OpenDQ documentation

version 0.2

OpenMote Technologies, S.L.
info@openmote.com

May 2015

Table of Contents

List of Figures

List of Tables

Chapter 1

OpenDQ introduction

OpenDQ is an open source project started in 2014 by OpenMote Technologies that implements a MAC (Medium Access Control) protocol targeted at data collection scenarios using low-power wireless technologies, i.e., active RFID (RadioFrequency IDentification) or WSN (Wireless Sensor Networks).

In a data collection scenario, as depicted in Figure 1.1, there are three types of devices; a computer, a gateway and the nodes. The nodes are computing devices equipped with a low-power wireless radio transceiver to communicate, various analog or digital sensors to acquire data from the environment and, finally, batteries that provide energy to the whole system. In a given scenario, i.e., a warehouse, the combination of the sensing, computing and communication capabilities of such devices allows to collect measurements of physical parameters, i.e., temperature or relative humidity, in a distributed manner. The acquisition of data by nodes can be performed periodically, i.e., every minute, or on demand, but the collection of data is always performed on demand. Thus, the role of the gateway is to trigger the collection process to obtain the information of all nodes that are within its communication range and to provide such information to the computer. Since the coverage may be limited due to wireless propagation or interference conditions, one or more gateways are typically deployed in a given scenario to ensure that all nodes can be collected appropriately. Finally, the computer is responsible trigger the data collection process and to collect and store the data provided by the gateway(s), as well as to provide it to third parties.

Since nodes are battery-powered computing devices that communicate using low-power wireless technologies, the data collection scenarios described above presents three main challenges:

- First, the number of nodes that may be present within the gateway communication range may be large, i.e., hundreds to thousands.
- Second, the number of nodes that may be present within the gateway communication range is unknown a priori, i.e., because nodes are mobile.
- Third, the traffic patterns generated by the nodes in the data collection phase is bursty, i.e., all nodes transmit their data simultaneously.

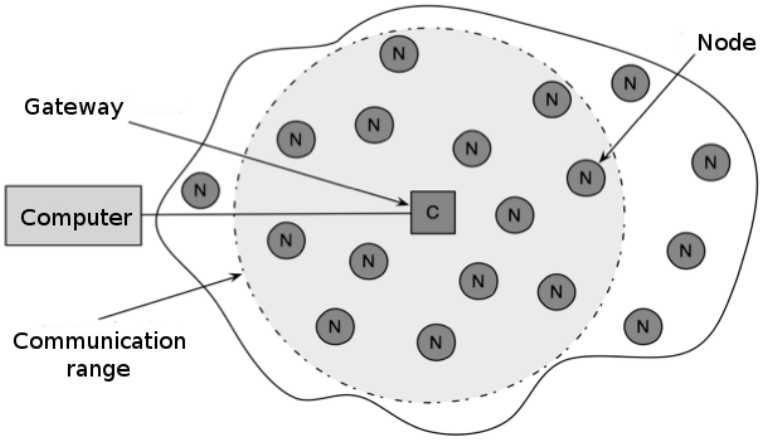


Figure 1.1: A data collection scenario

Given the above scale, mobility and traffic pattern constraints, existing MAC protocols targeted at data collection scenarios are not suitable. On the one hand, TSCH (Time Synchronized Channel Hopping), as defined in IEEE 802.15.4e [1], is not suitable because it assumes that nodes are statically deployed in a given location and cannot cope with bursty traffic patterns. On the other hand, FSA (Frame Slotted ALOHA), as defined in ISO 18000-7:2009 [2], is not suitable because it is not able to cope with a large number of devices and the bursty traffic patterns.

Considering the limitations of existing MAC protocols for data collection scenarios, OpenMote Technologies has developed the OpenDQ project. OpenDQ implements a MAC protocol that combines a packet-based PS (Preamble Sampling) mechanism with a DQ (Distributed Queuing) channel access mechanism. On one hand, the packet-based PS mechanism provides the means for the gateway to synchronize an indeterminately large number of nodes that are within its communication range to trigger the data collection process. On the other hand, DQ is responsible to manage access to the wireless channel to ensure that data from all the nodes present within the gateway communication range can be collected efficiently in terms of time and energy regardless of the number of nodes present in the gateway communication range[1].

Currently, the OpenDQ project is implemented using OpenMote hardware platform, which is based on a ARM Cortex-M3 micro-controller and a 2.4 GHz radio transceiver compatible the IEEE 802.15.4 standard [3]. However, the OpenDQ project can be easily ported to other computer architectures, i.e., MSP430, and low-power wireless communication technologies, i.e., Sub-GHz. The only constraint is that the implementation needs to be compliant with existing regulation for unlicensed bands, i.e., FCC in the US [4] or ETSI in Europe [5].

The remainders of this document assumes a certain knowledge of MAC protocols for WSN and RFID, as well as PS protocols, FSA and DQ. Thus, for readers not familiar

[1]For comparison purposes, the OpenDQ project also implements FSA (Frame Slotted ALOHA).

with the related topics it is advised to have a look at the following references.

For an introduction to MAC protocols for WSN and RFID have a look at the following references:

- "Dheeraj K Klair, Kwan-Wu Chin, and Raad Raad. A survey and tutorial of RFID anti-collision protocols. Communications Surveys & Tutorials, IEEE, 12(3):400–421, 2010. [6]
- A Bachir, M. Dohler, T. Watteyne, and K K Leung. MAC essentials for wireless sensor networks. Communications Surveys & Tutorials, IEEE, 12(2):222–248, 2010. [7].
- P Huang, L. Xiao, S Soltani, M Mutka, and N Xi. The Evolution of MAC Protocols in Wireless Sensor Networks: A Survey. Communications Surveys & Tutorials, IEEE, (99):1–20, 2012. [8].

For a survey on PS protocols have a look at the following reference:

- Cristina Cano, Boris Bellalta, Anna Sfairopoulou, and Miquel Oliver. Low energy operation in WSNs: A survey of preamble sampling MAC protocols. Computer Networks, 55(15):3351–3363, 2011. [9].

Finally, for references related to the DQ channel access mechanism have a look at:

- W. Xu and G. Campbell. A near perfect stable random access protocol for a broadcast channel. In Communications (ICC), 1992 IEEE International Conference on, pages 370–374, 1992. [10].
- W. Xu and G. Campbell. A distributed queueing random access protocol for a broadcast channel. SIGCOMM Comput. Commun. Rev., 23(4):270–278, 1993. [11].
- L. Alonso, R. Agusti, and O. Sallent. A near-optimum MAC protocol based on the distributed queueing random access protocol (DQRAP) for a CDMA mobile communication system. Selected Areas in Communications, IEEE Journal on, 18(9):1701–1718, 2000. [12].
- J. Alonso-Zarate, C. Verikoukis, E. Kartsakli, A. Cateura, and L. Alonso. A near-optimum cross-layered distributed queuing protocol for wireless LAN. Wireless Communications, IEEE, 15(1):48–55, 2008. [13].
- Pere Tuset, Francisco Vazquez, Jesus Alonso, Luis Alonso, and Xavier Vilajosana. LPDQ: a self-scheduled TDMA MAC protocol for one-hop dynamic low-power wireless networks. Elsevier Pervasive and Mobile Computing, Special Issue on "Internet of Things", 2014. [14].
- Pere Tuset, Francisco Vazquez, Jesus Alonso, Luis Alonso, and Xavier Vilajosana. Experimental energy consumption of FSA and DQ for data collection scenarios. MDPI Sensors, Special Issue on "Wireless Sensor Networks and the Internet of Things", (14):13416–13436, 2014. [15].

Chapter 2

OpenDQ hardware

As stated earlier, the OpenDQ project is designed to run on the OpenMote platform. The OpenMote platform is composed of three different boards: the OpenMote-CC2538, the OpenBase and OpenBattery. These boards are described in the following subsections. In addition to OpenDQ, the OpenMote-CC2538 and its companion boards can also run different operating systems and stacks for the IoT (Internet of Things). In particular, the OpenMote boards are fully compatible with Contiki ($http://www.contiki-os.org/$) and OpenWSN ($http://openwsn.berkeley.edu/$), and are partially supported by RioT ($http://www.riot-os.org/$).

2.1 OpenMote-CC2538

The OpenMote-CC2538 board, depicted in Figure 2.1, is the core of the OpenMote platform. It has an XBee Pro form factor and embeds the Texas Insturments SoC (System On Chip) [16, 17] that embeds a 32-bit ARM Cortex-M3 micro-controller and an IEEE 802.15.4 radio transceiver operating at the 2.45 GHz band. The micro-controller runs up to 32 MHz and includes 32 kbytes of RAM and 512 kbytes of Flash memory, as well as different peripherals (GPIOs, UART, SPI, I2C, ADC, Timers, etc.). The OpenMote-CC2538 also contains four LEDs (green, yellow, orange and red), two buttons (reset and user) and a SMA antenna connector. The OpenMote-CC2538 is intended to be plugged to the OpenBase and the OpenBattery boards to complete its functionality; for example, the OpenMote-CC2538 can be connected to a computer for debugging purposes using an OpenBase. The schematic of the OpenMote-CC2538 board can be found online in [18].

2.2 OpenBase

The OpenBase board, depicted in Figure 2.2, serves as an interconnect board that provides connectivity and debugging capabilities for the OpenMote-CC2538 board. Regarding connectivity, the OpenBase provides a UART interface, a USB interface and a

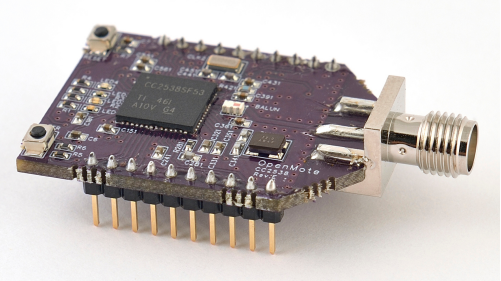


Figure 2.1: OpenMote-CC2538 board.

Ethernet interface. Power to the OpenBase can be provided via either USB port, i.e., USB_FTDI and USB_CC2538, and can be selected via the power switch. Regarding debugging, the OpenBase provides full access to the XBee pins, an ARM JTAG 10-pin connector and a current probe to measure the current consumption of the OpenMote-CC2538. The schematic of the OpenBase board can be found online in [19].

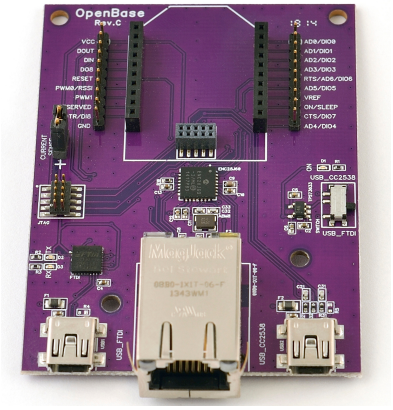


Figure 2.2: OpenBase board.

2.3 OpenBattery

The OpenBattery boards, depicted in Figure 2.3, provides energy to the OpenMote-CC2538 via 2xAAA batteries and provides access to three different sensors via an I2C bus. In particular, the OpenBattery provides a 3-axis acceleration sensor (Analog Devices ADXL346), a light sensor (Maxim MAX44009) and a temperature and relative humidity sensor (Sensirion SHT21). The schematic of the OpenBattery board can be found online in [20].

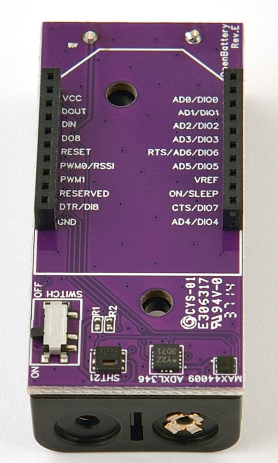


Figure 2.3: OpenBattery board.

Chapter 3

Installing OpenDQ

This section explains how to install the software that is required to compile, run and debug the OpenDQ project on a computer.

3.1 Prerequisites

OpenDQ requires Ubuntu 14.04 to operate, either the 32 bit or the 64 bit version. The reason why Ubuntu 14.04 LTS is selected as the supported operating system is because it is a LTS (Long Term Support) release, meaning that it is more stable that regular releases and that it will remain supported until April 2019. However, OpenDQ is also known to run on other versions of the GNU/Linux operating system, as well as on Windows and MacOS operating systems. However, these platforms are not supported by the current documentation.

The remaining of this subsection assumes that Ubuntu 14.04 LTS is installed in a computer that has access to the Internet. If you do not have Ubuntu 14.04 LTS installed on your computer, please download it (http://www.ubuntu.com/download) and install it in the computer prior to continuing. Please notice that it is also possible to install Ubuntu 14.04 LTS and OpenDQ in a virtual environment, i.e. VMWare Player or VMWare Workstation.

Once Ubuntu 14.04 is installed in a computer or virtual machine, please make sure that the operating system is updated with the latest security patches and software versions by executing the following commands. If the process is successful, a screen similar to Figure 3.1 will be displayed.

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

After installing and updating the operating system, proceed to install the remaining software required to compile, run and debug the OpenDQ project, as described in the following subsections.

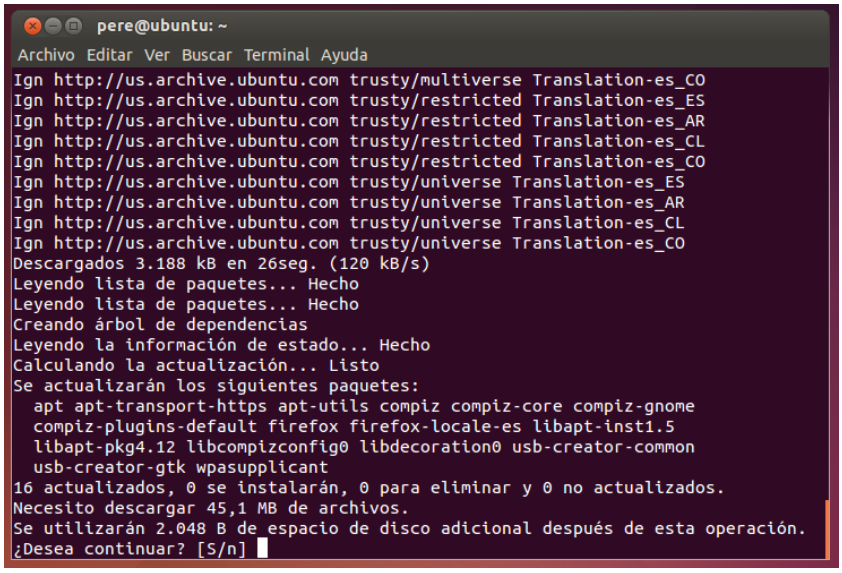


Figure 3.1: Updating the Ubuntu 14.04 LTS operating system.

3.2 Git

The OpenDQ project uses Git as the version control system to manage the source code. To install git and the related dependencies execute the following command on a terminal window.

```
sudo apt-get install git
```

3.3 Python

The OpenDQ project requires Python 2.7 and certain libraries, i.e., pyserial, numpy, scipy and matplotlib. To install Python and the related dependencies execute the following command on a terminal window.

```
sudo apt-get install python python-serial python-numpy python-scipy python-matplotlib
```

3.4 Toolchain

The OpenDQ project requires an arm-none-eabi-gcc toolchain to cross-compile the C source code for the ARM Cortex-M architecture. To install the arm-none-eabi-gcc toolchain execute the following commands on a terminal window.

```
sudo apt-get remove binutils-arm-none-eabi gcc-arm-none-eabi
sudo add-apt-repository ppa:terry.guo/gcc-arm-embedded
```

```
sudo apt-get update
sudo apt-get install build-essential gcc-arm-none-eabi=4.9.3.2015q1-0utopic14
```

3.5 Segger J-Link

The OpenDQ project requires the Segger J-Link software to load and debug the binaries for the ARM Cortex-M architecture. To install the Segger J-Link software go to the follwing link (https://www.segger.com/jlink-software.html) and download the latest TGZ archive for the J-Link software for your operating system, i.e. Linux 32 or 64 bits. Currently the version is Linux v.498b.

Once downloaded, uncompress it and copy it to the */opt* directory by executing the following commands.

```
tar xzvf Jlink_Linux_V498b_x86_64.tgz
sudo mkdir /opt/segger
sudo mv JLink_Linux_V498b_x86_64 /opt/segger/jlink
```

Finally, add the directory to the $PATH$ variable by editing the *.bashrc* file in the user home directory.

Chapter 4

Running OpenDQ

This section explains how to obtain the OpenDQ project from GitHub and how to load, execute and debug it on the OpenMote platform. The section assumes that the user has the appropriate hardware and has setup a working environment, as described in Section 2 and Section 3 respectively. If that is not the case, please refer to such sections prior to following this section.

4.1 Hardware requirements

In order to run and debug the OpenDQ software a minimum of hardware is required. At a minimum, two OpenMote-CC2538, one OpenBase and one OpenBattery are required. Such hardware can be bought as an OpenBronze kit, as depicted in Figure 4.1, from the OpenMote website. However, a OpenGold kit is required in order to be able to demonstrate all the characteristics of the OpenDQ implementation.

In addition to OpenBronze kit, the following hardware is required in order to run and debug the OpenDQ project:

- Segger J-Link EDU. Required to load and debug the OpenDQ project on the OpenMote-CC2538 hardware.
- ARM 20-to-10 pin adapter. Required to connect the Segger J-Link EDU to the OpenBase.
- USB mini cable. Required to power the OpenBase board.
- 2xAAA batteries. Required to power the OpenBattery board.

4.2 Obtain the OpenDQ project

The OpenDQ project is stored in GitHub (https://github.com). In order to obtain the OpenDQ project from GitHub navigate to the folder where you want the software to be installed. In this documentation it is assumed that the OpenDQ resides in a folder

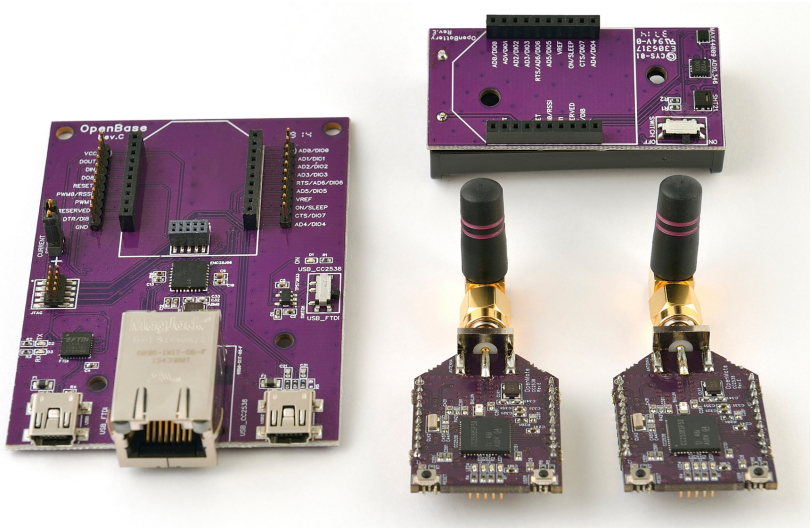


Figure 4.1: Contents of an OpenBronze kit.

named OpenDQ in the user home folder. To clone the OpenDQ project from GitHub issue the following commands on a terminal window.

```
cd ~
git clone http://github.com/OpenMote/OpenDQ ~/OpenDQ
```

If the process is successful the OpenDQ project will be in your home folder and a screen similar to Figure 4.2 should be displayed. Also, a folder with the contents displayed in Figure 4.3 should be created in the user home folder.

4.3 Generate the Texas Instruments CC2538 firmware library

Once the OpenDQ project source code has been obtain, the next step is to create the statically linked library that contains the firmware for the Texas Instruments CC2538 SoC. This step is required because the OpenDQ project has file names that collide with those of the Texas Instruments CC2538 firmware library. In order to compile the Texas Instruments CC2538 firmware library navigate to the appropriate directory and execute the $libcc2538_setup.py$ Python script using the following commands.

```
cd ~/OpenDQ/platform/cc2538/library
python libcc2538_setup.py
```

If the process is successful the Texas Instruments CC2538 firmware library will be automatically generated and copied to the appropriate directory and a screen similar to Figure 4.4 should be displayed. To check if the Texas Instruments CC2538 firmware

Figure 4.2: Cloning the OpenDQ project from GitHub.

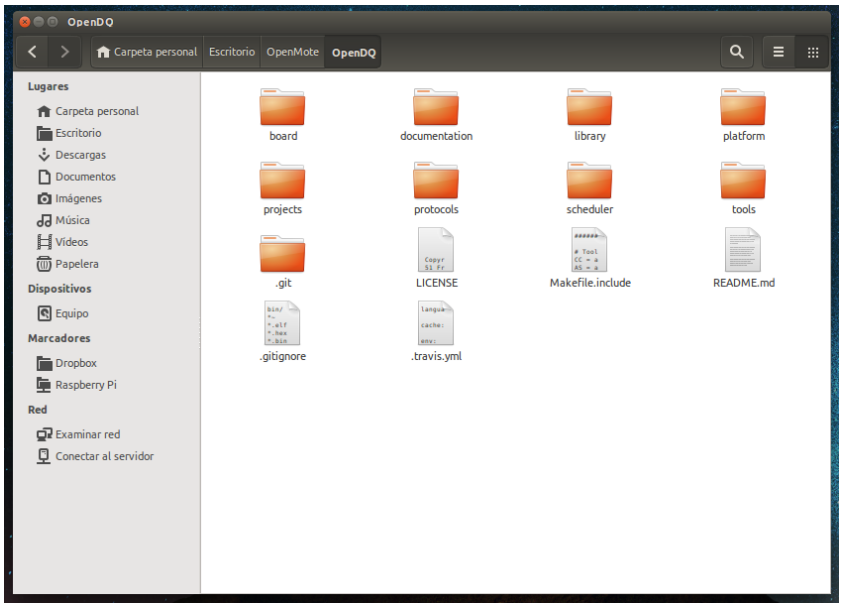


Figure 4.3: Main folder structure of the OpenDQ distribution.

library has been correctly installed, check that a file named *libcc*2538.*a* has been created in the */OpenDQ/platform/cc*2538 directory.

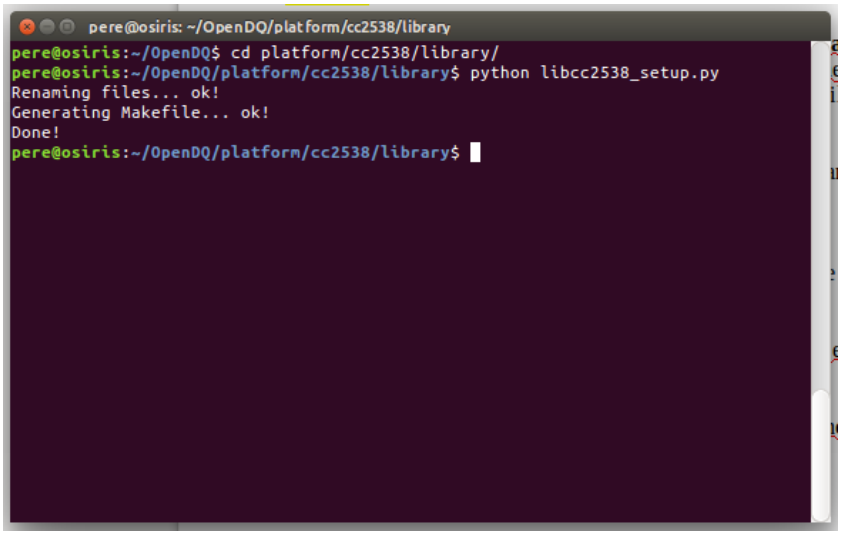


Figure 4.4: Generating the libcc2538 library.

4.4 Compile, load and debug the Node/Gateway projects

Once the Texas Instruments CC2538 firmware library (*libcc*2538.*a*) has been generated it is possible to compile the Node and Gateway projects. The mechanism to compile, load and debug the Node and Gateway firmware projects to the OpenMote-CC2538 board is based on a Makefile system that is deployed across all the project folder structure. Such mechanism does not need to be modified by the user unless new files need to be compiled.

The following steps assumes that an OpenMote-CC2538 is connected to an OpenBase which, in turn, is connected to a computer via a USB port (using the USB_FTDI port), as depicted in Figure 4.5. Please notice that the blue cable that connects the GND pin with the ON/SLEEP pin on the OpenBase are required to ensure that the OpenMote-CC2538 enters the bootload mode when the reset button is pressed. When the bootload mode is entered the 4 LEDs on the OpenMote-CC2538 become dim.

In order to compile, load and debug the Node or Gateway firmware projects navigate to the appropriate directory. For the Node project go to */OpenDQ/projects/Node/src*. For the Gateway project go to */OpenDQ/projects/Gateway/src*.

In order to compile the Node/Gateway project using the Makefile system execute the following command. If the process is successful, a screen similar to Figure 4.6 should be displayed.

```
make TARGET=cc2538 all
```

Once the Node/Gateway project has been build using the Makefile system a folder (bin) and three new files containing various versions of the firmware image (elf, hex and bin) will be created, as displayed in Figure 4.7. It is possible to clean the project

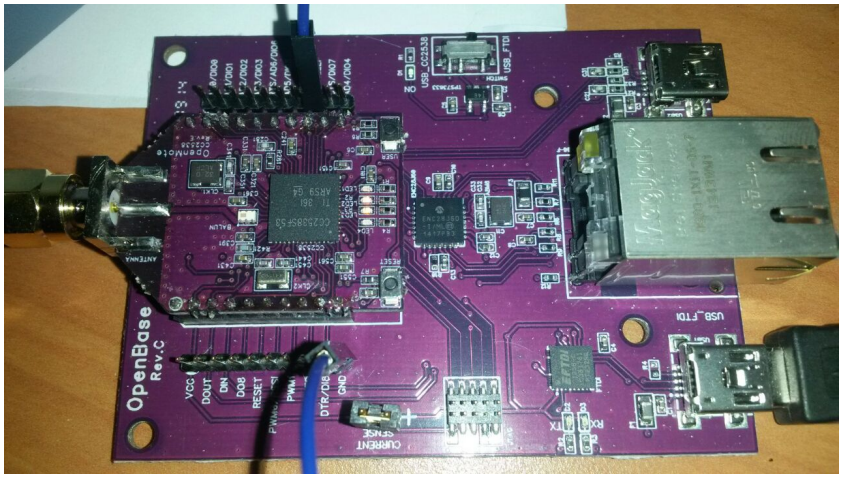


Figure 4.5: Connecting an OpenMote-CC2538 to a computer with an OpenBase.

```
pere@osiris: ~/OpenDQ/projects/Node/src
pere@osiris:~/OpenDQ/projects/Node/src$ make TARGET=cc2538 all
Building 'Node' project...
Compiling main.c...
Compiling ../../../kernel/scheduler.c...
Compiling ../../../library/src/crc16.c...
Compiling ../../../library/src/hdlc.c...
Compiling ../../../library/src/library.c...
Compiling ../../../library/src/packet_buffer.c...
Compiling ../../../library/src/serial.c...
Compiling ../../../library/src/virtual_timer.c...
Compiling ../../../platform/cc2538/cc2538_startup.c...
Compiling ../../../platform/cc2538/board.c...
Compiling ../../../platform/cc2538/bsp_timer.c...
Compiling ../../../platform/cc2538/cpu.c...
Compiling ../../../platform/cc2538/debug.c...
Compiling ../../../platform/cc2538/flash.c...
Compiling ../../../platform/cc2538/gpio.c...
Compiling ../../../platform/cc2538/ieee-addr.c...
Compiling ../../../platform/cc2538/leds.c...
Compiling ../../../platform/cc2538/radio.c...
Compiling ../../../platform/cc2538/random.c...
Compiling ../../../platform/cc2538/uart.c...
Compiling ../../../protocols/02a-mac/dq.c...
../../../platform/cc2538/radio.c: In function 'rf_error_interrupt':
../../../platform/cc2538/radio.c:459:14: warning: variable 'irq_error' set but n
ot used [-Wunused-but-set-variable]
Compiling ../../../protocols/02a-mac/fsa.c...
Compiling ../../../protocols/02a-mac/mac.c...
Compiling ../../../protocols/02a-mac/wor.c...
Linking 'Node'...
   text    data     bss     dec     hex filename
  21804    1024    4524   27352    6ad8 Node.elf
Building 'Node' done.
pere@osiris:~/OpenDQ/projects/Node/src$
```

Figure 4.6: Compiling the Node/Gateway project.

compilation by issuing the following command. After that, the folder and the firmware images will be removed.

```
make TARGET=cc2538 clean
```

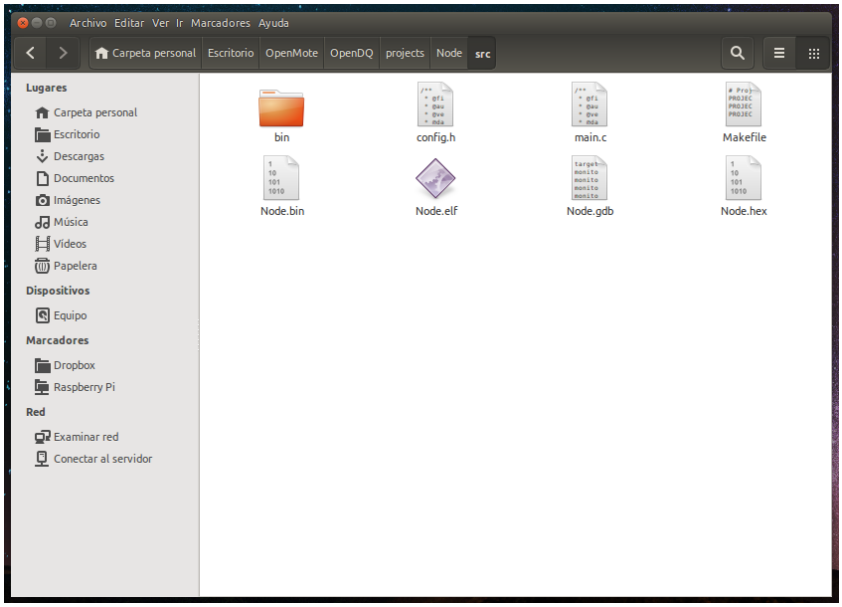


Figure 4.7: Folder containing the binaries for the Node project.

Once the Node/Gateway project has been build, there are two mechanism strategies to load the generated firmware image to the OpenMote-CC2538. The first mechanism is based on the CC2538 bootloader and uses the serial port on the OpenBase. The second mechanism is based on the JTAG and uses that JTAG port on the OpenBase. Both mechanisms are described next.

To load the firmware to the OpenMote-CC2538 using the bootloader mechanism execute the following command. Such command assumes that the OpenMote-CC2538 is in bootloader mode. If the execution is successful a screen similar to Figure 4.8 should appear. After that the OpenMote-CC2538 will start blinking the green LED periodically.

```
make TARGET=cc2538 bsl
```

To use the JTAG to load the Node/Gateway firmware image to the OpenMote-CC2538, first connect the Segger J-Link EDU probe to the OpenBase as depicted in Figure 4.9.

Once the Segger J-Link is connected to the OpenBase, open another terminal window. This new terminal will execute the Segger J-Link software that allows to connect the JTAG probe with the GDB client that is used to load the code. In the new terminal issue the following commands to go to the appropriate project directory and execute the Segger J-Link software. If connecting the JTAG to the OpenMote-CC2538 is successful, a screen similar to Figure 4.10 should appear.

```
cd ~/OpenDQ/projects/Application
make TARGET=cc2538 jlink
```

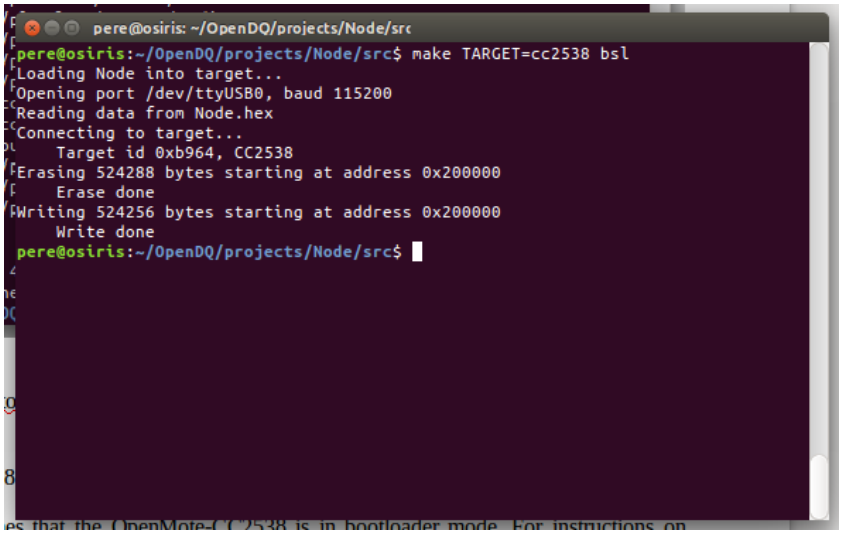


Figure 4.8: Loading the Node/Gateway firwmare using the OpenMote-CC2538 bootloader.

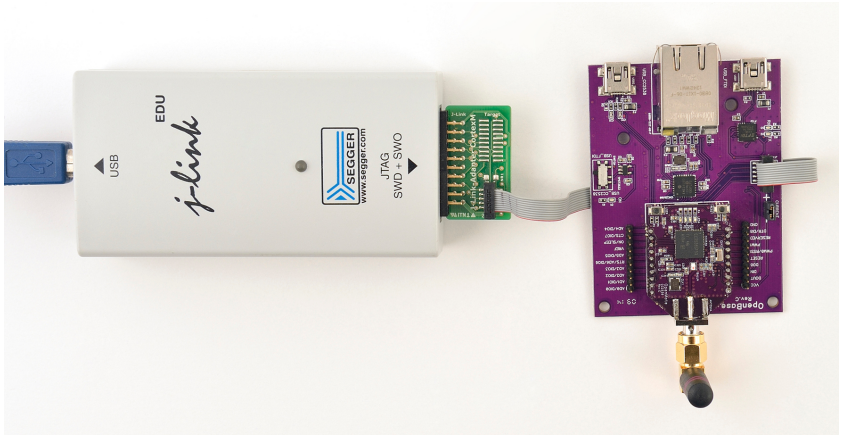


Figure 4.9: Connecting the Segger J-Link probe to the OpenBase.

On the original window, execute the following command to load the firmware image to the OpenMote-CC2538. If loading the firmware is successful, a screen similar to Figure 4.11 should appear. Once the firmware has been loaded, press the reset button on the OpenMote-CC2538 to start executing the code. After that the OpenMote-CC2538 will start blinking the green LED periodically.

```
make TARGET=cc2538 load
```

In addition to loading the code to the OpenMote-CC2538, with the JTAG mechanism it is also possible to debug execution of the code by either using GDB (command line)

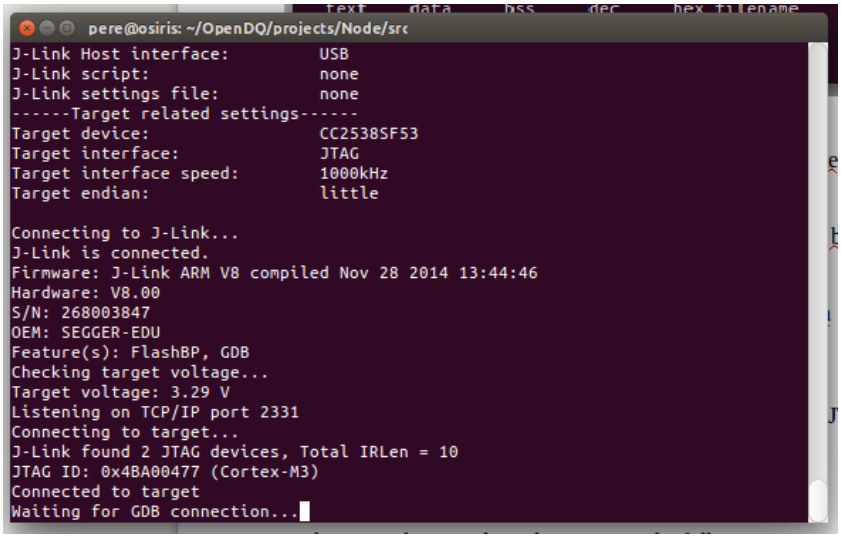


Figure 4.10: Connecting the JLink to the OpenMote-CC2538.

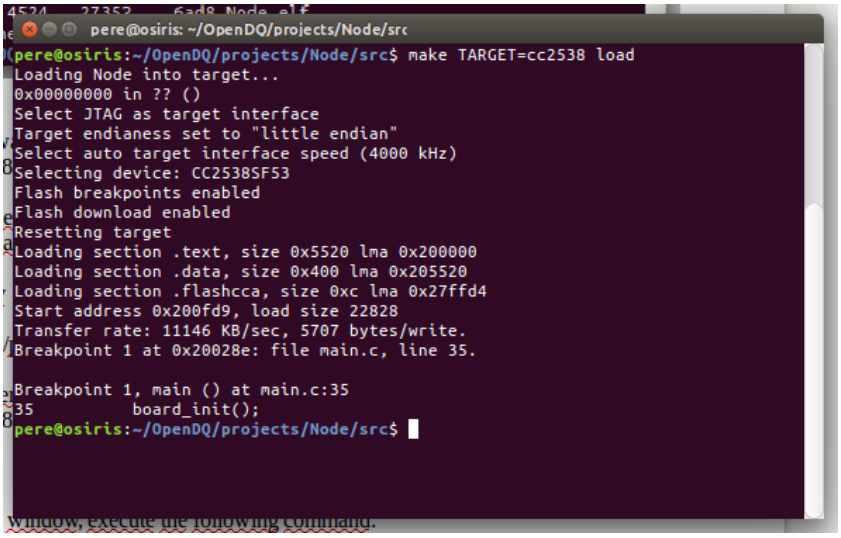


Figure 4.11: Loading the Node/Gateway project with JTAG.

or Nemiver (visual interface). In the original terminal window, execute the following commands to start either debugger. If GDB is used a screen similar to Figure 4.12 should appear, whereas if Nemiver is used a screen similar to Figure 4.13 should appear.

```
make TARGET=cc2538 debug
make TARGET=cc2538 nemiver
```

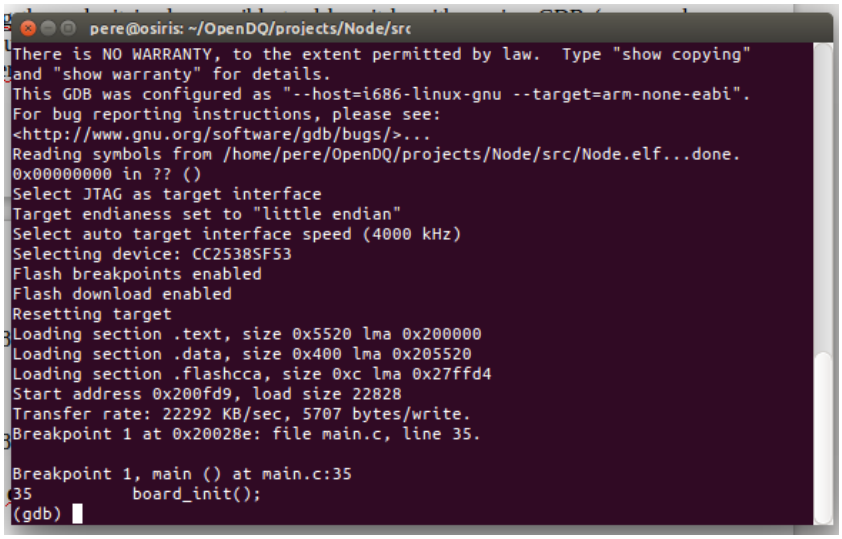


Figure 4.12: Debugging the Node/Gateway project with GDB.

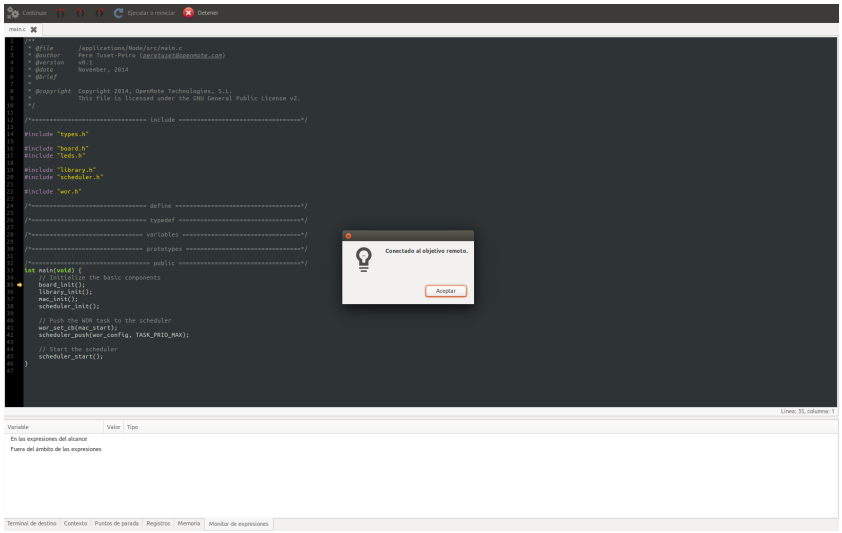


Figure 4.13: Debugging the Node/Gateway project with Nemiver.

4.5 Execute the Application project

To execute the Application project navigate to its directory and start the *OpenDQ_App.py* Python script by issuing the following commands. If the command is successfully executed, a screen similar to 4.14 will be displayed.

```
cd ~/OpenDQ/projects/Application
python OpenDQ_App.py
```

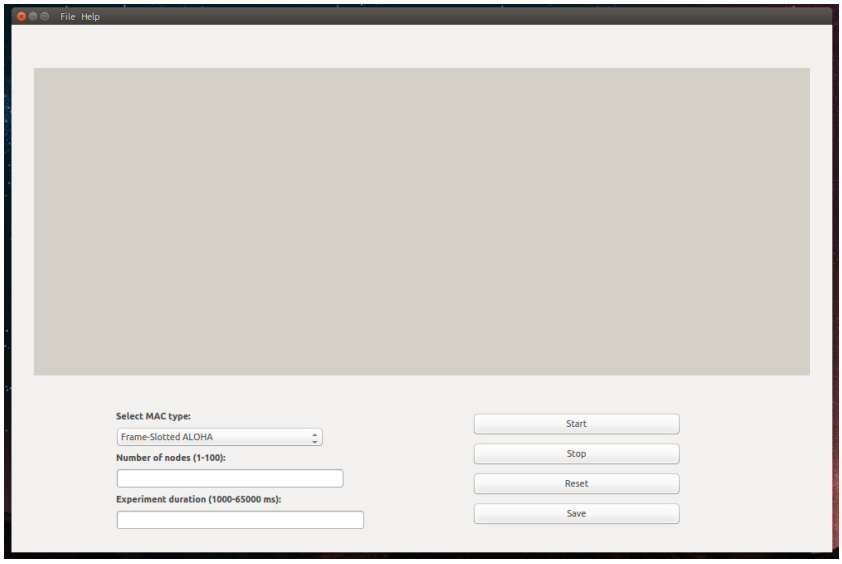


Figure 4.14: Opening the OpenDQ application.

The previous command assumes that an OpenMote-CC2538 board is connected to the computer via an OpenBase board using the FTDI port, as depicted in Figure 4.5. The command also assumes that the Linux kernel is able to detect and load the drivers of the FTDI FT232 chip to create a $/dev/ttyUSBX$ device and that the user has read and write permissions for the given device. If the $/dev/ttyUSBX$ is not detected or the user has no permissions to read and write from the given port the an image similar to Figure 4.15 will be displayed. If the user has no permissions to read and write from the given port try executing the command with *sudo* to temporarily gain such permissions.

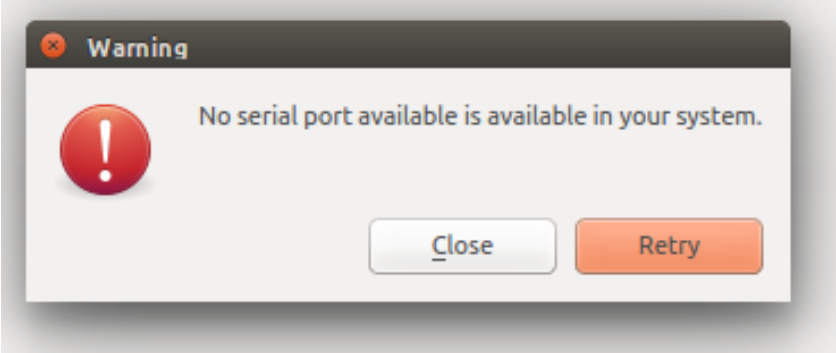


Figure 4.15: Serial port error while opening the OpenDQ application.

Chapter 5

Understanding OpenDQ

This chapter presents the firmware and software that compose the OpenDQ project. The firmware is the code written in C that execute on the OpenMote-CC2538 board, whereas the software is the code written in Python that execute on the target PC.

In general, the OpenDQ project is organized according to the following directory structure:

- **board**: Contains the board definitions for the OpenMote-CC2538 board.
- **documentation**: Contains the LaTeXsource code of the OpenDQ documentation.
- **library**: Contains the high-level firmware that runs on the OpenMote-CC2538 board.
- **platform**: Contains the low-level firmware that runs on the OpenMote-CC2538 board.
- **projects**: Contains the projects that implement the Node, the Gateway and the Application functionality.
- **protocols**: Contains the implementation of LPL, i.e., WOR, and the MAC layer protocols, i.e., FSA and DQ.
- **scheduler**: Contains the scheduler that runs on the OpenMote-CC2538 board and allows to execute tasks.
- **tools**: Contains the utilities that allow to flash the OpenMote-CC2538 board using the serial port.

In the following subsections a more detailed description of each folder in the is presented, including the files that compose it and its functionality. It is important to mention that the OpenDQ project uses a $Makefile$ system to build the firmware and load it to the OpenMote-CC2538 board. The $Makefile$ system is distributed among all directories and the base $Makefile$ file is stored in each project, i.e., Gateway or Node. Each project is responsible to include the remaining $Makefile$ include files, which are

spread across all directories to be able to generate the binary of each project. However, the user does not need to concern with the operation of the $Makefile$ system unless it needs to port the code to a new platform, as described in the previous section.

5.1 Board

The board folder contains the header files that define macro directives that associate generic pin and port names with particular pin and ports names for each existing platform. Currently, only the *openmote − cc*2538 board is defined. For example, in the *openmote − cc*2538.*h* file the LED_RED_PORT and LED_RED_PIN macros are associated with $GPIO\_C\_PORT$ and $GPIO\_PIN\_4$ respectively. There are other GPIO-dependent modules, i.e., the UART, that should also be moved to the *openmote − cc*2538.*h* file to ease portability of the code to other platforms based on the same SoC. However, at the moment of writing such modules have not been integrated yet in the *openmote − cc*2538.*h* header file.

5.2 Documentation

The *documentation* folder contains the source files, images and schematics of the OpenDQ documentation. The OpenDQ documentation is written in LaTeXfor the typesetting. Thus, to compile the documentation into a *pdf* file one needs to execute the $pdflatex documentation.tex$ command. It is possible that the command needs to be executed several times to ensure that the Figures, Tables and References are up to date.

5.3 Library

The *library* folder contains the high-level code that runs on the Texas Instruments CC2538 SoC and allows to control timing, manage packets and enable communications with the target computer via a serial port, among others. The library code is platform independent and is organized in two subdirectories, *inc* and *src*. The *inc* directory contains the header files that define the modules interface. The *src* directory contains the implementation of each module interface.

In particular, the *library* folder defines and implements the following components.

- **crc16**. Defined in *crc*16.*c* and *crc*16.*h* files. Provides a CRC (Cyclic Redundancy Check) implementation using the 16-bit CRC-CCITT protocol with a $0x1189$ polynomial. The implementation is based on a look-up table (256 x 8-bit entries, 256 bytes) stored in Flash and uses $0xFFFF$ as the initial seed. In order to compute the CRC, the code first needs to initialize the module by calling the *crc_init* function. After that, the code can call the *crc*16*_push* function for each byte to update the CRC state. At any iteration the code can also read the current value of the CRC state by calling the *crc*16*_get* function. After the computation of a message, the code can check the CRC state by calling the *crc*16*_check* function. If the CRC

check is successful the result will be 1. If the CRC check is unsuccessful the result will be 0.

- **hdlc**. Defined in *hdlc.c* and *hdlc.h* files. Provides an implementation of HDLC (High-Level Data Link Control) protocol. HDLC is a synchronous data-link layer protocol that provides frame structure to asynchronous protocols, i.e., UART (Universal Asynchronous Receiver-Transmitter). The HDLC includes a flag byte ($0x7F$) to indicate the start and end of frames. To ensure that the header and footer flags are not transmitted by error as part of the message, HDLC uses an escape byte ($0x7D$) and a escape mask ($0x20$). After the header flag, the HDLC includes a header that includes a command (1 byte) and the node address (2 bytes). After the payload the HDLC includes a footer with a 16-bit CRC. To compute the CRC, the HDLC module uses the CRC16 module described before. To use the HDLC module, the code first needs to initialize the module by calling the *hdlc_init* function. After that, the HDLC module can be used to create an HDLC frame to be received by using the *hdlc_open_rx*, *hdlc_put_rx* and *hdlc_close_rx* functions. The HDLC module can also be used to create an HDLC frame to be transmitted by using the *hdlc_open_tx*, *hdlc_put_tx* and *hdlc_close_tx* functions. In order to transmit and receive the HDLC frames the *serial* module is used.

- **packet_buffer**. Defined in *packet_buffer.c* and *packet_buffer.h*. Provides a statically allocated buffer of *packet_buffer_t* elements that is used by other modules, i.e., the MAC protocols, to transmit or receive packets to/from the radio transceiver. Currently, the *packet_buffer* variable can hold up to 16 packets, as defined by the $PACKET\_BUFFER\_SIZE$ macro. However, that can be changed at compile time to accommodate a larger number of *packet*s at the expense of increasing the RAM footprint. The *packet_buffer_t* structure contains the buffer where the bytes are allocated and its size, as well as several information fields, i.e., RSSI (Received Signal Strength Indicator), LQI (Link Quality Indicator) and CRC (Cyclic Redundancy Check). In order to use the *packet_buffer* module the code first need to call the *packet_buffer_init* function. After that, to transmit or receive a packet the code first needs to obtain an entry by calling the *packet_buffer_get*. This call returns a pointer to the *packet_buffer* entry to be used, which can then be used by reading or writing the appropriate fields. Once the entry has been used, either to transmit or receive, the code needs to release the entry by calling the *packet_buffer_release* function.

- **serial**. Defined in *serial.c* and *serial.h*. Provides an abstraction to transmit and receive messages to/from the computer via a serial port, i.e., UART. The serial module is implemented on top of the HDLC module, which ensures that messages being transmitted or received are correctly detected and decoded using the CRC module; if a message is corrupted it is discarded automatically. In order to use the *serial* module the code first need to call the *serial_init* function. After that, the code can use the *serial_push_message* to send messages to the computer through the serial port. The function *serial_register_mote2pc_cb*

can be used externally to register the execution of functions when a message has been transmitted. When a message is received from the serial port it is parsed by the *serial_parse_msg*. Tasks can also register to receive certain messages from the computer using the *serial_register_pc2mote_cb* function. Internally, the transmit and receive of a message is carried out byte by byte using the *serial_rx_init*, *serial_rx_byte*, *serial_rx_done*, *serial_tx_init*, *serial_tx_byte* and *serial_tx_done*. To hold the messages the *serial* module defines two 128-byte buffers, one to receive and the other to transmit, as well as a *serial_vars_t* structure where it stores all the data related to its operation, i.e., state, pointers to buffers, etc. Finally, the *serial* module also provides a function to check if the serial port is busy (*serial_busy*) and a function to reset it (*serial_reset*).

- **virtual_timer**. Defined in *virtual_timer.c* and *virtual_timer.h*. Provides a mechanism to define multiple timers that can run simultaneously but are multiplexed to a single physical timer. Currently, the *virtual_timer* implementation runs from the RTC (Real-Time Clock) on the OpenMote-CC2538 board. The RTC unit is clocked at 32.768 kHz and is 32 bit width, meaning that each *tick* is equivalent to 30.51 us and each *virtual_timer* can count up to 36 hours. The information related to each *virtual_timer*, i.e., status (stopped or running), type (periodic or one shot), total ticks, remaining ticks, callback and callback priority, is stored in a *virtual_timer_t* structure. To store all *virtual_timer*s and other relevant information, i.e., mode and timeout, the *virtual_timer* defines a *virtual_timer_vars* variable. The *virtual_timer*s are statically allocated in a buffer. Currently, the buffer variable can hold up to 16 *virtual_timer*s, as defined by the $VIRTUAL\_TIMER\_MAX\_TIMERS$ macro. However, that can be changed at compile time to accommodate a larger number of *virtual_timer*s at the expense of increasing the RAM footprint. In order to use the *virtual_timer* module the code first need to call the *virtual_timer_init* function. A *virtual_timer*s can then be started by calling the *virtual_timer_start* and passing the relevant information, i.e., type, ticks, callback and priority. The *virtual_timer_start* function returns a handler for the *virtual_timer* in use. When a *virtual_timer* expires the *virtual_timer* module pushes the task to the scheduler with the appropriate priority. Such approach is taken to ensure that code is not executed in an interrupt context, which could affect the timing behaviour of the MAC layer. Finally, *virtual_timer*s can be stopped during operation by calling the *virtual_timer_stop* function and passing in the *virtual_timer* handler that the *virtual_timer_start* function returns.

5.4 Platform

The *platform* folder contains the low-level code that runs on the Texas Instruments CC2538 SoC and allows to control its peripherals, i.e. GPIO, UART, timers and radio, among others. The platform code is platform dependent and is organized into two subdirectories, *inc* and *src*. The *inc* directory contains the header files that define the

modules interface, whereas the *src* directory contains the implementation of each module interface. In addition to providing the implementation of each module, the *src* directory also includes the source code of the Texas Instruments CC2538 Foundation Firmware library [21]. The library is used as a basis to implement the functionality of each module, i.e., GPIO, timers and UART, and needs to be compiled prior to building the Node or Gateway project.

The $platform$ folder contains the following modules.

- **board**. Contains the function that initializes the board by calling the other modules and provides a function to trigger the bootloader mode by pressing the user button.
- **bsp_timer**. Contains the functions to initialize and control the RTC (Real-Time Clock), which is driven at 32.768 kHz and is used by the $virtual\_timer$.
- **cpu**. Contains the functions to initialize the GPIOs and clocks to a default state. It initializes all the GPIOs to output low and configures the system clocks to 32.768 kHz for the RTC and 32 MHz for the CPU and the radio transceiver. It also contains the functions to reset the CPU, put the CPU in idle and sleep modes, and enable and disable the global interrupts.
- **debug**. Contains the functions to control the GPIO lines devoted to debug, which can be on, off or toggled.
- **flash**. Contains the functions to write to the Flash memory, which is used to trigger the bootloader mode when pressing the user button.
- **gpio**. Contains the functions to initialize and configure the GPIOs as input or output. It also provides functions to control the GPIOs (on, off or toggle) and to read its status (high or low). For input GPIOs it also provides the means to register interrupts.
- **ieee-addr**. Contains a function to read the IEEE EUI-64 address from Flash memory and a function that converts the IEEE EUI-64 address into a 16 bit addresses.
- **leds**. Contains the functions to initialize and control the LEDs (red, yellow, orange and green) on the board, which can be on, off or toggled.
- **radio**. Contains the functions to initialize and control the IEEE 802.15.4 radio transceiver. It provides functions to transmit and receive packets, as well as functions to set the channel, change the transmit power and read the RSSI.
- **random**. Contains the functions to initialize and generate pseudo-random numbers. The module is initially seeded by noise read from the radio transceiver and implements a Galois shift register with 16 taps to generate the pseudo-random number sequence.

- **uart**. Contains the functions to initialize and control the UART module. It provides functions to read and write bytes from the UART module. Since its operation is interrupt-driven, it also provides functions to register and unregister callbacks.

In addition to the modules that control the hardware, the *platform* folder also defines the following files.

- **cc2538_include.h**. Contains the headers of the Texas Instruments CC2538 Foundation Firmware library [21]. All modules that are implemented for the *cc*2538 platform need to include this file.
- **cc2538_linker.lds**. Contains the linker script for the *cc*2538 platform. The linker script describes the memory sections, i.e., RAM, Flash, etc., where the code and the variables are stored.
- **cc2538_startup.c**. Contains the start up code for the *cc*2538 platform. The start up code is responsible to initialize the board and to call the *main* function, which initializes the other modules and starts the application code.

5.4.1 Porting OpenDQ

In order to port the OpenDQ project to a new micro-controller based on the ARM Cortex-M architecture[1] there are only three steps required.

First, create a new directory in the *platform* folder. For example, to port the OpenDQ project to the *STM32L1* micro-controller a *stm32l1* directory needs to be created. Inside the *stm32l1* directory all the different modules need to be implemented according to their interface, as defined in the *inc* directory. That is, all the functions need to be implemented to provide the same functionality as the current *cc*2538 directory. Of special importance is the fact that the implementation of the *bsp_timer* module needs to provide a tick rate of 30.51 us. This is because the *virtual_timer* relies on the *bsp_timer* to provide accurate timing to all other modules, i.e., MAC protocols implementation.

Second, modify the *Makefile* system that is used to build the OpenDQ project in order to include the *stm32l1* platform. In particular, it is necessary to modify the *Makefile.include* under the *stm32l1* directory to ensure that all files are properly compiled and the libraries (if any) are linked correctly. The most important parts are defining the files that need to be compiled, the name of the linker script and the configuration for the tools related to the platform, i.e., JLinkGDBServer and bootloader script.

Third, create a file with the platform name in the *board* folder and define all the macro directives that associate generic pin and port names with particular pin and ports names of then new platform. This step is optional but highly recommended in order to maintain a clear separation between the platform specific and platform independent code.

[1] In order to port OpenDQ to other architectures, i.e., MSP430, it is also necessary to install the toolchain to compile and debug for that particular architecture.

After completing these three steps it should be possible to compile the Node or Gateway projects by issuing the following command $makeTARGET = stm32l1$ from the appropriate directory.

5.5 Projects

The *projects* folder is split into three subdirectories. First, the firmware that runs on the OpenMote-CC2538 and implements the gateway and node functionality is contained within the *Gateway* and *Node* directories. Second, the software that runs on the computer and implements the computer functionality to display the results is contained within the *Applications* directory. The firmware that runs on the OpenMote-CC2538 boards is written in C, whereas the the software that runs on the target computer, i.e. a PC or a Raspberry Pi, is written in Python. The following subsections provide an overview of the software architecture for the firmware and the software respectively.

- **Application**. The *Application* project contains the Python code that runs on a computer and controls the operation of the network, i.e., trigger the synchronization phase and the collect data from the nodes during the data collection phase. The application receives inputs using the GUI (Graphical User Interface) for the various input parameters, i.e., data collection duration and MAC protocol to execute. When the user triggers the data collection process by pressing the start button, the application sends a command to the gateway through the serial port. The command contains information regarding the duration of the data transmission phase and the MAC protocol to execute, i.e., FSA or DQ[2]. After that, the application waits to receive the packets send by the gateway through the serial port and calculates statistics based on the results, i.e., number of packets received for each node in the network during the data transmission phase. The end of the data transmission phase is determined by the reception of a command from the gateway using the serial interface. When such command is received the application stops collecting statistics and processes the results of the data collection phase to create a graphic that is displayed in the GUI.

- **Gateway**. The *Gateway* project implements all the functionality required for a gateway device to synchronize all the nodes that are present within its communication range and to provide feedback to the application regarding the packets received during the data transmission phase. Upon boot the gateway is in idle mode, waiting to receive a command from the application through the serial port to start the data collection phase. When the gateway receives such command it triggers the start of the synchronization phase using the appropriate parameters, i.e., duration and MAC protocol of the data transmission phase, which are passed as parameters from the application. During the synchronization phase the gateway

[2]In case FSA is selected, the application also determines the number of slots per frame. In case DQ is selected, the number of slots per frame is ignored.

transmits the wake-up packets that are used to synchronize the nodes. Once the synchronization phase finishes the gateway enters the data transmission phase. In the data transmission phase the gateway executes the configured MAC protocol, i.e., FSA or DQ, for the configured amount of time. For each packet received the gateway processes it and sends the relevant information to the application through the serial port. Once the duration of the data transmission phase is elapsed the gateway sends a command to the application to indicate such event. After that, the gateway resets itself and waits for a new trigger to start the synchronization and data transmission phases.

- **Node**. The *Node* project implements all the functionality required for a node device to wake-up using the WOR mechanism during the synchronization phase and to transmit data packets to the gateway using the appropriate MAC protocol, e.g., DQ or FSA, during the data transmission phase. Upon boot the node enters the WOR mode, where it periodically turns on the radio transceiver and tries to receive a wake-up packet from the gateway. When a wake-up packet from the coordinator is successfully received the node parses the information contained in the packet, which contains information regarding the time to start the data transmission phase and the MAC protocol to execute. When the data transmission phase starts the node transmits packets using the appropriate MAC protocol, i.e. FSA or DQ. During the data transmission phase the node transmits as much data packets as allowed by the MAC protocol in order to create a saturation condition. Depending on the MAC protocol being executed, i.e. FSA or DQ, the node includes information regarding the state of the MAC layer on the data packet. For example, in DQ the node includes information regarding the length of the CRQ and the DTQ. Such information is then forwarded by the gateway to the Application on the computer via the serial interface. The information is then used to create the statistics that are displayed after the data collection phase terminates. It is important to remark that nodes are not aware of the duration of the data transmission phase. Instead, the nodes reset themselves upon loosing synchronization with the gateway.

5.6 Protocols

The *protocols* folder contains the implementation of the synchronization mechanism, i.e., WOR, and the MAC protocols for the data transmission phase, i.e., DQ and FSA. Given the importance of such protocols in the OpenDQ operation, the description can be found in Sectionr̃efsec:06-operation.

5.7 Scheduler

The *scheduler* folder contains the system scheduler in two files named *scheduler.c* and *scheduler.h*. The system scheduler is responsible of queuing and executing tasks from

other modules. Tasks in the scheduler are represented by the *task_cb_t* structure, which is a pointer to the regular C function that has to be executed by the scheduler. To execute tasks the scheduler implements a priority-based cooperative mechanism with three priorities, i.e., minimum, medium and maximum, according to the *task_prio_t* enumerate.

The scheduler defines two data structures that are used to define and store the tasks to be executed.

- *task_list_t*. Defines the attributes of a task to be executed by the scheduler. The structure contains a pointer to the function to be executed (*task_cb_t*), the task priority (*task_prio_t*) and a pointer to the next task.
- *scheduler_vars_t*. Contains an array of *task_list_t* variables where the tasks to be executed by the scheduler are stored and also contains a pointer to the first task to be executed. Currently the *task_list_t* is limited to 16 tasks according to the $TASK\_BUFFER\_SIZE$ macro, but this value can be changed at compile time to accommodate more tasks at the expense of increasing memory footprint.

The scheduler contains three public functions, as described next.

- *scheduler_init*. Initializes the *scheduler_t* data structure. Needs to be executed as part of the initialization process, i.e., *main*.
- *scheduler_start*. Pushes the *scheduler_toggle_led*[3] task to the queue and starts the scheduling loop, which never returns. The scheduler loop will execute tasks as fast as they become available and will put the CPU in sleep mode in case there is no task to execute.
- *scheduler_push*. Adds a task to the scheduling queue. The tasks is defined as a callback to the function to be executed (*task_cb_t*) and its priority (*task_prio_t*). Notice that since the scheduler runs as a main loop with interrupts enabled, the execution of a task can be stopped to service an interrupt. To ensure that tasks can be safely added from an interrupt context the *scheduler_push* function temporarily disables and re-enables interrupts using the *cpu_disable_interrupts* and *cpu_enable_interrupts* functions.

5.8 Tools

The *tools* folder contains the $cc2538 - bsl$ Python script that is used to flash the OpenMote-CC2538 using the bootloader mode through the serial port instead of the JTAG. The $cc2538 - bsl$ script is called from the $Makefile$ system with the appropriate parameters by executing $makeTARGET = cc2538bsl$. The only pre-requisite is that

[3]The *scheduler_toggle_led* task blinks the green LED of the OpenMote-CC2538 board periodically using the $virtual_timer$ module to indicate that the system is running normally. By default the LED is off for 32704 ticks (998 ms) and on for 64 ticks (2 ms).

the OpenMote-CC2538 has to be in bootload mode prior to executing the command, otherwise the board will not be flashed. The benefits of using the bootloader mode to program the OpenMote-CC2538 board is that the Segger J-Link is not required. The disadvantage of using the bootloader mode to program the OpenMote-CC2538 is that real-time debugging of the code and variables is not possible. However, it is possible to debug the code execution flow without the Segger J-Link interface by using the serial port to output debugging messages. It is important to take into account that using the serial port to output debugging messages alters the timing of code execution, making it an unsuitable approach to debug real-time code.

Chapter 6

OpenDQ operation

This chapter describes the operation of the different protocols that are used in the OpenDQ project to enable distributed data collection using low-power wireless technologies. As a remainder, the operation of a data collection protocol is split into two different phases: synchronization and data transmission. In the synchronization phase the gateway uses a PS (Preamble Sampling) mechanism to wake-up and synchronize all the nodes that are within its communication range. In the data transmission phase the nodes that are synchronized use a MAC (Medium Access Control) protocol, i.e., FSA (Frame Slotted ALOHA) or DQ (Distributed Queuing), to transmit their data packets to the gateway. Finally, the gateway forwards the received data packets to the computer, where they are processed, stored and represented.

6.1 Overview

To enable distributed data collection using low-power wireless technologies, the OpenDQ project defines the physical and the data-link layers of the OSI (Open Systems Interconnect) model.

At the physical layer the OpenDQ project uses the IEEE 802.15.4 standard [3], which operates at the 2.45 GHz ISM (Industrial, Scientific and Medic) band, which is available world-wide. At the 2.45 GHz band (2400-2483 MHz, 83 MHz), the IEEE 802.15.4 standard defines 16 channels (from 11 to 26) with a 2 MHz bandwidth and a channel separation of 5 MHz. The physical layer uses an OQPSK (Offset Quadrature Phase-Shift Keying) modulation with DSSS (Direct Sequence Spread Spectrum), which provides an effective data rate of 250 kbps, as depicted in Figure 6.1. For a complete description of the IEEE 802.15.4 standard please refer to the standard document [3].

At the physical layer the IEEE 802.15.4 standard also defines the PPDU (Physical Protocol Data Unit). The PPDU is composed of a SHR (Synchronization Header, 5 bytes), which is composed of a PS (Preamble Sequence, 4 bytes) and a SFD (Start-of-Frame Delimiter, 1 byte), a PHR (Physical Header, 1 byte) and the PSDU (Physical Service Data Unit), as depicted in Figure 6.2. The PHR contains the number of payload bytes in the MPDU (MAC Protocol Data Unit), which is limited to 127 bytes and

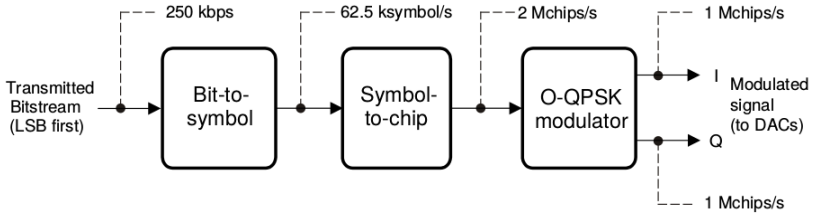


Figure 6.1: IEEE 802.15.4 physical layer modulation scheme. [Source: Texas Instruments]

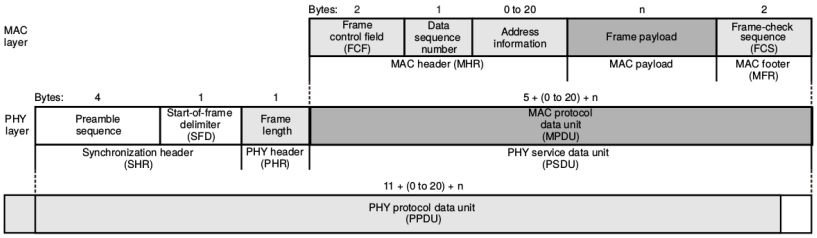


Figure 6.2: Schematic view of the IEEE 802.15.4 frame format. [Source: Texas Instruments]

contains a 2-byte FCS (Frame-Check Sequence) that enables to check the integrity of the data payload.

At the data-link layer, OpenDQ provides the implementation of the data-link layer protocols that are required for the synchronization and the data transmit phases of data collection. In particular, it provides an implementation of a packet-based PS mechanism that is used by the gateway during the synchronization phase to wake-up all the nodes that are within its communication range. It also provides the implementation of two MAC protocols, i.e., FSA and DQ, that are used by nodes to transmit their data packets to the gateway during the data transmission phase. Despite having a different purpose, there are two common aspects of all data-link layer protocols implemented in the OpenDQ project, as described next.

First, the data-link layer protocols are implemented as finite-state machines, where the future state of the system depends on the current state, as well as time progress and external events. To implement the finite-state machine of each protocol the states are defined and each state is implemented as a C function. Typically, each state is split into *init* and *done* functions to indicate its entry and exit points, but states can also have sub-states. Each C function carries all the tasks that are required in the current state, i.e., prepare a packet and turn on the radio transceiver to transmit it, and prepares the system to evolve to the next state, i.e., obtain a packet handler and turn on the radio transceiver to receive a packet. Thus, the advance from a state to the next state can be triggered by synchronous or asynchronous events. For example, a timer that expires would be a synchronous event, whereas a packet being received would be an

asynchronous event.

Second, the data-link layer protocols require time synchronization between the gateway and the nodes that are within its communication range to be able to wake-up and exchange data. In order to achieve such distributed synchronization a common time unit is defined. The tick is defined as 1/32.768 Hz, which translates into 30.51 us, and is derived from the RTC (Real-Time Clock) that each device has. One important aspect to take into account for the protocol implementation is that the RTC drift. That is, due to the physical construction limitations of crystals and environmental effects, i.e., temperature, the tick rate of RTCs change along time. For instance, the RTC clocks that are used on the OpenMote-CC2538 board have a nominal drift rate of 20 ppm. This implies that two devices of the network can drift up to ±40 ppm, one crystal going fast and the other going slow. This has several implication in the design of the data-link layer protocols since mechanisms to compensate for such drift are required to ensure proper operation of the network.

The operation of WOR for the synchronization phase, as well FSA and DQ for the data transmission phase, are explained in detail in the next subsections.

6.2 Wake-On Radio

WOR (Wake-On Radio), depicted in Figure 6.3, is the module that implements the packet-based PS (Preamble Sampling) mechanism that is used in the synchronization phase to allow the gateway wake-up all the nodes that are within its communication range and provide the information required for the data transmission phase. In particular, the WOR mechanism determines the time and channel at which nodes are expected to wake up to start the data transmission phase, as well as the MAC layer that they are expected to execute, i.e., DQ or FSA, to transmit their data packets to the gateway.

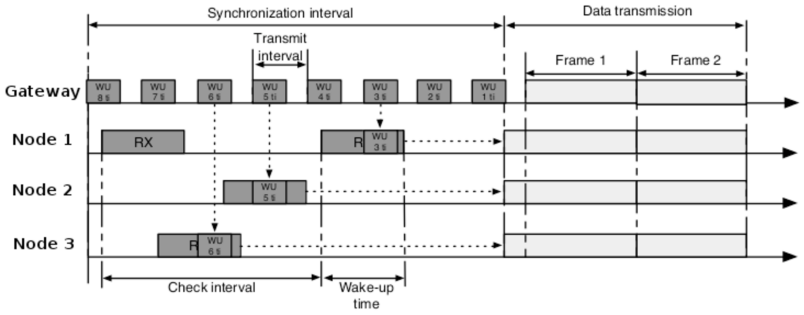


Figure 6.3: Wake-On Radio operation.

The WOR mechanism that is implemented in the OpenDQ project can be found in the *wor.c* and *wor.h* files under the *protocols* folder. The implementation of WOR contains two different flavours, one for the gateway and for the node, which are selected at compile time using a macro defined in the project configuration file ($config.h$).

PARAMETER	VALUE
WOR_TX_DURATION	65536 ticks
WOR_TX_PERIOD	32 ticks
WOR_TX_PACKETS	2048 packets
WOR_RX_PERIOD	32768 ticks
WOR_RX_DURATION	64 ticks

Table 6.1: Wake-On Radio configuration parameters.

The WOR implementation for the gateway is responsible to transmit the wake-up packets that enable the nodes and wake-up at the appropriate time to start the data collection phase. The main parameters of the WOR gateway implementation are the transmit duration, the transmit period and the number of packets, as summarized in Table 6.2. The transmit duration indicates how long does the gateway keep sending WOR packets in order to wake-up nodes, whereas the transmit period indicates how often does the gateway transmit such WOR packets. The number of packets transmitted in a period results from dividing the transmit duration by the transmit period. By default the transmit duration is 65536 ticks (2 seconds) and the transmit period is 32 ticks (976,32 us), which means that the gateway will transmit a total of 2048 packets, one every 976,32 us. Considering the physical layer data rate, i.e., 250 kbps, and length of a WOR packet, i.e., 13 bytes including the SHR, it takes 416 us to transmit a WOR packet. Thus, the transmit duty cycle of the gateway when transmitting the synchronization packets is around 50%.

In turn, the WOR implementation for the node is responsible to periodically turn on the radio transceiver to try to receive a WOR packet from the gateway to wake-up. The main parameters of the node implementation are the receive period and the receive duration, as summarized in Table 6.2. The receive period indicates how often does the system turn on the radio to try to receive a WOR packet from the gateway, whereas the receive duration indicates how long does the system keep the radio on to receive a WOR packet from the gateway. By default the receive period is 32768 ticks (1 second) and the receive duration is 64 ticks (1952 us), which translates into a radio duty cycle of 0.2%. It is important to notice that such values have been selected to ensure that any node will receive at least one WOR packet from the gateway regardless of its wake-up duty cycle. The values selected for the gateway and node are a good compromise between wake-up delay and energy consumption. However, it is possible to adjust such values to improve wake-up delay at the expense of increasing the energy consumption of the nodes.

In addition to the timing parameters, the WOR implementation also determines the WOR packet structure. Upon being triggered by the computer, the WOR packet is sent by the gateway periodically and contains the information required to synchronize the nodes. The WOR packet is then received by the node to configure the parameters of the data transmission phase. The WOR packet structure is defined in *wor_packet_t* (in *wor.c*) and has a payload of 5 bytes with the following structure. However, notice that the payload of the *wor_packet* (5 bytes) will include the SHR (5 bytes) and the PHR

(1 byte) at the beginning and the FCS (2 bytes) at the end, meaning that the whole *wor_packet* will have 13 bytes when transmitted by the radio transceiver. Also, notice that the *wor_packet_t* is *packed* to ensure that the compiler generates the appropriate data structure, i.e., without padding, since the structure is meant to be sent and received using the radio transceiver.

- MAC_TYPE [1 byte]. Indicates the MAC type, i.e., DQ or FSA, to be executed by Nodes in the data transmission phase.
- MAC_PACKET [1 byte]. Indicates the MAC packet, i.e. WOR, ARP, FBP, DATA and ACK. For WOR the packet type is always WOR.
- MAC_TIME [2 bytes]. Indicates the time (in ticks) at which the nodes are expected to wake up to begin the data-transmit phase.
- $MAC\_CHANNEL$ [1 byte]. Indicates the channel at which the nodes are expected to operate at the beginning of the data-transmit phase.

The WOR module also defines the wor_vars_t data structure to store the variables that are used during WOR execution for network synchronization. The implementation of both the gateway and the node require different variables during execution, but at the moment of writing this document both use the same data structure.

- *tx_duration*. Used by the gateway to keep track of the WOR remaining duration until the beginning of the data transmission phase.
- *tx_period*. Used by the gateway to keep track of the WOR period.
- *tx_packets*. Used by the gateway to keep track of the remaining packets to be transmitted. When the remaining packet is zero the gateway starts the data transmission phase.
- *rx_period*. Used by the node to store the period of the WOR cycle.
- *rx_duration*. Used by the node to store the duration of each idle listening cycle.
- *wor_cb*. Used by the node to store the callback to the appropriate MAC for the data transmission phase, i.e., FSA or DQ.
- *wor_channel*. Used by the node to store the channel for the data transmission phase.

The WOR module also defines the following public functions that are common for both the gateway and the node implementation.

- *wor_init*. Initializes the wor_vars data structure. Called from the MAC module *mac_init* function during the initialization phase.

- *wor_set_cb*. Registers a callback to the function that needs to be executed when the WOR finalizes and the data transmission phase begins. Typically it points to *mac_start*, which is responsible to start the appropriate MAC protocol based on the information received in the WOR packet.

- *wor_cancel_cb*. De-registers a callback to the function that needs to be executed when the WOR finalizes. Currently the function is not used.

The WOR module also defines the following private functions for the gateway implementation. As it is possible to observe, the gateway will transmit a given number of WOR packets (as determined by the parameters described above) and after that it will begin the execution of the appropriate MAC protocol, i.e., FSA or DQ.

- *wor_config*. Prepares to start the execution of the packet-based PS mechanism for the gateway by setting the appropriate values in the *wor_vars* and *mac_vars* data structures and pushing the *wor_start* task to the scheduler.

- *wor_start*. Prepares the *wor_packet* that will be transmitted to wake-up the nodes. It also turns on the radio transceiver, configures the radio channel and the transmit callbacks, and enables radio interrupts. After that, it copies the *wor_packet* to the radio transceiver and begins the transmission. Finally, it sets a time-out using the virtual timer module to indicate that the current transmit period has finished.

- *wor_tx_init*. Callback from the radio transceiver to indicate that a packet is now being transmitted.

- *wor_tx_done*. Callback from the radio transceiver to indicate that a packet has been successfully transmitted.

- *wor_done*. Callback from the virtual timer module that indicates that the current transmit period has finished. It turns off the radio and clears the callbacks, releases the packet buffer, and updates the WOR status. Finally, it decides the next action to take. If no more WOR packets need to be transmitted, it sets a time-out using the virtual timer module to start the data transmission phase with the appropriate MAC protocol. If more WOR packets need to be transmitted, it sets a time-out using the virtual timer module to start a new period using the *wor_start* function.

Similarly, the WOR module defines the following private functions for the node implementation. As it is possible to observe, the node will try to receive a WOR packet indefinitely by cycling between the *wor_start* and *wor_timeout* functions at the appropriate times. When a WOR packet is successfully received, the cycle will break and the node will progress to the *wor_done* function, where it will prepare for the execution of the MAC protocol according to the parameters set by the WOR packet transmitted by the gateway.

- *wor_config*. Prepares to start the execution of the packet-based PS mechanism for the node by setting the appropriate values in the *wor_vars* data structure and pushing the *wor_start* task to the scheduler.

- *wor_start*. Starts the execution of the packet-based PS mechanism. It registers the radio callbacks, obtains a *packet_buffer* entry, turns on the radio in receive mode and sets a time-out for the *wor_timeout* function using the *virtual_timer* module in case no WOR packet is received.

- *wor_rx_init*. Callback from the radio transceiver module that indicates that a packet has started to be received.

- *wor_rx_done*. Callback from the radio transceiver module that indicates that a packet has finished being received. If a packet is successfully received it processes the packet. If it is a valid WOR packet it updates the *mac_vars* data structure with the MAC protocol, timing and channel information. Finally, it turns off the radio transceiver.

- *wor_timeout*. Callback from the virtual timer module that indicates that the current interval has expired. If a valid WOR packet has been received it prepares for the beginning of the data transmission phase by pushing the *wor_done* task to the *virtual_timer* to be executed at the appropriate time. If a valid WOR packet has not been received it prepares for the beginning of a new cycle by pushing the *wor_start* task to the *virtual_timer* to be executed at the appropriate time.

- *wor_done*. Releases the *packet_buffer* entry obtained in the *wor_start* function and schedules the start of the data transmission phase by pushing the *wor_vars.wor_cb* task to the *virtual_timer* to be executed at the appropriate time. The *wor_vars.wor_cb* will contain a function to the appropriate entry point for the MAC protocol to be executed in the data transmission phase, i.e., *fsa_start* or *dq_start*.

6.3 Medium Access Control

The MAC (Medium Access Control) is a module that provides the basic functionality to control the operation of the various MAC protocols implemented in the OpenDQ project for the data transmission phase, i.e., FSA and DQ. The implementation of the MAC module contains a single flavour that is common for both the gateway and for the node. However, the configuration of the MAC module to be used in the data transmission phase is filled in differently depending on the device type, i.e., gateway or node. In case the device is a gateway, the configuration is filled in through the serial port. Contrarily, if the device is a node, the configuration is filled in through the WOR mechanism during the synchronization phase.

The MAC module that is implemented in the OpenDQ project can be found in the *mac.c* and *mac.h* files under the *protocols* folder. In particular, the MAC module provides the data structures, functions and variables to set the time at which the data transmission phase is expected to begin and also the protocol that the device is expected to execute.

The MAC module defines the following data types that are common for all MAC protocols. This data types are used instead of the raw values to enforce type checking in function calls.

- *mac_address_t* [*uint*16_*t*]. Used by the gateway or the nodes to determine the device that originates or has to receive a given packet. The 16-bit address is derived from the EUI-64 address by taking the least two significant bytes. There are also two values that are reserved: invalid ($0x0000$) and broadcast ($0xFFFF$) as defined in *mac_addr_type_t*.

- *mac_seq_number_t* [*uint*16_*t*]. Incremental sequence number that is used by nodes to ensure that they are synchronized with the gateway. If a node receives a packet that does not contain a valid sequence number it can assume that it has lost synchronization with the gateway and trigger the appropriate action, i.e., reset the value or reset itself.

- *mac_time_t* [*uint*16_*t*]. Time at which a certain event is expected to happen at the MAC layer. Each LSB is equivalent to one tick of the RTC (30.51 us). Since the counter is only 16-bit wide it wraps around after 65535 ticks or 2 seconds.

- *mac_channel_t* [*uint*8_*t*]. Channel of the IEEE 802.15.4 physical layer, i.e., between 11 and 26, at which the data transmission phase will begin. After that the channel of each transmission can change if channel hopping is enabled.

- *mac_slots_t* [*uint*8_*t*]. Number of slots. For FSA the number of slots indicates the number of slots per frame, whereas for DQ the number of slots per frame can be used to indicate the number of ARP slots in the access sub-period.

The MAC module defines the following data structures that are common for all MAC protocols.

- *mac_state_t*. Indicates if the MAC is synchronized or unsynchronized. The MAC can be in two possible states, unsynchronized ($0x00$) or synchronized ($0x01$). If the MAC is synchronized the synchronization LED (orange) will be turned on. IF the MAC is not synchronized the synchronization LED (orange) will be turned off.

- *mac_type_t*. Defines the MAC types that are supported. Currently the protocols implemented are FSA ($0x01$) and DQ ($0x02$). The gateway receives this value as a parameter from the computer and uses it in the synchronization phase using WOR to announce the type of MAC protocol that will be executed in the data transmission phase. Nodes receive this value as a parameter during the synchronization phase using WOR and use it to start the appropriate MAC protocol in the data transmission phase.

- *mac_packet_t*. Indicates the type of packet that is being transmitted or that has to be received. Currently there are five types of packets which are used by the different protocols. The packet types are WOR ($0x01$), ARP ($0x02$), FBP ($0x03$),

DATA ($0x04$) and ACK ($0x05$). The WOR implementation only uses the WOR packet type. The FSA implementation uses the FBP, DATA and ACK. Finally, the DQ implementation uses the ARP, FBP and DATA.

- *mac_packet_state_t*. Indicates the status of a packet. Currently a packet, either a DATA or ACK packet, can be in three different states, empty ($0x00$), error ($0x01$) or success ($0x02$). Empty is used when the energy level in the channel is below a given threshold and no packet has been detected. Error is used when the energy level in the channel is above a given threshold but a packet has not been successfully received. Finally, success indicates that a packet has been successfully received.

- *mac_rssi_state_t*. Indicate the RSSI (Received Signal Strength Indicator) status of a packet. Currently, the RSSI status of a packet can be in three states, none ($0x00$), below ($0x01$) or above ($0x02$). None indicates that the RSSI was not sensed, below indicates that is below a given threshold and, finally, above indicates that it is above a given threshold. The RSSI threshold is defined in each implementation in a macro named $FSA\_RSSI\_THRESHOLD$ or $DQ\_RSSI\_THRESHOLD$ depending on the protocol.

The MAC module defines the following public functions. This function are meant to be called from outside the MAC module itself, mainly from the WOR module to set the appropriate configuration parameters to be executed during the data transmission phase or by the MAC protocols, either FSA or DQ, to obtain their operating parameters during the execution of the data transmission phase.

- *mac_init*. Initializes the mac_vars data structure and calls the initialization functions of all the MAC protocols, e.g. FSA, DQ and WOR. Needs to be executed as part of the initialization process, i.e., *main*.

- *mac_start*. Initializes the radio transceiver to the default channel and starts executing the appropriate protocol for the data transmission phase. It is executed as a callback after the synchronization sub-period using the WOR mechanism finishes. To decide which MAC protocol to execute it uses the *mac_type* parameter stored in the mac_vars structure and pushes a new task to the scheduler.

- *mac_set_type*. Sets the MAC protocol that will be executed in the data transmission phase.

- *mac_set_channel*. Sets the MAC channel that the radio transceiver will used in the beginning of the data transmission phase.

- *mac_set_time*. Sets the time (in ticks) at which the data transmission phase will begin.

- *mac_toggle_synchronized*. Sets the current synchronization state of the MAC protocol, either synchronized or unsynchronized.

- *mac_next_channel*. Returns the next channel at which the MAC protocol, i.e., FSA or DQ, is expected to operate in. This function allows the gateway to generate a new channel. Currently, this function always returns the default channel (26) meaning that channel hopping is disabled.

To save the parameters required to operate, the MAC layer instantiates a variable *mac_vars* of type *mac_vars_t*. The *mac_vars* variable is independent from the MAC protocol being executed and stores the following information.

- *mac_type*. Type of MAC protocol that will be executed in the data transmission phase.
- *mac_packet*. Type of MAC packet that will be transmitted or that has been received.
- *mac_state*. State of the MAC layer, either synchronized or unsynchronized.
- *mac_time*. Time at which the MAC layer is expected to start the data transmission phase.
- *mac_slots*. Number of slots of the data transmission phase. For FSA the number of slots indicates the number of slots per frame, whereas for DQ the number of slots per frame can be used to indicate the number of ARP slots in the access sub-period.
- *mac_channel*. Channel that the radio transceiver will use in the beginning of the data transmission phase.
- *queue_mac_rx*. A pointer to the *packet_buffer_t* structure that is currently being used by the MAC layer to receive a packet from the radio.
- *queue_max_tx*. A pointer to the *packet_buffer_t* structure that is currently being used by the MAC layer to transmit a packet to the radio.

6.4 Frame-Slotted ALOHA

The FSA (Frame Slotted ALOHA) module implements a MAC protocol that allows the nodes present in the network transmit their data packets to the gateway during the data transmission phase. FSA operates in a time-slotted approach; time is divided into fixed-length frames and each frame contains a given number of fixed-length slots, which allow the nodes to transmit a data packet to the gateway and the gateway to transmit back an acknowledgement packet to the ndoes to indicate if the data packet was received successfully. In order to transmit their data packets to the gateway, nodes select one of the slots of the current frame at random and transmit their data packet. After transmitting their data packet nodes listen to receive an acknowledgement from the gateway, In turn, the gateway listens to each slot and sends an acknowledgement packet

after a packet is received. Based on such operation there are three possible outcomes for each slot[1]. If no node selects a given slot the result will be empty. If a single node selects a given slot the result will be success. Finally, if two or more nodes select a given slot the result will be a collision.

The FSA protocol that is implemented in the OpenDQ can be found in the $fsa.c$ and $fsa.h$ files under the $protocols$ folder. The implementation of FSA contains two different flavours, one for the gateway and one for the node, which are selected at compile time using a macro defined in the project configuration file ($config.h$).

The FSA implementation for the gateway is responsible to transmit the FBP at the start of every frame. The FBP contains information regarding the configuration of the frame, i.e., the number of slots per frame. Then, for every slot of the current frame the gateway tries to receive a data packet from any of the nodes that are within its communication range and, afterwards, transmits an acknowledgement packet to indicate if the data packet has been successfully received. The state of the data packet in a slot can be empty, success or collision depending on the number of nodes that have transmitted their data packet in the same slot. To determine the state of the data packet in each slot the gateway uses an approach based on the RSSI (Received Signal Strength Indicator) and the CRC (Cyclic Redundancy Check). In the middle of the data packet the gateway samples the RSSI present in the channel to determine if there is an ongoing transmission. At the end of the data packet the gateway checks the CRC of the packet. If the RSSI is below a threshold, the gateway determines that the state is empty, e.g., no node transmitted a data packet. If the RSSI is above a threshold and the CRC is valid the gateway determines that the state is success, e.g., a node transmitted a data packet and was received successfully. Finally, if the RSSI is above a threshold and the CRC is not valid the gateway determines that the state is collision, e.g. two or more nodes transmitted in the same slot. Such information is then embedded in the acknowledgement packet to allow nodes determine the status of their data packet. In addition to receiving the data packet and transmitting the acknowledgement packet, the gateway is also responsible to send information regarding the status of each data packet to the computer using the serial port.

In turn, the FSA implementation for the node is responsible to receive the FBP from the gateway at the start of every frame. When the node receives the FBP it configures the relevant parameters, i.e., the number of slots per frame, and decides in which slot it will transmit its data packet. The selection of the slot to transmit its data packet is done at random following a uniform distribution, where all the slots have the same probability. After selecting the slot, the node remains idle until the selected slot is about to begin. When the selected slot is about to begin, the node prepares and transmits the data packet and then prepares to receive an acknowledgement packet from the gateway. If the data packet has been received successfully by the gateway, the node removes the

[1]This assumes that all the nodes are received by the gateway with the same signal strength and there are no interferences from other networks. For example, if nodes are received with different signal strength then the node that is received with a higher signal strength could be received successfully even other nodes are transmitting. In such event the node that is successfully received has *captured the channel*. Alternatively, if a network is operating nearby it can cause interference to a node transmitting.

data packet from its transmit queue. Otherwise, the node keeps the packet in its transmit queue to retry in the next frame. After processing the acknowledgement packet the node then remains idle until the beginning of the next frame, which is indicated by the FBP transmitted by the gateway.

As described earlier, a frame is composed by a FBP followed by a LIFS period and a given number of slots. In turn, each slot is composed of a data packet followed by a SIFS period and an acknowledgement packet followed by anther SIFS period. The duration of each part is summarized in Table 6.4. Given this values, the shortest frame in FSA is composed of a single slot and would have a duration of 280 ticks (8822 us). Under such circumstances, the FSA implementation is able to transmit up to 117 data packets per second. Considering that each packet is 127 bytes long and the data rate at the physical layer is 250 kbps, the net throughput would be 118.8 kbps (47.5%). Increasing the number of slots per frame improves the number of data packets per second that can be transmitted since the overhead of the FBP is distributed among all data packets. With 10 slots per frame the FSA implementation is able to transmit up to 147 data packets per second, which gives a net throughput of 149.4 kbps (59.7%). However, such results are theoretical since collisions between different nodes, i.e., two nodes selecting the same slot to transmit its data packet, will reduce the effective throughput. Considering that FSA has a theoretical throughput of 36.8% for the optimal case, i.e., same number of slots per frame as nodes present in the network, the effective throughput for 10 slots per frame would only be 55 kbps (22%).

PARAMETER	**VALUE**
$FSA\_FBP\_DURATION$	32 ticks
$FSA\_DATA\_DURATION$	152 ticks
$FSA\_ACK\_DURATION$	32 ticks
$FSA\_SIFS\_DURATION$	16 ticks
$FSA\_LIFS\_DURATION$	32 ticks
$FSA\_SLOT\_DURATION$	216 ticks

Table 6.2: Frame Slotted ALOHA configuration parameters.

There are three additional configuration parameters for the FSA implementation, as described in the next listing.

- $FSA\_RADIO\_CHANNEL$. Indicates the channel at which FSA operates by default if channel hopping is not enabled. By default, channel hopping is disabled at the MAC layer (see Section 6.3) and the FSA implementation operates in channel 26.

- $FSA\_RSSI\_THRESHOLD$. Indicates the energy that is required to be present in the channel for the gateway to distinguish between successful, collision and empty data slots transmitted by Nodes. A data slot is considered successful if enough energy is present in the channel and the CRC of the data packet correct. A data slot is considered empty if the energy present in the channel is below the threshold.

Finally, a data slot is considered collision if the energy present in the channel is above the threshold but a data packet has not been received or it has been received with a wrong CRC. By default, the FSA implementation uses a threshold that is set to -85 dBm and changing its value has several implications on the FSA performance. Setting a value higher, i.e., 80 dBm, will reduce the maximum communication range because nodes will be considered as interference. Contrarily, setting a value lower, i.e., -90 dBm, will increase the maximum communication range but lead to false collision detection.

- FSA_UNSYNC_ERRORS. Indicates the number of de-synchronization events that a node can tolerate. A de-synchronization event occurs when a node does not receive the expected packet type or the packet is not successfully received. By default, the FSA implementation tolerates up to 16 de-synchronization events before the node resets itself.

In addition to the configuration parameters, the FSA module also defines the data structures that are used to store information related to the operation of the protocol during the data transmission phase. In particular the FSA module defines the fsa_vars_t and the fsa_debug_t data structures, as defined next.

The fsa_vars_t data structure is used to keep the variables that are used during data transmission phase using FSA by both the gateway and the node. In either case, i.e., gateway or node, there are some variables that remain unused. Thus, it would be interesting to separate this variables in to two different data structures, one for the gateway and one for the node. This would allow to reduce memory footprint and to provide a clearer approach to understand what each variable does in each case.

- *mac_packet* [*mac_packet_t*]. Type of packet, i.e., FBP, DATA, ACK.
- *mac_address* [*mac_address_t*]. Local address of the gateway or node.
- *seq_number* [*mac_seq_number_t*]. Sequence number of the packet.
- *next_channel* [*mac_channel_t*]. Channel at which FSA will operate for the next frame.
- *packet_success* [*uint8_t*]. Last packet was received successfully or not.
- *slot_total* [*uint8_t*]. Total number of slots in the frame.
- *slot_count* [*uint8_t*]. Current slot in the frame.
- *slot_selected* [*uint8_t*]. Slot that was selected in the current frame.
- *slot_remaining* [*uint8_t*]. Number of slots remaining in the current frame.
- *data_state* [*mac_data_state_t*]. State of the received packet.
- *data_address* [*mac_address_t*]. Source of the received packet.

- *rssi_status* [*mac_rssi_t*]. RSSI status of the received packet.
- *data_rssi* [*int8_t*]. RSSI value of the received packet.
- *data_rssi_threshold* [*int8_t*]. RSSI value to consider packets as collided.
- *total_packets* [*uint8_t*]. Number of transmitted packets.
- *unsync_error* [*uint8_t*]. Number of unsynchronization errors.

The *fsa_debug_t* data structure is used to keep debug information related to the FSA operation. The *fsa_debug_t* is packed and is transmitted through the serial line to the computer at the end of every data packet. Such information is used by the application to create the statistics of the data transmission phase.

- *mac_type* [*uint8_t*]. MAC type (FSA).
- *slot_count* [*uint8_te*]. Current slot in the frame.
- *data_state* [*uint8_t*]. State of the received packet, i.e., empty, successful or collision.
- *data_address* [*uint16_t*]. Source address of the received packet.
- *rssi_status* [*int8_t*]. RSSI state of the received packet, i.e., above or below.
- *fsa_total* [*uint8_t*]. Number of transmitted packets by the node.

In addition to the data structures described above, the FSA implementation also determines the structure and contents for the packets that are exchanged between the gateway and the nodes during the data transmission phase using the FSA protocol. In particular, it defines the *fsa_fbp_t* for the feedback packet, the *fsa_data_t* for the data packet and the *fsa_ack_t* for the acknowledgement packet.

The *fsa_fbp_t* packet is transmitted by the gateway at the beginning of each frame. The *fsa_fbp_t* packet is 10 bytes long and contains the information described in the following listing. Considering the SHR (5 bytes) at the physical layer and the FCS (2 bytes) at the data-link layer, the *fsa_ack_t* packet is 13 bytes long and takes 0.416 ms (13.64 ticks) to be completely transmitted.

- *mac_type* [*uint8_t*]. MAC type (FSA).
- *mac_packet* [*uint8_t*]. Packet type (FBP).
- *source* [*uint16_t*]. Address of the gateway that transmits the feedback packet.
- *destination* [*uint16_t*]. Broadcast ($0xFFFF$) since all nodes within communication range are expected to receive feedback packets.
- *seq_number* [*uint16_t*]. Sequence number of the current frame to allow nodes ensure that they are correctly synchronized.

- *next_channel* [*uint*8_*t*]. Channel at which the gateway will operate for the next frame. This would enable to do channel hopping to combat multi-path propagation and external interference.

- *slot_count* [*uint*8_*t*] The number of slots per frame in the next frame. This would enable to dynamically adjust the number of slots per frame based on collision estimation algorithms.

The fsa_data_t packet is transmitted by all the nodes that have selected the current slot to transmit their data packet to the gateway. The fsa_data_t packet is 125 bytes and long contains the information described in the following listing. Considering the SHR (5 bytes) at the physical layer and the FCS (2 bytes) at the data-link layer, the fsa_ack_t packet is 128 bytes long and takes 4.096 ms (134.25 ticks) to be completely transmitted.

- *mac_type* [*uint*8_*t*]. MAC type (FSA).

- *mac_packet* [*uint*8_*t*]. Packet type (DATA).

- *source* [*uint*16_*t*]. Address of the node transmitting the data packet.

- *destination* [*uint*16_*t*]. Address of the gateway that will receive the data packet.

- fsa_total [*uint*8_*t*]. Number of packets that the each node has transmitted so far.

- *data* [*uint*8_*t*]. Remaining bytes of the packet payload (118 bytes) that can be used to transport data from the node to the gateway.

The fsa_ack_t packet is transmitted by the gateway after the fsa_data_t packet to confirm the successful reception of the data packet. The fsa_data_t packet is 7 bytes long and contains the information described in the following listing. Considering the SHR (5 bytes) at the physical layer and the FCS (2 bytes) at the data-link layer, the fsa_ack_t packet is 10 bytes long and takes 0.320 ms (10.48 ticks) to be completely transmitted.

- *mac_type* [*uint*8_*t*]. MAC type (FSA).

- *mac_packet* [*uint*8_*t*]. Packet type (ACK).

- *source* [*uint*16_*t*]. Address of the gateway that transmit the acknowledgement packet.

- *destination* [*uint*16_*t*]. Address of the node that receives the packet to indicate which node has been successful.

- *data_state* [*uint*8_*t*]. Determines if the data packet of the current slot has been successfully received by the gateway.

In addition to the packet types described above, the FSA module also defines the following public functions that are common for both the gateway and the node implementation.

- *fsa_init*. Initializes the *fsa_vars* and the *fsa_debug_serial* variables and sets the local address of the node by calling the *ieee_addr_get_eui*16 function. It is called from the MAC module *mac_init* function during the initialization phase.
- *fsa_start*. Starts the execution of FSA at the beginning of the data transmission phase by pushing the *fsa_fbp_init* function to the scheduler.

The FSA module also defines the following private functions that are unique for the gateway implementation. The compilation of such functions is protected by the *MAC_DEVICE* macro, that is, the functions are only compiled if the *MAC_DEVICE* is equal to *MAC_GATEWAY* as defined in the *config.h* file in each project.

- *fsa_fbp_init*. Executed at the beginning of each frame. Resets the *fsa_vars* variables using the *fsa_vars_reset* function and prepares the payload of the feedback packet to be transmitted. After that, it registers the radio transmit interrupts, puts the feedback packet to the radio transceiver and transmits it. Finally, it creates a timeout for the duration of the feedback packet using the *virtual_timer* module that will execute the *fsa_fbp_done* function when it expires.
- *fsa_fbp_tx_init*. Callback from the radio transceiver to indicate that the feedback packet is now being transmitted.
- *fsa_fbp_tx_done*. Callback from the radio transceiver to indicate that the feedback packet has been successfully transmitted.
- *fsa_fbp_done*. Executed when the timeout configured from the *fsa_fbp_init* functions expires. Puts the radio transceiver back to idle mode and releases the entry from the *packet_buffer* module. Then, it increases the sequence number and computes the ticks remaining until the start of the next data packet. Finally, it creates a timeout using the *virtual_timer* module that will execute the *fsa_data_init* function when it expires.
- *fsa_data_init*. Executed when the timeout configured from *fsa_fbp_done* expires. Puts the radio transceiver in receive mode to get a data packet from the nodes. Finally, it computes the ticks remaining until the middle of data packet and creates a timeout using the *virtual_timer* module that will execute the *fsa_data_rssi* function when it expires.
- *fsa_data_rx_init*. Callback from the radio transceiver to indicate that a packet is now being received.
- *fsa_data_rx_rssi*. Callback from the *virtual_timer* module to indicate that it is in the middle of the data packet. It then samples the RSSI value and computes the

ticks remaining until the end of data packet. Finally, it creates a timeout using the *virtual_timer* module that will execute the fsa_data_done function when it expires.

- $fsa\_data\_rx\_done$. Callback from the radio transceiver to indicate that a packet has been successfully received. It obtains an entry from the $packet\_buffer$ module and copies the payload from the radio transceiver.

- fsa_data_done. Executed when the timeout configured from fsa_fbp_rssi expires. Puts the radio transceiver back to idle mode and releases the entry from the $packet\_buffer$ module. It processes the received packet and determines if it was empty, successful or collision using the CRC and the RSSI sample. With such information it fills in the fsa_debug variable using the fsa_vars_log function and transmits it using the through the serial port using the *serial_push_message* function. Finally, it creates a timeout using the *virtual_timer* module that will execute the fsa_ack_init function when it expires.

- fsa_ack_init. Executed when the timeout configured from fsa_data_done expires. It obtains an entry from $packet\_buffer$ module and prepares the payload of the acknowledgement packet to be transmitted based on the information available on the fsa_vars variable. After that, it registers the radio transmit interrupts, puts the acknowledgement packet to the radio transceiver and transmits it. Finally, it creates a timeout for the duration of the acknowledgement packet using the *virtual_timer* module that will execute the fsa_ack_done function when it expires.

- $fsa\_ack\_tx\_init$. Callback from the radio transceiver to indicate that the acknowledgement packet is now being transmitted.

- $fsa\_ack\_tx\_done$. Callback from the radio transceiver to indicate that the acknowledgement packet has been successfully transmitted.

- fsa_ack_done. Executed when the timeout configured from fsa_ack_init expires. Puts the radio transceiver back to idle mode, resets the fsa_vars variables related to the data packet and releases the entry from the $packet\_buffer$ module. Afterwards, it updates the number of slots that remain in the current frame. If there are more slots remaining in the current frame, it creates a timeout using *virtual_timer* module that will start a new slot by executing the fsa_data_init function when it expires. If there aren't more slots remaining in the current frame, it creates a timeout using the *virtual_timer* module that will start a new frame by executing the fsa_fbp_init function when it expires.

- fsa_vars_reset. Resets the local fsa_vars variables.

- fsa_vars_log. Copies the fsa_vars variables that will be transmitted through the serial port to the fsa_debug_serial variable.

Similarly, the FSA module defines the following private functions that are unique for the node implementation. The compilation of such functions is protected by the MAC_DEVICE macro, that is, the functions are only compiled if the MAC_DEVICE is equal to MAC_NODE as defined in the *config.h* file in each project.

- fsa_fbp_init. Executed at the beginning of each frame. Resets the fsa_vars variables using the fsa_vars_reset function and prepares to receive a feedback packet from the gateway. Finally, it computes the maximum number of ticks remaining until the end of the feedback packet and creates a timeout using the *virtual_timer* module that will execute the fsa_fbp_done function.

- $fsa\_fbp\_rx\_init$. Callback from the radio transceiver to indicate that a packet is now being received. It stops the current timeout and creates a new timeout using the exact number of ticks remaining until the expected end of the feedback packet. The function assumes that it is indeed a feedback packet, so it is possible to know the exact number of ticks that remain.

- $fsa\_fbp\_rx\_done$. Callback from the radio transceiver to indicate that a packet has been received. It obtains an entry from the $packet\_buffer$ module and copies the payload from the radio transceiver. It then checks that if payload is valid and contains a feedback packet. If so, it updates the FSA variables with the fsa_vars_update function.

- fsa_fbp_done. Executed when the timeout configured from either the fsa_fbp_init or the $fsa\_fbp\_rx\_init$ functions expires. It puts the radio transceiver back to idle mode and releases the entry from the $packet\_buffer$ module. Then, it checks if the packet is a valid feedback packet. If so, it indicates that it is synchronized with the gateway and selects a slot of the current frame to transmit its data packet to the gateway. Afterwards, it programs the beginning of the data packet transmission by creating a timeout using the *virtual_timer* module that will execute the fsa_data_init function when it expires. Otherwise, it indicates it has lost synchronization and programs the beginning of a new frame by creating a timeout using the *virtual_timer* module that will execute the fsa_fbp_init function when it expires.

- fsa_data_init. Executed when the timeout configured from fsa_fbp_done expires. It obtains an entry from the $packet\_buffer$ module and fills it in with the data payload. After that, it loads the packet to the radio transceiver and puts the radio transceiver in transmit mode. Finally, it creates a timeout for the duration of the data packet using the *virtual_timer* module that will execute the fsa_data_done function when it expires.

- $fsa\_data\_tx\_init$. Callback from the radio transceiver to indicate that the data packet is now being transmitted.

- $fsa\_data\_tx\_done$. Callback from the radio transceiver to indicate that the data packet has been successfully transmitted.

- fsa_data_done. Executed when the timeout configured from fsa_data_init expires. Puts the radio transceiver back to idle mode and releases the entry from the $packet\_buffer$ module. Finally, it computes the ticks remaining until the start of the acknowledgement and creates a timeout using the $virtual\_timer$ module that will execute the fsa_ack_init function when it expires.
- fsa_ack_init. Executed when the timeout configured from fsa_data_done expires. Puts the radio transceiver in receive mode to get an acknowledgement packet from the gateway. Finally, it creates a timeout for the duration of the acknowledgement using the $virtual\_timer$ module that will execute the fsa_ack_done function when it expires.
- $fsa\_ack\_rx\_init$. Callback from the radio transceiver to indicate that a packet is now being received.
- $fsa\_ack\_rx\_done$. Callback from the radio transceiver to indicate that a packet has been received. It obtains an entry from the $packet\_buffer$ module and copies the payload from the radio transceiver. It then checks if the payload is valid and contains an acknowledgement packet.
- fsa_ack_done. Executed when the timeout configured from fsa_ack_init expires. Puts the radio transceiver back to idle mode and releases the entry from the$packet_buffer$ module. Finally, it computes the ticks remaining until the start of the next frame and creates a timeout using the $virtual\_timer$ module that will execute the fsa_fbp_init function to start the new frame.
- fsa_vars_init. Initializes the contents of the fsa_vars variable.
- fsa_vars_reset. Resets the contents of the fsa_vars variable.
- fsa_vars_update. Updates the contents of the fsa_vars variable with the information on the feedback packet received from the gateway.

Finally, it is important to notice that there is an upper limit in the number of slots per frame that are currently supported in the FSA implementation due to the drift of the RTC, which as a nominal value of of ± 20 ppm. This means that every second each node can drift up to 20 us and the gateway and any given node can drift up to ± 40 us per second, one going fast and the other going slow. Since the synchronization point is the FBP transmitted by the gateway and there is a guard time of 16 ticks (488 us), the nodes can remain synchronized for around 12 seconds before loosing synchronization if they listen for the whole duration of the guard time. Hence, given the fact that the duration of a slot in a frame is 152 ticks (4637 us), the maximum number of slots per frame to ensure that nodes can keep synchronized with the gateway is 2587. However, this is the ideal result and in practice a limit of 255 slots per frame cannot be surpassed because the variable to store the number of slots per frame is only 8 bits . The tests that have been conducted with the current implementation ensure that the synchronization works for up to 100 slots per frame.

6.5 Distributed Queuing

The DQ (Distributed Queuing) module implements a MAC protocol that allows the nodes present in the network transmit their data packets to the gateway during the data transmission phase. Similarly to FSA, DQ operates in a time-slotted approach; time is divided into fixed-length frames. However, in DQ the notion of slot does not exist. Instead, the notion of sub-period is defined with in a frame. In particular, three sub-periods are defined within a DQ frame: access request sub-period, data transmission sub-period and feedback information sub-period, as depicted In Figure 6.4[2]. In order to transmit their data packets to the gateway, nodes apply a set of rules based on the current status of two queues, i.e., the CRQ (Collision Resolution Queue) and the DTQ (Data Transmit Queue), and also their relative positions within each queue. These rules can be classified into access request rules and data transmit rules. The former apply to the CRQ, whereas the latter apply to the DTQ. For example, an access request rule states that a node can only transmit an access request packet if the CRQ is empty or if the node holds the first position in the CRQ. Similarly, a data transmit rule states that only the node at the head of the DTQ is entitled to transmit a data packet in the data transmission sub-period of the current frame. This set of rules ensure that no collisions can occur during the data transmit sub-period because only the node at the head of the DTQ is allowed to transmit its data packet. In addition, this set of rules also ensures that collisions in the access request sub-period are solved in logarithmic time and that the assignment of network resources to nodes is fair in terms of average bandwidth. Such properties are because access to the CRQ is blocking, i.e., nodes have to wait until the CRQ is empty to request access to the network again.

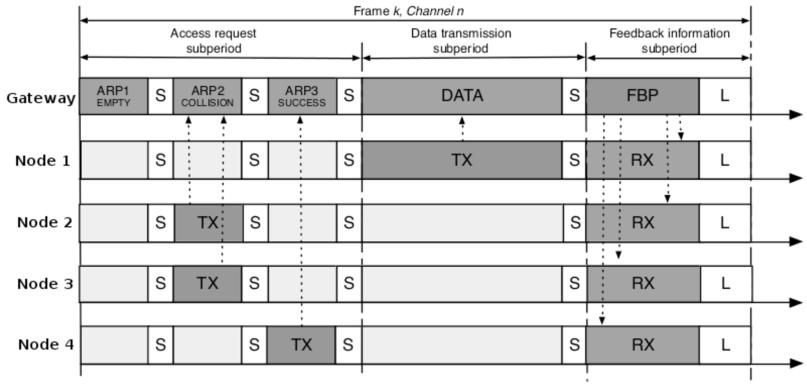


Figure 6.4: Distributed Queue operation.

The DQ protocol is implemented in the *dq.c* and *dq.h* files in the *protocols* folder. The implementation of DQ contains two different flavours, one for the gateway and one

[2]It is important to notice that the actual implementation of DQ is not as depicted. The order of the different sub-periods is changed in order to ease implementation. The actual order of the sub-periods is feedback information sub-period, access request sub-period and data transmission sub-period.

for the node, which are selected at compile time using a macro defined in the project configuration file (*config.h*), i.e., gateway or node.

The DQ implementation for the gateway is responsible to transmit the FBP in the feedback sub-period at the start of every frame. The FBP contains the information regarding the global status of the different queues, i.e., CRQ and DTQ, and also the outcome of the access request and the data transmission sub-periods of the previous frame. Such information is used by all nodes in the network to determine which action to take in the current frame, i.e., remain idle, transmit in the access request sub-period or transmit in the data transmission sub-period. Then, for every slot in the access request sub-period the gateway listens to nodes transmitting an access request packet to gain access to the network. The state of the access request packet in a slot of the access request sub-period can be empty, success or collision depending on the number of nodes that have transmitted in the same slot. To determine the state of the access request packet in each slot the gateway uses the same approach as in FSA, i.e., check the RSSI (Received Signal Strength Indicator) and the CRC (Cyclic Redundancy Check)[3]. Contrarily, the state of the data packet in the data transmission sub-period will be *always*[4] success. Based on the information from the slots in the access request sub-period and the data packet in the data transmission sub-period, the gateway updates the global status of the queues and creates the feedback packet that will be transmitted to the nodes in the feedback sub-period of the next frame. The process is repeated subsequently until all nodes in the network have transmitted their data packet, i.e., both the CRQ and the DTQ are empty and the status of all slots in the access request sub-period is empty.

In turn, the DQ implementation for the node is responsible to receive the FBP from the gateway in the feedback sub-period at the beginning of every frame. When a node receives the FBP from the gateway it updates the outcome of the access request and data transmission sub-periods of the previous frame and checks and updates the current status of the queues, i.e., CRQ and DTQ. Based on the global and local state of each queue the node will determine which action to take in the access request and the data transmission sub-periods of the current frame. A node can take three main actions in the access request and data transmission sub-period of each frame. If the CRQ is empty or the node holds the first position in the CRQ it will transmit an access request packet in a random slot in the access request sub-period of the current frame. The selection of the slot to transmit its access request packet in the access request sub-period is done at random following a uniform distribution, where all the slots have the same probability. Contrarily, if the node holds the first position in the DTQ the node will transmit a

[3]If the RSSI is below a threshold, the gateway determines that the state of the slot is empty, e.g., no node transmitted an access request packet. If the RSSI is above a threshold and the CRC is valid the gateway determines that the state of the access request packet is success. Finally, if the RSSI is above a threshold and the CRC is not valid the gateway determines that the state is collision, e.g., two or more nodes transmitted an access request packet in the same slot.

[4]The data transmit rules in DQ ensure that only one node in the network will be at the head of the DTQ at any given time. Thus, the node cannot collide with any other node during the transmission of its data packet. However, it is possible that the result of the transmission is not success due to multi-path propagation or external interference.

PARAMETER	VALUE
$DQ\_FBP\_DURATION$	44 ticks
$DQ\_ARP\_DURATION$	24 ticks
$DQ\_DATA\_DURATION$	152 ticks
$DQ\_SIFS\_DURATION$	16 ticks
$DQ\_LIFS\_DURATION$	32 ticks
$DQ\_SLOT\_DURATION$	364 ticks
DQ_ARP_COUNT	3 ARPs

Table 6.3: Distributed queuing configuration parameters.

data packet in the data transmission sub-period of the current frame. Otherwise, i.e., the node holds none or any other position in either queue, it will remain idle until the beginning of the next frame, where it will repeat the process. After the access request and the data transmission sub-periods, the node listens to the feedback packet at the beginning of the next frame to repeat the process until it succeeds in transmitting its data packet to the gateway.

The DQ implementation has 6 main configuration parameters that determine the time length of each frame, as summarized in Table 6.5. The first five parameters determine the length of each sub-period in a slot, i.e., $DQ\_FBP\_DURATION$, whereas the last parameter ($DQ\_SLOT\_DURATION$) represents the length of a whole DQ frame. As depicted in Figure 6.4, a DQ slot is composed of 3 ARPs, 1 DATA, 1 FBP, 1 LIFS and 4 SIFS which, considering the current duration of a frame (364 ticks), gives an effective frame rate of 90 frames/second. Taking into account that each frame contains 127 bytes of payload, the net throughput is 91.44 kbps. Thus, taking into account that the physical layer operates at 250 kbps, DQ provides a physical layer efficiency of 36%. Compared to FSA, as presented in Section6.4, the physical layer efficiency of DQ is smaller, i.e., 36% of DQ with respect to 59.7% of FSA (with 10 slots per frame). However, the effective throughput of FSA is smaller due to collisions caused by two nodes selecting the same slot to transmit their data packet, i.e., 36% of DQ with respect to 22% of FSA. Moreover, such result assumes that the gateway can know the number of nodes present in the network in advance and, thus, it can adjust the number of slots per frame to the optimal, i.e., same number of slots per frame as nodes in the network. However, such knowledge of the network cannot be assumed in most real environments and, thus, the real performance of FSA will be even worse. Finally, it is important to remark that the DQ efficiency can be increased by reducing the length of ARPs, as well as the length of LIFS and SIFS periods. However, changing the length of such parameters poses stringent requirements to maintain synchronization among nodes and, thus, it is not recommended.

Similarly to FSA, the DQ implementation has three additional configuration parameters, as described in the next listing.

- DQ_UNSYNC_ERRORS. Indicates the number of de-synchronization events that a node can tolerate. A de-synchronization event occurs when a node does

not receive the expected packet type or the packet is not successfully received. By default, the DQ implementation tolerates up to 16 de-synchronization events before the node resets itself.

- *DQ_RSSI_THRESHOLD*. Indicates the energy that is required to be present in the channel for the gateway to distinguish between successful, collision and empty data slots transmitted by Nodes. A data slot is considered successful if enough energy is present in the channel and the CRC of the data packet correct. A data slot is considered empty if the energy present in the channel is below the threshold. Finally, a data slot is considered collision if the energy present in the channel is above the threshold but a data packet has not been received or it has been received with a wrong CRC. By default, the FSA implementation uses a threshold that is set to -85 dBm and changing its value has several implications on the FSA performance. Setting a value higher, i.e., 80 dBm, will reduce the maximum communication range because nodes will be considered as interference. Contrarily, setting a value lower, i.e., -90 dBm, will increase the maximum communication range but lead to false collision detection.

- *DQ_RADIO_CHANNEL*. Indicates the channel at which FSA operates by default if channel hopping is not enabled. By default, channel hopping is disabled at the MAC layer (see Section 6.3) and the FSA implementation operates in channel 26.

In addition to the configuration parameters, the DQ module also defines the data structures that are used to store information related to the operation of the protocol during the data transmission phase. In particular the DQ module defines the dq_vars_t and the *dq_debug_t* data structures, as defined next.

The *dq_vars_t* data structure is used to keep the variables that are used during data transmission phase using DQ by both the gateway and the node. In either case, i.e., gateway or node, there are some variables that remain unused. Thus, it would be interesting to separate this variables in to two different data structures, one for the gateway and one for the node. This would allow to reduce memory footprint and to provide a clearer approach to understand what each variable does in each case.

- *packet_type* [*dq_packet_type_t*]. Type of packet, i.e., FBP, DATA or ARP.

- *mac_address* [*mac_address_t*]. Local address of the gateway or node.

- *seq_number* [*mac_seq_number_t*]. Sequence number of the packet.

- *next_channel* [*mac_channel_t*]. Channel at which DQ will operate for the next frame.

- *arp_count* [*uint8_t*]. Number of ARP slots in the access sub-period.

- *arp_selected* [*uint8_t*]. ARP selected in the access sub-period to transmit the ARP packet.

- *arp_transmitted* [*uint8_t*]. Number of ARP packets that a node has transmitted.
- *arp_rssi* [*int8_t*]. RSSI state of the packet, i.e. above or below threshold.
- *arp_rssi_threshold* [*int8_t*]. RSSI value to consider packets as collided.
- *arp_random* [*uint8_t*]. Value of the ARP in the current access sub-period. Currently nodes use their unique 16-bit address as the random value.
- *arp_total* [*uint8_t*]. Number of ARP packets that a node has transmitted to transmit to join the DTQ queue.
- *crq_wait* [*uint8_t*]. Position that a node holds when joining the CRQ to resolve a collision in the access sub-period.
- *dtq_wait* [*uint8_t*]. Position that a node holds when joining the DTQ to transmit a data packet in the data sub-period.
- *unsync_error* [*uint8_t*]. Number of unsynchronization errors.
- *crq_local* [*dq_crq_length_t*]. Local length of the CRQ queue.
- *pcrq_local* [*dq_crq_length_t*]. Local position in the CRQ queue.
- *dtq_local* [*dq_dtq_length_t*]. Local length of the DTQ queue.
- *pdtq_local* [*dq_dtq_length_t*]. Local position in the DTQ queue.
- *arp*1*_state* [*dq_arp_state_t*]. State of ARP#1 in the access sub-period, i.e., empty, success or collision.
- *arp*1*_random* [*dq_arp_random_t*]. Value of the ARP# 1 in the access sub-period. Currently nodes use their unique 16-bit address as the random value.
- *arp*1*_rssi* [*dq_arp_rssi_t*]. RSSI state of ARP#1 in the access sub-period, i.e., above or below the threshold.
- *arp*2*_state* [*dq_arp_state_t*]. State of ARP#2 in the access sub-period, i.e., empty, success or collision.
- *arp*2*_random* [*dq_arp_random_t*]. Value of the ARP#2 in the access sub-period. Currently nodes use their unique 16-bit address as the random value.
- *arp*2*_rssi* [*dq_arp_rssi_t*]. RSSI state of ARP#2 in the access sub-period, i.e., above or below the threshold.
- *arp*3*_state* [*dq_arp_state_t*]. State of ARP#3 in the access sub-period, i.e., empty, success or collision.
- *arp*3*_random* [*dq_arp_random_t*]. Value of the ARP#3 in the access sub-period. Currently nodes use their unique 16-bit address as the random value.

- *arp3_rssi* [*dq_arp_rssi_t*]. RSSI state of the ARP#3 in the access sub-period, i.e., above or below the threshold.
- *data_state* [*dq_data_state_t*]. State of the data packet in the data sub-period, i.e., empty, success or collision.
- *data_address* [*mac_address_t*]. Address of the node that transmitted its data packet in the data sub-period.
- *crq_global* [*dq_crq_length_t*]. Global length of the CRQ queue.
- *dtq_global* [*dq_dtq_length_t*]. Global length of the DTQ queue.

The *dq_debug_t* data structure is used to keep debug information related to the DQ operation. The *dq_debug_t* is packed and is transmitted through the serial line to the computer at the end of every data sub-period. Such information is used by the application to create the statistics of the data transmission phase.

- *mac_type* [*mac_type_t*]. MAC type (DQ).
- *arp1_state* [*dq_arp_state_t*]. State of ARP#1, i.e., empty, success or collision.
- *arp2_state* [*dq_arp_state_t*]. State of ARP#2, i.e., empty, success or collision.
- *arp3_state* [*dq_arp_state_t*]. State of ARP#3, i.e., empty, success or collision.
- *data_state* [*dq_data_state_t*]. State of DATA, i.e., empty, success or collision.
- *arp1_rssi* [*int8_t*]. RSSI state of ARP#1 in the access sub-period, i.e., above or below the threshold.
- *arp2_rssi* [*int8_t*]. RSSI state of ARP#2 in the access sub-period, i.e., above or below the threshold.
- *arp3_rssi* [*int8_t*]. RSSI state of ARP#3 in the access sub-period, i.e., above or below the threshold.
- *arp_total* [*uint8_t*]. Number of ARP packets that a node has transmitted to join the DTQ queue to transmit a data packet.
- *crq_wait* [*uint8_t*]. Position that a node holds when joining the CRQ to resolve a collision in the access sub-period.
- *dtq_wait* [*int8_t*]. Position that a node holds when joining the DTQ to transmit a data packet in the data sub-period.
- *arp1_random* [*dq_arp_random_t*]. Address of the node in ARP#1. The address will be empty ($0x0000$) if the ARP was empty or collision.

- *arp2_random* [*dq_arp_random_t*]. Address of the node in ARP#2. The address will be empty ($0x0000$) if the ARP was empty or collision.
- *arp3_random* [*dq_arp_random_t*]. Address of the node in ARP#3. The address will be empty ($0x0000$) if the ARP was empty or collision.
- *data_address* [*mac_address_t*]. Address of the node in DATA. The address will be empty ($0x0000$) if the DATA packet was empty or collision.
- *crq_local* [*dq_crq_length_t*]. Local length of the CRQ queue.
- *crq_global* [*dq_crq_length_t*]. Global length of the CRQ queue.
- *pcrq_local* [*dq_crq_length_t*]. Pointer to the position in the CRQ queue.
- *dtq_local* [*dq_dtq_length_t*]. Local length of the DTQ queue.
- *dtq_global* [*dq_dtq_length_t*]. Global length of the DTQ queue.
- *pdtq_local* [*dq_dtq_length_t*]. Pointer to the position in the DTQ queue.

In addition to the timing parameters, the DQ implementation also determines the structure for the DQ packets: *dq_fbp_t* for the feedback packet, *dq_data_t* for the data packet and *dq_arp_t* for the access request packet.

The *dq_fbp_t* packet is transmitted by the gateway in the feedback sub-period of each frame. The *dq_fbp_t* packet is 24 bytes long and contains the information described in the following listing. Considering the SHR (5 bytes) at the physical layer and the FCS (2 bytes) at the data-link layer, the *dq_fbp_t* packet is 27 bytes long and takes 0.864 ms (28,31 ticks) to be completely transmitted.

- *mac_type* [*uint8_t*]. MAC type (DQ).
- *packet_type* [*uint8_t*]. Packet type (FBP).
- *source* [*uint16_t*]. Address of the gateway transmitting the data packet.
- *destination* [*uint16_t*]. Broadcast ($0xFFFF$) since all nodes within communication range are expected to receive feedback packets.
- *seq_number* [*uint16_t*]. Sequence number of the current frame to allow nodes ensure that they are correctly synchronized.
- *arp1_state* [*uint8_t*]. State of ARP#1, i.e., empty, success or collision.
- *arp1_random* [*uint16_t*]. Value of the ARP# 1 in the access sub-period. Currently nodes use their unique 16-bit address as the random value.
- *arp2_state* [*uint8_t*]. State of ARP#2, i.e., empty, success or collision.

- *arp2_random* [*uint16_t*]. Value of the ARP#2 in the access sub-period. Currently nodes use their unique 16-bit address as the random value.
- *arp3_state* [*uint8_t*]. State of ARP#3, i.e., empty, success or collision.
- *arp3_random* [*uint16_t*]. Value of the ARP#3 in the access sub-period. Currently nodes use their unique 16-bit address as the random value.
- *data_state* [*uint8_t*]. State of DATA, i.e., empty, success or collision.
- *crq_global* [*uint16_t*]. Global length of the CRQ queue.
- *dtq_global* [*uint16_t*]. Global length of the DTQ queue.
- *arp_count* [*uint8_t*]. Number of ARPs in the access sub-period in the next frame. This could enable to dynamically adjust the number of number of ARPs in the access sub-period based on collision estimation algorithms.
- *next_channel* [*uint8_t*]. Channel at which the gateway will operate for the next frame. This could enable to do channel hopping to combat multi-path propagation and external interference.

The *dq_data_t* packet is transmitted by the node at the head of the DTQ in the data sub-period. The *dq_data_t* packet is 125 bytes long and contains the information described in the following listing. Considering the SHR (5 bytes) at the physical layer and the FCS (2 bytes) at the data-link layer, the *dq_data_t* packet is 128 bytes long and takes 4.096 ms (134.25 ticks) to be completely transmitted.

- *mac_type* [*uint8_t*]. MAC type (DQ).
- *packet_type* [*uint8_t*]. Packet type (DATA).
- *source* [*uint16_t*]. Address of the node transmitting the data packet.
- *destination* [*uint16_t*]. Address of the gateway that will receive the data packet.
- *arp_total* [*uint8_t*]. Number of ARP packets that a node has transmitted to join the DTQ queue to transmit a data packet.
- *crq_wait* [*uint8_t*]. Position that a node holds when joining the CRQ to resolve a collision in the access sub-period.
- *dtq_wait* [*uint8_t*]. Position that a node holds when joining the DTQ to transmit a data packet in the data sub-period.
- *data* [*uint8_t*]. Remaining bytes of the packet payload (116 bytes) that can be used to transport data from the node to the gateway.

The *dq_arp_t* packet is transmitted by the the nodes at that want to join the DTQ to transmit a data packet in the access sub-period. The *dq_arp_t* packet is 4 bytes long and contains the information described in the following listing. Considering the SHR (5 bytes) at the physical layer and the FCS (2 bytes) at the data-link layer, the *dq_arp_t* packet is 7 bytes long and takes 0.224 ms (7.34 ticks) to be completely transmitted.

- *mac_type* [*uint*8_*t*]. MAC type (DQ).
- *packet_type* [*uint*8_*t*]. Packet type (ARP).
- *random_number* [*uint*16_*t*]. Random value for the ARP. Currently nodes use their unique 16-bit address as the random value.

The DQ module also defines the following public functions that are common for both the gateway and the node implementation.

- *dq_init* Initializes the *dq_vars* and the *dq_debug_serial* variables and sets the local address of the node by calling the *ueee_addr_get_eui*16 function. It is called from the MAC module *mac_init* function during the initialization phase.
- *dq_start* Starts the execution of DQ at the beginning of the data transmission phase by pushing the dq_fbp_init function to the scheduler.

The DQ module also defines the following private functions that implement the gateway state machine of the DQ protocol. The compilation of such functions is protected by the MAC_DEVICE macro, that is, the functions are only compiled if the MAC_DEVICE is equal to $MAC\_GATEWAY$ as defined in the $config.h$ file in each project.

- dq_fbp_init. Executed at the beginning of each frame. Obtains an entry from the $packet\_buffer$ module and prepares the paypload of the feedback packet based on the information available in the *dq_vars*. After that, it registers the radio transmit interrupts, puts the feedback packet to the radio transceiver and transmits it. Finally, it creates a timeout for the duration of the feedback packet using the *virtual_timer* module that will execute the dq_fbp_done function when it expires.
- $dq\_fbp\_tx\_init$. Callback from the radio transceiver to indicate that the feedback packet is now being transmitted.
- $dq\_fbp\_tx\_done$. Callback from the radio transceiver to indicate that the feedback packet has been successfully transmitted.
- dq_fbp_done. Executed when the timeout configured from the dq_fbp_init functions expires. Puts the radio transceiver back to idle mode and releases the entry from the $packet\_buffer$ module. Then, it transmits the *dq_debug_serial* variable through the serial port using the *serial_push_message* function. Finally, it creates a timeout using the *virtual_timer* module that will execute the *dq_ack_init* function when it expires.

- *dq_arp_init*. Executed when the timeout configured from *dq_fbp_done* expires. Puts the radio transceiver in receive mode to get an access request packet from the nodes. Finally, it computes the ticks remaining until the middle of the access request packet and creates a timeout using the *virtual_timer* module that will execute the *dq_arp_rx_rssi* function when it expires.

- *dq_arp_rx_init*. Callback from the radio transceiver to indicate that a packet is now being received.

- *dq_arp_rx_rssi*. Callback from the *virtual_timer* module to indicate that it is the middle of the access request packet. It samples the RSSI value and then computes the ticks remaining until the end of the access request packet. Finally, it creates a timeout using the *virtual_timer* module that will execute the *dq_arp_done* function when it expires.

- *dq_arp_rx_done*. Callback from the radio transceiver to indicate that a packet has been successfully received. It obtains an entry from the *packet_buffer* module and copies the payload from the radio transceiver.

- *dq_arp_done*. Executed when the timeout configured from *dq_arp_rx_rssi* expires. Puts the radio transceiver back to idle mode. It first, parses the state of the current access request packet and determines if it was empty, successful or collision using the CRC and the RSSI sample. Finally, it releases the entry from the *packet_buffer* module and creates a timeout using the *virtual_timer* module. If this is the last access request packet, the timeout will execute the *dq_data_init* function to start the data sub-period to let the node at the head of the DTQ transmit its data packet. Otherwise, the timeout will execute the *dq_arp_init* function again to receive another access request packet.

- *dq_data_init*. Executed when the timeout configured from *dq_arp_done* expires. Puts the radio transceiver in receive mode to get a data packet from the nodes. Finally, it computes the ticks remaining until the middle of data packet and creates a timeout using the *virtual_timer* module that will execute the *dq_data_done* function when it expires.

- *dq_data_rx_init*. Callback from the radio transceiver to indicate that a packet is now being received.

- *dq_data_rx_done*. Callback from the radio transceiver to indicate that a packet has been successfully received. It obtains an entry from the *packet_buffer* module and copies the payload from the radio transceiver.

- *dq_data_done*. Executed when the timeout configured from *dq_data_init* expires. Puts the radio transceiver back to idle mode and releases the entry from the *packet_buffer* module. It processes the received packet and determines if it was empty, successful or collision using the CRC and the RSSI sample. With such

information it fills in the *dq_debug* variable using the *dq_vars_log* function and transmits it using the through the serial port using the *serial_push_message* function. Finally, it creates a timeout using the *virtual_timer* module that will execute the dq_fbp_init function when it expires.

- *dq_vars_reset*. Resets the contents of the *dq_vars* variable.
- *dq_vars_log*. Copies the *dq_vars* variables that will be transmitted through the serial port to the *dq_debug_serial* variable.

Similarly, the DQ module defines the following private functions that implement the node state machine of the DQ protocol. The compilation of such functions is protected by the MAC_DEVICE macro, that is, the functions are only compiled if the MAC_DEVICE is equal to MAC_NODE as defined in the $config.h$ file in each project.

- dq_fbp_init. Executed at the beginning of each frame. Resets the *dq_vars* variables and prepares to receive a feedback packet from the gateway. Finally, it computes the maximum number of ticks remaining until the end of the feedback packet and creates a timeout using the *virtual_timer* module that will execute the dq_fbp_done function.
- $dq\_fbp\_rx\_init$. Callback from the radio transceiver to indicate that a packet is now being received. It stops the current timeout and creates a new timeout using the exact number of ticks remaining until the expected end of the feedback packet. The function assumes that it is indeed a feedback packet, so it is possible to know the exact number of ticks that remain.
- $dq\_fbp\_rx\_done$. Callback from the radio transceiver to indicate that a packet has been received. It obtains an entry from the $packet\_buffer$ module and copies the payload from the radio transceiver. It then checks that if payload is valid and contains a feedback packet. If so, it updates the DQ variables with the *dq_vars_update* function.
- dq_fbp_done. Executed when the timeout configured from either the dq_fbp_init or the $dq\_fbp\_rx\_init$ functions expires. It puts the radio transceiver back to idle mode and releases the entry from the $packet\_buffer$ module. Then it checks if the packet received is a valid feedback packet. If so, it indicates that it is synchronized with the gateway and applies the DQ rules by executing the *dq_qdr_rules* function. Then it checks if the CRQ and DTQ lengths are consistent by executing the *dq_qdr_check* function. If the queues are inconsistent, it notifies the MAC module that it has lost synchronization and calculates the ticks remaining until the start of the feedback packet and creates a timeout using the *virtual_timer* module that will execute the dq_fbp_init function when it expires. If the queues are consistent, it first updates the CRQ and DTQ length and positions using the *dq_qdr_update* function and updates the DQ statistics. Afterwards, it checks if it can transmit an access request packet by executing the *dq_rtr_check* function. If so, it decides which

access request slot it will use by executing the *dq_arp_vars_set* and then it calculates the ticks remaining until the start of the selected access request packet and creates a timeout using the *virtual_timer* module that will execute the *dq_arp_init* function when it expires. Alternatively, it checks if it can transmit a data packet by executing the *dq_dtr_check* function. If so, it resets the variables related to the access request packet, calculates the ticks remaining until the start of the data packet and creates a timeout using the *virtual_timer* module that will execute the *dq_data_init* function when it expires. Finally, if it is not allowed to transmit an access request packet or a data packet, i.e., because it is waiting in either the CRQ or the DTQ queue, it calculates the ticks remaining until the start of the feedback packet and creates a timeout using the *virtual_timer* module that will execute the *dq_fbp_init* function when it expires.

- *dq_arp_init*. Executed when the timeout configured from *dq_fbp_done* or *dq_arp_done* expires. If the current access request packet is the one that has been selected to transmit, it obtains an entry from the *packet_buffer* module and fills it in with the access request packet payload. After that, it loads the packet to the radio transceiver and puts the radio transceiver in transmit mode. Finally it creates a timeout for the duration of the data packet using the *virtual_timer* module that will execute the *dq_arp_arp_done* function when it expires.

- *dq_arp_tx_init*. Callback from the radio transceiver to indicate that a packet is now being transmitted.

- *dq_arp_tx_done*. Callback from the radio transceiver to indicate that a packet has been successfully transmitted.

- *dq_arp_done*. Executed when the timeout configured from *dq_arp_init* expires. Puts the radio transceiver back to idle mode and releases the entry from the *packet_buffer* module. Finally, it increments the counter of elapsed access request packets. If this is the last access request packet it calculates the ticks remaining until the start of the feedback packet and creates a timeout using the *virtual_timer* module that will execute the *dq_fbp_init* function when it expires. If this is the selected access request packet it calculates the ticks remaining until the start of the data packet and creates a timeout using the *virtual_timer* module that will execute the *dq_data_init* function when it expires. Otherwise, if this is not the selected or the last access request packet it calculates the ticks remaining until the start of the next access request packet and creates a timeout using the *virtual_timer* module that will execute the *dq_arp_init* function when it expires.

- *dq_data_init*. Executed when the timeout configured from *dq_arp_done* expires. It obtains an entry from the *packet_buffer* module, fills it in with the data payload and resets the related counters. After that, it loads the packet to the radio transceiver and puts the radio transceiver in transmit mode. Finally, it creates a timeout for the duration of the data packet using the *virtual_timer* module that will execute the *dq_data_done* function when it expires.

- *dq_data_tx_init*. Callback from the radio transceiver to indicate that the data packet is now being transmitted.
- *dq_data_tx_done*. Callback from the radio transceiver to indicate that the data packet has been successfully transmitted.
- *dq_data_done*. Executed when the timeout configured from *dq_data_init* expires. Puts the radio transceiver back to idle mode and releases the entry from the $packet\_buffer$ module. Finally, it computes the ticks remaining until the start of the feedback packet and creates a timeout using the *virtual_timer* module that will execute the dq_fbp_init function when it expires.
- *dq_arp_vars_set*. Sets the value of the *dq_vars* variable with the appropriate information.
- *dq_arp_vars_reset*. Resets the value of the *dq_vars* variable.
- *dq_data_vars_reset*. Sets the value of the *dq_vars* variable with the appropriate information.
- *dq_vars_update*. Sets the value of the *dq_vars* variable with the information from the feedback packet.
- *dq_dtr_check*. Checks the data transmit rules to determine if a node can transmit in the data sub-period.
- *dq_rtr_check*. Checks the access transmit rules to determine if a node can transmit in the access request sub-period.
- *dq_qdr_check*. Checks if the length of the CRQ and the DTQ are consistent with the ones received in the feedback packet.
- *dq_qdr_update*. Updates the position in the CRQ and the DTQ based on the information received in the feedback packet.

Last but not least, the DQ module defines the following private functions that is common for both the gateway and the node implementation.

- *dq_qdr_rules*. Updates the status of the *dq_vars* variables based on the information received on the feedback packet.

Finally, it is important to mention that, contrarily to FSA, DQ does not have a limit in the time that nodes can maintain synchronization due to clock drift. This is because the current implementation of DQ forces all nodes to listen to all FBP that are transmitted by the gateway. Such packets allow nodes to compensate for the clock drift and keep synchronized with the gateway. However, this comes at the cost of an increased energy expenditure. Since nodes have to listen to all FBPs regardless of the state of the CRQ and DTQ queues, the energy consumption is increased. In that sense,

it would be possible to change the implementation so that nodes only listen to enough FBPs to maintain synchronization. However, such mode of operation is currently not implemented.

Chapter 7

OpenDQ results

This chapter presents the results that can be obtained using the OpenDQ project using FSA (Frame Slotted ALOHA) and DQ (Distributed Queuing) as MAC protocols for the data transmission phase. In order to reproduce the experiments it is necessary to have 11 OpenMote-CC2538 boards, 1 OpenBase board and 10 OpenBattery boards, as well as 20 AAA batteries. To reproduce the experiments program 1 OpenMote-CC2538 as Gateway and 10 OpenMote-CC2538 as Node. The Gateway device needs to be connected to the computer using the OpenBase, whereas the 10 Node devices need to be connected to an OpenBattery board. Once programmed, execute the OpenDQ application and follow the experiments and results described in the next sections.

7.1 Frame-Slotted ALOHA

The performance of FSA depends on the number of slots per frame (k) and the number of nodes that are present in the network (n). A number of slots per frame smaller than the number of nodes present in the network ($k < n$) will lead to a large number of collisions, whereas a number of slots per frame larger than the number of nodes present in the network ($k > n$) will lead to a large number of empty slots per frame. The theoretical optimal performance is achieved when the number of slots per frame is the same as the number of nodes in the network ($k = n$), which translates into a success probability of 36.8% ($1/e$).

This results, however, assume that the physical layer is ideal, e.g., all nodes are received with the same power at the gateway. Unfortunately, the reality is that even small differences in the power with which the network coordinator receives each node present in the network, i.e., caused by differences in impedance matching in the antenna front end or by physical distance of the nodes with respect to the gateway, will have an impact on FSA performance. Such impact is translated into nodes that are *closer* to the gateway have a higher probability of transmitting a successful packet, i.e., they capture the channel. This situation creates an unfairness in the network since nodes that are farther have a lower probability of transmitting a successful packet and, thus, have to spend more energy to transmit their packets. Since nodes are battery operated

devices, this creates a situation in which the battery of nodes that are at the edge will be depleted faster than nodes that are at the core.

In the following paragraphs the performance of FSA is demonstrated with various number of slots per frame and various number of contending nodes present in each configuration. The first experiment is with one slot per frame and only one node in the network ($k = 1$, $n = 1$). As it can be seen in Figure 7.1, the success probability of the node is 100% and the collision and empty probability is 0%. This is because the node does not contend with any other node to access the single slot that is available in each frame.

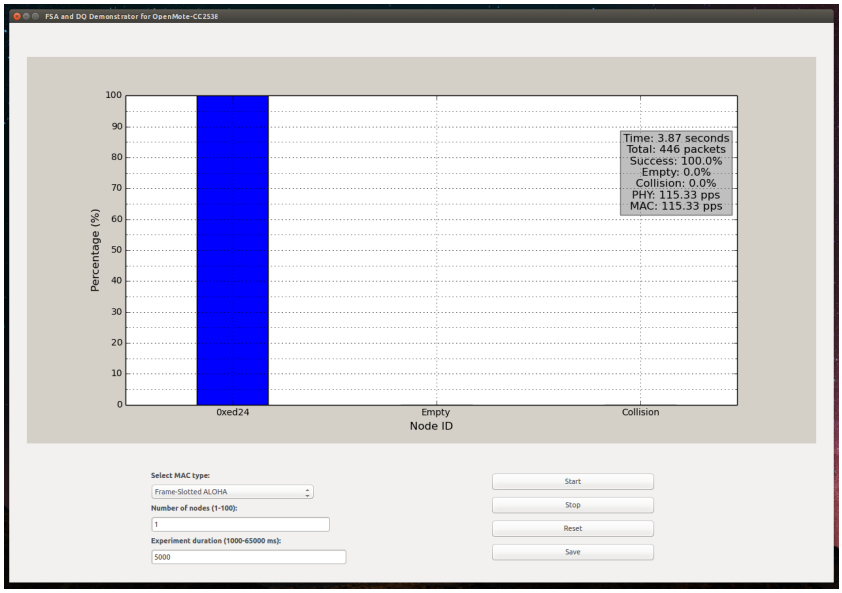


Figure 7.1: FSA results with $k = 1$, $n = 1$.

The second experiment is with two slots per frame and five nodes in the network ($k = 2$, $n = 5$). In such configuration the success probability of each node should be around 3% [1] and the overall success probability should be around 15%. However, as depicted in Figure 7.2, it is possible to observe that the success probability is not the same for each node, i.e., node $0x7faf$ is around 20% whereas node $0xeca4$ is around 5%. This is due to the capture effect, as explained earlier, which also affects the overall success probability in a positive way, i.e., it is around 51.59% instead of the theoretical 15%.

In the third experiment the number of slots per frame is the same but the number of contending nodes increases to ten ($k = 2$, $n = 10$). According to the formula presented earlier, under such conditions the success probability of each node should be around 0.1% and the overall success probability should be around 1%. However, as depicted

[1] In FSA the probability of a device successfully transmitting its data packet in a given frame with m slots and n contending nodes, assuming that there are neither transmission errors nor capture effect, can be calculated as $p_s = (\frac{1}{m}) \cdot (1 - \frac{1}{m})^{(n-1)}$.

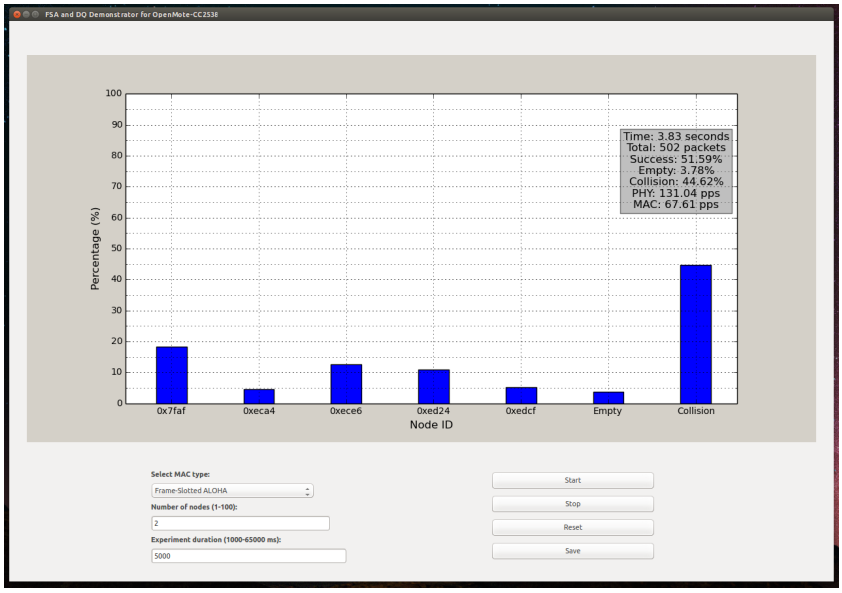


Figure 7.2: FSA results with $k = 2$, $n = 5$.

in Figure 7.3, the obtained overall success probability is above the theoretical value (12.15%) and it is also possible to observe that a given node ($xece6$) is still capable to transmit its data packets with a success probability higher than the remaining nodes (5%). As explained earlier, such outcome is due to the capture effect.

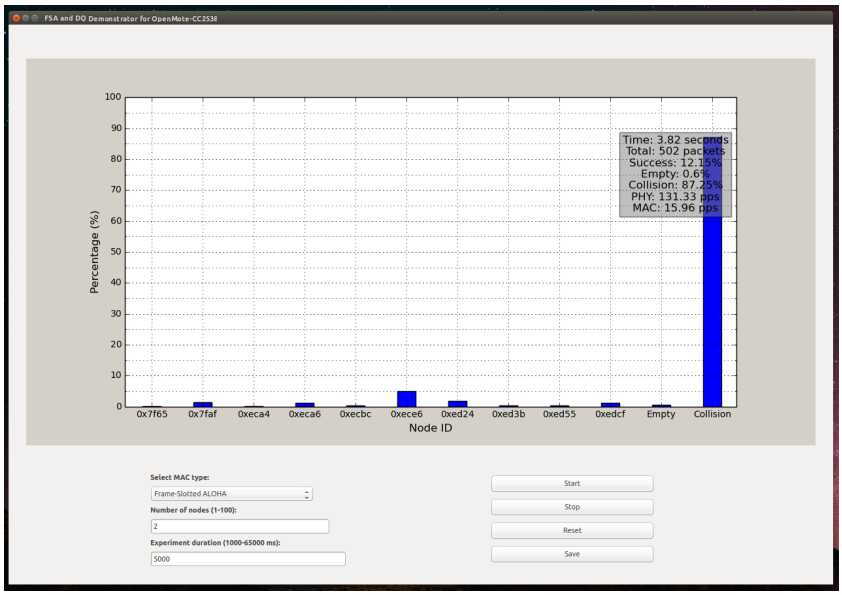


Figure 7.3: FSA results with $k = 2$, $n = 10$.

In the fourth experiment, the number of slots per frame is larger than the number of contending nodes ($k = 5$, $n = 2$). According to the formula presented earlier, under such conditions the success probability of each individual node should be 16% and the overall success probability should be 32%. As depicted in Figure 7.4, the individual node success probability is around 16% and the overall success probability is 32.78%. Such results are in accordance with the theoretical results because, given the low probability of two nods selecting the same slot, the capture effect does not play a significant role in the overall results.

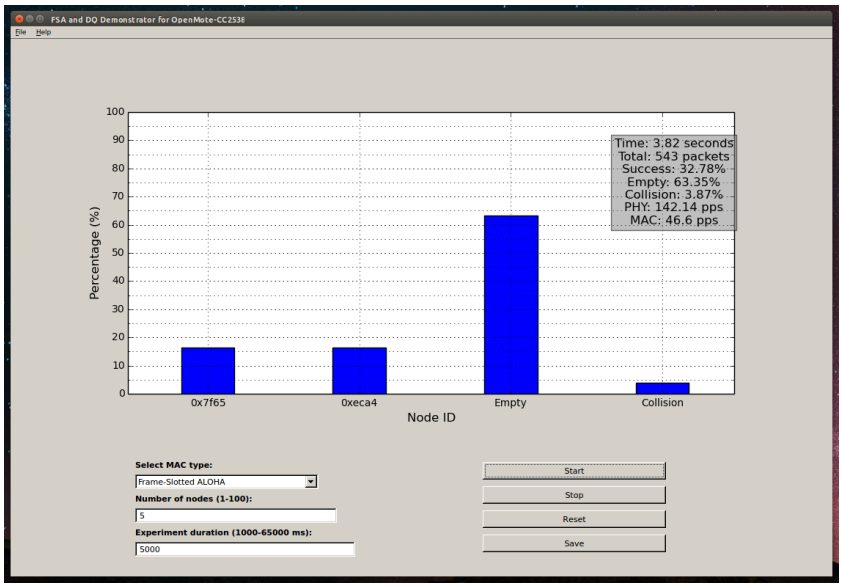


Figure 7.4: FSA results with $k = 5$, $n = 2$.

In the fifth experiment, the number of contending nodes is the same but the number of slots per frame increases to ten ($k = 10$, $n = 2$). According to the formula presented earlier, under such conditions the success probability of each individual node should be 9% and the overall success probability should be 18%. As depicted in Figure 7.5, the individual node success probability is around 9% and the overall success probability is 16.82%. Again, the obtained results are in accordance with the theoretical results because the capture effect does not play a significant role in the overall results.

Finally, in the sixth experiment the number of nodes is equal to the number of contending nodes ($k = 5$, $n = 5$). According to the formula presented earlier, under such conditions the success probability of each individual node should be 8.2% and the overall success probability should be 41%. As depicted in figure 7.6, the obtained overall success probability is above the theoretical value (61.33%) and it is also possible to observe that certain nodes are also capable to transmit its data packets with a success probability higher than the expected value ($> 10\%$). In contrast with the previous experiments, when the number of slots per frame is equal to the number of contending nodes the influence of the capture effect plays an important role.

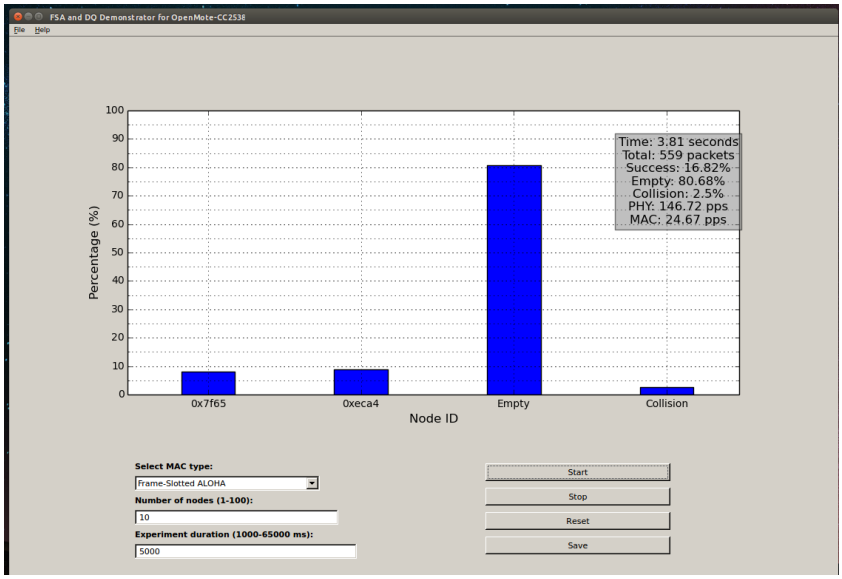


Figure 7.5: FSA results with $k = 10$, $n = 2$.

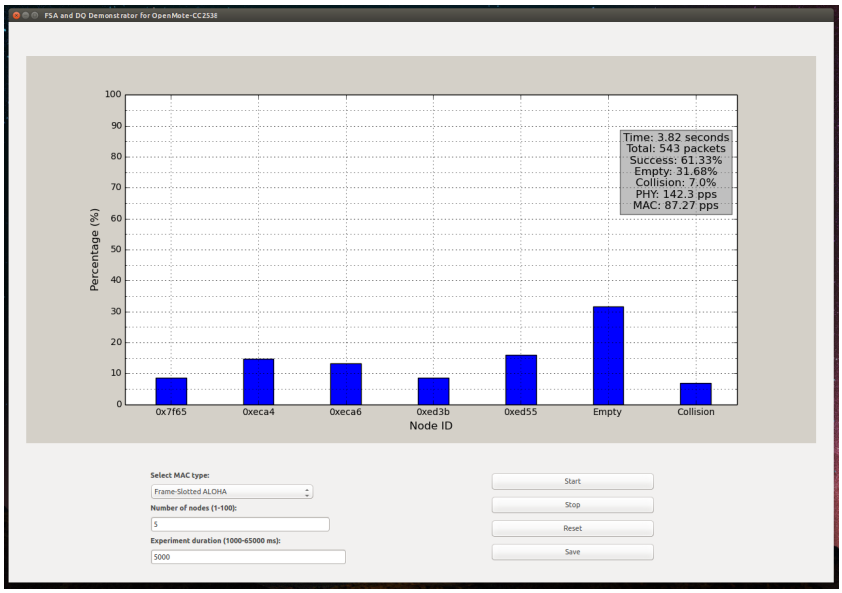


Figure 7.6: FSA results with $k = 5$, $n = 5$.

7.2 Distributed Queuing

Contrarily to FSA, the performance of DQ is independent of the number of nodes that are present in the network. Such behaviour is due to the fact that the process to transmit data packets is split into two sub-periods which are decoupled from each other and are

interleaved in time. Nodes first need to transmit an ARP (Access Request Packet) in the access sub-period to obtain access to the network. If the transmission of the ARP results in a collision with other nodes, these nodes join the CRQ (Collision Resolution Queue) to resolve such collision. The collision is resolved by repeating the process of transmitting an ARP in the access sub-period when the nodes are at the head of the CRQ. Once the collision is resolved the nodes that have solved the collision are allowed to join the DTQ (Data Transmit Queue). The nodes then need to wait until they are at the head of the DTQ to transmit their data packets in the data transmission sub-period.

The process to transmit ARP packets in the access sub-period and to join the network and to transmit a data packet in the data transmission sub-period is governed by a set of rules. For example, in order to transmit an ARP packet to join the network the CRQ must be empty, that is, the CRQ is blocking. Moreover, the collision resolution mechanism implemented in the CRQ is based on a split tree algorithm, which ensures that collisions are resolved in logarithmic time regardless of the distribution of collisions. Similarly, access to the DTQ is exclusive in the sense that a given node can only hold one position in the DTQ and that each position in the DTQ can only be hold by one node. This ensures that there are no collisions during the transmission of data packets in the data transmission sub-period which, in turn, ensures that the overall network performance is always close to the theoretical maximum regardless of the number of nodes.

Given the operation of DQ, in the following paragraphs its performance is demonstrated with various number of contending nodes present in each configuration. The first experiment is with only one node in the network ($n = 1$). Since a node needs to transmit an ARP in the access sub-period and then wait until the next data transmission sub-period to transmit its data packet, the success probability of the node is expected to be 50%. In turn, since there is only one node in the network the empty probability is also 50%. As it can be seen in Figure 7.7, the success probability of the node is 50% and the empty probability of the network is 50%, which is in accordance with the expected results.

The second experiment is with five nodes in the network ($n = 5$). Under such scenario, the success probability for each node is expected to be around 20% and the success probability of the whole network is expected to be close to 100%. As depicted in Figure 7.8, the success probability of each node is close to 20% and the success probability of the whole network is 98.36%, which is in accordance with the expected results. The empty percentage of the whole network (1.64%) is related to the number of periods that it takes to initially resolve the collisions among the five nodes and start transmitting data packets.

Finally, the third experiment is with ten nodes in the network ($k = 10$). Under such scenario, the success probability for each node is expected to be around 10% and the success probability of the whole network is expected to be close to 100%. As depicted in Figure 7.9, the success probability of each node is close to 10% and the success probability of the whole network is 99.18%, which is in accordance with the expected results. Again, the empty probability of the network (0.82%) is related to the number of periods that it

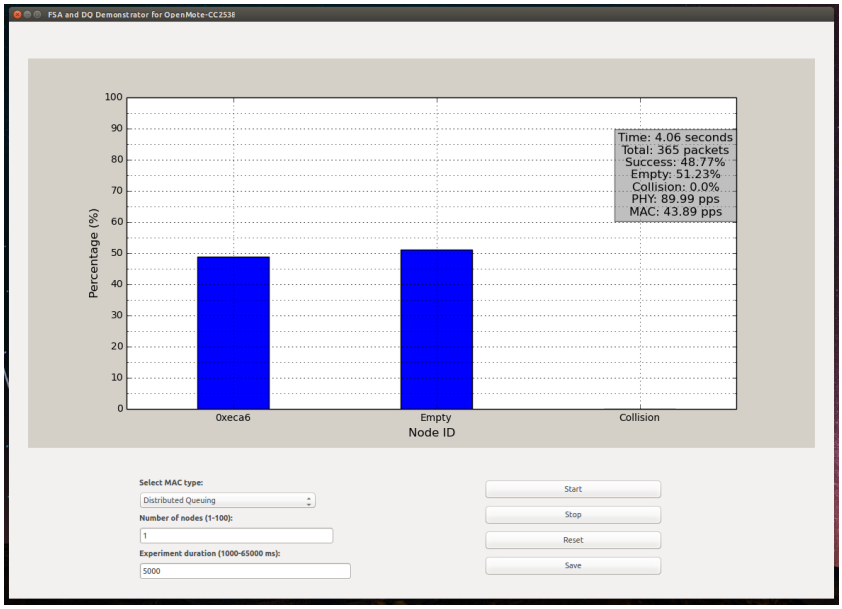


Figure 7.7: DQ results with $n = 1$.

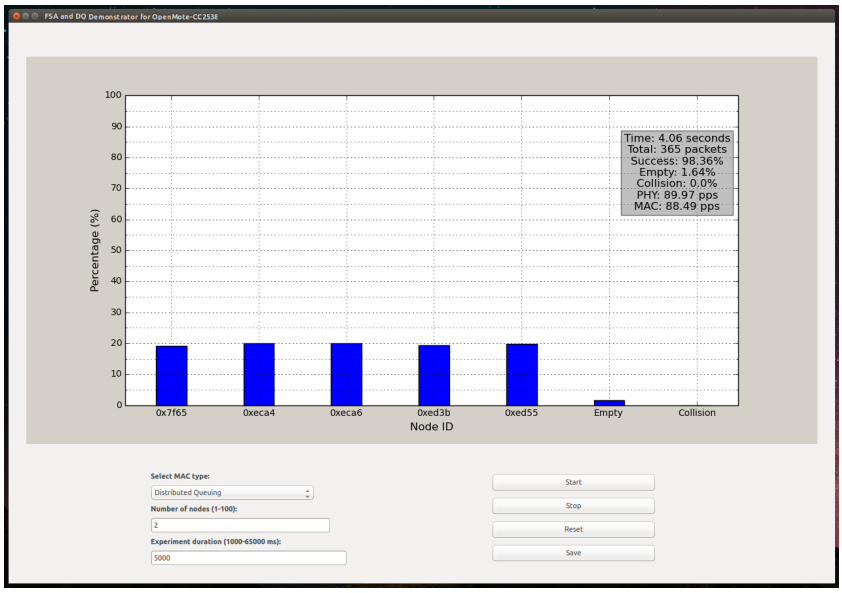


Figure 7.8: DQ results with $n = 5$.

takes to initially resolve the collisions among the ten nodes and start transmitting data packets.

Last but not least, it is important to remark that the capture effect does not affect the performance of DQ regardless the number of nodes that are present in the network,

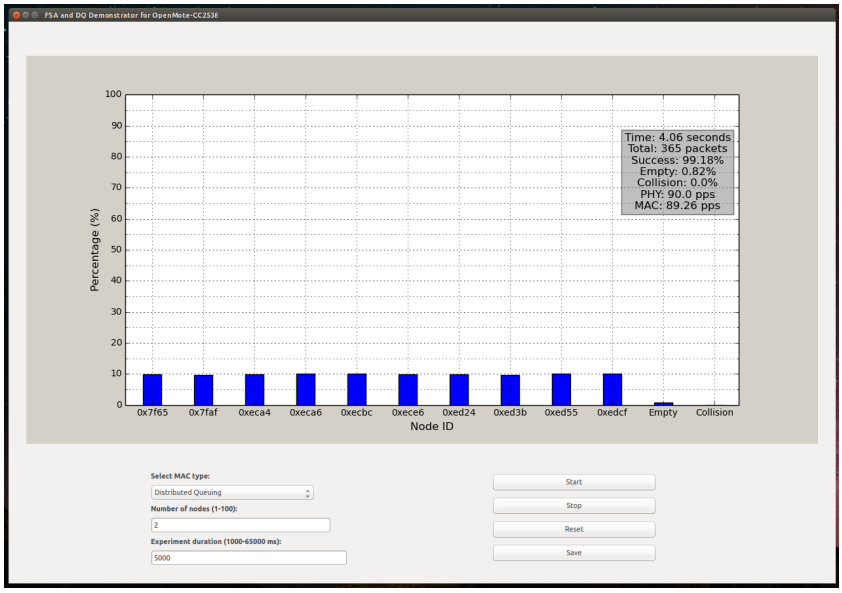


Figure 7.9: DQ results with $n = 10$.

as demonstrated in the second and third experiments. The rationale behind such fact is that the application of the rules to manage the CRQ and DTQ queues decouples the performance of the data-link layer from performance of the physical layer. That is, even the capture effect takes place at the physical layer, the data-link layer performance is decoupled from its effects because the CRQ is blocking and the DTQ ensures that there are no collisions in the transmission of data packets. In contrast to FSA, this represents an important benefit in scenarios where nodes are operated using batteries because all nodes will be depleted at the same rate regardless of their relative position to the gateway.

Bibliography

[1] *IEEE Draft Standard for Local and Metropolitan Area Networks Part 15.4: Low Rate Wireless Personal Area Networks (LR-WPANs) Amendment to the MAC sublayer.* 2011.

[2] ISO 18000-7:2009. *Radio frequency identification for item management – Part 7: Parameters for active air interface communications at 433 MHz.* ISO, Geneva, Switzerland, 2009.

[3] *IEEE Standard for Information Technology- Telecommunications and Information Exchange Between Systems- Local and Metropolitan Area Networks- Specific Requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs).* 2006.

[4] Federal Communications Commission. *Part 15 — Radio Frequency Devices; 15.231 Periodic operation in the band 40.66–40.70 MHz and above 70 MHz.* FCC Part 15.231, 2011.

[5] European Telecommunications Standards Institute. *Electromagnetic Compatibility and Radio Spectrum Matters (ERM); Short Range Devices (SRD); Radio equipment to be used in the 25 MHz to 1 000 MHz frequency range with power levels ranging up to 500 mW; Part 1: Technical characteristics and test methods.* EN 300 220-1, 2000.

[6] Dheeraj K Klair, Kwan-Wu Chin, and Raad Raad. A survey and tutorial of RFID anti-collision protocols. *Communications Surveys & Tutorials, IEEE*, 12(3):400–421, 2010.

[7] A Bachir, M. Dohler, T. Watteyne, and K K Leung. MAC essentials for wireless sensor networks. *Communications Surveys & Tutorials, IEEE*, 12(2):222–248, 2010.

[8] P Huang, L. Xiao, S Soltani, M Mutka, and N Xi. The Evolution of MAC Protocols in Wireless Sensor Networks: A Survey. *Communications Surveys & Tutorials, IEEE*, (99):1–20, 2012.

[9] Cristina Cano, Boris Bellalta, Anna Sfairopoulou, and Miquel Oliver. Low energy operation in WSNs: A survey of preamble sampling MAC protocols. *Computer Networks*, 55(15):3351–3363, 2011.

[10] W. Xu and G. Campbell. A near perfect stable random access protocol for a broadcast channel. In *Communications (ICC), 1992 IEEE International Conference on*, pages 370–374, 1992.

[11] Wenxin Xu and Graham Campbell. A distributed queueing random access protocol for a broadcast channel. *SIGCOMM Comput. Commun. Rev.*, 23(4):270–278, 1993.

[12] L. Alonso, R. Agusti, and O. Sallent. A near-optimum MAC protocol based on the distributed queueing random access protocol (DQRAP) for a CDMA mobile communication system. *Selected Areas in Communications, IEEE Journal on*, 18(9):1701–1718, 2000.

[13] J. Alonso-Zarate, C. Verikoukis, E. Kartsakli, A. Cateura, and L. Alonso. A near-optimum cross-layered distributed queuing protocol for wireless LAN. *Wireless Communications, IEEE*, 15(1):48–55, 2008.

[14] Pere Tuset, Francisco Vázquez, Jesus Alonso, Luis Alonso, and Xavier Vilajosana. LPDQ: a self-scheduled TDMA MAC protocol for one-hop dynamic low-power wireless networks. *Elsevier Pervasive and Mobile Computing, Special Issue on "Internet of Things"*, 2014.

[15] Pere Tuset, Francisco Vázquez, Jesus Alonso, Luis Alonso, and Xavier Vilajosana. Experimental energy consumption of FSA and DQ for data collection scenarios. *MDPI Sensors, Special Issue on "Wireless Sensor Networks and the Internet of Things"*, (14):13416–13436, 2014.

[16] Texas Instruments. CC2538 Datasheet: CC2538 System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee®/ZigBee IP Applications, 2014. Available online at: `http://www.ti.com/lit/ds/symlink/cc2538.pdf`.

[17] Texas Instruments. CC2538 User Guide: CC2538 System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee®/ZigBee IP Applications, 2014. Available online at: `http://www.ti.com/lit/ug/swru319c/swru319c.pdf`.

[18] OpenMote Technologies. OpenMote-CC2538 schematic, 2014. Available online at: `http://www.openmote.com/wp-content/uploads/2014/01/OpenMote-CC2538-Schematic.pdf`.

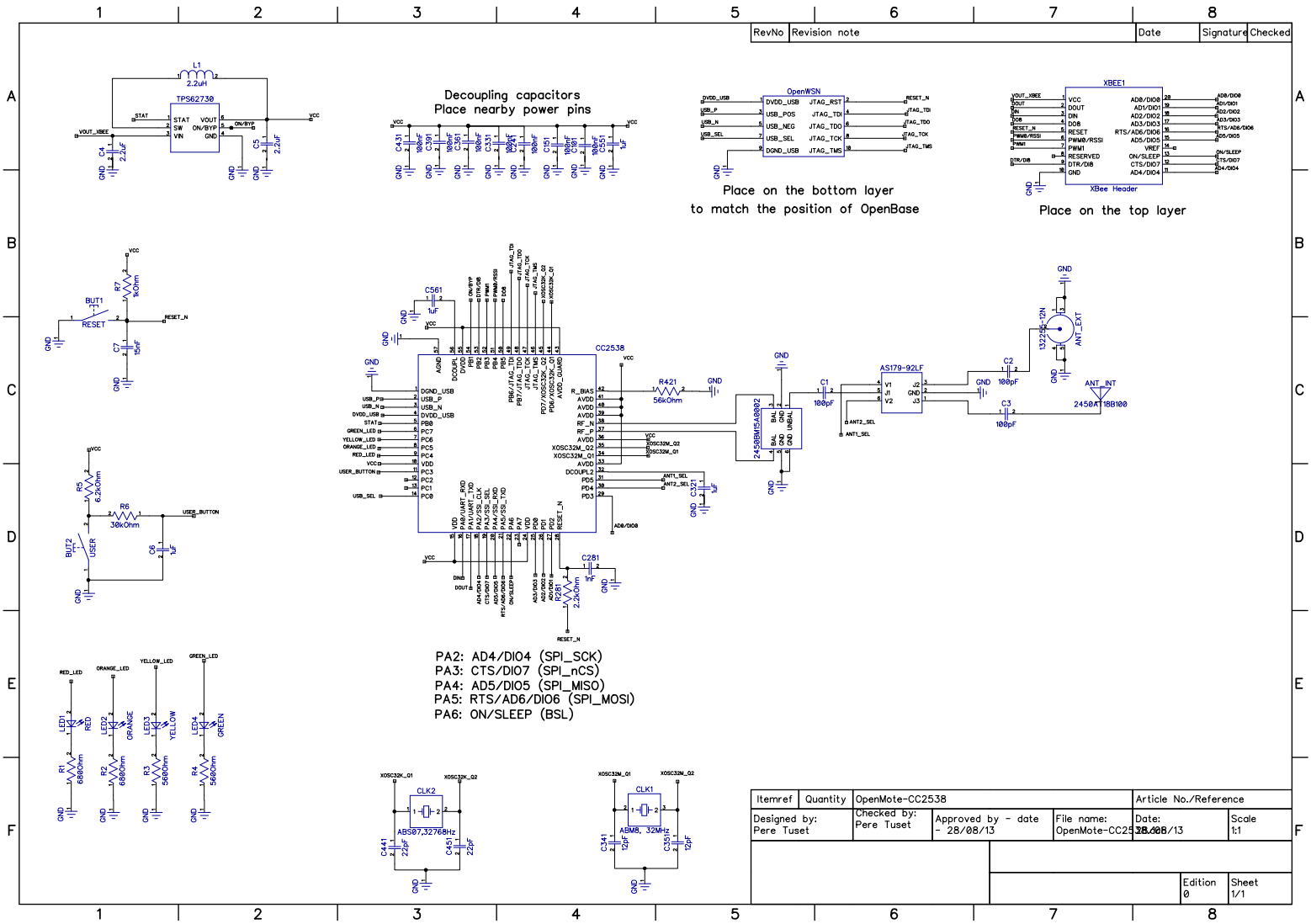
[19] OpenMote Technologies. OpenBase schematic, 2014.

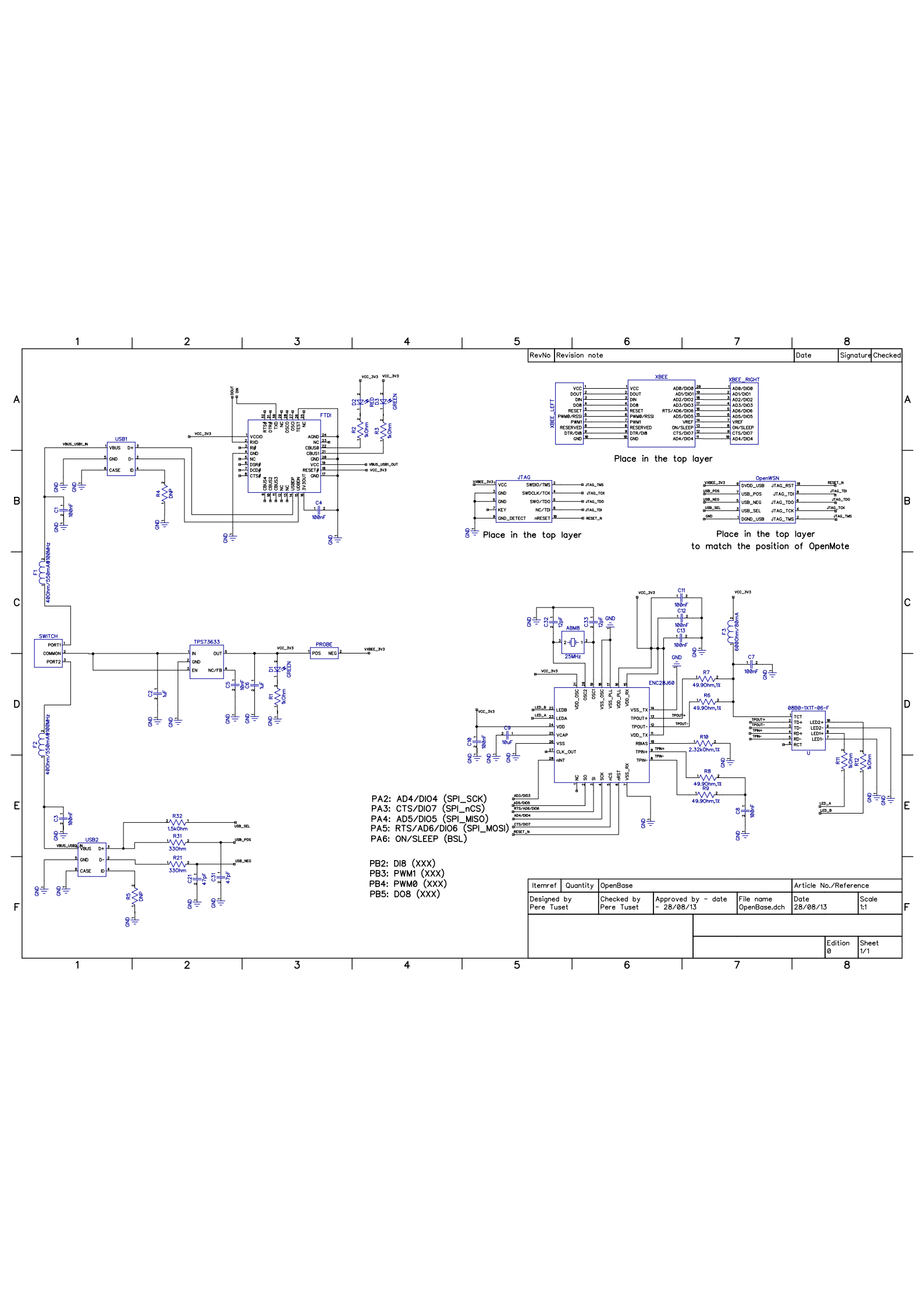
[20] OpenMote Technologies. OpenBattery schematic, 2014. Available online at: `http://www.openmote.com/wp-content/uploads/2014/01/OpenMote-CC2538-Schematic.pdf`.

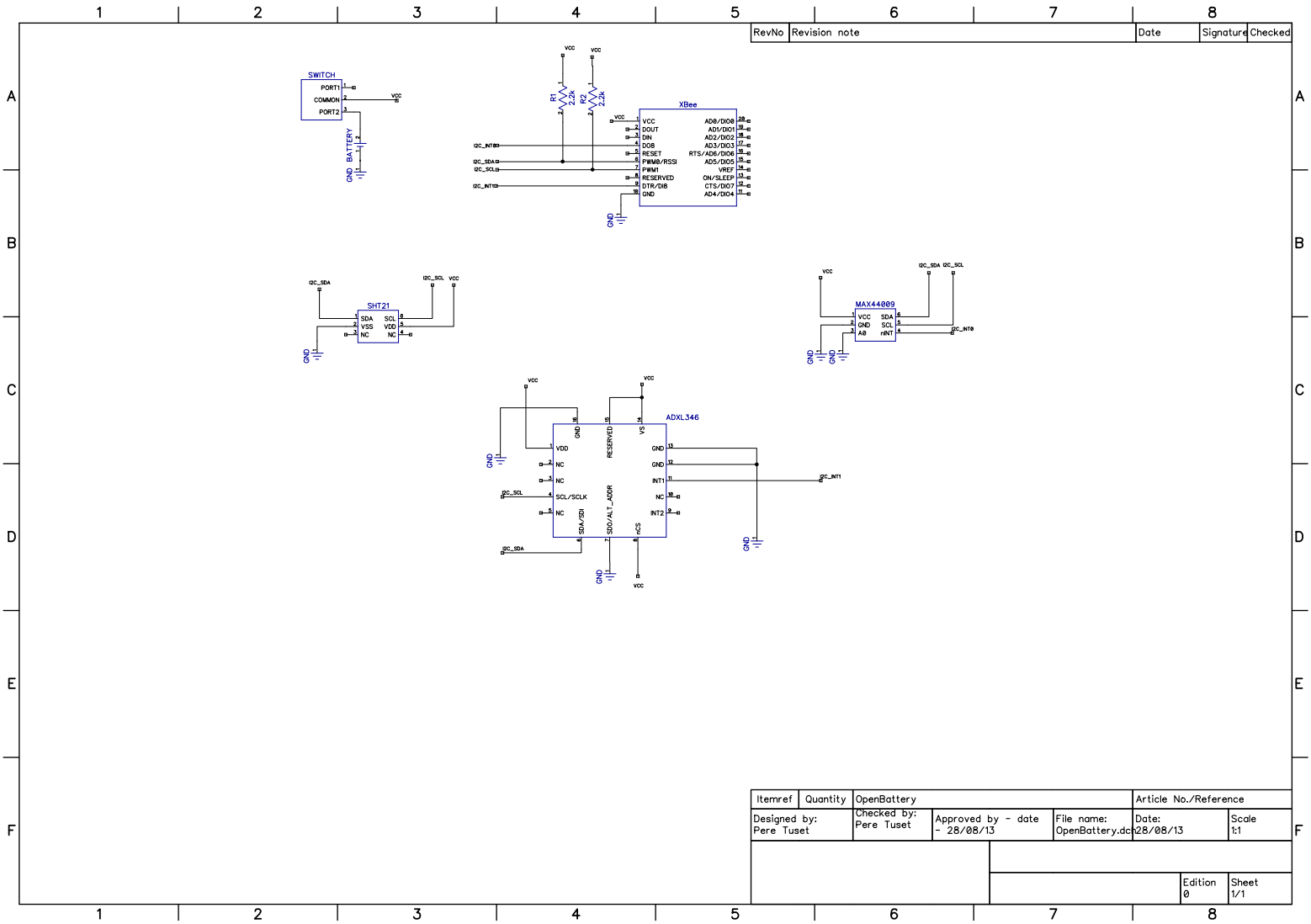
[21] Texas Instruments. CC2538 Foundation Firmware, 2014. Available online at: `http://www.ti.com/lit/zip/swrc271`.

Appendix A

Board schematics


RevNo
Revision note
Date
Signature
Checked
TPS62730
Decoupling capacitors
Place nearby power pins
OpenWSN
Place on the bottom layer
to match the position of OpenBase
XBEE1
XBee Header
Place on the top layer
CC2538
AS179-92LF
2450BM15A0002
ANT_EXT
ANT_INT
2450AT18B100
PA2: AD4/DIO4 (SPI_SCK)
PA3: CTS/DIO7 (SPI_nCS)
PA4: AD5/DIO5 (SPI_MISO)
PA5: RTS/AD6/DIO6 (SPI_MOSI)
PA6: ON/SLEEP (BSL)
ABS07,32768Hz
ABM8, 32MHz
Itemref
Quantity
OpenMote-CC2538
Article No./Reference
Designed by:
Pere Tuset
Checked by:
Pere Tuset
Approved by - date
- 28/08/13
File name:
Date:
Scale
1:1
Edition
0
Sheet
1/1


RevNo
Revision note
Date
Signature
Checked
XBEE
XBEE_LEFT
XBEE_RIGHT
Place in the top layer
JTAG
Place in the top layer
OpenWSN
Place in the top layer
to match the position of OpenMote
USB1
USB2
FTDI
TPS73633
SWITCH
PROBE
ABM8
25MHz
ENC28J60
08B0-1X1T-06-F
PA2: AD4/DIO4 (SPI_SCK)
PA3: CTS/DIO7 (SPI_nCS)
PA4: AD5/DIO5 (SPI_MISO)
PA5: RTS/AD6/DIO6 (SPI_MOSI)
PA6: ON/SLEEP (BSL)
PB2: DI8 (XXX)
PB3: PWM1 (XXX)
PB4: PWM0 (XXX)
PB5: DO8 (XXX)
Itemref
Quantity
OpenBase
Article No./Reference
Designed by
Pere Tuset
Checked by
Pere Tuset
Approved by - date
- 28/08/13
File name
OpenBase.dch
Date
28/08/13
Scale
1:1
Edition
0
Sheet
1/1



RevNo	Revision note	Date	Signature	Checked

Itemref	Quantity	OpenBattery			Article No./Reference	
Designed by: Pere Tuset		Checked by: Pere Tuset	Approved by - date - 28/08/13	File name: OpenBattery.dch	Date: 28/08/13	Scale 1:1
					Edition 0	Sheet 1/1