, retrieve, and display measurements on an actively updated web page. The update mechanism that we implemented using cron jobs is very rudimentary, and there are better solutions; for example, `cgi-bin` or `php` server-side programs. They execute code on-demand at the press of a button on the web page, and update the displayed information on the fly, but that is beyond the scope of our presentation.

So far we have used the Raspi as the hub in our sensor network to query the sensor nodes, and store and present the data, but we can even turn the Raspi into a node of a larger control system such as EPICS. This is the topic of the next chapter.

QUESTIONS AND PROJECT IDEAS

1. Where do you find help about programs installed on the Raspi?
2. How do you copy files on the Raspi?
3. How do you rename files on the Raspi?
4. How do you copy files between the Raspi and your desktop computer?
5. Find out what the program `ping` does.
6. Find out what the program `touch` does.
7. Describe the installation of the program `nmap` on the Raspi. Investigate what it does and how it may help you in debugging your network.
8. What is the `netmask`?
9. Start *Mathematica* from the menu and play around with it. Find out what it can do.
10. Start the `synaptic` software installation program from Section 5.3, enter “games” as the search keyword and explore.
11. Start `python`, execute `help()`, and follow the instructions.
12. Write a program that displays “Hello World” in Python.
13. Create a file with the name `look_at_me` in the home directory of user `pi`, and instruct the `cron` daemon to `touch` it every second Thursday of the month.
14. How do you get help about commands in `octave`?
15. Connect a USB web cam to the Raspi and make pictures with the `cheese` program. Use the `imagemagick` tools to cut a small portion from the image (crop) and convert it to another format, say gif.
16. Find out how to store float values in a MySQL database.
17. Find out how to store images in a MySQL database.
18. Measure the brightness with an LDR attached to a NodeMCU and log it once per hour in a MySQL database. If you are patient you can see days getting longer and shorter as the season progress.
19. Log the steps that a stepper motor takes and visualize them on a web page with RRDtool.
20. Make a web page that shows your Arduino sketches.
21. Find out how to specify colors in HTML.
22. Use HTML forms in a web page on the Raspi to set the brightness of a LED on the NodeMCU.
23. Prepare a web page about yourself with picture and all you care to display.

Control System: EPICS

The *Experimental Physics and Instrumental Control System* (EPICS) is used in a number of laboratories to control large particle accelerators, such as the Advanced Photon Source (APS) in the United States, the Swiss light source (SLS) in Zürich, or the European Spallation Source (ESS) in Sweden. Other users of EPICS are the fusion reactor International Thermonuclear Experimental Reactor (ITER) in France and the W. M. Keck astronomical observatory on Hawaii.

EPICS is based on a number of independent computers called *input-output controllers* (IOC) that announce their capabilities on the network such that other computers can interact with them. Almost any type of computer can participate in an EPICS system, and support libraries for many programming languages, such as C, Python, and MATLAB, are available. So it is no surprise that a Raspi can also serve as an IOC and join an EPICS control system, no matter how big it is. We illustrate this by configuring the Raspi to communicate the measurements collected from locally attached microcontroller-based sensor nodes to EPICS. This makes the measurements accessible in a larger control system context.

The first task is to install the EPICS software on the Raspi, and we follow [28] in doing so. The steps are somewhat arcane and the instructions resemble a cookbook. Normally these steps are done by an experienced system administrator.

6.1 INSTALLATION

First we need to download the EPICS Base package from www.aps.anl.gov/epics. At the time of writing, the current release is R3.15.5 and the downloaded package is called `base-3.15.5.tar.gz`. In order to avoid excessive use of `sudo` we create a subdirectory `/home/pi/epics` where all the software resides, but we also create a soft link of that directory to `/usr/local/epics` where it is available to all users on the Raspi. The detailed sequence of commands is the following:

```
cd /home/pi
mkdir epics
sudo ln -s /home/pi/epics /usr/local
cd epics
cp /home/pi/Downloads/base-3.15.5.tar.gz .
tar xzf base-3.15.5.tar.gz
```

where `tar` is an archiving program that unpacks (`x option`) compressed (`z option`) files (`f option`). Note the period `.` at the end of the `cp` command, which is the shorthand

notation for the current directory, `/home/pi/epics` in this case. The last command creates a subdirectory under `/home/pi/epics` with the name `base-3.15.5`. In order to avoid writing the version number over and over again, we create a soft link with the `ln -s` command, which essentially creates an alias in the same directory by executing

```
ln -s base-3.15.5 base
```

and we can henceforth refer to the directory with the EPICS base package via `/usr/local/epics/base` by virtue of the soft link from `/home/pi/epics` to `/usr/local/epics`. Now we have the source code in place, but in order to use it we still need to compile it.

And for the compiler and later the executable programs to find the EPICS sources, we need to copy the following lines into the file `/home/pi/.bash_aliases`:

```
export EPICS_ROOT=/usr/local/epics
export EPICS_BASE=${EPICS_ROOT}/base
export EPICS_HOST_ARCH='${EPICS_BASE}/startup/EpicsHostArch'
export EPICS_BASE_BIN=${EPICS_BASE}/bin/${EPICS_HOST_ARCH}
export EPICS_BASE_LIB=${EPICS_BASE}/lib/${EPICS_HOST_ARCH}
if [ "" = "${LD_LIBRARY_PATH}" ]; then
    export LD_LIBRARY_PATH=${EPICS_BASE_LIB}
else
    export LD_LIBRARY_PATH=${EPICS_BASE_LIB}:${LD_LIBRARY_PATH}
fi
export PATH=${PATH}:${EPICS_BASE_BIN}
```

If the file does not exist, just create a new copy with the above contents. Note that the environment variable `EPICS_ROOT` points to the subdirectory `/usr/local/epics`, the one we created in an earlier step. The other variables are defined relative to that and point to the places where the libraries and executables reside. Once we complete this step, we need to open a new command window, because this rereads the `.bash_aliases` file and we need these definitions before compiling the base system with

```
cd /home/pi/epics/base
make -j 4
```

which takes about 20 minutes depending on the version of Raspi. The option `-j 4` causes the compilation to start four jobs simultaneously, one for each CPU core. The terminal window is filled with the information about what part of the base system is currently compiled. Once the compilation completes without errors, we enter the command `caget` at the command prompt. It should respond with “no pv name specified...” and this indicates that the executables are available and located in the `PATH`. The “pv” is the name of a quantity that EPICS deals with, and is called a *process variable*. An example of such a name is the read-back current of power supply PSabc that might be called `PSabc:current`. Reading or changing the power supply current then refers to that process variable. Reading is done by entering `caget PSabc:current` on the command line. More on that topic comes in the next section. For the next test we start the `softIoc` program, and at the `epics>` prompt that appears, we type `iocInit`. This step should result in the response “iocrun: all initialization complete” and convinces us that we have successfully installed EPICS on the Raspi and can proceed to investigate it.

6.2 COMMUNICATING WITH EPICS

Before connecting the sensor nodes with the microcontrollers to EPICS, we explain the basic EPICS functionality by reading and controlling virtual parameters that only exist in the memory of the Raspi. Again, based on examples from [28], we prepare the following EPICS database file and name it `simple2.db`.

```
# simple2.db
record(bo,"raspi:trigger") {
    field(DESC,"trigger PV")
    field(ZNAM,"off")
    field(ONAM,"on")
}
record(stringout,"raspi:message") {
    field(DESC,"message on the RPi")
    field(VAL,"RPi default message")
}
record(calc, "raspi:random") {
    field(SCAN,"1 second")
    field(CALC,"RNDM")
}
record(ao,"raspi:A") {
    field(DESC,"variable A")
    field(VAL,"0")
}
record(ao,"raspi:B") {
    field(DESC,"variable B")
    field(VAL,"0")
}
record(calc,"raspi:C") {
    field(DESC,"sum of A and B")
    field(SCAN,"1 second")
    field(INPA,"raspi:A")
    field(INPB,"raspi:B")
    field(CALC,"A+B")
}
```

The file contains definitions of six process variables: `raspi:trigger`, `raspi:message`, `raspi:random`, `raspi:A`, `raspi:B`, and `raspi:C` in the record definitions. Each record has a type, which might be `bo`, `bi`, `ao`, or `ai`, standing for binary or analog input or output. Other types are `stringout` or `calc`. Let's describe the records one at a time. The first record is a binary output, `bo`, called `raspi:trigger`, which may be either 1 or 0 and contains three defining fields, a description, a text string for state zero (`ZNAM`), and one for state one (`ONAM`). The second record describes a string that is initialized in the `VAL` field. The third record is of type `calc` and performs some calculation, in this case a rather trivial one: It generates a new random number once per second as specified in the `SCAN` field. The next two records define process variables `raspi:A` and `raspi:B` that are initialized to zero. The sixth record is of type `calc` and calculates the sum of `raspi:A` and `raspi:B` once per second. Once the file `simple2.db` is saved, we start it with the following command:

```
softIoc -d simple2.db
```



```

pi@pidreil1:~
File Edit Tabs Help
pi@pidreil1:~ $ caget raspi:trigger
raspi:trigger          off
pi@pidreil1:~ $ caput raspi:trigger on
Old : raspi:trigger          off
New : raspi:trigger          on
pi@pidreil1:~ $ caget raspi:trigger
raspi:trigger          on
pi@pidreil1:~ $ caget raspi:message
raspi:message          RPi default message
pi@pidreil1:~ $ caput raspi:message Blabla
Old : raspi:message          RPi default message
New : raspi:message          Blabla
pi@pidreil1:~ $ caget raspi:message
raspi:message          Blabla
pi@pidreil1:~ $

```

Figure 6.1 Using the EPICS commands `caget` and `caput`.

which will start an EPICS server and publishes our six process variables. The latter we can verify by typing `db1` (database list) at the `epics>` prompt, which lists the process variables served from the running `softIoc` process.

We interact with the EPICS server from a second terminal window and enter `caget raspi:trigger` in it. The response is the name of the process variable and the current state, which initially is `off`. We change the state by entering `caput raspi:trigger on` and subsequently verify with `caget` that the state has indeed changed. Note that we change process variables with `caput` and retrieve their value with `caget`. We immediately try this with the `raspi:message` process variable by entering `caget raspi:message`, which displays the message from the `simple2.db` file. We change it with `caput raspi:message Blabla`, and the next `caget raspi:message` should display `Blabla` instead. Figure 6.1 shows these transactions. When reading a process variable with `caget`, we can suppress the echo of the variable name by using the `-t` (for terse) option and use `caget -t` instead. A companion program to `caget` is `camonitor`, which monitors one or several process variables and reports whenever one of them changes its value. We use it immediately to verify that the `raspi:random` process variable indeed produces a new random number once every second, by entering `camonitor raspi:random`. The last three records define process variables that are linked in such a way that the third `raspi:C` calculates the sum of the other two process variables. We test this functionality by using `caput` to enter the values 17 and 4 to `raspi:A` and `raspi:B`, respectively. Subsequently reading `raspi:C` will report 21. In passing we mention that a second computer on the same network, either a second Raspi or a desktop computer that has the EPICS base package installed, can access all process variables served by the first Raspi.

This dry run of EPICS worked without interfacing hardware. In order to communicate with our sensor nodes, we need two additional libraries to link EPICS to the hardware.

6.3 ASYN AND STREAM LIBRARIES

The external sensor nodes communicate with the Raspi via serial RS-232 lines, Bluetooth, Ethernet, or WLAN. EPICS requires interface libraries to be able to take over these

communication channels. For this purpose we install two additional packages, **asyn** and **streamdevice**. We start by installing the **asyn** package, which is available from <https://www.aps.anl.gov/epics/download/modules/> as **asyn4-30.tar.gz** (version 4.30 was current at time of writing). We then create a subdirectory **/home/pi/epics/modules/**, copy the downloaded file to it, and unpack the file with the following sequence of commands.

```
cd /home/pi/epics
mkdir modules
cp /home/pi/Downloads/asyn4.30.tar.gz modules
cd modules
tar xzf asyn4.30.tar.gz
ln -s asyn4.30 asyn
```

Before compiling the package, we need to edit the **./asyn/configure/RELEASE** file, change the **EPICS_BASE** variable to **/usr/local/epics/base**, and comment out the lines starting with **IPAC** and **SNCSEQ**. Once this is done we compile by entering **make -j 4** in the directory **/usr/local/epics/modules/asyn** and wait a few minutes for completion.

The second package we install is the **streamdevice** package. For this purpose we create a new subdirectory, **/home/pi/epics/modules/stream**, download the file **StreamDevice-master.zip** from <https://github.com/paulscherrerinstitute/StreamDevice> and unpack the zip file inside the newly created subdirectory

```
cd /home/pi/epics/modules/stream
unzip StreamDevice-master.zip
```

which creates a subdirectory **StreamDevice-master**. To make the build process compatible with using a plain **make** system, we need to remove the **GNUMakefile** from that subdirectory. Once that is done, we execute the following command in **/home/pi/epics/modules/stream**

```
makeBaseApp.pl -t support
```

and edit **/home/pi/epics/modules/stream/configure/RELEASE**. We make sure that **EPICS_BASE** points to **/usr/local/epics/base**, add the following lines to the bottom of the file

```
ASYN=/usr/local/epics/modules/asyn
```

and finally initiate the compilation. We run **make** once in **/home/pi/epics/modules/stream** and a second time in the **StreamDevice-master** subdirectory. This completes the preparation of the basic libraries, and we are ready to write IOCs that talk to our hardware, the microcontrollers with the sensors and actuators attached.

6.4 WRITING AN IOC

As examples, we link the temperature measurement on the UNO, connected by serial line, and on the NodeMCU, connected by WLAN, to EPICS, and publish the measurements as process variables. For this purpose, we create a subdirectory **/home/pi/epics/ioc** and a subdirectory **temp** in it. Inside that subdirectory we execute the following commands:

```
makeBaseApp.pl -t ioc temp
makeBaseApp.pl -i -t ioc temp
```

which creates the basic file infrastructure under **/home/pi/epics/ioc/temp**. We change to that directory with the **cd** command and add the two lines

```

ASYN=/usr/local/epics/modules/asyn
STREAM=/usr/local/epics/modules/stream

```

to the end of the file `./configure/RELEASE`.

Next, we prepare the protocol file that describes the communication protocol we used earlier: send 'T?' and receive 'T 21.2'. The file resides in the subdirectory `./tempApp/Db` under the `temp` base directory for this IOC. Inside it we create the following file named `temperature.proto`

```

# ./tempApp/Db/temperature.proto
Terminator = CR LF;
get_temp {
    out "T?";
    in  "T %f";
    ExtraInput = Ignore;
}

```

The content is rather straightforward to understand. First we define the terminating characters that denote the end of a line, and a function `get_temp` that sends the string `T?` to the device and expects `T` and a float (`%f`) number in return. Moreover, any extra characters should be ignored. It is possible to have more than one function defined in the same protocol file. The protocol file is the lowest level to define the communication; the next higher level is the database file, which we already encountered in Section 6.2. Here we use the following file

```

# ./tempApp/Db/temperature.db
record(ai, "$(USER):temp") {
    field(DESC, "Temperature")
    field(SCAN, "10 second")
    field(DTYP, "stream")
    field(INP, "@temperature.proto get_temp $(PORT)")
}

```

This file defines an analog input `ai` record for a process variable with the name `$(USER):temp`. Here `$(USER)` will be defined in the calling program. The description and update rate of 10 seconds are defined in the first two fields. The third field declares the record to be of type `stream`, and as input function (INP) we use the function `get_temp` from the protocol file `temperature.proto`. We use the communication interface with the name supplied in the variable `$(PORT)`. We then have to edit `./tempApp/Db/Makefile` and add the line

```
DB += temperature.db
```

to it. Next we need to edit `./tempApp/src/Makefile` and add the lines

```

temp_DBD += asyn.dbd
temp_DBD += stream.dbd
temp_DBD += drvAsynSerialPort.dbd
temp_DBD += drvAsynIPPort.dbd

```

after the line with `temp_DBD += base.dbd`, which instructs the build process to include the `asyn` and `stream` libraries as well as support for serial communication and Internet protocol ports such as network sockets. Then, near the bottom, following `temp_LIBS += $(EPICS_BASE_IOC_LIBS)`, add the lines

```
temp_LIBS += asyn
temp_LIBS += stream
```

which are needed to link against the two libraries.

In a last step, we define the startup program `st.cmd` for the IOC that is located in the `./iocBoot/ioctemp/` directory. In this file we add the following line

```
epicsEnvSet(STREAM_PROTOCOL_PATH,"../../tempApp/Db")
```

after the line with `< envPaths`. The added line describes where the protocol files are located. Following the line with `temp_register RecordDeviceDriver pdbname` we add the definition of the serial port we intend to use. For the serial line to the UNO, this looks like

```
drvAsynSerialPortConfigure("SERIALPORT", "/dev/ttyACM0", 0, 0, 0)
asynSetOption("SERIALPORT", -1, "baud", "9600")
asynSetOption("SERIALPORT", -1, "bits", "8")
asynSetOption("SERIALPORT", -1, "parity", "none")
asynSetOption("SERIALPORT", -1, "stop", "1")
asynSetOption("SERIALPORT", -1, "clocal", "Y")
asynSetOption("SERIALPORT", -1, "rtscts", "N")
```

and can be referred to by its symbolic name `SERIALPORT`. Finally, we need to load the database records using the definition of the variables `$(PORT)` and `$(USER)`

```
dbLoadRecords("db/temperature.db", "PORT='SERIALPORT', USER='raspi'")
```

where `SERIALPORT` replaces the place holder `$(PORT)` in the database record file `temperature.db`. Moreover, the name of the process variable is prepended by `raspi`, such that we can later access the temperature with the command `caget raspi:temp`. Now the software setup for the IOC is complete, and we compile it by running `make` in the directory `/home/epics/ioc/temp/`. Once the compilation successfully completes, we make the file `st.cmd` that is located in `./iocBoot/ioctemp/` executable by executing `chmod +x st.cmd` and run it with

```
./iocBoot/ioctemp/st.cmd
```

This starts the EPICS server and the process variable, here only `raspi:temp`, is published on the local network so that any computer with the EPICS base system installed and a working `caget` program can read the temperature from our Raspi.

It remains to connect the NodeMCU microcontroller to EPICS. Since the NodeMCU server from Section 4.6.3 listens on port 1137 at IP number 192.168.20.135 and uses the same protocol (send 'T?', receive 'T 22.1'), we just add the following two lines to the `st.cmd` file:

```
drvAsynIPPortConfigure("SOCKET1", "192.168.20.135:1137", 0, 0, 0)
dbLoadRecords("db/temperature.db", "PORT='SOCKET1', USER='node'")
```

The first line defines a symbol `SOCKET1` that points to the port on the NodeMCU, and the second line instructs EPICS to use the same database file `temperature.db` as before, and link the communication to the `PORT` corresponding to the one defined in the previous line. Once we add the two lines, we need to compile the project again. This we do by issuing `make` in the directory `/home/epics/ioc/temp/` and restarting `st.cmd` by executing `./iocBoot/ioctemp/st.cmd` from the same directory. The result of this exercise is that EPICS now publishes two process variables, `raspi:temp` from before and `node:temp` from

the NodeMCU connected by WLAN, which we can verify by entering the command `dbl` at the `epics>` prompt that `st.cmd` provides.

Most of the above actions we only need to do once, and adding additional sensors to the EPICS IOC is only moderately complex. It only requires writing the appropriate protocol file, say `new.proto` and database file `new.db`, and copying both to the `./tempApp/Db` directory. Then we need to add the database file to the project by adding `DB += new.db` to the Makefile in the same directory. Finally, we need to add `drvAsynXConfigure` to the `st.cmd` file and execute `dbLoadRecords` to link the database file to the appropriate PORT. Finally, we compile once again and run the `st.cmd` command.

6.5 STARTING THE IOC AT BOOT TIME

In the description above we need to start the IOC `st.cmd` by hand and keep a terminal window with the running program open all the time. To remedy this inconvenience, we create a startup script named `epicsioc` that launches the IOC when the system boots. The file contains the following lines:

```
#!/bin/sh
#/etc/init.d/epicsioc
### BEGIN INIT INFO
# Provides:          empicsioc
# Required-Start:    $remote_fs $syslog
# Required-Stop:     $remote_fs $syslog
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Simple script to start a program at boot
# Description:       Start and stop epicsioc server
### END INIT INFO
case "$1" in
  start)
    echo "Starting epicsioc"
    . /home/pi/.bash_aliases
    cd /home/pi/epics/ioc/temp/iocBoot/ioctemp
    /usr/bin/procServ -n "IOC" -L /tmp/epics.log -i ^D^C 20000 ./st.cmd
    ;;
  stop)
    echo "Stopping epicsioc"
    kill $(pidof procServ)
    ;;
  restart)
    $0 stop
    sleep 10
    $0 start
    ;;
  *)
    echo "Usage: /etc/init.d/epicsioc {start|stop|restart}"
    exit 1
    ;;
esac
exit 0
```

The file contains some `INIT INFO` that is required by the `init` program, which orchestrates the startup of the Raspbian operating system. In the subsequent `case` structure, three actions are specified: `start`, `stop`, and `restart`. The `start` option reads the default EPICS environment variables and then changes to the subdirectory with the `st.cmd` program. There it executes the `procServ` program that in turn starts the `st.cmd` program. The purpose of `procServ` is to redirect the input and output of `st.cmd` to the network socket 20000 to which we can connect via `telnet`, and interact with the program in the same way we did earlier when we started it in a terminal window. The advantage of using `procServ` is that we can close the `telnet` program while `st.cmd` keeps running independently. The other options give the running process the name `IOC`, write a log file, and ignore the control sequences to close the process. The `stop` case determines the process id of `procServ` with `pidof` and terminates the process with the `kill` command. The `restart` case executes `stop` first, waits 10 seconds, and then executes `start`. The default case, denoted by `*`, displays a brief usage note.

Before actually running the program, we need to install the `procdserv` and `telnet` packages with the normal installation procedure:

```
sudo apt-get install procdserv telnet
```

In case one of the programs is already present in the current system, no new software is installed. Once all required programs are installed and the script is written, we copy it, using `sudo`, to the system directory `/etc/init.d`, where all startup scripts for the operating system reside. We make it executable using

```
sudo chmod 755 /etc/init.d/epicsioc
```

and start it by hand in order to verify that the script works as intended with the command

```
sudo /etc/init.d/epicsioc start
```

and if everything works we can install it permanently by executing

```
sudo update-rc.d epicsioc defaults
```

which will automatically start `epicsioc` at boot time. If we want to connect to the now-running-autonomously process `st.cmd`, we use `telnet` when logged onto the Raspi

```
telnet localhost 20000
```

and see the normal output of the `st.cmd` program, plus some administrative info from `procServ` prepended by `@@@`. We can enter commands such as `dbl` to list all process variables, and immediately receive the output back on the `telnet` window. The port opened by `procServ` is only accessible from the Raspi itself. If we want to log from another computer, we use `ssh` to first log onto the Raspi, and then execute `telnet`. This can be automated by executing

```
ssh -t pi@192.168.1.22 telnet localhost 20000
```

from a remote desktop computer and where `192.168.1.22` is the IP-number of the Raspi. In this way, the Raspi works unattended as an Epics IOC but we can connect to it at any time using `telnet` in order to follow the performance and interact with `st.cmd`.

QUESTIONS AND PROJECT IDEAS

1. Why is a standardized control system advantageous to use in comparison to a home-grown system?
2. Write protocol and database files to interface the sensors discussed in previous chapters.
3. Write protocol and database files to connect an Arduino UNO running the sketch in Section 4.5.2 to control a DC motor.
4. Write protocol and database files to connect an Arduino UNO running the sketch in Section 4.5.3 to control a model-servo.
5. Write protocol and database files to connect an Arduino UNO running the sketch in Section 4.5.4 to control a stepper motor.
6. Research other control systems used in industry or research institutions.
7. Research the documentation of the stream library on how to treat information that appears on the RS-232 line unsolicited, such as a continuously appearing stream of position data from a GPS receiver.
8. In this chapter we address the Raspi by IP number 192.168.1.22 and in Section 5.4 by 192.168.20.1. Explain, why this not a typo.

Messaging System: MQTT

EPICS is not the only system to integrate a large number of sensors and actuators under a common interface. Another such system is the *message queue telemetry transport* (MQTT) [6] protocol that was originally created by, among others, IBM, in order to collect information from widely distributed infrastructures such as oil pipelines in a battery-saving and energy-efficient as well as robust and secure way. Today it is a protocol commonly used to pass messages between devices that constitute the Internet of Things (IoT) and is now ISO-standard ISO/IEC 20922. MQTT is based on a publish/subscribe method between clients with a message-passing *broker* in the middle. The broker acts as the hub passing messages from one client that publishes data to another that subscribes to published data. The name of the parameters that are passed around are referred to as “topic.” They are organized hierarchically, such that the name

`weatherstations/stationA/node1/temperature`

refers to the temperature sensor on a weather station, called station A and connected to node 1. When subscribing to topics it is possible to use wildcards such as the plus-sign “+”, which is a single-level wild-card, or the hash “#,” which is a multi-level wild-card. Robustness and reliability of transmission is guaranteed by specifying three levels of *quality of service* (QoS), where the simplest level 0 implies that the sensor only publishes data without any acknowledgment from the broker. Levels 1 and 2 implement increasingly advanced levels of handshake signals that ensure the arrival of data. Even a *last will* is available, which is transmitted to subscribing clients should a publishing client disconnect. Normally the broker immediately transmits any received data to the subscribing clients, but it is possible to specify that the broker retains the last good data point and transmits it to subscribers in case the publisher is offline, and to new subscribers upon their first connection. Since MQTT was intended to operate across public networks, encryption and security are part of the protocol.

The core functionality of MQTT resides on the broker, such that clients can be very simple and connect and disconnect at will. This makes it possible to use clients that only wake up from a battery-saving deep sleep mode, perform a measurement, send it to the broker, and go back to sleep again. This attractive feature makes MQTT very popular for IoT applications and we therefore also discuss it as a complement to EPICS. In the following sections we operate a broker on the Raspberry Pi and use NodeMCUs as clients that publish and subscribe. In a final section we discuss a simple gateway that links MQTT to EPICS in order to benefit from the best of both worlds. In this way we can access lightweight clients connected over a public network across the gateway from our EPICS control system.

After this overview of MQTT, let us follow the theme of the book and describe a working system that provides the basic functionality.

7.1 BROKER

The `mosquitto` broker is widely supported on many platforms, and also on the Raspberry Pi. We install it, after bringing the system up to date with `sudo apt-get update` and `sudo apt-get upgrade`, using the command

```
sudo apt-get install mosquitto mosquitto-clients
```

where the `mosquitto` package contains the broker and the `mosquitto-clients` package contains the command line executables `mosquitto_pub` and `mosquitto_sub`. We use them to test the base functionality on the Raspi alone, without external clients.

Immediately after the installation, `mosquitto` is already running and is registered as a service that starts after every reboot. During our initial tests we inspect the logging output with the following command, which helps to debug the problem in case something unexpected happens:

```
tail -f /var/log/mosquitto/mosquitto.log
```

where `tail -f` takes the filename as argument and shows the last lines of the file, but, instead of stopping the display at the end of the file, keeps appending newly generated messages. Now we open a second terminal window and start publishing MQTT data using the `mosquitto_pub` command-line client with

```
mosquitto_pub -d -h localhost -i Pub1 -t dummy/value -m 42 -r
```

where `mosquitto_pub` is the executable, `-d` enables debugging output, and `-h` specifies the IP address of the broker; here it is `localhost`, because the broker runs on the same Raspi where we run the `mosquitto_pub` client. The parameter specified after `-i` identifies the publisher, `-t` the topic, and `-m` specifies the message, here the value 42. Appending `-r` instructs the broker to retain the message and send it in case the publisher is offline. The debugging output can be disabled by omitting the `-d` option. Running `mosquitto_pub -h` gives a short overview of available commands, and the manual page, accessible with `man mosquitto_pub`, provides more information.

The `mosquitto_sub` executable provides the receiving end of MQTT message-passing. We run it in another terminal window where we execute

```
mosquitto_sub -d -h localhost -i Sub1 -t dummy/value
```

to subscribe to the topic `dummy/value` on the broker running on `localhost`. We identify the subscribing client by `Sub1` and enable debugging with `-d`. After starting the executable, we are greeted with either an empty line or with the last published message that has the retain flag `-r` enabled. The `mosquitto_sub` keeps running, and if we publish new messages in the second terminal with `mosquitto_pub`, they immediately appear in the subscriber window.

At this point we have a working broker whose functionality we verified with the command-line programs `mosquitto_pub` and `mosquitto_sub` running on the same Raspi as the broker. In the next section we replace these command-line clients by external clients running on NodeMCUs that publish temperature data and at the same time subscribe to another topic to turn a cooling fan on or off.

7.2 NODEMCU CLIENTS

The hardware connected to the NodeMCU is very simple in this case. We connect ground and supply pins of an LM35 temperature-sensor to the respective pins on the NodeMCU. The signal pin of the LM35 is connected to the single analog input A0. Second, we connect a transistor to an output pin as shown in previous chapters to turn the fan on and off. In order to use MQTT functionality in the Arduino IDE, we need to install one of the many available libraries. We select the “MQTT PubSub” library and install it from within the IDE by going to the library manager located at *Sketch*→*Include library*→*Manage Libraries*, enter “MQTT PubSub” in the search field, and install the suggestion that shows up. The following sketches are based on the `mqtt_esp8266` example that is included in the installation.

It turns out that adapting the example code to write a sketch that serves as both publisher and subscriber is straightforward, and the following code fulfills that purpose.

```
// MQTT client, V. Ziemann, 170816
const char* ssid = "messnetz";
const char* password = ".....";
const char* broker = "192.168.20.1";
const int fan_pin=D4;
#include <Ticker.h>
volatile uint8_t do_something=0;
Ticker tick;
void tick_action() {do_something=1;} // executed regularly
#include <ESP8266WiFi.h>
#include <PubSubClient.h>
WiFiClient espClient;
PubSubClient client(espClient);
void on_message(char* topic, byte* msg, unsigned int length) {
    char ch[30]; memcpy(ch,msg,length); ch[length]='\0';
    if (strstr(topic,"node1/fan")==topic) {
        int val=(int)atof(ch);
        Serial.print(" Fan="); Serial.println(val);
        if (val==0) {
            digitalWrite(fan_pin,HIGH);
        } else {
            digitalWrite(fan_pin,LOW);
        }
    } else if (strstr(topic,"node2/temp")==topic) {
        Serial.print("Temperature on node2 = "); Serial.println(ch);
    }
}
void setup() { //.....setup
    pinMode(fan_pin,OUTPUT);
    Serial.begin(115200);
    WiFi.begin(ssid,password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500); Serial.print(".");
    }
    Serial.print("\nWifi connected to "); Serial.println(ssid);
    Serial.print("with IP address: "); Serial.println(WiFi.localIP());
    client.setServer(broker, 1883); // 1883 = default MQTT port
```

```

    client.setCallback(on_message); // execute when a message arrives
    tick.attach(5,tick_action); // execute tick_action every 5 seconds
}
void loop() { //.....loop
    while (!client.connected()) {
        if (client.connect("PubSub1")) { // identification
            client.subscribe("node1/fan"); // external fan control
            client.subscribe("node2/temp"); // temp on other nodemcu
        } else {
            delay(5000);
        }
    }
    client.loop();
    if (do_something) {
        do_something=0;
        char message[30];
        int temperature=(int)(3.3*100*analogRead(A0)/1023.0);
        sprintf(message,"%d",temperature);
        client.publish("node1/temp",message);
    }
}
}

```

In the first few lines of the sketch we define the usual network credentials and the IP number of the broker as well as the pin to which the fan is connected. Then we include the header for the Ticker library, which provides a timer that is executed at regular intervals. After declaring the variable `do_something`, we declare the `tick` object and define the function `tick_action()` to be executed in regular intervals. All this function does is to set the variable `do_something` to one. We declare `do_something` volatile, because it changes asynchronously from the main loop. Next we include the WiFi header, the MQTT PubSubClient functionality, and declare both `WiFiClient` and a `PubSubClient` named `client`. The following function named `on_message()` is executed every time a MQTT message arrives. Its arguments are the topic, the message, and the length of the message. Within the function we first convert the received message to a character string, because we want to handle it using the same mechanism we used in previous chapters. Then we enter the usual construction where we check which topic has arrived; if it is `node1/fan` we convert the message to an integer value `val` and turn a pin on or off, depending on whether `val` is zero or not. If the received topic is `node2/temp` we only display it on the serial line, but could easily add a test to turn the fan on or off, depending on the temperature received.

Once variables and auxiliary functions are declared, we define the `setup()` function and configure the mode of the used pins, the serial line, and WiFi. The function `client.setServer()` connects to the broker on the default MQTT port 1883, and the call to the `client.setCallback()` function registers the function `on_message()` to be executed when a new message arrives. Finally, we start the ticker process to execute the function `tick_action()` once every 5 seconds. In the `loop()` function we first ensure that the connection to the broker is up and running. If it is not running we connect to the broker with the call to the `client.connect()` function and provide the identification of the NodeMCU as `PubSub1`. In the above example using `mosquitto_sub`, this corresponds to the parameter following the `-i` command line switch. The subsequent calls to `client.subscribe()` register the strings used as argument with the broker, which subsequently sends any updated values. If the connection with the broker fails, a new attempt is made after 5 seconds.

Once the connection to the broker is established, we call the `client.loop()` function to service any background activities related to MQTT, and finally check whether the variable `do_something` was set, which happens every time the ticker fires. If `do_something` was set, we reset it to zero and publish the parameter `node1/temp`.

In a terminal window on the Raspi, we can start `mosquitto_sub` to subscribe to the topic `node1/temp` and should see the updated temperature every 5 seconds. Furthermore, in order to test multiple NodeMCU clients to communicate, we can program a second NodeMCU with the same sketch, but swap the reference to `node1` and `node2`. Moreover, if we only want to publish values from a NodeMCU client, all code related to receiving messages can be removed from the sketch, such as the `on_message()` function and the two calls to `client.subscribe()`. If we want to use a client with subscription-only functionality, we can remove everything related to the tick function and the variable `do_something`. In any case, using the above sketch as a base should make it possible to serve almost any need to connect NodeMCUs to a MQTT network.

So far, the MQTT system is unrelated to other control systems such as EPICS. So, if we require interoperability, we need to provide a gateway that translates the message formats, and that is what we describe in the next section.

7.3 GATEWAY TO EPICS

We want the gateway to seamlessly translate between the systems and to achieve the following functionality: The gateway will listen on a network socket on port 51883 to communicate with EPICS, and uses default ports for anything else. On EPICS we intend to use stream-based protocol files such that all strings that EPICS sends to our gateway will have the format `topic value`, and we configure the gateway to publish this as topic `topic` with the message contents `value`. This is all we need to do to publish MQTT topics from EPICS. In order to receive MQTT messages on EPICS, we need to configure the gateway to subscribe to the messages and pass any incoming messages on to EPICS. For this we decide that any message from EPICS to the gateway having the format `SUBSCRIBE topic` will instruct the gateway to subscribe to `topic`. We also implement a command to `UNSUBSCRIBE` as a matter of order.

We chose to implement the gateway using the Python language, because it has powerful support for both conventional network sockets and for MQTT. The former is included in standard installations, and for MQTT we use the `paho.mqtt` library that we install from the command line using the Python installer program `pip` with command

```
sudo pip install paho-mqtt
```

and after installing it we are ready to write the gateway. The powerful libraries make the following Python-program rather compact.

```
# Epics to MQTT gateway, V. Ziemann, 170817
import socket,atexit,paho.mqtt.client
def cleanup(): #.....cleanup
    sock.close()
    atexit.register(cleanup)

def on_message(c,u,msg): #.....on_message
    print "msg = ",msg.topic," ",msg.payload," ",msg.qos
    epics.send(msg.topic + " " + msg.payload + "\r\n")
mqttc=paho.mqtt.client.Client()
```

```

mqttc.connect("localhost",1883)
mqttc.on_message=on_message
mqttc.loop_start()

sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
sock.bind('',51883)
sock.listen(1)
while 1:
    epics,address = sock.accept()
    print "Connected from ",address
    while 1:
        msg=epics.recv(1024)
        if not msg: break
        words=msg.split()
        if len(words)<2: break;
        if words[0].upper()=="SUBSCRIBE":
            mqttc.subscribe(words[1],1)
        elif words[0].upper()=="UNSUBSCRIBE":
            mqttc.unsubscribe(words[1])
        else:
            mqttc.publish(words[0],words[1])
        pass
    epics.close()
    print "Disconnect from ",address

```

At the start of the program, we import support for sockets, MQTT and `atexit`. The latter is used to execute code asynchronously when terminating the program, by registering the `cleanup()` function to close the network socket. Next we define the function `on_message`, which is called when a subscribed MQTT message arrives. After printing the message, which is useful for debugging, we send it via the network socket named `epics` to EPICS. Note that we append both a carriage return `\r` and a line feed `\n` in order to match the convention for line terminators used in EPICS. Now we are ready to define the MQTT client `mqttc`, and connect it to the broker that runs on the same computer as the gateway, namely on `localhost`, and to the standard MQTT port 1883. On the following line we register the `on_message` function to be executed at every `on_message` event of `mqttc`, and start the MQTT event-loop with the `mqttc.loop_start()` function. Note that the sequence of connecting to the broker, registering a call-back function for arriving messages, and starting the event loop mimics the code in the sketch that runs on the NodeMCU. After configuring the MQTT connections, we define a socket `sock` that uses the IPv4 (`AF_INET`) protocol and TCP (`SOCK_STREAM`). We bind the socket to port 51883 where it listens for requests from EPICS. We instruct the socket to only accept one connection at a time and enter into an infinite loop in which the socket waits for an incoming request from EPICS in the function `sock.accept()`. This function blocks until a request arrives, when it returns a handle `epics` to the new connection and the IP number `address` of the connecting computer. Once the connection is established, the gateway enters a second loop and waits for a command on the `epics` socket. If a message with an invalid format arrives, the connection closes and otherwise splits the received message `msg` into words, and then branches depending on the first word. If, after conversion to uppercase, it is `SUBSCRIBE`, the gateway calls the `mqttc.subscribe()` function with the topic as argument. If it is `UNSUBSCRIBE` it calls the `mqttc.unsubscribe()` function. In all other cases the two words are interpreted as topic

and value, and published using the `mqttc.publish()` function. If messages with incorrect format are received or the calling EPICS computer disconnects, the `epics` socket closes and prints a message. At this point the outer `while 1:` loop is still active and the gateway reverts to the `sock.accept()` function and waits for new connections.

We start the gateway by executing

```
python epics2mqtt.py
```

from the command line, and writing a boot script similar to the one for EPICS from Section 6.5 is left as an exercise. The basic functionality of the gateway can be easily tested using `netcat` to emulate EPICS and send strings to the gateway. If the terminal window with the `mosquitto_sub` command from the end of Section 7.1 is still running, we can send messages to it with the command

```
echo "dummy/value 57" | netcat -C localhost 51883
```

which sends the string following `echo` to socket 51883 on the local computer, but this is where the gateway listens, and publishes this message on MQTT on our behalf.

If we connect to the gateway with the command `netcat -C localhost 51883` executed from the command line, we can issue the string `SUBSCRIBE node1/temp` to subscribe to the topic `node1/temp`. Executing the following command in another terminal window

```
mosquitto_pub -d -h localhost -i Pub1 -t node1/temp -m 23
```

will cause the string `node1/temp 23` to appear in the window with `netcat` running. Issuing the command `UNSUBSCRIBE node1/temp` in `netcat` stops the subscription, and messages will no longer appear in the `netcat` window.

Finally, we prepare EPICS protocol and database files that implement the following behavior. Executing `caput node1/fan 1` publishes the topic `node1/fan` with message 1, and `caput SUBSCRIBE node1/temp` subscribes to the topic `node1/temp` such that we asynchronously receive the messages in EPICS. The first turns the fan on and the second reports the temperature measured by the LM35 on the NodeMCU. The protocol file to implement this behavior is stored in the following file, named `mqtt.proto`:

```
# ./tempApp/Db/mqtt.proto
Terminator = CR LF;
set_fan {out "node1/fan %i";}
subscribe {out "SUBSCRIBE %s";}
unsubscribe {out "UNSUBSCRIBE %s";}
get_temp {in "node1/temp %f";}

```

First we define the Terminator that matches the `\r\n` used in the gateway code, and then define functions that either input or output values in the same way we used in the previous chapter. Note that the functions have the MQTT names hard coded as character strings and that the functions to subscribe and unsubscribe have a string as argument. The corresponding database file, called `mqtt.db`,

```
# ./tempApp/Db/mqtt.db
record(ao, "node1/fan") {
    field(DESC, "Fan on node1")
    field(DTYP, "stream")
    field(OUT, "@mqtt.proto set_fan $(PORT)")
}

```

```

record(stringout, "SUBSCRIBE") {
    field(DESC, "subscribe to topic")
    field(DTYP, "stream")
    field(OUT, "@mqtt.proto subscribe $(PORT)")
}
record(stringout, "UNSUBSCRIBE") {
    field(DESC, "unsubscribe from topic")
    field(DTYP, "stream")
    field(OUT, "@mqtt.proto unsubscribe $(PORT)")
}
record(ai, "node1/temp") {
    field(DESC, "Temperature on node1")
    field(DTYP, "stream")
    field(INP, "@mqtt.proto get_temp $(PORT)")
    field(SCAN, "I/O Intr")
}

```

links the functions defined in the protocol file to process variables where the MQTT variables have the same name in EPICS, and we add two process variables handling subscription. Since they register a name of a variable, their EPICS record type is `stringout`. All variables that we receive from MQTT via the gateway arrive asynchronously, at the rate they are published, and we therefore need to use the line `field(SCAN, "I/O Intr")` in the database file, which handles data that arrive without explicitly being requested. Note that we only need to subscribe to values we want to read with `caget`. MQTT variables we want to set, such as turning on the fan, need no subscription.

After having defined the protocol and database file, we need to add `mqtt.db` to the Makefile in the `./temp/tempApp/Db` directory and add the following definition for the `PORT` in `st.cmd` in `./iocBoot/ioctemp/`.

```

drvAsynIPPortConfigure("SOCKET", "192.168.20.1:51883", 0, 0, 0)
dbLoadRecords("db/mqtt.db", "PORT='SOCKET', USER='mqtt'")

```

The IP address points to the IP on which both broker and gateway run, and the port on which the gateway listens. Once this is operational, we have access to MQTT topics from EPICS.

We've come a long way, starting with small sensors that are connected to microcontrollers, which, in turn, communicate with a more powerful computer, a Raspi in our case, to retrieve, store, and present the measurement values. And finally, we even connected the Raspi to full-grown control systems where all measurement values are published and are available to all computers on that network. Since there is a large number of EPICS programs available to perform online analysis, logging, and alarm management, among other tasks, we can say that the individual measurement values from our sensors have trickled up the data-handling chain, all the way to a generic top layer. And that was the main task we initially set out to illustrate.

All examples we encountered so far were deliberately chosen to be rather simple, in order to illustrate the mechanisms, but in the coming chapters we advance to more complex projects, and start with a weather station with distributed sensors.

QUESTIONS AND PROJECT IDEAS

1. What is the purpose of the broker?
2. Find out about public brokers on the Internet.
3. Connect the humidity and barometric pressure sensor to a NodeMCU and have the data published via MQTT.
4. Control the brightness of a LED connected to a NodeMCU via MQTT.
5. Connect a DC motor with H bridge to a NodeMCU and control it via MQTT.
6. Connect a stepper motor with H bridge to a NodeMCU and control it via MQTT.
7. Connect a model-servo to a NodeMCU and control it via MQTT.
8. Prepare a boot script for the gateway following the example in Section 6.5.
9. Write gateway between MQTT and a MySQL database such that the database is automatically filled with newly arriving data samples.
10. Write gateway that fills an RRDtool database with data from MQTT.
11. Write a gateway to interface `octave` to MQTT such that a plot is continuously updated with new data.
12. Find an MQTT client for your smartphone and use it to read the temperature from the NodeMCU.
13. Discuss the similarities and differences between EPICS and MQTT. Under what circumstances do you prefer one or the other?
14. Write an MQTT client for the NodeMCU that mimics the query-response behavior: It receives a query, does something, and publishes a response.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Example: Weather Station with Distributed Sensors

Our first project is a weather station that measures barometric pressure, humidity, and temperature at a number of locations inside and outside a building. We use `rrdtool` to prepare plots of the measurements over periods of 4 hours, 2 days, a week, and a month. Moreover, we prepare database files to enable integration into an EPICS control system. We select the NodeMCUs as microcontrollers for the sensor nodes because they are very easy to program, and flexible that they may be deployed all over the place without having to pull wires. Note that we only show how to connect a single sensor node, but multiple copies only differ by their IP number. We can duplicate any interfacing software by simply changing the IP number appropriately.

As sensors, we choose the I2C-based sensors BMP180 for barometric pressure and HYT221 for humidity, both of which also provide temperature information. Since the analog input is not used, we may add an LM35 temperature sensor as well, even though it is not necessary. We show such a sensor node built on a breadboard in Figure 8.1. We see the NodeMCU on the right, which is connected to the power rails via its 3.3 V and ground terminals. The three sensors are placed towards the left on the breadboard and their respective pins are also connected to the power rails. The I2C pins for SDA and SCL for the barometric and the humidity sensor are directly connected to the respective pins on the NodeMCU microcontroller, which are D1 for SCL and D2 for SDA. The pins for positive supply voltage and ground of the LM35 are connected to the lower power rail and its analog output pin to the analog input pin A0 of the NodeMCU. We also added 4.7 μ F and 100 nF decoupling capacitors to the power rails.

The program running on the microcontroller follows the template we used earlier, and is shown below.

```
/* Simple socket server to serve barometric pressure, humidity,
 * and temperature. Author: V. Ziemann, 170117
 */
#include <SFE_BMP180.h>
#include <Wire.h>
SFE_BMP180 pressure;
double P,TP,H,TH;
const int HYT=0x28; // I2C address for HYT221
const char* ssid    = "messnetz";
```

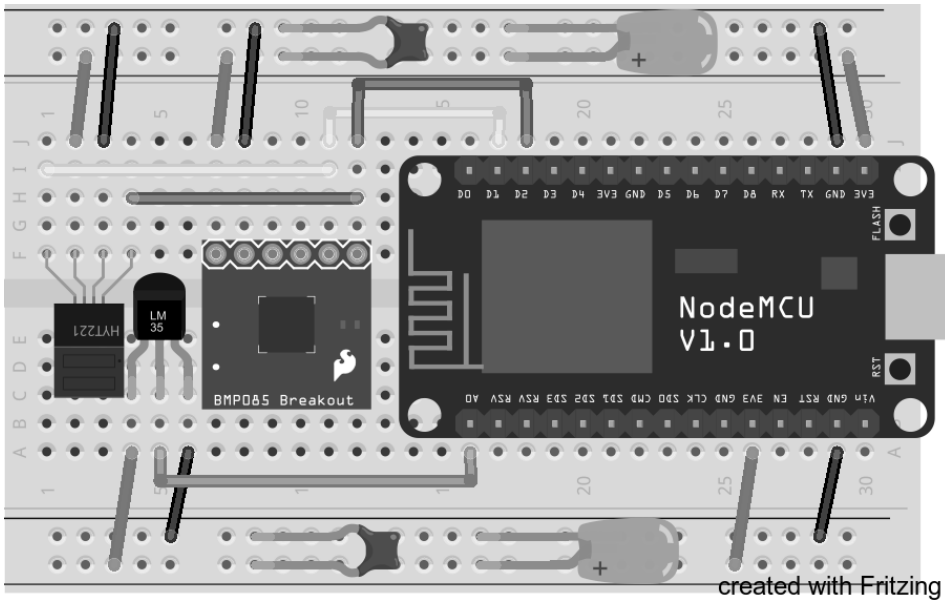


Figure 8.1 The weather-node circuit with the NodeMCU on the right and the LM35, HYT221, and BMP180 sensors towards the left.

```
const char* password = ".....";
const int port = 1137;
#include <ESP8266WiFi.h>
WiFiServer server(port);
void setup() { //.....setup
  pinMode(LED_BUILTIN,OUTPUT);
  digitalWrite(LED_BUILTIN,LOW);
  Serial.begin(115200);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.print("\nWiFi connected to "); Serial.println(ssid);
  Serial.print("Server IP address: "); Serial.println(WiFi.localIP());
  server.begin();
  Serial.print("Server started on port "); Serial.println(port);
  digitalWrite(LED_BUILTIN,HIGH);
  if(!pressure.begin()) {Serial.println("Cannot init BMP180!");}
}
void loop() { //.....loop
  char line[30];
  float volt,temp;
  int t0;
  WiFiClient client = server.available();
```

```

while (client) {
  digitalWrite(LED_BUILTIN,LOW);
  while(!client.available()) {
    delay(5);
    if (!client.connected()) break;
  }
  client.readStringUntil('\n').toCharArray(line,30);
  Serial.print("Request: "); Serial.println(line);
  if (strstr(line,"TH?")==line) {
    client.print("TH "); client.println(TH,2);
  } else if (strstr(line,"H?")==line) {
    int b1,b2,b3,b4,raw;
    Wire.beginTransmission(HYT);
    Wire.requestFrom(HYT,4);
    delay(200);
    if (Wire.available()==4) {
      b1=Wire.read(); b2=Wire.read(); // humidity rawdata
      b3=Wire.read(); b4=Wire.read(); // temperature rawdata
      Wire.endTransmission();
    }
    raw=(256*b1+b2) & 0x3FFF; // humidity
    H=100.0*raw/16384.0;
    raw=((256*b3+b4) & 0xFFFC)/4; // temperature
    TH=165.0*raw/16384.0-40.0;
    client.print("H "); client.println(H,2);
  } else if (strstr(line,"TP?")==line) {
    client.print("TP "); client.println(TP,2);
  } else if (strstr(line,"P?")==line) {
    int is=pressure.startTemperature();
    if (is!=0) {
      delay(is);
      is=pressure.getTemperature(TP);
      if (is!=0) {
        is=pressure.startPressure(3); // oversampling=3
        delay(is);
        is=pressure.getPressure(P,TP);
        if (is!=0) {
          Serial.print("BMP180: T="); Serial.print(TP,2);
          Serial.print(" P="); Serial.println(P,2);
          client.print("P "); client.println(P,2);
        } else {
          Serial.println("Error: BMP180 getPressure failed");
        }
      } else {
        Serial.println("Error: BMP180 getTemperature failed");
      }
    } else {
      Serial.println("Error: BMP180 startTemperature failed");
    }
  }
}

```

```

    } else if (strstr(line,"T?")==line) {
        temp=100*3.3*analogRead(0)/1023;
        client.print("T "); client.println(temp,1);
    } else {
        Serial.println("unknown command, disconnecting");
        client.stop();
    }
    client.flush();
}
digitalWrite(LED_BUILTIN,HIGH);
yield();
}

```

At the top we include the I2C library `Wire.h`, and we use the `SFE_BMP180` library to interface the BMP180 pressure because it makes the sketch more compact. Next, we instantiate the `pressure` object that provides the pressure and temperature measurements from the sensor. Then we define the I2C address for the HYT221 humidity sensor and the WLAN information, before creating the `WiFiServer server()`. In the `setup()` function we first establish the WLAN connection and then start the server, just as in Section 4.6.3. At the end of the `setup()` function, we initialize the pressure sensor and report an error, if this fails. In the `loop()` function, we first declare some variables before waiting for a client to connect. Once it is available, we wait until a request from the client is received, and then it is decoded in the following `strstr()` construction. In the request for the humidity measurement (H?), both the humidity and the temperature from the sensor are determined, but only the humidity returned. The temperature that is returned as a result of a TH? request is therefore always the one determined in the previous request for humidity. The same mechanism is used for the pressure measurement, which is started by a P? request. It retrieves both pressure and temperature data from the sensor. Last, the temperature from the LM35 analog sensor can be requested. If an unknown command is received, the network connection is closed. Finally, the `yield()` function permits the microcontroller to do internal bookkeeping.

On the Raspi, we use a `cron` job to query the microcontroller once every minute to read the measurement values and place them into a `rrdtool` data base. The following Python script reads the five values from the microcontroller via WLAN, and prepares an output string called `out` with values separated by a colon “:,” which already conforms to the syntax of the `rrdtool` `update` command.

```

# read_from.py reads queries from socket 1137 at IP
import socket, atexit, time, sys
def cleanup():
    sock.send("quit\n")
    sock.close()
if len(sys.argv) < 2:
    print("Usage: read_from.py IP <list of queries>")
    sys.exit(2)
else:
    atexit.register(cleanup)
    sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    sock.connect((sys.argv[1],1137)) # socket 1137
    out=''
    for query in sys.argv[2:]:

```

```

sock.send(query + "?\n")
time.sleep(0.3)
reply=sock.recv(1000);
if (len(out))>0:
    out+=':'
out+=reply[len(query):].strip()
print(out)

```

The above Python script requires the IP number of the sensor node and the list of parameters to query as command-line arguments. It should be called in the following form:

```
python read_from.py 192.168.20.144 P TP H TH T
```

and it will return the five values separated by colons. In the script, we first import the necessary functionality and define the `cleanup()` function to close the socket at the end of a transaction. Then we check whether sufficient command line arguments are present, and exit with a usage note. If sufficient command-line arguments are present, we register the `cleanup()` function, open the socket to the IP number specified as the first command-line argument, and then loop over the list of the rest of the command line arguments. Inside the loop we build the query by appending the question mark and newline character, wait a short time to give the sensor node some time to do the measurement, and then receive its reply. The next two lines strip the unwanted characters from the reply and prepend a colon unless the first entry is handled. Finally the output string `out` is returned.

The following shell script calls the `read_from.py` script to obtain the measurement values, and stuffs them into the database.

```

#!/bin/bash
DB=/home/pi/rrdtool/weather.rrd
TEMP=$(/usr/bin/python /home/pi/rrdtool/read_from.py \
    192.168.20.144 P TP H TH T)
/usr/bin/rrdtool update $DB N:$TEMP

```

Note that we place the arguments of `read_from.py` on the following line, and use the continuation character, a backslash, to indicate that. But before we start to fill the database with the `rrdtool update` command, we must create it with the following command. We assumed it is executed in the `/home/pi/rrdtool` directory.

```

rrdtool create weather.rrd --step 60 \
    DS:P:GAUGE:180:900:1100 \
    DS:TP:GAUGE:180:-20:100 \
    DS:H:GAUGE:180:0:100 \
    DS:TH:GAUGE:180:-20:100 \
    DS:T:GAUGE:180:-20:100 \
    RRA:AVERAGE:0.5:1:2880 \
    RRA:AVERAGE:0.5:10:2880 \
    RRA:AVERAGE:0.5:60:2880

```

It creates the `weather.rrd` database file that expects values once every 60 seconds, and contains five measurement columns (DS) for the pressure, temperature from the pressure sensor, humidity, temperature from the humidity, sensor and the temperature from the LM35 sensor. Then it defines the archive (RRA), which is filled by single averages for 2880 samples, which amounts to 2 days. The next two lines define averages over 10 samples or 10

minutes, for a total of 2880 samples or 20 days, and finally, hourly averages over 60 samples for 120 days.

Now we just have to wait until some measurements make it into the database, and then create graphs in the same way we did in Section 5.6.3 using the `rrdtool graph` command. The command to create one set of graphs for temperature, pressure, and humidity, called `weathergraph.sh`, is the following:

```
#!/bin/bash
# /home/pi/rrdtool/weathergraph.sh
S=$1
DB=/home/pi/rrdtool/weather.rrd
PICDIR=/home/pi/public_html/weather
/usr/bin/rrdtool graph $PICDIR/temperature${S}.png -s $S \
    -w 800 -h 200 \
    -t "Temperature" -v "T [C]" -l 16 -u 26 -r \
    DEF:t0=$DB:T:AVERAGE LINE1:t0#FF0000:"LM35" \
    DEF:t1=$DB:TP:AVERAGE LINE1:t1#00AF00:"T(P)" \
    DEF:t2=$DB:TH:AVERAGE LINE1:t2#0000FF:"T(H)";
/usr/bin/rrdtool graph $PICDIR/pressure${S}.png -s $S \
    -w 800 -h 200 \
    -t "Barometric Pressure" -v "P [mbar]" -l 950 -u 1050 -r \
    DEF:t0=$DB:P:AVERAGE LINE1:t0#FF0000:"Pressure";
/usr/bin/rrdtool graph $PICDIR/humidity${S}.png -s $S \
    -w 800 -h 200 \
    -t "Humidity" -v "Humidity [%]" \
    DEF:t0=$DB:H:AVERAGE LINE1:t0#FF0000:"Humidity";
```

It starts by copying the first command line argument, which should contain the start time of the display to the variable `S`, declares the database file to use, and the location where the graphs will be stored. For this we choose a directory under `/home/pi/public_html`, which is accessible to the `apache` web server. The first `rrdtool graph` produces the temperature plot with the temperatures from the three sensors. Note that the graph file carries the time period appended to its name. The second produces the pressure, and the last the humidity graph, all in the same way discussed in Section 5.6.3. In order to produce graphs with time spans of 8 hours, 1 day, 1 week, 1 month, and 3 months, we call `weathergraph.sh` with different starting dates as command line arguments, and place the five commands in a separate script, called `allweathergraph.sh`. It is reproduced here.

```
#!/bin/bash
# /home/pi/rrdtool/allweathergraph.sh
/home/pi/rrdtool/weathergraph.sh -8h > /dev/null
/home/pi/rrdtool/weathergraph.sh -1d > /dev/null
/home/pi/rrdtool/weathergraph.sh -1w > /dev/null
/home/pi/rrdtool/weathergraph.sh -1m > /dev/null
/home/pi/rrdtool/weathergraph.sh -3m > /dev/null
```

The call to `weathergraph.sh` uses the absolute path because we want to run it from a cron job to create new graphs every 10 minutes. We also redirect the output into the big bit-bucket `/dev/null`. We execute this script regularly as a cron job with the command `crontab -e` in order to add the line

```
*/10 * * * * /home/pi/rrdtool/allweathergraph.sh
```

Raspi Weather Station (8 hours)

Display data for: [8 hours](#) [1 day](#) [1 week](#) [1 month](#) [3 months](#)

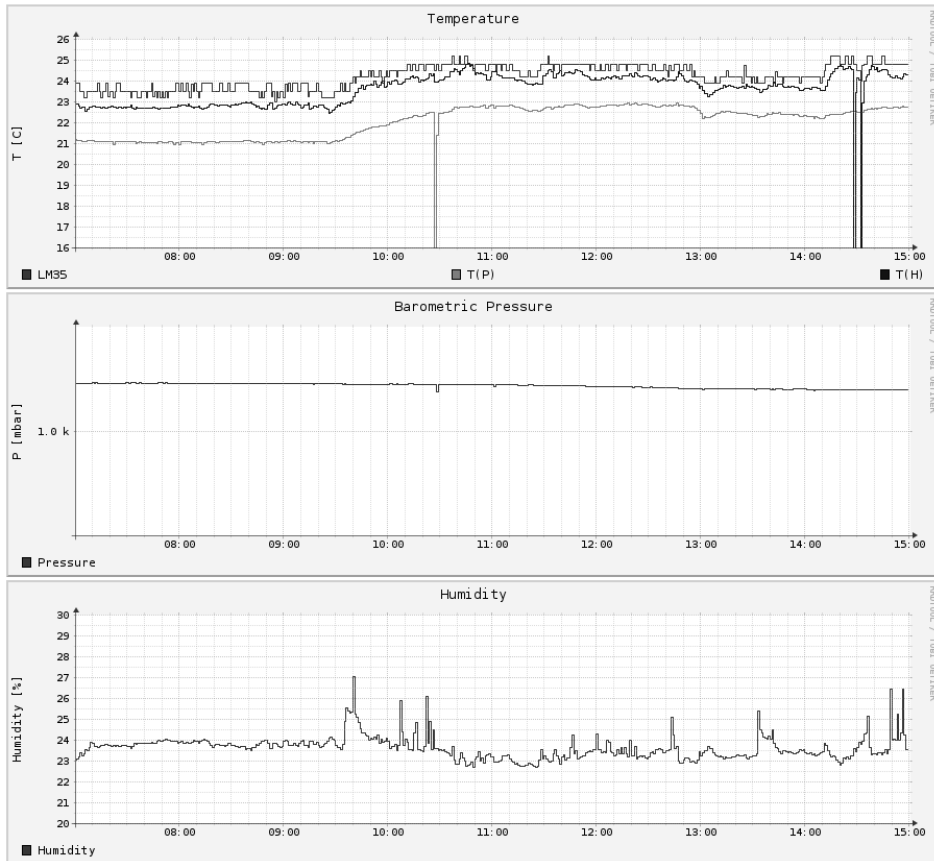


Figure 8.2 The weather station web page served by the Raspi.

in the crontab file.

It remains to prepare the `index.html` file in the directory with the graphs. The following HTML file presents the data as the simple web page shown in Figure 8.2.

```
<!DOCTYPE HTML>
<HTML>
  <HEAD>
    <TITLE>Raspi Weather Station</TITLE>
    <META http-equiv="refresh" content="120">
  </HEAD>
  <BODY>
    <H1 ALIGN=CENTER>Raspi Weather Station (8 hours)</H1>
    Display data for:
    <A HREF=index.html>8 hours</A>__
```



```

<A HREF=index-1d.html>1 day</A>__
<A HREF=index-1w.html>1 week</A>__
<A HREF=index-1m.html>1 month</A>__
<A HREF=index-3m.html>3 months</A>
<HR>
<BR>
<IMG SRC="temperature-8h.png" ALT="Temperature">
<IMG SRC="pressure-8h.png" ALT="Pressure">
<IMG SRC="humidity-8h.png" ALT="Humidity">
</BODY>
</HTML>

```

The file contains the already-known header information with the title **Raspi Weather Station**, and the automatic refresh rate of 120seconds. Then a line with options of different time ranges follows. Each entry points to a different web page residing in the same directory, but serving the graphs for the other ranges. Remember, the graphs are automatically updated by the cron job, but the different `index-XX.html` pages are constant.

Having the weather data displayed for different time ranges is certainly adequate for home use, but in a laboratory we want to integrate the data from the sensor nodes into the control system. If we already have the EPICS system running, all we have to do is to prepare the protocol and database files for our weather data. The former, written in a more compact way, is the following.

```

# ./tempApp/Db/weather.proto
Terminator = CR LF;
get_T {out "T?"; in "T %f"; ExtraInput = Ignore; }
get_P {out "P?"; in "P %f"; ExtraInput = Ignore; }
get_TP {out "TP?"; in "TP %f"; ExtraInput = Ignore; }
get_H {out "H?"; in "H %f"; ExtraInput = Ignore; }
get_TH {out "TH?"; in "TH %f"; ExtraInput = Ignore; }

```

It follows the already-known pattern of sending a query with a question mark appended, and then receives the query, followed by the measurement value. To accompany this file, which orchestrates the low-level communication, we need a data base file, named `weather.db`, with the following contents.

```

# ./tempApp/Db/weather.db
record(ai, "$(USER):T") {
    field(DESC, "Temperature")
    field(SCAN, "10 second")
    field(DTYP, "stream")
    field(INP, "@weather.proto get_T $(PORT)")
}
record(ai, "$(USER):P") {
    field(DESC, "Pressure")
    field(SCAN, "10 second")
    field(DTYP, "stream")
    field(INP, "@weather.proto get_P $(PORT)")
}
record(ai, "$(USER):TP") {
    field(DESC, "TemperaturePressure")
}

```

```

    field(SCAN, "10 second")
    field(DTYP, "stream")
    field(INP, "@weather.proto get_TP $(PORT)")
}
record(ai, "$(USER):H") {
    field(DESC, "Humidity")
    field(SCAN, "10 second")
    field(DTYP, "stream")
    field(INP, "@weather.proto get_H $(PORT)")
}
record(ai, "$(USER):TH") {
    field(DESC, "TemperatureHumidity")
    field(SCAN, "10 second")
    field(DTYP, "stream")
    field(INP, "@weather.proto get_TH $(PORT)")
}

```

Here we find analog input `ai` records that define the process variables `$(USER):XX`, and link them to the respective protocol functions. Moreover, we need to include the new database file in the `Makefile` by adding the line

```
DB += weather.db
```

and add the following two lines to the `./temp/iocBoot/ioctemp/st.cmd` EPICS command file

```

drvAsynIPPortConfigure("SOCKET","192.168.20.144:1137",0,0,0)
dbLoadRecords("db/weather.db", "PORT='SOCKET',USER='weather'")

```

and finally, call `make` in the base directory of the `temp` hierarchy. Then we execute `st.cmd` as before, either from the command line or automatically with an init script as discussed at the end of Section 6.5. The process variables are now accessible from any computer on the network. To access the barometric pressure, we issue `caget weather:P`, and similarly for the other variables. In this way, they can be seamlessly included in logging, monitoring, or other display programs that EPICS provides. Adding more sensors is as simple as adding protocol and database files as well as including them in the `Makefile` in the `Db` directory, and adding two lines in the `st.cmd` program.

QUESTIONS AND PROJECT IDEAS

1. What is the average barometric pressure at sea level, on a rainy day, in a heavy storm, in a hurricane, in La Paz?
2. What is the typical humidity where you live, in a rain forest, in Death Valley, or at the La Silla Observatory in Chile?
3. Add a dust sensor to turn the weather station into an environmental logging station.
4. Add an LDR or a phototransistor to the weather station to log the brightness.
5. Add a remote-controlled fan that stirs the air upon request.
6. Rewrite the client program for the weather station running on the NodeMCU and give it an MQTT interface.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Example: Geophones

In the second example, we use the SM-24 geophone from Figure 2.12 to record ground vibration spectra. Our plan is to attach the SM-24 to a battery-powered microcontroller that samples 1024 values at a high rate and transmits the measurements via WLAN to the host computer, our Raspi. There we use octave to postprocess and Fourier-transform the samples, as well as present the results. Besides using octave, we also make the data available to EPICS, such that the standard EPICS programs for display and post-processing can be used.

For the sensor node, we use the NodeMCU microcontroller and the `Ticker.h` library that allows sampling with a rate of 1000 times per second. Since the frequency range of the SM-24 sensor is from 10 to 240 Hz, the sampling rate of 1 kHz is four times the maximum frequency and should be adequate. The SM-24 sensor only produces a very small and bipolar output voltage in the mV range. We therefore need to amplify the voltage in order to match the input voltage range of 0 to 3.3 V of the ADC on the NodeMCU.

We base the amplifier on the circuit shown in Figure 2.20, but increase the amplification to $\times 100$ and add a few components to adapt it to our sensor and arrive at the circuit shown in Figure 9.1. The amplification is mostly given by the ratio of R_3 and R_7 to R_1 and R_2 . We also add a 1.5 k Ω resistor R_9 across the input terminals in order to damp the resonant peak at 10 Hz, which the SM-24 sensor exhibits according to the datasheet. If we consider the internal coil resistance (375 Ω) of the sensor, mounting an additional 2.2 μ F capacitor across the input terminals creates a low-pass filter with a cutoff frequency of around 200 Hz. As discussed in Section 2.2.4, this avoids aliasing of higher frequencies into the digitizing bandwidth from 0 Hz to the Nyquist frequency of 500 Hz. The other components have the same functionality as discussed in Section 2.2.2.

After the signal from the geophone is amplified, we use the ADC on the NodeMCU microcontroller and collect 1024 samples: one sample every millisecond, and then pass the digitized samples to the host computer via WLAN. The code that achieves this is the following:

```
// Minimal time-series-server, V. Ziemann, 170324
const char* ssid = "messnetz";
const char* password = "zxcvZXCv";
const int port = 1137;
#include <ESP8266WiFi.h>
WiFiServer server(port);
const uint16_t npts=1024; // number of samples
const int sample_period=1; // ms
```

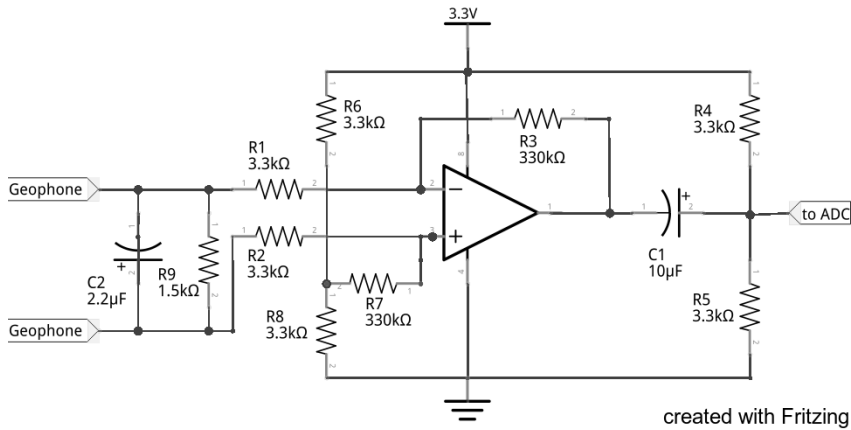


Figure 9.1 The amplifier to interface an SM-24 geophone to the NodeMCU.

```
#include <Ticker.h>
Ticker SampleFast;
uint16_t sample_buffer[npts];
volatile uint16_t isamp=0,sample_buffer_ready=0;
char line[30];
void samplefast_action() { //.....samplefast_action
    sample_buffer[isamp]=analogRead(0);
    isamp++;
    if (npts == isamp) {
        SampleFast.detach();
        isamp=0;
        sample_buffer_ready=1;
    }
}
void setup() { //.....setup
    pinMode(LED_BUILTIN,OUTPUT);
    digitalWrite(LED_BUILTIN,LOW);
    Serial.begin(115200);
    WiFi.begin(ssid,password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500); Serial.print(".");
    }
    Serial.println("");
    Serial.print("Wifi connected to "); Serial.println(ssid);
    Serial.print("Server IP address: "); Serial.println(WiFi.localIP());
    server.begin();
    Serial.print("Server started on port "); Serial.println(port);
    digitalWrite(LED_BUILTIN,HIGH);
}
void loop() { //.....loop
    WiFiClient client = server.available();
    while (client) {
```

```

while (!client.available()) {
    delay(5);
    if (!client.connected()) break;
}
client.readStringUntil('\n').toCharArray(line,30);
Serial.print("Received: "); Serial.println(line);
if (strstr(line,"WF?")==line) {
    digitalWrite(LED_BUILTIN,LOW);
    sample_buffer_ready=0;
    SampleFast.attach_ms(sample_period,samplefast_action);
    while (!sample_buffer_ready) {delay(2);} // wait until done
    for (int i=0;i<npts-1;i++) {
        client.print(sample_buffer[i]); client.print(", ");
    }
    client.println(sample_buffer[npts-1]);
    digitalWrite(LED_BUILTIN,HIGH);
} else {
    Serial.println("unknown command, disconnecting");
    client.stop();
}
client.flush();
}
yield();
}

```

At the top of the code, we first enter the credentials and port number for the WLAN, and then import support to create the `server()` in the next line. The constants `npts` and `sample_period` specify the number of samples to acquire and the number of milliseconds to wait between acquisitions, respectively. The `Ticker.h` library adds support for timed and interrupt-driven functions. The variables declared next are related to filling the `sample_buffer`. The function `samplefast_action()` is called automatically by the timer and is executed once a millisecond. In this function we first do an analog conversion and place the value into the `sample_buffer` before incrementing the variable `isamp`, such that the next sample ends up in the following slot in the array. Once the desired number `npts` of samples are acquired, we disable the interrupt with the call to `SampleFast.detach()`, set the sample pointer `isamp` to zero, and set the `sample_buffer_ready` flag to signal the main program that the acquisition of the `npts` samples is complete. In the `setup()` function, we first set the mode of the pin with the built-in LED and turn it on before enabling the serial line for debugging, and connect to the WLAN. Then we start the server process with the call to `server.begin()` and report everything to the serial line, before turning the LED off.

The organization of the `loop()` function resembles earlier examples. We wait for a client to connect, and then parse the request. If the query-string is `WF?`, we turn on the LED and ensure the variable `sample_buffer_ready` is zero before we start the acquisition with the call to `SampleFast.attach_ms()`. The arguments of the function are the periodicity in ms to call the function specified as the second argument. This launches the automatic acquisition that continues in the background. All we need to do in the `loop()` function is to monitor whether `sample_buffer_ready` is still zero, in which case we wait a bit longer. But once it becomes nonzero, which happens in the `samplefast_action()` function after the desired number of samples is collected, we break the waiting loop. Finally, we are ready

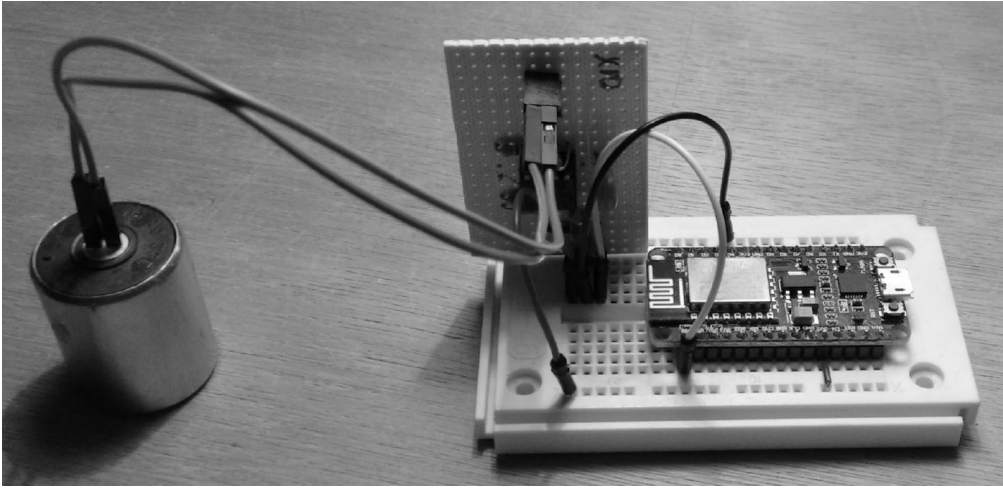


Figure 9.2 The prototype with the SM-24 geophone connected to the amplifier board and the NodeMCU.

to send the values, separated by commas, with `client.print()` function calls, back to the client and turn the LED off.

We program this sketch into the NodeMCU, whence it waits for a client to connect and to request samples. Here we use `octave` to request a time series of samples from the geophone and display both the received raw time-series data and its Fourier transform, the spectrum. The code that achieves this is the following:

```
% getTimeSeries.m, V. Ziemann, 170324
dev=tcp("192.168.20.144",1137);
npts=1024;
tcp_write(dev,"WF?\n");
sleep(1);
data=zeros(1,npts);
for i=1:npts
    data(i)=str2double(tcp_getvalue(dev));
end
tcp_close(dev);
subplot(2,1,1)
plot(data)
xlim([0,npts])
xlabel('Time [ms]')
ylabel('Amplitude [ADC bits]')
subplot(2,1,2);
data=data-mean(data);
fftdata=2*abs(fft(data))/npts;
frequency=(0:(npts/2-1))*500/(npts/2);
plot(frequency,fftdata(1:npts/2))
xlabel('Frequency [Hz]')
ylabel('Spectral density [ADC bits]')
```

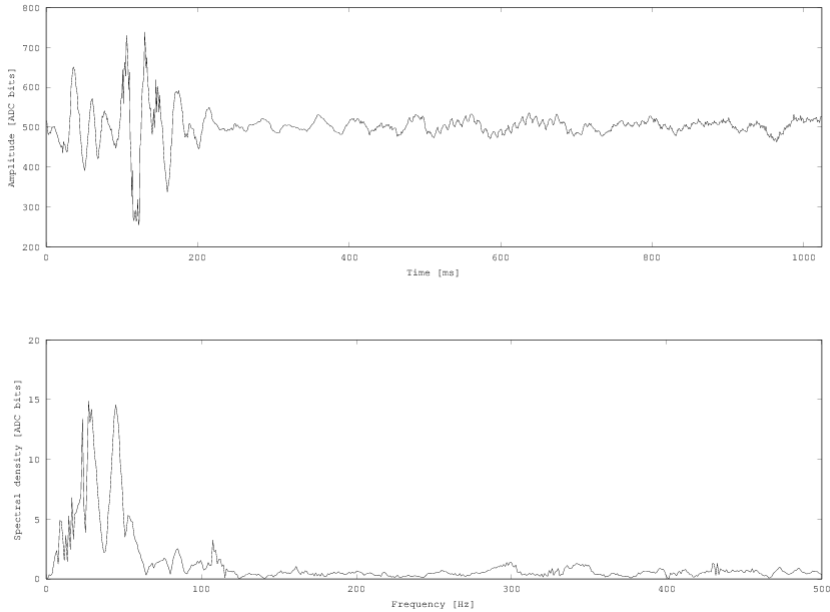


Figure 9.3 The raw time series from the geophone and the corresponding spectrum.

```
print('spectrum.png', '-S1000,700');
```

Also, the octave script follows earlier examples. We first define the socket to connect to in the `tcp()` function call and then specify the number of points to acquire. This has to match the number of samples declared on the NodeMCU. Next, we send the query string `WF?` to the socket, wait a while, and then collect `npts` samples, which we store in the array `data()`. For this we use the function `tcp_getvalue()` that extracts one comma-separated value from the socket. We discuss that function in more detail below. The received value is encoded as a character string, and we therefore need to convert it to a float value with the `str2double()` function. Once all samples are received and stored in `data()`, we close the socket.

In the following lines, we create two subplots of which the upper contains the raw samples. Since the sample period is a millisecond, we state that as the horizontal axis label. The vertical axis is just the raw ADC conversion, which comes from the 10-bit ADC on the NodeMCU, and lies between 0 and 1023. For the spectrum that we show in the lower subplot, we first subtract the mean of the values in order to avoid a huge spectral peak at zero. It is caused by placing the signal in mid-range of the ADC with the preamplifier. Then we Fourier transform the samples and create an array `frequency` with the frequency values from zero to the Nyquist frequency before plotting the spectrum and labeling the axes. Finally, we use the `print()` function to create an image file that contains the displayed plot with the specified size in pixels, 1000 × 700 in this case.

In the octave script, we use the function `tcp_getvalue` to read a single sample from the input stream. The function is similar to the `queryResponse.m` function we used in Section 5.5.4. Here is the octave code.


```
% tcp_getvalue.m, V. Ziemann, 170324
function out=tcp_getvalue(dev)
i=1;
int_array=uint8(1);
while true
    val=tcp_read(dev,1);
    if ((val==',' ) || (val==0xA)) break; end
    int_array(i)=val;
    i=i+1;
end
out=char(int_array);
```

We need to provide the handle `dev` to the socket as input parameter, and the function returns a character string that contains all characters until either a comma or the next end-of-line character. In the function, we read one character at a time from the socket, and stop reading once the comma or end-of-line character `0xA` appears. Then we return all received data after conversion to characters.

We show the sensor node with SM-24 geophone, amplifier, and NodeMCU in Figure 9.2. Running the octave script with the NodeMCU connected to the same WLAN as the Raspi produces a file `spectrum.png`, shown in Figure 9.3. Running the octave script again will overwrite the file, and tapping with a finger on the table while recording samples results in a much more noisy spectrum.

Recording waveforms in octave and processing them further is convenient to produce plots of the spectra, but if we want to include the geophones in our EPICS control system, we need to provide the database and prototype files. We start with the database record for the time series or waveform. It is reproduced below,

```
# .../Db/geophone.db
record(waveform, "$(USER):wf") {
    field(DESC,"Geophone waveform")
    field(DTYP,"stream")
    field(SCAN,"10 second")
    field(NELM,"1024")
    field(FTVL,"FLOAT")
    field(INP,"@geophone.proto get_wf $(PORT)")
}
```

where we have to declare a `waveform` record, because we want to acquire the entire stream of 1024 samples in one step. The first three fields provide the description, the declaration as a `stream` device, and the rate at which the waveforms are collected. The next two fields declare that we record 1024 float values, before linking to the protocol file `geophone.proto` and the function `get_wf` in the last field of type `INP`. The referenced protocol file `geophone.proto` is reproduced below.

```
# .../Db/geophone.proto
Terminator = CR LF;
get_wf {
    ExtraInput = Ignore;
    replyTimeout=2000;
    out "WF?";
    separator=",";
```

```
    in "%i";
}
```

Here we first define the termination string of one command as `CR LF`, and then define the `get_wf` function. For robustness, we require it to ignore any input that does not make sense, and wait 2000 ms for the reply from the NodeMCU. The request for a new waveform consists of sending `WF?` and receiving integers ("`%i`"), separated by commas. The number of values to receive from the NodeMCU, 1024, is already specified in the `geophone.db` file.

In a final step, we need to tell EPICS where to find the NodeMCU and what socket to connect to. This is done, as discussed in Chapter 6 on EPICS, in the `st.cmd` command file, which contains the following two lines:

```
drvAsynIPPortConfigure("SOCKET2","192.168.20.144:1137",0,0,0)
dbLoadRecords("db/geophone.db","PORT='SOCKET2',USER='geophone'")
```

The first defines the IP and port number of the connected NodeMCU device, and the second line specifies the database file `geophone.db` that describes the protocol that is used when communicating with the NodeMCU. After adding `DB += geophone.db` to the Makefile, we recompile the EPICS IOC and then execute `st.cmd` from the command line, or, once we make sure that everything works as expected, we create an init file to start the IOC at boot time, following the procedure we discussed in Section 6.4. At this point, with a running IOC continuously retrieving waveforms from the NodeMCU, we can obtain the most recent waveform with `caget geophone:wf`, which first returns the number of samples, here 1024, followed by the 1024 samples. Other EPICS programs, on any computer connected to the same network, can also obtain the same waveforms, a new one every 10 seconds, as specified in the `geophone.db` record file.

QUESTIONS AND PROJECT IDEAS

1. Discuss the advantage of using interrupt-driven acquisition over measuring 1000 samples in a loop with a `delay(1)` between acquisitions.
2. What is the purpose of the low-pass filter discussed in this chapter?
3. What is the maximum frequency we can determine uniquely when sampling at a rate of 1000 samples per second?
4. What is the resolution (smallest detectable difference of frequencies) of the spectrum shown in Figure 9.3? How can you increase it?
5. Discuss why the sample rate of the `Ticker.h` library is limited to 1000 samples per second.
6. Use a pin diode as sensor and sample it at a rate of 1000 samples per second. Explore different flickering light sources such as lamps, TV screens, or computer monitors.
7. Use the `tone()` function to generate an oscillating signal on an output pin. Then sample that pin with the built-in ADC and observe the signal. What is the highest frequency you can uniquely observe? What happens when you go beyond that frequency?
8. Instead of the geophone, use a microphone and “observe” your voice.

9. Add an MCP3304 ADC to the NodeMCU and connect two geophones, fill two `sample_buffer` arrays with data in the interrupt handler, and send both waveforms to octave. If the geophones are placed on a large table far apart, try out to determine the speed of sound by tapping on the table, and determine the shift in time between the two waveforms.

Example: Monitor for the Color of Water

In this example, we determine the color of water, or more accurately stated, the absorption of material dissolved in water. An example is algae, which often proliferate during the summer and turn the water green because algae absorb predominantly red light. In our experiment, we investigate the absorption of light of different colors by turning a three-color LED on and off, and observe the resulting modulation of the signal recorded by a phototransistor. If red light is absorbed, the resulting modulation depth of red is reduced, likewise for the other colors. By comparing the situation with LED on versus off, the system removes some of the influence of ambient background light, at least to some extent. On the left in Figure 10.1, we illustrate the absorption measurement with the RGB-LED on the left and the phototransistor on the right, with the absorbing material in between. We note that we can also use such a system to determine variations in the reflection from a surface, as is shown on the right in Figure 10.1.

We construct the setup shown in Figure 10.1, using an Arduino UNO, and an RGB-LED, which houses a red, a green, and a blue LED in the same housing with a single connector for a common cathode or anode. In our case we use one with a common anode that is connected to the positive supply voltage. The respective color lights up if the controlling pin on the UNO is LOW. The phototransistor is an SFH3310, but any other model, sensitive in the visual spectral range, should work. The very simple setup is shown in Figure 10.2.

On the left we see the Arduino UNO and on the bottom right a small breadboard

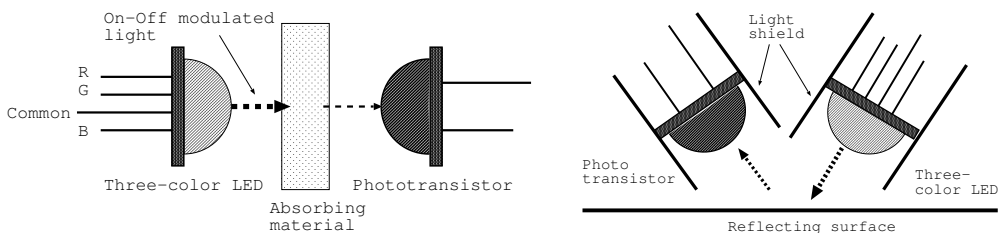


Figure 10.1 The setup to measure the color-dependent absorption (left) and the reflection from a surface (right).

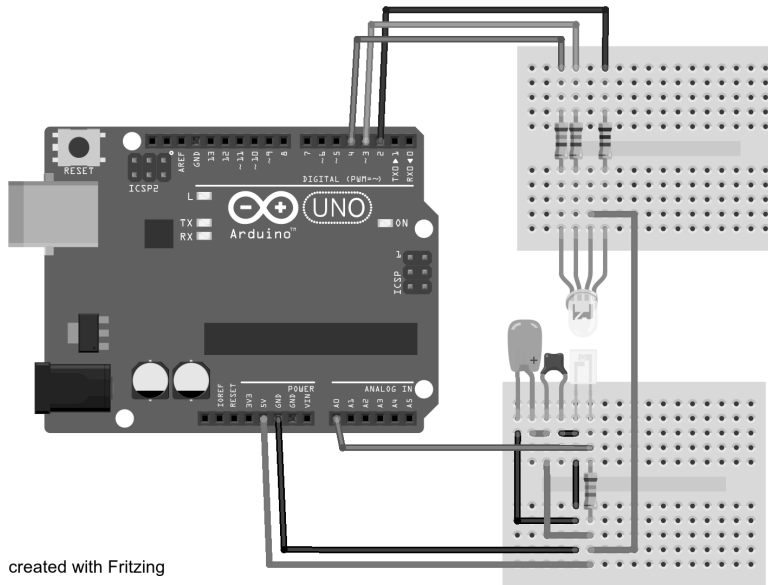


Figure 10.2 The setup to measure the water color with an Arduino UNO.

with the SFH3310 phototransistor. The emitter is connected to ground and the collector is connected via a $68\text{ k}\Omega$ resistor to the positive supply voltage. The resistor depends somewhat on the phototransistor and is determined by the requirement for a clearly visible modulation depth on the analog input pin, which is connected to the collector with the blue wire. A little experimenting will result in a reasonable value for the resistance. We add a $10\text{ }\mu\text{F}$ and a 100 nF capacitor to buffer the supply voltage. The RGD-LED is placed on the upper breadboard with the common anode connected to the positive supply voltage, and the three “color pins” via current-limiting resistors to pins D2, D3, and D4 on the UNO. We chose $220\text{ }\Omega$ for the red LED, $180\text{ }\Omega$ for the green, and $150\text{ }\Omega$ for the blue in order to account for the different voltage drops of the different-color LEDs.

In the sketch that runs on the UNO, we need to toggle the LEDs one at a time and synchronously record the intensities recorded by the phototransistor. This is achieved by the following code.

```
// Color monitor, V. Ziemann,170823
int repeat=20;
void alloff() { //.....alloff
    digitalWrite(2,HIGH); // red
    digitalWrite(3,HIGH); // green
    digitalWrite(4,HIGH); // blue
}
float measure(int pin, int repeat) { //.....measure
    int hi,lo;
    float sum=0;
    alloff();
    delay(10);
    for (int i=0;i<repeat;i++) {
```

```

    digitalWrite(pin,LOW);
    delay(5);
    lo=analogRead(0);
    digitalWrite(pin,HIGH);
    delay(5);
    hi=analogRead(0);
    sum+=(hi-lo);
}
return sum/repeat;
}
void setup() { //.....setup
    Serial.begin(9600); while (!Serial) {}
    pinMode(2,OUTPUT); // red
    pinMode(3,OUTPUT); // green
    pinMode(4,OUTPUT); // blue
    alloff();
    digitalWrite(2,LOW);
}
void loop() { //.....loop
    if (Serial.available()) {
        char line[30];
        Serial.readStringUntil('\n').toCharArray(line,30);
        if (strstr(line,"OFF")==line) {
            alloff();
        } else if (strstr(line,"RED")==line) {
            alloff();
            digitalWrite(2,LOW);
        } else if (strstr(line,"GREEN")==line) {
            alloff();
            digitalWrite(3,LOW);
        } else if (strstr(line,"BLUE")==line) {
            alloff();
            digitalWrite(4,LOW);
        } else if (strstr(line,"COLOR?")==line) {
            float red=measure(2,repeat); // red
            float green=measure(3,repeat); // green
            float blue=measure(4,repeat); // blue
            Serial.print("COLOR "); Serial.print(red);
            Serial.print("\t");Serial.print(green);
            Serial.print("\t");Serial.println(blue);
        } else if (strstr(line,"REPEAT?")==line) {
            Serial.print("REPEAT "); Serial.println(repeat);
        } else if (strstr(line,"REPEAT ")==line) {
            repeat=(int)atof(&line[6]);
        }
    }
    delay(10);
}

```

First we declare a variable `repeat` and a function to turn all the LEDs off. The input

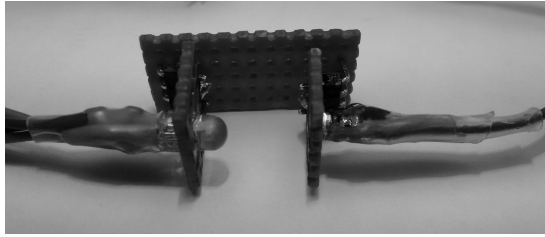


Figure 10.3 The color sensor with the RGB-LED on the left and the phototransistor on the right.

parameters of the function `measure()` are the pin to be toggled and the number of times to repeat the measurement. We add a `delay` of 5ms in order for the voltage to stabilize after the LED is switched on or off, before measuring the voltage at the collector of the phototransistor with the `analogRead()` function. The difference of the reading with LED on and off is accumulated in the variable `sum`, and finally the average of the accumulated differences is returned to the calling program. In the `setup()` function, we initialize the serial communication and the pins for the RGB-LED, turn all LEDs off, and then turn the red LED on in order to indicate that the system is up and running. The `loop()` function uses the same mechanism we use throughout this book to turn all LEDs OFF or turn one of the colors RED, GREEN, or BLUE on. If the query is COLOR? the system measures the variation on the phototransistor when toggling the LEDs on and off, and returns three numbers for the three color-variations. Finally, the query REPEAT `nnn` sets the variable `repeat` to `nnn` and REPEAT? returns the current value.

This system measures the absorption of any material placed between the LED and the phototransistor, which are pointing at each other as shown in Figure 10.1, and in Figure 10.3 where we show a simple prototype. Moreover, in order to measure the absorption in water, we need to encapsulate the LED and transistor in a protective coating, for which we use two layers of heat-shrink insulation material to cover the solder connection of wires to the pins of the LED and phototransistor.

If we want to deploy the system at a somewhat remote location, it is rather straightforward to extend the MQTT client program from Chapter 7 to monitor the color and publish the results once every minute. For this purpose, we use a NodeMCU instead of a UNO, and connect the RGB-LED to pins D2, D3, D4 on the NodeMCU, and the collector of the phototransistor to the analog input pin A0. The following sketch implements this functionality.

```
// MQTT client water color, V. Ziemann, 170906
const char* ssid = "messnetz";
const char* password = ".....";
const char* broker = "192.168.20.1";
#include <Ticker.h>
volatile uint8_t do_something=0;
Ticker tick;
void tick_action() {do_something=1;} // executed regularly
#include <ESP8266WiFi.h>
#include <PubSubClient.h>
WiFiClient espClient;
```

```

PubSubClient client(espClient);
float measure(int pin, int repeat) { //.....measure
    int hi,lo;
    float sum=0;
    digitalWrite(D2,HIGH);
    digitalWrite(D3,HIGH);
    digitalWrite(D4,HIGH);
    delay(10);
    for (int i=0;i<repeat;i++) {
        digitalWrite(pin,LOW);
        delay(5);
        lo=analogRead(A0);
        digitalWrite(pin,HIGH);
        delay(5);
        hi=analogRead(A0);
        sum+=(hi-lo);
    }
    return sum/repeat;
}

void setup() { //.....setup
    pinMode(D2,OUTPUT);
    pinMode(D3,OUTPUT);
    pinMode(D4,OUTPUT);
    Serial.begin(115200);
    WiFi.begin(ssid,password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500); Serial.print(".");
    }
    Serial.print("\nWifi connected to "); Serial.println(ssid);
    Serial.print("with IP address: "); Serial.println(WiFi.localIP());
    client.setServer(broker, 1883); // 1883 = default MQTT port
    tick.attach(60,tick_action); // execute tick_action every 60 seconds
}

void loop() { //.....loop
    while (!client.connected()) { // try to connect to broker
        if (!client.connect("PubSub1")) {delay(5000);}
    }
    client.loop();
    if (do_something) {
        do_something=0;
        char message[30];
        float red=measure(D2,20); // red
        float green=measure(D3,20); // green
        float blue=measure(D4,20); // blue
        sprintf(message,"%d %d %d",(int)red,(int)green,(int)blue);
        client.publish("node1/color",message);
    }
    yield();
}

```


We first declare the network credentials and the IP address of the broker before including the `Ticker.h` library. It orchestrates the repeated measurements by calling the `tick_action()` function at well-defined time-intervals. Inside the function we only set the variable `do_something` to one, because this variable is monitored in the `loop()` function and triggers some activity, if it is set. Next, we include the WiFi libraries, and declare a `WiFiClient` and the MQTT `PubSubClient` named `client`, before defining the `measure()` function, which is a straight copy from the previous example. The `setup()` function declares the mode of the pins, configures the serial line, connects to WLAN, configures the broker, and starts the ticker to call `tick_action` every 60 seconds, which triggers the measurements. In the `loop()` function, we first ensure that the NodeMCU is connected to the broker before calling `client.loop()` to service any background activities pertaining to MQTT. Then we test whether the variables `do_something` is set, and execute the measurement with the `measure()` function for each color, construct a string that contains the three values, and publish the measurement.

Using this system, it is possible to submerge a sensor similar to the one shown in Figure 10.3 in a garden pond, place the NodeMCU in an enclosure next to the pond, possibly battery powered, and make sure it is within reach of the WLAN `messnetz`. That is all that is needed to monitor algae in the pond.

QUESTIONS AND PROJECT IDEAS

1. How do the reported color-variations change if you remove the capacitors from the small breadboard? Explain why!
2. How do the reported color-variations change if the waiting time of 5 ms in the `measure()` function is reduced to 1 ms? What happens if it is increased?
3. Connect the RGB-LED to output pins with pulse-width modulation capability and write a sketch that allows you to set a number of standard colors, such as orange or magenta.
4. Write a sketch to set the color of the LEDs via MQTT.
5. Build a reflectivity sensor similar to the one shown on the right in Figure 10.1, and connect it to the Arduino UNO.

Example: Capacitance Measurement

In this chapter, we measure the unknown capacitance of a capacitor by first charging it, then turning off the charging supply and discharging through a parallel resistor, while repeatedly measuring the voltage drop across the capacitor. The voltage drops exponentially $U \propto e^{-t/\tau}$ with time constant $\tau = RC$, where R is the discharging resistor and C the unknown capacitance. Thus, from a linear least-squares fit to the logarithm of the voltage, we can determine the time constant, and from the known resistor value, also the unknown capacitance.

Figure 11.1 shows the experimental setup. The Arduino UNO is on the right and the small breadboard showing the capacitor with the $R = 33\text{ k}\Omega$ resistor to discharge the capacitor connected in parallel; both are connected to ground and analog pin A0 on the Arduino. The capacitor can be charged by pulling digital pin D2 high; it is connected by a wire and a $220\text{ }\Omega$ resistor. The latter resistor limits the initially flowing current, when the capacitor is fully discharged.

The task of the UNO is to first charge the capacitor, by configuring the digital output pin D2 as **OUTPUT** and setting its value to **HIGH**. Once a measurement is requested, the D2 is reconfigured as input with `pinMode(2,INPUT)`, and setting the output nevertheless to **LOW**, to ensure that the internal pull-up resistors are *not used*. Once the charging voltage is disconnected, we repeatedly read analog pin A0 from an interrupt service routine until the required number of samples is collected in memory. Once the raw data are collected, we make a linear least-squares fit to the data and determine the capacitance from the slope. This plan of action is realized in the following Arduino sketch.

```
// Capacitance measurement, V. Ziemann, 170629
const int npts=100;
volatile int isamp=0,sample_buffer_ready=0;
uint16_t sample_buffer[npts],nsamp=npts,timestep=5;
float R=33e3; // 33 kOhm
#include <MsTimer2.h>
void timer_action() { //.....timer_action
    sample_buffer[isamp]=analogRead(A0);
    isamp++;
    if (nsamp == isamp) {
        MsTimer2::stop();
```

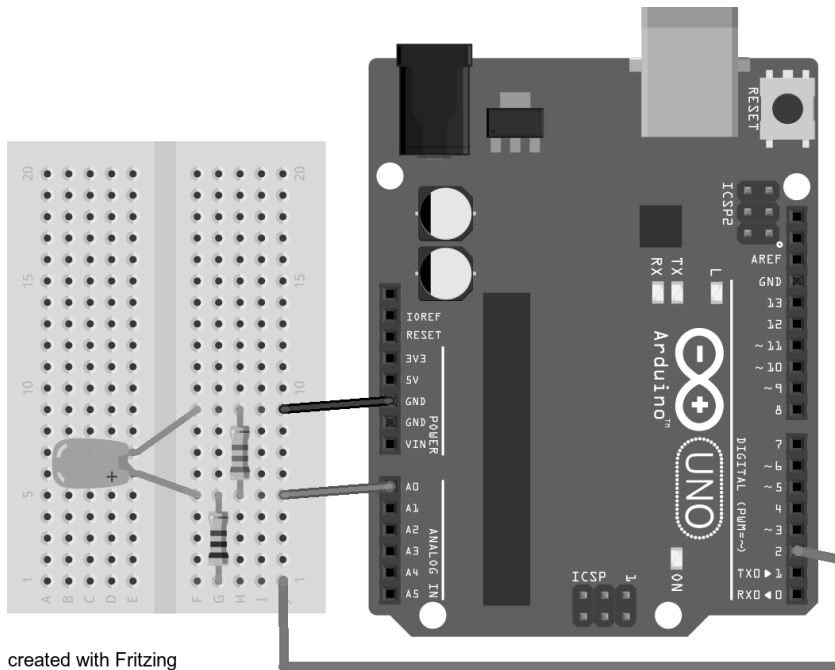


Figure 11.1 The setup to measure the capacitance.

```

    isamp=0;
    sample_buffer_ready=1;
}
}
double linfit(int n, uint16_t y[]) { //.....linfit
    double ay0=0,ay1=0;
    double S0=n;
    double S1=0.5*n*(n+1);
    double S2=n*(n+1.0)*(2.0*n+1)/6.0;
    for (int k=0;k<n;k++) {
        ay0+=k*log(y[k]);
        ay1+=log(y[k]);
    }
    return (S0*ay0-S1*ay1)/(S2*S0-S1*S1);
}
void setup() { //.....setup
    Serial.begin(9600);
    while (!Serial) {delay(10);}
    pinMode(2,OUTPUT);
    digitalWrite(2,HIGH);
}
void loop() { //.....loop
    if (Serial.available()) {
        char line[30];

```

```

Serial.readStringUntil('\n').toCharArray(line,30);
if (strstr(line,"CAP?")) {
    nsamp=(int)atof(&line[5]);
    nsamp=min(nsamp,npts);
    if (nsamp==0) nsamp=npts;
    pinMode(2,INPUT);
    digitalWrite(2,LOW); // disables internal pullup
    MsTimer2::set(timestep, timer_action);
    MsTimer2::start();
    sample_buffer[isamp]=analogRead(A0);
    isamp++;
} else if (strstr(line,"WF?")) {
    Serial.print("WF "); Serial.println(nsamp);
    for (int i=0; i<nsamp; i++) Serial.println(sample_buffer[i]);
} else if (strstr(line,"TIMESTEP ")) {
    timestep=(int)atof(&line[9]);
    Serial.print("TIMESTEP "); Serial.println(timestep);
} else if (strstr(line,"TIMESTEP?")) {
    Serial.print("TIMESTEP "); Serial.println(timestep);
} else if (strstr(line,"RESISTOR ")) {
    R=atof(&line[9]);
    Serial.print("RESISTOR "); Serial.println(R);
} else if (strstr(line,"RESISTOR?")) {
    Serial.print("RESISTOR "); Serial.println(R);
}
}
}
if (sample_buffer_ready==1) {
    sample_buffer_ready=0;
    pinMode(2,OUTPUT); // start charging capacitor
    digitalWrite(2,HIGH);
    delay(100);
    double slope=linfit(nsamp,(uint16_t)sample_buffer);
    double capacitance=-1e6*timestep*1e-3/(slope*R); // in uF
    Serial.print("CAP "); Serial.println(capacitance,4);
    if (sample_buffer[0] < 250*sample_buffer[nsamp-1]) {
        Serial.println("***Time too short, double TIMESTEP");
    }
}
}
delay(1);
}

```

The sketch follows the usual format, where we first declare a number of variables in which `npts` is the maximum number of points that can be acquired. The data samples are stored in the `sample_buffer` once every `timestep` milliseconds. We also declare the resistance `R` of the resistor used to discharge the capacitor. Then we include the `MsTimer2.h` header file and library, which provides the functionality to repeatedly call a function without user intervention. We use `MsTimer2.h` because the `Ticker.h` library we used on the NodeMCU in the previous chapter is unavailable for the UNO. The `MsTimer2.h` listens to an internal clock and then calls the function after the specified time interval has elapsed. The function it calls is normally referred to as the interrupt-service routine, and it is called `timer_action()`

in our sketch. This function is called every `timestep` milliseconds, and first reads analog pin A0, then stores the value in the `sample_buffer` and increments `isamp`. If `isamp` reaches `nsamp`, the timer is stopped to prevent the acquisition of further data points. Then `isamp` is set to zero to prepare for the next acquisition and the variable `sample_buffer_ready` is set to 1 to indicate that an acquisition is complete and data is ready for further processing. Technically we can process the data within the `timer_action` function, but normally this routine should contain only time-critical actions, such as acquiring samples, and should be kept as compact as possible, because it is called asynchronously to all other processing and we should avoid excessive disturbances. The way to handle this problem is to flag that the acquisition is ready, and in the main program, check for this flag and perform the postprocessing. In this way, the time-critical and the slower processes are efficiently decoupled. The `linfit()` function takes the acquired data as well as the number of data points as input parameters, internally performs the least-squares fit to the logarithm of the data points, and returns the slope of the fit. The discussion of the algorithm is somewhat technical, and is deferred to Appendix B.

The `setup()` function initializes the serial line, declares digital pin D2 as output, and sets it to HIGH to start charging the capacitor. The `loop()` function uses the standard query-response protocol, and a capacitance measurement starts with the command `CAP? nnn`, where `nnn` is the desired number of measurement points; per default the maximum number `npts` is acquired. Then digital pin D2 is placed in high-impedance input mode. We also ensure that the internal pull-up resistor is disabled, and start the timer process with the `Mstimer::set()` function. It takes the time between samples and the function to call as argument, and then we can start the timer. Before leaving this subroutine, we take the first data sample. The command `WF?` can be used to retrieve the waveform data of the last acquisition. This is useful for crosschecking the fitting in, for example, octave. The remaining commands are used to set and read the `TIMESTEP` or the `RESISTOR` values. Once the handling of the commands from the serial line is complete, we test whether the variable `sample_buffer_ready` is set, which indicates that an acquisition is complete and we can start postprocessing the raw data. First we reset the `sample_buffer_ready` to zero, to prevent repeated calls, and then start charging the capacitor again by configuring pin D2 as output, and wait a little while. Next we calculate the `slope` from the samples. Since the slope is inversely proportional to the time constant $\tau = RC$, we can solve this for the capacitance C with the equation defining the `capacitance`. The factor `1e6` causes the displayed value to be shown in μF . Finally, we check whether the exponential decay of the voltage actually extends over a sufficiently long time, and display a warning if that is not the case.

We can measure capacitances by sending commands from other programs that communicate over the serial line. From octave we use the following program to initiate a measurement, and plot the data as shown in Figure 11.2.

```
% capacitance_plot.m, V. Ziemann, 170630
% pkg load instrument-control
close all; clear all
s=serial('/dev/ttyACM0',9600);
sleep(2);
srl_flush(s);
srl_write(s,"CAP? 100");
sleep(1);
reply=serialReadline(s);
capacitance=str2double(reply(4:end));
```

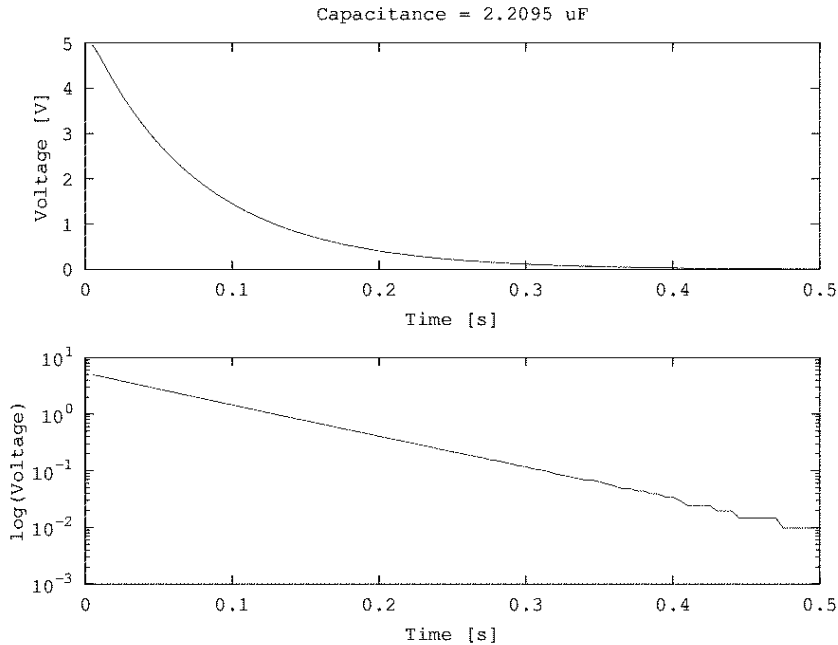


Figure 11.2 The waveform of the voltage on the capacitor as a function of time, while a $2.2\text{ }\mu\text{F}$ Tantal capacitor discharges through a $33\text{ k}\Omega$ resistor on a linear scale (above) and on a logarithmic scale (below).

```

srl_write(s,"TIMESTEP?");
reply=serialReadline(s);
if reply(1:8)=="TIMESTEP"
    timestep=str2double(reply(9:end));
else
    disp(reply)
    close(s);
    return
end
srl_write(s,"WF?");
reply=serialReadline(s);
nsteps=str2num(reply(3:end));
xx=zeros(nsteps,1); yy=xx;
for k=1:nsteps
    xx(k)=k*timestep*1e-3; % in seconds
    yy(k)=str2double(serialReadline(s))*5/1023; % in Volt
end
close(s)
subplot(2,1,1); plot(xx,yy);
xlabel("Time [s]"); ylabel("Voltage [V]");
title(["Capacitance = " num2str(capacitance) " uF"]);
subplot(2,1,2); semilogy(xx,yy);

```

```
xlabel("Time [s]"); ylabel("log(Voltage)");
print('capacitance.png', '-S1000,700')
```

We need to keep in mind that we need to load the instrument-control toolbox in order to use the serial communication functionality and clear the workspace, before actually opening the serial line. First we remove any remaining characters from the input queue, and initiate a measurement with the `CAP?` command. After a short waiting period, we read the reply with the `serialReadline()` function, which is tailored after the `queryResponse()` function from Section 5.5.4. It is reproduced here.

```
% get response up to termination character
function out=serialReadline(dev,term_char)
    if (nargin==1) term_char=10; end % defaults to LF=0x0A
    i=1;
    int_array=uint8(1);
    while true % loop forever
        val=srl_read(dev,1); % and read one byte
        if (val==term_char) break; end % until term_char appears
        int_array(i)=val; % stuff byte in output
        i=i+1;
    end
    out=char(int_array); % convert to characters
```

This function reads characters one at a time until an end-of-line character is found, and then returns the received characters to the calling program, where the numerical value is extracted. For the plot, we need to know the `TIMESTEP` used in the measurement, and request it from the UNO with the `TIMESTEP?` command. If the reply starts with `TIMESTEP`, all is well, and we extract the numerical value. If we retrieve something else from the serial input queue, the time window for the measurements was too short, and the measurement is invalid. In that case, the program closes the serial port and returns control to the octave command prompt. If, on the other hand, we obtain the numerical value of the time step, we request the waveform data with the `WF?` command. The first returned line contains the number of points acquired. The following lines contain the data points. Each new line is retrieved with a call to `serialReadline()` until all data points are copied to the array `yy`. Finally, we plot the curves with linear and logarithmic vertical scale. The latter shows a clear linear dependence, which indicates the exponential dependence of the voltage on time with a well-defined time constant. In the last line, we produce a png file shown in Figure 11.2 that can be used to document the work.

QUESTIONS AND PROJECT IDEAS

1. Measure the *resistance* of an unknown resistor by comparing to a known resistor in a voltage-divider.
2. Ask a friend to build a pulse generator that produces pulses with variable width; for example, a 10 ms pulse every 200 ms. Then use a second UNO to measure the pulse pattern, by sampling an IO-pin using the `MsTimer2.h` library. Display the pulse pattern on the host computer.

Example: Profile of a Laser Beam

In this example we measure the transverse profile of the laser beam from a laser pointer by carefully moving an obstacle across the beam and observing the change of the signal from a photoresistor. Figure 12.1 illustrates the method. We use a laser module from a sensor kit that is mounted on a breadboard and exposes two wires, one for ground and one for the positive supply voltage. In order to adjust the intensity of the laser, we use pulse-width modulation of the positive supply voltage. The sensor is a voltage divider consisting of a light sensitive resistor (LDR) and a $10\text{ k}\Omega$ resistor. The obstacle is made of a piece of black plastic. As the mover, we use the frame salvaged from an old CD-ROM drive, which moves a small wagon with a stepper motor back and forth over a distance of about 40 mm, which is approximately the width of the readable area of a CD. The stepper motor drives a spindle, and that pulls and pushes the wagon with good precision.

Figure 12.2 shows the salvaged frame from the CD drive. The frame has plenty of holes, which are convenient to add screws that in turn are used to attach laser, sensor, and obstacle. The laser on its small breakout board is located at the top right with the stepper motor beneath and hidden from view, but the spindle is clearly visible, running right to left near the top of the frame. There it engages the white plastic part that is attached to the wagon and moves it, when the motor turns. On the wagon there is a black piece of plastic visible that intercepts the laser beam and prevents the laser from hitting the LDR on the lower right on the frame. The LDR is also mounted on a small breakout board that is part of

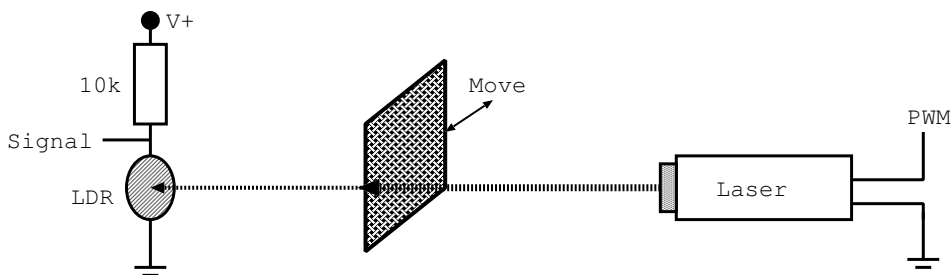


Figure 12.1 The schematic setup to measure the beam size of a laser pointer.

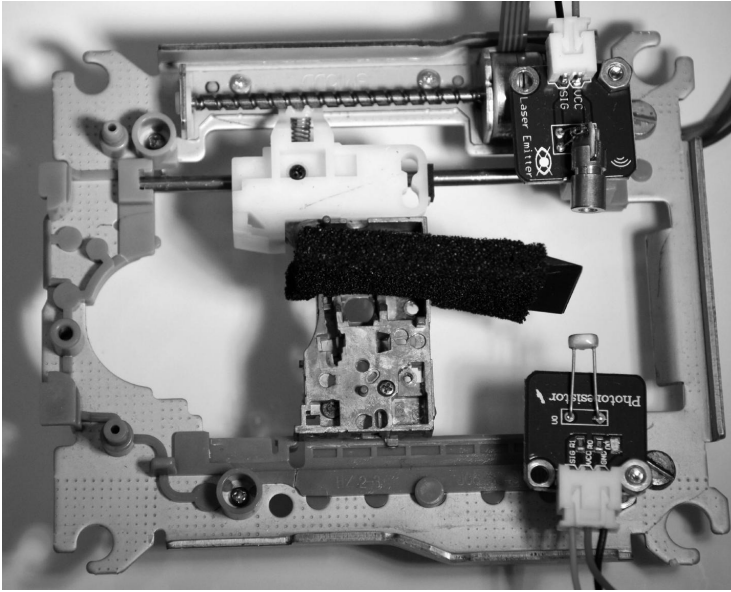


Figure 12.2 The chassis from the CD-ROM drive with the laser mounted on the top right and the photoresistor on the bottom right. The black obstacle is mounted on the carriage that can be moved via the spindle on the top by a small stepper motor that is located below the laser.

a sensor kit, and we use it because the mounting holes make assembly and attaching the LDR to the screws on the frame easy. As controller we use an Arduino UNO that drives the motor with an L293D H-bridge driver, uses pulse-width modulation on pin D9 to control the laser power, and reads the signal from the voltage divider with the LDR on analog input A0. Figure 12.3 shows the schematics. We use an external power supply to provide the voltage for the stepper motor that is connected to the terminals labeled PA,...,PD.

The next task is to program the UNO to control the power of the laser, move the stepper motor, and read the sensor, all in a well-orchestrated fashion. We base the sketch on the program to drive a stepper motor with an H bridge from Section 4.5.4 and augment the code with sections for the laser and the sensor, as shown below.

```
// Laser profile measurement, V. Ziemann, 170628
char line[30];
int settle_time=2, stepcounter=0;
int laser_power=10;
const int PA=2,PB=3,PC=4,PD=5,ENABLE=6;
const int LASER=9;
void set_coils_fullstep(int istep) { //.....set_coils
    bool patA[]={1,1,0,0};
    int pat_length=4;
    int ii;
    istep=istep % pat_length;
    if (istep < 0) istep+=pat_length;
    digitalWrite(PA,patA[istep]);
```

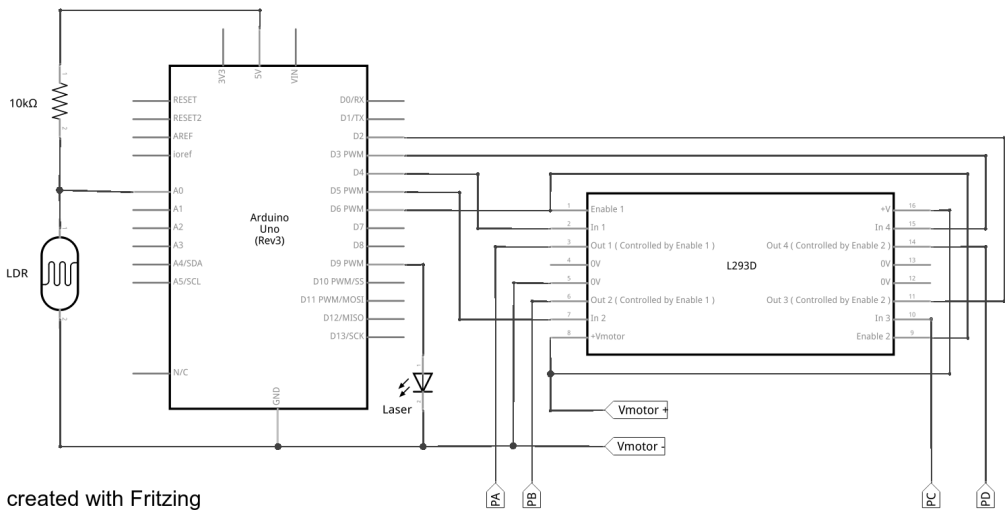


Figure 12.3 The schematics of the circuit. The stepper motor of the frame is connected to points labelled PA,...,PD.

```

    ii=(istep+2) % pat_length;
    digitalWrite(PB,patA[ii]);
    ii=(istep+3) % pat_length;
    digitalWrite(PC,patA[ii]);
    ii=(istep+1) % pat_length;
    digitalWrite(PD,patA[ii]);
    delay(settle_time);
}

void setup() { //.....setup
  Serial.begin (9600);
  while (!Serial) {};}
  pinMode(PA,OUTPUT);
  pinMode(PB,OUTPUT);
  pinMode(PC,OUTPUT);
  pinMode(PD,OUTPUT);
  pinMode(ENABLE,OUTPUT);
  digitalWrite(ENABLE,HIGH);
  analogWrite(LASER,laser_power);
}

void loop() { //.....loop
  if (Serial.available()) {
    Serial.readStringUntil('\n').toCharArray(line,30);
    if (strstr(line,"FMOVE ")==line) {
      int steps=(int)atof(&line[6]);
      digitalWrite(ENABLE,HIGH);
      if (steps > 0) {
        for (int i=0;i<steps;i++) set_coils_fullstep(stepcounter++);
      } else {

```

```

        for (int i=0;i<abs(steps);i++) set_coils_fullstep(stepcounter--);
    }
} else if (strstr(line,"STEPS?")==line) {
    Serial.print("STEPS "); Serial.println(stepcounter);
} else if (strstr(line,"STEPS ")==line) {
    stepcounter=(int)atof(&line[6]);
} else if (strstr(line,"WAIT?")==line) {
    Serial.print("WAIT "); Serial.println(settle_time);
} else if (strstr(line,"WAIT ")==line) {
    settle_time=(int)atof(&line[5]);
} else if (strstr(line,"DISABLE")==line) {
    digitalWrite(ENABLE,LOW);
} else if (strstr(line,"ENABLE")==line) {
    digitalWrite(ENABLE,HIGH);
} else if (strstr(line,"LDR?")==line) {
    Serial.print("LDR "); Serial.println(analogRead(A0));
} else if (strstr(line,"POWER?")==line) {
    Serial.print("POWER "); Serial.println(laser_power);
} else if (strstr(line,"POWER ")==line) {
    laser_power=(int)atof(&line[6]);
    Serial.print("POWER "); Serial.println(laser_power);
    analogWrite(LASER,laser_power);
} else if (strstr(line,"FSCAN")==line) {
    int steps=(int)atof(&line[6]);
    digitalWrite(ENABLE,HIGH);
    delay(100);
    for (int i=0;i<abs(steps);i++) set_coils_fullstep(stepcounter++);
    delay(100);
    for (int i=0;i<abs(steps);i++) {
        set_coils_fullstep(stepcounter--);
        delay(50);
        unsigned long sum=0;
        for (int k=1; k<10; k++) {sum+=analogRead(A0); delay(10);}
        Serial.println(sum);
    }
    digitalWrite(ENABLE,LOW);
} else if (strstr(line,"CALIBRATE")==line) {
    for (int power=0; power<256; power++) {
        analogWrite(LASER,power);
        delay(100);
        unsigned long sum=0;
        for (int k=1; k<10; k++) {sum+=analogRead(A0); delay(10);}
        Serial.println(sum);
    }
    analogWrite(LASER,10);
} else {
    Serial.println("unknown");
}
}

```

```
}
```

At the top of the sketch, a number of variables are declared for the time to wait between stepper motor steps and the initial setting of the pulse-width modulation for the `laser_power`. Then the used pins for the stepper motor and the `LASER` are declared before defining the `set_coils_fullstep()` function, which is identical to the full-step version we used earlier in Section 4.5.4. We use full-step mode because the step resolution is sufficient, and in half-step mode there is a small visible disturbance on the sensor from the different currents drawn on alternate steps in half-step mode, where exciting one coil and two coils take turns. In the `setup()` function, we initialize the serial communication, declare the mode of the pins used for the motor, and initialize the laser power to be 10 (of up to 255). The `loop()` function is built in much the same way as before. It expects to receive single-line commands on the serial line. As in Section 4.5.4, the `FMOVE` command moves the motor by the specified number of steps. Then there are commands to set and return the `STEPS` and the time to `WAIT` or `ENABLE` the motor driver. The command `LDR?` reads the sensor value from analog pin A0, and the `POWER` command sets and reads the laser excitation. Finally we encounter the command `FSCAN nnn` where `nnn` is the scan range retrieved by reading the rest of line with the `atof()` function we used before. Here we assume that the scanner actually blocks the laser, and first needs to be retracted before collecting data. Inside the case block it first reports to the serial line what it is about to do, enables the motor driver, and moves the obstacle out of the way. Then comes a `for` loop, where the motor performs individual steps and within each step, takes ten measurements of the LDR on analog pin A0 and adds them up. At the end of each step the measurement value is written to the serial line. Once the number of `steps` is complete, the motor driver is disabled. The last command implemented is `CALIBRATE`, which measures the sensor response as a function of the laser power. The latter we may later use to convert the raw sensor readings to a linear intensity scale should the need arise.

As a first test of the system, we open the *Serial Monitor* in the Arduino IDE, make sure that the baud rate is set to the one specified in the `setup()` function, 9600 baud in our case, and query the LDR by sending `LDR?` to the UNO. The response has the format `LDR nnn`, where `nnn` is the raw value from the call to the `analogRead()` function. Then we query the laser power with `POWER?` and set it to a new value with `POWER nnn`, where `nnn` is a value between 0 and 255. The brightness of the laser spot should change accordingly. Next we move the stepper motor a few steps, for example, with `FSCAN 20`, and move it back to the starting position with `FSCAN -20`. The wagon with the obstacle should move back and forth. As a last point, we execute `FSCAN 60`, which will move the obstacle by 60 steps and then returns to the starting position step by step, while reading the sensor and reporting the value to the serial line. If all these initial tests complete satisfactorily, we progress to automatizing the process, and for that we chose octave.

Basically, we hook up the Arduino UNO to the Raspi, write an octave script to open a serial line, send commands to the UNO, and receive the response, very similar to what we did in previous examples. Before writing the octave script, we need to calibrate the motion of the obstacle in order to be able to show the width in mm instead of steps. In moving the wagon a large distance, I use 250 full steps, and, taking a photo of the setup with a ruler lying next to it before and after moving it, is easy to calculate the calibration constant `xscale` in mm/step. In my case, 250 steps moved the wagon by 39.2 mm, which explains the value for `xscale` near the top of the following script.

```
% scanplot2.m, V. Ziemann, 170628
close all;
clear all
```

```

xscale=0.161; % mm/fullstep
s=serial('/dev/ttyACM0',9600); % set device correctly
sleep(3);
srl_flush(s);          % flush the input queue
nsteps=60;
srl_write(s,"FSCAN 60");
sleep(5);
xx=zeros(1,nsteps);
yy=xx;
for i=1:nsteps
    xx(i)=i*xscale;
    yy(i)=str2double(serialReadline(s));
end
close(s)
yy=smooth3(yy);          % smooth data
subplot(2,1,1);
plot(xx,yy);             % raw sensor data
ylabel('arb. units');
subplot(2,1,2);
dy=yy(2:end)-yy(1:end-1); % derivative
if (yy(1) > yy(end)) dy=-dy; end
plot(xx(1:end-1),dy);
xlabel('x [mm]');
ylabel('arb. units');
title(['FWHM = ', num2str(xscale*fwhm(dy)),'%5.2f'), ' mm'])
print('laser_profile.png','-S1000,700')

```

The rest of the script should be familiar by now. After opening the serial line and flushing whatever characters are present, we define the number of steps and write **FSCAN 60** to the UNO. Then we wait a short time and start retrieving the values from the serial line with the `serialReadline()` function we already encountered in the previous chapter. It reads characters from the serial line until the termination character is encountered, and returns the obtained characters as a character string. But back to the main script. While retrieving data and copying it to the array `yy`, we also fill the array `xx` with the properly scaled values for the horizontal axis. Once the loop completes, we close the serial device.

After all data are available within the script, we start processing it. Since we will calculate the derivative of the raw data later, which is a process that is very sensitive to noise, we weakly smooth the sensor values in array `yy` with the `smooth()` function. It averages three consecutive values and replaces the central value with the average. The octave script for `smooth3()` is

```

% smooth three consecutive points, V. Ziemann, 170628
function out=smooth3(y);
y=[y(1),y(1:end),y(end)]; % add extremities
f=ones(1,3)/3.0;          % filter function
out=conv(y,f);             % convolute
out=out(3:end-2);          % ensure same length

```

It first creates a new array with the extreme points doubled up to avoid ugly artefacts at the end points. In the next line we create the filter function `f` that consists of three values $1/3$ and use it to convolute the data with the filter. Since the convolution creates an output

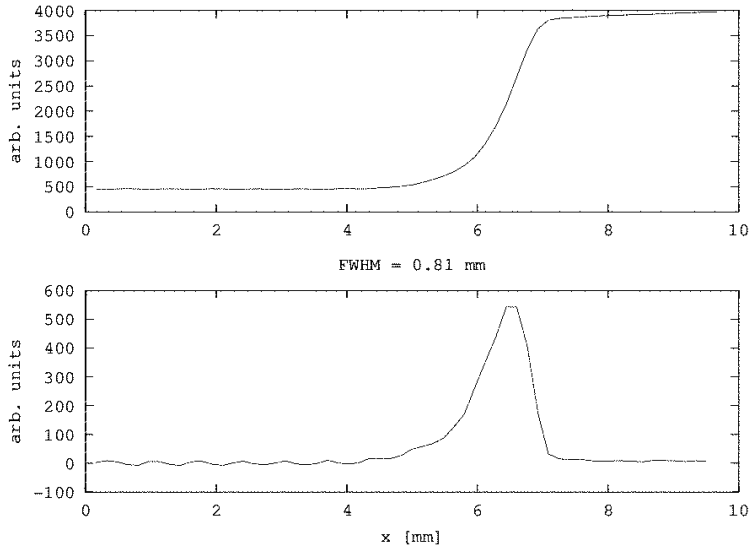


Figure 12.4 The raw sensor value as a function of the position of the obstacle and the derived laser beam profile, which shows a moderate asymmetry.

array that is longer than the original, we remove the extreme points before returning the smoothed array as the value `out`. Once these initial preparations are complete, we plot the smoothed raw values and add a label to the vertical axis. Then we calculate the difference between consecutive values, which is motivated by the fact that in each step a little of the laser beam is obscured and the *change* in remaining intensity is proportional to the intensity in the small band that the obstacle transverses during that step. The result is a signal that is proportional to the transverse intensity profile of the laser beam. For the resulting plots, see Figure 12.4. In the `title` statement, we add a title to the lower plot with the full width at half maximum (FWHM) of the laser profile, which is about 0.8 mm. Finally, we produce an image file, which is precisely how we prepared Figure 12.4.

In this example, we use the FWHM rather than the standard deviation of the profile, because the FWHM is a rather robust measure and works more reliably with profiles that are non-Gaussian, asymmetric, and have tails that are moderately populated. The standard deviation is heavily biased by the latter. The script for the FWHM is straightforward and is reproduced here.

```
% FWHM, V. Ziemann, 170628
function fwhm=fwhm(data)
N=length(data);
xmax = -1e30;
xmin=min(data);
imax=-1;
for i=1:N
    if (data(i) > xmax)
        xmax=data(i);
        imax=i;
    end
end
fwhm = xmax - xmin;
```

```

    end
end
ileft=imax;
while (data(ileft) > (xmax+xmin)/2 && ileft>1)
    ileft=ileft-1;
end
iright=imax;
while (data(iright) > (xmax+xmin)/2 && iright<N-1)
    iright=iright+1;
end
fwhm=iright-ileft-1;

```

In the `fwhm()` function we first locate the minimum and maximum of the data points as well as the location of the maximum. Then we search towards the left until the value is less than halfway between maximum and minimum. This results in the location `ileft` of that point. Then we repeat the process on the right-hand side and return the difference of the right and left halfway points as the FWHM. The minus one in the last equation accounts for the fact that both searches overshoot before terminating the respective `while` loops. This is partially compensated by subtracting one from the final result.

In this example, we built a laser-beam profile monitor from scratch using a small stepper motor and an LDR to illustrate the basic steps and how to combine sensors and actuators to build an integrated system that is controlled from `octave` or any other language that communicates over serial devices, such as EPICS, LabView, or C. The system is far from perfect and many limitations and improvements come to mind. For example: Is diffraction on the edge of the obstacle a limitation? Is the step size adequate or should we use a better, microstepping motor-driver? Does the profile change with different laser intensities? Do we need to correct for intensity-related nonlinearities of the sensor system? This is actually the purpose of the `CALIBRATE` command to provide the base information. Does the profile show the same asymmetry when we scan from the other side? We may also want to know the vertical beam size and any cross-plane correlations to determine some basic aberrations. Is the profile the same after swapping position of LDR and 10 k Ω resistor? We might want to add a second motor moving the obstacle longitudinally in the direction of the laser beam. Adding a lens and measuring the beam profile at several places allows us to determine the M^2 of the laser beam, which is a measure of the laser quality and is unity for a diffraction-limited beam. But these exercises we leave for the interested reader to do at home, while we move on to build a robot that detects fire.

Example: Fire-Seeking Robot

In this final example, we discuss a small robot that detects heat radiation from a fire, moves towards the heat source, and starts beeping once it reaches the source of the radiation. This is, of course, a very simple prototype of an autonomous fire extinguisher. We require the robot to either move autonomously, or to be manually controlled by a remote controller. Despite the simple description, the project is rather ambitious and requires us to solve the following subtasks:

1. detect the heat source and its location;
2. control the speed and direction of motors;
3. detect collisions, preferably before they happen;
4. sense joysticks and switches on the remote control;
5. send messages between remote controller and robot.

We base the hardware of the robot on a ready-made chassis with two DC motors and an on-board battery pack with five AAA cells providing up to 7.5 V. Onto that chassis we mount a full-size breadboard to house a NodeMCU controller, an H-bridge motor-driver, and some linear voltage converters to provide 5 and 3.3 V. To provide additional IO-pins we add a second Arduino that acts as a slave to the NodeMCU. On the robot chassis is a dedicated place for a model-servo onto which we mount a very small breadboard for the BPX38 IR phototransistors we use as flame sensors, and the HC-SR04 distance sensor. Figure 13.1 shows the chassis with the servo and the two breadboards already mounted. Mounting the sensors on the movable breadboard makes it possible to point them in different directions by turning the model-servo. We configure the NodeMCU as an access point that spans its own WLAN network. We also start a server to receive commands from the remote control. Apart from listening to the remote on the controller, we implement a simple state machine, to which we can hand the control of the robot, which subsequently runs autonomously.

The remote control is based on a second NodeMCU. It connects to the WLAN spanned by the first NodeMCU on the robot and to the server running on it. On the remote controller we connect a MCP3408 8-channel ADC via SPI to the NodeMCU, and set the input voltages of the ADC channels with two joysticks, two potentiometers, several voltage-dividers, and two switches. The ADC readings are then interpreted as motor speed and model-servo setting, and transmitted to the robot. The voltage on one ADC channel is set by a number of switches and voltage dividers, which allows us to use a larger number of switches than

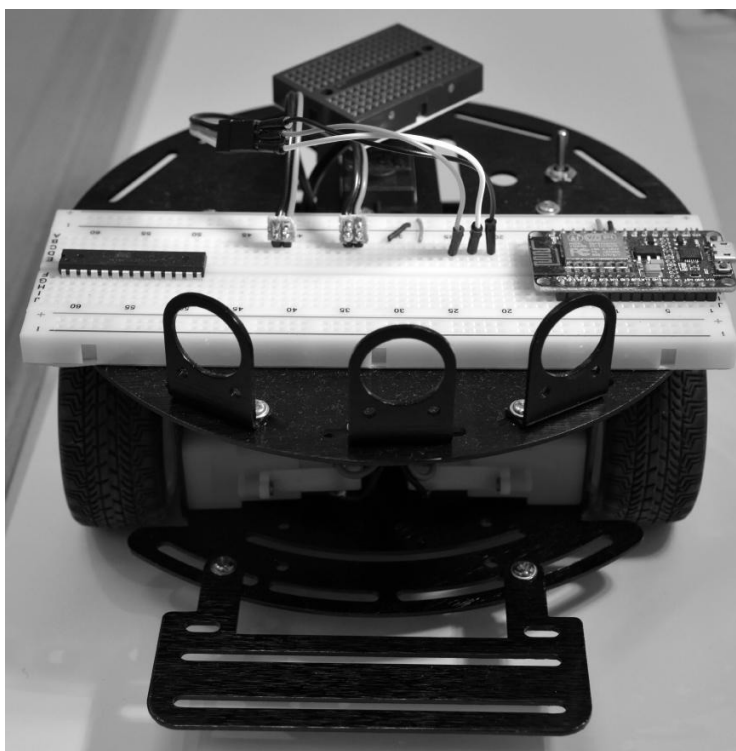


Figure 13.1 The chassis of the robot with two breadboards. The smaller one can be turned by operating a model-servo. Mounted on the larger breadboard are a NodeMCU on the right and a second chip on the left, a bare ATmega328 that was initially tested and later replaced by an Arduino NANO.

available digital IO pins, albeit at the expense of being able to sense only one button at a time.

We show a simplified version of the remote controller with less than all possible channels set up on a breadboard in Figure 13.2, where we see the NodeMCU on the right and the MCP3208 ADC on the left. The MCP3208 has the same pin assignment as the MCP3304 from Section 4.4.4, and is connected to the SPI port of the NodeMCU with CLK, MISO, MOSI, and CS lines, connected to D5, D6, D7, and D8, respectively. A joystick, visible immediately to the left of the breadboard, is connected to the power rails and the wiper to ADC channels A0 and A1. Likewise, the potentiometer, visible on the far left, is connected to ADC channel A5. Two buttons, visible near the bottom of Figure 13.2, connect voltage dividers consisting of a 10 k Ω resistor to the positive supply voltage, and a 68, and 47 k Ω resistor, respectively. By using further resistors to ground with different values, it is possible to detect more buttons—one at a time. Pressing the buttons supplies a definite voltage level to ADC channel A7. Finally, there is a switch connected to IO pin D4. To clarify the wiring, we also show the schematic corresponding to the breadboard layout in Figure 13.3.

The sketch running on the remote controller that uses the hardware described in the previous paragraph is the following. Since the sketch is fairly long, we show and describe the preparatory sections first.

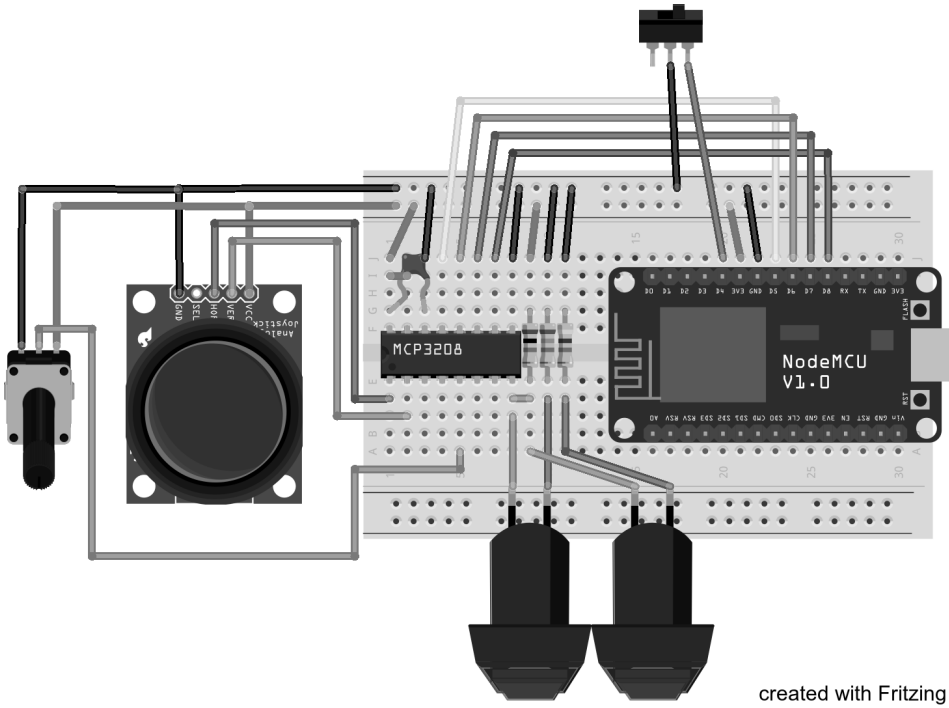


Figure 13.2 Simplified setup of the remote controller on a breadboard.

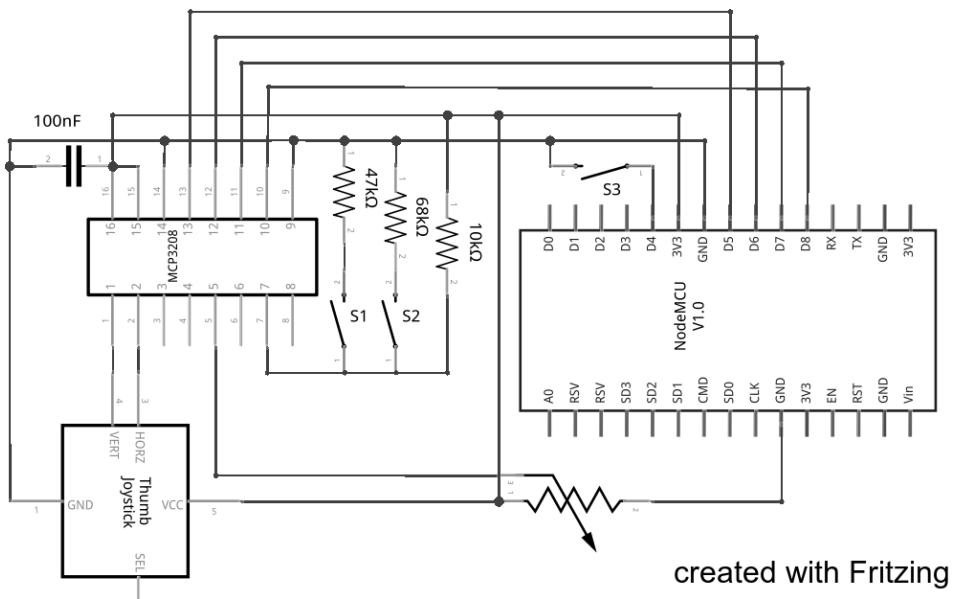


Figure 13.3 The schematic of the remote controller.

```

// RCsenderUDP, V. Ziemann, 170705
const char* ssid      = "FireBot";
const char* password = ".....";
const char* host = "192.168.4.1";
const int port=1137;
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>
WiFiUDP client;
int adclast[8]={0,0,0,0,0,0,0,0},D4last=HIGH;
int adccalib[8];
//.....ADC
#include <SPI.h>
#define CS 15
int mcp3208_read_adc(uint8_t channel) { // 8 single ended
    int adcvalue=0, b1=0, hi=0, lo=0, reading;
    digitalWrite (CS, LOW);
    byte commandbits = B00001100; // Startbit+(single ended=1)
    commandbits |= ((channel>>1) & 0x03);
    SPI.transfer(commandbits);
    commandbits=(channel & 0x01) << 7;
    b1 = SPI.transfer(commandbits);
    hi = b1 & B00011111;
    lo = SPI.transfer(0x00);    // input is don't care
    digitalWrite(CS, HIGH);
    reading = (hi << 7) + (lo >> 1);
    return reading;
}

void send_string(char line[]) { //.....send_string
    client.beginPacket(host,port);
    client.write(line);
    client.endPacket();
}

void setup() { //.....setup
    pinMode(LED_BUILTIN,OUTPUT);
    digitalWrite(LED_BUILTIN,LOW);
    pinMode(CS,OUTPUT);
    digitalWrite(CS,HIGH);
    SPI.begin();
    SPI.setFrequency(100000);
    SPI.setBitOrder(MSBFIRST);
    SPI.setDataMode(SPI_MODE0);
    pinMode(LED_BUILTIN,OUTPUT);
    digitalWrite(LED_BUILTIN,LOW);
    Serial.begin(115200);
    pinMode(D3,INPUT_PULLUP);
    pinMode(D4,INPUT_PULLUP);
    delay(1000);
    WiFi.mode(WIFI_STA); // needed for reliable communication
    WiFi.begin(ssid,password);

```

```

while (WiFi.status() != WL_CONNECTED) {
    Serial.print("."); delay(500);
}
Serial.print("\nConnected to "); Serial.print(ssid);
Serial.print(" with IP address: ");
Serial.println(WiFi.localIP());
client.begin(port);
for (int k=0;k<8;k++) {
    adccalib[k]=mcp3208_read_adc(k);
}
digitalWrite(LED_BUILTIN,HIGH);
}

```

At the top of the sketch, the WLAN name and passphrase are defined, as well as the robot's IP number, which defaults to 192.168.4.1 for the WLAN spanned by the robot. Then we include WiFi libraries and declare the `client`, which is the socket that connects to the server. Note that we use a UDP connection instead of a TCP, because the latter uses extensive handshaking between server and client and this makes the response of the robot to remote commands very sluggish. UDP connections do not use handshaking, which makes the connection less reliable, but much more responsive. Since we continuously send commands to the robot, losing a single command is not very serious. Next we declare a number of variables that we discuss later, and define the `mcp3208_read_adc()` function to read a single channel from the ADC. It is very similar to the one we discussed in Section 4.4.4, except that we request single-ended ADC measurements, and we receive 12 data bits instead of 13. The function receives the requested channel as input parameter and returns the 12-bit ADC reading. The `send_string()` function encapsulates the construction and sending of UDP packages. In the `setup()` function, we configure a number of IO pins, and initialize the SPI communication and the serial line. Then we define the `WiFi.mode` to be `WIFI_STA` or a client, rather than an access point, the latter otherwise being the default, and connect to the WLAN. Once connected, we write the received IP number to the serial line and connect to the server on the robot with the call to the `client.begin()` function. Finally, all ADC channels are read and the values saved in the array `adccalib[]`, which is used to calibrate the center positions of the joysticks.

After the preparatory sections, we are ready to show and discuss the main part of the program, the `loop()` function.

```

void loop() { //.....loop
    char line[30];
    int adc[8];
    for (int k=0;k<8;k++) {adc[k]=mcp3208_read_adc(k);}
    if ((abs(adc[0]-adclast[0]) > 16) || (abs(adc[1]-adclast[1]) > 16)) {
        adclast[0]=adc[0]; adclast[1]=adc[1];
        int val0=(adc[0]-adccalib[0])*1023.0/2048.0;
        int val1=(adc[1]-adccalib[1])*1023.0/2048.0;
        sprintf(line,"RSPEED %d",(int)(val0+0.5*val1)); send_string(line);
        sprintf(line,"LSPEED %d",(int)(val0-0.5*val1)); send_string(line);
    }
    if (abs(adc[5]-adclast[5]) > 16) {
        adclast[5]=adc[5];
        int val=adc[5]*180.0/4095;
        sprintf(line,"SERVO %d",val); send_string(line);
    }
}

```

```

}
if (abs(adc[7]-adclast[7]) > 5) { // check the buttons
  adclast[7]=adc[7];
  if (adc[7] < 1000) {
    Serial.println("Red button right pressed");
    sprintf(line,"FINDFIRE 1"); send_string(line);
  } else if (adc[7] < 1700) {
    Serial.println("Blue button right pressed");
    sprintf(line,"RANGE?"); send_string(line);
  } else if (adc[7] < 2250) {
    Serial.println("Joystick button right pressed");
    sprintf(line,"NEXTEVENT 0"); send_string(line);
  } else if (adc[7] < 3580) {
    Serial.println("Joystick button left pressed");
    sprintf(line,"NEXTEVENT 1"); send_string(line);
  } else if (adc[7] < 3660) {
    Serial.println("Blue button left pressed");
    sprintf(line,"BEEP 1000"); send_string(line);
    Serial.println(line);
  } else if (adc[7] < 3750) {
    Serial.println("Red button left pressed");
    sprintf(line,"NEXTEVENT 3"); send_string(line);
  }
}
}
if (digitalRead(D4) != D4last) {
  D4last=digitalRead(D4);
  sprintf(line,"D0 %d",D4last); send_string(line);
}
yield();
int packetsize=client.parsePacket();
if (packetsize) {
  char line[30];
  int len=client.read(line,30); line[len]='\0';
  Serial.print("Message:"); Serial.println(line);
}
if (Serial.available()) {
  Serial.readStringUntil('\n').toCharArray(line,30);
  send_string(line);
}
}
}

```

In the `loop()` function, we first read all ADC channels and then test whether channels A0 and A1 have changed significantly with respect to the last time; the values are saved in the array `adclast[]`. This construction is useful, because it prevents continuous sending of commands, and only does that if a value has changed. If that is the case, the difference of the current ADC reading with respect to the calibration value is stored in variables `val0` and `val1`. The first value is interpreted as the desired speed of the motors, and the second value as the direction. Thus we send commands to set `val0+0.5*val1` as speed to one motor and `val0-0.5*val1` to the other. Next the reading of ADC channel A5 is scaled to a value between 0 and 180 and transmitted to set the model-servo on the robot. ADC channel A7 is

used to interpret the buttons pressed. In the current realization, six buttons are connected, and pressing them sends a number of different commands to the robot. Testing for ADC values progresses from smaller to larger values, such that the buttons are prioritized in a natural way. If a button connected to a smaller resistor is pressed, all buttons connected to higher-valued resistors are ignored. The details of what action the commands cause on the robot when a button is pressed we explain later, when we discuss the software running on the robot. The call to the `yield()` function ensures that all background processes for WLAN and serial communication can complete pending tasks. The call to the `client.parsePacket()` function checks whether a message from the robot has arrived. Here we only copy the received message to the serial line. But it is easy to envision other ways to handle this; for example, by showing them on an LCD display connected via a I2C interface to pins D1 and D2 on the NodeMCU controller. These pins are not used in the present circuit and are available for expansions. Finally, we check whether anything has arrived on the serial line and pass it on to the robot. The direct two-way communication with the robot is a very convenient way to debug the system, by sending commands from the serial console if the remote controller is connected via USB cable to the host computer.

And this brings us to the electronics on the robot. In Figure 13.4 we show the circuit diagram to implement the functionality discussed earlier in this chapter. The central component is the NodeMCU microcontroller visible on the right of the solderless breadboard, with the L293D H-bridge motor driver and adjacent 7805 linear voltage regulator to its left. The voltage regulator receives power from the battery, whose negative pole is connected to system ground and the positive voltage to the input pin of the 7805. The positive battery voltage is also routed to the motor and logic power pins of the L293D motor driver. The 5 V output voltage of the 7805 voltage regulator is connected to the lower power rails from where it provides power to the Vin pin of the NodeMCU at its bottom right. The 5 V power is further routed to the model-servo and the HC-SR04 sonar sensor visible above the large breadboard. The 3.3 V voltage regulator on the NodeMCU provides power to the upper power rail, which carries 3.3 V and is routed to the respective circuits. The respective power rails have large electrolytic capacitors of 470 μ F connected, to buffer intermittent voltage requirements. Special care is needed to ensure that the three different voltages are routed correctly. Higher than permissible voltages can destroy some of the integrated circuits. The positive voltage from the battery only goes to the 7805 and L293D, and 5 V power the NodeMCU via its Vin pin, the servo, and the sonar. To facilitate the correct wiring, color images are made available on this book's web site.

Motor 1 is connected to the pins on the lower side of the L293D H-bridge motor driver, and the controlling input pins are connected to IO pins D3 and D4 on the NodeMCU. Likewise, motor 2 is connected to the upper half of the L293D, and the corresponding input pins are routed to pins D5 and D6 on the NodeMCU. The enable pins of the motor driver are permanently wired to 3.3 V. The model-servo is connected to the 5 V power rail, and its control wire connects to pin D7 on the NodeMCU, while the buzzer connects to pin D8. The HC-SR04 distance sonar is powered from the 5 V rail. It is triggered by a pulse from NodeMCU pin D1 and the echo is received on pin D2. Since the NodeMCU only operates on 3.3 V, we need to use a voltage divider made of a 10 k Ω and a 22 k Ω resistor to step the 5 V echo signal from the HC-SR04 to approximately 3.3 V. On the robot, the distance sensor is actually mounted on the small breadboard visible on the top left in Figure 13.4, which, in turn, is mounted on the movable axis of the model-servo that permits it to scan the surroundings for obstacles. Initially only the distance sensor and the middle BPX38 phototransistor with a pull-up resistor to the 3.3 V rail are mounted on the small breadboard to scan for both obstacles and sources of infrared radiation. The BPX38 is most sensitive

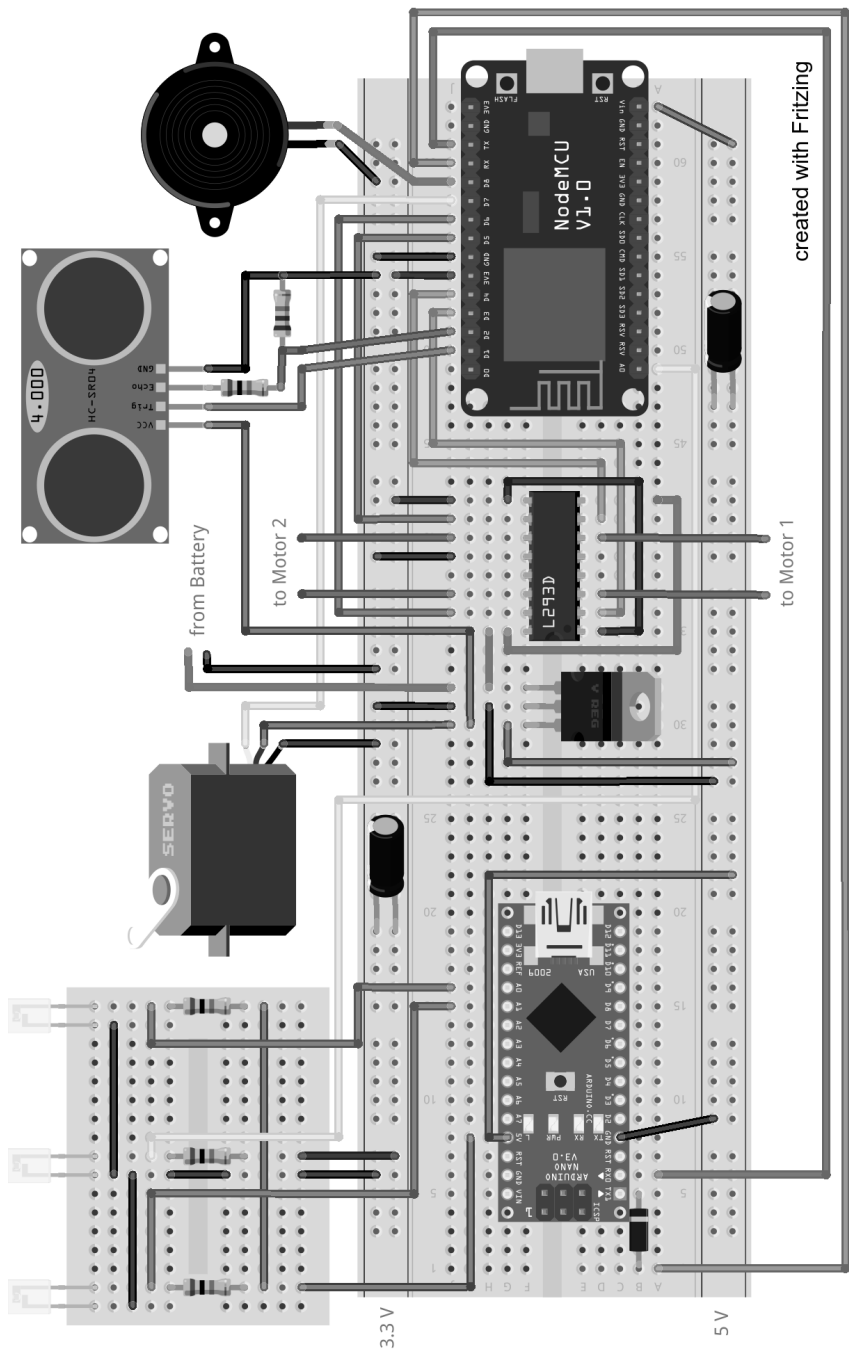


Figure 13.4 The electronics circuit of the robot (color version available online).

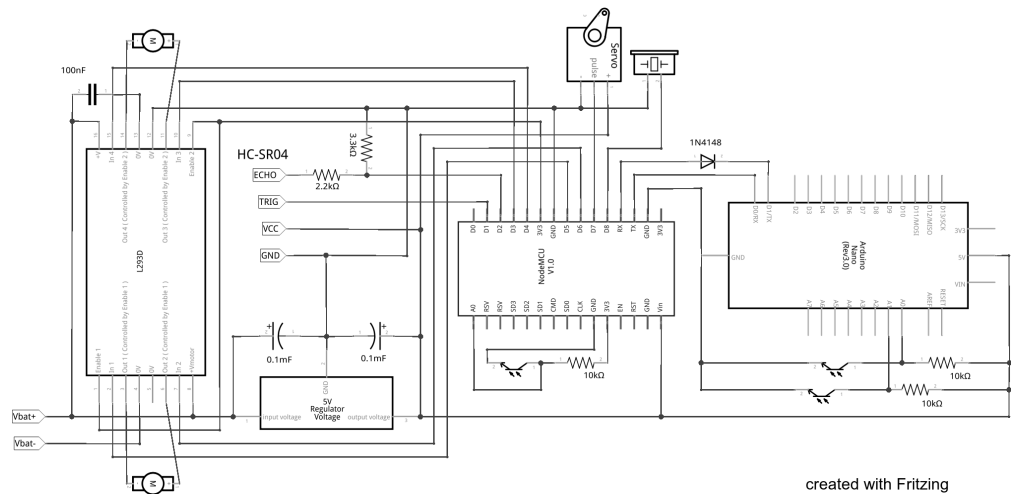


Figure 13.5 The schematic of the robot electronics.

around a photon wavelength of 880 nm, which is in the infrared part of the electromagnetic spectrum that is emitted as heat radiation by, for example, a fire or an old-fashioned light bulb. The emitter of the BPX38 is connected to ground, and the collector connects to the analog input pin A0 of the NodeMCU and via a 10 k Ω resistor to the 3.3 V power rail. In this configuration we need to continuously scan the IR and distance sensor with the servo in order to find the fire, which is a bit cumbersome. If we had two IR sensors to read out simultaneously, we could compare excitation and have instantaneous information about the location of the fire, without scanning with the servo. Therefore, we will add two extra IR phototransistors to the far left and far right on the small breadboard, wired in the same way as the middle one.

But we do not have enough analog input terminals on the NodeMCU, and have used almost all the IO pins. This significantly limits our ability add new functionality to the robot. We therefore add an Arduino-NANO to the large breadboard and program it to behave as a slave to the NodeMCU, and to communicate over the serial line. The wires that connect the respective TX and RX pins are crossed and behave similar to a null-modem cable. The NANO behaves almost like a UNO and is also programmed in the same way, by selecting *Arduino Nano* from the *Tools*→*Board* menu in the Arduino IDE. The only obviously visible difference is that the NANO has eight instead of six analog input pins, while there are also 13 digital IO pins. Since we power the NANO with 5 V supplied to the pin labeled “5 V”, all IO pins are operating on 5 V logic levels. In order not to damage the NodeMCU that operates at 3.3 V, we use a reverse-biased diode in the connection from the TX pin of the NANO to the RX pin of the NodeMCU. The diode blocks the 5 V from reaching the NodeMCU, but if the TX is pulled low, the signal on the input pin of the NodeMCU is also pulled low. The prototype circuit works with a normal switching 1N4148 diode, but ideally one should use a Schottky diode, which has a smaller voltage drop. Once the essential communication between NodeMCU and the slave-NANO works, we connect the two additional phototransistors from the small breadboard to analog input pins A0 and A1 on the slave NANO. This completes the description of the hardware on the robot chassis, and we can turn to programming the NodeMCU.

The code that runs on the robot is the following, again split in two blocks because it is rather long. First we show and discuss the preparatory sections.

```
// RCreceiver, V. Ziemann, 170701
#define Max(a,b) ((a)>(b)?(a):(b))
#define Min(a,b) ((a)<(b)?(a):(b))
const char *ap_ssid = "FireBot";
const char *ap_password = ".....";
const int port=1137;
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>
WiFiUDP server;
#include <Servo.h>
Servo myServo;
int servo_pos=90,servo_inc=5;
int right_sensor=-1,left_sensor=-1;
long lasttime=0,sleeptime=1000,nextevent=1;
void send_string(char line[]) { //.....send_string
    server.beginPacket(server.remoteIP(),port);
    server.write(line);
    server.endPacket();
}
int range() { //.....range
    digitalWrite(D1,LOW);
    delayMicroseconds(2);
    digitalWrite(D1,HIGH);
    delayMicroseconds(10);
    digitalWrite(D1,LOW);
    int val=(int)(0.017*pulseIn(D2,HIGH));
    if (val<10) tone(D8,1000,200);
    return val;
}
void motor_speed(int left, int right) { //.....motor_speed
    left=Max(-1023,Min(1023,left));
    analogWrite(D3,0);
    analogWrite(D4,0);
    if (left<0) {analogWrite(D3,abs(left));}
    else {analogWrite(D4,abs(left));}
    right=Max(-1023,Min(1023,right));
    analogWrite(D5,0);
    analogWrite(D6,0);
    if (right<0) {analogWrite(D5,abs(right));}
    else {analogWrite(D6,abs(right));}
}
void motor_stop() { //.....motor_stop
    analogWrite(D3,0);
    analogWrite(D4,0);
    analogWrite(D5,0);
    analogWrite(D6,0);
}
```

```

void setup() { //.....setup
  pinMode(LED_BUILTIN,OUTPUT);
  digitalWrite(LED_BUILTIN,LOW);
  pinMode(D1,OUTPUT); // HCSR04-TRIG
  digitalWrite(D1,LOW);
  pinMode(D2,INPUT); // HCSR04-ECHO
  pinMode(D3,OUTPUT);
  pinMode(D4,OUTPUT);
  analogWrite(D3,0);
  analogWrite(D4,0);
  pinMode(D5,OUTPUT);
  pinMode(D6,OUTPUT);
  analogWrite(D5,0);
  analogWrite(D6,0);
  Serial.begin(38400);
  WiFi.softAP(ap_ssid,ap_password);
  IPAddress myIP = WiFi.softAPIP();
  server.begin(port);
  Serial.print("\nAccess point and server started at address: ");
  Serial.print(myIP); Serial.print(" and port: "); Serial.println(port);
  Serial.print("with SSID: "); Serial.println(ap_ssid);
  pinMode(D7,OUTPUT); // D7=Servo, D8=Tone
  myServo.attach(D7);
  digitalWrite(LED_BUILTIN,HIGH);
  tone(D8,880,500);
  lasttime=millis();
}

```

This code runs on the NodeMCU on the robot, and first defines the `Max()` and `Min()` functions to determine the larger and smaller of two input values. Then it defines the name and passphrase of the WLAN as well as the used port number before including WiFi libraries for UDP and defining the `server`. We also include support for model-servos and declare one instance `myServo`, as well as variables needed for scanning the servo, reading the IR phototransistors, and the state machine. We then declare the convenience function `send_string()` to send a UDP package back to the remote controller. The `range()` function controls the HC-SR04 distance sensor in the way we discussed in Section 4.4.5. It ensures that the trigger pin D1 is `LOW` before pulling it `HIGH` for 10 μ s, and then waits for the echo to arrive on pin D2 with the `pulseIn()` function. The factor 0.017 converts the duration of the echo in microseconds to distance in cm. If an obstacle is closer than 10 cm, the buzzer briefly beeps before returning the distance as the function value. The `motor_speed()` function receives the speed, with sign indicating the direction, for the motors as input values. First it clamps the values to be within ± 1023 , turns both motors briefly off, and then, depending on the sign, sets the pulse-width modulation period of one or the other control pin to the desired value with a call to `analogWrite()`. It first handles one motor controlled by pins D3 and D4, and then the other motor, controlled by D5 and D6. The `motor_stop()` function turns all relevant IO pins off.

In the `setup()` function we mostly configure the IO pins as input or output according to their purpose. The `LED_BUILTIN` pin is actually D0 and controls an LED on the NodeMCU circuit board, which is convenient for debugging. Pulling D0 `LOW` lights the LED, and pulling it `HIGH` turns it off again. D1 and D2 are connected to the trigger and echo pin of the

distance sensor, and D3 to D6 to control the motors. Then we initialize the serial line to communicate at 38400 baud to accommodate the capabilities of the slave NANO before starting to span the WLAN with the call to the `WiFi.softAP()` function, which configures the NodeMCU as an access point with the supplied name and passphrase. The call to `WiFi.softAPIP()` returns the IP number of the access point, normally 192.168.4.1. Next we start the server to listen on the selected port for packets to arrive from the remote controller. Finally, we attach a servo controller to pin D7, turn the LED off, sound a short tone, and remember the elapsed time in the variable `lasttime`, which is needed for scheduling the state machine; but more on that topic later.

Having declared all variables and defined all preparatory functions, we can continue to discuss the `loop()` function of the sketch.

```
void loop() { //.....loop
  char line[30];
  int packetsize=server.parsePacket();
  if (packetsize) {
    int len=server.read(line,30);
    line[len]='\0';
    if (strstr(line,"LSPEED ")==line) {
      int val=(int)atof(&line[7]);
      val=Max(-1023,Min(1023,val));
      analogWrite(D3,0);
      analogWrite(D4,0);
      if (val<0) {analogWrite(D3,abs(val));}
      else {analogWrite(D4,abs(val));}
    } else if (strstr(line,"RSPEED ")==line) {
      int val=(int)atof(&line[7]);
      val=Max(-1023,Min(1023,val));
      analogWrite(D5,0);
      analogWrite(D6,0);
      if (val<0) {analogWrite(D5,abs(val));}
      else {analogWrite(D6,abs(val));}
    } else if (strstr(line,":")==line) {
      line[len]='\n'; line[len+1]='\0';
      Serial.println(line);
    } else if (strstr(line,"D0 ")==line) {
      int val=(int)atof(&line[3]);
      if (val==0) {
        digitalWrite(LED_BUILTIN,HIGH);
      } else {
        digitalWrite(LED_BUILTIN,LOW);
      }
    } else if (strstr(line,"SERVO ")==line) {
      int val=(int)atof(&line[6]);
      myServo.write(val);
    } else if (strstr(line,"BEEP ") {
      int val=(int)atof(&line[5]);
      Serial.print("BEEP val= "); Serial.println(val);
      tone(D8,440,val);
    } else if (strstr(line,"A0?")) {
```

```

    int val=analogRead(A0);
    Serial.print("A0 "); Serial.println(val);
    sprintf(line,"A0 %d",val); send_string(line);
} else if (strstr(line,"RANGE?")) {
    int val=range();
    sprintf(line,"RANGE %d",val); send_string(line);
} else if (strstr(line,"SCANRANGE ")==line) {
    int val=(int)atof(&line[10]);
    int minval=2000,minpos=-1;
    if (val>0) {
        myServo.write(10);
        delay(1000);
        for (int k=10;k<170;k+=5) {
            myServo.write(k); delay(200);
            val=range();
            if (val<minval) { minval=val; minpos=k;}
            sprintf(line,"SCANRANGE %d %d",k,val); send_string(line);
        }
        myServo.write(minpos);
        sprintf(line,"MINIMUM at %d",minpos); send_string(line);
    } else {
        myServo.write(90);
    }
} else if (strstr(line,"FINDFIRE ")==line) {
    int val=(int)atof(&line[9]);
    int minval=2000,minpos=-1;
    if (val>0) {
        myServo.write(10);
        delay(1000);
        for (int k=10;k<170;k+=5) {
            myServo.write(k); delay(200);
            val=analogRead(A0);
            if (val<minval) { minval=val; minpos=k;}
            sprintf(line,"FINDFIRE %d %d",k,val); send_string(line);
        }
        myServo.write(minpos);
        sprintf(line,"MINIMUM at %d",minpos); send_string(line);
    } else {
        myServo.write(90);
    }
} else if (strstr(line,"NEXTEVENT ")==line) {
    nextevent=(int)atof(&line[10]);
} else if (strstr(line,"SLEEPTIME ")==line) {
    sleeptime=(int)atof(&line[10]);
} else {
    Serial.println("unknown");
}
}
yield();

```

```

if (millis()>lasttime+sleeptime) { // next scheduled event
  switch (nextevent) {
    case 1: // determine range
      sprintf(line,"RANGE %d",range()); send_string(line);
      sprintf(line,"A0 %d",analogRead(A0)); send_string(line);
      nextevent=1;
      break;
    case 2: // scan with servo
      servo_pos+=servo_inc;
      if (servo_pos>170) {servo_inc=-servo_inc;}
      if (servo_pos<10) {servo_inc=-servo_inc;}
      myServo.write(servo_pos);
      nextevent=2;
    case 3: // request direction sensors
      if (range()<10) {
        motor_stop();
        nextevent=1;
      } else {
        Serial.println(":A0?\n:A1?");
        sleeptime=50; nextevent=4;
      }
      break;
    case 4: // read direction sensors and take action
      if ((right_sensor>0) && (left_sensor>0)) { // new data
        if ((left_sensor<250) || (right_sensor<250)) {
          sprintf(line,"SENSORS %d %d",left_sensor,right_sensor);
          send_string(line);
          int val0=(int)(600+0.2*(right_sensor-left_sensor));
          val0=Max(-1023,Min(1023,val0));
          int val1=(int)(600-0.2*(right_sensor-left_sensor));
          val1=Max(-1023,Min(1023,val1));
          motor_speed(val0,val1);
        } else {
          motor_stop();
        }
        right_sensor=-1; left_sensor=-1;
        sleeptime=1000; nextevent=3;
      }
    default:
      break;
  }
  lasttime=millis();
}
yield();
if (Serial.available()) {
  Serial.readStringUntil('\n').toCharArray(line,30);
  send_string(line);
  if (strstr(line,".A0 ")==line) {
    right_sensor=(int)atof(&line[3]);
  }
}

```

```

    } else if (strstr(line, ".A1 ")==line) {
        left_sensor=(int)atof(&line[3]);
    }
}
}

```

Here we first check whether a UDP packet has arrived from the remote controller with the call to the `server.parsePacket()` function. If a packet of size `packetsize` is available, it is read with `server.read()` and its length is determined. To avoid ugly output, we ensure that the last character is a NULL character, to indicate the end of a string and start testing what type of command has arrived. If it is `LSPEED` we interpret the rest of the line as the speed value with the call to `atof()`, clamp the values to the acceptable range, and set the speed of the motor, in the same way we do in the `motor.speed()` function. If `RSPEED` is received, the speed of the other motor is adjusted. If the line starts with a colon `:` it is passed on to the serial line. In this way we can send commands to the slave NANO. If the command starts with `D0` we turn the built-in LED on and off. The `SERVO` command sets the angle of the model-servo, and `BEEP nnn` makes a sound of 440 Hz for the duration of `nnn` ms. The `A0?` command returns the analog value read from the analog pin on the NodeMCU to the serial line and to the remote controller, and the `RANGE?` command likewise returns the distance to an obstacle as determined by the HC-SR04 sensor. The following command, `SCANRANGE 1`, starts the servo to move, which causes the distance sensor to point towards different directions, while the HC-SR04 scans the distance to an obstacle, and simultaneously records the direction and distance to the closest object. Finally, the servo is moved to point towards the minimum distance, and returns the direction at which the closest object is found. Calling `SCANRANGE` with argument 0 causes the servo to move to its middle position. The `FINDFIRE` command performs the same action, but scans the middle IR photodiode connected to analog pin A0 on the NodeMCU to determine the direction of a heat source. Note that the phototransistor pulls the signal line towards ground, which causes the voltage on pin A0 to approach zero, if a heat source is detected. The last two commands, `NEXTEVENT` and `SLEEPTIME`, allow us to set variables related to the state machine that governs the autonomous running of the robot. They are attached to buttons on the remote controller and can be changed by pressing these buttons.

The next section of the code, following the `yield()` command, implements this very simple state machine to operate asynchronously with all other actions that the NodeMCU performs. First we test whether the current time exceeds the value for the next scheduled event, `lasttime+sleeptime`, has elapsed. If that is the case, we branch according to the value of the `nextevent` variable. If it is 1 we only determine the distance from the HC-SR04 sensor with the `range()` function and the reading of the middle phototransistor. Then we set `nextevent=1`, which will repeat the same action after `sleeptime` has elapsed. If `nextevent` is 2, we sweep the servo position back and forth across its range, one step at a time. The next two event codes, 3 and 4, implement the autonomous motion of the robot towards a heat source. If `nextevent` is 3, we first test whether an object is closer than 10 cm, stop the motors, and branch to event code 1 discussed above. If there is no close obstacle, the commands `:A0?` and `:A1?` are sent on the serial line. Here we employ the convention that a command prepended with a colon is interpreted by the slave NANO, which in this case is requested to read its analog pins A0 and A1, to which the two outer phototransistors on the small breadboard are connected. Since we have two transistors, the difference of the reading will provide information about the direction of the heat source. The command is only dispatched in event code 3, but by setting the `sleeptime` to 50 ms and `nextevent=4`, we will execute event code 4 about 50 ms later. The response from the slave NANO is recorded

asynchronously, and we discuss the sketch running on the NANO a little later. After 50 ms, event code 4 is executed, and there we test whether the phototransistors detected a valid signal, and whether one of the signals is sufficiently small to be interpreted as a heat source. In that case, the sensor readings are sent to the remote controller, and the motor speed of one motor is set to a constant, not-too-large value, here 600, plus a small contribution proportional to the difference between the sensor readings. This small contribution is added to the speed of one motor and subtracted from the other. In this way we implement a very simple proportional control loop that feeds the error signal to the motors, to make the robot turn towards the heat source. If no heat source is detected, we stop the motors. Before leaving this part of the program, we set the variables `right_sensor` and `left_sensor` to an invalid value to indicate that no new value is present. Finally, we make the state machine sleep for 1000 ms and restart with event code 3 in order to repeat the process of reading the sensors and feeding the difference of their reading to the motors. Before leaving this part of the program, we update the `lasttime` variable in order to know when to handle the next event code. After the call to `yield()`, we test whether any characters are available on the serial line, and if the response starts with `.A0` or `.A1`, we interpret the numerical value as the `right_sensor` or the `left_sensor` reading. We point out that handling the communication via UDP packets and on the serial line is handled asynchronously, and interleaved with the state machine executing the events enumerated by the variable `nextevent`.

The serial line serves a dual purpose: First, it displays debugging information on the host computer if that is connected, and second, it communicates with the slave NANO. In order to distinguish the latter, we adapted the query-response protocol used earlier by the convention that all communication *to* the slave NANO starts with a colon “:,” and all communication *from* the slave starts with a period “.” such that those commands can be easily filtered out from the serial line. The sketch that runs on the slave NANO is the following.

```
// Slaveduino, V. Ziemann, 170723
void setup() { //.....setup
  Serial.begin(38400);
  while (!Serial) {}
  pinMode(13,OUTPUT);
  digitalWrite(13,LOW);
  pinMode(8,INPUT_PULLUP);
}
void loop() { //.....loop
  char line[30];
  if (Serial.available()) {
    Serial.readStringUntil('\n').toCharArray(line,30);
    if (strstr(line,":A0?")==line) {
      Serial.print(".A0 "); Serial.println(analogRead(A0));
    } else if (strstr(line,":A1?")==line) {
      Serial.print(".A1 "); Serial.println(analogRead(A1));
    } else if (strstr(line,":D13 ")==line) {
      int val=(int)atof(&line[4]);
      if (val==0) {digitalWrite(13,LOW);} else {digitalWrite(13,HIGH);}
    } else if (strstr(line,":D8?")==line) {
      Serial.print(".D8 "); Serial.println(digitalRead(8));
    }
  }
  delay(3);
}
```

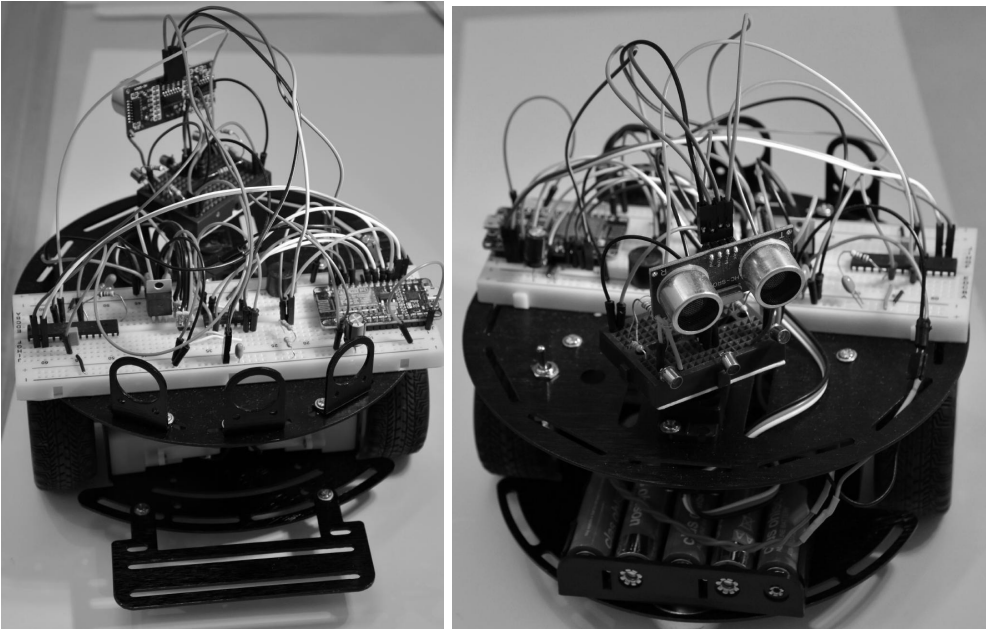


Figure 13.6 The operational robot from the back (left) and from the front (right).

```

    }
}

```

This sketch follows earlier examples, and in the `setup()` function we configure the serial line and the used pins to be `OUTPUT` or `INPUT`, in this case even with internal pull-up resistor enabled. In the `loop()` function we test whether data is available on the serial line and then test the different requests the slave is able to handle, namely responding to `:A0?`, `:A1?`, `:D13`, and `:D8?`, which all start with a colon, while any other query is silently ignored. Note also that any reply back to the NodeMCU via the serial line is prepended by a period. In the code running on the NodeMCU we only react to `:A0?` and `:A1?` and their respective response, but extending the code running on the slave NANO is easy to implement.

In Figure 13.6 we show an early prototype of the operational robot from the back and from the front. On the left-hand image we see the larger breadboard with the NodeMCU on the right. In the center of the breadboard, hidden behind the wires, is the L293D, and next to it the voltage regulator. In this prototype, instead of NANO, we use an ATmega328 that is programmed in an Arduino UNO, removed from its socket, and inserted in the breadboard on the robot. After adding a 16 MHz crystal and two 22 pF ballast capacitors, it works equally well on the breadboard. On the right we see the front of the robot with the small breadboard slightly turned to the side. On it the distance sensor and below it the three phototransistors are visible. Owing to the breadboards, the wiring has a distinct “spaghetti-flavor,” which makes the robot rather fragile, but also very convenient for developing and testing new functionality.

At this point we can control our simple robot with the remote controller and also make it follow a simple algorithm to find a heat source autonomously. We have to admit that the performance of the prototype system is less than impressive; the heat source has to be rather close to be detectable and the motion towards it is awkward. But our main purpose

is to show how to implement all the functionality in a simple prototype system. Building a marketable system requires significantly more effort.

The hardware with one large breadboard directly mounted on the robot chassis and a smaller one movable on the model-servo makes the system very versatile and extendable; for example, connecting other sensors or making the robot follow other algorithms. We use a slave NANO as an IO extender, but also other extenders are available, such as the MCP23017 that provides up to 16 digital IO pins, or the PCA9685 that controls up to 16 pulse-width modulated outputs or model-servos. Both extenders are controlled via the I2C-port on the NodeMCU and require two IO-pins only. The system is open for further experiments, and to expand the system. Other improvements comprise fine-tuning the algorithms to make the robot more robust, and whatever else come to mind.

Now that we have discussed the hardware and software for our projects, we are ready to use the equipment in experiments, gather measurements, and interpret them. Once we reach firm conclusions, we want to present the results in a seminar and eventually write a report about the experiment. These two points, related to presenting our work, are the topic of the final chapter.

QUESTIONS AND PROJECT IDEAS

1. Discuss the pros and cons of using TCP versus UDP.
2. Add a second state-machine (thread) to the robot that periodically reads out an MQ-x gas-sensor and sounds a special alarm, once it reports a significant presence of a gas.
3. Add an LCD display with I2C interface to the remote controller, to show status messages from the robot.
4. *Build a remote-controlled boat* instead of a robot. The boat can be powered by a fan salvaged from a computer. We can steer it by directing the air stream behind the propeller with fins controlled by a model-servo.
5. *Build a line-follower* that uses LDR or phototransistors pointed towards the floor. Program it to follow a white (or black) line made of masking tape.
6. *Build a vending machine* that detects the size and weight of inserted coins and that pushes a chewing gum or other desirable item from a safe place to the bin where we can pick it up.
7. Construct the *model crane* from question 13 in Chapter 3. Equip the containers with a marker, such as a periodically flashing LED, and invent a scheme to run the crane autonomously.
8. Contemplate how to remotely control a *sailboat*. What actuators do you need? In case you want to add autonomous control, which sensors do you need?

Presenting and Writing

After having gone through the basic electronics and programming examples, both on Arduinos and on the Raspi, and after completing several projects, we need to communicate our activities to our colleagues and condense our activities into a well-motivated and concise sequence of descriptions. This can be in the form of a presentation with slides, or as a report for a thesis or a journal. As a template for the contents of either presentation or report, we use the weather station example, just to illustrate the concepts with a specific example. We start with the discussion of a presentation with slides.

14.1 PREPARING A PRESENTATION

The key issues when preparing a presentation are *motivation* and a good *story line*, as well as substantial subject matter. Why is that so? The audience of a seminar usually is rather heterogenous and we have to provide a funnel to guide them from their different backgrounds to the subject matter of the seminar. Often a good start is stating a problem that is easy to understand and then pointing out how our project solves or at least alleviates the problem. The weather station came up in my home lab, because we use pressure vessels filled with liquid helium as well as high-voltage equipment. The potential effect of barometric pressure on the pressure vessels and the relevance of humidity for high voltage is intuitively understandable for most people. So that is a viable motivation as to why we care about the weather station as a device that helps us to correlate weather-based data with other measurement data.

Once we have captured the attention of the audience with a catchy motivation, we need to keep it alive by following a well-conceived *story line* that places the relevant topics—the subject matter of the seminar—in a logically coherent sequence where one detail follows the previous in a natural way. My suggestion is to start with easily understandable facts and then progressively increase the complexity of the discussed material. We must avoid situations where the audience starts wondering *why* we talk about a particular topic and loses track of our story line. Since the story line addresses the overall organization of the presentation it needs to be sorted out beforehand.

A method to organize a presentation that I found useful is the following: I start by estimating the number of slides that fit into the allotted time. An average slide typically requires 2 to 3 minutes for the audience to absorb. Thus, for a 20-minute presentation I target about 8 to 10 slides as a rule of thumb. Then I prepare a title page with a catchy title that encapsulates the essential point of the seminar, followed by one slide that motivates the problem I intend to address. This I follow up with a brief and qualitative description of the idea of how to solve it that I will elaborate in the remainder of the seminar. At

this point the audience should have a good idea about the scope of my seminar and should be convinced that the idea I pursue has a decent chance of addressing the problem in a meaningful way. On the subsequent slides it is useful to concentrate on the *flow* of some quantity or some information from one stage to the next. In the example with the weather station, the information flows from the sensor via the microcontroller to the host computer where it is presented, either on a web page or accessible via control system. This flow of information we can mimic in the organization of the slides.

For the bulk of the presentation, I recommend preparing the allotted number of empty slides and giving each slide a title. The title should address one key issue per slide and the sequence of slides should follow the logical flow we have identified beforehand. In the weather-station example we have the issues: hardware with sensors, microcontroller, host computer, then software running on the respective computing devices and protocols used to interface the devices. Once we have the slides with their titles, we can shuffle them around until we are satisfied with our story line.

In the next step I place one or several pictures or graphs on each slide to illustrate the topic on the slide. Once all, or at least most, slides are equipped with a picture, I add a few bullet-points with keywords to each slide. They remind me of what I intend to say when presenting. They help me to understand my slides after 6 months, and those in the audience to remember what I said after 2 months. The slides are not a substitute for a self-contained report, but serve as illustrations for my presentation during a seminar.

Once all slides contain a picture and a few keywords, it is time to review the story line again and see whether linking of the slides works. This addresses the flow from one slide to the next and whether it comes in a natural and well-motivated way. If many forward references are necessary, I consider reordering the slides to achieve a more natural flow where information required at some stage is already discussed on a previous slide. Sometimes forward references are difficult to avoid, but I try to minimize them.

When almost all slides are completed, I suggest preparing a final slide with a clear synopsis of the main results and possibly some comments about how to extend the work. After a final pass through the slides, with a check for coherence to ensure that the title and final slides act as parentheses to enclose the subject matter, the presentation is ready.

Just for convenience I summarize the above guidelines in a presentation cookbook:

Motivation: the problem and intended way to address it.

Think of a story line and the logical flow of a concept, preferably with increasing complexity.

Write a title on each slide and sort to follow the story line.

Add pictures to slides.

Add keywords to slides.

Add slide with conclusions and outlook.

Final check of coherence and linking. Done!

Naturally, my presentation cookbook is subjective, and should not keep you from using a working way of preparing slides, but if you get stuck very early on in the process of preparing a seminar, my guidelines may help you to get started.

Presenting one's work in a seminar is the first step, and writing it up in a report or a thesis is the next.

14.2 PREPARING A REPORT

When preparing a report or a thesis in general, I follow the same guidelines used for the preparation of a presentation. I need to *motivate* what I am about to describe and then adhere to a *story line* that follows the *flow* of a concept. As a first step I normally try to formulate a catchy title and an abstract of 100 to 200 words to explain to myself what I intend to write in the report. Often this is a reasonable starting point even for the final version of the abstract.

For the main part of the report, instead of starting with empty slides and filling the title line, I start by identifying a logical sequence of chapters, sections, or subsections, and give each one a title before ordering them in a sequence that follows the flow of the argument. In the next step I add pictures or other illustrative material to the respective sections. This normally results in a valid skeleton for the report that I need to flesh out in the next step. I start this process by adding bullet lists to the sections with topics that are relevant in the respective section. I do this for all originally identified chapters or sections, and often in this stage I note that I need to split sections into two or more or that I can combine sections with similar contents into a single one. As a guideline I assume that each topic in the bullet list will require one paragraph with about 200 words or so to discuss adequately. I also try to roughly balance the length of sections, but this is of minor concern. Once I know what contents should show up in the respective sections, I start writing the text and “fill in the blanks.” Since key components—images, graphs, or tables—are already in place, I start each section with a short description of how it fits into the flow and then describe the material. I also try to write each section from start to end, because that helps to maintain a logical flow. At other times, if I do not come up with a decent start, I just start at a place where I have a good idea about how to present it and retrofit the missing parts later. In that case, however, I need to pay special attention later to ensure the flow.

A particularly important section is the *Introduction*, where I need to convince the reader that the report treats a relevant and interesting subject by stating the chosen problem and how to solve it. But ideally the problem should also be put into the context of previous work. What have other researchers done before on related problems? How does their work differ from mine? This section, typically one or two paragraphs long, requires a number of key references from the published literature. At the end of the introduction, I sometimes give a brief outline of the report in the fashion of a commented table of contents. By this time I have hopefully convinced the reader to continue with the remainder of my report. And this is the prime task of the introduction: a brief statement of the problem, the context, and a brief outline of things to come.

The *main part* of the report should follow the story line and the flow I initially decided upon. I write one or two paragraphs for each topic in the bullet list and make sure that the paragraphs are well connected in the sense that the reader knows how the following paragraph links to the present one. I try to avoid having the reader wonder what the lines he is presently reading have to do with the overall story line. Placing crosslinks with references to topics addressed earlier and how they connect to the present paragraph are helpful to make the text denser, and creates additional associations for the reader. Phrases such as, “where we use the result from page...” or “as discussed in Section...” illustrate the idea.

In the *Conclusions*, I give a synoptic review of the main part and the results of the report. I try to organize the introduction and summary jointly as an executive summary. Often, readers first read those two sections before deciding to spend time on the report in its entirety. The most important part of the conclusions is a concise summary of the key

results. I sometimes follow this up with a number of questions that came up during the research and that may lead to further work, either by me or some other researcher.

I like to point out that the organizational triplet of introduction, main body, and conclusion vaguely resembles the form of a sonata, where first the themes are introduced. In the main body of the sonata the themes are elaborated and in the coda they often reappear in the original form once again. In this section we discussed the structure of the report and my recommendations are again subjective, but may help in case you are stuck.

After the discussion of the overall organization of the report, we now look at some of the ingredients and start with the presentation of data, often using graphics and plots.

14.3 PRESENTING DATA

Many aspects of scientific work are presented with the help of figures describing the experimental setup, or graphs showing the data, sometimes with comparison to a model. In [29] E. Tufte names three guiding principles to achieve graphical excellence.

Clarity is the first principle, and it requires carefully explaining the experiment from which the data originate, describing what quantities are plotted on a graph, and properly labeling the axes. Moreover, using a legible font of adequate font-size and avoiding low-contrast colors such as yellow or light green on white background, is mandatory.

Precision, the second principle, dictates presenting data honestly and truthfully. If data points are excluded, the reason and method to select the discarded points need to be explained. The graphics data of a manuscript must be proofread in the same way as the main text. Make sure to catch all errors, such as omitted exponents or lost or incorrect axis labels. Choose axes adequately. A bad example is to display the average temperature on Earth in Kelvin with zero displayed. Global warming is invisible on such a plot.

Efficiency, the third principle, aims at presenting information in a parsimonious way: to present the largest possible insight with the least effort. In particular, display only data that advance the main argument, but then explain *all* displayed features in the plot, either in the caption or in the text. If the plot is part of a seminar, the speaker may explain it as well. On the other hand, any unnecessary data, which Tufte calls “chart-junk,” should be omitted from a graph. A bad example is to present all available measurement data, even though a well-chosen subset suffices to make a point.

In *The Elements of Graphing Data*, W. Cleveland discusses a large number of guidelines to produce graphical data, with many examples. He summarizes the guidelines in a concise list of “rules” such as

- to make the data the most prominent feature and avoid cluttering the data area;
- to keep the number of tick marks limited and preferably outside the data area;
- to have compatible scales when comparing two data sets;

and many more. Consulting the book is highly recommended in case of questions about graphing data.

Scientific journals have a strong interest that their authors produce high-quality articles, and those often include graphical data. They often expand the general terms of Tufte and Cleveland and provide comprehensive style guides, both for written and for graphical material. One good example is [30], which inspired the following points.

- Ensure your audience understands four things about the data points: what *quantity* they represent, their physical *unit*, their *magnitude*, and the *uncertainty*.

- Explain the error bars: Are they standard errors or confidence limits?
- Always label axes with quantity plotted and with units. I normally display units in square brackets. In journals, the use of title text is discouraged in favor of text in the caption.
- If there are several plots in the same graph, preferably label them, use a legend, or explain them in the figure caption.
- Choose the vertical scale such that at least 80 % of the range is used.
- If an axis covers more than 2 orders of magnitude, consider using logarithmic scale, if appropriate.
- Avoid “eye-candy” such as 3-D bar-graphs or pie-charts.
- When only a few data points need to be presented, say 10 or less, a table is often preferable to a plot.

These basic guidelines should provide a starting point to produce presentable plots.

The last point in the above list refers to data presented in *numerical form* in tables or in the main text. Under no circumstances should you state more than a sensible number of significant figures! If a measurement generates data that is accurate to 1 %, two or maybe three significant figures are adequate. Just because a computer displays results in double precision with 10 or more figures, this does not mean all figures are significant. And finally, error bars in numeric form should never be displayed with more than one or sometimes two significant figures.

And this brings us to writing the main text. Therefore, a few words about writing good English are in order.

14.4 GOOD ENGLISH

There are a number of style guides for the English language available, from the classic *Elements of Style* [31], the MLA handbook [32], S. Pinker’s *Sense of Style*, [33] to S. King’s *On Writing* [34]. Apart from these more general guides, several scientific journals such as *Nature* [35] or *Reviews of Modern Physics* [36] make style guides available for their authors. In particular, the latter style guide is very readable, and its Appendix A on “Writing a better scientific article,” is highly recommended. In this section I highlight some of the topics in more detail.

- In one of the style guides [35] we find the sentence “Nature journals prefer authors to write in the active voice. . . .” The *active voice* usually makes the presentation clearer, more vigorous, and often more concise. Experiments are not done by themselves, but we, the experimenters, perform them. Moreover, often the subject matter is already difficult to understand, so we should make the presentation as clear as possible without obfuscating who does what. Science does not become more objective just because the person who did the experiment hides behind the passive voice.
- Authors should strive for *economy* in their presentation and avoid unnecessarily complicated constructions such as “owing to the fact that,” which we can usually replace with “because.” Often replacing passive constructions with active ones helps to disentangle complex sentences.

- Avoid overloading the reader with sentences containing a large number of subclauses to explain every conceivable exception in one sentence. At the end of the sentence, the reader may have forgotten the beginning and what the subject of the sentence is.
- Be friendly to your reader and invite her into your intellectual world by more or less addressing her directly, using phrases such as “let us” or “we now return.” This helps to make the report more accessible by mimicking normal colloquial speaking patterns. The reader is not alienated by a stiff and abstract presentation but feels welcome to mentally participate in your exposé.
- Commit yourself to what you write and avoid unnecessary hedging by conditionals. This is easily done by trying to replace any occurrence of “might be” or “could be” with “is.” Other hedging phrases to look out for are “may” and “could,” or anything that hints at you being scared of writing what you really mean.
- If in doubt, use English instead of Latin phrases. In that sense, “first” is a better choice than “initial,” use “place” instead of “location.”
- Avoid acronyms unless a long term with a commonly used acronym appears in several places throughout the report. In that case, introduce the acronym at the first occurrence by writing out the full term first and adding the acronym in brackets. Henceforth use the acronym consistently!
- Avoid jargon! Someone uninitiated in the jargon specific to your particular project might want to read—and understand—your report.
- Pay special attention to grammatical correctness and especially to agreement errors. Ask an English-speaking colleague to proofread your report.
- Before releasing your report, make sure to run it through a spell-checker, and read it a final time to verify that all is correct!

These guidelines are certainly incomplete, but hopefully will help you to write an interesting and readable report. I suggest you also read one of the more comprehensive style guides such as [36]. A further source of advice regarding writing and presenting scientific work in general is *Nature* publication’s web page, *English communication for scientists* [37]. A wonderful source of synonyms and antonyms is the *WordNet browser* [38]. You may consider using either the online version or a version installed on your computer in order to make your writing more lively.

14.5 POSTSCRIPTUM

And at this point we have reached the end of the book. In the process we talked about various sensors, both analog and digital, connected them to microcontrollers, massaged the data into a common format, and passed them on to a host computer. There we postprocessed the measurements, stored them in databases, and prepared them for a presentation or report. I hope that you found useful, interesting, and inspiring topics between Preface and Postscriptum that will help you in your projects that have a data-acquisition aspect.

I tried out all circuitry and programming presented in the book, but the odd bug may have crept in. If you find one, please do not keep the bug, but share it with me so I can improve future versions of the book. Of course, the general disclaimer applies, namely that the code in the book is provided *as is* and users are advised to use caution, but are in any case responsible themselves for using the program code.

Basic Circuit Theory

As a reminder, we will briefly review the basic theory of electronic circuits. We start by considering *Ohm's law*, which states that the current I in a segment of a circuit is proportional to the voltage U with the resistance R as the proportionality constant, or $U = IR$. This is a consequence of the balance of the electric field accelerating electrons in the *resistor* and the friction force experienced by electrons due to scattering with vibration modes, phonons, of the ions that make up the material. The force pulling on the electrons is proportional to U and the friction force is proportional to the electron's drift velocity v , or by $-\alpha v$ where α is a material-dependent friction coefficient. Under stationary conditions, the two forces need to balance, and we have $v = U/\alpha$. But the current I is proportional to the drift velocity v and we find $I \propto v = U/\alpha$, which is the essence of Ohm's law, and the friction coefficient α is inversely proportional to the resistance R . In case the voltage is varying moderately slowly compared to the relaxation time of the electrons, the current directly follows the applied voltage; the current I and the voltage U are in phase.

For *capacitors* the situation is different, because the two plates of a capacitor are electrically separated and do not allow constant transport of charges under stationary conditions. They can, however, store a charge Q on the plates, and the capacitance C is the proportionality constant between Q and the applied voltage U , given by $Q = CU$. Since the charge Q changes as a consequence of current I flowing onto the plates, we have $U = (1/C) \int^t I dt'$, or by differentiation, $dU/dt = (1/C)I$. A sinusoidally oscillating voltage $U \propto e^{i\omega t}$ with frequency ω will therefore be related to the current by $U = (1/i\omega C)I$, and we can identify $Z_C = 1/i\omega C$ as the generalized resistance, the *impedance*, of the capacitor. Here i is the imaginary unit.

Inductors are made of coils that store energy in their magnetic field, and if we turn off the current, a voltage develops. Expressing this behavior in a formal way, we have $U \propto dI/dt$, with the inductance L of the coil as the proportionality constant. If we again assume that the coil is excited by a sinusoidal current $I \propto e^{i\omega t}$ with frequency ω , we find that we have $U = (i\omega L)I$ and identify $Z_L = i\omega L$ as the impedance of the inductor.

Now that we have resistances and impedances of common elements found in simple circuits, we may ask how to combine them to networks. This question is answered by *Kirchhoff's laws*, of which the first states that under stationary conditions, all currents flowing into a network node must add up to zero. This is a statement about the preservation of charges, and what comes in must also come out, because under stationary conditions piling up charges is not allowed. The second law states that the voltage differences around a loop in the network have to add up to zero, which is the requirement that the voltages at each node of the network with respect to a reference node are unique.

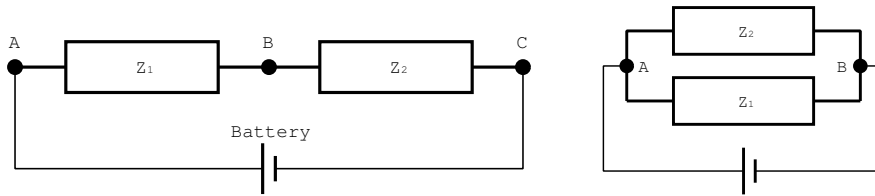


Figure A.1 Two impedances connected in series (left) and in parallel (right).

We immediately use these laws to investigate the resistance or impedances connected in series, as shown on the left in Figure A.1. The currents in node B need to balance, and this implies that the current I_1 passing through Z_1 is the same as the current I_2 passing through Z_2 . At the same time, the voltage U provided by the battery is the same that is dropped across the two impedances. We thus find $U = I_1 Z_1 + U_2 Z_2 = I_1 (Z_1 + Z_2)$, and we find that the total impedance of the circuit equals the sum of the impedances coupled in series.

Considering the circuit on the right in Figure A.1, we know that the currents through the two branches have to add up: $I_t = U/Z_t = I_1 + I_2 = U/Z_1 + U/Z_2$. We can simplify this to $1/Z_t = 1/Z_1 + 1/Z_2$, which implies that the individual impedances add as reciprocals if the impedances are coupled in parallel.

Note that the argument in the previous two paragraphs is valid for normal resistors and constant (DC) voltages and currents, but works as well for AC voltages if we use the complex frequency-dependent impedances for inductors and capacitances. We also note that all relations among currents and voltages are *linear*, such that we can use the super-position principle and analyze circuits for each voltage or current source independently and then add the contributions in the end. A further consequence is *Thevenin's theorem*, which states that any subcircuit connected via two terminals a and b to the rest of the circuit can be replaced by a voltage source U_{th} and a series resistor R_{th} . It turns out that U_{th} and R_{th} can be very easily determined. U_{th} is given by the voltage between unconnected (open-circuit) terminals a and b , and the resistance R_{th} by the current $I_{th} = U_{th}/R_{th}$ that flows through the short-circuited link between terminals a and b .

Apart from the linear circuit elements, the impedances, there are also nonlinear elements, and *diodes* are among them. Their voltage-current behavior follows an exponential dependence given by $I \propto (e^{(eU - E_g)/kT} - 1)$, where $E_g \approx 1.2 \text{ V}$ is the bandgap energy of the semiconductor material, here silicon. We see that for negative voltages the current is very small, while for positive currents it grows exponentially once the threshold of the bandgap voltage is passed. Practically, diodes conduct current in one direction and block current in the opposite direction. This makes them perfect to rectify voltages in the way we discussed in Section 2.2.5. A diode is conducting if it is forward biased and the cathode is at a more negative voltage than the anode.

Transistors are other nonlinear elements based on two diodes coupled antiparallel; they come in two types, dependent on whether the central tap, called the base terminal, is an anode or a cathode. Injecting an additional current into the base terminal enables a larger current flow through the two outer terminals, called collector and emitter. In the main body of the text we use transistors for switching applications, but with suitable ancillary circuitry they can be used as amplifiers as well. That goes beyond the scope of this appendix and is covered in books on electrical engineering, such as [15] or [16].

Least-Squares Fit and Error Propagation

In Chapter 11 we used the `linfit()` function on the Arduino to fit a straight line through a number n of data points sampled at equidistant times. Here we briefly discuss the inner workings of that function, but use the opportunity to expand that topic, because fitting and error propagation analysis are essential for any experimental activity. We therefore extend the fitting procedure to comprise generalized least-square problems, and give a rudimentary overview of the corresponding error analysis and of error-propagation in general.

But we start with the problem of determining the slope a and intercept b of a fit to a straight line. These parameters are determined by the requirement to minimize the sum of squared residuals $r_i = y_i - at_i - b$ for a number of data points (t_i, y_i) . We can restate the problem in the form of a linear equation that needs to be inverted in the least-squares sense

$$\begin{pmatrix} \vdots \\ y_i \\ \vdots \end{pmatrix} = \begin{pmatrix} \vdots & \vdots \\ t_i & 1 \\ \vdots & \vdots \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \quad (\text{B.1})$$

which we write in abbreviated form

$$y = Ax \quad (\text{B.2})$$

where A is the $n \times 2$ matrix in the previous equation and $x = (a, b)^T$. Here the superscripted T denotes the transpose, and for the squared sum of the residuals $\sum_{i=1}^n r_i^2$ we write χ^2 . The latter we express as

$$\begin{aligned} \chi^2 &= \sum_{i=1}^n r_i^2 = (y^T - x^T A^T)(y - Ax) \\ &= y^T y - x^T A^T y - y^T A x + x^T A^T A x . \end{aligned} \quad (\text{B.3})$$

Minimizing this expression with respect to x results in a condition for the sought solution vector x . The condition for a minimum is the requirement that the gradient with respect to the fit parameters x^T is zero. We somewhat sloppily write it as

$$0 = \frac{\partial \chi^2}{\partial x^T} = -2A^T y + 2A^T A x \quad (\text{B.4})$$

where we used $x^T A y = y^T A^T x$ because the expressions are scalars and any matrix of

the form $A^T A$ is symmetric. This allows us to combine terms and we arrive at the previous equation. For a more detailed derivation see [39]. Left-multiplying with the inverse of $(A^T A)$, provided the matrix is nonsingular, isolates the sought-after fit-parameter x and we obtain

$$\begin{pmatrix} a \\ b \end{pmatrix} = x = (A^T A)^{-1} A^T y. \quad (\text{B.5})$$

The right-hand side is often called pseudo-inverse of the matrix A , and we need to evaluate this expression in order to find a and b . If the t values are equidistant, we can absorb the step size in a redefinition of a , and the matrix A obtains the form

$$A = \begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \\ \vdots & \vdots \end{pmatrix} \quad (\text{B.6})$$

such that $A^T A$ becomes

$$A^T A = \begin{pmatrix} \sum_{k=1}^n k^2 & \sum_{k=1}^n k \\ \sum_{k=1}^n k & \sum_{k=1}^n 1 \end{pmatrix} = \begin{pmatrix} n(n+1)(2n+1)/6 & n(n+1)/2 \\ n(n+1)/2 & n \end{pmatrix} \quad (\text{B.7})$$

which accounts for the definition of **S0**, **S1**, **S2** in the `linfit` routine. Inverting this 2×2 matrix is trivial, and calculating $A^T y$ is done in the loop where `ay0` and `ay1` are calculated. The `linfit` function then returns the parameter a , the slope, and in the main program we need to rescale the result by multiplying the slope with the step size. It is easy to cross-check the results with octave, and the fitting is done with the `polyfit()` function, but if we want to do the calculation on the Arduino, we need methods like the one shown in this appendix.

Note also that generalizing the method to higher-order polynomials is simple. We only need to add columns in Equation B.1 with t_i^n and can rescale the fit parameters a, b, \dots by the appropriate power of the step size, and we are left with adding powers of positive integers for which closed expressions exist. This means that the matrix $A^T A$ can always be calculated in closed form for any number of data points n . Only the inversion of a matrix with rank of the number of fit parameters remains. Being able to precompute most of the matrices and possibly inverting them on the host computer for fixed n makes this method rather suitable for microcontrollers.

And going beyond fitting polynomials, we realize that any problem that can be cast into the form $y = Ax$ is solved by the pseudo-inverse given by $x = (A^T A)^{-1} A^T y$. Often the column-vector y contains the measurements that have a linear dependence on fit-parameters x , and that dependence is characterized by a system-matrix A . Here A is the response matrix of the measurements y on some parameters x . We observe that the pseudo-inverse provides a linear map of values y in the “measurement-space” onto the values x in the “fit-parameter space.” In general, there are more measurements denoted by n than fit-parameters m , thus $n > m$ and the system is overdetermined.

Up to now we assume that all measurements contribute equally to the fit result, but this is often not the case, because there are measurement values y that are unreliable for some reason, and they have large associated errors. If we denote the measurement uncertainty for measurement y_i by σ_i , we weigh each measurement in Equation B.2 by the inverse of σ_i . In this way a large error bar σ_i effectively “turns off” one measurement. We formalize this by introducing the matrix $\Lambda = \text{diag}(1/\sigma_1, 1/\sigma_2, \dots, 1/\sigma_n)$ and left-multiplying Equation B.2 by Λ with the result

$$\Lambda y = \Lambda A x. \quad (\text{B.8})$$

Redoing the calculation that led to the pseudo-inverse now leads to

$$x = (A^T \Lambda^2 A)^{-1} A^T \Lambda^2 y \quad (\text{B.9})$$

which takes the measurement error bars into account.

The next question we may ask is how the error bars of the measurements σ_i determine the error bars of the fit parameter x . In order to visualize the calculation, we assume that the error bars σ_i come from repeatedly measuring y_i and using the sample average $\langle y_i \rangle$ as input to the calculation, and that the root-mean squared $\sigma_i^2 = \langle (y_i - \langle y_i \rangle)^2 \rangle$ determines the error bar. Here we use the angle brackets to denote averaging over the ensemble of measurements. If, in order to simplify the notation, we introduce the abbreviation $R = (A^T \Lambda^2 A)^{-1} A^T \Lambda^2$, we obtain for the ensemble average of the fit-parameter $\langle x_i \rangle$, written for one component

$$\langle x_i \rangle = \left\langle \sum_{j=1}^n R_{ij} y_j \right\rangle = \sum_{j=1}^n R_{ij} \langle y_j \rangle \quad (\text{B.10})$$

because taking the average is a linear operation and allows us to extract the sum from the average. The error bars of the fit-parameters are then given by the covariance matrix $C(x)$. Considering only one component of the covariance matrix $C(x)_{ij}$, we find

$$\begin{aligned} C(x)_{ij} &= \langle (x_i - \langle x_i \rangle)(x_j - \langle x_j \rangle) \rangle \\ &= \left\langle \left(\sum_{k=1}^n R_{ik} (y_k - \langle y_k \rangle) \right) \left(\sum_{l=1}^n R_{jl} (y_l - \langle y_l \rangle) \right) \right\rangle \\ &= \sum_{k=1}^n \sum_{l=1}^n R_{ik} R_{jl} \langle (y_k - \langle y_k \rangle)(y_l - \langle y_l \rangle) \rangle \\ &= \sum_{k=1}^n \sum_{l=1}^n R_{ik} R_{jl} C(y)_{kl} \end{aligned} \quad (\text{B.11})$$

which can be written in matrix form as

$$C(x) = RC(y)R^T. \quad (\text{B.12})$$

This equation is very powerful, because it allows us to calculate the covariance matrix of the fit-parameters as a function of the covariance matrix of the measurements and the pseudo-inverse that we abbreviated R . Now we realize that the covariance matrix of the measurements is $C(y) = \text{diag}(\sigma_1^2, \sigma_2^2, \dots, \sigma_n^2) = \Lambda^{-2}$ and inserting this and the definition of R leads to the result

$$C(x) = (A^T \Lambda^2 A)^{-1} \quad (\text{B.13})$$

which contains the squared error bars of the fit parameters on the diagonal, such that

$$\sigma(x)_i = \sqrt{C(x)_{ii}} \quad (\text{B.14})$$

and the off-diagonal elements contain information of the correlations among the fit-parameters.

We point out that Equation B.12 describes error propagation from variables y to variables x , and the derivation depends on the linearity of the map from y to x . If the underlying map from some set of variables y to another set x is non-linear such that we can write $x = x(y)$, we can still linearize around some value \bar{y} such that the Jacobi matrix

$J_{ij} = \partial x_i / \partial y_j|_{y=\bar{y}}$ describes how much small deviations in y in the vicinity of \bar{y} change the value of x . This is how error bars in y cause error bars in x . We can therefore use the Jacobi matrix J instead of the matrix R to describe the covariance matrix in the new variables x in Equation B.12.

To illustrate this, we consider a simple example $x = y_1/y_2$ with covariance matrix

$$C(y) = \begin{pmatrix} \sigma_1^2 & r_{12} \\ r_{12} & \sigma_2^2 \end{pmatrix}. \quad (\text{B.15})$$

The Jacobi matrix is given by

$$J = \left(\frac{\partial x}{\partial y_1}, \frac{\partial x}{\partial y_2} \right) = \left(\frac{1}{y_2}, -\frac{y_1}{y_2^2} \right) \quad (\text{B.16})$$

and for the covariance matrix $C(x)$ we find

$$C(x) = JC(y)J^T = \sigma_1^2 \frac{1}{y_2^2} - 2r_{12} \frac{y_1}{y_2^3} + \sigma_2^2 \frac{y_1^2}{y_2^4} \quad (\text{B.17})$$

which agrees with the “conventional wisdom” for adding measurement errors for uncorrelated variables y_1 and y_2

$$\sigma(x)^2 = \sigma_1^2 \left| \frac{\partial x}{\partial y_1} \right|^2 + \sigma_2^2 \left| \frac{\partial x}{\partial y_2} \right|^2$$

On the other hand, if the variables are correlated, which is expressed by a nonzero component r_{12} in the covariance matrix, using Equation B.12 with the Jacobi-matrix substituted for R yields consistent results.

QUESTIONS AND PROJECT IDEAS

1. Verify that Equation B.13 is correct by inserting the definition of R into Equation B.12.
2. Determine the covariance matrix for the linear fit to determine slope a and intercept b . Assume that all measurement errors are equal to 5% of the maximum measurement value y if you have $n = 10, 100$, and 1000 measurements. How large are the resulting error bars for a and b ? Discuss the relevance of the off-diagonal element in the covariance matrix.
3. Provided the covariance matrix of two variables y_1 and y_2 are given by Equation B.15, calculate the covariance matrix of the variables $x_1 = y_1/y_2$ and $x_2 = y_2/y_1$. Are the errors of x_1 and x_2 correlated?

Bibliography

- [1] Fritzting. Project web site: <http://fritzing.org/>.
- [2] ATLAS collaboration. Project web site: <https://cern.ch/atlas>.
- [3] CMS collaboration. Project web site: <https://cern.ch/cms>.
- [4] Large Hadron Collider. Project web site: <https://cern.ch/lhc>.
- [5] CERN the European Center for Nuclear Research. Project web site: <https://cern.ch>.
- [6] MQTT. Project web site: <http://mqtt.org/>.
- [7] EPICS. Project web site: <http://www.aps.anl.gov/epics>.
- [8] Arduino. Project web site: <https://www.arduino.cc>.
- [9] Raspberry Pi. Project web site: <https://www.raspberrypi.org>.
- [10] J. Fraden. *Handbook of modern sensors*. Springer Verlag, Berlin, third edition, 2004.
- [11] J. Wilson, editor. *Sensor Technology Handbook*. Elsevier, Amsterdam, 2005.
- [12] R. B. Northrop. *Introduction to Instrumentation and Measurements*. CRC Press, Boca Raton, third edition, 2014.
- [13] M. Coplan, J. Moore, and C. Davies. *Building Scientific Apparatus*. Cambridge University Press, Cambridge, UK, fourth edition, 2009.
- [14] C. Kittel. *Introduction to Solid State Physics*. Wiley, Hoboken, NJ, eighth edition, 2005.
- [15] T. Giurma and P. Peebles. *Principles of Electrical Engineering*. McGraw-Hill, New York, 1991.
- [16] Analog Devices: Analog Circuit Design Tutorials. Project web site: <http://www.analog.com/en/education/education-library/tutorials/ebooks.html>.
- [17] D. Lancaster. *Active Filter Cookbook*. Newnes, Oxford, second edition, 1996.
- [18] G. Franklin, J. Powell, and A. Emami-Naeni. *Feedback Control of Dynamic Systems*. Pearson, Boston, seventh edition, 2015.
- [19] M. Margolis. *Arduino Cookbook*. O'Reilly, Sebastopol, CA, 2011.
- [20] E. Bartmann. *Die elektronische Welt mit Raspberry Pi entdecken*. O'Reilly, Sebastopol, CA, 2013.

- [21] M. Kofler, C. Kühnast, and C. Scherbeck. *Raspberry Pi, das umfassende Handbuch*. Galileo Press, Bonn, 2014.
- [22] S. Monk. *Raspberry Pi Cookbook*. O'Reilly, Sebastopol, CA, 2014.
- [23] Scientific Programming Language GNU Octave. Project web site: <https://www.gnu.org/software/octave/>.
- [24] Python Software Foundation. Project web site: <https://www.python.org/>.
- [25] MySQL Database. Project web site: <https://dev.mysql.com>.
- [26] RRDtool. Project web site: <https://oss.oetiker.ch/rrdtool/>.
- [27] Apache web server. Project web site: <https://httpd.apache.org/>.
- [28] Installing EPICS on the Raspberry Pi. <https://prjemian.github.io/epicspi/>.
- [29] E. Tufte. *The visual display of quantitative information*. Graphic Press, Cheshire, 1983.
- [30] C. Mack. How to write a good scientific paper: Figures, part 1. *J. Micr/Nanolith*, 12:040101–1, 2013.
- [31] W. Strunk and E. White. *The Elements of Style*. Harcourt, 1920. Online available from Project Gutenberg at <https://www.gutenberg.org/ebooks/37134>.
- [32] J. Gibaldi. *MLA Handbook for Writers of Research papers*. Modern Language Association of America, New York, NY, sixth edition, 2003.
- [33] S. Pinker. *The Sense of Style*. Penguin Books, London, 2015.
- [34] S. King. *On writing*. Hodder, London, 2012.
- [35] Nature style guide. http://www.nature.com/authors/author_resources/how_write.html.
- [36] K. Friedman. Reviews of modern physics style guide, <http://journals.aps.org/files/rmpguide.pdf>.
- [37] Nature. English communication for scientists, <http://www.nature.com/scitable/ebooks/english-communication-for-scientists-14053993/contents>.
- [38] C. Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge, MA. Online version: <http://wordnet.princeton.edu>.
- [39] W. Press et al. *Numerical Recipes*. Cambridge University Press, Cambridge, second edition, 1992.

Index

7805, 28, 211

Accelerometer, 13, 34, 35, 73

Acronym, 142, 228

ADC, *see* Analog-to-digital

ADXL335, 13, 34

Aliasing, 26, 177

Amplifier, 17–20, 22, 177, 178

Analog-to-digital, 1, 16, 20, 23–26, 33–35,
62, 68, 73, 76, 77, 79, 80, 117, 177,
181, 184, 205, 206, 209, 210

Apache2, 141–144, 172

Arduino, iii, 1, 3, 55–58, 60, 63–68, 71, 74,
76, 77, 79, 82–86, 89, 91–95,
97–99, 101, 103–105, 110–112, 117,
123–125, 139, 146, 185, 186, 191,
198, 201, 205, 213, 221, 231

Arduino NANO, 112, 213

Arduino sketch

 Analog input, 61, 63

 Blink, 58

 BMP180 pressure sensor, 69, 71

 Button, 60

 Capacitance measurement, 191

 Color monitor, 186, 188

 DC motor with H bridge, 90

 DHT11 humidity sensor, 80

 DS18B20 temperature sensor, 83

 Geophone client, 177

 HC-SR04 distance sensor, 83

 HYT221 humidity sensor, 67

 Laser-beam profile, 198

 LM35 temperature logger, 65

 MCP23017 IO-extender, 74

 MCP3304 ADC with SPI, 77, 79

 MCP4921 DAC, 101

 MLX90614 IR thermometer, 66

 Model-servo, 91

 MPU6050 accelerometer, 72

 Query-response, 64

 RC receiver, 214

 RC sender, 206

 Rotary encoder, 84

Serial communication, 61, 63

Socket communication, 109

Sound, 103

Stepper motor, 93, 95, 99

Switching and PWM, 86

Weather station, 167

Web server, 107, 214

Bandgap, 7, 10, 27, 37, 39, 230

Barometric pressure, 1, 32, 33, 68, 165, 167,
175, 223

Basic Unix programs, 115

Battery, 28, 112, 157, 177, 190, 205, 211

Bluetooth, 104–105, 110, 113, 124, 128, 130
 pairing and setup, 105

BMP180, 32, 33, 70, 71, 111, 167, 170

BPW34, 14, 15

BPX38, 15, 205, 211, 213

Breadboard, 16, 33, 77, 89, 93, 98, 167, 185,
186, 191, 197, 205–207, 211, 213,
219, 221, 222

Brightness, 40, 85, 87, 107–109, 111, 112,
146, 165

Button, 32, 59–61, 65, 74, 76, 109, 115, 116,
119, 123, 145, 206, 211

Capacitance, 3, 13, 14, 21, 22, 27, 29, 33,
34, 37, 38, 42, 191, 194, 229, 230

Commutator, 43, 44

Comparator, 20, 23, 24, 38

Compile, 59, 62, 148, 151, 153, 154

Conclusions, 224, 225

Conduction band, 7, 8, 15, 39

Control system, 3, 147–156, 161, 174

Covariance matrix, 233, 234

Crane, 54, 222

Cron process, 133, 134, 137, 140, 141, 144,
145

DAC, *see* Digital-to-analog

Darlington, 41, 42, 50, 53, 85, 87, 93

Database, iv, 117, 118, 132–140, 149,
152–154, 156, 163–165, 167, 171,
172, 174, 175, 182, 183

- DC motor, 12, 43–46, 52, 53, 89, 90, 94, 165, 205, 210, 211, 219, 220
- Decoupling capacitor, 63, 167
- Delta-sigma ADC, 24, 25
- Desktop, 114, 116, 120, 121, 123, 124, 146, 150, 156
- DHCP, 106, 108, 121, 122
- DHT11, 36, 37, 80, 81
- Dielectric constant, 8, 33, 36
- Digital-to-analog, 24, 51, 52, 101, 102
- DNS, 106
- DRV8825, 49, 98, 99
- DS18B20, 37, 82, 83
- Dust sensor, 37, 112, 175
- EPICS, iv, 1, 3, 147–157, 161–165, 167, 174, 175, 177, 182, 183
- EPICS database file
 - geophone, 182
 - MQTT interface, 163
 - simple, 149
 - temperature, 152
 - weather station, 174
- ESP-01, 56–58, 112
- ESP8266, 55–57, 105, 110
- Fermi level, 7, 38
- Filter, 1, 21–22, 26, 29, 32, 51, 53, 111, 177, 183
- Flatfile database, 132–134, 137
- Flow, 12, 15, 30, 42, 45, 52, 93, 224, 225, 230
- Fluid level sensor, 9
- Flyback diode, 43, 50, 52, 87–89
- Force sensitive resistor, *see* Strain gauge
- Gas sensor, 9–11, 112
- Gateway, 157, 161–165
- Geophone, 13, 14, 177, 178, 180–183
- GPIO, 65
- GPS, 36, 112
- Grammar, 228
- Ground vibrations, 3, 177
- Gyroscope, 34, 35, 73
- H bridge, 45, 46, 48, 50, 53, 85, 88–90, 94–96, 165, 198, 205, 211
- Hall sensor, 1, 13, 19, 35, 44, 112
- HC-SR04, 31, 83, 112, 205, 211, 215, 219
- HDMI, 113, 114
- HTML, 108, 109, 112, 142, 143, 173
 - examples, 142, 143, 145, 173
 - HTML forms, 109, 112, 146
 - http-equiv, 145
- HTTP, 112
 - GET, 108, 109
 - header, 108
 - return code, 108
- Humidity, 1, 33, 36, 67, 68, 71, 80, 82, 165, 170–172, 223
- Hydraulics, 52
- HYT-221, 33, 67, 68
- I2C, 30, 32–35, 52, 57, 65–68, 70–76, 111, 113, 167, 170, 211
- Imagemagick, 118
- Impedance, 5, 18, 21, 53, 194, 229, 230
- Inductance, 21, 22, 38, 229
- Infrared, 12, 15, 30, 40, 110, 213
- Internet of Things, 1, 3, 56, 157
- Interrupt, 74, 76, 81, 84, 85, 111, 179, 183, 184, 191, 193
- IOC, 147, 153, 154, 156, 183
- IP, 116, 121, 122, 124, 128, 131, 141, 153, 162, 164, 167, 171, 183, 209, 216
- iptables, 121
- Jacobi matrix, 233
- Jargon, 228
- Joystick, 9, 10, 205, 206, 209
- Keyword, 5, 83, 119, 224
- KODI, 113
- L293D, 46, 88, 89, 94, 95, 97, 198, 211, 221
- LDO regulator, 28
- LDR, 5–7, 17, 146, 175, 197, 198, 201, 204, 222
- Least-squares, 231
- LM35, 10, 11, 27, 61–63, 106, 107, 139, 159, 163, 167, 170
- Loudspeaker, 103
- MATLAB, 3, 118, 137, 147
- MCP1700, 28
- MCP23017, 74, 76, 86, 111, 222
- MCP3208, 206
- MCP3304, 35, 77–80, 111, 184, 206
- MCP4921, 52, 101
- Microcontroller, 26, 28, 30, 33, 39, 43, 53,

- 55, 57, 60, 65, 73, 74, 77, 80, 123, 153, 167, 170, 177, 211, 224
- Microstepping, 48–50, 53, 98, 100, 204
- MIDI, 40, 110, 111
- MISO, 35, 79, 80, 101, 206
- Mkocfile, 137
- MLX90614, 12, 65, 66, 112
- MOSFET, 30, 42
- MOSI, 35, 79, 101, 206
- Motivation, 223, 224
- MPU6050, 112
- MQ-x, 9, 11, 112
- MQTT, 3, 157–165, 188, 190
- Mrs. Robinson guideline, iii
- MySQL, iv, 134–138, 146, 165
- Nano text-editor, 118
- Netcat, 109, 119, 120, 125, 163
- Netmask, 121
- Network address translation, 121
- Network addressed storage, 114
- Network explained, 106
- NodeMCU, 56–58, 77–80, 105–110, 119, 123–125, 128, 130, 131, 146, 153, 154, 159–161, 165, 167, 168, 175, 177–184, 188–190, 205–221
- NPN transistor, 41, 42, 50, 87
- Octave, 118, 128–132, 134, 137, 138, 181, 182, 196, 201, 202
- Octave script
 - Capacitance measurement, 194
 - Flatfile reader, 134
 - FWHM, 203
 - Laser beam profile, 201
 - MySQL access, 137
 - Query-response, 129, 130
 - Smooth points, 202
 - Temperature logger, 130
- Open collector, 41, 42
- OpenELEC, 113
- OpenWRT, 113
- Operational amplifier, 17, 18, 20, 25, 52
- Optocoupler, 39, 40, 110
- PCA9685, 222
- Phototransistor, 15, 37, 40, 112, 175, 185–188, 205–222
- Pictures, 146, 224, 225
- Piezo buzzer, 103, 211
- Pin diode, 14, 15, 38, 183
- PIR sensor, 30, 31
- Power supply, 26–28
- Presentation, iii, 3, 115, 132, 223, 224, 226–228
- Propeller, 12, 52, 112, 222
- Pseudo-inverse, 232
- PT100, 7, 8
- Pull-up, 29, 32, 55, 61, 76, 80, 82, 84, 85, 191, 211, 221
- Pulse-width modulation, 40, 45–47, 51, 53, 85, 86, 88–90, 92, 190, 197, 198, 201, 215
- Python, 115, 123, 125–128, 132–134, 136, 138–140, 161, 171
- Python script
 - EPICS to MQTT gateway, 161
 - Flatfile writer, 132, 133
 - Geophone reader, 180
 - MySQL access, 136
 - Query-response, 125
 - Read from network socket, 128
 - Read from serial line, 127
 - Weather station reader, 170
- Query-response protocol, 3, 63, 65, 86, 90, 92–94, 100–102, 117, 124, 125, 129, 165, 194, 220
- R-2R resistor network, 51
- Raspberry Pi, iii, v, 3, 111, 113–118, 120–125, 129, 133, 134, 137, 141, 145–148, 151, 155–158, 161, 164, 170, 174, 201
- Raspbian, 114, 115, 155
- Reed switch, 29
- Relay, 42, 43
- Remote control, 32, 111, 175, 205–207, 211, 215, 216, 219–222
- Report, iii, 1, 222–226, 228
- Resistance, 5, 7–9, 16, 21, 22, 33, 34, 40, 45, 50, 53, 61, 177, 193, 229, 230
- Robot, 3, 46, 103, 204–206, 209–215, 219–222
- Rotary encoder, 29, 84
- Router, iii, 106, 113, 124
- RRDtool, 138–141, 143–145, 170–172
- RS-232, 28, 31, 36, 55, 103, 105, 110, 151
- Sailboat, 222

- SCPI, 65
- Screen program, 103–105, 117, 145
- SD card, 114, 115
- Semiconductor, 7–9, 14, 39
- Servomotor, 46–47, 52, 53, 91, 92, 112, 165, 205, 213, 216, 219
- SFH3310, 15, 185, 186
- Slides, 223–225
- SM-24, 13, 14, 177, 178
- Socket, 110, 128, 130, 131, 155, 161–163, 171, 181–183, 209
- Solar cell, 16
- Solenoid, 52
- Spell-checker, 228
- SPI, 35–36, 52, 55–57, 76–80, 101–102, 113, 205, 206, 209
- State machine, 205, 215, 216, 219, 222
- Stepper motor, 47–50, 52, 53, 92–95, 97–100, 146, 197–199, 201
- Story line, 223–225
- Strain gauge, 10, 32, 33
- Structured query language, 136
- Switch, 29–30, 39–43, 85–87
- Synaptic, 119, 146

- TCP, 130, 131, 162, 222
- Telnet, 106, 109, 119, 124, 125, 155, 156
- Temperature, 7–12, 27, 32, 33, 36, 37, 56, 61–63, 65, 67, 68, 70, 71, 73, 82, 83, 106–108, 130, 131, 133, 139, 140, 144, 145, 151, 153, 157, 158, 160, 161, 163, 165, 167, 170–172, 226
- Terminal program, 36, 63, 103, 115–120, 123–125, 133, 148, 150, 154, 155, 158, 161, 163
- Thermistor, 8, 36
- Thermocouple, 11, 12
- Thermopile, 12, 65
- Tilt switch, 29
- TIP-120, 87
- TV, 32, 53, 111, 113

- UDP, 209, 215, 219, 220
- ULN2003, 50, 53, 93
- Unix manual pages, 117, 119, 120, 134, 139, 141, 158
- USB, 58, 59, 61, 103–105, 110, 112, 113, 117, 123, 124, 129, 146, 211
- User guide, iii
- Valence band, 7, 15, 39
- Valve, 39, 52
- VNC, 120
- Voltage regulator, 28, 221
- VXI, 65

- Web server, iv, 3, 106–109, 112, 141–145
- Wheatstone bridge, 5, 7, 9, 16, 25, 33, 34, 37
- WiFi, 105–110, 113, 160, 167, 177, 188, 206, 214
- Wordnet, 228
- WPA, 57, 106, 121