Practical Contiki-NG

Programming for Wireless Sensor Networks

—

Agus Kurniawan
# Table of Contents

About the Author ........................................................................... ix

About the Technical Reviewer ......................................................... xi

Acknowledgments ........................................................................... xiii

Introduction ...................................................................................... xv

Chapter 1: Introduction to Wireless Sensor Networks ................. 1
  Introduction to Wireless Sensor Networks ........................................ 2
  Introduction to Contiki OS ............................................................... 4
  Reviewing WSN Hardware for Contiki ............................................ 6
    MICAz ......................................................................................... 7
    Mica2 ....................................................................................... 7
    TelosB ....................................................................................... 8
    Iris ............................................................................................ 9
    Custom TinyOS Motes ............................................................... 10
    Z1 Platform .............................................................................. 11
    Contiki-NG ARM-based Boards ................................................. 12
  Introducing Contiki-NG ............................................................... 13
  Set Up Development Environment for Contiki .......................... 13
    Instant Contiki ........................................................................... 13
    Manual Installation ..................................................................... 18
    Set Up Contiki-NG Development Environment ....................... 19
  Connect WSN Hardware to Computer ......................................... 22
  Contiki and Raspberry Pi ............................................................. 25
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello World Application for Contiki</td>
<td>26</td>
</tr>
<tr>
<td>Explanation</td>
<td>28</td>
</tr>
<tr>
<td>Contiki Simulator</td>
<td>35</td>
</tr>
<tr>
<td>Setting Up</td>
<td>36</td>
</tr>
<tr>
<td>Running Contiki Application</td>
<td>37</td>
</tr>
<tr>
<td>Debugging Contiki Application</td>
<td>44</td>
</tr>
<tr>
<td>Hardware Debugger</td>
<td>44</td>
</tr>
<tr>
<td>LED Indicators</td>
<td>45</td>
</tr>
<tr>
<td>Debugging Using Contiki Simulator</td>
<td>45</td>
</tr>
<tr>
<td>The Contiki printf() Function</td>
<td>45</td>
</tr>
<tr>
<td>Summary</td>
<td>46</td>
</tr>
</tbody>
</table>

## Chapter 2: Basic Contiki-NG Programming ........................................47

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contiki-NG Programming Model</td>
<td>47</td>
</tr>
<tr>
<td>Contiki-NG Basic Syntax</td>
<td>49</td>
</tr>
<tr>
<td>Creating a Project</td>
<td>49</td>
</tr>
<tr>
<td>Contiki-NG Basic Programming Language</td>
<td>50</td>
</tr>
<tr>
<td>Review Protothreads</td>
<td>52</td>
</tr>
<tr>
<td>Extending the Contiki-NG Library</td>
<td>55</td>
</tr>
<tr>
<td>Contiki-NG Demo: Threading App</td>
<td>57</td>
</tr>
<tr>
<td>Contiki-NG Coding Conventions</td>
<td>61</td>
</tr>
<tr>
<td>Demo: Build Contiki-NG Application</td>
<td>61</td>
</tr>
<tr>
<td>Summary</td>
<td>66</td>
</tr>
</tbody>
</table>

## Chapter 3: Concurrency .....................................................................67

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to Concurrency</td>
<td>67</td>
</tr>
<tr>
<td>Concurrency Approach in Contiki-NG</td>
<td>68</td>
</tr>
<tr>
<td>Introducing Contiki-NG Processes</td>
<td>69</td>
</tr>
<tr>
<td>Working with Timers</td>
<td>72</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

Clock Library ......................................................... 72
Timer Library ......................................................... 73
Stimer Library ......................................................... 74
Etimer Library ......................................................... 75
CTimer Library ......................................................... 76
Rtimer Library ......................................................... 77
Put It All Together ................................................... 78
Threading ............................................................... 83
Task Scheduling ....................................................... 88
Summary ............................................................... 94

## Chapter 4: Contiki-NG and Computer Communication .......... 95

Communication Models for Contiki-NG ................................ 96
Serial Communication ................................................. 97
Communication Between Contiki Mote and Computer ............... 97
   Access Contiki Motes via Serial Communication .................. 98
Contiki Shell .......................................................... 104
Contiki-NG Shell ....................................................... 112
Customizing Contiki Shell ........................................... 115
Communication among Contiki Motes ................................ 126
   Sending Broadcast Messages ....................................... 127
   Receiving Broadcast Messages ..................................... 128
Demo: Middleware Application ....................................... 129
Middleware Application for Contiki-NG ............................. 132
   What Is Middleware? ............................................... 133
   Middleware Architecture for Contiki-NG ......................... 133
Implementation ....................................................... 134
Testing .............................................................. 134
Summary ............................................................. 135
# TABLE OF CONTENTS

## Chapter 5: Sensing and Actuating .................................................. 137

- What Are Sensing and Actuating?..................................................... 137
- Review Sensor and Actuator Devices .............................................. 139
  - Temperature and Humidity ....................................................... 139
  - Soil Moisture ............................................................................ 141
  - Gas Sensor .................................................................................. 142
  - LED .......................................................................................... 143
  - Active Buzzer ............................................................................ 144
  - Motor ......................................................................................... 144
- Sensing in Contiki-NG ................................................................. 145
  - Demo ....................................................................................... 146
  - Creating a Project ..................................................................... 147
  - Writing a Program .................................................................... 148
  - Testing ........................................................................................ 151
- Actuating in Contiki-NG ............................................................. 152
  - Creating a Project ..................................................................... 153
  - Writing a Program .................................................................... 154
  - Testing ........................................................................................ 155
- Customizing Sensor and Actuator Devices .................................. 156
  - Expansion Connector .................................................................. 156
  - Sensor and Actuator Drivers for Contiki-NG ............................ 159
- Summary ...................................................................................... 165

## Chapter 6: Networking ................................................................. 167

- Networking in Contiki-NG .......................................................... 168
  - Network Layer ......................................................................... 170
  - MAC Layer ............................................................................... 172
  - RDC Layer ............................................................................... 173
  - Radio Layer ............................................................................... 173
TABLE OF CONTENTS

Network Simulation Using COOJA ......................................................... 174
  Creating Simulation Project ......................................................... 174
  Adding UDP Server Mote ............................................................. 177
  Adding UDP Client Motes ............................................................ 180
  Running a Simulation ................................................................. 183
IPv6 Networking ............................................................................. 185
Routing on Contiki-NG ................................................................. 189
  Introducing Basic Routing ........................................................... 189
  Single-Hop and Multi-Hop Networking ......................................... 190
  Routing on Contiki-NG ................................................................. 191
IPv6 Multicast ................................................................................. 194
Contiki-NG NullNet ......................................................................... 199
6 LoWPAN Network ................................................................. 201
  A Brief Introduction ..................................................................... 201
  Implementing a 6LoWPAN Network on Contiki-NG ..................... 203
  6 LoWPAN Implementation using COOJA .................................... 209
Build Your Own RESTful Server for Contiki-NG ......................... 214
  Preparation ................................................................................ 215
  Implementing the Demo ............................................................. 215
  Testing the Demo .................................................................... 223
Summary ....................................................................................... 224

Chapter 7: Storage ........................................................................ 225
  Storage Models in Contiki-NG .................................................... 225
  Working with Local Storage ...................................................... 227
  Coffee: File System in Contiki-NG ............................................ 231
  Demo: Contiki-NG and MySQL .................................................. 238
  Preparation ................................................................................ 239
  Design a Database .................................................................... 239
# TABLE OF CONTENTS

Build a 6LoWPAN Router ................................................................. 241  
Develop a Contiki-NG Sensor Application ..................................... 241  
Develop Middleware Application .................................................... 242  
Testing the Project ...................................................................... 246  
Summary .................................................................................... 248

## Chapter 8: Contiki-NG and Cloud Server ................................. 249  
Introduce Cloud Server ............................................................... 250  
Why Use Cloud Computing? ......................................................... 251  
Cloud-Computing Services ......................................................... 252  
Types of Cloud-Computing Deployments ..................................... 252  
Review Cloud Server Platforms .................................................. 253  
  Microsoft Azure ........................................................................ 253  
  Amazon AWS ......................................................................... 254  
  Google Cloud ......................................................................... 254  
  IBM Cloud .............................................................................. 254  
Comparing Features of All Global Clouds ..................................... 255  
Connecting Contiki-NG Motes to Cloud Servers ......................... 255  
Demo 1: Contiki-NG and Microsoft Azure .................................... 256  
  Preparation ............................................................................. 257  
  Developing Application ........................................................... 261  
  Testing Contiki-NG and Azure Application ................................. 268  
Demo 2: Contiki-NG and Amazon AWS ....................................... 271  
  Preparation ............................................................................. 272  
  Developing Application ........................................................... 282  
  Testing Contiki-NG and AWS IoT ............................................ 285  
Summary .................................................................................... 287

Index .......................................................................................... 289
About the Author

Agus Kurniawan is a lecturer, researcher, IT consultant, and author. He has 17 years of experience in various software and hardware development projects, delivering materials in training and workshops, and technical writing. He has been awarded the Microsoft Most Valuable Professional (MVP) award 14 years in a row.

His topic interests are software engineering, embedded systems, networking, and security systems. He has been working as a lecturer and researcher at the Faculty of Computer Science, Universitas Indonesia. Currently, he is pursuing a PhD in computer science at the Freie Universität in Berlin, Germany. He can be reached on his blog at http://blog.aguskurniawan.net and Twitter at @agusk2010.
About the Technical Reviewer

Chaim Krause is first and foremost a #geek. Other hashtags used to define him are (in no particular order) #autodidact, #maker, #gamer, #raver, #teacher, #adhd, #edm, #wargamer, #privacy, #liberty, #civilrights, #computers, #developer, #software, #dogs, #cats, #opensource, #technicaleditor, #author, #polymath, #polyglot, #american, #unity3d, #javascript, #smartwatch, #linux, #energydrinks, #midwesterner, #webmaster, #robots, #sciencefiction, #sciencefact, #universityofchicago, #politicalscience, and #bipolar. He can always be contacted at chaim@chaim.com and goes by the Nom de Net Tinjaw.
Acknowledgments

I would like to thank Apress for all their help in making this book possible. Specifically, I would like to thank Natalie Pao and Jessica Vakili, my coordinating editors, for helping me to stay focused and to overcome many obstacles. Without them, this book would not have been possible.

Special thanks to James Markham, my development editor, and Chaim Krause, technical reviewer, Nirmal Selvaraj, Production Editor, C. Dulcy Nirmala, Senior Executive Project Manager for all his suggestions during the editorial review process to help make this a great book.

I would also like to thank the Contiki and Contiki-NG communities anywhere in the world for contributing and making learning and developing Contiki-NG easy.

Last, but not least, a thank you to my wife, Ela, and my children, Thariq and Zahra, for their great support while I completed this book.
Introduction

Wireless Sensor Networks (WSN) are one of the research and technology topics for which researchers, developers, and makers develop applications with specific purposes. Contiki-NG is one of the WSN platforms used to build WSN programs supported by various hardware platforms. This book is designed for developers and researchers who want to build Contiki-NG programs for general and specific purposes.

For the Readers

This book assumes you have some programming experience. It is also written for someone who has developed programs using C/C++ and wants to develop a Wireless Sensor Network (WSN) application with the Contiki-NG platform.

How This Book Is Organized

This book is designed with a step-by-step approach. You will learn how to develop a WSN with the Contiki-NG platform. You will also explore some Contiki-NG libraries and APIs to implement certain scenarios.

You will learn how to develop a Contiki-NG program. This book explains how Contiki-NG performs sensing and actuating. You will also see how to communicate with other Contiki-NG platforms and external systems, such as cloud servers.
Required Software, Materials, and Equipment

In general, you need a computer with Linux, Windows, or Mac OS installed. Linux is recommended. You should install all toolchains and Contiki-NG project codes on your computer.

We need Contiki-NG mote hardware to implement our demo. This book uses Telosb/sky and TI LaunchPad CC2650 boards for testing.
CHAPTER 1

Introduction to Wireless Sensor Networks

Wireless Sensor Networks (WSN) are a research and technology topic for which researchers, developers, and makers develop applications for specific purposes. In this chapter, we will learn and explore what a WSN is and try to develop one using the WSN platform Contiki and its update, Contiki-NG.

The following is a list of topics we will cover in this chapter:

- Introduce Wireless Sensor Networks.
- Introduce Contiki OS.
- Explore WSN hardware and platform.
- Introduce Contiki-NG project.
- Set up Contiki-NG development environment.
- Build a simple Contiki application.
- Work with Contiki simulator.
- Debug Contiki application.
Introduction to Wireless Sensor Networks

A Wireless Sensor Network (WSN) is a board system with the connectivity capability to sense data and/or to perform actions. Sometimes the WSN board is called a WSN mote. The main objective of implementing a WSN mote is to capture physical objects in digital form and then transfer them to a certain server. Research on WSN topics is an intense study area since there are a lot of problems in need of solving, such as mote hardware design, networking, infrastructure, and security.

Nowadays, hardware manufacturers grow up fast. You will find that there are a lot of new boards on the market, such as Arduino, Raspberry Pi, BeagleBone, Intel Edison, NodeMCU, Teensy, Tessel, and so on. This is the era of the Internet of Things (IoT). It’s estimated that there are billions of IoT devices connected to the Internet, based on Gatner’s report. Since IoT board demand is high, the board price could be quite cheap. Furthermore, the open source hardware movement has had an impact on the growing board industry. People can design and make their own boards for special purposes.

Back to our WSN mote topic—how to describe a WSN mote? In general, a WSN mote consists of a microcontroller (MCU), sensor/actuator, and wireless module. You can see it in Figure 1-1. The MCU is the center of processing in a WSN mote. It has a responsibility to ensure the system runs well. In other designs, the MCU can be replaced by an MPU (microprocessor), depending on whether there is a battery issue or not. The second part is the sensor/actuator. A sensor can capture physical objects, such as temperature, humidity, and compass direction. An actuator can perform a certain action, such as lighting an LED, generating sound, or running a motor. Some WSN motes may provide sensor devices only, but other WSN motes may use both a sensor and an actuator inside the board. Again, it depends on your design. The last part is the wireless module. It’s used to transfer data from the mote to a network device; for instance, gateway, server application, or cloud application. Selecting the
wireless module type will have an impact on what protocol will be used to communicate with other motes and servers. Most WSN motes use IEEE 802.15.4 as their network protocol.

![General model of Wireless Sensor Network (WSN) mote](image)

**Figure 1-1.** General model of Wireless Sensor Network (WSN) mote

Sometimes makers will build WSN devices in several forms based on their roles. A WSN mote may only consist of an MCU and a radio module, without sensor and actuator devices. Some WSN motes will provide an MCU, sensor/actuator devices, and a radio module.

How can the WSN mote reach the server? This is a common issue in WSN implementation. If a WSN mote has the same protocol as the servers on the network module, it can communicate with the server directly. This method may take more battery usage in the mote since most servers use advanced protocol without battery issues. Alternatively, we can use a gateway, which is used as a bridge between WSN motes and servers. Some gateways have the capability to translate the WSN mote’s protocol to the target server’s protocol so they can exchange data. The gateway can be implemented to monitor the existing WSN motes. You can see how WSN motes communicate with another system through a gateway in Figure 1-2.
Introduction to Contiki OS

Contiki is a WSN platform that provides software and hardware. Contiki was created by Adam Dunkels in 2002. Now, the Contiki project involves both companies and contributors. This project has released open source software and hardware. The operating system (OS) in Contiki uses Protothread, which combines multithreading and event-driven programming. On the hardware side, the Contiki project provides hardware schemes so that we can build our own Contiki boards. You can reach the official website for Contiki at http://www.contiki-os.org.

The programming model of the Contiki platform implements a preemptive multithreading architecture and an event-driven model. The Contiki programming language uses C syntax for writing programs.

Contiki provides hardware abstractions that encapsulate hardware complexity. This approach makes Contiki work with various hardware, including MCUs and radio modules. General libraries for sensing, actuating, and communication are also provided by Contiki. Users should get more attention on their problems. You can see the general architecture of Contiki in Figure 1-3.

Figure 1-2. A connectivity model for several WSN motes
Kernel, the program loader, the language run-time, and the communication service are static modules within the ROM of Contiki OS. All user programs will be loaded into Loaded Program. Only the kernel and the communication service will be used by the Contiki OS RAM.

Contiki uses a GCC compiler to compile C source code files. We develop Contiki applications written in *.c files. After they are compiled, we obtain the binary file. Basically, it converts the application from C program syntax to a native binary file for a specific hardware target. We also can run a C program on the Contiki simulator to verify program behaviors. You can see the flow of programming in Figure 1-4.

**Figure 1-3. Contiki general architecture**
To run Contiki on top of hardware, that hardware needs to fulfill some requirements, especially about the MCU and network module. In this section, we will explore various WSN hardware types that we can use to implement Contiki. A box that consists of a WSN board that has some sensors or actuators is called a WSN mote. If you have a TinyOS mote, you can use that mote to run a Contiki application.

In general, a list of supported Contiki OS can be found on the official website at this link: http://www.contiki-os.org/hardware.html. For Contiki-NG, you can see a list of Contiki-NG boards at https://github.com/contiki-ng/contiki-ng/wiki#the-contiki-ng-platforms. We will next review some famous WSN mote models that you can use for experimental purposes.
MICAz

MICAz is a mote from Crossbow Technology, MEMSIC. This mote uses an ATmega128L microcontroller and the CC2420 radio chip. ATmega128L is an 8-bit microcontroller from Atmel. This MCU has capabilities such as 128K of in-system, self-programmable flash program memory, 4K of EEPROM, and 4K of internal SRAM. You can explore this MCU by reading its datasheet at http://www.atmel.com/images/doc2467.pdf.

On the radio side, this mote uses a CC2420 chip that implements IEEE 802.15.4 protocol. This chip has energy-saving capabilities. You can work in sleep mode on a network layer stack. You can read more information about MICAz at http://www.memsic.com/wireless-sensor-networks/. You can see a MICAz board form in Figure 1-5 (source: http://www.memsic.com/wireless-sensor-networks/).

Figure 1-5. MICAz mote model

Mica2

Mica2 is similar to the MICAz mote in that the mote uses an Atmel ATmega128L microcontroller; it uses the CC1000 radio chip for the wireless module. The battery connector is also provided to work
standalone. You can see the Mica2 mote in Figure 1-6 (source: http://www.snm.ethz.ch/snmwiki/pub/uploads/Projects/mica2.jpg). To develop a program for this mote, you should use a development board to flash the program.

Figure 1-6. Mica2 mote model

TelosB

TelosB is the famous model that researchers and makers use for TinyOS implementation. This mote also can be used for the Contiki platform. TelosB motes use an MSP430 microcontroller from Texas Instruments (TI). The MSP340 series in TelosB are built from MSP430x15x, MSP430x16x, and MSP430x161x. Many manufacturers build WSN motes based on the TelosB design. You can see a TelosB mote in Figure 1-7 (source: https://www.advanticsys.com/shop/mtmcm5000msp-p-14.html).
Iris

Iris is a WSN mote from Crossbow. It's built from an ATmega1281 microcontroller and the Atmel AT86RF230 radio chip. Atmel ATmega1281 has a flash memory that is about 128K and 8K of RAM so you can write programs in more spaces. You can see Iris mote in this site, http://www.memsic.com/wireless-sensor-networks/. In Figure 1-8, you can see my Iris mote from Crossbow, which is connected to my notebook.

Figure 1-7. MTM-CM5000-MSP board based on TelosB from Advanticsys
Custom TinyOS Motes

Independent makers or manufacturers can build their own TinyOS motes, including sensors with specific purposes. The scheme and layout of the TinyOS mote have already been shared so it's not difficult to build your own.

BTnode is a mote based on TinyOS and uses an Atmel ATmega 128K MCU and Chipcon CC1000 radio module. This mote was developed by ETH Zurich. Currently, they have released BTnode revision 3. You can see it in Figure 1-9. Further information about BTnode can be found at http://www.btnode.ethz.ch/.
For more information, you can get a list of compatible TinyOS and Contiki motes at https://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes. You can review some WSN motes for your own development.

**Z1 Platform**

The Z1 platform is a general-purpose development platform for WSN. This board uses an MSP430F2617 low-power microcontroller. The Z1 platform radio modules use a CC2420 transceiver and are IEEE 802.15.4 compliant, which operates at 2.4GHz with an effective data rate of 250Kbps. For further information, you can visit https://zolertia.io/. You can see a form of Z1 platform in Figure 1-10.
Currently, Contiki-NG offers support for boards with ARM MCU. Based on the Contiki-NG document at https://github.com/contiki-ng/contiki-ng/wiki#the-contiki-ng-platforms, we can use several boards with ARM MCU to develop Contiki-NG applications. The following is the list of supported Contiki-NG boards:

- cc2538dk: TI cc2538 development kit
- jn516x: NXP jn516x series
- nrf52dk: Nordic Semiconductor nRF52 development kit
- openmote-cc2538: OpenMote cc2538
- srf06-cc26xx: TI cc26xx and cc13xx platforms
- zoul: Zolertia Zoul platforms: Firefly, RE-mote, and Orion

In this book, I use five board models: Telosb, TI CC2650 LaunchPad, TI CC2650 Sensortag, TI CC1350 LaunchPad, and TI CC1350 Sensortag.
Introducing Contiki-NG

Contiki-NG is a new version of the Contiki project. Contiki-NG provides an RFC-compliant, low-power IPv6 communication stack, enabling Internet connectivity. If you are working with Contiki, I recommend using the latest version—Contiki-NG. This project can be found at https://github.com/contiki-ng/contiki-ng.

One advantage of Contiki-NG is that it supports many hardware platforms, including MCU-based ARM. You can bring your favorite boards to build applications for Contiki-NG. This book will cover Contiki-NG for developing and implementing.

Set Up Development Environment for Contiki

To build a Contiki application, we need to prepare our development environment. Currently, the Contiki development environment can be deployed on the Linux platform. There are two methods: instant Contiki and manual installation. Then, we will set up a Contiki-NG development environment.

We will review deployment for both Contiki and Contiki-NG in the next section.

Instant Contiki

Contiki provides a complete development environment under Ubuntu Linux that is available in virtual-machine form. This approach is easy and low risk without breaking your current OS.
Follow these steps:

- Download VMWare Workstation player (Free) or VMWare Workstation (not free). For VMWare player, you can download it from http://www.vmware.com/go/downloadplayer/. You can also use Virtualbox instead of VMWare.
- After downloading Instant Contiki, you can extract it to a specific folder. You should see some files as shown in Figure 1-11.

![Instant Contiki files](image)

**Figure 1-11. Extracted instant Contiki files**

VMWare Workstation Player is free. It is available for Windows and Linux. For Mac users, there is no free VMWare Workstation Player. You can use VMWare Fusion. I have installed VMWare Workstation Player on Windows 10. You can see it in Figure 1-12.
Now, you can open Instant Contiki using VMWare Workstation Player by clicking **Open a Virtual Machine**. Navigate to the *.vmdk file in the folder to which Instant Contiki file was extracted.

After this succeeds, you can see Instant Contiki on VMWare Workstation Player, as shown in Figure 1-13.

**Figure 1-12. VMWare Workstation player 12 on Windows 10**
By default, Instant Contiki is configured to use 1GB RAM. You can customize this by clicking *Edit virtual machine settings*. You should get a dialog as shown in Figure 1-14.
Since my computer has 16GB of RAM, I set my Instant Contiki with 4GB of RAM. If you are done, click the OK button to save and close the dialog.

Now, you can run Instant Contiki by clicking **Play virtual machine**; you should see the Ubuntu desktop. You can see it in Figure 1-15.

If your WSN mote has an MSP430-based MCU such as Sky and Telosb, you should install the ggc-msp430 library in order to develop your Contiki application. You can type this command in Ubuntu Terminal:

```
$ sudo apt-get install gcc-msp430
```

Your virtual machine is ready for Contiki development.
Troubleshooting

If you are running VMWare Workstation Player on Windows 8.1/10 or later, you may get problems due to conflicts with Hyper-V. Since VMWare Workstation and Hyper-V cannot run together on one computer, you should uninstall and disable Hyper-V on Windows 8.1/10.

From my experience, I use Windows 10 with installed Hyper-V. When I use the VMWare application, I disable my Hyper-V. You do so using command prompts with the Administrator level. Type this command:

$ bcdedit /set hypervisorlaunchtype off

Then, you can restart Windows.

If you want to reenable Hyper-V, you can type this command:

$ bcdedit /set hypervisorlaunchtype on

Manual Installation

If you have a computer with installed Linux, you can install the Contiki development environment on your platform.
First, you should install all required libraries to run Contiki OS. You can type these commands in Terminal:

$ sudo apt-get update
$ sudo apt-get install build-essential
$ sudo apt-get install git
$ sudo apt-get install binutils-msp430 gcc-msp430 msp430-libc msp430mcu mspdebug gcc-arm-none-eabi gdb-arm-none-eabi

You also need the Java run-time and SDK to run the Contiki OS simulator. In this case, we use Open JDK. You can use Java from Oracle. Type this command:

$ sudo apt-get install openjdk-8-jdk openjdk-8-jre ant libncurses5-dev

Since Contiki OS runs a 32-bit environment, if you have Linux OS with 64-bit, you can install these libraries:

$ sudo apt-get install lib32ncurses5

Now, you can download the Contiki OS source code. Open Terminal and navigate to the specific folder where Contiki OS files will be extracted:

$ git clone https://github.com/contiki-os/contiki

Your computer is ready for Contiki OS development.

Set Up Contiki-NG Development Environment

Currently, Contiki-NG does not provide an Instant Contiki, so it must be installed manually. The installation process can be found here: https://github.com/contiki-ng/contiki-ng/wiki/Toolchain-installation.
In general, setting up Contiki-NG can be done with the following steps:

- Install required libraries.
- Install compiler and its dependences for specific hardware platform.
- Download and configure Contiki-NG.

The first step to deploy Contiki-NG is to install all required libraries for development. In this section, I use Ubuntu Linux for development testing. I use Ubuntu LTS 16.04 with x64 platform. Type these commands on Linux Terminal to install libraries:

```
$ sudo apt update
$ sudo apt install build-essential doxygen git curl wireshark python-serial
```

While you are installing Wireshark, you should enable the feature that allows non-superuser capture packets (select “yes”). Add your account into the wireshark group. You can type this command, substituting <user> with your Linux account:

```
$ sudo usermod -a -G wireshark <user>
```

The next step is to install the compiler and its dependencies, based on the Contiki mote hardware. If you have a Contiki mote based on MSP430, you can install the compiler and libraries for MSP430. You can type this command:

```
$ sudo apt install gcc-msp430
```


For Contiki mote–based ARMs such as CC2538DK and Zoul, you can install the ARM compiler by typing these commands:
We also install JDK and Ant for the Contiki simulator, COOJA, that are built from Java. You can install these libraries by typing this command:

```
$ sudo apt install default-jdk ant
```

If you have several Java versions within Linux, you should set your Java preference. It is done by calling `update-alternatives`:

```
$ update-alternatives --config java
```

Cooja needs to have the environment variable `JAVA_HOME` set to work with native Contiki motes. You should set `JAVA_HOME` with the path found above in the `.profile` file:

```
$ echo 'export JAVA_HOME="/usr/lib/jvm/default-java"'> ~/.profile
```

Most Contiki motes are attached to the computer via USB. When we access them, we probably need administrator privileges. To be able to access the USB without using the administrator level, your account should be part of the groups `plugdev` and `dialout`. You add them by typing these commands. Be sure to change `<user>` to your Linux account:

```
$ sudo usermod -a -G plugdev <user>
$ sudo usermod -a -G dialout <user>
```

This is the case for an x64 platform. You probably will get an error when connecting via serial communication on a serial tool from Contiki-NG. You can install the following required libraries:

```
$ sudo apt-get update
$ sudo apt-get install lib32ncurses5 lib32z1
```
After finishing all tasks, you can reboot your computer.

Now, you can download Contiki-NG by cloning using git. Perform the following commands in Terminal:

```
$ git clone https://github.com/contiki-ng/contiki-ng.git
$ cd contiki-ng/
$ git submodule update --init --recursive
```

Now, you are ready to develop Contiki-NG.

**Connect WSN Hardware to Computer**

After all required libraries for Contiki are installed, you can connect the WSN hardware to the computer. Depending on the WSN hardware model, your WSN mote should be recognized by the computer. Some WSN hardware may need a hardware driver so your computer detects it. You can see my WSN mote, TelosB, connected to the computer in Figure 1-16.

After connecting to the WSN mote, you can verify whether the WSN mote is recognized by the computer. Type this command:

```
$ ls /dev/ttyUSB*
```

If your mote isn’t recognized by the computer, it’s probably detected as /dev/ttyACM*.
This program will query all connected WSN motes on your computer. If found, it will display a list of serial ports from WSN motes. You can see my WSN mote in Figure 1-17.

**Figure 1-16.** TelosB mote is connected to computer through USB cable

This program will query all connected WSN motes on your computer. If found, it will display a list of serial ports from WSN motes. You can see my WSN mote in Figure 1-17.

**Figure 1-17.** WSN mote is detected on /dev/ttyUSB0
If you use Instant Contiki, your WSN mote may be detected on Host OS. For instance, I use Windows 10 to run Instant Contiki. The detected WSN mote can be viewed on Device Manager, shown in Figure 1-18.

To connect a WSN mote from the host computer to a virtual machine on VMWare, you should transfer to a virtual machine. Since the hardware USB can only attach to one computer, the host computer can’t access the WSN mote while it is being used by Ubuntu on the virtual machine.

Figure 1-18. WSN mote is detected by Device Manager

How to transfer a WSN mote from the host to a virtual machine? You can click menu Player ➤ Removeable Devices ➤ <WSN_mote_name> ➤ Connect. You can see it in Figure 1-19.
After connecting, you can perform Contiki development as usual on Ubuntu. You can verify the connected mote using Terminal.

Contiki and Raspberry Pi

Raspberry Pi is a very small computer. This board can run several operating systems. There are many Raspberry Pi models that you can use for development. The official OS is Raspbian OS, which is based on Debian Linux.

To deploy Contiki on Raspberry Pi, I recommend you use Raspbian OS for your board. Then, you can install Contiki manually without Instant Contiki since Raspberry Pi has limited RAM. Raspberry Pi has USB connectors so our WSN mote can be attached to the board. You can see my TelosB that is attached to a Raspberry Pi in Figure 1-20.
Hello World Application for Contiki

In this section, we will learn to build the first program for Contiki. We use the existing program sample from Contiki or Contiki-NG. It’s a hello-world program, which is located in the `<contiki_folder>/examples/hello-world` folder. The Hello World program displays simple words—“Hello, world”—in the WSN mote’s Terminal. This program consists of `hello-world.c` and `Makefile` files. After the program is compiled, you should see several files, such as `*.obj` and `*.hex` files.

To write a C program, use your favorite editor, such as vi, vim, and nano. Also, use a visual editor; for instance, Eclipse, Sublime Text, and Visual Studio Code. I use Visual Studio from Microsoft. You can download it on [https://code.visualstudio.com](https://code.visualstudio.com). It is available for Linux, Mac, and Windows. Figure 1-21 shows Visual Studio Code to open the Blink application.
To start to write a Contiki program from scratch, you create a folder called hello-world. Then, you create hello-world.c within the hello-world folder. You also can run a program sample, hello-world, from <contiki_folder>/examples/hello-world folder. Navigate to your Terminal and then build that program.

If you want to build a project from scratch, you must continue your development. The following is the complete code for the hello-world.c file.

```c
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*-----------------------------------------------*/
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/*-----------------------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data)
{
    PROCESS_BEGIN();

    printf("Hello, world\n");

    PROCESS_END();
}
```
Explanation

This program declares a Contiki process thread, called hello_world_process. To print message to Terminal, use the printf() function:

```c
PROCESS_THREAD(hello_world_process, ev, data)
{
    PROCESS_BEGIN();
    printf("Hello, world\n");
    PROCESS_END();
}
```

hello_world_process will be executed automatically using AUTOSTART_PROCESSES().

```c
PROCESS(hello_world_process, "Hello world process");
```
AUTOSTART_PROCESSES(&hello_world_process);

Last, you should make a Makefile file. The following is the content of the Makefile file:

```plaintext
CONTIKI_PROJECT = hello-world
all: $(CONTIKI_PROJECT)

CONTIKI = /home/user/Documents/book/contiki
include $(CONTIKI)/Makefile.include
```

Three items are required in our Makefile file, which are described as follows:

- **CONTIKI_PROJECT** is used to declare our project name.
- **CONTIKI** is a Contiki root directory where Contiki libraries are.
- We include $(CONTIKI)/Makefile.include file in our program.

Change CONTIKI for your Contiki root folder where the Contiki source code files, https://github.com/contiki-os/contiki or https://github.com/contiki-ng/ for Contiki-NG, are located.

If all required program setup is done, we can compile C program using make by passing in the WSN platform. For our demo, we use the native app as the target, so we pass native while compiling:

```bash
$ make TARGET=native
```

This compiling generates a `<project_name>`.native binary file. You can see it in Figure 1-22. A hello-world.native file is generated.
To run the Contiki app, you can type this command:

```
$ ./hello-world.native
```

If you get an error due to security issues, you probably need to run this program with administrator privilege. You can type this command:

```
$ sudo ./hello-world.native
```

It shows THE IPv6 address and displays “Hello, world” in Terminal. You can see it in Figure 1-23(a) for Contiki app and (b) for Contiki-NG app. To stop your Contiki program, you can press CTRL+C.
Figure 1-23. Running Contiki application as native, (a) Contiki app and (b) Contiki-NG app
Next, we deploy the Contiki application to the Contiki hardware. For instance, I use TelosB as my WSN mote target. TelosB is a Sky platform from Contiki because TelosB uses MSP430 MCU.

Before deploying the Contiki program to the Contiki mote, make sure your Contiki mote is already attached to your computer. For instance, my Contiki mote, TelosB, is detected as /dev/ttyUSB0. You can verify the attached Contiki mote using the following command:

```bash
$ ls /dev/ttyUSB*
```

Now, you can compile and save the target platform by typing this command:

```bash
$ make TARGET=sky
$ make TARGET=sky savetarget
```

A sample of the compiling output is shown in Figure 1-24.

![Flashing Contiki program to Sky platform on TelosB hardware](image)

**Figure 1-24.** Flashing Contiki program to Sky platform on TelosB hardware
This compiling will generate a `<project_name>.<platform>` file. In my case, this will generate a `hello-world.sky.ihex` file:

```
$ make hello-world.upload
```

If you get a permission error, you can run it using the administrator level:

```
$ sudo make hello-world.upload
```

This program will be flashed to Contiki hardware. It may take several minutes.

For Contiki-NG motes from Texas Instruments, you will probably get errors while uploading a program to the board. You can use SmartRF Flash Programmer. You can download it from [http://www.ti.com/tool/FLASH PROGRAMMER](http://www.ti.com/tool/FLASH PROGRAMMER). Unfortunately, this tool currently only runs on Windows platforms.

To see the program output from the Contiki hardware, we listen to incoming messages from the serial port of the Contiki hardware. You can type this command:

```
$ sudo make login
```

You also can specify a serial port of the Contiki hardware. For instance, Contiki hardware on serial port `/dev/ttyUSB0`. You can type this command:

```
$ sudo make MOTES=/dev/ttyUSB0 login
```

You can see the running Contiki program in Figure 1-25.
You may not get any message from the hello-world program. Since this program runs the first process, we may miss this process. Try to reset the Contiki hardware so Contiki OS will reboot. Then, run the program.

It displays “Hello, world” in the listening program. You can see it in Figure 1-26. To stop your Contiki listening program, you can press CTRL+C.
If you don’t have a WSN mote on which to run the Contiki program, you can use the Contiki simulator, COOJA. Although COOJA has limitations, it’s still useful for reviewing and debugging. COOJA is a part of the Contiki tools. Its location is `<contiki_root>/tools/cooja`; see Figure 1-27. Further information about COOJA can be found at https://github.com/contiki-os/contiki/wiki/An-Introduction-to-Cooja.

In this section, we will set up COOJA. Then, we will run a simple Contiki program on COOJA.

Let’s start.

**Figure 1-26. Getting messages from Contiki hardware**
Before you run COOJA, you should configure your computer to update and check missing libraries for the COOJA application. Your computer should connect to the Internet.

You can configure COOJA by typing these commands:

```bash
$ cd /<contiki-ng_path>/tools/cooja
$ git submodule update --init
```

If this succeeds, you can run the COOJA application by typing this command:

```bash
$ ant run
```

You should get the COOJA application, shown in Figure 1-28.
After you launch the COOJA application, you can run the Contiki application using COOJA. First, create a new simulation on COOJA. You can click File ➤ New simulation. You can see it in Figure 1-29.
Figure 1-29. Add a new simulation for Contiki

You should get the dialog shown in Figure 1-30. Fill in the simulation name. If done, click the Create button.

Figure 1-30. Filling in simulation name
Now, you will get the Contiki simulator editor, COOJA, that is shown in Figure 1-31.

![Contiki simulator with Cooja](image)

**Figure 1-31.** *Contiki simulator with Cooja*

The next step is to add the WSN motes on the simulator. You can do it by clicking menu Motes ➤ Add motes ➤ Mote platform (see Figure 1-32). Please select your Mote platform preference. For demo, I use Sky platform.
Then, you will get the dialog shown in Figure 1-33. In the Contiki process/Firmware field, select your Contiki program. For demo purposes, we use the hello-world application. Select the hello-world.c file.

**Figure 1-32. Adding WSN motes**

**Figure 1-33. Select Contiki program for running**
Compile the Contiki program by clicking the Compile button. You can see the sample of compilation output in Figure 1-34.

![Figure 1-34. Compiling Contiki program on COOJA](image)

After compiling has completed, you will be asked to enter a number of WSN motes with their positions. For demo purposes, I fill in one WSN mote, as in Figure 1-35.

![Figure 1-35. Filling in number of WSN motes](image)
If done, click the Add motes button. Now you can see your motes in the network map. For instance, you can see my WSN mote in Figure 1-36.

Figure 1-36. A WSN mote on COOJA Contiki simulator

To run the simulation, you can click the Start button on the Simulation Control dialog. See it in Figure 1-37.
Figure 1-37. Running Contiki simulator by clicking Start button

After the Start button has been clicked, the Contiki program will run on COOJA. You should see the message “Hello, world” on the output panel. You can see it in Figure 1-38.

Figure 1-38. Running hello-world Contiki program on COOJA
Debugging Contiki Application

Sometimes you want to trace your program after the program has been deployed into a WSN mote. In another scenario, you may want to investigate your Contiki program before deploying it to a WSN mote. Debugging is one solution for investigating your Contiki program.

In this section, you will learn various debugging methods with which to check your program. You can choose the best method that fits your case.

Hardware Debugger

Most WSN motes do not provide built-in hardware debuggers, so if you want to debug a Contiki-NG program through the hardware approach, you need additional hardware. Regarding what WSN mote model you want to perform debugging on, you should check your MCU model.

For Atmel MCU on MICAz, Mica2, and Iris platforms, you need JTAG Atmel. Otherwise, for MCU-based MSP430, you use MSP430 USB Debugging Interface from Texas Instruments (TI). This tool is shown in Figure 1-39 (source: http://www.ti.com/tool/msp-fet). You can buy it on this site: http://www.ti.com/tool/msp-fet.

Figure 1-39. MSP430 USB debugging interface
Hardware debuggers usually have capabilities in a remote debugger. For professional purposes, I recommend you use this tool.

**LED Indicators**

In general, a WSN mote is equipped with several LEDs. We can use these LEDs to indicate a specific task; for instance, we turn on a specific LED to inform us that the WSN mote performs sensing. You can find an LED library in the `<contiki>/core/dev/leds.h` header file. In the next chapter, we will try to develop a Contiki application utilizing GPIO (General Purpose Input/Output).

Based on my experience, this approach is easy to implement. The downside of this method is having no more information; for instance, you might want to get a certain value from a specific task on your program.

**Debugging Using Contiki Simulator**

With the third approach, you can use the Contiki simulator, COOJA. You can compile and attach a Contiki program to the Contiki simulator. You can see messages that are generated using the `printf()` method. For instance, the hello-world application prints messages using `printf()`.

**The Contiki printf() Function**

The last method that I like is using the `printf()` function. The idea is that our program writes messages using `printf()`, and then the messages will pass through to the serial port. To listen to incoming messages from the serial port, we can use the login command on Terminal.

This approach is easy and low cost for debugging your Contiki application. Just place a `printf()` function on the code you are investigating.
Summary

We have learned what Wireless Sensor Networks (WSN) and Contiki are. We also have set up Contiki and Contiki-NG development environments and run a sample program to the WSN mote and a simulator. Last, we learned how to debug the Contiki program.

In the next chapter, we will focus on the Contiki-NG programming language. We will learn how to build a Contiki-NG program and run it on Contiki-NG hardware and a simulator.
CHAPTER 2

Basic Contiki-NG Programming

Contiki-NG uses the C programming language to develop applications for WSN motes. In this chapter, you will learn this basic language for creating Contiki-NG programs. Program samples are provided to accelerate your learning speed.

The following is a list of topics that will be covered:

- Contiki-NG programming model
- Contiki-NG basic syntax
- reviewing protothreads
- extending the Contiki-NG library
- Contiki-NG coding conventions
- showing a demo for building a Contiki-NG application

Contiki-NG Programming Model

Contiki-NG uses the C programming language, which is component-driven. You build some components and then connect each component. Most WSN motes work in sleep mode. If there is any task to be executed, the program will perform the task through hardware interrupts. If the task is completed, the Contiki-NG program will go back to sleep mode.
Contiki-NG programming applies protothreads. In general, you can develop Contiki-NG applications using the approach shown in Figure 2-1. Start by creating a project. Then, create a C program and apply the protothreads approach for developing a Contiki-NG program. When finished writing the program, compile and upload it to the WSN mote.

**Figure 2-1. Programming flow of a Contiki-NG program**

If you open the source code of Contiki-NG OS, you can see several folders, shown in Figure 2-2. Information about these folders can be read in Table 2-1.

**Figure 2-2. Contiki-NG program structure**
In the next section, we will review the basics of C programming, such as the language syntax used to develop Contiki-NG programs.

## Contiki-NG Basic Syntax

In general, Contiki-NG adopts the C programming language. In this section, we will review some C programming language basics. You can use any text editor to write C code.

### Creating a Project

The Contiki-NG program does not provide project templates to build a program. If you want to create a new project, you start by creating a new folder and a Makefile file. Figure 2-3 shows a collection of Contiki-NG project samples.
You also can use program samples from the Contiki project
(https://github.com/contiki-os/contiki). You should check if the one
you choose is compatible with the Contiki-NG project (https://github.com/
contiki-ng/contiki-ng).

**Contiki-NG Basic Programming Language**

Contiki-NG adopts the C general programming language for
implementation. If you have experience in C programming, it won’t
require much effort to learn Contiki-NG programming. Contiki-NG
programming is similar to C as far as declaring variables and using
conditional statements and looping. You can see samples of the C
programming language in Table 2-2.
To improve your skills in the Contiki-NG programming language, I recommend you practice writing Contiki-NG programs.
Review Protothreads

Protothreads are lightweight threads designed for memory-constrained systems, such as small embedded systems or WSN nodes. You can see a protothreads implementation for Contiki-NG at `<contiki_root>/os/sys/pt.h`.

The following is the content of the `pt.h` file:

```c
#ifndef PT_H_
#define PT_H_
#include "sys/lc.h"

struct pt {
    lc_t lc;
};

#define PT_WAITING 0
#define PT_YIELDED 1
#define PT_EXITED  2
#define PT_ENDED   3

#define PT_INIT(pt)   LC_INIT((pt)->lc)
#define PT_THREAD(name_args) char name_args
#define PT_BEGIN(pt) { char PT_YIELD_FLAG = 1; if (PT_YIELD_FLAG) {;} LC_RESUME((pt)->lc)
#define PT_END(pt) LC_END((pt)->lc); PT_YIELD_FLAG = 0; \PT_INIT(pt); return PT_ENDED; }
#define PT_WAIT_UNTIL(pt, condition)          \
do {                                        \
    LC_SET((pt)->lc);                        \
    if(!(condition)) {                       \
        return PT_WAITING;                   \
    }                                        \
}
```

Chapter 2  BasiC Contiki-ng proGramminG
while(0)
#define PT_WAIT_WHILE(pt, cond) PT_WAIT_UNTIL((pt), !(cond))
#define PT_WAIT_THREAD(pt, thread) PT_WAIT_WHILE((pt), PT_SCHEDULE(thread))
#define PT_SPAWN(pt, child, thread) do {
    PT_INIT((child));
    PT_WAIT_THREAD((pt), (thread));
} while(0)
#define PT_RESTART(pt) do {
    PT_INIT(pt);
    return PT_WAITING;
} while(0)
#define PT_EXIT(pt) do {
    PT_INIT(pt);
    return PT_EXITED;
} while(0)
#define PT_SCHEDULE(f) ((f) < PT_EXITED)
#define PT_YIELD(pt) do {
    PT_YIELD_FLAG = 0;
    LC_SET((pt)->lc);
    if(PT_YIELD_FLAG == 0) {
        return PT_YIELDED;
    }
} while(0)
The following is a list of function descriptions based on the `pt.h` header:

- **PT_INIT(pt)** function is used to initialize a protothread.
- **PT_THREAD(name_args)** is a macro that is used to declare a protothread.
- **PT_BEGIN(pt)** is used to declare the starting point of a protothread.
- **PT_END(pt)** is used to end a protothread.
- **PT_WAIT_UNTIL(pt, condition)** is used for blocking the protothread until the specified condition is true.
- **PT_WAIT_WHILE(pt, cond)** is used for blocking and waiting while the condition is true.
- **PT_WAIT_THREAD(pt, thread)** is used to schedule a child protothread. The current protothread will block until the child protothread completes.
- **PT_SPAWN(pt, child, thread)** is used to spawn a child protothread and waits until it exits.
• PT_RESTART(pt) will block and cause the running protothread to restart its execution at the place of the PT_BEGIN() call.

• PT_EXIT(pt) is used to exit from a protothread.

• PT_SCHEDULE(f) is used to schedule a protothread. The return value of the function is non-zero if the protothread is running or zero if the protothread has exited.

• PT_YIELD(pt) yields from the current protothread.

• PT_YIELD_UNTIL(pt, cond) yields from the protothread until a condition occurs.

Extending the Contiki-NG Library

Contiki-NG OS consists of several libraries and apps that you can extend based on your case. To extend Contiki-NG OS functionalities, you can follow the same approach used while building C programs.

You add additional libraries by adding a C header and source code. Our Contiki-NG program will consume our libraries. For this demo, we create a header file called mycounter.h with the following code:

```c
#ifndef MYCOUNTER_H
#define MYCOUNTER_H

int next_counter(int current);

#endif
```
CHAPTER 2  BASIC CONTIKI-NG PROGRAMMING

This C header provides one function, next_counter(). This function will be implemented into the mycounter.c file. We perform a counter increment with a maximum value of 99. You can write the complete code as follows:

```c
#include "mycounter.h"

int next_counter(int current)
{
    if(current>99)
        current = 1;
    else
        current++;
    return current;
}
```

To access the extended library, we declare that header file. Then, we call library functions from our code. The following is a skeleton code sample to access the header file in a C program:

```c
#include "mycounter.h"

...

static int counter = 0;

counter = next_counter(counter);
printf("Counter: %d\n", counter);
```

Next, we will implement our extended library and use it in a Contiki-NG program.
Contiki-NG Demo: Threading App

Now, we will try to build a Contiki-NG program that uses a header file. We create a folder called demo-counter. Then, we create several files as follows:

- mycounter.h
- mycounter.c
- demo-counter.c
- Makefile

You can see the project structure in Figure 2-4.

**Figure 2-4. Project structure for demo-counter project**

In this demo, we create a header file called mycounter.h. This C header provides one function, next_counter(), that generates an incremented number based on the input. The next_counter() function will be implemented into the mycounter.c file. In the previous section, we declared this library.

Next, we access mycounter.h in our main program. It’s implemented in the demo-counter.c file. We use an event timer library from Contiki-NG, etimer. We generate an incremented number every three seconds.
The following is the complete code for demo-counter.c:

```c
#include "contiki.h"
#include "sys/etimer.h"
#include "mycounter.h"

#include <stdio.h> /* For printf() */

PROCESS(counter_process, "Counter process");
AUTOSTART_PROCESSES(&counter_process);

static struct etimer timer;
static int counter = 0;

PROCESS_THREAD(counter_process, ev, data)
{
    PROCESS_BEGIN();

    printf("Demo Counter\n");
    while(1) {
        etimer_set(&timer, 3 * CLOCK_SECOND);
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));

        counter = next_counter(counter);
        printf("Counter: %d\n", counter);
    }

    PROCESS_END();
}

We also create a Makefile to configure our compiling. You can write these scripts for the Makefile. Change the CONTIKI value with your Contiki-NG root folder:

CONTIKI_PROJECT = demo-counter
all: $(CONTIKI_PROJECT)
```

58
PROJECT_SOURCEFILES += mycounter.c

CONTIKI = /home/user/Documents/book/contiki
include $(CONTIKI)/Makefile.include

Save the program.

For testing, we run this program with a native target. You can open Terminal and navigate to the project folder. Then, you can compile and run the program. Type these commands:

$ make TARGET=native
$ ./demo-counter.native

If it succeeds, you will see an incremented number in Terminal. You can see a sample of the program output in Figure 2-5.

Figure 2-5. Program output for demo-counter
How does it work?

This program is simple. First, we declare our Contiki-NG process, `counter_process`. We also declare our event timer and number variable:

```c
PROCESS(counter_process, "Counter process");
AUTOSTART_PROCESSES(&counter_process);

static struct etimer timer;
static int counter = 0;

PROCESS_THREAD(counter_process, ev, data)
{

}
```

Inside the `counter_process` function, we perform looping with the event timer. After raising an event, we call the `next_counter()` function to get an incremented number. Then, we print this value in Terminal:

```c
PROCESS_THREAD(counter_process, ev, data)
{
    PROCESS_BEGIN();
    printf("Demo Counter\n");
    while(1) {
        etimer_set(&timer, 3 * CLOCK_SECOND);
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));
        counter = next_counter(counter);
        printf("Counter: %d\n", counter);
    }
    PROCESS_END();
}
```
Contiki-NG Coding Conventions

Imagine you’re developing a program with a team of more than three developers. Each developer has a programming style and method. When the codes are merged, it may raise problems since the codes are not consistent in writing style. This problem could be solved if the code was written in the same style.

Contiki coding conventions are rules for a writing style. They consist of guidelines for how to write Contiki-NG programs. You can read the full coding conventions for Contiki at https://github.com/contiki-os/contiki/blob/master/doc/code-style.C.

Demo: Build Contiki-NG Application

In this section, we are going to develop a Contiki-NG application called virtual-sensor. We will use the general library for the sensor, which will be used for temperature and humidity readings. We will create a virtual sensor in a header file that exposes two functions, read_temperature() and read_humidity(). You can see this in Figure 2-6.

![Figure 2-6. A simple library for virtual sensor](image)
Now, we can start to develop. Create a project folder called demo-sensor. Then, create files as follows:

- **mysensor.h** is a header file that declares `read_temperature()` and `read_humidity()` functions.
- **mysensor.c** implements `read_temperature()` and `read_humidity()` functions.
- **demo-sensor.c** is the main program that uses `mysensor` library.
- **Makefile** is used for compiler parameters.

This project structure is shown in Figure 2-7.

---

**Figure 2-7. Project structure for demo-sensor**

Now, we implement all files, starting with `mysensor.h`. It's a header file that exposes two functions:

```c
// mysensor.h
#ifndef MYSENSOR_H
#define MYSENSOR_H

struct Sensor {
    char name[15];
    float value;
};
#endif
```

62
struct Sensor read_temperature();
struct Sensor read_humidity();
#endif

These sensor functions will be used as a general sensor that is implemented for temperature and humidity in the mysensor.c file. The following is the complete code for mysensor.c:

// mysensor.c
#include "mysensor.h"
#include <string.h>
#include <stdlib.h>

float random_value(float min, float max)
{
    float scale = rand() / (float) RAND_MAX;
    return min + scale * (max - min);
}

struct Sensor read_temperature()
{
    struct Sensor temp;
    strncpy(temp.name, "Temperature", 15);
    temp.value = random_value(0, 35);
    return temp;
}

struct Sensor read_humidity()
{
    struct Sensor humidity;
    strncpy(humidity.name, "Humidity", 15);
    humidity.value = random_value(40, 80);
    return humidity;
}
We generate a random number for our functions. Moreover, we use
the Sensor library in our main program, demo-sensor.c. We use an event
timer for the reading period. The following is the complete code for
demo-sensor.c:

#include "contiki.h"
#include "sys/etimer.h"
#include "mysensor.h"
#include <stdio.h>

PROCESS(sensor_process, "Sensor process");
AUTOSTART_PROCESSES(&sensor_process);
static struct etimer timer;

PROCESS_THREAD(sensor_process, ev, data)
{
    PROCESS_BEGIN();
    printf("Demo Virtual Sensor\n");
    while(1) {
        etimer_set(&timer, 3 * CLOCK_SECOND);
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));

        struct Sensor temp = read_temperature();
        printf("%s: %.2f\n", temp.name, temp.value);
        struct Sensor hum = read_humidity();
        printf("%s: %.2f\n", hum.name, hum.value);
        printf("------------\n");
    }
    PROCESS_END();
}

CHAPTER 2  BASIC CONTIKI-NG PROGRAMMING

64
Last, we create a Makefile for compiler parameters. We set PROJECT_SOURCEFILES with mysensor.c. Change the CONTIKI value to your Contiki root folder. The following is the content of the Makefile:

```
CONTIKI_PROJECT = demo-sensor
all: $(CONTIKI_PROJECT)
PROJECT_SOURCEFILES += mysensor.c

CONTIKI = /home/user/Documents/book/contiki
include $(CONTIKI)/Makefile.include
```

Now, we can compile and upload the program into the WSN mote. We can compile and run it for targeting native. Open Terminal and navigate to the project folder. Then, type these commands:

```
$ make TARGET=native
$ ./demo-sensor.native
```

This program will display sensor values after the event timer is raised. You can see my program output in Figure 2-8.
Summary

We have explored the Contiki-NG basic programming language. This is a core language based on the C programming language that will help you to develop a Contiki-NG application. We built a Contiki-NG application that utilized a header library and event timer with Contiki-NG program features.

In the next chapter, we will learn how to work with concurrency in Contiki-NG programs in order to deal with multiple jobs.
CHAPTER 3

Concurrency

Multi-tasking can increase the scalability of your Contiki-NG applications. This chapter will explore how to work with concurrency in Contiki-NG applications.

The following is a list of topics that will be covered in this chapter:

- Introduction to concurrency
- Concurrency approach in Contiki-NG
- Threading
- Task scheduling

Introduction to Concurrency

Concurrency is the ability to perform more than one task at the same time. Let’s say you have a WSN mote with multiple sensor devices. You want to sense through all the sensors at the same time.

We illustrate a concurrency in Figure 3-1. A process can handle multiple tasks with different problem models. We can achieve concurrency by applying a queue that is either FIFO (First In First Out) or LIFO (Last In First Out). Another solution is to apply asynchronous code in our program.

In this chapter, we will explore how to implement concurrency in Contiki-NG applications. Some samples are provided to show how to do it.
Concurrency Approach in Contiki-NG

Contiki-NG provides concurrency features. We can use several approaches to develop application-based concurrency. There are four methods with which we can implement concurrency in Contiki-NG applications, as follows:

- Processes
- Timers
- Threading
- Task scheduling

We will discuss and implement these methods in the next section.

Figure 3-1. A process performs some tasks
Introducing Contiki-NG Processes

Contiki-NG applications can run on a process that executes in either cooperative or preemptive mode. Cooperative mode is regular execution in the microcontroller. A process with preemptive mode runs with interruptions resulting from I/O or timers.

In general, we can create a process in Contiki-NG by calling \texttt{PROCESS()}. We can define a process as

\begin{verbatim}
PROCESS(name, process_name);
\end{verbatim}

where

- \texttt{name} is a variable of the process, and
- \texttt{process\_name} is a process name that is represented as a string.

Then, we implement this process using \texttt{PROCESS\_THREAD}. We can declare this as follows:

\begin{verbatim}
PROCESS\_THREAD(name, ev, data)
{
  PROCESS\_BEGIN();
  // do something
  PROCESS\_END();
}
\end{verbatim}

The following is a list of macros APIs for processes:

- \texttt{PROCESS\_BEGIN()}: Declares the beginning of a process protothread
- \texttt{PROCESS\_END()}: Declares the end of a process protothread
- \texttt{PROCESS\_EXIT()}: Exits the process
• PROCESS_WAIT_EVENT(): Waits for any event
• PROCESS_WAIT_EVENT_UNTIL(): Waits for an event, but with conditions
• PROCESS_YIELD(): Waits for any event; equivalent to PROCESS_WAIT_EVENT()
• PROCESS_WAIT_UNTIL(): Waits for a given condition; may not yield the microcontroller
• PROCESS_PAUSE(): Temporarily yields the microcontroller

For demo purposes, we will create a simple Contiki-NG app. Create a folder called demo-process. Then, create two files, demo-process.c and Makefile. The first step is to write a program for demo-process.c. We define three processes as follows:

```c
#include "contiki.h"
#include <stdio.h>

PROCESS(myprocess1, "process 1");
PROCESS(myprocess2, "process 2");
PROCESS(myprocess3, "process 3");
AUTOSTART_PROCESSES(&myprocess1, &myprocess2, &myprocess3);
```

You can see that we created three processes. We pass these process variables to AUTOSTART_PROCESSES(). We implement the following code for our processes:

```c
PROCESS_THREAD(myprocess1, ev, data)
{
    PROCESS_BEGIN();

    printf("This message from process 1\n");
```
Save this program. We continue to Makefile. We declare our program and the path of the Contiki-NG root directory via CONTIKI:

CONTIKI_PROJECT = demo-process
all: $(CONTIKI_PROJECT)

CONTIKI = /home/user/Documents/book/contiki
include $(CONTIKI)/Makefile.include

Change the Contiki-NG path directory in CONTIKI.

To compile and run the program, you can open Terminal and navigate to a program folder. You can type the following commands. For instance, we run on local native OS:

$ make TARGET=native
$ ./demo-process.native
If this succeeds, you will see messages from each process in Terminal. See Figure 3-2.

![Figure 3-2. Running demo-process app](image)

### Working with Timers

We can perform some activities by utilizing timer objects. Contiki-NG has one clock and several timer modules, such as `timer`, `stimer`, `ctimer`, `etimer`, and `rtimer`. All these libraries can be found in the `<contiki>/os/sys` folder.

We will explore these libraries in the next section.

### Clock Library

The Clock library can be used for doing general activities with time. It is declared in `clock.h` from the `<contiki>/os/sys` folder. You can see the content of the `clock.h` file here:
clock_time_t clock_time(); // Get the system time.
unsigned long clock_seconds(); // Get the system time in seconds.
void clock_delay(unsigned int delay); // Delay the CPU.
void clock_wait(int delay); // Delay the CPU for a number of clock ticks.
void clock_init(void); // Initialize the clock module.
CLOCK_SECOND; // The number of ticks per second.

To use this library, call these functions directly from the program. For instance, you can access a clock time by calling the clock_time() function:

```c
    clock_time_t t = clock_time();
    printf("Timer start: %lu \n", t);
```

**Timer Library**

The Timer library provides functions for setting, resetting, and restarting timers, and for checking if a timer has expired. This library is found in the timer.h file and defines several functions as follows:

```c
void timer_set(struct timer *t, clock_time_t interval); // Start the timer.
void timer_reset(struct timer *t); // Restart the timer from the previous expiration time.
void timer_restart(struct timer *t); // Restart the timer from current time.
int timer_expired(struct timer *t); // Check if the timer has expired.
clock_time_t timer_remaining(struct timer *t); // Get the time until the timer expires.
```

CHAPTER 3  CONCURRENCY
An application must manually check if its timers have expired. To use the Timer library, we call `timer_set()`. Then, we can verify an expired timer by calling the `timer_expired()` function:

```c
timer_set(&timer_timer, 3 * CLOCK_SECOND);
if(timer_expired(&timer_timer)) {
    t = clock_time();
    printf("timer expired: %lu \n", t);
}
```

**Stimer Library**

The Stimer library is similar to the timer library, but uses time values in seconds. The following is a list of Stimer functions defined in the `stimer.h` file.

```c
void stimer_set(struct stimer *t, unsigned long interval); // Start the timer.
void stimer_reset(struct stimer *t); // Restart the stimer from the previous expiration time.
void stimer_restart(struct stimer *t); // Restart the stimer from current time.
int stimer_expired(struct stimer *t); // Check if the stimer has expired.
unsigned long stimer_remaining(struct stimer *t); // Get the time until the timer expires.
```

We can use the Stimer library with the same approach as the timer library. We set a time by calling `stimer_set()`. Then, we check for an expired timer using the `stimer_expired()` function:
stimer_set(&stimer_timer, 3);
if(stimer_expired(&stimer_timer)){
    t = clock_time();
    printf("stimer expired: %lu \n", t);
}

Etimer Library

The Etimer library is an event timer library that generates an event. We can verify this event using PROCESS_WAIT_EVENT_UNTIL(). You can see event timer declarations in the etimer.h file:

void etimer_set(struct etimer *t, clock_time_t interval); // Start the timer.
void etimer_reset(struct etimer *t); // Restart the timer from the previous expiration time.
void etimer_restart(struct etimer *t); // Restart the timer from current time.
void etimer_stop(struct etimer *t); // Stop the timer.
int etimer_expired(struct etimer *t); // Check if the timer has expired.
int etimer_pending(); // Check if there are any non-expired event timers.
clock_time_t etimer_next_expiration_time(); // Get the next event timer expiration time.
void etimer_request_poll(); // Inform the etimer library that the system clock has changed.
For demo purposes, we can call etimer_set() to set our event timer. Then, we wait for the expired event using PROCESS_WAIT_EVENT_UNTIL():

```c
etimer_set(&etimer_timer, 3 * CLOCK_SECOND);
while(1){
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&etimer_timer));
    etimer_reset(&etimer_timer);
}
```

CTimer Library

The CTimer library provides a function callback that will be called when timer expiration occurs. CTimer functions are defined in the ctimer.h file:

```c
void ctimer_set(struct ctimer *c, clock_time_t t, void(*f)(void *), void *ptr); // Start the timer.
void ctimer_reset(struct ctimer *t); // Restart the timer from the previous expiration time.
void ctimer_restart(struct ctimer *t); // Restart the timer from current time.
void ctimer_stop(struct ctimer *t); // Stop the timer.
int ctimer_expired(struct ctimer *t); // Check if the timer has expired.
```

For this demo, we can define a function, perform_ctime_function(). This function is passed to ctimer_set() when we set the ctimer library:

```c
void perform_ctime_callback()
{
    printf("Process demo_timer3: ctimer callback called\n");
}
```
Rtimer Library

The Rtimer library provides scheduling and execution for real-time tasks. We can define a specific execution time when we set `rtimer` using the `rtimer_set()` function. You can see the `rtimer` function in the `rtimer.h` file:

```c
RTIMER_CLOCK_LT(a, b); // This should give TRUE if 'a' is less than 'b', otherwise false.
RTIMER_ARCH_SECOND; // The number of ticks per second.
void rtimer_arch_init(void); // Initialize the rtimer architecture code.
rtimer_clock_t rtimer_arch_now(); // Get the current time.
int rtimer_arch_schedule(rtimer_clock_t wakeup_time); // Schedule a call to rtimer_run_next().
```

For implementation, we declare a function that is passed in the `rtimer_set()` function. We also set the execution time:

```c
static rtimer_clock_t timeout_rtimer = RTIMER_SECOND / 2;
void
perform_rtime_callback()
{
    printf("Process demo_timer3: rtimer callback called\n");
}

rtimer_set(&rtimer_timer, RTIMER_NOW() + timeout_rtimer, 0, perform_rtime_callback, NULL);
```
Now, we will try to use Contiki-NG to run a demo about timers. We create a folder, called demo-timer. We create two files, demo-timer.c and Makefile. We put all code from all the timer libraries in the demo-timer.c file.

The following is the complete code for the demo-timer.c file:

```c
#include "contiki.h"
#include "sys/clock.h"
#include "sys/timer.h"
#include "sys/stimer.h"
#include "sys/etimer.h"
#include "sys/ctimer.h"
#include "sys/rtimer.h"
#include <stdio.h>

static int counter;
static struct timer timer_timer;
static struct stimer stimer_timer;
static struct etimer etimer_timer;
static struct ctimer ctimer_timer;
static struct rtimer rtimer_timer;

PROCESS(demo_timer1, "demo timer stimer");
PROCESS(demo_timer2, "demo etimer");
PROCESS(demo_timer3, "demo ctime");
PROCESS(demo_timer4, "demo rtime");
AUTOSTART_PROCESSES(&demo_timer1, &demo_timer2,
                     &demo_timer3, &demo_timer4);

static rtimer_clock_t timeout_rtimer = RTIMER_SECOND / 2;

void
perform_ctime_callback()
```

Chapter 3  ConCurrenCy
{  
printf("Process demo_timer3: ctimer callback called\n");

if(counter>=2){
    printf("ctimer is stopped\n");
    ctimer_stop(&ctimer_timer);
}
else
    ctimer_reset(&ctimer_timer);
}

void
perform_rtime_callback()
{
    printf("Process demo_timer3: rtimer callback called\n");

    if(counter<2){
        rtimer_set(&rtimer_timer, RTIMER_NOW() + timeout_rtimer, 0,
                   perform_rtime_callback, NULL);
    }
    else
        printf("rtimer is stopped\n");
}

PROCESS_THREAD(demo_timer1, ev, data)
{
    PROCESS_BEGIN();
    counter = 0;
    clock_time_t t = clock_time();
    printf("Timer start: %lu \n", t);

    timer_set(&timer_timer, 3 * CLOCK_SECOND);
    stimer_set(&stimer_timer, 3);
while(1){
    if(timer_expired(&timer_timer)){
        t = clock_time();
        printf("timer expired: %lu \n", t);
        timer_reset(&timer_timer);
        counter++;
    }
    if(stimer_expired(&stimer_timer)) {
        t = clock_time();
        printf("stimer expired: %lu \n", t);
        stimer_reset(&stimer_timer);
        counter++;
    }
    if(counter>=2)
        break;
}
printf("demo_timer1 process end\n");
PROCESS_END();

PROCESS_THREAD(demo_timer2, ev, data)
{
    PROCESS_BEGIN();

    printf("demo etimer\n");
    clock_time_t t = clock_time();
    printf("etimer start: %lu \n", t);

    printf("set etimer 3 clock_second\n");
etimer_set(&etimer_timer, 3 * CLOCK_SECOND);

    while(1){
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&etimer_timer));
    }
}
etimer_reset(&etimer_timer);
if(counter>=2)
    break;
}
printf("demo_timer2 process end\n");
PROCESS_END();
}

PROCESS_THREAD(demo_timer3, ev, data)
{
    PROCESS_BEGIN();
    printf("demo ctime\n");
    while(1) {
        ctimer_set(&ctimer_timer, CLOCK_SECOND, perform_ctime_callback, NULL);
        PROCESS_YIELD();
    }

    PROCESS_END();
}

PROCESS_THREAD(demo_timer4, ev, data)
{
    PROCESS_BEGIN();
    printf("demo rtime\n");
    while(1) {
        rtimer_set(&rtimer_timer, RTIMER_NOW() + timeout_rtimer, 0, perform_rtime_callback, NULL);
        PROCESS_YIELD();
    }

    PROCESS_END();
}
Next, we write the Makefile file with the following code:

```makefile
CONTIKI_PROJECT = demo-timer
all: $(CONTIKI_PROJECT)

CONTIKI = /home/user/Documents/book/contiki
include $(CONTIKI)/Makefile.include
```

Change `CONTIKI` to your Contiki-NG root folder. Now, you can compile and run this program. Open Terminal and navigate to your program folder. Then, type these commands:

```
$ make TARGET=native
$ ./demo-timer.native
```

This program will run in Terminal. You can run it on your Contiki-NG hardware. You can see the program output on Terminal in Figure 3-3.

![Running demo-timer app](image)

**Figure 3-3. Running demo-timer app**
Threading

Contiki-NG OS removes multithreading from the source code. You can read about it at https://github.com/contiki-ng/contiki-ng/pull/172. This probably is also supported in the next version. If you want to work with multithreading, you can use regular Contiki. Contiki provides threading so that we can perform many tasks through \texttt{mt\_thread} in the \texttt{mt.h} header file. This file is found in the Contiki source code in the <contiki\_root>/core/sys/ folder. You can see several functions in the \texttt{mt.h} file as follows:

\begin{verbatim}
void \texttt{mt\_init}(void) : Initializes the library.
void \texttt{mt\_remove}(void) : Uninstalls the library.
void \texttt{mt\_start}(\texttt{struct \_mt\_thread } *thread, \texttt{void \(* \_function\)(void *)},
void *\_data) : Starts a thread.
void \texttt{mt\_exit}(void) : Exits a thread.
void \texttt{mt\_exec}(\texttt{struct \_mt\_thread } *thread) : Execute as thread.
void \texttt{mt\_yield}(void) : Release control voluntarily.
void \texttt{mt\_stop}(\texttt{struct \_mt\_thread } *thread) : Stops a thread.
\end{verbatim}

In this section, we will develop multithreading using Contiki. Please skip this section if you still work with Contiki-NG.

You can see threading states for a Contiki app in Figure 3-4.
Each thread runs on a process. Each thread can communicate by either sharing resources or exchanging data. Figure 3-5 shows three threads running on a process.

We can create a thread within a Contiki process with the following method:

```c
static struct mt_thread thread1;
int n1=10;
mt_init();
mt_start(&thread1, thread_compute, &n1);
```

Our thread is passed a function, called `thread_compute`. We also give a parameter for our function. The `thread_compute()` function can be declared as follows:

```c
void
thread_compute(void *data)
{
    printf("calling thread_compute()\n");
}
```

Figure 3-4. Threading states for a Contiki app
// do something
mt_yield();
mt_exit();
}

You can see that our function is executed and then that thread is closed using \texttt{mt\_yield()} and \texttt{mt\_exit()}.

\textbf{Figure 3-5.} \textit{Threads in Contiki process}

For demo purposes, we create two threads. Each thread will execute a loop for a certain time based on the input data. You can start by creating a folder, called demo-thread. Then, create two files, demo-thread.c and Makefile.

First, we write code in the demo-thread.c file. We define two threads. We also create a function, called thread_compute(), that will be passed on our threads.

You can write the following complete code for the demo-thread.c file:

```c
#include "contiki.h"
#include "sys/mt.h"
#include <stdio.h>

PROCESS(mythread, "demo thread");
AUTOSTART_PROCESSES(&mythread);
```
static int count;

void
thread_compute(void *data)
{
    printf("calling thread_compute()\n");

    int num = *((int*)data);
    printf("num: %d\n", num);
    int i;

    int val = 0;
    for(i=0;i<num;i++){
        val+=2;
    }
    printf("num: %d, total: %d\n", num, val);
    count++;
    mt_yield();
    mt_exit();
}

PROCESS_THREAD(mythread, ev, data)
{
    PROCESS_BEGIN();
    printf("demo threading\n");

    static struct mt_thread thread1;
    static struct mt_thread thread2;

    int n1=10, n2=20;

    mt_init();
    mt_start(&thread1, thread_compute, &n1);
    mt_start(&thread2, thread_compute, &n2);
mt_exec(&thread1);
mt_exec(&thread2);
while(1) {
    if(count>1) {
        mt_stop(&thread1);
        mt_stop(&thread2);
        break;
    }
}

printf("thread exit\n");
mt_remove();
PROCESS_END();
}

Now, we can create the Makefile. We declare the project name and Contiki path, which is defined ion CONTIKI:

CONTIKI_PROJECT = demo-thread
all: $(CONTIKI_PROJECT)

CONTIKI_WITH_RIME = 1
CONTIKI = /home/user/Documents/book/contiki
include $(CONTIKI)/Makefile.include

You can save all the files. You can compile and run the program with these commands in Terminal:

$ make TARGET=native
$ ./demo-thread.native
A sample of the program output can be seen in Figure 3-6.

**Figure 3-6. Program output for demo-thread**

## Task Scheduling

Task scheduling means that all tasks are scheduled to be executed. In a Contiki-NG application, task scheduling collects all tasks in process, timers, and threading.

In this section, we will try to implement task scheduling by utilizing the Etimer library. Our task list will be stored on the Contiki-NG list that you can find in the `<contiki>/os/lib/list.h` file. Figure 3-7 shows our demo to implement task scheduling.
Now, create a folder, called `demo-scheduling`. We also create a `demo-scheduling.c` file for our main program. The following is a complete program for the `demo-scheduling.c` file:

```c
#include "contiki.h"
#include "sys/clock.h"
#include "sys/rtimer.h"
#include "lib/list.h"
#include <stdio.h>

PROCESS(demo_taskscheduling, "demo task scheduling");
AUTOSTART_PROCESSES(&demo_taskscheduling);

static struct rtimer rtimer_timer;

struct simple_task {
    struct simple_task *next;
    int value;
};

static struct simple_task *s;
static int number_task = 10;
static int is_completed = -1;
```

**Figure 3-7. Executing tasks**

Now, create a folder, called demo-scheduling. We also create a demo-scheduling.c file for our main program. The following is a complete program for the demo-scheduling.c file:

```c
#include "contiki.h"
#include "sys/clock.h"
#include "sys/rtimer.h"
#include "lib/list.h"
#include <stdio.h>

PROCESS(demo_taskscheduling, "demo task scheduling");
AUTOSTART_PROCESSES(&demo_taskscheduling);

static struct rtimer rtimer_timer;

struct simple_task {
    struct simple_task *next;
    int value;
};

static struct simple_task *s;
static int number_task = 10;
static int is_completed = -1;
```
PROCESS_THREAD(demo_taskscheduling, ev, data)
{
    PROCESS_BEGIN();

    printf("demo task schedule\n");
    list_init(task_list);
    int i;

    printf("prepare task data\n");
    struct simple_task task[10];
    for(i=0;i<number_task;i++){
        task[i].value = i + 1;
        list_add(task_list, &task[i]);
    }

    void
    perform_rtime_callback(struct rtimer *t, void *ptr)
    {
        printf("task callback called\n");
        int num = *((int*)ptr);
        printf("perform task=%d\n", num);

        s = list_item_next(s);
        if(s!=NULL)
        {
            rtimer_set(&rtimer_timer, RTIMER_NOW() + (RTIMER_SECOND/2), 1,
                        perform_rtime_callback,&s->value);
        }else{
            printf("all tasks completed\n");
            is_completed = 1;
        }
    }
}

Chapter 3  CONCURRENCy

LIST(task_list);

void
perform_rtime_callback(struct rtimer *t, void *ptr)
{
    printf("task callback called\n");
    int num = *((int*)ptr);
    printf("perform task=%d\n", num);

    s = list_item_next(s);
    if(s!=NULL)
    {
        rtimer_set(&rtimer_timer, RTIMER_NOW() + (RTIMER_SECOND/2), 1,
                   perform_rtime_callback,&s->value);
    }else{
        printf("all tasks completed\n");
        is_completed = 1;
    }
}
s = list_head(task_list);
rtimer_set(&rtimer_timer, RTIMER_NOW() + (RTIMER_SECOND/2), 1,
   perform_rtime_callback,&s->value);

while(1) {
   if(is_completed>0)
      break;
}
printf("program exit\n");
PROCESS_END();
}

To compile and run the program, you should create a Makefile file with the compiler configuration. We can create these scripts:

CONTIKI_PROJECT = demo-scheduling
all: $(CONTIKI_PROJECT)

CONTIKI = /home/user/Documents/book/contiki
include $(CONTIKI)/Makefile.include

Change CONTIKI to your Contiki-NG root path.

Now, you can compile and run this program. For instance, we compile and run it for a local native app. Open a Terminal and type these commands:

$ make TARGET=native
$ ./demo-scheduling.native

If it succeeds, all tasks will be executed. You can see the program output in Figure 3-8.

How does this work?
This program starts by declaring a task list in the struct model:

```c
struct simple_task {
    struct simple_task *next;
    int value;
};

static struct simple_task *s;
static int number_task = 10;
static int is_completed = -1;
LIST(task_list);
```

**Figure 3-8. Program output for demo-scheduling**

This program starts by declaring a task list in the struct model:
Then, it generates the task list in the main process:

```c
list_init(task_list);
int i;
printf("prepare task data\n");
struct simple_task task[10];
for(i=0;i<number_task;i++){
    task[i].value = i + 1;
    list_add(task_list, &task[i]);
}
```

Next, it runs a task by picking it up from the task list:

```c
s = list_head(task_list);
rtimer_set(&rtimer_timer, RTIMER_NOW() + (RTIMER_SECOND/2), 1,
    perform_rtime_callback,&s->value);
```

Each task will be executed in the `perform_rtime_callback()` function. After completion, it gets a task again and then executes it:

```c
void
perform_rtime_callback(struct rtimer *t, void *ptr)
{
    printf("task callback called\n");
    int num = *((int*)ptr);
    printf("perform task=%d\n", num);
    s = list_item_next(s);
    if(s!=NULL){
        rtimer_set(&rtimer_timer, RTIMER_NOW() + (RTIMER_SECOND/2), 1,
            perform_rtime_callback,&s->value);
    }else{
        printf("all tasks completed\n");
        is_completed = 1;
    }
}
```
CHAPTER 3  CONCURRENCY

To exit from our program, we can check our program state:

```c
while(1) {
    if(is_completed>0)
        break;
}
```

Summary

We have learned how to implement concurrency in Contiki-NG OS. Performing several tasks in the Contiki-NG application has been reviewed. We explored process, timers, threading, and task scheduling.

In the next chapter, we will learn how to communicate between a Contiki-NG application and the computer.
CHAPTER 4

Contiki-NG and Computer Communication

Dada communication capabilities available on WSN motes enable the exchange of data among WSN motes. In this chapter, we will learn how Contiki-NG aids communication among WSN motes and from a WSN mote to a computer. We will also explore how to build middleware that enables WSN motes to communicate with other systems.

The following is a list of topics we will cover in this chapter:

- Communication models for Contiki-NG
- Serial communication
- Building communication among Contiki-NG motes
- Building communication between computer and Contiki-NG motes
- Developing middleware
Communication Models for Contiki-NG

How you want to communicate between WSN motes and computers will determine what kind of communication model you use. As you know, each WSN mote usually has network capability so it can exchange data with WSN motes and computers.

A communication model for Contiki-NG is depicted in Figure 4-1. Some WSN motes have the capability to connect to servers directly, but other motes may not, so those motes use a hub/gateway/middleware to communicate with servers.

**Figure 4-1. Communication models in Contiki-NG**

In this chapter, we will learn how WSN motes communicate with others. We need at least two motes to implement our demo.
Serial Communication

*Serial communication* can be defined as a process used to send and receive a bit at a time sequentially. Sometimes serial communication is called UART (Universal Asynchronous Receiver/Transmitter). Technically, we already used serial communication when we uploaded Contiki-NG programs to WSN motes in previous chapters. We also communicate with WSN motes via the `printf()` function. Then, we listen to the messages from `printf()` using the command `make login`.

A serial communication protocol is represented in Figure 4-2. In hardware implementation, a serial communication needs at least three pinouts. These are Rx (Receiver), Tx (Transmitter), and GND (Ground) pins.

---

**Figure 4-2.** Serial communication

**Communication Between Contiki Mote and Computer**

In this section, we will build Contiki and Contiki-NG programs to communicate between a Contiki mote and a computer. This is useful because we can control our Contiki motes from the computer. Running applications on a computer has benefits, such as interacting with the database server and communicating with the cloud server.
There are a lot of methods used to communicate between a computer and a Contiki mote. We will focus on two methods: serial communication and shell. We will explore these methods in the next section.

Let's build!

## Access Contiki Motes via Serial Communication

Since our WSN motes are attached to the computer through serial communication, we can initiate communication between the computer and a WSN mote. Some application platforms, such as C/C++, Java, Python, C#, and Node.js, provide libraries with which to implement serial communication.

In this section, we will try to access WSN motes using Python through serial communication. To implement serial communication in Python, we use the `pyserial` library. You can find this library at [https://pypi.python.org/pypi/pyserial](https://pypi.python.org/pypi/pyserial). This library can be installed using pip.

You can verify if your computer has installed pip or not. You can type this command in Terminal:

```
$ which pip
```

If you don’t get a response, it means your computer is missing the pip program. You can install pip using this command in Terminal:

```
$ sudo apt-get install python-pip
```

Now, you can install the `pyserial` library by typing this command in Terminal:

```
$ pip install pyserial
```

You also can install the `pyserial` library using `easy_install`. You can type this command in Terminal:

```
$ easy_install -U pyserial
```
Make sure your computer is connected to the Internet.

Next, you must write a Python program to listen for incoming messages on a specific serial port. You should know which serial port is used by the Contiki mote. You need it because you will use it on your program.

To find a serial port from the Contiki mote, you can open Terminal and type this command:

```
$ ls /dev/tty*
```

You should see a list of connected serial ports available on your computer. Depending on your Contiki mote model, you can verify the port by turning the mote on/off so you should which a new attached serial port. For instance, my Contiki mote is detected as /dev/ttyUSB0. You can see it in Figure 4-3.

Now, you will develop the Python program. You will print all incoming data from the serial port to the console. Set the baudrate to 115200 for the serial-port speed. Create a file, called `contiki-viewer.py`, and write these complete scripts. Change the `PORT` value to that for your Contiki-NG mote:

```python
#!/usr/bin/python
import serial

PORT = '/dev/ttyUSB0'
ser = serial.Serial(
    port=PORT,
    baudrate=115200,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
    timeout=0)

print("connected to: " + ser.portstr)
ser.write("help\n")
while True:
```
line = ser.readline()
if line:
    print(line),
ser.close()

Save this program.

Figure 4-3. Getting a list of serial ports in your computer

To simulate this demo, you should upload the Contiki-NG program to a Contiki-NG mote. You can use the same program from Chapter 1, Hello World. You should upload that program into the Contiki mote.

After Contiki-NG is deployed with a Contiki-NG program, you can run the Python program, contiki-viewer.py, by typing this command:

$ python contiki-viewer.py
If you have error issues related to security access, you probably should run it with administrator privileges. Type this command:

$ sudo python contiki-viewer.py

After it has executed, you will see that this program is waiting for incoming messages from the serial port, as shown in Figure 4-4.

Figure 4-4. Executing Python program for listening to the serial port

If you do not see anything, you should reset your Contiki mote. Now the message from the Hello World program can be seen on the console. You can see my program depicted in Figure 4-5.
I have also tested with Contiki-NG on a TelosB mote. It works. You can see that program output in Figure 4-6.

Figure 4-5. Reading data from serial port of Contiki using Python
How does this work?

This program starts by initializing the serial library and activating the serial port of the Contiki-NG mote:

```python
import serial

PORT = '/dev/ttyUSB0'
ser = serial.Serial(
    port=PORT,
    baudrate=115200,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
    timeout=0)
```

**Figure 4-6.** Reading data from serial port of Contiki-NG using Python
After the serial port is activated, we listen for incoming message from that serial port. We call `readline()` to read data from the serial port. Once we receive data, we print the data to the console:

```python
while True:
    line = ser.readline()
    if line:
        print(line),

ser.close()
```

Last, we close our serial port by calling the `close()` method.

**Contiki Shell**

Contiki OS provides a shell API that we can utilize to communicate with internal Contiki-NG motes. We find the shell API in the Contiki source code in `<Contiki-root>/apps/shell`. You should see several API objects, shown in Figure 4-7.

Contiki shell is very useful. Let’s say you build and deploy a Contiki application onto a Contiki mote. Then, you want to analyze what is happening inside the Contiki mote. To do this, you can build a custom Contiki shell related to your needs. You call the shell from the Contiki-NG mote and perform your analysis.

One important thing that you should know is the limitations on Contiki mote storage and resources. Building more Contiki shell APIs means using more mote resources. Make sure your Contiki shell is optimal for your Contiki mote model.

In this section, we will learn how to build a Contiki shell application and then deploy it to a Contiki mote. For this simple demo, we will use a Contiki sample from the Contiki source code. You can see it at `<Contiki-root>examples/example-shell`. This program runs for native platforms such as a computer.
Open the example-shell.c file. You should see the PROCESS_THREAD(example_shell_process, ev, data) function, as follows:

```
PROCESS_THREAD(example_shell_process, ev, data)
{
    PROCESS_BEGIN();

    serial_shell_init();

    shell_base64_init();
    shell_blink_init();
    /*shell_coffee_init();*/
    shell_download_init();
    /*shell_exec_init();*/
    shell_file_init();
    shell_httpd_init();
    shell_irc_init();
    /*shell_ping_init();*/ /* uIP ping */
    shell_power_init();
    /*shell_profile_init();*/
    shell_ps_init();
    /*shell_reboot_init();*/
    shell_rime_debug_init();
    shell_rime_netcmd_init();
    shell_rime_ping_init(); /* Rime ping */
    shell_rime_sendcmd_init();
    shell_rime_sniff_init();
    shell_rime_init();
    /*shell_rsh_init();*/
    shell_run_init();
    shell_sendtest_init();
    /*shell_sky_init();*/
    shell_tcpsend_init();
```
shell_text_init();
shell_time_init();
shell_udpsend_init();
shell_vars_init();
shell_wget_init();

PROCESS_END();

Figure 4-7. Contiki shell APIs in Contiki source code

Each shell API is defined as `shell_xxx()`. Remarked codes are not supported for native platforms such as a computer.
Now, you can compile and run this program. Open Terminal and navigate to the `<Contiki-root>/examples/example-shell` folder. Then, type these commands:

```
$ make TARGET=native
$ ./example-shell.native
```

If it succeeds, you should see the Contiki shell that is shown in Figure 4-8.

![Figure 4-8. Executing example-shell application](image)

For this demo, you call one of the Contiki shells. To get a list of Contiki shells, you can call the `help` shell command:

```
Contiki > help
Contiki > echo hello world!
```
You should get a list of supported shells in the console. Then, execute `echo` shell command. A sample of the program's output can be seen in Figure 4-9.

![Fig 4.9](image)

**Figure 4-9. Running shell sample**

You have run a Contiki shell on the computer as a native application. Now, you will build a Contiki shell for a Contiki mote. For demo purposes, I will use TelosB hardware that’s type is Sky.

You will use a sample program from the Contiki source code. You can find `shell-sky` at `<Contiki-root>examples/shell-sky` folder.
If you open the sky-shell.c file, you should see the `PROCESS_THREAD(sky_shell_process, ev, data)` code as follows:

```c
PROCESS_THREAD(sky_shell_process, ev, data)
{
    PROCESS_BEGIN();

#if WITH_PERIODIC_DEBUG
    ctimer_set(&debug_timer, 20 * CLOCK_SECOND, periodic_debug, NULL);
#endif /* WITH_PERIODIC_DEBUG */

    serial_shell_init();
    shell_blink_init();
    /* shell_file_init();
        shell_coffee_init();*/
    /* shell_download_init();*/
    shell_rime_sendcmd_init();
    /* shell_ps_init();*/
    shell_reboot_init();
    shell_rime_init();
    shell_rime_netcmd_init();
    /* shell_rime_ping_init();
    shell_rime_debug_init();
    shell_rime_debug_runicast_init();*/
    /* shell_rime_sniff_init();*/
    shell_sky_init();
    shell_power_init();
    shell_powertrace_init();
    /* shell_base64_init();*/
    shell_text_init();
    shell_time_init();
    /* shell_sendtest_init();*/
    shell_collect_view_init();
```
You only use some essential shells due to the Contiki mote’s resource limitation. Other shell APIs are remarked.

Now, compile and upload this program into your Contiki mote. For instance, I use the Contiki-NG mote–based Sky. Open Terminal and navigate to the `<Contiki-root>examples/shell-sky` folder. You can type these commands to compile and upload the program to the Contiki-NG mote with `sky` as target:

```
$ make TARGET=sky
$ make TARGET=sky savetarget
$ make shell-sky.upload
```

After it has completed, you can monitor your Contiki mote. You can type this command:

```
$ make login
```

Then, reset your Contiki mote in order to use the Contiki shell. If it succeeds, you should see the Contiki shell terminal as follows:

```
$ ./example-shell.native
Contiki-3.x-3330-g719f712 started
Rime started with address 2.1
MAC nullmac RDC nullrdc NETWORK Rime
2.1: Contiki>
```
You can see a sample of the Contiki shell in Figure 4-10.

Figure 4-10. Running Contiki shell on Contiki-NG mote

From the Contiki shell, you can test it using the commands `help` and `echo`. You can type these commands:

Contiki> help
Contiki> echo hello world!

A sample of the program output from the Contiki shell can be seen in Figure 4-11.

If you want to exit from the Contiki shell, you can type `exit`. If you get problems quitting from the Contiki shell, you can press CTRL+C to enforce the exit.
Contiki-NG Shell

Contiki-NG has modified Contiki shell. Contiki-NG shell is defined into a module. You should enable a shell module if you want to use it. The shell structure can be found in the `<Contiki-NG-root>/os/services/shell` folder. You can see it in Figure 4-12.
You can enable the shell module in the Makefile file from your projects. Put this script in to enable this module:

```
MODULES += os/services/shell
```

Then, you can compile and upload your project program into your Contiki mote. Since Contiki-NG needs more space on the Contiki mote, your mote probably does not fit the program, so you will get an error while compiling and flashing the program.

After you succeed in uploading program, you can test it by connecting to the Contiki mote via a serial tool. You should get “>” on the serial Terminal. Not all Contiki motes can run the Contiki-NG shell due to ROM space size. My TelosB board cannot work with this shell. So, I tested it with the TI CC2650 LaunchPad board. It works. I then run this command to access the Contiki-NG serial:

```
$ make login PORT=/dev/ttyACM0 BOARD=launchpad/cc2650 TARGET=srf06-cc26xx
```
If you do not see “>,” try to reset the board. Then, press Enter. The program output can be seen in Figure 4-13.

Now, you can test Contiki-NG with several commands. Try to execute these shell commands:

> help
> ip-addr

Figure 4-13. Running Contiki-NG shell on TI CC2650 LaunchPad board

These commands will list all registered Contiki-NG shells and display current IP address. You can see my program output in Figure 4-14.
Customizing Contiki Shell

In some cases you may need to customize the Contiki shell to fit your problems. The thing that you should be aware of is your space and resource usage while implementing a Contiki shell.

In this section, we will learn how to customize a Contiki shell on both Contiki and Contiki-NG. In Contiki-NG, a Contiki shell is called an NG shell. Each customizing shell topic will be explored in the next sections.

Custom Contiki Shell

In the previous section, we learned how to access Contiki and Contiki shells. Now, we will build our own Contiki shell API. For this simple demo, we will develop a Contiki shell with an addition math operation. This API will receive two number parameters. These number parameters will be added and then sent back as the Contiki shell output.
To build our own Contiki shell, we can add our shell objects to the `<Contiki-root>/apps/shell` folder. For our scenario, we add two files, `shell-math.c` and `shell-math.h`. You can see them in Figure 4-15.

There are two steps to building a custom Contiki shell. First, we create an object file (*.c) in which to declare all Contiki shell implementations. We declare the Contiki shell API using `SHELL_COMMAND()`. Then, we implement the shell API on a process by declaring it in `PROCESS()`.

Last, we register and initialize our Contiki shell API by calling the `shell_register_command()` function. This function will be called on the `shell_math_init()` function. The Contiki program that will use this shell should call `shell_math_init()` function to access the shell API.

*Figure 4-15. Adding an additional Contiki shell API*
For implementation, we start to write the program in the shell-math.c file. We receive two number input parameters. Then, we perform the addition operation. The following is the complete program for the shell-math.c file:

```
#include "contiki.h"
#include "shell.h"
#include <ctype.h>
#include <stdio.h>
#include <string.h>

PROCESS(shell_math_process, "math");
SHELL_COMMAND(math_command,
    "math",
    "math: math number1 number2",
    &shell_math_process);

PROCESS_THREAD(shell_math_process, ev, data)
{
    char *numbers;
    int n1 = 0;
    int n2 = 0;
    char buf[32];

    PROCESS_BEGIN();

    numbers = data;
    if(numbers == NULL || strlen(numbers) == 0) {
        shell_output_str(&math_command,
            "math number1 number2: number must be given", "");
        PROCESS_EXIT();
    }
}
```
char * pch;
pch = strtok (numbers, " ");
if (pch != NULL)
{
    n1 = atoi(pch);
pch = strtok (NULL, " ");
if (pch != NULL)
{
    n2 = atoi(pch);
}else
{
    shell_output_str(&math_command,
        "math number1 number2: number must be given", "");
    PROCESS_EXIT();
}
}else
{
    shell_output_str(&math_command,
        "math number1 number2: number must be given", "");
    PROCESS_EXIT();
}
int s = n1 + n2;
sprintf(buf, "%d + %d= %d", n1, n2, s);
shell_output_str(&math_command, buf, "");

PROCESS_END();
*/

void
shell_math_init(void)

118
{  
    shell_register_command(&math_command);
}

/*----------------------------------------------------------*/

Input data from the user can be obtained from the data variables in the function parameters. We parse the data into two number variables.

Next, we write a header file, called shell-math.h, for our Contiki shell API. We only declare our function, shell_math_init(). The following is the complete program for the shell-math.h file:

```c
#ifndef __SHELL_MATH_H__
#define __SHELL_MATH_H__

#include "shell.h"

void shell_math_init(void);

#endif
```

Save all the files.

Now, we should configure Makefile to include these files in compilation. You can open Makefile.shell in the same folder with the shell API files. Open this file and add your shell files. You can see them in the codes here:

```sh
...

shell_src = shell.c shell-reboot.c shell-vars.c shell-ps.c \  
  shell-blink.c shell-text.c shell-time.c \  
  shell-file.c shell-run.c \  
  shell-coffee.c \  
  shell-power.c \  
  shell-base64.c \  
  shell-memdebug.c \
```
Our Contiki shell API is now ready for compiling.

The next step is to develop a Contiki application to use our Contiki shell API, math shell. We create a folder, called shell-math-demo. Then, we create two files:

- shell-math-demo.c
- Makefile

The shell-math-demo.c file consists of a program to use the Contiki math shell API. The following is the complete program in the shell-math-demo.c file:

```c
#include "contiki.h"
#include "shell.h"
#include "serial-shell.h"
#include "collect-view.h"

PROCESS(sky_shell_process, "Sky Contiki shell");
AUTOSTART_PROCESSES(&sky_shell_process);

#define WITH_PERIODIC_DEBUG 0
#if WITH_PERIODIC_DEBUG
static struct ctimer debug_timer;
static void
periodic_debug(void *ptr)
{
```
ctimer_set(&debug_timer, 20 * CLOCK_SECOND, periodic_debug, NULL);
collect_print_stats();
}
#endif /* WITH_PERIODIC_DEBUG */
/*-----------------------------------------------------------*/
PROCESS_THREAD(sky_shell_process, ev, data)
{
    PROCESS_BEGIN();

    #if WITH_PERIODIC_DEBUG
    ctimer_set(&debug_timer, 20 * CLOCK_SECOND, periodic_debug, NULL);
    #endif /* WITH_PERIODIC_DEBUG */

    serial_shell_init();
    shell_blink_init();
    shell_reboot_init();
    shell_sky_init();
    shell_power_init();
    shell_powertrace_init();
    shell_text_init();
    shell_time_init();
    shell_collect_view_init();
    shell_math_init();

    PROCESS_END();
} /*-----------------------------------------------------------*/
CHAPTER 4  CONTIKI-NG AND COMPUTER COMMUNICATION

Furthermore, we write scripts in the Makefile. We include Contiki and the project path. You write these complete scripts as follows:

```
CONTIKI_PROJECT = shell-math-demo
all: $(CONTIKI_PROJECT)

APPS = serial-shell powertrace collect-view
CONTIKI = ../..

CONTIKI_WITH_RIME = 1
include $(CONTIKI)/Makefile.include
```

CONTIKI_WITH_RIME = 1 is used to enable the RIME protocol since Contiki shell uses it on some shell commands. We also supply a testing configuration by including this Makefile, `/home/user/nes/testbed-scripts/Makefile.include`, because some shell commands need it.

Save all files.

Now, we compile our program, shell-math-demo. Open Terminal and navigate to the shell-math-demo folder. Type these commands:

```
$ make TARGET=sky
$ make shell-math-demo.sky.upload TARGET=sky
```

After our program is uploaded to Contiki mote, we can monitor the mote. You can type this command:

```
$ make login
```

If it succeeds, you should see the Contiki shell. Now, we can call our Contiki shell API. We can type this command in the Contiki shell:

```
Contiki> math 10 5
```

This shell will call our Contiki shell API. Value inputs 10 and 5 will be executed to perform the addition operation. You can see a sample of the program output in Figure 4-16.
Custom Contiki-NG Shell

Contiki-NG applies shell with a different approach. To customize a shell, we can add our shell command to the shell-commands.c file. You can find that file in the `<contiki-ng-root>/os/services/shell` folder. You can see it in Figure 4-17.

We add a new shell command to the shell-commands.c file. We define a “hello” command. We print the message "Hi, this is a custom shell" to Terminal. Write this code:

```
static
PT_THREAD(cmd_hello(struct pt *pt, shell_output_func output, char *args))
{
    PT_BEGIN(pt);
```
CHAPTER 4  CONTIKI-NG AND COMPUTER COMMUNICATION

SHELL_OUTPUT(output, "Hi, this custom shell
")

PT_END(pt);

}  

Next, we also register our shell in the shell_command_t shell_commands struct. We define the shell name, method call, and description. For instance, you can see the code for our custom NG-shell here:

struct shell_command_t shell_commands[] = {
    { "hello",   cmd_hello,   "'> hello': say hello" },
    { "help",    cmd_help,    "'> help': Shows this help" },
    { "reboot",  cmd_reboot,  "'> reboot': Reboot the board by watchdog_reboot()" },
...  
};

Save all files.
Now, you can use the NG shell on your project. For instance, we can use a hello-world project and enable the Contiki-NG shell. We only enable shell service on Makefile:

```
MODULES += os/services/shell
```

After that, we compile and flash this program onto the Contiki-NG mote. You can now remote into the Contiki mote Terminal using `mote login`. For instance, I remote access my TI LaunchPad CC2650 using this command:

```
$ make login TARGET=srf06-cc26xx PORT=/dev/ttyUSB0
```
You should change TARGET and PORT to reflect your Contiki board. Please press the Reset button on the Contiki mote if you do not see anything in Terminal. Now, you can test your own shell:

```
shell> help
shell> hello
```

After calling our NG shell, we can see the shell response. You can see my shell output in Figure 4-18.

![Shell Output](image)

**Figure 4-18.** Running custom NG shell on TI LaunchPad CC260

### Communication among Contiki Motes

In this section, we will build a communication among Contiki-NG motes. There are a lot of methods for communicating among Contiki-NG motes. To show how to communicate among Contiki motes, we use broadcast via UDP protocol in Contiki-NG OS. UDP is a communication stack that provides a set of lightweight communication primitives ranging from local-area broadcast to reliable network flooding.
For this demo, we need at least two Contiki-NG motes. One mote will act as a sender and the other mote will be a receiver. You can see our demo scenario in Figure 4-19.

**Figure 4-19. Communication among Contiki-NG motes**

Our demo scenario is that a mote sends data to all motes. If a mote receives data, it will be shown in Terminal. We use a program sample from Contiki-NG.

**Sending Broadcast Messages**

The objective of a mote sender is to broadcast data to other motes. For this demo, we use the `simple-udp` module that is located in the `<contiki-ng-root>/os/net` folder. We can use the `simple_udp_sendto()` function to broadcast a message. This method is defined in the `simple-udp.h` header file:

```c
int simple_udp_sendto_port(struct simple_udp_connection *c,
    const void *data, uint16_t datalen,
    const uip_ipaddr_t *to, uint16_t to_port);
```
Note:

- `simple_udp_connection` is a simple-udp object
- `data` is the data that will be sent
- `datalen` is the length of data
- `to` is the IP address of the target
- `to_port` is the port of the target

For instance, we send data to a specific IP address:

```c
simple_udp_sendto(&udp_conn, &count, sizeof(count), &dag->dag_id);
```

## Receiving Broadcast Messages

A mote receiver listens for an incoming message that is sent by a mote sender. To build a mote receiver, we can listen for broadcast messages by creating a callback/event function. We can use `simple_udp_register()` to register our callback function.

For instance, we listen for incoming broadcast messages using this code:

```c
static void
udp_rx_callback(struct simple_udp_connection *c, 
                 const uip_ipaddr_t *sender_addr, 
                 uint16_t sender_port, 
                 const uip_ipaddr_t *receiver_addr, 
                 uint16_t receiver_port, 
                 const uint8_t *data, 
                 uint16_t datalen)
{
    unsigned count = *(unsigned *)data;
    LOG_INFO("Received response %u from ", count);
```
LOG_INFO_6ADDR(sender_addr);
LOG_INFO_("\n");
}
...
simple_udp_register(&udp_conn, UDP_CLIENT_PORT, NULL,
    UDP_SERVER_PORT, udp_rx_callback);

Demo: Middleware Application

Now, we implement our sender and receiver program for Contiki-NG motes. We use a program sample from Contiki-NG, rpl-udp. You can find this project in the <contiki-ng-root>/examples/ folder.

The rpl-udp project consists of two programs, udp-client.c and udp-server.c. The UDP client app (udp-client.c) will send and receive broadcast messages. The UDP server (udp-server.c) will listen for incoming broadcast messages.

In the Makefile file, we configure our project and Contiki-NG paths. Write these scripts for the Makefile file:

all: udp-client udp-server

.PHONY: renode
renode: all
ifeq ($(TARGET),cc2538dk)
    $(error Only the cc2538dk TARGET is supported for Renode demo scripts)
endif
ifndef SCRIPT
    $(warning SCRIPT not defined! Using "rpl-udp.resc" as default)
CHAPTER 4  CONTIKI-NG AND COMPUTER COMMUNICATION

renode rpl-udp.resc
else
ifeq ($(wildcard $(SCRIPT)),)
  $(error SCRIPT "$(SCRIPT)" does not exist!)
endif
renode $(SCRIPT)
endif

CONTIKI=../..
include $(CONTIKI)/Makefile.include

Save all files.

Since we need at least two Contiki-NG motes, we should know the serial ports that are used by our Contiki-NG motes. You can check this using this command:

$ ls /dev/ttyUSB*

You should see a list of serial ports that are used by the Contiki-NG motes. For instance, my two TelosB motes are detected, shown in Figure 4-20.

Figure 4-20. Getting a list of connected Contiki-NG motes
To compile and upload the program to a specific Contiki-NG mote, we should pass MOTES with the serial port of the targeted Contiki-NG mote. For instance, we flash a program to a Contiki mote on serial port /dev/ttyUSB0. You can type these commands:

$ make TARGET=sky
$ make TARGET=sky savetarget
$ make udp-client.upload TARGET=sky MOTES=/dev/ttyUSB0

For the second Contiki-NG mote, you can type these commands:

$ make udp-server.upload TARGET=sky MOTES=/dev/ttyUSB1

After all programs are uploaded to the Contiki-NG motes, we can monitor the data exchange among Contiki-NG motes. For Contiki-NG mote 1, we can monitor messages using this command:

$ make login TARGET=sky MOTES=/dev/ttyUSB0

For Contiki-NG mote 2, we can execute this command to see exchange data in Terminal. Type this command:

$ make login TARGET=sky MOTES=/dev/ttyUSB1

You should see an incoming message in Terminal. You can see it in Figure 4-21.
In this section, we will explore using middleware for a Contiki-NG application. In our scenario, WSN motes will broadcast sensor data. Then, we will use a Java tool application from Contiki-NG to read these sensor data on a computer. The result of reading this data will be displayed in Terminal.

Let’s build!

Figure 4-21. Running Contiki-NG application on mote 1 and mote 2
What Is Middleware?

Middleware is a “bridge” application that connects Contiki-NG motes to other systems in internal and external networks. A sink can be represented as middleware. You can see a general architecture scenario of middleware for a Contiki-NG application in Figure 4-22.

Implementing middleware for a Contiki-NG application could enhance Contiki-NG’s capabilities. As you know, Contiki-NG applications have a limitation in protocol stacks since they have limited hardware resources. Middleware can act as a bridge to connect Contiki-NG motes to other systems, such as database servers, external servers, and RESTful servers.

Middleware Architecture for Contiki-NG

In this section, we will build simple middleware for a Contiki-NG application. For this demo scenario, the middleware will listen for sensor data and print them to Terminal using Python. A general design for our demo is shown in Figure 4-23.
Implementation

To implement our demo, we use our previous program, rpl-udp (<contiki-ng_root/examples/rpl-udp), to be flashed into the Contiki-NG motes. We also use a previous Python program (section: “Access Contiki Motes via Serial”) to listen on the serial port contiki-viewer.py.

Testing

First, we compile the program and upload it to the motes. For instance, I have two WSN motes that are attached on serial ports /dev/USB0 and /dev/USB1. Then, we run contiki-viewer.py with a specific port. You can see the program output in Figure 4-24. In the next chapter, we will work more with middleware.
Summary

We have learned to communicate among Contiki-NG motes. We also developed an application to communicate with Contiki-NG motes from a computer. Finally, we built simple middleware for a Contiki-NG application using a Python application tool.

In the next chapter, we will learn to focus on sensing and actuating in Contiki-NG motes.
CHAPTER 5

Sensing and Actuating

Sensing and actuating are core activities in the Wireless Sensor Network lifecycle. Most WSN makers build WSN motes that include sensor or/and actuator devices. In this chapter, we will explore various sensor and actuator devices. Moreover, we will build a Contiki-NG program to access these sensor devices and control the actuator devices.

The following is a list of topics we will cover in this chapter:

- What are sensing and actuating?
- Reviewing sensors and actuators
- Sensing in Contiki-NG
- Actuating in Contiki-NG
- Customizing sensor and actuator devices

What Are Sensing and Actuating?

Sensing and actuating are two common terms that are used in embedded topics. In the WSN context, sensing is a process that converts a physical object to digital data. For instance, sensing temperature. The sensor device senses the environment’s temperature and then converts it to digital form. Actuating is a process in which the MCU sends digital data
to an actuator device to perform some task, such as turning on LED lights, sounds, or a motor.

**Figure 5-1. Sensing and actuating in a WSN mote**

Sensing and actuating are shown in Figure 5-1. Sensor devices that are attached to a WSN mote will sense and then convert the information to digital data form. A WSN mote also can send digital data to the outside environment through actuator devices. Both sensor and actuator devices can communicate to the MCU via I/O interfaces. These interfaces provide various protocols depending on how they are implemented. You can see a communication model in Figure 5-2.

**Figure 5-2. Communication model for sensor/actuator devices and MCU**
We will start to explore sensor and actuator devices in the next section. Then, we will develop a Contiki-NG program for accessing those devices.

**Review Sensor and Actuator Devices**

In this section, we will explore common sensor and actuator devices that are used in embedded development environments, including in real applications. All sensor and actuator devices may be found in your local store. However, you can buy them in online stores such as Element14, Digikey, Mouser, SparkFun, Adafruit, DFRobot, SeeedStudio, and more. You can also find them on cheap online stores from China, like Alibaba, Aliexpress, Banggood, and DealeXtreme.

We will now review some sensor and actuator devices that are easier to find. Sensor device samples are temperature, humidity, soil moisture, and gas sensor. Actuator device samples can be LED, active buzzer, and motor. We will check them in the next section.

**Temperature and Humidity**

Temperature and humidity sensors are used to measure current environment temperature and humidity levels. Some manufactures make these sensors into one chip, but others are still available in separation sensors for temperature and humidity.

Most WSN motes are designed to include these temperature and humidity sensors. This is useful for our development and testing. We can verify that our program performs sensing and actuating. One example is the SHT1x (SHT10, SHT11, SHT15) chip from Sensirion. You can read its datasheet at [https://www.sparkfun.com/datasheets/Sensors/SHT1x_datasheet.pdf](https://www.sparkfun.com/datasheets/Sensors/SHT1x_datasheet.pdf). SparkFun provides the SHT15 sensor in a module that is ready to use. It’s the SparkFun Humidity and Temperature Sensor.
Breakout—SHT15. This product can be bought on the SparkFun website at https://www.sparkfun.com/products/13683. You can see the module in Figure 5-3.

**Figure 5-3.** SparkFun Humidity and Temperature Sensor Breakout—SHT15

In general, SHT1x provides two-wire pins to be used to access sensor data. If you see a SparkFun Humidity and Temperature Sensor Breakout (Figure 5-3), this module has four pins: VCC, GND, DATA, and SCK. We can develop a program to access this sensor data from MCU.

We also can use a low-cost humidity and temperature sensor. It’s the DHT22 module. This sensor is easier to find. You can check it out on the SparkFun website at https://www.sparkfun.com/products/10167; it is called Humidity and Temperature Sensor—RHT03 (DHT22). The DHT22 sensor provides a single-wire digital interface that is used to access sensor data for temperature and humidity. The DHT22 sensor form is shown in Figure 5-4.
Soil Moisture

A soil moisture sensor can be used to measure the moisture level in soil. This sensor can be applied in monitoring systems for gardens. There are many soil-moisture models that you can use in your design. One with a low cost is a Soil Moisture Sensor from SparkFun, found at https://www.sparkfun.com/products/13322. The SparkFun Soil Moisture Sensor can be attached via an analog pin (ADC pin) to obtain the moisture level. You can see a SparkFun Soil Moisture Sensor in Figure 5-5.
In some scenarios, you may need to monitor for gas such as carbon monoxide (CO). You can attach a carbon monoxide sensor to a WSN mote through an analog pin so as to detect it. One such sensor is Carbon Monoxide Sensor (MQ-7). This sensor can detect gas concentrations anywhere from 20 to 2000 ppm. You can find it on the SparkFun website at https://www.sparkfun.com/products/9403. Figure 5-6 shows the Carbon Monoxide Sensor (MQ-7). Further information about this gas sensor (MQ-7) can be found at https://cdn.sparkfun.com/datasheets/Sensors/Biometric/MQ-7%20Ver1.3%20-%20Manual.pdf.
LED

An LED is a simple actuator device. It can be used for lighting indicator and notification, processing status, or indicating a certain state. There are various models and colors for LEDs. You can choose it to fit with your case. Figure 5-7 shows a sample LED.

Figure 5-7. A LED with red color in 5mm size
(Source: https://www.sparkfun.com/products/9590)
Active Buzzer

Sometimes you need an actuator that generates a continuous sound to inform the user of a particular state, such as low power on battery, problem on a certain system module, or waiting for an action. An active buzzer can be used as an actuator device to indicate a certain notification. This actuator is low cost and easier to find.

In general, an active buzzer has two pins, GND and SIG. It’s easy to use because we just send a digital value (3.3V or 5V) on the SIG pin to generate a continuous sound. The sound will be stopped if we set 0V (GND) on the SIG pin. You can see this actuator in Figure 5-8.

![Active buzzer](image)

Figure 5-8. Active buzzer

Motor

To control mechanical stuff, you may need a motor that is designed for the MCU to make a movement or rotation. There are various motor models. A servo motor is a basic motor that can be integrated with a WSN mote. You should keep in mind that some motor models need a lot of power, and they should not get that power from your WSN mote. You can use an external power adapter for your motor. The manufacturer also provides a motor driver that addresses motor power issues.

One servo motor is the Hitec HS-5035HD servo with Ultra Nano Size. You can buy it on the SparkFun website at [https://www.sparkfun.com/products/14210](https://www.sparkfun.com/products/14210). Figure 5-9 shows a Hitec HS-5035HD servo.
Sensing in Contiki-NG

Each sensor in Contiki-NG should implement sensors.h from the <contiki-ng-root>/os/lib folder. We can access the sensor device library using the following code:

```c
SENSORS_ACTIVATE(sensor_xyz);
...
val = sensor_xyz.value(SENSOR_XYZ_TYPE);
```

`sensor_xyz` is a defined variable for the sensor device. This is a part of each Contiki-NG device platform. Each sensor device has sensor types that will be called on the `value()` method. Your sensor code (*.c and *.h) should be put in the `dev/` folder from the Contiki-NG platform. You can see a list of sensor codes in Figure 5-10.

Figure 5-9. Hitec HS-5035HD servo with Ultra Nano Size
In the next section, we will build a simple demo for sensing using the existing sensor in Contiki-NG. I use a TelosB for testing.

**Demo**

We will build a Contiki-NG application to read the current temperature and humidity via a Contiki-NG mote. For testing, I use TelosB as Contiki-NG mote. This board provides an SHT11 sensor that can sense temperature and humidity. You can read the sensor datasheet at https://www.sensirion.com/en/environmental-sensors/humidity-sensors/digital-humidity-sensors-for-accurate-measurements/.

For this demo, we will build the scenario that is shown in Figure 5-11. A Contiki-NG mote with sensor device is attached to a computer. We will sense temperature and humidity via the SHT11 sensor. After acquiring
the temperature, the mote will send this sensor data to a serial port on
the computer. We will print this sensor data in Terminal by reading the
serial port.

![Figure 5-11. Sensing demo scenario](image)

**Creating a Project**

To create a new project on Contiki-NG, you can create a folder, for
instance, sensing. Then, add sensing.c and Makefile files. Our program
that uses the sensor device will be implemented in the sensing.c file. We
define our project configuration in the Makefile file.

The following is the content of the Makefile file:

```makefile
CONTIKI_PROJECT = sensing
all: $(CONTIKI_PROJECT)

CONTIKI = /home/agusk/contiki-ng/
include $(CONTIKI)/Makefile.include
```

Change CONTIKI to your Contiki-NG path.

Next, we will write a Contiki-NG program on the sensing.c file.
Writing a Program

We will build a Contiki-NG program to acquire the temperature every five seconds. We use \texttt{etimer} to implement our timer object.

The following is the complete program in the \texttt{sensing.c} file:

```c
#include "contiki.h"
#include "dev/sht11/sht11-sensor.h"

#include <math.h>
#include <stdio.h> /* For printf() */
/*----------------------------------------*/
PROCESS(sensing_process, "Sensing process");
AUTOSTART_PROCESSES(&sensing_process);
/*----------------------------------------*/
PROCESS_THREAD(sensing_process, ev, data)
{
    static struct etimer et;
    static int val;
    static float s = 0;
    static int dec;
    static float frac;

    PROCESS_BEGIN();

    printf("Demo sensing...\n");
    while(1)
    {
        etimer_set(&et, CLOCK_SECOND * 5);
        SENSORS_ACTIVATE(sht11_sensor);

        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

        val = sht11_sensor.value(SHT11_SENSOR_TEMP);
        if(val != -1)
```
{ 
    s = ((0.01*val) - 39.60);
    dec = s;
    frac = s - dec;
    // print float data
    printf("Temperature=%d.%02u C . VAL=%d\n", dec, (unsigned int)(frac * 100),val);
}

val=sht11_sensor.value(SHT11_SENSOR_HUMIDITY);
if(val != -1)
{
    s= (((0.0405*val) - 4) + ((-2.8 * 0.000001)*(pow(val,2))));
    dec = s;
    frac = s - dec;
    // print float data
    printf("Humidity=%d.%02u %%. VAL=%d\n", dec, (unsigned int)(frac * 100),val);
}

etimer_reset(&et);
SENSORS_DEACTIVATE(sht11_sensor);

PROCESS_END();
} /*----------------------------------------------------------*/
How does it work?

Firstly, we activate the timer and our SHT11 sensor on the TelosB:

```c
etimer_set(&et, CLOCK_SECOND * 5);
SENSORS_ACTIVATE(sht11_sensor);
PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
```

After etimer has raised the time, we sense temperature by calling `value()` from the `sht11_sensor` object and passing the `SHT11_SENSOR_TEMP` parameter. Since we face displaying float data, we calculate decimal and fraction separately:

```c
val = sht11_sensor.value(SHT11_SENSOR_TEMP);
if(val != -1)
{
    s = ((0.01*val) - 39.60);
    dec = s;
    frac = s - dec;
    // print float data
    printf("Temperature=%d.%02u C . VAL=%d\n", dec, (unsigned int)(frac * 100),val);
}
```

We also do a similar task to sense humidity. We pass `SHT11_SENSOR_HUMIDITY` to read humidity from sensor:

```c
val = sht11_sensor.value(SHT11_SENSOR_HUMIDITY);
if(val != -1)
{
    s = (((0.0405*val) - 4) + ((-2.8 * 0.000001)*(pow(val,2))));
    dec = s;
    frac = s - dec;
```
// print float data
printf("Humidity=%d.%02u %% . VAL=%d\n", dec, (unsigned int) (frac * 100), val);
}

Last, we reset our timer and sensor objects:

etimer_reset(&et);
SENSORS_DEACTIVATE(sht11_sensor);

**Testing**

Now, you can test the program. You should compile and deploy this program into Contiki-NG:

$ make TARGET=sky
$ make TARGET=sky savetarget
$ make sensing.upload TARGET=sky

After uploading a program into Contiki-NG, we try to monitor the serial output from the Contiki-NG mote. You can type this command to do so:

$ make login TARGET=sky

If it succeeds, you should see the current temperate and humidity values in Terminal. You can see my program output in Figure 5-12.
In this section, we will learn how to work with actuating in Contiki-NG. There are a lot of actuator devices that you can use with Contiki-NG. For this demo, I will use a button and an LED as actuator devices. In general, these actuator devices are available on some Contiki-NG motes. This demo is shown in Figure 5-13. If the user presses the button, the LED will be toggled. I use TelosB for this demo as it already has a button and an LED.
Creating a Project

You can start a new project by creating a folder, called actuating. Then, add Makefile and actuating.c files. In the Makefile file, we configure our project. You can write the following Makefile content:

```
CONTIKI_PROJECT = actuating
all: $(CONTIKI_PROJECT)

CONTIKI = /home/agusk/contiki-ng/
include $(CONTIKI)/Makefile.include
```

Change CONTIKI to your Contiki-NG path.
Next, we will build a program in the actuating.c file.
Writing a Program

The program will listen for the pressed button from the user. If the user presses the button, we toggle the LED. The following is the content of the actuating.c file:

```c
#include "contiki.h"
#include "leds.h"
#include "dev/button-sensor.h"
#include <stdio.h>

PROCESS(sensing_process, "Sensing process");
AUTOSTART_PROCESSES(&sensing_process);

PROCESS_THREAD(sensing_process, ev, data)
{
    PROCESS_BEGIN();

    printf("Demo actuating...\n");
    SENSORS_ACTIVATE(button_sensor);
    leds_off(LEDS_ALL);

    while(1)
    {
        PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event &&
                                 data == &button_sensor);

        leds_toggle(LEDS_BLUE);
    }
    SENSORS_DEACTIVATE(button_sensor);
    PROCESS_END();
}
```

Chapter 5  Sensing and Actuating
How does it work?
First, we activate the button by calling SENSOR_ACTIVATE(). We then turn all LEDs off for initialization.

```
SENSORS_ACTIVATE(button_sensor);
leds_off(LEDS_ALL);
```

Then, we wait for the button to be pressed by the user. We can use PROCESS_WAIT_EVENT_UNTIL() to detect the pressed button. We will toggle the LED after the button is pressed:

```
while(1)
{
    PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event &&
                            data == &button_sensor);

    leds_toggle(LEDS_BLUE);
}
```

## Testing

Now, you can test the program. You should compile and deploy this program into Contiki-NG:

```
$ make TARGET=sky
$ make TARGET=sky savetarget
$ make actuating.upload TARGET=sky
```

You can press a user button on the Contiki-NG mote. You will see an LED light up while you press the button. You can see the user button and the LED on a TelosB in Figure 5-14.
Customizing Sensor and Actuator Devices

You probably want to expand your sensors and actuators on your Contiki-NG mote. In the real world, some Contiki-NG motes do not provide built-in sensors and actuators. Others only have one or two sensor devices.

In this section, we will explore how to add additional sensor and actuator devices to Contiki-NG motes.

Expansion Connector

Some Contiki-NG motes are designed to enable you to expand their board. One option is to provide an expansion connector. The board design exposes MCU pins in order to make additional external sensors, actuators, or other devices interact with the system.
For instance, TelosB has an expansion connector available to enable makers to attach sensors and actuators to the board. You can see it in Figure 5-15.

![Expansion connector on TelosB](image)

**Figure 5-15.** Expansion connector on TelosB

I bought my TelosB from ADVANTICSYS. This product is MTM-CM5000-MSP. Based on its document, found at [https://www.advanticsys.com/shop/mtmcm5000msp-p-14.html](https://www.advanticsys.com/shop/mtmcm5000msp-p-14.html), these expansion connectors are as depicted in Figure 5-16.

As seen in Figure 5-16, we can attach sensor and actuator devices to the mote. GPIO pins such as ADC and I2C are accessible from the main board. You can do wiring for your sensor and actuator devices.
Another board, such as Zoul from Zolertia, https://zolertia.io/zoul-module/, can be attached to your own board. Zolertia also provides a complete development kit, Firefly. You can develop Contiki-NG on top of the board. Further information about Firefly can be found at https://zolertia.io/product/firefly/. Figure 5-17 shows a form of the Firefly board.
Each sensor or actuator device attached to Contiki-NG should provide an API driver. In this section, we will explore how to make device drivers for Contiki-NG.

Each Contiki-NG mote platform has a different development style. You should decide what Contiki-NG platform you will use for additional sensor and actuator devices. If you have a plan to develop sensors and actuators for all platforms, you can put your drivers at <contiki-ng-root>/arch/dev. Otherwise, you can put them at <contiki-ng-root>/arch/platforms/<your-platform>. You can see it in Figure 5-18.
For demo purposes, we will develop a driver for the Sky platform. We build a sensor that is attached into TelosB via an ADC0 pin. Next, we will write a driver program for the Sky platform.

We add two files, mycustom-sensor.h and mycustom-sensor.c. These files are put in the `<contiki-ng-root>/arch/platform/sky/dev` folder. You can see them in Figure 5-19. We extend our custom sensor driver from the sensors.h and sky-sensors.h files.

The following is the content of the mycustom-sensor.h file:

```c
#ifndef MYCUSTOM_SENSOR_H_
#define MYCUSTOM_SENSOR_H_

#include "lib/sensors.h"

extern const struct sensors_sensor mycustom_sensor;

#define MY_CUSTOM_SENSOR 0

#endif /* MYCUSTOM_SENSOR_H_ */
```

Figure 5-18. Platform API on Contiki-NG
Now we implement the mycustom-sensor.c file. The ADC0 pin is attached on ADC12MEM0. Since we implement sky-sensors.h, we should implement the value(), configure(), and status() methods.

The following is the content of the mycustom-sensor.c file:

```c
#include "contiki.h"
#include "dev/mycustom-sensor.h"
#include "dev/sky-sensors.h"

#define INPUT_CHANNEL   (1 << INCH_11)
#define INPUT_REFERENCE SREF_0
#define MYCUSTOM_MEM    ADC12MEM0
```

**Figure 5-19. Adding driver files into Sky platform**

Now we implement the mycustom-sensor.c file. The ADC0 pin is attached on ADC12MEM0. Since we implement sky-sensors.h, we should implement the value(), configure(), and status() methods.

The following is the content of the mycustom-sensor.c file:

```c
#include "contiki.h"
#include "dev/mycustom-sensor.h"
#include "dev/sky-sensors.h"

#define INPUT_CHANNEL   (1 << INCH_11)
#define INPUT_REFERENCE SREF_0
#define MYCUSTOM_MEM    ADC12MEM0
```
const struct sensors_sensor mycustom_sensor;

/*-----------------------------------------------------------*/
static int
value(int type)
{
    switch(type) {
        case MY_CUSTOM_SENSOR:
            return MYCUSTOM_MEM;
    }
    return 0;
}
/*-----------------------------------------------------------*/
static int
configure(int type, int c)
{
    return sky_sensors_configure(INPUT_CHANNEL, INPUT_REFERENCE,
        type, c);
}
/*-----------------------------------------------------------*/
static int
status(int type)
{
    return sky_sensors_status(INPUT_CHANNEL, type);
}
/*-----------------------------------------------------------*/
SENSORS_SENSOR(mycustom_sensor, "MYCUSTOMSENSOR", value,
    configure, status);

Last, we should add our driver file into the Makefile.sky file from the
Sky platform. It is located at <contiki-ng-root>/arch/platform/sky/
Makefile.sky. In our case, we move the mycustom-sensor.c file into the
Makefile.sky file. You can see the bold code for adding the driver file here:
CONTIKI_TARGET_SOURCEFILES += contiki-sky-platform.c \
    sht11.c sht11-sensor.c light-sensor.c battery-sensor.c \ 
    button-sensor.c mycustom-sensor.c

include $(CONTIKI)/arch/platform/sky/Makefile.common

MODULES += os/net/mac os/net/mac/framer os/net \ 
    arch/dev/cc2420 arch/dev/sht11 arch/dev/ds2411 \ 
    os/storage/cfs

Your custom sensor driver is ready for Sky platform. You can use it as usual. For instance, you create a project by creating a folder, custom-sensing. Then, you add Makefile and custom-sensing.c files.

We will access our driver, mycustom-sensor, in our project. We use the same program from the sensing project but change it to use our own sensor. The following is the complete program for the custom-sensing.c file:

```c
#include "contiki.h"
#include "dev/mycustom-sensor.h"

#include <stdio.h> /* For printf() */
/*-----------------------------------------------*/
PROCESS(sensing_process, "Sensing process");
AUTOSTART_PROCESSES(&sensing_process);
/*-----------------------------------------------*/
PROCESS_THREAD(sensing_process, ev, data)
{
    static struct etimer et;
    static int val;

    PROCESS_BEGIN();

    printf("Demo sensing...
");
    while(1)
    {
```
etimer_set(&et, CLOCK_SECOND * 5);
SENSORS_ACTIVATE(mycustom_sensor);

PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

val = mycustom_sensor.value(MY_CUSTOM_SENSOR);
if(val != -1)
{
    printf("CUSTOM SENSOR VAL=%d\n", val);
}

etimer_reset(&et);
SENSORS_DEACTIVATE(mycustom_sensor);

}

PROCESS_END();

Last, we should configure our project in the Makefile file:

CONTIKI_PROJECT = custom-sensing
all: $(CONTIKI_PROJECT)

CONTIKI = /home/agusk/contiki-ng/
include $(CONTIKI)/Makefile.include

Change the CONTIKI value to your Contiki-NG path. Now, you can compile and upload this program to TelosB:

$ make TARGET=sky
$ make TARGET=sky savetarget
$ make custom-sensing.upload TARGET=sky
After uploading a program into Contiki-NG, we try to monitor the serial output to see our custom sensor on Terminal:

$ make login TARGET=sky

You should see your reading values from ADC on Terminal.

Summary

We have reviewed some sensor and actuator devices. We also have learned how to work with sensing and actuating in Contiki-NG. Last, we developed a custom sensor to attach to a Contiki-NG mote through its expansion connector.

In the next chapter, we will learn how to build networking and communication in Contiki-NG. That is at the core of Contiki-NG’s features.
CHAPTER 6

Networking

Contiki-NG comes with rich network-stack features to allow communication with others. In this chapter, we will explore the available network features on the Contiki-NG platform. Several scenarios will be provided to enable practice with implementing projects-based Contiki-NG, either in physical motes or in mote simulations.

The following is a list of topics that will be covered in this chapter:

- Networking in Contiki-NG
- Working with network simulation using COOJA
- IPv6 networking
- Routing on Contiki-NG
- IPv6 Multicast
- Working with Contiki-NG NullNet
- Working with a 6LoWPAN network
- Building a RESTful server for Contiki-NG

© Agus Kurniawan 2018
A. Kurniawan, Practical Contiki-NG, https://doi.org/10.1007/978-1-4842-3408-2_6

167
Networking in Contiki-NG

Contiki-NG still uses a traditional OSI (Open Systems Interconnection) stack to implement the Contiki-NG Network Protocol stack, called NETSTACK, to communicate among nodes. The Contiki-NG NETSTACK is shown in Figure 6-1. In this figure, we can see that the Contiki-NG NETSTACK implements four layers, as follows:

- Network layer (NETSTACK_NETWORK)
- MAC layer (NETSTACK_MAC)
- RDC (Radio Duty Cycling) layer (NETSTACK_RDC)
- Radio layer (NETSTACK_RADIO)

The Network layer (NETSTACK_NETWORK) in an OSI layer can be represented as Application, Transport, Network, Routing, and Adaptation. I will introduce each layer in the next section.

**Figure 6-1.** Contiki-NG Network Protocol stack
In NETSTACK implementation, Contiki-NG provides network libraries. You can find them in the `<contiki-ng>/os/net` folder. You can see this folder in Figure 6-2. The following is several code-sample implementations for Contiki-NG NETSTACK:

- **Application layer:** `http-socket.c, websocket.c, websocket-http-client.c, and mqtt.c`
- **Transport:** `udp-socket.c` and `tcp-socket.c`
- **Network & Routing:** `uip6.c` and `rpl.c`
- **MAC:** `mac.c` and `csma.c`

I recommend you review and learn the code for the Contiki-NG network stack, found in the `<contiki-ng>/os/net` folder. Thus, you will get more knowledge about how Network project builds Contiki-NG system.
Network Layer

Contiki-NG relies on an IPv6 stack. All TCP/UDP sockets in Contiki-NG use uIP (uiP.h and uiP.c from `<contiki-ng>/os/net/ipv6`), which implements for IP, UDP, and TCP protocols in minimized models.

Currently, the network layer contains two sublayers, the upper IPv6 layer and the lower adaptation layer. These sublayers run on the top of IEEE 802.15.4 with Time-Slotted Channel Hopping (TSCH).

Regarding routing, Contiki-NG applies RPL (Routing Protocol for Low-power and Lossy Networks (LLNs)), which adopts the RFC standard, RFC 6550. RPL develops a routing graph from the root node or AP (Access Point).
If the routing graph has a form as cyclic graph and is built from a root node, it is called a DODAG (Destination Oriented Directed Acyclic Graph). A DODAG routing graph form can be seen in Figure 6-3.

**Figure 6-3. DODAG routing graph form**

RPL routing in Contiki-NG supports three directions of traffic, as follows:

- Upward: from any node toward a root
- Downward: from the root to any node
- Any-to-any: flows among arbitrary pairs of nodes in the DODAG graph

For RPL implementation, Contiki-NG provides RPL classic and RPL lite. RPL classic is the original Contiki RPL implementation, called ContikiRPL. I recommend you read the ContikiRPL paper on this site: [http://www.diva-portal.org/smash/get/diva2:1042739/FULLTEXT01.pdf](http://www.diva-portal.org/smash/get/diva2:1042739/FULLTEXT01.pdf).

You can find code implementations for both RPL classic and RPL lite in Contiki-NG. RPL classic can be found at `/net/rpl-classic` and RPL lite at `/net/rpl-lite` from the Contiki-NG code root. See them in Figure 6-4.
The nullnet library from Contiki-NG can be used to test your packet from upper to lower layers. This library can be found at <contiki-ng>/os/net/nullnet.

**MAC Layer**

The MAC layer is designed to address collisions in packet traffic and to apply back-off if there is traffic. Contiki-NG applies CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) for MAC layer implementation. Contiki-NG uses CSMA/CA on the IEEE 802.15.4 protocol. You can see program implementation in the <contiki-ng>/os/net/mac folder.

In the CSMA/CA algorithm, a mote will sense the medium before sending packets. If another mote is sending a packet, the mote will apply back-off with a certain value depending on the RDC layer. If the medium is free, the mote will send packets that have been prepared by the network layer.

Contiki-NG also provides nullmac for testing that is a part of nullnet from the network layer. nullmac will forward packets from the upper layer to the radio driver and vice versa.
The Radio Duty Cycling (RDC) layer saves energy by allowing a node to keep its radio transceiver off most of the time. Contiki-NG supports the ContikiMAC protocol based on the principles behind low-power listening. ContikiMAC uses Time Slotted Channel Hopping (TSCH) that is a part of the MAC layer of the IEEE 802.15.4e-2012 amendment.

Radio Layer

The radio layer is the lowest layer in the Contiki-NG NETSTACK. The radio layer is handled by radio module from the Contiki-NG mote. Most radio layers work on the IEEE 802.15.4 protocol mechanism.
Network Simulation Using COOJA

In Chapter 1, we learned how to work with COOJA to build a Contiki-NG simulation. In this section, we will continue to apply COOJA to create a network simulation. For this simple demo, we will use the same scenario as in Chapter 4. We will perform broadcasting among Contiki-NG motes.

To implement the demo, we will perform the following tasks:

- Create a simulation project.
- Add a UDP server mote.
- Add UDP client motes.
- Run the simulation.

Each task will be performed in the following sections.

Creating Simulation Project

The first step is to create a simulation project using COOJA. From your platform Terminal, you can run the COOJA tool. For instance, I run it from Ubuntu Linux. You can type these commands:

```
$ cd contiki-ng/tools/cooja
$ sudo ant run
```

After it has executed, you should get the COOJA application that is shown in Figure 6-5.
To start a new simulation, you can click File ➤ New simulation, shown in Figure 6-6.

Figure 6-5. COOJA application

Figure 6-6. Create a new simulation in COOJA
Then, you should get the dialog that is shown in Figure 6-7. Fill in the simulation name with default settings on the Advanced Settings panel. For instance, I fill in the simulation name with My simulation.

![Create new simulation dialog](image)

**Figure 6-7. Fill in simulation name and other settings**

If done, you can click the Create button to create the simulation. After that, COOJA will create a new simulation for you. Figure 6-8 shows a simulation dashboard from COOJA. Now you are ready to configure the simulation.

In the next section, we will add a UDP server mote to our simulation.
Adding UDP Server Mote

In this section, we will add a mote as the UDP server. This mote will listen to incoming messages. Once a message is received from a client, the UDP server will reply by sending that message. For the demo, we will use a mote with the Sky platform type.

To add a new mote on COOJA, go to Add Motes ➤ Create a mote type ➤ Sky mote. You can see this menu in Figure 6-9.
You will get the dialog shown in Figure 6-10. You can fill in the description of the mote. Then, set the Contiki firmware from the UDP server, udp-server.c. You can find udp-server.c in the <contiki-ng>/examples/rpl-udp/ folder. Click the Browse button and navigate to the udp-server.c file.

You will get the dialog shown in Figure 6-10. You can fill in the description of the mote. Then, set the Contiki firmware from the UDP server, udp-server.c. You can find udp-server.c in the <contiki-ng>/examples/rpl-udp/ folder. Click the Browse button and navigate to the udp-server.c file.
After selecting the udp-server.c file, you can compile this file to ensure there are no errors in the program. Click the Compile button to compile the program. If it succeeds, you should see the successful compilation on the Compilation Output tab, shown in Figure 6-11.

![Create Mote Type: Compile Contiki for sky]

**Figure 6-11. Compile the UDP server mote**

If you have compiled the program, you can click the Create button. Then, you should get the dialog that is shown in Figure 6-12.

![Add motes (Sky Mote Type #sky1)]

**Figure 6-12. Configure the mote number and its position**
In this scenario, we only add one mote for the UDP server. You can fill in 1 for the “Number of new motes” field and select Random positioning for the “Positioning” dropdown.

If done, you can click the Add Motes button. You should see this mote on the Network panel in the simulation dashboard. You can see it in Figure 6-13.

![Image of COOJA network panel showing a single mote](image)

**Figure 6-13.** UDP server is deployed on COOJA

Next, we will add some motes for the UDP client. We will perform this task in the next section.

### Adding UDP Client Motes

After we have created one UDP server mote, we can create some UDP client motes. For this demo, we will add five motes. To add a new mote, you perform this task as in the previous section.
For a UDP client mote, you can put udp-client.c as the Contiki firmware. You can find it in the `<contiki-ng>/examples/rpl-udp/` folder. You can see the UDP client program in Figure 6-14.

![Figure 6-14. Adding UDP client mote](image)

Then, you should compile the UDP client firmware to ensure there are no errors in the program. You can click the Compile button. You can see my successful compilation from the UDP client in Figure 6-15.

![Figure 6-15. Compiling UDP client mote](image)
You can add motes to the Network panel by clicking the Create button in the dialog shown in Figure 6-15. After clicking, you should get the dialog that is shown in Figure 6-16. For this demo, fill in 5 in the “Number of new motes” field. All fields are default values.

![Add motes (Sky Mote Type #sky2)](image)

**Figure 6-16.** Adding five UDP client motes

After filling in the number of motes, you can click the Add Motes button. Then, you should see five UDP client motes in the Network panel in the simulation dashboard. Now you have six motes in the simulation, shown in Figure 6-17.
Running a Simulation

Once we have added the UDP server and client motes into the COOJA simulation, we can run the simulation. To do so, click the Start button on the Simulation Control dialog, as shown in Figure 6-18. If you don’t see this dialog, you can open it by going to Simulation ➤ Control panel.
After clicking the Start button, you should see the packet network and graphic simulation on COOJA. You can see it in Figure 6-19.

Figure 6-19. A simulation is running on COOJA
If the distance between motes is far, some motes probably won’t connect or receive UDP messages. You can change the mote location so each mote can be connected.

Now you can run the simulation again. You should see a graphic simulation on COOJA. A sample of the simulation can be seen in Figure 6-20.

**Figure 6-20. Changing mote location**

**IPv6 Networking**

IPv6 (Internet Protocol Version 6) is the Internet’s next-generation protocol, designed to replace the current protocol, IP Version 4 (IPv4). Most network systems still use IPv4 to communicate with other systems. Sample IP addresses from IPv4 and IPv6 are shown here:

// IPv4
192.168.0.2
// IPv6
2021:db8:ffff:1:201:02ff:fe03:0415
To test whether your network uses IPv4, IPv6, or both, you use a browser and navigate to http://test-ipv6.com. Based on IPv6 and IPv4 statistics from http://ipv6-test.com/stats/, IPv6 penetration in the market still does not dominate. Figure 6-21 shows a comparison of protocol support for IPv6 and IPv4. For instance, on July 2017, IPv6-only dominates at about 54.7 percent.

**Figure 6-21. Comparing IPv4 and IPv6 protocol support worldwide**

For more learning material about IPv6, I recommend you read some textbooks or technical articles related to IPv6 technology. This book does not cover IPv6 technology in any depth.

Contiki-NG can work with an IPv6 network by default. Contiki-NG projects provide uIP as the network stack implementation for IP, UDP, and TCP protocols included in the basic ICMP protocol. The uIP library can run on constrained devices, such as Tmote Sky/TelosB, TI cc26xx/cc13xx, Firefly, RE-mote, and Orion.

For our IPv6 demo on Contiki-NG, we can perform testing using a Contiki-NG shell (NG shell). To enable an NG shell for your program, you should add a shell module to your Makefile file:

```makefile
MODULES += os/services/shell
```
Then, you can compile and upload to your mote. Since the shell module needs more space, you should use a mote platform that can handle NG shell spaces. In this demo, I use a TI CC2650 LaunchPad board.

For testing, we need two motes at least. Upload your project; for instance, hello-world with NG shell enabled. After uploading Contiki-NG firmware onto the boards, we can remote into the NG shell from a serial terminal.

For instance, I remote into the TI CC2650 LaunchPad via the /dev/ttyACM0 port:

```
$ make TARGET=srf06-cc26xx BOARD=launchpad/cc2650 login PORT=/dev/ttyACM0
```

Open a new Terminal window. Then, perform a serial remote on the second mote. For instance, my second mote runs on the /dev/ttyACM2 port:

```
$ make TARGET=srf06-cc26xx BOARD=launchpad/cc2650 login PORT=/dev/ttyACM2
```

After connecting to the NG shell, we can perform some tests related to IPv6 operations. You can check the current IPv6 address on each mote.

You can type this command in the NG shell:

```>
ip-addr
```

You should see an IPv6 address for each mote. Next, you can perform a ping to one of the motes. For instance, I perform a ping on a mote with the IPv6 address `fe80::212:4b00:d77:6f82`:

```>
ping fe80::212:4b00:d77:6f82
```

You should get a response from ping operations. Last, our mote can discover its neighbor. By default, Contiki-NG enables `UIP_ND6_AUTOFILL_NBR_CACHE` to be discoverable. You can perform this command to discover a mote’s neighbor:

```>
ip-nbr
```
All these operations can be seen in Figure 6-22.

**Figure 6-22.** IPv6 testing via Contiki-NG shell
Routing on Contiki-NG

Routing is the process of moving packets across a network from source to destination. This process is usually performed by dedicated network devices, such as routers and computers. Routing is a key feature of the Internet. Routing topics as a research area is still popular. Some researchers propose algorithms to address routing problems. In this section, we will review routing and how to implement it in the Contiki-NG platform.

Introducing Basic Routing

We already know that routing is applied to address how the packet is delivered from one point to another. Consider the paths depicted in Figure 6-23. For instance, we want to send a packet from A to D. Which path do you want to take?

Figure 6-23. Data network flow on some motes

Figure 6-23 shows a number of path options to send a packet from A to D. We can take one of three path options: A-B-D, A-D, and A-C-D. Which one is the best path? This is a challenge.
When selecting a path, you must consider what your goal is and what your success criteria are. We can define a cost for each path. Then, we can select the best path with the lowest cost. Battery usage can be one of the cost parameters when you want to perform a certain routing algorithm.

That is one routing issue. Unfortunately, this book does not focus on routing algorithms. I recommend you read about routing topics in textbooks or technical articles. We will use the current routing algorithms that are applied in Contiki-NG.

**Single-Hop and Multi-Hop Networking**

When a packet is transferred from the source to the final destination, it probably goes through a number of network devices. In networking, we will find a term called a *hop*. *Hop* is a term used to describe the different network devices a packet has to go through to reach its final destination point.

We can categorize networking models by the number of hops, such as single-hop and multi-hop networks. To better understand these network models, see the network topology depicted in Figure 6-24. A single-hop network would be pointed to the A-D path. The packet is sent from A to D through a single router.

Multi-hop networks send a packet through two or more networks to reach its final destination address. From Figure 6-24, we can see that the E-F path is considered a multi-hop network.
The Contiki-NG platform supports both single-hop and multi-hop networks. Depending on your network design, your Contiki-NG program should be aware of single-hop and multi-hop networks.

**Routing on Contiki-NG**

Currently, Contiki-NG supports two RPL routing methods: RPL classic and RPL lite. RPL classic is the original Contiki’s RPL implementation, called ContikiRPL. If you want to read further about ContikiRPL, I recommend you read this paper: [http://www.diva-portal.org/smash/get/diva2:1042739/FULLTEXT01.pdf](http://www.diva-portal.org/smash/get/diva2:1042739/FULLTEXT01.pdf). Otherwise, Contiki-NG applies RPL lite for default RPL implementation.
Since Contiki-NG applies RPL lite by default, we test RPL. For instance, we have two motes for testing. Enable NG shell on your program. You can use the hello-world program with NG shell enabled. Flash this program to all motes.

Now you can access the Contiki-NG motes via NG shell. First, test an RPL root on the first mote. You can type this command:

```
> rpl-set-root 1
```

Then, you can check the RPL status using this command:

```
> rpl-status
```

You also can check the RPL route table that is applied on this mote. Type this command:

```
> routes
```

You can see my program output that has performed those tasks in Figure 6-25.
Figure 6-25. RPL operations on the first mote
Next, you also can check the RPL status and its routes on another mote. You can type these commands in the NG shell:

> rpl-status
> routes

A sample of the program output is shown in Figure 6-26.

![Figure 6-26. RPL operations on the second mote](image)

**IPv6 Multicast**

We can categorize data communication based on how the data is transferred. There are three models in data communication: unicast, broadcast, and multicast.

Unicast describes communication where a piece of information is sent from one point to another point. In the Unicast model, there is one sender and one receiver. Network protocols–based TCP transport such as http, smtp, ftp, and telnet support the unicast transfer mode.
Broadcast is a communication model where a piece of information is sent from one point to all other points. In this model, there is still one sender, but the information is sent to all connected receivers. There may be no receivers. ARP (Address Resolution Protocol) uses broadcast to send an address resolution query to all computers.

Multicast describes communication where a piece of information is sent from one or more points to a set of other points. In this model, there may be one or more senders. The information is distributed to a number of receivers.

The main difference between multicast and broadcast is to provide opt-in option for receivers. The receiver that wants to receive data should register to gain access to the sender. This registration will inform the network that you are interested and have opted in to receiving data. Otherwise, the receiver never receives the data. To compare how unicast, broadcast, and multicast send packets, see Figure 6-27.

![Figure 6-27. Unicast, broadcast, and multicast](image)

To work with IPv6 multicast in a Contiki-NG project, you should add the multicast module to your project. You can add this script in the Makefile file:

```make
MODULES += os/net/ipv6/multicast
```
We also activate RPL classic routing on the Contiki-NG project. You can add this script into the `Makefile` file:

```
MAKE_ROUTING = MAKE_ROUTING_RPL_CLASSIC
```

For testing, we can use a program sample from Contiki-NG. We can use the multicast project from the `<contiki-ng>/examples/multicast/` folder. For this demo, we need at least three motes. One mote is deployed for `sink.c` and another mote is deployed for `root.c`. The rest should be flashed for `intermediate.c`.

After deploying all programs to the motes, you can try to remote in to each mote. In general, the `sink.c` program will listen for incoming messages from `root.c`. Before listening for the message, the `sink.c` program should join the existing multicast group by calling the `join_mcast_group()` function. This program listens on port 3001:

```
if(join_mcast_group() == NULL) {
    PRINTF("Failed to join multicast group\n");
    PROCESS_EXIT();
}
```

The `root.c` program will send a UDP message every second. This is done by calling the `uip_udp_packet_send()` function. The `intermediate.c` program does not join the existing multicast group. Since the `intermediate.c` program has been activated for RPL classic routing, this program can forward any multicast message.

The demo of the multicast project can be seen in Figure 6-28. You can see the program output from the `sink.c` and `root.c` programs.
If you don’t have three motes or more, you still can simulate this project using COOJA. You can open the `<contiki-ng>/examples/multicast/multicast.csc` file in the COOJA application. There are eight
motes in this simulation. After loading the project into COOJA, you can run
the multicast simulation. You can see the simulation output in Figure 6-29.
You can see all messages from the motes in the Radio Messages window.

Figure 6-29. Multicast demo on COOJA tool
Contiki-NG NullNet

If you want to investigate a packet among Contiki-NG network stack layers, you can use NullNet. To work with NullNet, add this script to the Makefile file:

```
MAKE_NET = MAKE_NET_NULLNET
```

If you want to send data, you should the data and its size in the `nullnet_buf` and `nullnet_len` variables, which are pre-defined from `net/nullnet/nullnet.h`. Then, send the data by calling `NETSTACK_NETWORK.output(NULL)`. To receive packets from NullNet, you call the `nullnet_set_input_callback(callback_func)` function with the passing callback function.

For this demo, you can run program samples from the `<contiki-ng>/examples/nullnet` folder. There are two demos: broadcast and unicast. You can run these programs on motes directly or in the COOJA application. You can see my program output in Figure 6-30 on a mote from the demo samples on Contiki-NG motes.
You can also test this program demo using COOJA. Just select the `nullnet-broadcast.csc` and `nullnet-unicast.csc` files from the COOJA application. After they are loaded, you can run these demos and see radio messages from COOJA application and review pack flow.

Figure 6-30. NullNet demo on Sky mote
6LoWPAN Network

6LoWPAN is an acronym for IPv6 over Low-Power Wireless Personal Area Networks. 6LoWPAN is an open standard defined in RFC 6282. This standard enables WSN motes to communicate with other systems via an Internet network.

In this section, we will learn the basics of 6LoWPAN and how to implement it on the Contiki-NG platform.

A Brief Introduction

6LoWPAN is a network standard used to enable WSN motes to communicate with external networks, such as Internet networks. 6LoWPAN uses IPv6 as the identity for all motes. A 6LoWPAN network is shown in Figure 6-31.

---

**Figure 6-31.** 6LoWPAN network
WSN networks apply IPv6 for all communication. Each mote can communicate with the other motes. If one mote wants to communicate with an external system, such as a server, computer, or any legacy application in a different network, the mote just sends a message as usual. The 6LoWPAN router will take responsibility for communication between internal and external networks.

A 6LoWPAN router will record all addresses from motes and other network devices that are connected to the 6LoWPAN router. If an external system such as a server sends a message to one of the WSN motes in the WSN network, the 6LoWPAN router will forward the message to the mote. Otherwise, 6LoWPAN will inform the requester of failure.

In a network-stack view, we can compare a 6LoWPAN network to a WiFi network from the OSI layer side. You can see this comparison in Figure 6-32.

As you can see in Figure 6-32, 6LoWPAN works on the data-link layer in the OSI model. This standard uses IEEE 802.15.4 for physical layer implementation. In the upper layer of the 6LoWPAN, this standard applies IPv6 on RPL.

![Figure 6-32. Comparing WiFi and 6LoWPAN to OSI model](image)

In the next section, we will try to implement 6LoWPAN on Contiki-NG.
Implementing a 6LoWPAN Network on Contiki-NG

Contiki-NG implements the 6LoWPAN network stack via an RPL border router that acts as a 6LoWPAN router. You can see a network diagram in Figure 6-33 for implementing 6LoWPAN on the Contiki-NG platform. You can put the RPL border router on a computer or a Raspberry Pi or any network device.

How do you implement an RPL border router in Contiki-NG?

Basically, you need a Contiki-NG mote as the RPL border router mote. This mote will work as a bridge to serve all requests from motes in the WSN network or network devices from an external network. To create an RPL border router mote, you should add the rpl-border-router library into your project. You can add this script to your Makefile file:

```makefile
MODULES += os/services/rpl-border-router
```

Then, run a program, called tunslip6, from Contiki-NG. You can find it in the `<contiki-ng>/tools/` folder. This tool will communicate with a mote that has already been deployed as an RPL border router via serial port. You can type this command:

```
$ sudo ./tunslip6 <prefic> -s <serial_port>
```

`<prefic>` is a prefix from the IPv6 address that will assign to all motes on the RPL border router application. `<serial_port>` is a serial port from the mote running the RPL border router program.
For this demo, we can run a program sample from the `<contiki-ng>/examples/rpl-border-router/` folder. This project includes an RPL border router and web server modules. We need at least two motes to simulate the 6LoWPAN router and its communication.

Compile this project and flash it to one of the Contiki-NG motes. After it has been deployed to the mote, you can work remotely on the non-border router mote to see the assigned IPv6 address.

Last, you should run the tunslip6 program on the computer/Raspberry Pi to which the RPL border router mote was attached. You can run this command on the `rpl-border-router` project:

```
$ make TARGET=srf06-cc26xx BOARD=launchpad/cc2650 connect-router
```
After being executed, that command will run tunslip6 with a default port. You should probably change the values for the target mote platform and its serial port.

By default, the RPL border router has an IPv6 address of \texttt{fd00::1/64}. You can change it in your \texttt{Makefile}. For instance, if you define the prefix of your IPv6 address as \texttt{fd00::1/64}, all motes within the WSN network will be assigned as \texttt{fd00::xxxx}. In this demo, my mote is assigned as \texttt{fd00::212::4b00::d77::6fd2}.

Now you can ping the mote using the assigned IPv6 address from your computer. For instance, type this command:

\begin{verbatim}
$ ping6 fd00::212:4b00:d77:6fd2
\end{verbatim}

This program will get a response from the target mote because this mote can communicate with the computer through the RPL border router. You can see the program output in Figure 6-34. I ping my mote from Linux Ubuntu. It shows my computer can contact the Contiki-NG mote.

If you do not see a list of IPv6 addresses from your motes, you can restart your RPL border router application, tunslip6. Then, see the list of IPv6 addresses from the motes.
Figure 6-34. Performing ping on one of the Contiki-NG motes from a computer
If you use a program sample from `<contiki-ng>/examples/rpl-border-router/`, you should get the web server within the program. You can test by opening a browser and navigating to `http://[ip6_address_from_mote]`. You can see my browser output in Figure 6-35.

![Image](image1.png)

**Figure 6-35.** Accessing web server on Contiki-NG mote from computer browser

If you have another mote, you can upload the Contiki-NG firmware to that mote. Then, the RPL border router will detect it. If you open a browser and navigate to the IPv6 address from the RPL border router mote, you should show its neighbor. You should see the IPv6 address from that mote. For instance, you can see it in Figure 6-36.

![Image](image2.png)

**Figure 6-36.** Displaying mote neighbor on RPL border router
You can also test it by performing a ping. For instance, the target mote is fd00::212:4b00:6083. We can perform a ping as follows:

$ ping6 fd00::212:4b00:6083

You should get a response from that mote. You can see it in Figure 6-37.

Sometimes you don’t see the IPv6 address from a new mote in your browser (Figure 6-36), but you know the local IPv6 address of the new mote. You can verify it by remoting into the mote with the local IPv6 address. For instance, the IPv6 address is fe80::212:4b00:6083.

Now, change the prefix to fd00::xxx. The new IPv6 address shows fd00::212:4b00:6083. Try to ping it.

If you still have problems in which motes are not detected by the RPL border router, you probably need to restart the RPL border router mote.

Figure 6-37. Performing ping on other mote in WSN network from computer
6LoWPAN Implementation using COOJA

In the previous section, we learned how to implement 6LoWPAN on a physical device from a Contiki-NG mote. In this section, we want to implement 6LoWPAN on a simulation platform through the COOJA tool.

For this demo, we will use the program sample from the `<contiki-ng>/examples/rpl-border-router/` folder. First, you can run the COOJA application. Create a new simulation project.

Next, you should add a new mote with the Sky platform. You can set `<contiki-ng>/examples/rpl-border-router/border-router.c` for the Contiki firmware, shown in Figure 6-38.

![Add RPL border router into COOJA](image)

You can compile this Contiki firmware. If there is no error, you can click the Create button. In this demo, you will create one mote.

The next step is to add the serial socket tool to the mote. You can do this by right-clicking on the mote. Then, you will see the context menu shown in Figure 6-39. Select Mote tools for Sky 1 ➤ Serial Socket (SERVER).
You should now see the dialog shown in Figure 6-40. You can run it by clicking the Start button. We use the default listen port, 60001.

**Figure 6-39. Add serial socket tool on RPL border router mote**

You should now see the dialog shown in Figure 6-40. You can run it by clicking the Start button. We use the default listen port, 60001.

**Figure 6-40. A serial socket dialog to set a listen port**
Next, you should run tunslip6 from the `<contiki-ng>/tools/` folder. You can open Terminal and navigate to `<contiki-ng>/tools/`. Then, you can run this command:

```bash
$ sudo ./tunslip6 -a 127.0.0.1 aaaa::1/64
```

Tunslip6 will run and connect to the RPL border router from a mote within the COOJA application. You can see the program output from tunslip6 in Figure 6-41.

![Figure 6-41. Running tunslip6 program on local computer](image)

Once tunslip6 is running, you can run the simulation on COOJA. This makes the mote run. You can see the IPv6 translator of the mote from tunslip6 in Figure 6-42.
You should see the IPv6 address of the COOJA mote in Terminal. For instance, my mote IPv6 address is aaaa::212:7401:1:101. Now, you can open a new Terminal. Then, try to perform ping6.

You can type this command:

```
$ ping6 aaaa::212:7401:1:101
```

You should get responses from your COOJA mote. You can see my ping6 response in Figure 6-43.
To get more practice, you can run a web server program on a new mote in COOJA. Then, you can access that web server from a browser on a local computer.

**Figure 6-43. Pinging a mote from computer**

To get more practice, you can run a web server program on a new mote in COOJA. Then, you can access that web server from a browser on a local computer.
Build Your Own RESTful Server for Contiki-NG

You have learned how to build a 6LoWPAN with RPL border router. In this section, we will build a simple project based on 6LoWPAN. We will develop WebSense, which publishes sensor values to the web server. Since a Contiki-NG mote has limited resources, it cannot serve many requests. To address this issue, we can implement a middleware server—for instance, a RESTful server.

Each mote will send a result of sensing to the RESTful server. Then, the RESTful server will take over to distribute the data to all requesters. A RESTful server can run on top of a proven web server, such as Apache, nginx, and IIS.

WebSense is shown in Figure 6-44. Each mote can communicate with the RESTful server through the 6LoWPAN router. A client that wants to consume sensor data should open a connection to the RESTful server through WebSocket.

Figure 6-44. Logic design for the WebSense project
A client system will be implemented using HTML5 with WebSocket API. Sensor data will be visualized in the HTML5 application. Also, the WebSense RESTful server will apply Node.js.

We will implement this WebSense project in the next section.

**Preparation**

First, we should have two mote devices. One mote will be used for the 6LoWPAN router. The rest will be applied for WebSense node implementation.

Since our RESTful server uses Node.js, your computer should install Node.js runtime. You should install all required libraries to run Node.js. Type these commands:

```bash
$ sudo apt-get update
$ sudo apt-get install build-essential
```

We will use Node.js LTS version. For instance, I use Node.js LTS 6.x. You can install it by typing these commands:

```bash
$ curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
$ sudo apt-get install nodejs
```

The next step is to develop a program for the project. We will do so in the next section.

**Implementing the Demo**

We will implement the WebSense project as shown in Figure 6-45. It is a physical design from our demo. We deploy the RPL border router into one of the motes. This mote will be attached to a computer that will deploy Node.js too. This project will use real mote devices. You can also use COOJA for testing.
Computers and server machines should be connected to a network in order to simulate sensor data visualization.

**Figure 6-45. Implementing WebSense project**

To implement this demo, we will perform the following tasks:

- Implement 6LoWPAN router.
- Develop a program for WebSense node.
- Develop a program for RESTful server.
- Test the project.

Each task will be presented in the following sections.
Implementing 6LoWPAN Router

To implement the 6LoWPAN router, we deploy the RPL border router module into the mote. In a previous section, you learned about 6LoWPAN router implementation. We will use it again in this project.

Select one of the motes to become a 6LoWPAN router. Attach it into a computer and run connect-router or run tunslip6 programs manually with specific port and address settings.

Writing a Program for WebSense Node

In this section, we will develop a program for the WebSense node. The goal of this program is to serve all requests for sensor data. Technically, the program will run a mini web server and serve HTTP requests. Once the RESTful server requests sensor data, this node will send it. To simplify this demo, the program will generate a random value for sensor data.

You can see the project structure in Figure 6-46. You can create a folder, called websense. We put some files in it, such as Makefile, project-conf.h, and websense.c. For web server implementation, we use the httpd-simple module from Contiki-NG.

![Figure 6-46. Project structure for WebSense node](image)
The application will serve requests for sensor data. This task will be handled in websense.c in generate_routes() function. You can write this code as follows:

```c
static
PT_THREAD(generate_routes(struct httpd_state *s))
{
    char buff[15];
    PSOCK_BEGIN(&s->sout);
    int temperature = 15 + rand() % 25;
    sprintf(buff,"{"temp":%u}", temperature);
    printf("send json to requester\n");
    SEND_STRING(&s->sout, buff);
    PSOCK_END(&s->sout);
}
```

We also need to modify httpd-simple.c in order to cover JSON requests. We declare http_content_type_json for JSON content. Then, we pass it into the HTTP header as follows:

```c
const char http_content_type_json[] = "Content-type: application/json\r\n";
static
PT_THREAD(send_headers(struct httpd_state *s, const char *statushdr))
{
    /* char *ptr; */
    PSOCK_BEGIN(&s->sout);
    SEND_STRING(&s->sout, statushdr);
```
SEND_STRING(&s->sout, http_content_type_json);
PSOCK_END(&s->sout);
}

Writing a Program for RESTful Server

A RESTful server will run a web server on top of Node.js. We build the project structure that is shown in Figure 6-47. To visualize sensor data, we apply jQuery (https://jquery.com) and Flot (http://www.flotcharts.org) libraries for JavaScript. Download those files. Then, you can put those files into the <project>/public/js folder.

![Figure 6-47. Project structure for RESTful server](image)

The application runs with the Node.js runtime. Make sure you have already installed it. Next, we install required the libraries for the RESTful server. We will use Express (http://expressjs.com) for the web framework and Socket.io (https://socket.io) for WebSocket implementation for Node.js.

First, create a package.json file inside the project folder. You can type these scripts:

```json
{
  "name": "sensor",
  "version": "1.0.0",
  "description": "visualizing real-time sensor",
```
If done, save this file. Now, you can install all required libraries. You can type this command in the project folder. Make sure your computer is connected to the Internet:

$ npm install

Now you can write `index.js` for the application. This program will open a port 3000 to listen for HTTP requests. The program also requests sensor data from the WebSense node every three seconds. You can type these scripts for `index.js`:

```javascript
var express = require('express');
var request = require('request');
var app = express();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.use(express.static('public'));

app.get('/', function(req, res){
    res.sendFile(__dirname + '/index.html');
});
```
io.on('connection', function(socket) {
    var dataPusher = setInterval(function () {
        request.get('http://[fd00::212:4b00:797:6083]/', function (err, res, body) {
            if (err) {
                console.log(err);
                return;
            }
            var obj = JSON.parse(body);
            socket.broadcast.emit('data', obj.temp);
        });
    }, 3000);

    socket.on('disconnect', function() {
        console.log('closing');
    });
});

http.listen(3000, function() {
    console.log('listening on *:3000');
});

You should change the IPv6 address of the WebSense node.

Next, we create index.html in the <project>/public folder. This program will communicate with the RESTful server though JSON communication. If the program receives sensor data, it will be stored into an array.
Then, the program will create a graphic for visualizing sensor data. You can type these scripts for index.html:

```html
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Visualizing Real-Time Sensor Data</title>
    <script language="javascript" type="text/javascript" src="/js/jquery-3.2.1.js"></script>
    <script language="javascript" type="text/javascript" src="/js/jquery.flot.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script language="javascript" type="text/javascript">
        var socket = io.connect();
        var items = [];
        var counter = 0;
        socket.on('data', function (data) {
            items.push([counter, data]);
            counter = counter + 1;
            if (items.length > 20) {
                items.shift();
                $.plot($("#placeholder"), [items]);
            }
        });
    </script>
</head>
<body>
<h1>Real-Time Sensor Data Visualization</h1>
<br/>
<div id="placeholder" style="width:600px;height:300px;"/>
</body>
</html>
```
Testing the Demo

Ensure the programs for the motes are already deployed. Now you can run the RESTful server by typing this command:

$ node index.js

Firstly, we open a browser and navigate to the IPv6 address of the 6LoWPAN router. You should see the IPv6 addresses from the motes as shown in Figure 6-48.

![Figure 6-48. Accessing web server from RPL border router mote](image)

After the RESTful server is running, you can test it by opening a browser. Navigate to the RESTful server’s IP address with port 3000. For instance, you can open a browser on your local server and navigate to http://localhost:3000. You should see sensor data visualization. You can see this in Figure 6-49.
What’s next?
You can create more sensor sources in different sensor types from several motes. Then, you can modify sensor data visualization to cover multiple sensor types.

Summary
We have explored Contiki-NG networking. We have learned about routing models in Contiki-NG. Furthermore, we have worked with IPv6 multicast. We also implemented 6LoWPAN on physical motes and COOJA simulations. Last, we build sensor data visualization in real-time from a physical mote.

In the next chapter, we will learn to work with storage management in Contiki-NG.
CHAPTER 7

Storage

Wireless Sensor Networks (WSN) are designed to perform sensing and then send the sensing data to a gateway or a certain server. You probably will want to perform some computations before sending the data. These computations involve some parameters that should be kept by WSN devices. Keeping data in a WSN device requires storage. In this chapter, we will learn how to work with storage in Contiki-NG.

The following is a list of topics that will be covered in this chapter:

- Storage models in Contiki-NG
- Working with local storage
- Working with Coffee file system
- Contiki-NG and MySQL

Storage Models in Contiki-NG

Contiki-NG is designed for small devices with optimized computation. You will probably need to store data from your project, such as sensor data or program parameters. These data can be persistent data. The data also is used to analyze and investigate something from your project.
Storage models in Contiki-NG are shown in Figure 7-1. We define a storage model of Contiki-NG based on where the data will be stored. We can put our data in internal or external storage. Internal storage is a part of the internal MCU storage, such as ROM and RAM. Otherwise, we can extend our Contiki-NG storage by adding external storage.

From Figure 7-1, we can see that external storage comes in two models: local storage and external storage. We can attach any external storage, such as SD card and micro SD card, to a Contiki-NG mote. Then, we can save our data into these storage devices through SPI or I2C protocols. Remote storage can be represented as network storage. This means Contiki-NG will save data to a remote data server, such as MySQL, PostgreSQL, SQL Server, or Oracle, via a network protocol.

Figure 7-1. Storage models in Contiki-NG
Each storage model in Contiki-NG has advantages and disadvantages. You can choose which model based on your needs. You can see the storage model comparison table in Table 7-1.

**Table 7-1. Comparing Storage Models in Contiki-NG**

<table>
<thead>
<tr>
<th>Features</th>
<th>ROM</th>
<th>RAM</th>
<th>Local Storage</th>
<th>Remote Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can read?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can write?</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Access speed</td>
<td>Fast</td>
<td>Fast</td>
<td>Moderate</td>
<td>Moderate/Slow</td>
</tr>
<tr>
<td>Access protocols</td>
<td>Check MCU datasheet from the mote</td>
<td>Check MCU datasheet from the mote</td>
<td>SPI, I2C</td>
<td>HTTP, RESTful, TCP</td>
</tr>
</tbody>
</table>

ROM and RAM provide good speed for reading and writing data, but not all MCU devices have large storage sizes for the ROM and RAM. A local storage approach is a good choice if you plan to perform logging for system evaluation. The remote storage option is best if you need to perform data consolidation. You can do data analysis and predictive analytics from your collected data.

**Working with Local Storage**

In this section, we will learn how to work with local storage in Contiki-NG. We will review how a program uses resources in RAM and ROM via Contiki-NG. For instance, we want to analyze the hello-world program from Contiki-NG samples. I will test it with targeting on LaunchPad CC2650. First, we compile the hello-world program. You can type these commands:

```
$ make TARGET=srf06-cc26xx BOARD=launchpad/cc2650
$ make TARGET=srf06-cc26xx BOARD=launchpad/cc2650 savetarget
```
Then, we analyze resource allocation for the hello-world program using this command:

```
$ size hello-world.srf06-cc26xx
```

You can see the program output in Figure 7-2. Text shows the size of the code section in bytes that will be stored in ROM. data and bss show sections that contain variables and will be stored in RAM. From Figure 7-2, we can see that the hello-world program fits in ROM with its 50644-byte size.

Figure 7-2. Analyzing binary program on Contiki-NG with LaunchPad CC2650 platform

If you use the Sky platform, you can verify the hello-world program size with a similar approach. You can type these commands:

```
$ make TARGET=sky
$ make TARGET=sky savetarget
$ size hello-world.sky
```
After executing those commands, you can see the program output in Figure 7-3. This program reserves 41126 bytes in ROM.

![Image of terminal output](image)

**Figure 7-3.** Analyzing binary program on Contiki-NG with Sky platform

If you want to get a detail of the program address usage from RAM, call this command:

```
$ make hello-world.ramprof
```

After it has been executed, you should see a list of program addresses in RAM. A sample of program output for the LaunchPad CC2650 board can be seen in Figure 7-4. For instance, the events program is fitted on 00000384.
We also can analyze program addresses in ROM from our program. You can type this command from the hello-world program:

```
$ make hello-world.flashprof
```

You should see a list of program addresses. A program output can be seen in Figure 7-5 for the LaunchPad CC2650 board.
Coffee: File System in Contiki-NG

Coffee is one type of file system implementation in Contiki-NG. Technically, Coffee is designed for making a virtual file system on a RAM stack of Contiki-NG. We can find the Coffee library in the
<contiki-ng-root>/os/storage/cfs/ folder. There are three files: cfs.h, cfs-coffee.h, and cfs-coffee.c. This module implements basic operations for reading and writing files, like the POSIX file API.

Since Coffee uses RAM as storage, it’s a temporary storage option. You can use this storage to store your temporary data, such as computation parameters and a counter number. You should be aware that Coffee data is deleted when the Contiki-NG mote is restarting or stopping.

We access a file through the Coffee File System (CFS) library. This library API interface can be found in the <contiki-ng-root>/os/storage/cfs/cfs.h file. You can see a list of functions from the cfs.h file in Table 7-2.

**Table 7-2. CFS Functions from cfs.h Header File**

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int cfs_open(const char *name, int flags)</td>
<td>Open a file</td>
</tr>
<tr>
<td>void cfs_close(int fd)</td>
<td>Close a file</td>
</tr>
<tr>
<td>int cfs_read(int fd, void *buf,</td>
<td>Read data from a file</td>
</tr>
<tr>
<td>unsigned int len)</td>
<td></td>
</tr>
<tr>
<td>int cfs_write(int fd, const void *buf,</td>
<td>Write data to a file</td>
</tr>
<tr>
<td>unsigned int len)</td>
<td></td>
</tr>
<tr>
<td>cfs_offset_t cfs_seek(int fd,</td>
<td>Move to a specific position in</td>
</tr>
<tr>
<td>cfs_offset_t offset, int whence)</td>
<td>a file</td>
</tr>
<tr>
<td>int cfs_remove(const char *name)</td>
<td>Remove a file</td>
</tr>
<tr>
<td>int cfs_opendir(struct cfs_dir *dirp,</td>
<td>Open a directory</td>
</tr>
<tr>
<td>const char *name)</td>
<td></td>
</tr>
<tr>
<td>int cfs_readdir(struct cfs_dir *dirp,</td>
<td>Read a directory entry</td>
</tr>
<tr>
<td>struct cfs_dirent *dirent)</td>
<td></td>
</tr>
<tr>
<td>void cfs_closedir(struct cfs_dir *dirp)</td>
<td>Close a directory</td>
</tr>
</tbody>
</table>
To work with CFS, your project should include the cfs library. You can add it to the Makefile file. Add this script:

```makefile
MODULES += os/storage/cfs
```

For this demo, we will create a simple program to show how to work with CFS on Contiki-NG. We will create a file and write data to that file. Next, we will open a file and read data from the file. In this demo, I will use the Sky platform.

First, create a folder, called file-cfs-demo. You can add two files, Makefile and file-cfs-demo.c. Configure the project in Makefile. You can write these scripts for the Makefile file:

```makefile
CONTIKI_PROJECT = file-cfs-demo
MODULES += os/storage/cfs
CONTIKI = ../..
all: $(CONTIKI_PROJECT)
include $(CONTIKI)/Makefile.include
```

You probably need to change the CONTIKI value to your Contiki-NG project directory. Now, you can write the program for the file-cfs-demo.c file. Write the following code skeleton from our demo:

```c
#include "contiki.h"
#include "cfs/cfs.h"
#include "cfs/cfs-coffee.h"
#include "lib/crc16.h"
#include "lib/random.h"

#include <stdio.h>
#include <string.h>
```
This program will run the `coffee_file_demo()` function on the main program. We will implement the CFS demo on that function.

First, we initialize program variables, including file descriptors for opening and reading file handlers:

```c
int wfd, rfd, afd;
unsigned char buf[32];
int r;
```

We set all buffer data with certain values:

```c
for(r = 0; r < sizeof(buf); r++) {
    buf[r] = r + 4;
}
```
We open a file, mycfs, by calling the cfs_open() function:

```c
printf("opening file for writing\n");
wfd = cfs_open("mycfs", CFS_WRITE);
if(wfd < 0) {
    printf("Error creating file\n");
    return;
}
```

Next, we write data to a file by calling the cfs_write() function. We also print all data to Terminal so you can see the data on the Contiki-NG Terminal:

```c
printf("writing data into file\n");
printf("write: ");
for(r = 0; r < sizeof(buf); r++) {
    printf("%d ", buf[r]);
}
printf("\n");
r = cfs_write(wfd, buf, sizeof(buf));
if(r < 0) {
    printf("Error writing data into file\n");
    cfs_close(wfd);
    cfs_remove("mycfs");
    return;
} else if(r < sizeof(buf)) {
    printf("Error writing data into file\n");
    cfs_close(wfd);
    cfs_remove("mycfs");
    return;
}
After writing the data, we should close the opened file:

```c
printf("close file\n");
cfs_close(wfd);
```

We have created a file and written data into the file. We will continue to read data from a file. We use a different file descriptor that we have declared.

We open a file and read data from the file using the `cfs_read()` function:

```c
printf("opening file for reading\n");
rfd = cfs_open("mycfs", CFS_READ);
if(rfd < 0) {
  printf("Error opening file\n");
cfs_remove("mycfs");
  return;
}
printf("reading data\n");
memset(buf, 0, sizeof(buf));
r = cfs_read(rfd, buf, sizeof(buf));
if(r < 0) {
  printf("Error reading file\n");
cfs_close(rfd);
cfs_remove("mycfs");
  return;
}
```

After reading the data, we print it to Terminal. Then, we close that file:

```c
printf("read: ");
for(r = 0; r < sizeof(buf); r++) {
  printf("%d ", buf[r]);
}
printf("\n");
printf("close file\n");
cfs_close(rfd);
```
Last, we delete our created file, since Contiki-NG has limited resources. Save this program. Then, you can compile and flash it to your Contiki-NG board. For instance, I flash this program to my Sky board:

$ make TARGET=sky
$ make TARGET=sky savetarget
$ make file-cfs-demo.upload

Now, you can perform monitoring on the target board. You can type this command:

$ make login

You should see the program output in Terminal. If not, you can reset your board. A sample of the program output can be seen in Figure 7-6.

Figure 7-6. Coffee file system demo on Sky platform
Demo: Contiki-NG and MySQL

In this section, we will build a Contiki-NG program that interacts with MySQL. The object of the demo is to show how Contiki-NG can store sensor data in a DBMS (Database Management System) system.

For this demo, we choose MySQL for the DBMS. In general, we will build our demo implementation as shown in Figure 7-7.

All sensor devices are attached to Contiki-NG motes that are deployed on a private WSN network. To communicate with an outside network, we will use a 6LoWPAN router. You learned about the 6LoWPAN router in Chapter 6.

We also implement middleware that works as a gateway. The middleware retrieves sensor data from Contiki-NG. Then, it stores sensor data into MySQL. We will develop this middleware using Node.js. Data communication between a middleware application and Contiki-NG uses JSON.

This demo needs two Contiki-NG motes at least. One mote will be used as the 6LoWPAN router. The rest will be deployed as sensor programs. In this demo, we will perform some tasks as follows:

- Design and build database
- Build 6LoWPAN router
- Develop Contiki-NG sensor application
- Develop Middleware application

We implement each task in the next sections.

**Preparation**

We need to prepare our demo. First, we install MySQL on a computer that is connected to the same network. For Ubuntu/Debian, you can install MySQL by typing these commands in Terminal:

```bash
$ sudo apt-get update
$ sudo apt-get install mysql-server
$ mysql_secure_installation
```

Another option is to download MySQL for your platform from [https://www.mysql.com/downloads/](https://www.mysql.com/downloads/).

You also should install MySQL Workbench. This is an optional task. MySQL Workbench is designed to build a design database and manage MySQL Server. You can install MySQL Workbench by typing this command:

```bash
$ sudo apt install mysql-workbench
```

**Design a Database**

In this section, we will build a database on MySQL. We create a database, called `contiki-ng-db`. Furthermore, we should create a table to store the sensor data.

We can create the database and table using MySQL Workbench. Create a table on the database, called `sensor`. You can create a table with the scheme that is shown in Table 7-3.
Table 7-3. Designing a Table for Demo

<table>
<thead>
<tr>
<th>Table Field</th>
<th>Properties</th>
</tr>
</thead>
</table>
| idsensor    | Datatype: INT  
Checked: Primary Key (PK), Not Null (NN), Auto Increment (AI) |
| sensor_name | Datatype: VARCHAR (15)  
Checked: Not Null (NN) |
| sensor_val  | Datatype: FLOAT  
Checked: Not Null (NN) |
| created     | Datatype: DATETIME |

This design is shown in Figure 7-8.

![Database design](image)

Figure 7-8. Database design for the project

Once done, you should deploy the database and table designs to MySQL Server. You also must configure security access, creating one user to access the database that will used for your application.
Build a 6LoWPAN Router

In this section, we will develop programs for Contiki-NG, using one Contiki-NG mote as a 6LoWPAN router. You already learned about that topic in Chapter 6. You can run the program sample from the `<contiki-ng>/examples/rpl-border-router/` folder.

Compile this project and flash it to one of the Contiki-NG motes. In this demo, I use a LaunchPad CC2560 board for testing. After deploying it to the mote, you can build the 6LoWPAN router by typing this command in Terminal:

```
$ make TARGET=srf06-cc26xx BOARD=launchpad/cc2650 connect-router
```

You can change the TARGET and BOARD if you use a different Contiki-NG platform.

Develop a Contiki-NG Sensor Application

We build sensor applications to perform sensing. For simple problems, I generate random values for the temperature and humidity sensors. You can implement your own real sensors for your project.

We will modify the websense program from the previous chapter. We call this project websense-db. You can see the project structure in Figure 7-9. We construct two sensor data: temperature and humidity. In the websense-db.c file, we add our sensor data in the `generate_routes()` process:

```c
static
PT_THREAD(generate_routes(struct httpd_state *s))
{
    char buff[35];

    PSOCK_BEGIN(&s->sout);
    //SEND_STRING(&s->sout, TOP);
```
int temperature = 15 + rand() % 25;
int humidity = 80 + rand() % 10;

sprintf(buff,"\"temp\":%u,\"hum\":%u\", temperature, humidity);
printf("send json to requester\n");

SEND_STRING(&s->sout, buff);
//SEND_STRING(&s->sout, BOTTOM);

PSOCK_END(&s->sout);
}

You can modify this code to replace the random values with real results from your sensor readings.

![Image of project structure of websense-db](image)

**Figure 7-9.** *Project structure of websense-db*

**Develop Middleware Application**

This middleware application has the responsibility of retrieving sensor data from Contiki-NG and then sending these data to MySQL Server. This application will be developed using Node.js. Set up the Node.js development environment by downloading it from [http://nodejs.org](http://nodejs.org).

To access MySQL from the Node.js application, you should install the MySQL driver for Node.js. You can use the official driver from MySQL. In this demo, we will use the mysql library. You can see it at [https://github.com/mysqljs/mysql](https://github.com/mysqljs/mysql).
First, we create a folder, called `sensor-db`. Then, we create a package.json file that consists of the required libraries for our application. You can write the following scripts to the `package.json` file:

```json
{
    "name": "sensor-db",
    "version": "1.0.0",
    "description": "saving sensor data to MySQL",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "Agus Kurniawan",
    "dependencies": {
        "express": "^4.15.2",
        "mysql": "^2.15.0",
        "request": "latest"
    }
}
```

Save these scripts. Now, you should install all the required libraries by typing this command:

```
$ npm install
```

This action will download all declared libraries in the `package.json` file.

Next, we build the Node.js program by creating a file, `index.js`. The program will run as web server and retrieve sensor data every five seconds. First, we declare our variables and database parameters in the `index.js` file:

```javascript
var express = require('express');
var request = require('request');
var app = express();
var http = require('http').Server(app);
```
var mysql = require('mysql');
var connection = mysql.createConnection({
    host : 'localhost',
    user : '<user>',
    password : '<db-password>',
    database : 'contiki-ng-db'
});

Change MySQL database parameters such as user and password as needed.

We connect to MySQL and run a web server through an ExpressJS engine. In this case, I use a web server with port 3000. You should change it to your own port. See here:

connection.connect();
app.use(express.static('public'));

app.get('/', function(req, res){
    res.send('WebSense DB');
});

http.listen(3000, function(){
    console.log('listening on *:3000');
    console.log('websense db was started');
});

Last, we retrieve the sensor data from the Contiki-NG mote. For this demo, I set a specific IPv6 address from the Contiki-NG mote: fd00::212:4b00:797:6083. You can change it to your Contiki-NG IPv6 address.
After obtaining the sensor data, we send this data to MySQL by calling
query(). We use a SQL statement to insert the data into the MySQL
database:

```javascript
var dataPusher = setInterval(function () {
    request.get('http://[fd00::212:4b00:797:6083]/',
    function(err,res,body){
        if(err){
            console.log(err);
            return;
        }
        var obj = JSON.parse(body);
        console.log(obj);

        connection.query({
            sql: 'INSERT INTO sensor(sensor_name,sensor_val,
                created) values(?,?,now())',
            timeout: 40000, // 40s
            values: [obj.temp,obj.hum]
        }, function (error, results, fields) {
            if(error){
                console.log(err);
                return;
            }
            console.log('inserted data to MySQL');
        });
    }, 5000);

    Save all the code. We will now test our program.
```
Testing the Project

In this section, we will test our project. Make sure you have deployed the Contiki-NG program for the 6LoWPAN router and sensor mote.

First, we activate the 6LoWPAN router. Navigate to the `<contiki-ng>/examples/rpl-border-router/` folder. Then, run the 6LoWPAN router application. I run it on LaunchPad CC2650:

```
$ make TARGET=srf06-cc26xx BOARD=launchpad/cc2650 connect-router
```

You should change the target and board to your own Contiki-NG board.

After running the program, you should turn on all sensor devices on the Contiki-NG motes. You should see all neighboring Contiki-NG motes. You can see my program output from the 6LoWPAN router program in Figure 7-10.

![Program output from rpl-border-router application](image)

**Figure 7-10.** Program output from rpl-border-router application
Now, you can run the middleware application. Open a new Terminal window and navigate to the middleware application. Then, type this command to run it:

$ node index.js

You should see sensor data from the Contiki-NG mote. Then, it is stored in the MySQL database. You can see the program output from the middleware application in Figure 7-11.

![Figure 7-11. Program output from middleware application](image)

You also can verify the sensor data in MySQL. Open MySQL Workbench. You should see all sensor data, as in Figure 7-12.
Summary

We have learned how to work with storage. We also reviewed the Coffee File System (CFS) in Contiki-NG. We also tested for file manipulation using CFS. Last, we developed a project to interact with the MySQL database. We store all sensor data in the database.

In the next chapter, we will learn how to work with a Cloud platform in Contiki-NG and then interact with it.
CHAPTER 8

Contiki-NG and Cloud Server

Cloud technology provides a serverless solution to many IT problems. It eliminates the effort required to provide hardware and software. These services could be SaaS (Software as Service), PaaS (Platform as Service), or IaaS (Infrastructure as Service). In this chapter, we will integrate Contiki-NG with Cloud servers. We will also review several Cloud platforms and then try to work with them.

The following is a list of topics we will cover in this chapter:

- Introduce Cloud server
- What is Cloud computing?
- Types of Cloud-computing deployments
- Review of Cloud server platforms
- Working with Contiki-NG and Microsoft Azure
- Working with Contiki-NG and Amazon AWS
Introduce Cloud Server

If you have experience with the software development cycle, you know it starts with getting requirements and ends with developing and deploying to the production machine servers. We should provide a server location, called a data center, which is used to house the machine servers.

Building a data center is not easy. Technically, a data center has class levels. Each class level has some criteria that should be met. We also should factor in electricity usage. The physical security of the server can be one of the key factors when you build your own data center.

You can also bring your machine servers to commercial data centers. In this option, you should think about security and policy. Whether you put your machine servers in your own data center or in a commercial data center, you still should manage all software within the machines yourself. Decide whether you should manage all your machines in one location.

Cloud computing can solve some issues related to configuring and deploying applications on machines. Several Cloud-computing companies provide automatic deployment and configuration for your application. They also provide geo-location server services that enable your application to deploy only in some regions. These services can serve all requests from different locations. You can optimize customer relationships by bringing your application to a customer’s local region.

Figure 8-1 shows how to deploy an application to a physical machine. You can select one of the options based on your needs. Each option has pros and cons. In this chapter, we will focus on how to integrate Contiki-NG with Cloud-computing technology. We will communicate with Cloud servers from the Contiki-NG environment.
Why Use Cloud Computing?

Cloud computing is designed to offer high flexibility for computing processes performed over the Internet. One of the big questions related to Cloud computing is why do we use it? This is a challenging question, especially for those who have already deployed an application or system on on-premises servers (local servers). For newcomers, bringing a deployment to Cloud computing is new ground.

Figure 8-1. Application deployment strategies on physical machine
Cloud-Computing Services

To integrate our application with Cloud computing, we should know the types of Cloud-computing services. In general, we can categorize Cloud-computing services as one of the following:

- Infrastructure-as-a-service (IaaS)
- Platform-as-a-service (PaaS)
- Software-as-a-service (SaaS)

Infrastructure-as-a-service (IaaS) provides a rent infrastructure, such as virtual machines (VMs), storage, networks, and operating systems. These providers serve your needs for machine servers, which are included in its managing dashboard. In general, you just pay for the resources you have already used.

Platform-as-a-service (PaaS) supplies an on-demand environment for developing and testing. You can create server and mobile applications without worrying about setting up or managing resource infrastructure.

Software-as-a-service (SaaS) provides all delivering software for your daily activities or certain projects. Providers usually offer subscription schemes for all delivering software. We can subscribe to and stop SaaS at any time.

Types of Cloud-Computing Deployments

To understand how to deploy Cloud computing, we should know several types of Cloud computing, such as the following:

- Public Cloud
- Private Cloud
- Hybrid Cloud
Public Cloud is a common type of Cloud deployment. You install and configure your application/platform on a provider’s Cloud. Your data will be stored into their storage. Some companies probably decline to store their data on public storage servers due to compliance policies.

A private Cloud enables your solution to be deployed into local servers. Cloud providers usually provide software to build a private Cloud with their platform.

Last, a hybrid Cloud combines the public and private Cloud approaches. You can work with a hybrid Cloud to ensure your business is run well.

**Review Cloud Server Platforms**

In this section, we will review several Cloud platforms that provide Cloud services–based IoT (Internet of Thing). This is a brief review so you are familiar with them when we implement IoT by enabling Cloud technology.

**Microsoft Azure**

Microsoft is a well-known company that provides software services. We all know Microsoft products, such as Windows, Microsoft Office, and SQL Server. Now, Microsoft has expanded its software business by implementing Cloud services, called Microsoft Azure.

You can easily set up Windows servers on Microsoft Azure. All configurations are done though the Azure portal. You can access it at [https://azure.microsoft.com](https://azure.microsoft.com). If you are a developer or a consultant with a Microsoft technology background, Microsoft Azure is a good choice to deploy Cloud computing.

Microsoft Azure is a global Cloud provider that provides Cloud services in several country regions. It can serve your needs locally.
Amazon AWS

Amazon started with its e-Commerce business. To support that business, Amazon built an IT business with Cloud technology, called Amazon AWS. Now Amazon AWS is a leading Cloud provider worldwide, available in most IT market segments. Storage and push notification services from Amazon AWS can be integrated with your mobile application. There are a lot of AWS services for your project needs. You can learn more about Amazon AWS, including registration, from the official website at https://aws.amazon.com.

You can integrate IoT projects with Amazon AWS. It provides a Cloud service, called AWS IoT. This service can serve your IoT needs. You can find it at https://aws.amazon.com/iot/.

Google Cloud

If you want to search for something online, you probably use Google. That’s Google. Google Cloud provides a solution for your Cloud needs. You can find Google Cloud services at https://Cloud.google.com.

If you are an Android developer, Google Cloud is probably the best choice to integrate with your mobile application.

IBM Cloud

IBM has long history with servers and software. This software provides most IT services. In the Cloud-computing era, IBM now offers Cloud services, called IBM Cloud, to leverage your business. To get more information about IBM Cloud, you can visit the IBM Cloud site at https://www.ibm.com/Cloud/.
Comparing Features of All Global Clouds

If you are interested in Cloud services from various worldwide Cloud providers, I recommend to read this site: http://compareCloud.in. You can see a feature comparison among Cloud providers such as AWS, Azure, Google Cloud, IBM Cloud, Oracle Cloud, and Alibaba Cloud.

Connecting Contiki-NG Motes to Cloud Servers

Contiki-NG is designed for IPv6 networks. Several Contiki-NG hardware devices have network protocol capabilities to connect to Cloud servers directly. In general, we need a 6LoWPAN router as a bridge between Contiki-NG motes and a Cloud server.

In Figure 8-2, we show a simple model of how Contiki-NG motes communicate with Cloud servers. We put in a 6LoWPAN router as a bridge to all Contiki-NG motes. The 6LoWPAN router will perform request/response mapping between Contiki-NG motes and outside servers.

Figure 8-2. Communicating between Contiki-NG and Cloud server
In the next section, we will build two demos to perform a communication between Contiki-NG and Cloud servers. For these demos, we will use Microsoft Azure and Amazon AWS.

**Demo 1: Contiki-NG and Microsoft Azure**

This demo has the objective of showing how to communicate between Contiki-NG and Microsoft Azure. We will perform the demo described in Figure 8-3. We will develop three applications to implement this demo, as follows:

- Sensor application on Contiki-NG
- 6LoWPAN router application on Contiki-NG and computer
- Middleware application
- Azure IoT Hub application

The middleware application will retrieve sensor data from Contiki-NG and then push that data to the Azure IoT Hub. Furthermore, the Azure IoT Hub will distribute the sensor data to all subscribed devices. You can see this process in Figure 8-3.
Next, we will implement our demo by performing some tasks, as follows:

- Prepare to set up Azure IoT.
- Develop application for Contiki-NG and middleware application.
- Test all programs.

Each task will be implemented in the next section.

**Preparation**

Before we develop a program in which Contiki-NG will access Azure IoT Hub, we should prepare Microsoft Azure. You should have an active account for Microsoft Azure to perform this demo. To prepare our development for Azure IoT Hub, we perform the following tasks:

- Create Azure IoT Hub.
- Add a new IoT device for Azure IoT.
- Copying IoT device keys for developing program.

Each step will be performed in the next section.
Creating Azure IoT Hub

To create Azure IoT Hub, you should have an active Azure account. You can log on to https://portal.azure.com/ with your account. On the left-hand menu on the Azure dashboard, click “IoT Hub” so you see the Azure IoT Hub dashboard, as shown in Figure 8-4.

![Figure 8-4. Azure IoT Hub dashboard](image)

Fill in all required fields to create a new IoT Hub, which is included in your Azure subscription scheme. For instance, I created an Azure IoT Hub, called contiki-ng. After creating an Azure IoT Hub, you should see it on the Azure IoT Hub dashboard, which is shown in Figure 8-5.
The next step is to register a new IoT device in order for it to access Azure IoT Hub. We will perform this task in the next section.

Registering a New IoT Device

Each IoT device that will access Azure IoT Hub should be registered in order to get an access key. Open your Azure IoT Hub and then click the “IoT Devices” menu so you see a list of IoT devices.

You can register a new IoT device by clicking the “+ Add” icon. You should get a registration form, shown in Figure 8-6.

Fill in your IoT device name. Select “Symmetric Key” for authentication type. You can check the box for “Auto Generate Keys” to generate keys automatically. Please select “Enable” for activating your IoT device. Click the Save button if you are done to add the IoT device.

Figure 8-5. Created Azure IoT Hub for Contiki-NG

The next step is to register a new IoT device in order for it to access Azure IoT Hub. We will perform this task in the next section.
After you have registered all IoT devices that will access Azure IoT Hub, you need to copy all IoT device keys. To obtain these keys, you can click your IoT devices on the Azure IoT Hub to get detailed information from the device.

You should see the IoT device key in the “Primary key” and “Connection string” fields. You can see it in Figure 8-7. You can copy the text that is indicated by an arrow. We will use these keys in our program.

Next, we will develop programs for Contiki-NG and a middleware application to communicate with the Azure IoT Hub. We will perform this task in the next section.

Figure 8-6. Creating a new IoT device

Copying Device Keys
Developing Application

There are three applications that we are going to develop. First, we will develop two programs for Contiki-NG, starting with the 6LoWPAN router and sensor application. A 6LoWPAN router will be implemented by deploying an rpl-border-router program to the Contiki-NG mote. The sensor application runs a web server that waits for incoming requests from the sensor.

The second program that we will develop is a middleware application. This program will request sensor data from Contiki-NG. Then, this program will push it to the Azure IoT Hub.

The last program is a sensor consumer program. The program will subscribe to the Azure IoT Hub to get sensor data from Contiki-NG. Once the middleware application pushes the sensor data to the Azure IoT Hub,
the sensor consumer program will get the sensor data that is pushed by the Azure IoT Hub.

We will implement these programs in the next section.

**Developing Programs for Contiki-NG**

On the Contiki-NG side, we develop two programs for Contiki-NG, starting with the rpl-border-router and sensor program. You can find the rpl-border-router in the Contiki-NG program samples. You should compile and upload rpl-border-router to one of the Contiki-NG motes. You can read how to implement rpl-border-router in Chapter 6.

A sensor program is a web server that serves requests for sensor data. We use the webservice program from Chapter 6. We set our project name as websense-Cloud. We rename websense.c as websense-Cloud.c. You can see the project structure of websense-Cloud in Figure 8-8.

![Figure 8-8. Project structure for Contiki-NG and Azure](image)

In the Makefile file, you can configure the demo project to include http-simple.c and the Contiki-NG project. You can write these scripts:

```makefile
CONTIKI_PROJECT = websense-Cloud
all: $(CONTIKI_PROJECT)

CONTIKI = ../..
PROJECT_SOURCEFILES += httpd-simple.c

include $(CONTIKI)/Makefile.include
```
We also modify the websense-Cloud.c file that is copied from the websense.c file (Chapter 6). In the generate_routes() method, we modify code in order to serve requests for sensor data. You can write this code:

```c
static
PT_THREAD(generate_routes(struct httpd_state *s))
{
    char buff[35];
    PSOCK_BEGIN(&s->sout);
    //SEND_STRING(&s->sout, TOP);

    int temperature = 15 + rand() % 25;
    int humidity = 80 + rand() % 10;

    sprintf(buff, "{"temp":%u,"hum":%u}\n", temperature, humidity);
    printf("send json to requester\n");
    SEND_STRING(&s->sout, buff);
    //SEND_STRING(&s->sout, BOTTOM);
    PSOCK_END(&s->sout);
}
```

Save all changes. Compile the rpl-border-router and websense-Cloud programs. Then, upload those to your Contiki-NG motes.

Next, we will develop the middleware application using Node.js.

**Building Azure Middleware Application**

We will develop an Azure middleware application to request sensor data from Contiki-NG and push sensor data to Azure IoT Hub. For implementation, we use Node.js.
First, create a folder for your project. Open Terminal and navigate to your project folder. Then, initialize your project, including required libraries. Type these commands:

```bash
$ npm init
$ npm install azure-iot-device azure-iot-device-mqtt express --save
```

Now, we will start to write a middleware application for Azure. Create a file, called `middleware-azure.js`. Initialize all required libraries and run the web server on port 3000. You also need information such as hostname, device ID, and shared access key. Write this code:

```javascript
'use strict';

var express = require('express');
var request = require('request');
var app = express();
var http = require('http').Server(app);

var clientFromConnectionString = require('azure-iot-device-mqtt').clientFromConnectionString;
var Message = require('azure-iot-device').Message;

var connectionString = 'HostName={youriothostname};DeviceId=myFirstNodeDevice;SharedAccessKey={yourdevicekey}';
var mydeviceId = 'contiki-ng-01';
var client = clientFromConnectionString(connectionString);

app.get('/', function(req, res){
    res.send('WebSense Azure Cloud');
});

http.listen(3000, function(){
    console.log('listening on *:3000');
    console.log('websense azure Cloud was started');
});
```

Chapter 8  Contiki-ng and Cloud Server
You should change value connectionString to the following values:

- {youriothostname} is your domain address (IP address) from your Azure IoT Hub
- myFirstNodeDevice and mydeviceId are your registered device ID
- {yourdevicekey} is a shared access key. You can find it in the primary key field from your registered IoT device; see Figure 8-7.

Next, we declare two functions. One of these functions, printResultFor(), is used to print all messages to Terminal. The other is the callback function, connectCallback, which requests sensor data from Contiki-NG motes:

```javascript
function printResultFor(op) {
  return function printResult(err, res) {
    if (err) console.log(op + ' error: ' + err.toString());
    if (res) console.log(op + ' status: ' + res.constructor.name);
  };
}

var connectCallback = function (err) {
  if (err) {
    console.log('Could not connect: ' + err);
  } else {
    console.log('Client connected');

    // Create a message and send it to the IoT Hub every second
    setInterval(function(){
      request.get('http://[fd00::212:4b00:797:6083]/',
        function(err,res,body){
          console.log('Message sent: ' + body);
        })
    });
  }
}
```
if(err){
    console.log(err);
    return;
}
var obj = JSON.parse(body);
console.log(obj);

var temperature = obj.temp;
var humidity = obj.hum;
var data = JSON.stringify({ deviceId: mydeviceId, temperature: temperature, humidity: humidity });
var message = new Message(data);
message.properties.add('temperatureAlert', (temperature > 30) ? 'true' : 'false');
console.log("Sending message: " + message.getData());
client.sendEvent(message, printResultFor('send'));
}

You should replace the value [fd00::212:4b00:797:6083] with the IPv6 address from the Contiki-NG mote that runs the websense-Cloud application.

Finally, we call our callback function from the Azure object:

client.open(connectCallback);
console.log('Contiki-NG Azure Middleware started.');

Save all these codes.
Developing Sensor Consumer Program

The last step is to develop a sensor consumer program to subscribe to Azure IoT Hub in order to retrieve sensor data.

We use the same project from the previous section. We need the azure-vent-hubs library to create the subscription. Type this command:

$ npm install azure-event-hubs --save

Then, we create a file, called azure-sensor-subscriber.js. We initialize a required library and configure a connection string for the Azure IoT Hub. Write this code:

'use strict';

var EventHubClient = require('azure-event-hubs').Client;
var connectionString = 'HostName={youriothostname};DeviceId=
myFirstNodeDevice;SharedAccessKey=
{yourdevicekey}';

var printError = function (err) {
    console.log(err.message);
};

var printMessage = function (message) {
    console.log('Message received: ');
    console.log(JSON.stringify(message.body));
    console.log('');
};

Change connectionString to the previous value.
Now, we subscribe and listen to incoming messages from Azure IoT Hub. Write this code:

```javascript
var client = EventHubClient.fromConnectionString(connectionString);
client.open().then(client.getPartitionIds.bind(client)).then(function (partitionIds) {
    return partitionIds.map(function (partitionId) {
        return client.createReceiver('$Default', partitionId, { 'startAfterTime' : Date.now()}).then(function(receiver) {
            console.log('Created partition receiver: ' + partitionId)
            receiver.on('errorReceived', printError);
            receiver.on('message', printMessage);
        });
    });
}).catch(printError);
```

Save all files.

We have written all programs for this demo. Next, we will test our project.

**Testing Contiki-NG and Azure Application**

In this section, I will assume you have already deployed all programs into your Contiki-NG motes. First, run the rpl-border-router program on the 6LoWPAN router. For instance, I run it on a TI LaunchPad CC2650. Then, navigate to the rpl-border-router project folder. Then, run this:

```
$ make TARGET=srf06-cc26xx BOARD=launchpad/cc2650 connect-router
```
After execution, that command will run tunslip6. Change TARGET to your Contiki-NG platform. If it succeeds, you can turn on another Contiki-NG mote that has installed the sensor application.

Try to perform ping6 to check that your Contiki-NG mote can be reached from a computer. You can see the program output from the rpl-border-router application in Figure 8-9.

![Image of 6LoWPAN router application output]

Figure 8-9. Running 6LoWPAN router application

Now, you can run the middleware application in Terminal. You can type this command:

$ node middleware-azure.js

Then, you can run the sensor application to listen for incoming sensor data from Azure IoT Hub. Open a new Terminal and then type this command:

$ node azure-sensor-subscriber.js
The middleware application will request sensor data from the Contiki-NG mote every three seconds. After obtaining sensor data, the middleware application will push the data to the Azure IoT Hub.

Once sensor data has reached the Azure IoT Hub, it will be distributed to all subscribers.

You can see a sample of the program output of the middleware application and the sensor application in Figure 8-10.
Demo 2: Contiki-NG and Amazon AWS

In this demo, we will perform the same scenario as in the first demo. We will use Amazon AWS. To communicate with IoT devices, Amazon AWS provides a service called AWS IoT. You can read about it at https://aws.amazon.com/iot/.

Our demo scenario can be seen in Figure 8-11. This is similar to the first demo. A middleware application will get sensor data from Contiki-NG and then push it to Amazon AWS IoT.

Next, we will implement our demo by performing some tasks, as follows:

- Prepare to set up AWS IoT.
- Develop applications for Contiki-NG and middleware.
- Test all programs.

Each task will be implemented in the next section.

Figure 8-11. Demo project scenario for Contiki-NG and Amazon AWS IoT
Preparation

In this section, we will set up AWS IoT to create a thing. Then, we will configure security tasks, such as creating a certificate and its keys. To set up AWS IoT for our demo, we will perform the following tasks:

- Create a new IoT thing.
- Create a policy for AWS IoT.
- Attach a policy and a thing to a certificate.

These tasks will be explained in the next section.

Creating a New IoT Thing

Each IoT device that wants to access AWS IoT should be registered so as to obtain the access keys included with the security certificate.

First, open a browser and navigate to https://aws.amazon.com/iot/, so you should see the AWS IoT dashboard shown in Figure 8-12.

![AWS IoT console management](image)

**Figure 8-12.** AWS IoT console management
On the left-hand menu, click “Manage Things.” If you don’t create things yet, you can click “Register a thing” to register a new IoT device.

After clicking, you should get a form like that shown in Figure 8-13. To simply begin a registration process, click “Create a single thing.”

![Creating AWS IoT things](image)

**Figure 8-13. Creating AWS IoT things**

You should fill in the IoT device information. A thing name is required. For instance, I called my thing contiki-ng-middleware. You can see it in Figure 8-14.
If you have finished creating a thing, you should get the form that is shown in Figure 8-15. To access AWS IoT, we need a certificate. In this demo, we create a new certificate. You can use your own certificate. Click “Create certification.”

Figure 8-14. Setting an IoT device

Figure 8-15. Adding a certificate
After clicking that button, you should see the form that is shown in Figure 8-16. To activate your certificate, click the “Activate” button.

![Certificate created screen](image)

**Figure 8-16. Generating certificate and keys**

Download all files for the certificate and public/private keys. You also need to download a root CA (Certificate Authority) for AWS IoT from the link shown in Figure 8-16. Put them in a folder; for instance, cert.s. You can see my certificate and key files in Figure 8-17.
Now, you should see the created thing in the Manage Things dashboard, shown in Figure 8-18. You can create more than two things for AWS IoT.

Next, we will create a policy to enable our thing to access the AWS IoT Hub.
Creating a Policy for AWS IoT

In this section, we will create a policy for AWS IoT. We need this policy to manage all device access. You can find a policy dashboard by clicking “Secure,” then “Policies.” You can see it in Figure 8-19.

Figure 8-19. AWS IoT policy dashboard

Click “Create a policy.” Then, you should see the form shown in Figure 8-20. Fill in the policy name. Next, you should add statements to work with AWS IoT. You should add resource, action, and effect for four items, as follows:

```json
{
    "Version": "2012-10-17",
    "Statement": [
```
"Effect": "Allow",
"Action": "iot:Connect",
"Resource": "*"
},
{
"Effect": "Allow",
"Action": "iot:Publish",
"Resource": "*"
},
{
"Effect": "Allow",
"Action": "iot:Subscribe",
"Resource": "*"
},
{
"Effect": "Allow",
"Action": "iot:Receive",
"Resource": "*"
]"}
After creating a policy, you can verify your policy in Secure ➤ Policies from the AWS IoT dashboard. You can see it in Figure 8-21.

Figure 8-20. Adding a new policy for AWS IoT

After creating a policy, you can verify your policy in Secure ➤ Policies from the AWS IoT dashboard. You can see it in Figure 8-21.

Figure 8-21. A created policy shows up in Policies dashboard.

Next, we map a policy and a thing to a certificate. We will perform this in the next section.
Attaching a Policy and a Thing to a Certificate

Your policy and thing should be mapped to a certificate that is already activated in AWS IoT. To perform this task, you can visit Secure ➤ Certificates. You should see your certificate. Click the ... icon so you see the context menu shown in Figure 8-22.

![Open context menu of a certificate of AWS IoT](image)

**Figure 8-22.** Open context menu of a certificate of AWS IoT

Click menu Attach policy so you see the form dialog that is shown in Figure 8-23. Select your policy that has already been created. If done, click the Attach button.
Next, we should attach the thing to a certificate. From Figure 8-22, select menu “Attach thing” so you get the form dialog shown in Figure 8-24.

**Figure 8-23.** Attach a policy to a certificate

**Figure 8-24.** Add an IoT device (thing) to a certificate
Select your thing that was already created.

Finally, you have to set up AWS IoT. Next, we will develop programs for Contiki-NG and the middleware application.

**Developing Application**

In this section, we will develop applications for Contiki-NG and Node.js to access AWS IoT.

**Developing Programs for Contiki-NG**

Since our demo scenario is similar to the previous demo, we use the same configuration. You need one Contiki-NG mote as the 6LoWPAN router. Other Contiki-NG motes run the sensor application.

Please read the first demo to build programs for Contiki-NG.

**Building Middleware Application for AWS IoT**

We build an AWS middleware application to request sensor data from Contiki-NG and push sensor data to AWS IoT. For implementation, we use Node.js.

To create a project, you can create a folder. Then, open Terminal and navigate to your project folder. Then, initialize your project, including the required libraries. Type these commands:

```
$ npm init
$ npm install aws-iot-device-sdk request express --save
```

Now, we start to write a middleware application for AWS. Create a file, called `middleware-aws.js`. We now call the required libraries and configure AWS IoT. You can write this code:

```javascript
var awsIot = require('aws-iot-device-sdk');
var express = require('express');
var request = require('request');
```
var app = express();
var http = require('http').Server(app);

var device = awsIot.device({
  keyPath: 'certs/private.pem.key',
  certPath: 'certs/certificate.pem.crt',
  caPath: 'certs/root-CA.pem',
  host: '<hostname>.iot.<region>.amazonaws.com',
  clientId: 'contiki-ng',
  region: '<region>'
});

You should change the values for keyPath, certPath, caPath, and region to those for AWS IoT.

We also run a web server using Express on port 3000:

app.get('/', function(req, res){
  res.send('WebSense Azure Cloud');
});

http.listen(3000, function(){
  console.log('listening on *:3000');
  console.log('websense aws Cloud was started');
});

Next, we listen for the connect event from AWS IoT. Once the middleware application is connected to AWS IoT, we request sensor data from Contiki-NG every three seconds. For testing, I request data from Contiki-NG with the IPv6 address [fd00:212:4b00:797:6083]. You can change it. You also can change mydeviceId:

var isSubscribe = true;
device
  .on('connect', function() {
    console.log('connected to AWS IoT.');
});
// optional to subscribe
if (isSubscribe)
    device.subscribe('contiki-ng-sensor');

setInterval(function(){

    request.get('http://[fd00::212:4b00:797:6083]/', function (err, res, body){
        if (err){
            console.log(err);
            return;
        }
        var obj = JSON.parse(body);
        console.log(obj);
        var temperature = obj.temp;
        var humidity = obj.hum;
        mydeviceId = 'fd00::212:4b00:797:6083';
        var data = JSON.stringify({ deviceId: mydeviceId,
                                 temperature: temperature, humidity: humidity });
        device.publish('contiki-ng-sensor', data);
        console.log('sent: ', JSON.stringify(data));
    });
}, 3000);
});
Last, we can subscribe to the contiki-ng-sensor channel:

```javascript
// optional to subscribe
device
  .on('message', function(topic, payload) {
    console.log('recv: ', topic, payload.toString());
  });

console.log('Contiki-NG AWS Middleware started.');
```

Save all code.

Next, we will test this project.

## Testing Contiki-NG and AWS IoT

Deploy all programs to Contiki-NG. Then, run the 6LoWPAN router and the middleware application:

```
$ node middleware-aws.js
```

This program will connect to AWS IoT. After connecting, this application will retrieve sensor data from Contiki-NG. It will then push the sensor data to AWS IoT.

Since the middleware application subscribes to AWS IoT, this program will receive incoming data from AWS IoT. You can see my program output in Figure 8-25.
For a subscriber tool, you can use the Subscription test application. You can find it on the Test menu.

After clicking it, you will see the form shown in Figure 8-26. Select “Subscribe to a topic.” Then, fill in “contiki-ng-sensor” for listening incoming messages.
You have finished integrating between Contiki-NG and the Cloud server. You practice more to hone your skills.

Summary

We have learned what Cloud computing is. We also developed programs to allow interaction between Contiki-NG motes and Cloud servers. We tested these programs using Microsoft Azure and Amazon AWS. Next, you could explore various features in Azure and AWS. You can use other Cloud platforms to integrate with Contiki-NG.
Index

A, B

Address Resolution Protocol (ARP), 195
Amazon AWS, Contiki-NG and add IoT device to certificate, 281
attach policy, 280, 281
AWS IoT policy, 277
policy creation, 277, 279
AWS IoT registration, 272
certificate creation, 274
console management, 272
contiki-ng-middleware, 273
Creating AWS IoT things, 273
public/private keys, 275
AWS IoT service, 271
AWS IoT setup, 272
AWS middleware application, 282
contiki-ng-sensor channel, 285
programs, 282
testing
incoming messages, 287
6LoWPAN router, 285
middleware application, 285–286
Azure IoT Hub
Azure middleware application, 263
copying IoT device keys, 260–261

creation
Azure IoT Hub
dashboard, 258
contiki-ng, 258–259
register new IoT device, 259
Auto Generate Keys, 259
registration form, 260
sensor consumer program
azure-vent-hubs library, 267
connectionString value, 267
Azure middleware application
callback functions, 265–266
connectionString value, 265
printResultFor() functions, 265
requests sensor data, 265–266
sensor data, 263
shared access key, 264

C
Carbon monoxide sensor, 143
Cloud computing, 251
services
infrastructure-as-a-service (IaaS), 252
platform-as-a-service (PaaS), 252
software-as-a-service (SaaS), 252
INDEX

Cloud computing (cont.)
  types
    hybrid cloud, 253
    private cloud, 253
    public cloud, 253
Cloud server
  Amazon AWS, 254
  data center, 250
  geo-location server
    services, 250
  Google Cloud, 254
  IBM Cloud, 254
  Microsoft Azure, 253
  physical machine
    deployment, 250–251
Coding conventions, 61
Coffee File System (CFS), 248
Communication models, 96
  Contiki motes, 97, 126–127
    receiving broadcast
    messages, 128–131
    sending broadcast
    messages, 127–128
    serial communication, 98,
    100–104
Concurrency, 67
  demo-process app, 72
  process, 69–70
  task scheduling, 88, 90, 92–94
  threading, 83–88
  working with timers, 72
    Clock library, 72–73
    CTimer library, 76
    demo-timer, 78–82
  Etimer library, 75
  RTimer library, 77
  Stimer library, 74, 75
  Timer library, 73, 74
  Contiki-NG, 13
    middleware application, 132
      architecture, 133
      defined, 133
      implementation, 134
      testing, 134–135
    set up development
      environment, 19–21
  Contiki-NG motes
    connecting to Cloud Servers, 255
    6LoWPAN router, 255
  Contiki-NG programs
    rpl-border-router program, 262
    sensor program, 262
    websense-Cloud project, 262
      generate_routes()
      method, 263
  Contiki-NG shell, 112
    code structure, 113
    customizing, 123–126
    executing sample, 115
    TI CC2650 LaunchPad
      board, 114
  Contiki OS, 4
    architecture, 5
    programming flow, 6
    set up development
      environment, 13
      Instant Contiki, 13–18
    manual installation, 18–19

290
WSN hardware, 6
  Iris, 9–10
  Mica2, 7–8
  MICAz, 7
  TelosB, 8–9
  TinyOS motes, 10–11
  Z1 platform, 11–12
Contiki simulator, 35
  application debugging, 45
  hardware debugger, 44–45
  LED indicators, 45
  printf() function, 45
  running, 37–43
  setting up, 36
counter_process function, 60

D, E, F, G
Demo-counter, 57

H
Hello World Application for Contiki, 26, 28–30, 32–34
Humidity and Temperature Sensor, 140–141

I, J, K
Internet Protocol Version 6 (IPv6) networking, 185–188

L
Libraries, 55–56

M
Microcontroller (MCU), 2
Microsoft Azure, Contiki-NG and Azure IoT Hub
  application, 256
  communication, 257
6LoWPAN router application, 256
middleware application, 256
sensor application, 256
testing
  6LoWPAN router application, 269
  middleware application, 270
  rpl-border-router program, 268
  sensor application, 270

N, O
NETSTACK, 168
Networking
  Contiki-NG NullNet, 199–200
  IPv6, 185–188, 194–198
  6LoWPAN network, 201
    implementing, 203–205, 207–208
    overview, 201–202
    using COOJA, 209–212
  MAC layer, 172
  NETSTACK, 168–169
  network layer, 170–171
  radio layer, 173
  RDC layer, 173
INDEX

Networking (cont.)
   RESTful server for Contiki-NG, 214
   implementing demo, 215–216
   6LoWPAN router, 217
   preparation, 215
   program, 219–222
   testing demo, 223–224
   WebSense node, 217–218

Routing on Contiki-NG, 189
   basic routing, 189–190
   operations on first mote, 193
   operations on second mote, 194
   single-hop and multi-hop networking, 190–191

Simulation using COOJA, 174
   running, 183, 185
   simulation project, 174–176

UDP client motes, 180–182
UDP server mote, 177, 179–180

P, Q

Programming model, 47–48
   basic syntax, 49
   creating project, 49–50
   programming language, 50–51
   program structure, 48
Protothreads, 52–53, 55

R

Radio Duty Cycling (RDC)
   layer, 173
Raspberry Pi, Contiki and, 25

S

Sensing and actuating, 137
   communication model, 138
   in Contiki-NG, 145
   demo, 146–147
   program, writing, 148–150, 154–155
   project, creating, 147, 153
   testing, 151–152, 155–156
   customizing, 156
   drivers for Contiki-NG, 159–165
   expansion connector, 156, 158–159
   sensor and actuator devices, 139
   active buzzer, 144
   gas sensor, 142–143
   LED, 143
   motor, 144–145
   soil moisture sensor, 141–142
   temperature and humidity sensors, 139–141
   WSN mote, 138
Serial communication, 97
Shell, Contiki, 104
  API, 106
  application, 107
  customizing, 115–120, 122
  example, 105
  help command, 111
  limitations, 104
  running sample, 108–109
  sky as target, 110
Storage models, 225–227
  CFS, 231–234, 236–237
  Contiki-NG and MySQL, 238
    database, designing, 239–240
  6LoWPAN router, 241
  middleware application, 242–245
  preparation, 239
  sensor application, 241–242
  testing the project, 246–248
  local storage, working with, 227–231

T
  TelosB, 146

U
  Universal Asynchronous Receiver/Transmitter (UART), 97

V
  Virtual-sensor, 61
    code for, 64–65
    functions, 63
    program output, 65
    project structure, 62

W, X, Y, Z
  Wireless Sensor Networks (WSN), 1–3, 225
    connectivity model, 4
    general model of, 3
    hardware to computer, 22–25