Implementation of an IEEE 802.15.4 Based MAC/PHY on a FPGA

by

Allyson K. Giannikouris

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2011

Abstract

The IEEE 802.15.4 standard defines the implementation of a Low-Rate Wireless Personal Area Network (WPAN). While the current version of the standard was ratified in 2006, there is still no readily available Medium Access Control (MAC) layer and/or Physical (PHY) layer for Altera Field Programmable Gate Arrays (FPGAs) in the public domain. This research investigates the implementation of the standard using an Altera FPGA for the MAC layer and PHY layer drivers. The Freescale MC13192 transceiver was used for the physical portion of the PHY layer, which includes the RF front end of the system.

The purpose of this research was to implement a basic full function device (FFD), which is capable of acting as a node in the network, as well as co-ordinating it. This allows a simple network to be tested by loading the same code on two FPGA boards, with one configured to act as a coordinator and the other as a device. The flexibility of the standard means that there are several implementation choices to be made, each of which limits the compatibility with devices using other implementation options. The implementation and design decisions made in producing a preliminary core are described in detail. The implementation of the MAC layer primitives is discussed at length as these were not available as source code. These primitives are the building blocks for the core functions of the system. Specifically, the functionality of the Energy Detection (ED) scan, stream transmit and stream receive functions are explored in detail. The code has been implemented using C and is run on the Altera Nios II soft-core processor. The work presented here is an initial implementation meant to serve as a foundation for further research. Additional functionality defined by the standard could be added, or optimization of individual functions could be explored. The current implementation also has the potential to serve as the foundation for research into various sensors which may be part of end devices in the network.

Acknowledgments

I would like to thank Dr. William Bishop, Lecturer from the Electrical and Computer Engineering department at the University of Waterloo for acting as my supervisor. Without his help and support in debugging this project it would not have been successful.

I would like to thank Dr. Mark Aagaard, Associate Professor and Dr. Hiren Patel, Assistant Professor from the Electrical and Computer Engineering department at the University of Waterloo for acting as reviewers. Their feedback for improving my thesis is greatly appreciated.

This work would not have been possible without the financial support of the Ontario government, through an OGS-ST scholarship. Financial support from the Electrical and Computer Engineering department at the University of Waterloo was also instrumental in the completion of this research.

Dedication

This is dedicated to my husband Michael. I could not have done it without his patience and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The IEEE 802.15.4 standard defines a low-rate wireless personal area network (WPAN)[1]. While some research has been done in the past on the implementation of the MAC and PHY layers of this standard on FPGA[2, 3], the implementations are not publicly available. Another challenge in implementing this standard is the flexibility of implementation that it allows. As a result, two systems compliant with the standard are not necessarily compatible with one another, depending on the configuration selected.

The implementation of a reconfigurable IP core ultimately allows users to quickly adapt the FPGA implementation for use with 802.15.4 compliant hardware produced by various vendors, regardless of the implementation options selected. A modular design also makes it possible to shrink the code footprint by eliminating functionality not used by a particular configuration.

1.1 Motivation

When the idea of implementing the standard on an FPGA was first explored, research was done to determine what was currently available in the public domain. It was observed early on that despite the standard being ratified for over 7 years, there seemed to be very few readily available implementations. Most of the implementations found were device or product family specific. These were often provided in the form of pre-compiled code by the device manufacturer. The inner workings of the code were clearly meant to remain a black box to the user. There are two problems with the use of such code. First, it cannot be easily adapted to meet the needs of a particular system. For example, unneeded features cannot easily be removed to trim the footprint of the code. Second, it is not trivial to adapt the code to an alternate transceiver as many of the code bases stray from the standard to take advantage of a given transceiver's features. Many of these same conclusions were drawn by Flora and Bonnet[4] in their review of commercially available implementations from 5 manufacturers and two open source platforms published in 2009.

The lack of a readily available FPGA core or open-source code implementation of the standard raised one obvious question: Why, after 7 years and a revision to the standard, does one not exist? The answer becomes very clear when the inner workings of the Freescale implementation are explored. The short answer is the standard, even the revised version, is impossible to implement without some modifications. The details of these will be explored in-depth when the software design of the research presented here is discussed in Chapters 4 and 5.

1.2 Intended Purpose

The goal of this research is to develop a basic implementation of the 802.15.4 standard for the Nios II processor on an Altera FPGA. This implementation can then serve as the foundation for future work. It allows further exploration of optimizations to the specific portions of the standard, as well as supports research that relies on it as a building block for a larger system. For example, the 802.15.4 standard has been used extensively in the exploration of sensor network optimization[5].

The implementation presented here is a preliminary implementation, with only the basic functionality needed to satisfy the standard. There is a considerable amount of additional functionality still to be added. For example, no security is supported. The current implementation also supports only one of two communication modes. At this time beacon-based synchronous communication is not supported. Once basic functionality is stable, the code will be made publicly available for research use. A copy can be requested by contacting Dr. Bill Bishop at wdbiship@uwaterloo.ca.

1.3 State of the Art

According to market research conducted by ABI Research, the three biggest players in the IEEE 802.15.4 IC market are Texas Instruments, Freescale and NXP[6]. Nine companies in the market were evaluated based on both their innovative features and implementation of the standard. Protocol stack availability was considered as part of the innovation score, while factors such as perceived market share, vendor size and the amount of time they have been in the 802.15.4 market factored into the implementation score.

Texas Instruments received the highest ranking from ABI. They currently offer several 802.15.4 compliant chips, both as transceivers and integrated in a system-on-chip(SOC)[7]. They provide royalty free MAC layer code, but only in the form of pre-compiled libraries. They also provide an open-source ZigBee protocol stack, Z-Stack, for use with their products.

NXP acquired Jennic in July 2010, building their portfolio of low-power RF products[8]. Their 802.15.4 compliant solutions are still referred to as Jennic products. They currently offer one 802.15.4-compliant chip, the JN5148[9]. It is a 32-bit RISC processor with support for the ZigBee Pro, 6LoWPAN, and the proprietary JenNet protocol stacks and application framework. It also allows application development directly on top of the MAC layer. Because the processor is integrated it is not practical for use in development on other controller platforms, such as FPGA.

Freescale is currently selling their fourth generation of 802.15.4 and ZigBee platform ICs, the MC1323x series[10]. This particular series implements SOC technology. It's predecessors, the MC1321x and MC1322x series also incorporated both the transceiver and controller in a single package. Available controllers range from 8-bit MCUs to 32-bit ARM7 processors. A second generation transceiver-only option is provided by the MC1320x family. It is the successor to the MC1319x family. The 802.15.4 stack is available as part of their Beekit software codebase[11]. An 802.15.4-2006 compliant version was released with the HCS08 family MAC Codebase 2.0.0, available with BeeKit version 1.9.11 or later, in the fall of 2010[12]. Their code base is freely available, but only partially open source. The source code for the PHY layer is available, but the MAC layer is provided as pre-compiled library files. Only the MAC header files are provided in source format.

While Atmel was not one of the top contenders in the 802.15.4 market according to ABI, they are unique. They are currently providing a fully open source implementation of their 802.15.4 stack, unlike the other three[13]. However, the license for use is very explicit that the software may only be used in connection with development for Atmel products. They also offer their chips in a range of packages, including single chip solutions, transceivers and modules[14]. Like the Freescale transceivers, the SPI bus is used for communication with the standalone chips. The modules combine the transceiver, controller and the required antennas, in a complete FCC certified package.

1.4 Published Research

The published research related to the standard is limited considering the time it has been available. That said, research has been published on the implementation of various aspects of the standard, as well as attempts to optimize certain cores or actions. Efforts have also been made to extend the standard to new areas for broader compatibility.

J. Flora and P. Bonnet[4], have done a considerable amount of work based on the 802.15.4 standard. They have done research with both the original 2003 version, as well as the revised 2006 version. Their goal was to create a platform independent implementation, insofar as that is possible, of the 802.15.4 standard. While their goal is similar to that of the work presented here, they went about achieving it in a different manner. In their case, TinyOS[15] was used as the foundation for their implementation. It is an open-source operating system targeted specifically to the small footprint seen in sensor networks. It provides the messaging structure necessary to interface between the MAC layer and the application running on it. Using TinyOS allowed them to be processor independent, however a physical PHY with RF components is still required in a working system. For this, they used the Freescale MC13192, the same chip used in testing the research presented here.

One of their most critical observations with respect to the work presented here is a fundamental flaw in the standard. This flaw makes it impossible to implement a compliant device in the specified manner. The standard largely neglects detailed timing specifications, while at the same time placing limits on the time in which certain actions must be executed. As a result, the structure specified by the standard is impossible to implement in such way that these time limits are not violated. Flora and Bonnet showed this empirically using an estimate of the number of clock cycles need to complete the time-sensitive tasks. One of the proposed solutions is to shift tasks such as CRC calculation to the PHY layer. This saves the time used passing invalid frames to the MAC instead of discarding them immediately. The CRC check can also be executed incrementally, increasing the speed of frame processing. In reality this method is actively employed by commercial implementations, including the Freescale MC13192 used in this research[16].

The settings used for the CSMA-CA algorithm can impact the performance of the PAN. D. Rohm et al. have investigated the impact of these settings on beaconless networks under different traffic loads[17]. Specifically, they looked at the impact of the *macMinBE, macMaxBE* and *macMaxCSMABackoffs*. The optimum values are highly dependent on the traffic load being experienced by the network. For the research presented here, only a very simple network was tested. If more complex networks are tested in the future, this work should be revisited.

Some researchers have targeted specific portions of the standard for optimization. For example, a revised orphan algorithm has been proposed by Garcia-Sanchez et al.[18]. Using the current algorithm, a considerable amount of time and energy is needed to reconnect with the parent network. The optimized algorithm proposes three changes, including the seemingly trivial change of having the application keep a record of known network channel(s). The amount of memory consumed by doing so is trivial, and the time saved by not starting the scan at channel 1 can be significant. The remaining optimizations take advantage of the deterministic nature of beaconed networks. These are less relevant to the research presented here as only a non-beaconed network is used. Thus, the details of these will not be discussed. It should be noted that the authors did not develop their own MAC/PHY layer in conducting this research. Instead, they used commercially available ZigBee devices from Crossbow.

The published research on FPGA-based implementations to date shows that only portions of the standard were actually implemented. R. Ahmad et al. focused solely on the implementation of the CRC block[3]. They were successful in using Verilog and a Xilinx FPGA to create a CRC block capable of running at 250 kbps, the same speed as transmissions in the 2.4 GHz band. In order to test their implementation it was not necessary to implement any functionality from the MAC or PHY layers. They simply fed a predetermined digital data stream to the input of their CRC block and verified the output. Another implementation by H. Li and Z. He includes a complete MAC layer[19]. They used an 8051 CPU core on a Xilinx FPGA. The MAC layer functions were implemented using embedded software, as well as a hardware controller for repetitive functions. Their published research provides only a high level overview of their implementation, the source code of which is not openly available.

Other researchers have considered the potential of the standard beyond the RF field. One team has produced a proof of concept system in which they combine traditional RF transceivers with powerline communication (PLC)[20]. Powerline networks take advantage of the AC power systems already present in our homes to transmit information between network nodes. Since these are also low data rate networks with many of the same criteria as targeted by the standard, they surmised that the 802.15.4 MAC layer could be applied to these devices as well. Slight adaptations of the MAC layer in order to meet timing requirements were required, but they succeeded in demonstrating a heterogeneous system using the 802.15.4 standard as a basis for communication using RF and PLC mediums.

Additional FPGA-based research has been done using the Bluetooth standard[21, 22]. This is also a short-range wireless communication standard, but it is aimed at higher bandwidth requirements, allowing functions such as audio and video streaming to be implemented[23]. The Bluetooth protocol has been available since 1998 and has proliferated in the consumer electronics market. It was first ratified by the IEEE in 2002 as IEEE Standard 802.15.1. Given this, it is not surprising that an extensive body of research exists. While the standards are still inherently different, there is still insight that can be gained from this work.

Chapter 2

Background

The research presented here is based on IEEE standard 802.15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs), published by the IEEE Computer Society[1]. For brevity it is referred to as the IEEE 802.15.4 standard, or simply the standard from this point forward. The fundamentals of the standard will be explored at a high-level. The network structure described by the standard, as well as the MAC and PHY layers are then reviewed in detail. ZigBee devices are the most common application, using the standard as a foundation[5]. The relationship between ZigBee devices and the standard is briefly explored.

2.1 Fundamentals of the IEEE 802.15.4 Standard

The first edition of the IEEE 802.15.4 standard was released in 2003[24]. The revised edition was released in 2006, and it is backwards compatible with few exceptions. The MAC/PHY presented here is based on the more recent version. This standard is targeted to low power, low data rate devices. Common applications include home automation and medical implementations, often in conjunction with ZigBee. The standard defines logical blocks, or layers. This theoretically allows various layers to be implemented independently of the others, as the communication between them is standardized. While only a brief high-level over view is provided here, Y. Xiao and Y. Pan have provided a more thorough review in their book[25].

The high level block diagram of the system defined by the standard is shown in Figure 2.1. There are three primary layers in the system: user application, MAC and PHY. The user application layer is often referred to as the network layer or by its abbreviation, NWK. As the name implies, the implementation of this layer, and the functionality it provides, are left to the user. That said, the NWK layer is required to have the ability to process incoming messages from the MAC, as well as generate messages to be sent to the MAC. The exchange of these messages between the layers is handled by a standardized interface known as a Service Access Point (SAP). As shown in Figure 2.1 there are two SAPs defined between the NWK and MAC layers. This allows the messages to be sorted based on the portion of the MAC layer that generated or will be handling them. The two SAPs between the MAC and PHY layers provide similar functionality. The implementation of the SAPs is discussed in more detail in Section 4.2.

2.1.1 Network Structure

The network structure defines the relationship between device nodes. It should not be confused with the application/NWK layer. The standard provides flexibility in the type of network implemented. Both star and peer-to-peer topologies are supported, as shown in Figure 2.2. Cluster tree networks can also be formed, greatly extending the range of the network. For all of these network topologies there is also flexibility in the type of communication used. They can be based on superframes or polling. In all cases, transactions take place between a coordinator and a device. The network they form is referred to as a personal area network, or PAN.

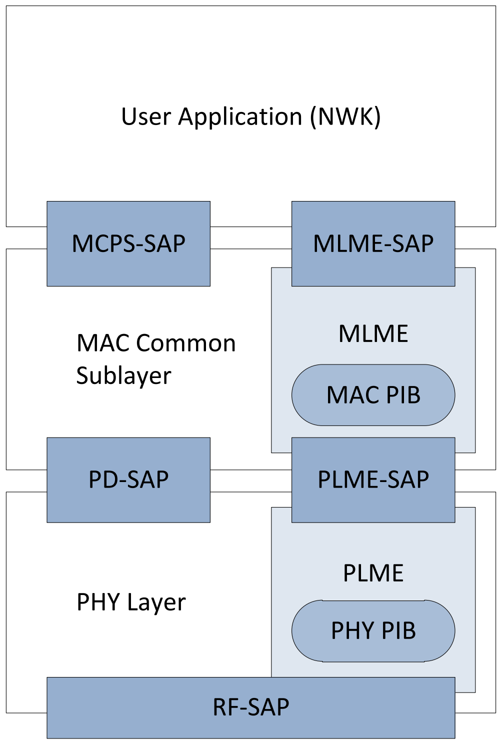


Figure 2.1: High Level Diagram of the 802.15.4 Standard

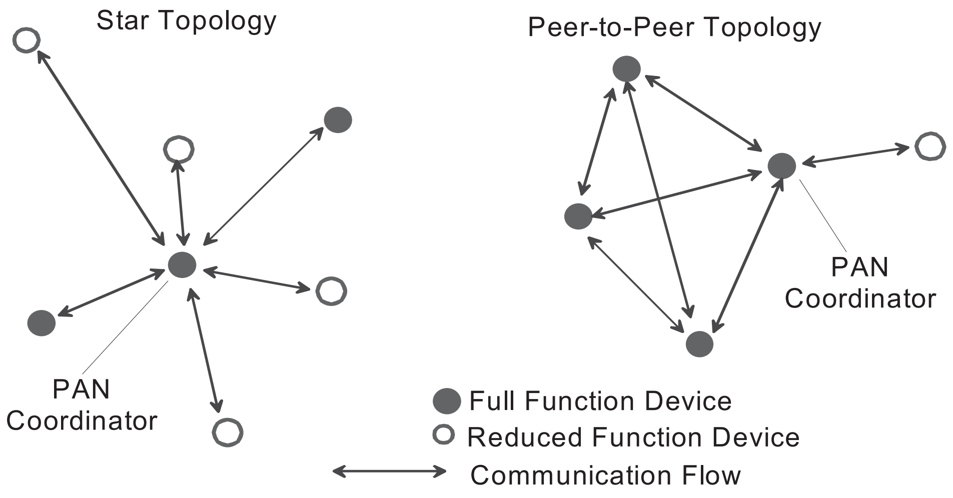


Figure 2.2: Star and peer-to-peer topology examples[1]

[1]From IEEE Std 802.15.4-2006 Copyright 2006, by IEEE. All rights reserved.

The 802.15.4 standard allows for two different classes of device to be implemented: a full-function device (FFD) and a reduced function device (RFD). The reduced function device differs from the full function device in that it does not have the ability to act as a coordinator for the network. It is, however, still capable of functioning as part of a beaconed or non-beaconed network, depending on the implementation selected. It is intended to provide a smaller footprint for device nodes with limited resources that require only a basic feature set. As shown in Figure 2.2, direct device-to-device communication is only allowed between FFDs. FFDs are also capable of taking on special roles within the network, namely coordinator and PAN coordinator. A PAN coordinator can be thought of as the master coordinator of the entire network, with every network having exactly one. The coordinators are used to extend the range of the network and provide a link from the PAN coordinator to devices which are unable to connect directly to the PAN coordinator. They are also used to implement cluster tree networks that increase the number of devices that can be supported.

In the case of a superframe based system, the coordinator sends out periodic beacons that define the start of the superframe. Each superframe consists of 16 equal slots, with the beacon occupying the first slot. There is some flexibility in how the remaining slots are used. They may all be allocated to what is known as a contention access period (CAP), or they may be divided between the CAP and a contention-free period (CFP). The CFP can consist of a maximum of seven slots. During the CAP, devices on the network must compete with one another for transmission time using a slotted CSMA-CA mechanism. All communication must be complete before the next beacon. In the case of the contention free period, what is known as guaranteed time slots (GTSs) are allocated to specific devices. Once again the device must ensure all communication is complete before the start of the next GTS or beacon. There is also the option of having an inactive period between the active period and the next beacon. This allows the coordinator to enter a low power mode between beacons and is useful in networks where infrequent data transfers are required. While synchronization allows for lower latencies, it also requires a more complex coordinator implementation to manage it.

The alternative method is for the coordinator not to use beacons, instead relying on polling-style interaction with devices. This is suitable when devices in the network do not require synchronization and can tolerate higher latencies. Eliminating period beacons also leads to a less complex controller implementation as capabilities such as time slot management and GTS allocation are no longer required. While beacons are no longer needed for normal transfers, they are still required for network initialization and device discovery. This is the method used in the implementation presented here.

2.1.2 MAC Layer

The MAC layer is the heart of the system. It is where the majority of the data processing and other major system functions are executed. The MAC layer is further subdivided into three cores: the Mac Common Part Sublayer (MCPS), the Mac Layer Management Entity (MLME), and the PAN Information Base (PIB). There are also supporting functions implemented in the MAC layer that are not specified in detail by the standard. However, they are critical to the correct operation and a functional implementation. These include the messaging system to support the SAPs, a storage structure for pending Acknowledgments, and in the case of a coordinator, a storage structure for pending messages.

The MCPS is primarily concerned with the transfer of data. It is also implicitly connected to the MLME, which allows those functions to use the data services it provides. The full set of primitives provided by this core are discussed in Section 4.4.

The MLME is the largest and most complex of the MAC cores. As the name suggests, it is responsible for the functions required to implement and maintain a network. This includes tasks such as device association, device disassociation, starting a PAN, scanning for available networks or open channels, and more. The full set of primitives provided by this core are discussed in Section 4.5.

The PIB is contained within the MLME. It is essentially a table of all variables and constants needed to setup and maintain the network. In the case of the PAN coordinator, it contains critical information about the network it is managing, such as whether or not it is accepting new devices, various pieces of beacon information, and superframe construction information. In the case of both end devices and the PAN coordinator, information such as the channel of operation, the device's address, the coordinator's address, whether or not security is enabled, and the information needed by a device to participate in a beacon enabled network are found here. This core is discussed in more detail in Section 4.3.

2.1.3 PHY Layer

The PHY layer consists of two main components: the physical transceiver and the software driver to control it. The physical components include the chip set for the actual radios, as well as a controller responsible for managing PHY settings and coordinating data transactions. The standard allows for the system to operate on the 2450 MHz, 915 MHz, and the 868 MHz bands, but an implementation of the standard need only operate on one. Because this layer is primarily hardware, it is considerably less flexible than the MAC layer.

The need for RF components makes building the PHY a complex undertaking. As a result, off-the-shelf transceivers are often used. They are available as loose chips as well as mounted on daughter cards that include external components, the RF antennas and a header for interfacing to the controller. One advantage of using an off the shelf chip is that they are already certified by the Federal Communications Commission (FCC) in the United States[26], or Industry Canada for products sold in Canada[27]. Certification is required for the operation of all radio equipment. Fortunately for manufacturer's the US and Canada regulations are virtually identical, thus compliance with one almost always implies compliance with the other[28].

From a software perspective, the standard defines the interface between the PHY and the MAC as a pair of SAPs. There are far fewer primitives defined for the PHY layer compared to the MAC, as its core purpose is to support low level data transactions. Similar to the MAC, it can be divided into three cores: PHY Data (PD), PHY Layer Management Entity (PLME), and the PHY PAN Information Base (PHY-PIB).

The PD portion is responsible for handling the transfer of messages from the MAC layer and ultimately generating the wireless signal. It also receives the wireless data and transfers it to the MAC. The PD portion is formed by only a single primitive family, `PD-DATA`.

The PLME handles all the support necessary for maintaining the network and successfully executing data transactions. It includes primitives allowing the MAC layer to read and write PHY-PIB values, set the state of the transceiver, and execute special functions. The supported special functions are energy detection (ED) scan and clear channel assessment (CCA).

Like the MAC layer, the PHY-PIB table is contained within the PLME and is used to store information needed for the operation of the PAN. This table is much smaller than the one contained in the MAC layer, consisting of only eight parameters. Four of these parameters are read only and may be a constant for a given PHY. A comparison between the PHY layer as defined by the standard and that of this implementation is located in Chapter 5.

2.1.4 ZigBee and IEEE 802.15.4

One of the most common uses for the standard is in ZigBee devices. According to ABI Research, ZigBee comprised about 40% of IEEE 802.15.4 compliant chip shipments in 2010, a number that is expected to grow to 55% by 2016[29]. As mentioned previously, the IEEE 802.15.4 protocol is used to implement the foundation upon which ZigBee applications are built[5]. ZigBee is not a single standard, but rather a collection of nine different standards. Each one is targeted to a specific application or industry, including smart energy, home automation, retail, health care and input devices. The standards are published by the ZigBee Alliance, a consortium of business, university, and government members from around the world. Since 2002 they have been working towards the goal of providing standards for low-power device networks which are capable of running on harvested energy and/or battery power for years at a time. They also aim for a network that is simple to set up and easily expandable with low maintenance needs. ZigBee takes advantage of the 802.15.4 PHY/MAC layers while using their own security and network layers, as well as an application framework, to extend the standard.

Chapter 3

Hardware Design and Verification

There are two key hardware components required to implement the system. The first is the physical transceiver, which includes the RF components used for communication. This functionality is realized using a Freescale transceiver daughter card. The second component is the controller. While this functionality is often realized using a micro-controller, an FPGA with a soft-core processor has been used for this implementation. The FPGA is mounted on a project board, providing easy access to various peripherals and interfaces.

3.1 PHY Transceiver

A pair of Freescale 13192USLK[30], 802.15.4 compliant transceivers are used for the PHY hardware. Freescale provides users with the code necessary to implement a compliant system using these transceivers and several of their micro-controller families[12]. The code to implement the drivers for the PHY layer is provided as C source code. Given this, it was possible to use the PHY layer code as a starting point and modify it as needed for implementation on the FPGA. Modifications include changes to I/O port assignment and handling, interrupt service routines, in-line assembly, and the serial peripheral interface

(SPI) interface between the FPGA project board and PHY hardware. For this implementation, a daughter card was connected to the FPGA using a standard ribbon cable[31]. This device uses the SPI bus protocol to communicate with the controller.

The hardware on this device is responsible for processing the message data sent to it over the SPI bus and appending the information needed to complete the data frame, specifically the frame check sequence (FCS). The format of this frame is discussed in detail in Section 4.4. All configuration and initiation of transceiver actions is done by reading and/or writing registers over the SPI bus. The bus is configured for 8-bit transfers, with the MSB shifted first. The clock is set to 8 MHz, with both clock polarity and clock phase set to 0. For each SPI transaction the transceiver expects three 8-bit transfers to take place. The first byte contains the 6-bit address of the target register in bits 5:0, while bit 7 is used as a R/W signal. The next two bytes are the word of data being transfered. This is done in big endian format, hence the bytes must be re-ordered by the FPGA before sending and upon receipt as the MAC/PHY software uses little endian formatting.

There are three additional outputs from the transceiver connected to the controller. The `IRQ` line must always be connected for the transceiver to function as it is used to indicate errors as well as the completion of requested actions. The use of this signal is discussed in more detail in Section 5.3. The `IDLE` and `CRC` (cyclic redundancy check) lines are optional. However, they provide the ability to increase the speed of various operations. The `CRC` line indicates that the received data frame is valid, without the need to read a register over the SPI bus. The `IDLE` line provides a quick check on the state of the requested action, indicating whether the transceiver is actively executing it or has returned to Idle.

3.2 Controller

Many of the commercially available solutions which implement the 802.15.4 standard are closely tied to a specific micro-controller or family of micro-controllers. They may also be tied to a specific transceiver or family of transceivers. In contrast, implementation of the standard on an FPGA provides users with a much more flexible and configurable device, both in terms of the processor core and custom hardware that can be added. This means that the implementation options can easily be varied, whether it be through the use of define statements in code or a more sophisticated IP core GUI in the future.

The initial implementation of the MAC/PHY layer was done using an Altera Cyclone II 2C35 FPGA on an Altera DE-2 project board[32]. This particular device is far from the fastest or most powerful FPGA produced by Altera today. The Cyclone line of FPGAs is targeted for low cost, low power solutions in contrast with the Stratix devices which are targeted for large, high-speed applications.

Even within the Cyclone family it is an older device, with the Cyclone V series now on the market. A comparison of the Cyclone II 2C35 with a comparable Cyclone V device is shown in Table 3.1. The comparable Cyclone V device was chosen based on the number of logic elements. From this, it is clear that one of the major changes is the amount of memory available. While there are fewer RAM blocks on the Cyclone V EP4CE30, they are M9K blocks. Since different blocks are used, it is necessary to calculate the number of bits actually contained in the RAM blocks of each device. Equation 3.1 shows that the Cyclone II has about 430 kbits of RAM, while Equation 3.2 shows that the Cyclone V has 541 kbits of RAM. This is equivalent to about a 25 % increase in available RAM. A second major increase has occured in the number of embedded multipliers, which has seen an increase of nearly 50 %. Additional information on both families of devices can be found on the Altera website[33].

$$4,096\,\text{bits} * 105\,\text{blocks} \approx 430\,\text{kBits} \tag{3.1}$$

$$8,192\,\text{bits} * 66\,\text{blocks} \approx 541\,\text{kBits} \tag{3.2}$$

Table 3.1: Comparison of Cyclone II and Cyclone V devices[1]

	Cyclone II	**Cyclone V**
Device	2C35	EP4CE30
Logic Elements	35,000	30,000
RAM Blocks	105 (M4K)	66 (M9K)
Embedded Memory (kBits)	473	594
18-Bit x 18-Bit Embedded Multipliers	35	66
PLLs	4	4
Maximum User I/O Pins	475	532
Speed grade	-6, -7, -8	-6, -7, -8

The nature of the MAC/PHY layer requires some form of processor capable of running software. While it is conceivable that the system could be implemented solely in hardware, it would be an extremely complex and cumbersome system. The architecture defined by the standard relies on message passing between the layers, which can be implemented much more efficiently in software. A pure hardware system would also be less flexible from an application standpoint as the user would no longer be able to run a C-code application. Since an Altera FPGA is being used, it is logical to use the Altera Nios II soft-core processor, as it is easily integrated. Altera also provides an Eclipse-based software development environment with built-in support for the Nios II processor. This processor core provides the user with flexible functionality, both in terms of processing power and interfacing capability. Since this system is intended to support low data rate and low power systems, the Nios II/e processor provides the required functionality with the smallest available Nios II footprint.

[1]Based on information available from the Altera website[33].

However, it does not support level 2 debugging. This level of debugging is a requirement for using hardware breakpoints, so the midrange Nios II/s was substituted. However, the cores are interchangeable as long as the additional functionality of the Nios II/s, such as instruction cache, are not required. As a result, once debugging is complete it should be possible to use the Nios II/e to decrease the system footprint. For a complete description of the functionality of the Nios II processor core, the handbook should be consulted[34].

The interfaces included in the processor are shown in Figure 3.1. The shaded interfaces are those used for debugging purposes, and can be removed in the final product. The SDRAM interface is connected to the onboard SDRAM chip, which has been used to store the program code. The SysId is a module required by Altera that is used for synchronization between the software generation files and the hardware loaded onto the FPGA board. The remaining interfaces are simply the FPGA side of the MC13192 interface described in Section 3.1. Information on using the software drivers provided by Altera for the various interfaces can be found in the provided documentation[35, 36].

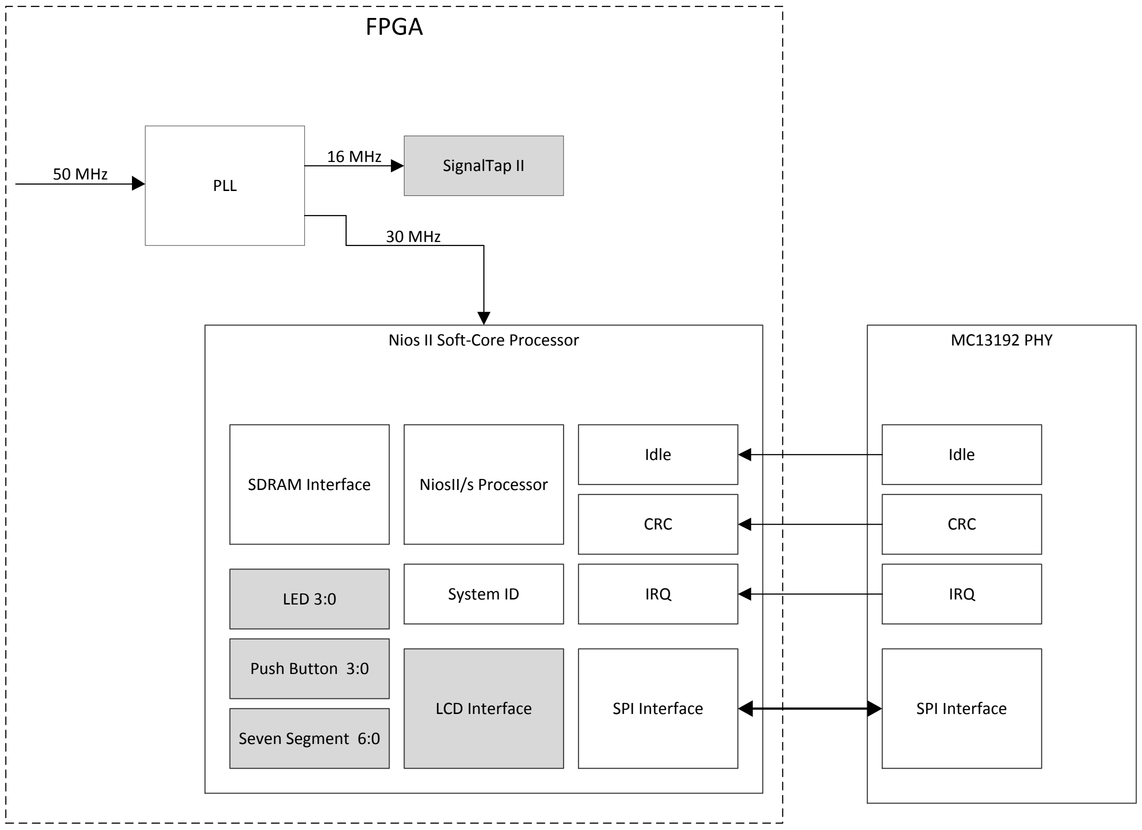


Figure 3.1: Block diagram of hardware used to implement the system

Chapter 4

MAC Software Design and Verification

Implementing the entire standard from nothing is a challenge that cannot be completed by a single person in the time available for this thesis. Instead, a more reasonable solution was to use freely available code as a starting point. Since a Freescale transceiver was selected, it was logical to use the corresponding code base. The 802.15.4-2006 compliant code base for the HCS08 family of micro-controllers was used as a starting point. The Freescale code base provides source files for the PHY and NWK layers, but not for the MAC. The MAC layer code is provided in the form of pre-compiled libraries. As a result, it was necessary to implement the MAC source code, using the provided header files as a guide.

As previously described, the standard allows a high degree of flexibility related to the implementation of compliant devices. While the ultimate goal is to implement a code base which allows the user to produce any valid implementation, only a limited number of features has been implemented so far. In this implementation, no security has been included. Only the beaconed network mode is supported, meaning that GTS is not supported. To be able to test and verify the functionality of even a simple network, a PAN coordinator is needed. Thus it was necessary to implement a FFD.

The implementation of the MAC layer is largely consistent with what is described by the standard. As noted before, only the header files were available from the Freescale implementation. These provided some clues as to their implementation, particularly message handling, but the majority of the layer had to be implemented using the standard and Freescale's reference manual[37] for guidance. Very few changes were made to the provided header files, with two exceptions of note. First, the MAC enumerated IDs were adjusted to be compliant with those listed in Table 78 of the standard[1]. Second, some structs were added to facilitate simplified information passing between functions. These structs are discussed in more detail with their corresponding functions.

As previously discussed in Chapter 2.1.2, the MAC layer can be divided into three core components: MCPS, MLME, and PIB. There are also several supporting components which must be considered to provide a more complete review of the MAC layer. In this case they include security, SAPs, Acks and MAC commands. To provide the clearest picture of the MAC layer, security, SAPs and the PIB are discussed first. This is followed discussion of the MCPS, MLME and MAC commands. Acks are then explored to complete the discussion.

4.1 Security

The 802.15.4 standard implements 3 core security services: data confidentiality, data authenticity, and replay protection. The implementation of cryptographic operations and key storage is assumed to be done in the higher level NWK layer, and as such is not explicitly specified by the standard.

The Freescale code base that was used as a starting point for this research was originally written to form the foundation of a ZigBee compliant code base. The security framework used in ZigBee differs from that specified by the IEEE standard. When the approaches were contradictory, Freescale implemented the ZigBee compliant framework. As a result the MAC layer is designed to support ZigBee Security Services Specification V.092. This means that CCM security levels are used in place of the suites described by the IEEE standard. One noted limitation that results is that secured beacons will not be processed[37].

At this time, no security is supported by the implementation presented here. However, one of the considerations while writing the software was the ease of adding security support in the future. In order to accomplish this, it was necessary to ensure that the appropriate security information was still being passed between functions where required. Since the same four pieces of security related information are always required, a struct combining this information was implemented. This minimizes the number of parameters being passed between functions. The code used to implement the structure is shown in Figure 4.1. When no security is used, the struct pointer passed between functions only leads to the security level. "If Defined" statements ensure that when security is not being used, the parsing and/or processing of information can be removed from functions at compile time to minimize the code footprint.

```
#ifdef NO_SECURITY
    typedef struct macSecurityInfo_tag{
        uint8_t securityLevel;
    } macSecurityInfo_t;
#else
    typedef struct macSecurityInfo_tag{
        uint8_t securityLevel;
        uint8_t keyIdMode;
        uint8_t keySource[8];
        uint8_t keyIndex;
    } macSecurityInfo_t;
#endif
```

Figure 4.1: Struct used to simplify passing of security information between functions

4.2 Service Access Points (SAPs)

SAPs are used for asynchronous communication between the NWK and MAC layers. They provide the framework for messages to be passed between the layers and to be placed in the appropriate queue for handling. The content of the messages passed through the SAPs is defined by the available primitives.

The 802.15.4 standard primitive families use a request-confirm sequence to send messages between the layers. Less commonly, an indication or indication-response message pair can also be used. The choice of primitive type is dependent on the direction of data flow, as shown in Figure 4.2. Requests are always generated by the NWK layer and sent to the MAC. Confirms are generated by the MAC and send to the NWK layer in response to these requests. In the case where communication is taking place between devices, there may also be an indication or an indication/response pair of primitives for a given family. In this case, the indication primitive is generated by the MAC upon the receipt of the information passed by a request primitive in the originating node via the RF medium to the receiver's PHY layer. This tells the receiver's NWK layer that some action on its behalf is being requested by the sender. In cases where the sender is requesting information from the receiver, the response primitive is used to return the information. Response functionality is rarely required. It is only used when network information is being passed to a new device, as in the case of an association, or when a device is trying to re-join a PAN after loosing the connection.

In the case of MLME and MCPS primitives, messages are generated by the sender and placed in a queue for processing by the receiver. This is shown in Figure 4.3 where the names of the receiving queues, as well as the function responsible for handling the received messages, are shown. As noted previously, deviation from the standard was necessary in the implementation of the PHY layer. As a result, no SAPs are used and no queues are needed for the MAC/PHY layer interface.

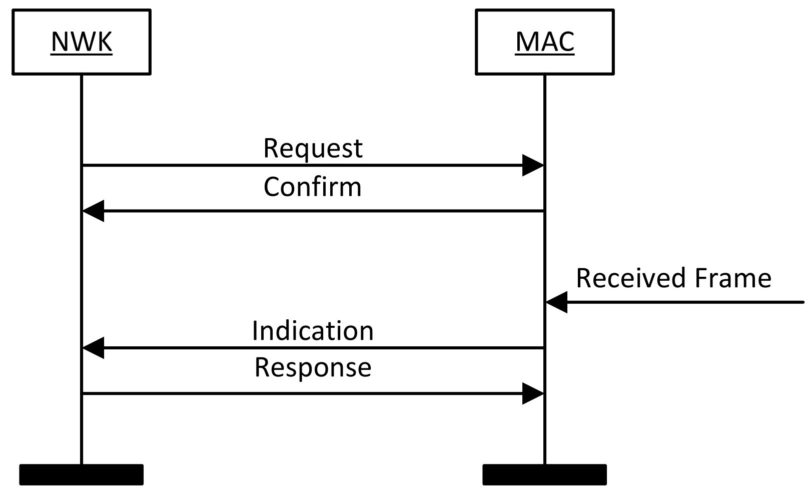


Figure 4.2: Service primitive types

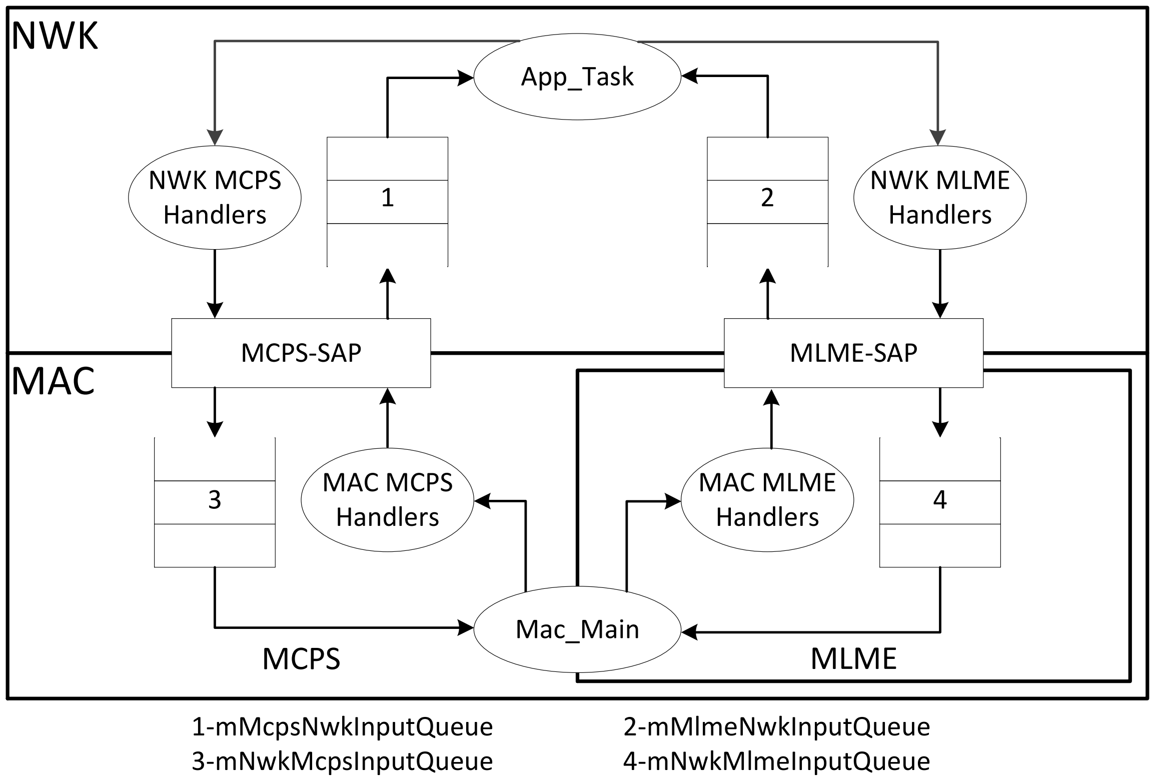


Figure 4.3: Block diagram of SAPs and associated queues

There are, however, some request primitives that are always handled immediately upon their receipt. Because of this, it is not necessary to generate the confirm primitive, as any feedback to the calling layer can be done within the calling function. The primitives that operate in this manner are the `MLME-RESET.request`, `MLME-GET.request`, and `MLME-SET.request`. Using `MLME-GET.confirm` and `MLME-SET.confirm` has been replaced with using the return code of the `MSG_Send()` function to check the status[37].

The normal sequence of events starts with the request primitive being generated in the calling layer. It is packaged as the appropriate message type, based on the sender and receiver of the message. The four possible types are `nwkToMcpsMessage_tag`, `mlmeMessage_tag`, `mcpsToNwkMessage_tag`, and `nwkMessage_tag`. The message is then sent to the receiving layer via the corresponding SAP handler, which is once again determined by the sender and receiver. The message is then added to the appropriate queue until the target layer runs the process responsible for handling its received messages. When this process is run is determined by the task scheduler. When the receiver has performed the required actions, it generates a confirm primitive. Once again, the primitive is packaged as the corresponding message type and sent to the request's sender via a SAP handler. The message is then queued and processed by the sender the next time the layer's main task runs. The sender now has the status or data it was seeking when the request primitive was sent.

All messages are passed using a standard format. Each message is defined by a struct which consists of a message type field, as well as a message data field. The type is set from an enumerated list of available primitives for the cores between which the message is to be passed. The message data field is comprised of a union of the structs used to hold the information passed by each primitive, as defined by the standard. There are four different lists of message types and corresponding message fields: NWK to MCPS, NWK to MLME, MCPS to NWK and MLME to NWK. These correspond to the two directions messages can be sent through the two SAPs. This message structure is illustrated for the NWK to MCPS case in Figure 4.4. In this case, there are only two possible message types. For other SAPs there are far more, such as the NWK to MLME case where 15 different types are handled.

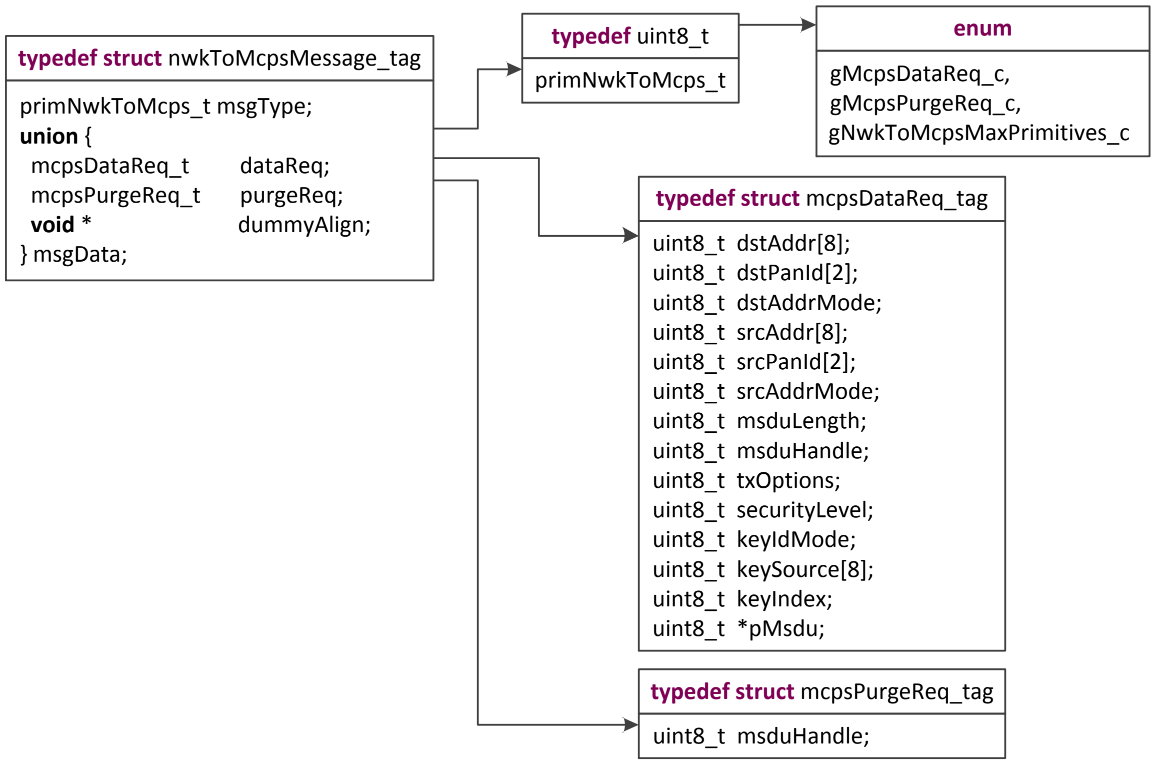


Figure 4.4: Message format structure for the MCPS SAP in the NWK to MCPS direction

The allocation of memory for messages is handled by the MAC. During MAC initialization, the heap of memory allocated for messages is subdivided into pools. This allows for the creation of different sized message blocks to accommodate primitives requiring large amounts of data to be passed without wasting resources when primitives with minimal data needs are used. The number of pools, the number of message blocks in each pool, and the size of each block is determined at compile time by settings in the `AppToMacPhyConfig.h` header file. Currently, the addition of message blocks and the creation of new pools are not supported at run time, although support could be added in the future.

During initialization, an array of anchors is created with one anchor per pool. The pools are always ordered by increasing block size, which ensures the smallest available message block of sufficient size will be used to send a message. Each anchor is simply a pair of pointers: one each for the head and tail of the corresponding message pool. A pool array of two pools with two message blocks in each list is shown Figure 4.5.

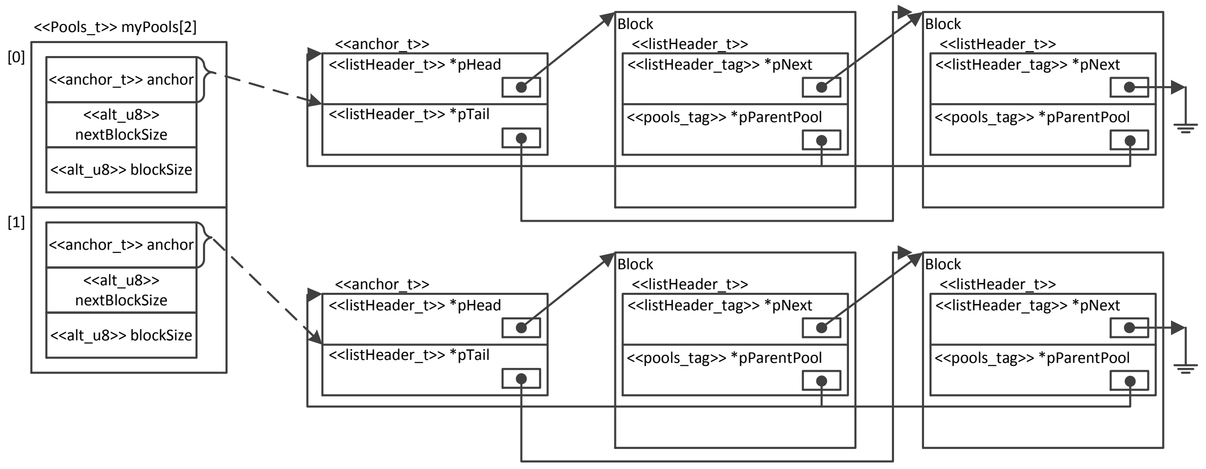


Figure 4.5: Pool array with corresponding message lists

The available message blocks within each pool are stored using a singly-linked list structure. Each message block has a header which consists of two pointers: one to the next block in the list, and the other to the corresponding pool anchor. This additional pointer provides a mechanism for returning the message block to its parent pool once the message has been received and processed. The remaining memory allocated to the block is available for use by the message being passed. The state of a single pool when two message blocks have been added is shown in Figure 4.6. Notice that both *pParentPool* pointers point to the same anchor. In the current implementation, two pools are used. The first consists of 5 messages of 36 bytes each, while the second has 5 messages of 176 bytes each.

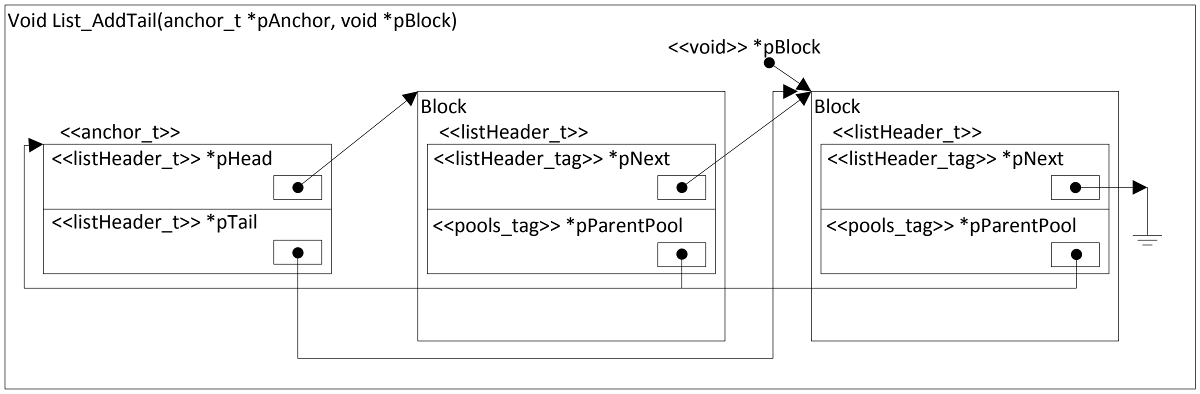

Void List_AddTail(anchor_t *pAnchor, void *pBlock)
<<void>> *pBlock
<<anchor_t>>
<<listHeader_t>> *pHead
<<listHeader_t>> *pTail
Block
<<listHeader_t>>
<<listHeader_tag>> *pNext
<<pools_tag>> *pParentPool
Block
<<listHeader_t>>
<<listHeader_tag>> *pNext
<<pools_tag>> *pParentPool

Figure 4.6: List After List_AddTail Applied

To validate the messaging structure, extensive testing was performed. Various test functions were written with the goal of testing all typical and corner cases of the system. A sample of the tests used, as well as their expected results is shown in Table 4.1. The test number shown is the same as that which is printed to the console with the result of the test when `MsgMainTest()` is called. The source code for these tests is located in `MacMsgTest.c` and is only compiled when `#define testing` has been included.

Table 4.1: List of Messaging System Test Cases and expected results

Test Number	Description	Expected Output
1	Allocate a small message buffer	The first data byte of the buffer: 0
2	Free the small message buffer	The first data byte of each buffer in the small pool: 1,2,3,4,0,
3	Allocate a large message buffer	The first data byte of the buffer, expected value 10
4	Free the large message buffer	The first data byte of each buffer in the large pool: 11,12,13,10,
5	Allocate a small message buffer when none are available	A large pool should be allocated instead. The first data byte of the buffer: 11

4.3 PAN Information Base (PIB)

The PIB table is used to store information about the current state of various MAC settings and control values. As the MAC primitives were being implemented it was found that while the PIB ID constants were defined, the table itself was implemented in the library files. As a result, the code to implement the table had to be written. The challenge of doing so is twofold. First, the identifiers, while all unique, are not necessarily sequential. Hence an array indexed by these identifiers would be highly inefficient and sparsely populated. The second challenge lies in the fact that the attributes are of varying data types, ranging from Boolean values to 64-bit addresses. As a result a structure combining the ID and Data field is not practical. To overcome this, a struct was used with elements ordered by ID. This code is located in `MacPib.h`. A separate enumerated type declaration in the `NwkMacInterface.h` header file associates the ID and entry names, as required by the standard. Some elements included in the PIB Table are proprietary to Freescale's implementation. These are listed in Table 4.2. A full list of elements as defined by the standard can be found in Table 86 of the 802.15.4 standard[1]. In keeping with the handling of integers larger than 8-bits elsewhere in the code, byte arrays are used in declaring all integer entries exceeding a byte.

Table 4.2: Freescale proprietary PIB Table elements

Attribute	**Type**	**Notes**
aMPibRole	Integer (8)	0=device 1=coordinator 2=PAN coordinator
aMPibLogicalChannel	Integer (8)	range 11 to 26
aMPibTreemodeStartTime	Integer (16)	for beaconed tree mode
aMPibPanIdConflictDetection	Boolean	
aMPibNBSuperFrameInterval	Integer (16)	
aMPibBeaconResponseLQIThreshold	Integer (8)	

From within the MAC layer the table can be directly accessed. However, there are times when the upper layer needs to know the current state of a variable or set the value of a variable. This is done using the `MLME-GET.request` and `MLME-SET.request` primitives. The 802.15.4 standard calls for `MLME-GET.confirm` and `MLME-SET.confirm` primitives to be issued by the MAC layer in response to the respective request. However, Freescale had implemented the primitives such that they are handled synchronously, with the result returned within the calling function. The `MLME-GET.request` function is capable of returning the value of any entry in the PIB table struct. However, the standard specifies that some functions be read-only to any layer other than the MAC. This can be achieved by only including those entires which can be modified in the case statement which forms the foundation of the `MLME-SET.request` handler in the MAC layer. An attempt to set a read-only entry will result in a value of `errorInvalidParameter` being returned. The implementation of both primitives is discussed in detail in Section 4.5.

4.4 MAC Common Part Sublayer Primitives

The MCPS is responsible for handling the transfer of data between the NWK and PHY layers. It is not surprising that only two primitive families are defined for this portion of the MAC. The only one required by the standard is the `MCPS-DATA` family, which has been implemented and tested. It is described in more detail below. The second primitive defined by the standard is `MCPS-PURGE`. This family is only used by a coordinator, and is an optional enhancement. It allows the NWK layer to remove data from the pending message queue. It has not been implemented due to time constraints, but can easily be added in the future. The primitives available in both of these families are shown in Table 4.3. NI indicates an unimplemented primitive while a - indicates a primitive not defined by the standard.

Table 4.3: MCPS-SAP Primitives

Family	**Request**	**Confirm**	**Indication**
MCPS-DATA	T	T	W
MCPS-PURGE	NI	NI	-

NI-Not Implemented, T-Tested, W-Written

4.4.1 Data

To carry out a single data transaction, all three primitives in the `MCPS-DATA` family are used. The transaction begins with the sender's NWK layer sending a `MCPS-DATA.request` to its MAC. The MAC will then generate the message header and append the data bytes before passing the frame to the PHY layer for transmission. The `PD-DATA.request` primitive performs the setup for a stream transaction and transfers the first word of data via the SPI bus to the transceiver. The MAC layer then sends the `MCPS-DATA.confirm` message to the sender's network layer with a status of `SUCCESS`. When the message is successfully transmitted over the RF medium and received, the receiving PHY will pass the message to its MAC layer. The message header is then stripped and the data is passed on to the NWK layer using the `MCPS-DATA.indication` primitive. This message sequence is illustrated in Figure 4.7.

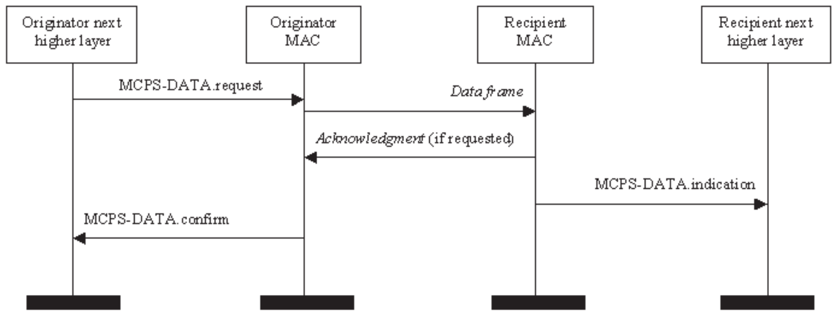


Figure 4.7: Message sequence chart describing the MAC data service[1]

The implementation of the `MCPS-DATA` primitives require a large number of variables to be passed between functions. As a result, a new struct was declared so that only one pointer is passed between functions with the information required to generate the header of the data frame. The code of the new struct is shown in Figure 4.8. Not every element in the struct will be needed every time, which becomes apparent when addressing is discussed, but the small additional overhead of unused entries far outweighs the complexity of passing each entry individually.

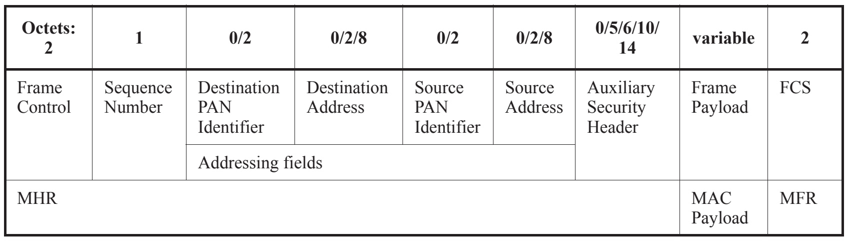
```
typedef struct hdrGenericAddr_tag{
    uint8_t  srcAddress[8];
    uint8_t  srcPanId[2];
    uint8_t  srcAddrMode;
    uint8_t  dstAddress[8];
    uint8_t  dstPanId[2];
    uint8_t  dstAddrMode;
    bool_t   ack;
    bool_t   panIDcomp;
} hdrGenericAddr_t;
```

Figure 4.8: Struct used to simplify passing address information for header generation in MCPS

Whenever data is transmitted from one device to another, the general frame format shown in Figure 4.9 is used. This general format still provides a great deal of flexibility

[1]From IEEE Std 802.15.4-2006 Copyright 2006, by IEEE. All rights reserved.

for the contents of the transmitted frame. By looking at the number of octets used by the various header fields, it is easy to see that the vast majority are optional or vary in length depending on the settings used.



Octets: 2	**1**	**0/2**	**0/2/8**	**0/2**	**0/2/8**	**0/5/6/10/14**	**variable**	**2**
Frame Control	Sequence Number	Destination PAN Identifier	Destination Address	Source PAN Identifier	Source Address	Auxiliary Security Header	Frame Payload	FCS
		Addressing fields						
MHR							MAC Payload	MFR

Figure 4.9: General MAC frame format[2]

The Frame Control Field is used as a guide to decoding the frame. It contains the information needed for a device to determine where the addresses, security information, and start of payload data are located within the received message. The format of this two-byte field is shown in Figure 4.10. The first three bits identify the type of frame being transmitted. In this case, a data frame is illustrated, so the bits are set to 001. The Security Enabled bit is used to indicate whether or not an auxiliary security header is present. Because security is not supported by this implementation, it is always 0. The Frame Pending bit is set to one if there are additional frames to be transfered to the receiver once the current transaction completes. This will be set to the appropriate value when the frame header is being generated. The Ack Request bit is set according to what has been received from the NWK layer in the *txOptions* field of the `MCPS-DATA.request` message. The destination and source address modes are also specified by the NWK layer in the message. For both addresses, the mode can be set to one of three things. If the address and PAN ID are not included in the header, the appropriate mode field is set to 00. If the address field contains the 16-bit short address, the mode is set to 10. If the extended 64-bit address is used, it is set to 11. The PAN ID Compression bit is only examined by

[2]From IEEE Std 802.15.4-2006 Copyright 2006, by IEEE. All rights reserved.

the receiver if neither address mode has been set to 00. In this case, if both addresses are associated with the same PAN ID, the transmission can be shortened by two bytes by only sending it once. If the transmission has been shortened, the receiver is notified by setting the PAN ID Compression bit to 1. The Frame Version field is used to indicate whether a given frame is backwards compatible with the 2003 version of the standard. Because security is not used, and channel pages are not supported by this implementation, the frames are always backwards compatible. As a result this field will always be set to 00. If security is added in the future, the frame version should be changed to 01.

Bits: 0–2	**3**	**4**	**5**	**6**	**7–9**	**10–11**	**12–13**	**14–15**
Frame Type	Security Enabled	Frame Pending	Ack. Request	PAN ID Compression	Reserved	Dest. Addressing Mode	Frame Version	Source Addressing Mode

Figure 4.10: Format of the Frame Control field[3]

The sequence number is obtained using a counter called *macDsn* stored in the MAC PIB table which increments after each message sent. This allows for a correlation between pending Acks from sent frames and received Acks. The lengths of the various addressing fields are determined by the information passed from the NWK to MAC layer in the `MCPS-DATA.request` message. Because security is not used, the length of the auxiliary security header will always be 0. The Frame Control, Sequence Number, Addressing fields and Auxiliary Security Header are collectively referred to as the MAC header, or MHR.

According to the standard, the frame check sequence (FCS) is to be generated by the MAC layer. A cyclic redundancy check (CRC) is used to ensure received frames are not corrupt. Upon receipt of a frame, the MAC layer is expected to calculate the CRC and compare it to the received FCS. However, this would lead to violations on Ack timing, as discussed in Section 1.4. One way to reduce the time needed for this check is to implement it in the PHY hardware, as Freescale has done with the MC13192. As a result, when the

[3]From IEEE Std 802.15.4-2006 Copyright 2006, by IEEE. All rights reserved.

MAC layer prepares data frames, the FCS is not included. When the length is passed to the transceiver for transmission, it is increased by 2 to account for the FCS that will be added. The PHY transceiver also takes care of calculating the CRC upon receipt, using the `CRC` line to indicate that the received frame has passed the check.

The `MCPS-DATA.request` was one of the first primitives to be fully tested and validated. A test function was used to create the messages in the format used by the NWK layer. These messages were then passed directly to the request handler, allowing the function to be tested independent of the SAPs. A breakpoint towards the end of the request function allowed the variable viewer in the debug environment to be used to verify that the contents of the message being passed to the `PD-DATA.request` function were complete and correctly formatted. All possible addressing combinations were tested to ensure that MHR generation was fully validated. The code used to test this primitive is shown in Appendix A. It should be noted that the validity of the values being entered in the various message fields was unimportant as the message was not actually being transmitted. What was important, was that they be correctly placed in the transmit message frame.

4.5 MAC Layer Management Entity Primitives

The MLME is responsible for starting and maintaining network connections. It is also required for all MAC functionality outside of data transfers. The needed functionality is realized using primitive families to carry out various functions. There are up to four primitives within each of these families: request, indication, response and confirm. In all cases, requests are initiated by the NWK layer and handled by the MAC layer, as are responses. Confirms and indications are initiated by the MAC layer and handled by the NWK layer.

In many cases multiple pieces of information, or data elements, are transfered by a single primitive. If a data element is larger than 8 bits, it is declared as a byte array in little endian format. The MLME primitive families are listed in Table 4.4. For each family, the state of the defined primitives in the current implementation has been indicated. The ones which are not implemented are indicated with NI, while those no longer required due to changes in implementation are indicated by NA. Those which have been partly developed are indicated by P, while W is used to indicate primitives which have been written but are untested. Those which have been fully tested and verified are indicated by T. Each family of primitives is discussed in more detail in the following sections with the exception of GTS, Rx-Enable, Sync and Sync-Loss. The GTS, Sync and Sync-Loss families are only used in beacon enabled networks, which are unsupported at this time. The Rx-Enable family of primitives is optional on both RFD and FFD devices, and is currently unused. Information on the unused primitive formats and functionality can be found in the standard[1].

4.5.1 Associate

The Associate primitives are used when a device is attempting to connect with a coordinator to join a PAN. For the purposes of testing this implementation, the information that would normally have been exchanged through the association sequence was simply hard coded. This was due to time constraints and allowed the focus to be on achieving data transactions. The indication and confirm primitives in this family serve only to pass information from the MLME to the NWK. As a result they are straightforward to implement and have been written. In the case of the confirm primitive, it simply packages information provided to it in the corresponding message format and sends it to the MLME-NWK SAP. This is easily tested by simply calling the `MacMlmeAssociateConfirm()` function, which implements this primitive, and verifying that the NWK receives it. The indication also acts as a mechanism for packaging and sending information to the NWK layer. While the functionality was not tested, verification will be trivial in the future. For the MAC

Table 4.4: MLME Primitives

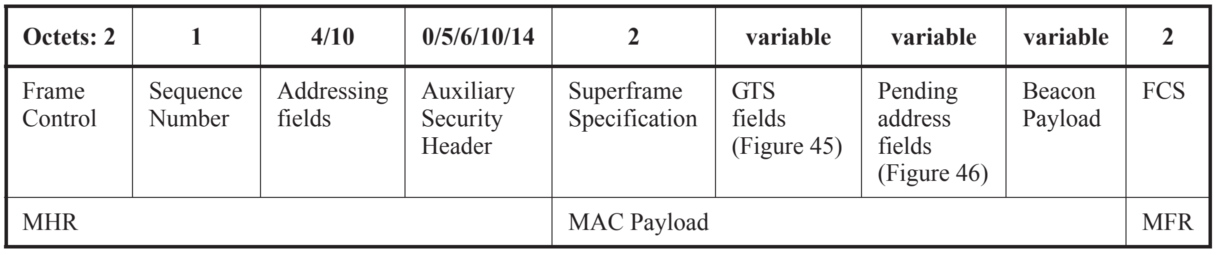
Family	**Request**	**Confirm**	**Indication**	**Response**
MLME-ASSOCIATE	NI	T	W	NI
MLME-BEACON-NOTIFY	-	-	P	-
MLME-COMM-STATUS	-	-	W	-
MLME-DISASSOCIATE	NI	W	NI	-
MLME-GET	T	NA	-	-
MLME-GTS	NI	NI	NI	-
MLME-ORPHAN	-	-	W	W
MLME-POLL	T	T	-	-
MLME-RESET	T	NA	-	-
MLME-RX-ENABLE	NI	NI	-	-
MLME-SCAN	T/P	T	-	-
MLME-SET	T	NA	-	-
MLME-START	T	T	-	-
MLME-SYNC-LOSS	-	-	NI	-
MLME-SYNC	NI	-	-	-

NA-Not Applicable, NI-Not Implemented, P-Partially Implemented, T-Tested, W-Written

layer to meet even the most basic requirements of the standard, the request and response need to be implemented and the verification of this family must be completed.

4.5.2 Beacon Notify

The `MLME-BEACON-NOTIFY.indication` is the only defined member of this primitive family. It is used by the MLME to notify the NWK layer that a beacon has been received. It will only be generated when the *aMPibAutoRequest* entry of the PIB table is set to false or when there is a data payload sent with the beacon. The format of the beacon frame is shown in Figure 4.11. The indication primitive passes information to the NWK layer including the coordinator address, channel, super frame information, GTS information and link quality. The list of pending addresses and beacon payload, if any, are also forwarded from the received frame.



Octets: 2	1	4/10	0/5/6/10/14	2	variable	variable	variable	2
Frame Control	Sequence Number	Addressing fields	Auxiliary Security Header	Superframe Specification	GTS fields (Figure 45)	Pending address fields (Figure 46)	Beacon Payload	FCS
MHR				MAC Payload				MFR

Figure 4.11: Beacon frame format[4]

The current implementation does not support beacon-based networks. However, a beacon is still sent out during startup, and devices may be on to receive it. Once the `MLME-ASSOCIATION.request` functionality is implemented, a beacon will also be generated as part of the coordinator's response to the new device.

4.5.3 Comm Status

The Comm Status primitive only exists in the form of an indication. It is issued by the MLME and sent to the NWK layer following a transmission which resulted from a response primitive. More specifically, it is issued following an `MLME-ASSOCIATE.response` primitive or `MLME-ORPHAN.response` primitive. This primitive is only used by a coordinator, as this will be the only device responding to `MLME-ASSOCIATE.request` and `MLME-ORPHAN.request` primitives from other devices in the network. The association message sequence chart shown in Figure 4.12 demonstrates how this primitive is used by the coordinator as part of the association process.

In the current implementation, the code to implement the Comm Status indication has been written but not tested. Since the indication simply takes information from the

[4]From IEEE Std 802.15.4-2006 Copyright 2006, by IEEE. All rights reserved.

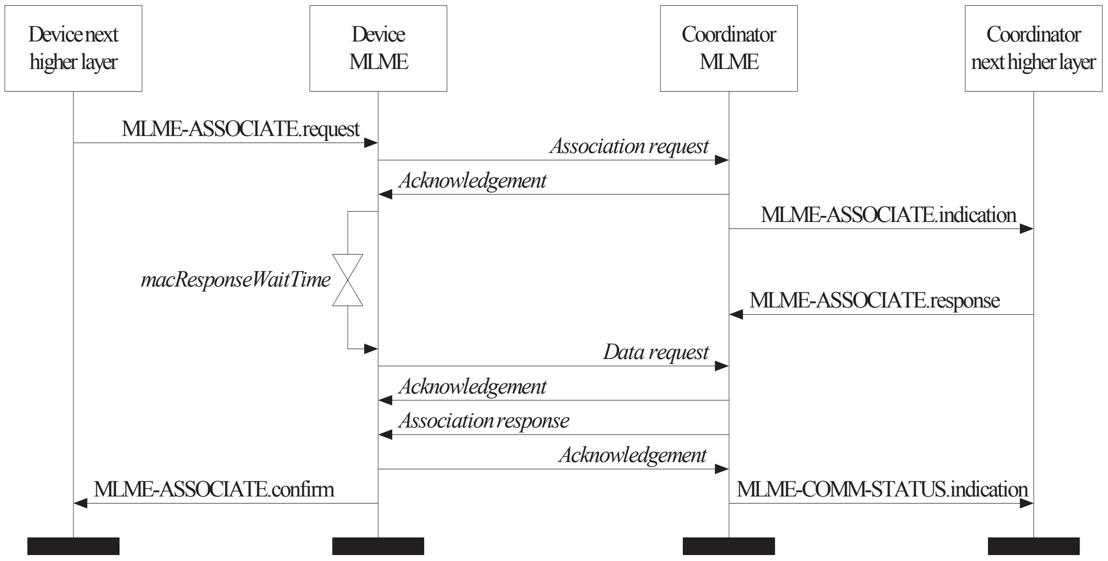


Figure 4.12: Message sequence chart for association[5]

received message and packages it as the appropriate message type for transmission to the NWK layer via the MLME-SAP, it is very similar to the functionality of various confirm primitives which have been verified. As a result, the likelihood of functionality errors being present is quite low, and testing should be straightforward. Testing was not completed in the current implementation due to time constraints.

4.5.4 Disassociate

Like the Associate primitives, writing and testing the Disassociate functionality was not critical to achieving basic communication. As a result, the request functionality was never implemented. The confirm functionality was trivial as it simply packages the status, as well as some address information, into a message and sends it to the MLME-SAP. While this code has been written it has not been tested at this time. The request primitive must be implemented for the MAC layer to meet the standard. The standard should be consulted

[5]From IEEE Std 802.15.4-2006 Copyright 2006, by IEEE. All rights reserved.

for a full description of the expected behaviour of the `MLME-DISASSOCIATE.request` primitive.

4.5.5 Get

The `MLME-GET.request` primitive allows the NWK layer to read PIB Table values through the SAP. Since the PIB Table is private to the MAC layer, they cannot be directly accessed. While on the surface this primitive seems trivial, it is complicated by the structure of the PIB Table itself. Because the elements in the table vary in size, it is not possible to simply index into the table. As noted in Section 4.3, the PIB Table itself is actually a struct to accommodate the various size requirements. Rather, the `MLME-GET.request` function must deal with each valid element access individually. The easiest way to do this is with a case statement. All elements contained in the PIB Table can be read using this primitive.

The functionality of this primitive was verified by writing a test function which creates a message of the appropriate type and passes it to the `MLME-GET.request` function. Messages are created for one entry of each possible data type as testing each entry individually is inefficient. Because loops cannot be taken advantage of, it is necessary to pass the ID or name of each entry tested manually in the test code. Because the table is implemented as a struct, the first and last entries were also tested.

4.5.6 Orphan

The Orphan family is optional for RFDs as they are only used by a coordinator, and consists of an indication and response primitive. It is used when orphaned devices are attempting to reconnect with their network. The `MLME-ORPHAN.indication` message will be sent from the MLME to the NWK layer when an orphan notification MAC command is received. The format and use of MAC commands is described in Section 4.6.

Once the indication has been received by the NWK layer, it is responsible for determining whether or not the orphan device was part of its network. This is done by comparing the device's address with those of known devices on the network. The details of how the address list is stored, as well as how the comparison is carried out are beyond the scope of the standard. If the NWK layer determines that the device is in its list, it sends an `MLME-ORPHAN.response` message to the MLME. The MLME will then generate a Coordinator Realignment MAC command, which is sent to the orphan device with the information it needs to re-join the network.

At this time, the association functionality has not been fully implemented. As a result, it is not possible for the coordinator to maintain a list of devices in the network. Since network parameters were being hard coded for the purpose of creating a simple network, support for orphan devices was unnecessary. The code for both primitives has been written, but remains untested. A mechanism for tracking associated devices must also be added to the NWK layer before functionality can be fully verified.

4.5.7 Poll

The `MLME-Poll` family consists of request and confirm primitives. The request is generated by the NWK layer and sent to the MLME when data is being requested from a coordinator. Upon receipt of the request, the MLME calls the Data Request MAC command and passes the address and security information from the network layer to it. The Data Request command will in turn register a pending Ack. If an Ack is received within the time limit, and the payload length is non-zero, the `MLME-POLL.confirm` will be generated with a status of *SUCCESS* and sent to the NWK layer. If the Frame Pending subfield is set to zero, the status returned will be *NO_DATA*. If the pending Ack expires before an Ack is received, the `MLME-POLL.confirm` will be generated with a status of *NO_ACK*.

The `MLME-POLL` family is actually one part of a larger procedure used for transferring data in polling-based networks. This larger procedure is discussed in Section 6.2. It should be noted that while this family is primarily used in non-beaconed networks, it can still be used in beaconed networks if the controller is not actively pushing data to the end devices.

Poll request messages were created with various addressing combinations using the `MacMlmePollRequestTest`. Breakpoints were then used in conjunction with the variables view in the debug screen to ensure that the information being passed to the Data Request MAC command was correct. The confirm functionality was tested in a similar manner by passing various statuses to the primitive. While the status response associated with various Ack results has been written and appears to be correct, it should be verified again once receive functionality allows for more thorough data transmission testing.

4.5.8 Reset

The reset primitive is used to return the MAC layer to its startup state without reseting the rest of the system. For the current implementation, it is necessary to empty all SAP message queues. This includes those on the NWK side of the interface as message allocation and tracking is done in the MAC layer and as a result all message pools must be returned to their default state. The list of pending Acks, and in the case of a coordinator pending messages, are also cleared on reset.

Reseting the PIB table is a special case. It is the only portion of the MAC which the user can choose whether or not to reset. While this may seem odd, when the type of information contained in the PIB is considered it becomes clear why it may be desirable not to reset it. It takes a significant amount of time and effort to setup the short address, MAC address, coordinator, channel etc. and if these have not changed, there is no reason to do it again.

The functionality of this primitive was verified by passing the proper message type to the reset function twice. The first time it was configured to reset the PIB table, the second time not to. That all settings had been restored to default throughout the MAC was verified using a combination of print statements and breakpoint. The print statements provided an easy method of confirming the message blocks had been returned to their default state, while the breakpoint allowed the variable viewer in the debug environment to be used to check the settings of PIB table values as well as other elements such as pending Ack and pending message lists.

4.5.9 Scan

The scan primitives are used by the MLME in establishing and maintaining the network. Four different scan types can be preformed: ED scan, active scan, passive scan, and orphan scan. The desired type of scan is indicated by the NWK layer using the `MLME-SCAN.request` primitive by setting the appropriate value in the *scanType* field. The 802.15.4 definition includes a *channelPage* field in the primitive definition, which is not supported in this implementation. Since the Freescale PHY only supports a single channel page, this information is unnecessary.

The ED scan and active scan are optional for RFDs. They are intended for use by a coordinator. The ED scan allows the device to scan a set of channels in order to determine the maximum energy detected on each channel during a set amount of time. The PHY is set to the desired channel and the `PLME-ED.request` primitive is repeatedly issued until time expires. The procedure is then repeated for all other channels in the set. Only the maximum value returned on a given channel is stored. After all requested channels have been scanned, an array of the maximum values is returned to the NWK layer.

The active scan is intended for use by a prospective PAN coordinator. It can be used by the device to select a PAN identifier which is not currently in use in its personal operating space (POS) in order to avoid conflicts. Alternately, it can be used by a device prior to association. The scan is performed by setting the PHY to the desired channel then sending a Beacon Request MAC command. Once the command is transmitted, the device waits for the number of symbols as given by Equation 4.1 to elapse. In this equation, n is the value of the *sacnDuration* parameter in the `MLME-SCAN.request`. The information received from all unique beacons during this time is recorded. Once the time has elapsed, the process is repeated for the next channel in the list until all have been scanned. While an active scan is being performed, any received frames which are not beacon frames are discarded. If a beacon frame with a payload is received that contains the device in its list of pending addresses, the MAC will not extract the pending data.

$$\begin{aligned} scanTime &= [aBaseSuperFrameDuration * (2^n + 1)] \\ scanTime &= [290 * (2^n + 1)] \end{aligned} \tag{4.1}$$

The passive scan is required in all devices. This type of scan allows a device to locate coordinators transmitting beacons within its POS. However, unlike the active scan a Beacon Request MAC command is not sent. The device simply listens for beacons on the selected channel. This type of scan is well suited for use by a device before association with a PAN. As with the previously considered scan types, any non-beacon frames received during the scan are automatically discarded. If a beacon frame with payload is received that includes the scanning device in its list of pending addresses, the device will not extract the data. The scan can be initiated for a given set of channels, where each channel is scanned for a given period of time. The time for each scan is calculated using Equation 4.1, where n is the value of the *scanDuration* parameter in the `MLME-SCAN.request` primitive.

The orphan scan allows a device to attempt to locate its coordinator in the event that synchronization is lost. The PHY is set to the desired channel and an Orphan Notification MAC command is sent. Any frames received during the scan that are not coordinator realignment command fames are discarded. The scan will terminate when either *macResponseWaitTime* has elapsed or it has received a Coordinator Realignment MAC command. If a coordinator realignment command was not received, the next channel is scanned until all specified channels have been scanned.

When the an Orphan Notification MAC command is received by a coordinator, an `MLME-ORPHAN.indication` primitive is sent to the NWK layer. The NWK layer then searches its device list for a match to the device indicated by the primitive. If a match is found, it sends an `MLME-ORPHAN.response` primitive. This will in turn send a Coordinator Realignment MAC command to the device. The whole process of searching the device list and generating a response must complete within *macResponseWaitTime*, otherwise the response will be missed by the device. If no match is found by the NWK layer in the device list, the coordinator will simply discard the request.

In the current implementation, passive, active and orphan scans have only been partially implemented. The focus was placed on implementing and testing ED scanning as it is actually used in the startup procedure. ED scanning was tested by calling the function from the application with a channel list consisting of all channels supported by the transceiver. SignalTap II was used to verify that the time for each scan was consistent with the transceiver specification[16]. A breakpoint was used after the scan completed to confirm that the values from the scan had been properly saved to the array and passed to the NWK layer for channel selection.

4.5.10 Set

The `MLME-SET.request` primitive allows the NWK layer to modify PIB table values through the MLME-SAP. Since the PIB Table is private to the MAC layer, they cannot be directly accessed. While on the surface this primitive seems trivial, it is complicated by the structure of the PIB table itself. Because the elements in the table vary in size, it is not possible to simply index into the table. As noted in Section 4.3, the PIB table itself is actually a struct in order to accommodate the various size requirements. As a result, the `MLME-SET.request` function must deal with each valid element access individually. The easiest way to do this is with a case statement. Having to handle each valid element access individually results in more complex code, however, it also makes it simple to protect elements which are read only outside of the MAC layer. These elements simply aren't included in the case statement, and an attempt to access them will result in an error being returned. As mentioned previously in Section 4.2, the `MLME-SET.confirm` primitive is not required as the request is handled synchronously.

Testing of the `MLME-SET.request` primitive was also complicated by the nature of the implementation. It was not possible to use a loop to step through entires. Instead, one entry of each data type was tested. The first and last entry in the PIB table, as well as an invalid entry were also tested. It is possible to set an entry by sending either the corresponding identifier as defined by the standard, or the attribute name. The attribute names have all been defined in the code as equal to their identifier. For example, the MAC short address can be set in the table using either 0x53 or `gMPibShortAddress_c` as the `pibAttribute` in the message sent from the NWK to the MLME. The functionality of this primitive has been fully tested and validated.

4.5.11 Start

The start family is used by the NWK layer of an FFD to instruct the MAC to begin operating as a PAN coordinator. This should only be done after a reset of the MAC layer, including the PIB table, has been performed. An active scan to determine a suitable channel is also recommended. In beaconed network, the receipt of an `MLME-START.request` by the MAC layer would cause it to begin transmitting period beacons based on the settings provided.

The standard calls for the MLME to set the PIB attributes if the start request is not attempting to realign and existing coordinator. However, the Freescale implementation sets the PIB attributes in the NWK layer before sending the `MLME-START.request` primitive. While this gives the same result in the case of a non-beaconed network, it would lead to violation of the standard in the case of a beaconed network. If a coordinator is being realigned, the beacon related PIB attributes are only to be changed once the frame is successfully transmitted. In the event of a channel access failure or invalid parameters being requested, no changes are made.

This primitive was tested after the MLME-SAP was confirmed to be working. Thus it was possible to confirm expected functionality using the application provided by Freescale in the network layer. While the standard recommends an active scan, an ED scan was used to select a suitable channel. Since a simple network is being tested and no other coordinators are present, it makes more sense to ensure that other equipment in the area is not interfering with the selected channel. A breakpoint was then used at the end of the MLME handler to confirm that the parameters passed in the message had been properly processed to create the message being passed to the PHY for transmission. The receipt of the `MLME-START.confirm` primitive by the NWK layer was verified using a print statement.

4.6 MAC Commands

Mac commands are used to support MAC primitives requiring the transfer of information between devices. While the MAC primitives serve as a communication mechanism between the MAC and NWK layers, fulfillment of the primitive request often requires interaction with other devices in the network. This is achieved through the use of MAC commands. A full list of commands defined by the standard is shown in Table 4.5. Whether or not an acknowledgment is required, as well as the commands status in the current implementation are also shown. An RFD is not required to be capable of transmitting and receiving all MAC commands. Those which are required are indicated by an X in the table. The commands have been divided in the code such that the footprint can be trimmed for an RFD by removing the `#define FFD` deceleration.

The `MLMEBeaconRequest()` primitive is listed as optional for an RFD device. However, the Freescale implementation of an end device on a non-beacon network uses an active scan to find the coordinator. In order for this to work, the `MLMEBeaconRequest()` command must be available. As a result it is defined for a RFD device in this implementation.

Table 4.5: MAC Command Frames[6]

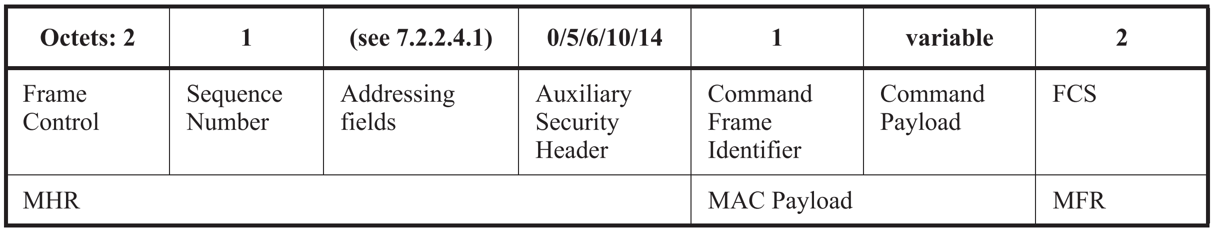
Command Frame Identifier	**Command Name**	**RFD Tx**	**RFD Rx**	**ACK**	**Status Tx**	**Status Rx**
0x01	Association request	X		Yes	P	NI
0x02	Association response		X	Yes	P	P
0x03	Disassociation notification	X	X	Yes	P	P
0x04	Data request	X		Yes	T	W
0x05	PAN ID conflict notification	X		Yes	T	W
0x06	Orphan notification	X		No	T	W
0x07	Beacon request			No	P	NI
0x08	Coordinator realignment		X	Opt.	W	W
0x09	GTS request			Yes	NI	NI

NI-Not Implemented, P-Partially Implemented, T-Tested, W-Written

The MAC commands use a frame format derived from the general MAC frame discussed in Section 4.4.1. The general MAC command frame is shown in Figure 4.13. Every MAC

[6]Adapted from IEEE Std 802.15.4-2006[1]

command frame starts with the MHR field, which contains the Frame Control, Sequence Number and Addressing fields. The information is identical to the Data frames, with the exception of the Frame Type bits of the Frame Control field, which are 011 for a MAC command. The MAC Payload is slightly different from that of the Data frames as the first byte always contains the Command Frame Identifier in order to indicate the command type. These Frame Identifiers are listed in Table 4.5 for each of the commands. For some commands, additional payload data may also be transfered. As mentioned previously, the FCS will be generated by the PHY when the frame is sent.



Octets: 2	**1**	**(see 7.2.2.4.1)**	**0/5/6/10/14**	**1**	**variable**	**2**
Frame Control	Sequence Number	Addressing fields	Auxiliary Security Header	Command Frame Identifier	Command Payload	FCS
MHR				MAC Payload		MFR

Figure 4.13: MAC command frame format[7]

Because each function for sending commands receives the address information in different formats, some challenges are created. For every MAC command sent, it is necessary to generate the appropriate MHR using information provided by the calling primitive. In order to standardize the passing of header information between functions, a struct was declared as shown in Figure 4.14. This allows for a single pointer to be passed, as opposed to a large number of individual data elements. All MAC commands can then use a single function for header generation, as well as a single function for decoding headers upon receipt of a MAC command. Centralizing the generation and decoding functions greatly reduces code duplication while also reducing the chance of a bug.

[7]From IEEE Std 802.15.4-2006 Copyright 2006, by IEEE. All rights reserved.

```
typedef struct hdrMacCmdAddr_tag{
    uint8_t  srcAddrMode;
    uint8_t  *dstAddress;
    uint8_t  dstPanId[2];
    uint8_t  dstAddrMode;
    bool_t   ack;
    bool_t   panIDcomp;
} hdrMacCmdAddr_t;
```

Figure 4.14: Struct used to simplify passing address information for MAC command header generation

4.6.1 Data Request

The data request MAC command is called when the MAC layer when it receives the `MLME-POLL.request` primitive. It is responsible for generating the frame sent to the transceiver to poll the coordinator for available data. Since the current system implementation uses polling to initiate data transmissions, this command is critical. The settings for the address mode in the poll request are also used for the command, and an Ack is always required. The data request frame only uses one byte of payload, the command ID of 0x04.

The send functionality of this command was tested as part of `MLME-POLL.request` testing. This MAC command is called from the poll request, which supplies the command with pointers to address and security information needed to create the frame header. Breakpoints were used in conjunction with the variable view of the debugger to confirm that the message being passed to the PHY layer for transmission was correctly formatted. Testing the possible address combinations that can be sent to `MLME-POLL.request` ensured that the functionality was fully verified.

The receive functionality of this command has been written, but not tested. Once messages are being successfully received, functionality will be easy to verify by sending different address combinations. Verifying the functionality of this command will include verifying that message headers are being correctly parsed, a function shared by all receive commands.

4.6.2 PAN ID Conflict Notification

This command is sent by a device to the PAN coordinator when a PAN ID conflict is detected. The frame control settings used in the MHR field are specified by the standard and shown in Table 4.6. The PAN ID Conflict frame uses only one byte of payload, the command ID of 0x05.

Table 4.6: Frame Control Field of PAN ID Conflict Notification MAC command

Bits	Name	Setting
2-0	Fame Type	011
3	Security Enabled	0
4	Fame Pending	0
5	Ack. Request	1
6	PAN ID Compression	1
9-7	Reserved	-
11-10	Dest. Addressing Mode	11
13-12	Fame Version	00
15-14	Source Addressing Mode	11

The code to implement both the send and receive variations of this command have been written, but only send has been tested. Testing of the send command is trivial as the only information passed to it is a pointer to security information. Since security is not currently supported, there is only one possible variation of the command used at this time. The functionality of the command was tested by passing it the security level of 0 then using a break point to confirm that the message being passed to the PHY for transmission was properly formatted. Testing the receive variation will be trivial once the function used to parse received frame headers is tested and verified.

4.6.3 Orphan

This command is used by a device in the event that it loses synchronization with the coordinator. When a coordinator receives this command, an `MLME-ORPHAN.indication` message is sent to the NWK layer. The settings for the Frame Control field defined by the standard for this command are shown in Table 4.7. The orphan frame only uses one byte of payload, the command ID of 0x06. The orphan device procedure was described previously in Section 4.5.

Table 4.7: Frame Control field of Orphan MAC command

Bits	**Name**	**Setting**
2-0	Fame Type	011
3	Security Enabled	0
4	Fame Pending	0
5	Ack. Request	0
6	PAN ID Compression	1
9-7	Reserved	-
11-10	Dest. Addressing Mode	10
13-12	Fame Version	00
15-14	Source Addressing Mode	11

Both the send and receive function to implement this command have been written. However, only the send functionality has been tested so far. Like the PAN ID Conflict Notification command, only the security information is passed to the send, making it trivial to test. The same method described for testing the PAN ID Conflict Notification send was applied here. Testing the receive variation will be trivial once the function used to parse received frame headers is tested and verified.

4.6.4 Coordinator Realignment

The are two scenarios where the Coordinator Realignment command is used. The first scenario is when an Orphan notification command has been received, and the coordinator has confirmed the sender is part of its network. In this case it is being sent from the coordinator to the orphan device. The second case occurs when PAN configuration attributes have been changed. These settings can only be changed by a `MLME-START.request` being received from the NWK layer. As a result this command is always generated as part of the handling of the request in the MLME. In this case it is being broadcast from the coordinator to all devices on the network. The Frame Control settings specified by the standard for this command are dependent on the intended recipient, as shown in Table 4.8. For this command, additional payload bytes are appended after the command ID of 0x08. They contain the PAN configuration information needed by the receiving device to re-configure its PIB Table. If the command is sent to an orphaned device, the new short address will be part of this payload. If the command is being broadcast, the destination PAN ID will be set to 0xffff and the destination short address will be set to 0xffff in the addressing fields to indicate it is intended for all devices.

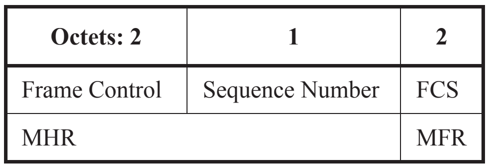
The Coordinator Realignment command is sent as part of the startup procedure, but PAN information is being hard coded at this time. As a result testing the functions which have been written to send and receive this command was low priority. In the future, testing of these functions can be done in much the same way as testing was completed for other MAC commands. The Coordinator Realignment command is passed multiple pieces of information in the send variation, so testing will require the function to be called multiple times for thorough verification of the parameter variations.

Table 4.8: Frame Control field of Coordinator Realignment MAC command

Bits	**Name**	**Setting**	
		Device	**Broadcast**
2-0	Fame Type	011	011
3	Security Enabled	0	0
4	Fame Pending	0	0
5	Ack. Request	0	0
6	PAN ID Compression	0	0
9-7	Reserved	-	-
11-10	Dest. Addressing Mode	11	10
13-12	Fame Version	00	00
15-14	Source Addressing Mode	11	11

4.7 Acknowledgment

Acknowledgment messages, or Acks, are used to confirm that information has been successfully received by the intended device. If the sender expects to receive an Ack, the `Ack.Request` bit of the Frame Control Field will be set to one when the message is transmitted. The frame format used when sending an Ack is shown in Figure 4.15. The format of the Frame Control Field was shown previously in Figure 4.10. The settings used for the Frame Control Field entries are shown in Table 4.9. The sequence number is set to sequence number of the frame being acknowledged. As always, the FCS is generated by the PHY transceiver hardware.



Octets: 2	**1**	**2**
Frame Control	Sequence Number	FCS
MHR		MFR

Figure 4.15: Acknowledgment frame format[8]

[8]From IEEE Std 802.15.4-2006 Copyright 2006, by IEEE. All rights reserved.

Table 4.9: Frame Control field of ACK message

Bits	Name	Setting
2-0	Fame Type	010
3	Security Enabled	0
4	Fame Pending	1/0
5	Ack. Request	0
6	PAN ID Compression	0
9-7	Reserved	-
11-10	Dest. Addressing Mode	00
13-12	Fame Version	00
15-14	Source Addressing Mode	00

It is possible that a device could have more than one pending Ack at any given time. As a result it is necessary to initialize a data structure which can be used to store a list of pending Acks. A singly linked list is sufficient for this purpose.

The 802.15.4 standard requires that when a device receives a message for which an Ack has been requested it must respond within `aTurnAroundTime` symbols. In order to ensure that this condition has been met when the sender receives the Ack, an expiry time is included in the pending Ack structure. Depending on the command with which the pending Ack is associated, the required action in the event that a pending Ack expires without a response from the receiver varies. In some cases the pending Ack is simply discarded. In others, a confirm primitive with a status of `NO_ACK` is generated and sent through the MLME_NWK SAP to the next higher layer. The associated action in the event of an expired Ack for each command type is shown in Table 4.10.

Table 4.10: Action taken when pending Ack expires

Command ID	**Registering Command**	**Action on Expiry**
0x01	Association request	none
0x02	Association response	generate `MLME-ASSOCIATE.confirm` with a status of `NO_ACK`
0x03	Disassociation notification	generate `MLME-DISASSOCIATE.confirm` with a status of `NO_ACK`
0x04	Data request	generate `MLME-POLL.confirm` with a status of `NO_ACK`
0x05	PAN ID Conflict notification	none
0x08	Coordinator Realignment	generate `MLME-COMM-STATUS.indication` with a status of `NO_ACK`
0x0A	MCPS Data request	generate `MCPS-DATA.confirm` with a status of `NO_ACK`

Chapter 5

PHY Software Design and Verification

The PHY layer is where the majority of deviations from the standard occur. Because the transceiver is physically separate from the controller, the SAPs between MAC and PHY layers are not practical. Instead, PHY primitives are directly called from the MAC layer. These primitives encapsulate the majority of the SPI bus communication necessary to execute the desired action. In essence, the SAPs have been replaced with the SPI bus and `IRQ` line which now form the interface to the transceiver. This new relation between MAC and PHY layers is shown in Figure 5.1.

Because of the extensive changes, the PHY driver implementation will be reviewed. The primitives described in the standard for both the PD and PLME cores of the PHY layer will then be compared to the actual implementation. The use of the `IRQ` line for system control, and how this control is used will also be reviewed. This proved to be one of the most difficult portions of the PHY implementation, and the challenges encountered will be discussed.

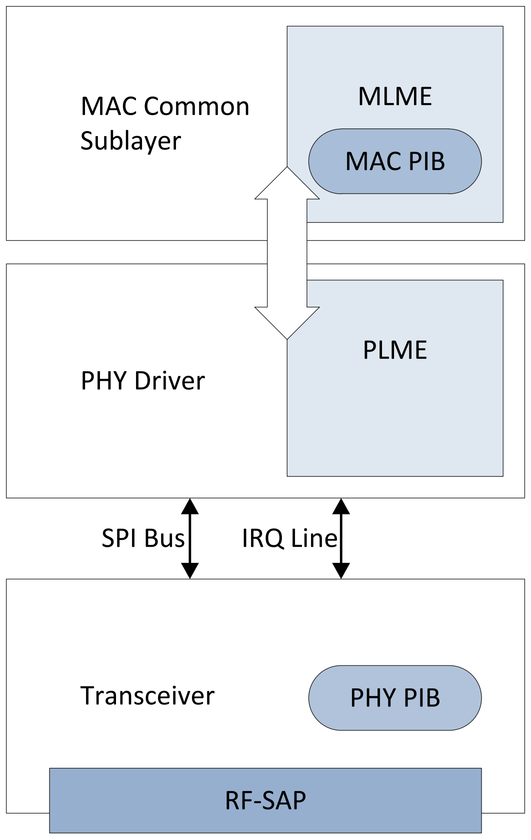


Figure 5.1: Revised PHY and MAC layer interface

5.1 PHY Driver

The PHY driver provided by Freescale was implemented using a state machine which mirrors that of the transceiver. For correct operation it is critical that the state information be kept in sync. Figure 5.2 shows the states used in this implementation. It also shows the signals necessary to transition between the states. In this case, time-triggered events are not used. It should also be noted that most events are configured to occur in stream mode. The exception is ED, which performs an ED scan. Since this mode can only be entered on PAN startup, it can be run with stream mode disabled. Due to timing considerations, CCA is configured to run with the transceiver in stream mode. The alternate procedure recommended by Freescale is followed in this case[16]. In all cases the device must return to the IDLE state before it can transition to one of the 4 action states.

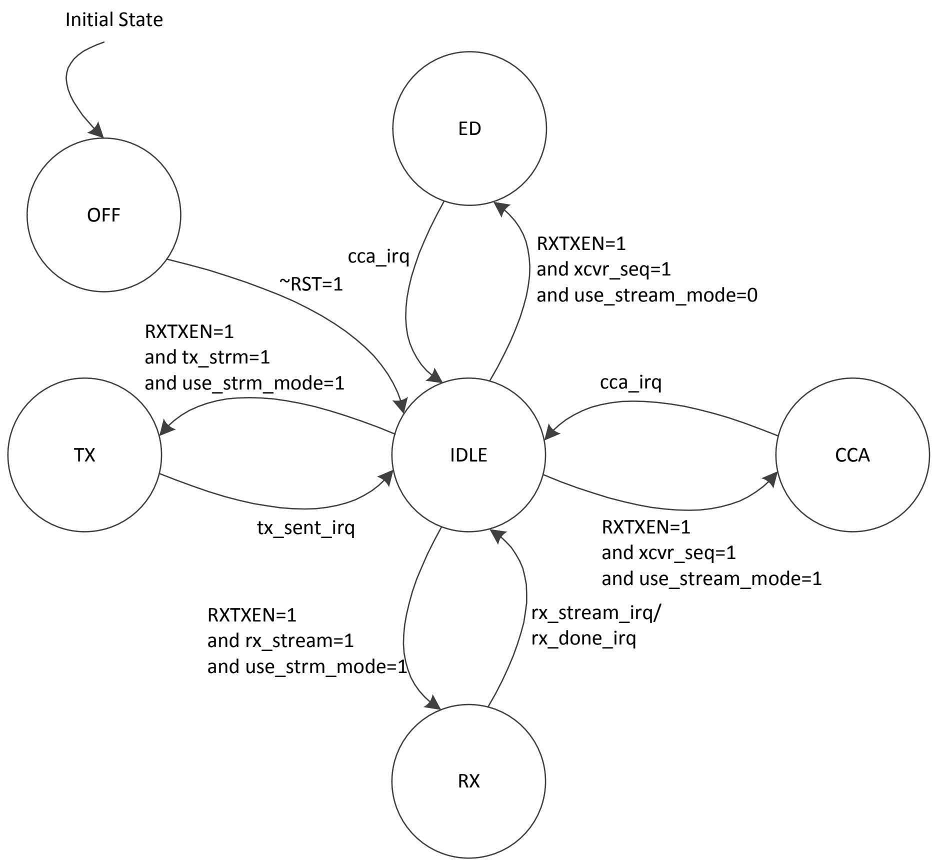


Figure 5.2: State diagram for stream mode PHY

In order to use the transceivers, some changes to the drivers were necessary. There are some peculiarities in the method used by Freescale to setup successive PHY actions. An action is defined by the transition from Idle to one of the active states, and the subsequent transition back to Idle. If the transceiver is not already in the process of executing an action, the desired action will be setup immediately. If an action is currently in progress, *mpfPendingSetup* will be set to the appropriate action but no changes will be made to the transceiver. When the action in progress completes, the status of *mpfPendingSetup* is checked. If it is not NULL, the appropriate action is executed. Note that this implementation only supports one pending action at a time. According to comments in the code, this has been done in order to free the processor for other tasks while waiting for the transceiver to complete an action. While this may seem advantageous, there are very few tasks undertaken by the transceiver during which time the processor can do other meaningful work. One of the longest actions to execute is the ED scan. However, these scans only occur in the coordinator while a channel is being selected for the network. Because it is still in the setup procedure, there are no other actions for the controller to complete. Another change from the Freescale implementation is the removal of the use of a timer to initiate each action. Instead they are started when `RXTXEN` is set high. This is achieved by not setting the `tmr_trig_en` bit when configuring the action.

5.2 Primitives

The primitives in the PHY layer operate on the same basic principles as those in the MAC layer. Because the PHY layer is focused on low level functions, there are far fewer primitives compared to the MAC layer. The primitives defined by the standard for both the PD-SAP and PLME-SAP must be explored in detail as their implementation as defined is not possible. In this implementation equivalent functionality is achieved through other means.

5.2.1 PHY Data (PD)

The PD core of the PHY layer is solely responsible for data transmission and receipt. This functionality is achieved with the use of a single primitive family, `PD-DATA`. The primitives within this family, as well as their state in the current implementation are shown in Table 5.1.

Table 5.1: PD-SAP Primitives

Primitive	Function Name	Status	File Location
PD-DATA.request	PhyPdDataRequest	T	Data.c
PD-DATA.confirm	-	NA	-
PD-DATA.indication	-	NA	-

NA-Not Applicable, T-Tested

The `PD-DATA.request` primitive source code was provided by Freescale. However, when it was compared to the 802.15.4 standard it was found to be non-compliant. Specifically, the standard calls for `PD-DATA.confirm` to be sent to the originating MAC layer when the data frame is successfully transmitted. If the transmission is not successful, the `PD-DATA.confirm` primitive is used to return an error code. In the Freescale code this primitive is of type void and never called. The message sequence chart of a successful data transmission according to the standard is shown in Figure 5.3 below.

Because the PHY in this implementation is a separate piece of hardware, the SAP interface between the MAC and PHY is impractical, as previously discussed. Instead, the `IRQ` line can be used in place of the `PD-DATA.confirm`, with bit 14 of the `IRQ_Status (0x24)` register indicating the completion of a stream transmission. An interrupt is also generated after each word has been sent, telling the MAC layer to load the next word. This is indicated using bit 6 of the `IRQ_Status (0x24)` register.

The purpose of the `PD-DATA.indication` is to notify the receiver's MAC layer that a message has been received and needs to be processed. Once again, this function is realized using the `IRQ` line. Since a stream receive is used, an interrupt is generated after every word and indicated using bit 7 of the `IRQ_Status (0x24)` register. When the full message has been received bit 13 will also be set.

The first step is achieving a successful data transmission was to implement and test

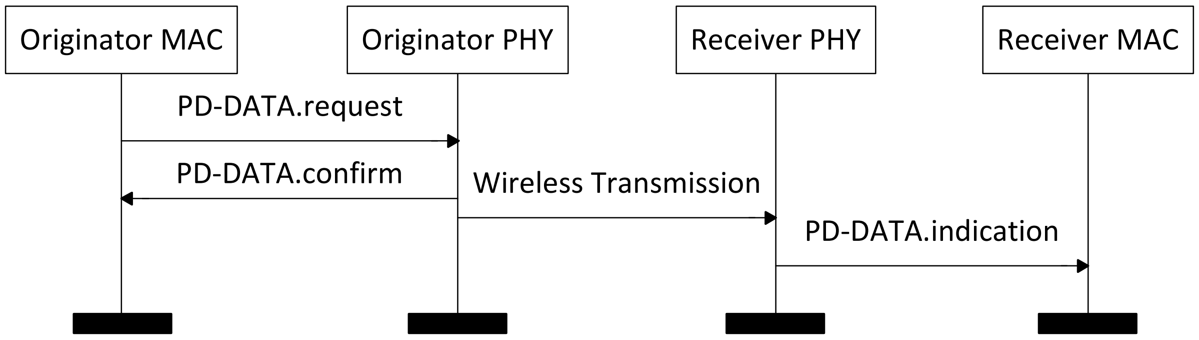


Figure 5.3: Sequence chart for a successful `PD-DATA` transaction

the stream transmit functionality of the transceiver. While the PHY routines for doing so were provided as source code with the Freescale implementation, their use result in the first word being successfully transmitted and generating the expected interrupt. This interrupt would then be serviced by sending the next word to the transceiver, with the expectation that an interrupt would be generated once it was sent. However, the second interrupt would never occur. Instead, PLL Lock errors and stream errors were seen. In order to debug the issue, the transceiver documentation was consulted[16]. By examining the suggested procedure for initiating a stream transmission it was noted that there were discrepancies with the provided code. As a result the `PD-DATA.request` was modified to conform with the documentation. It was then tested again and confirmed to be functioning as expected. Functionality was confirmed by examining the SignalTap II output of the SPI bus, a sample of which is shown in Figure 5.4 for a successful stream transmission. It was configured to trigger on the falling edge of the `IRQ` signal, which caused the transfer of all data after the first word to be recorded. It was possible to verify that the first word and the length of the data frame to be transmitted were being correctly written using a separate SignalTap II recording. While this only confirms that the data is being transfered between the transceiver and PHY driver as expected, it was all the testing that could be done. The equipment needed to verify the RF signal being transmitted was not available. Rather, successful RF transmission will be confirmed by successfully receiving transmitted frames.

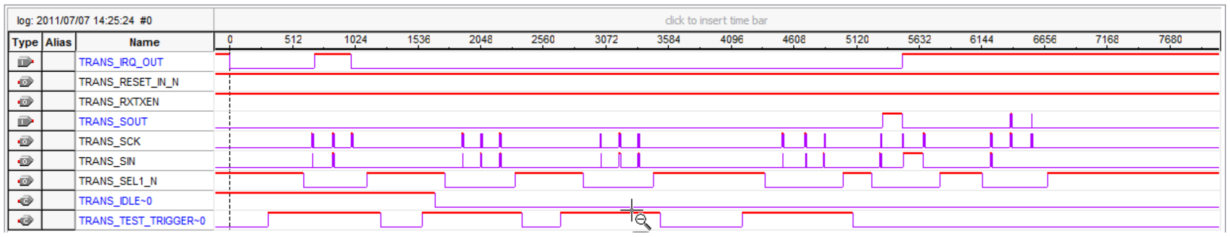


Figure 5.4: SignalTap II Output showing successful stream transmission

5.2.2 PHY Layer Management Entity (PLME)

The majority of the PLME primitives defined by the standard are not implemented in software. Rather, their intended functionality is realized within the transceiver itself and abstracted from the user, or serves no useful purpose due to the necessary interface changes between the PHY and MAC layers. Table 5.2 shows the PLME primitives that are defined by the standard. The status of each in the current implementation is also shown. Those still required in this implementation are described in more detail below.

The `PLME-CCA.request` primitive is used to trigger a clear channel assessment before attempting to send data over the network, in order to decrease packet collisions. Since only a basic network was used for testing, this was of low priority. The code is written, but no testing has been done at this time.

The `PLME-ED.request` primitive is responsible for performing the setup necessary to execute a single ED measurement. In this implementation that means setting `xcvr_seq`$=1$ and `RXTXEN=1`. It is not necessary for the routine to set `use_stream_mode`$=0$ as this is already done in the PHY initialization routine. According to the standard, the primitive is responsible for checking if the transceiver is disabled or the transmitter is enabled, returning an error code to the MLME in either case via the `PLME-ED.confirm` primitive. However, because the changes to MAC/PHY interface have resulted in tighter coupling of

Table 5.2: PLME-SAP Primitives

Primitive	**Function Name**	**Status**	**File Location**
PLME-CCA.request	PhyPlmeCcaRequest	W	CcaEd.c
PLME-CCA.confirm	-	W	-
PLME-ED.request	PhyPlmeEdRequest	T	CcaEd.c
PLME-ED.confirm	PhyPlmeEdConfirm	T	PhyMac.c
PLME-GET.request	-	NA	-
PLME-GET.confirm	-	NA	-
PLME-SET-TRX -STATE.request	PhyPlmeTxRequest	T	SetRxTxState.c
	PhyPlmeRxRequest	T	SetRxTxState.c
	PhyPlmeForceTrxOffRequest	W	SetRxTxState.c
PLME-SET-TRX -STATE.confirm	-	NA	-
PLME-SET.request	PhyPlmeSetCurrentChannelRequest	T	GetSet.c
PLME-SET.confirm	-	NA	-

NA-Not Applicable, NI-Not Implemented, T-Tested, W-Written

the layers, the MLME can ensure the PHY is in the IDLE state before it calls the PLME-ED.request function, making this check unnecessary. The `PLME-ED.confirm` primitive is still used, but its functionality has been reduced. According to the standard it is to pass both a status and the detected energy level to the MLME. In this implementation it is called when the `cca_irq` is received and used to pass only the energy level.

The `PLME-GET` and `PLME-SET` primitive are intended to provide the MAC layer with access to the variables stored in the PHY PIB table. This is analogous to the functionality provided to the NWK layer by the `MLME-GET` and `MLME-SET` primitives. However, the PHY PIB information is not held in a table structure as the MAC PIB. Instead, some are constants defined by the transceiver's implementation while others are controlled by setting register values via the SPI interface. This renders explicit implementation of the PLME-GET primitives unnecessary as a simple SPI read to the correct register or consultation of the transceiver reference manual is all that is needed. While the PLME-SET.request primitive is still used, its functionality has been modified. According to the standard it should provide the ability to set any of the four entries in the PHY PIB table which are not read-only attributes. Of these four, only setting the phyCurrentChannel requires more than a SPI register write. In this case, the channel number must be correlated to the appropriate settings for the LO1_Int_Div (0xF) and LO1_Num (0x10) registers, which set the numerator and divisor of the local oscillator to produce the correct RF frequency. This function has also been implemented to set the PA_Lvl (0x12) register, which controls the power level of the transmit signal. Since this whole procedure is considerably more complex than a single SPI write, it is logical to use a function for implementation.

Similarly, in the case of the PLME-SET-TRX-STATE primitive, while the described functionality is still present, it is not implemented in the stated manner. According to the standard, the PLME-SET-TRX-STATE.request primitive is passed an enum to indicate which mode it should set the transceiver to. Instead of passing this enum, the functionality has been implemented using three different functions, one for each of the modes that can be selected.

5.3 Interrupt Service Routine (ISR)

The interrupt service routine, and the correct functioning thereof, is one of the most crucial pieces in making the system work. The transceiver relies on a single interrupt line to the controller, IRQ, which is active low. When the line goes low, the controller is expected to read the IRQ_Status (0x24) register. Each of the 16 bits in this register are used to indicate the occurrence of a different event. There are, however, some bits whose meaning changes depending on the current configuration of the transceiver. Specifically, bits 14, 13, 7 and 6 are used to indicate different events depending on whether the transceiver is operating in packet mode or streaming mode[16]. In this implementation the transceiver is set to operate in streaming mode once the ED scans in the startup procedure are complete. The transceiver then stays in this mode until it is reset. As a result, the code need only deal with the interpretation of these bits in streaming mode. The adaptation of the existing Freescale code to the FPGA with respect to interrupts can be considered in two stages: the changes necessary to setup the interrupt service routine, and the changes necessary to achieve the expected behaviour from the routine.

In terms of setup, the Freescale code included a series of functions used to abstract the register operations necessary in determining which interrupt had been triggered, acknowledging an interrupt, masking an interrupt etc. The macros for doing so were adapted where appropriate to Altera I/O calls. The calls necessary to register the interrupt handler with the Nios II vector table were also added.

Making the necessary configuration changes proved to be much simpler than achieving the expected functionality. When the software is started, the first interrupt received from the transceiver indicates that it has been restarted and is now in the *cIdle* state. The next interrupt is of more interest in attempting to verify the functionality of the ISR in general, as well as its response to the the ED scan routine.

It was found that several changes were necessary in order to have the correct ISR subroutine called, as well as to stop the routine from hanging. First, it was found that the routine used to setup the transceiver for the scan was also responsible for determining which of the ISR subroutines would be called based on the current value of *gIsrFastAction*. However, in the Freescale code they were assigning the associated routine to *gIsrPending-FastAction*, despite the fact that there was no subsequent routine to update *gIsrFastAction* to the value of *gIsrPendingFastAction* if an action was not already in progress. As a result all PHY setup functions were changed to assign the associated routine directly to *gIsrPendingFastAction*. The Freescale implementation also relied on a timer to trigger the start of actions, which has been disabled in this implementation. Both of these code changes are illustrated in Table 5.5 for the case of setting up an ED scan, as shown by the code which has been commented out.

```
void SetupPendingEd(void)
{
    uint16_t command = cCCA_MASK | cCCA_ED | SEQ_C; // | cTMR_TRIG_EN;
    SetupAction(command);
    MC1319xDrv_RxAntennaSwitchEnable();
    mPhyTxRxState = cRxED;
    gIsrFastAction=DoFastEdEof;
    \\gIsrPendingFastAction=DoFastEdEof;
}
```

Figure 5.5: The revised code for configuring an ED scan is shown, with changes commented out.

According to the Freescale MC13192 reference manual[16] the interrupt line is connected to an open drain device,as expected from an active low signal. There is a programmable $40\,\mathrm{k}\Omega$ pull-up resistor in the transceiver chip, which is controlled by `GPIO_Data_Out (0x0C)` bit 7. An optional external pull-up resistor can also be used, as long as it is $> 4\,\mathrm{k}\Omega$. According to the reference manual, this interrupt can be serviced every $6\,\mu$s when the load is $< 20\,$pF. As a result one possible cause of slow de-assertion is a small pull-up resistance.

In order to verify the pull-up resistance of the `IRQ` line, it was measured with a multimeter. It was found to measure just under $8\,\mathrm{k}\Omega$, which suggest that there is an external pull-up resistor on the line. Using the basic equation for adding parallel resistances, it was possible to estimate this external pull-up to be a $10\,\mathrm{k}\Omega$ resistance, as shown in Equation 5.1. This was then verified by disabling the $40\,\mathrm{k}\Omega$ pull-up and repeating the measurement with the multi-meter, confirming the predicted value. A sample of the response time observed is shown in Figure 5.6. Using SignalTap II tools to count the number of cycles and converting to time showed it took 762 μs before the `IRQ` line was reset in this particular case. This is two orders of magnitude longer than expected, and is clearly unacceptable for normal operation of the system. By zooming in on the same trace it is also possible to confirm that the ISR is functioning as expected. When the `IRQ` line goes low, it is followed after a short delay by a read to register `0x2D`, as shown in Figure 5.7. This is the register which holds the result of the ED measurement. While not shown by the figure, this read is followed by a read to register `0x24` to de-assert the `IRQ`. Thus it appears that the procedure described by the Freescale reference manual is being followed[16].

$$\left(\frac{1}{10\,\mathrm{k}\Omega} + \frac{1}{40\,\mathrm{k}\Omega}\right)^{-1} = 8\,\mathrm{k}\Omega \tag{5.1}$$

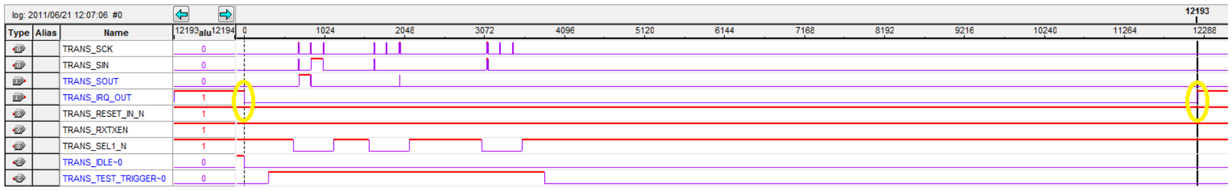


Figure 5.6: An example of the ISR routine triggered in response to the completion of an ED scan. Note the extended length of time before the IRQ line is de-asserted.

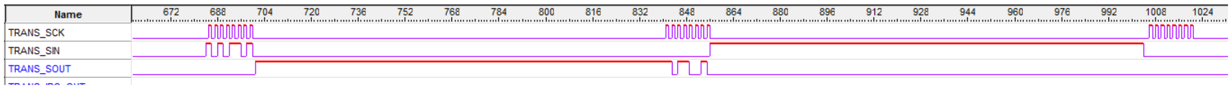


Figure 5.7: A closer view of reading registers 2D to obtain the results of the scan.

After various software based attempts to decrease the response time of the `IRQ` line de-assertion, alternative methods of controlling the system were considered. One possibility is to rely on the `IDLE` line from the transceiver to indicate when the ED scan has been completed, instead of relying on `IRQ` being asserted. According to the Freescale documentation[16], the active-high `IDLE` line will be asserted while the device is actively executing an action, whether it be an ED scan, CCA, transmit or receive. The ED scan is the first of these to be run during program execution, so it will be used as an example here.

Figure 5.8 shows the SignalTap II output of the successive scans being triggered by the falling edge of the `IDLE` signal. While the issue of only the first (not seen in figures) and second scans being run with `IRQ` based timing has been eliminated, another issue is apparent. During the first recorded ED scan the `IDLE` signal is asserted for a considerable length of time, approximately 336 μs. However, it is clear that after the second recorded ED scan is setup, the `IDLE` signal is only high for a very short time, approximately 2 μs, as indicated by 1. Closer examination of the signals shows that this is the result of glitches on the IRQ line when it is transitioning back to its de-asserted state. Because *gIsrFastAction* has been configured to call the routine which handles saving the results of the current ED scan and triggers setup of the next, scans are now effectively being triggered by two conflicting sources. The solution is to set *gIsrFastAction* to *DummyFastIsr*. Hence the ISR will now clear `IRQ_Status (0x24)` without interfering with the current ED scan if it is triggered.

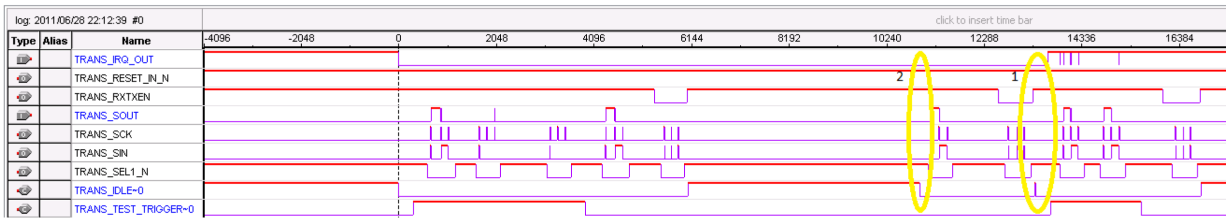


Figure 5.8: Using the IDLE signal to trigger successive ED scans

The expected behaviour of the system is that an interrupt will be generated when the ED scan completes. From Figure 5.8, it is clear that the first recorded ED scan is completing while the `IRQ` line is still low. Hence, even though `IRQ_Status(0x24)` has already been read, a new interrupt bit is being set before the IRQ line is de-asserted, as highlighted by 2. This is easily handled by adding a read of the `IRQ_Status(0x24)` in the code that is executed when the `IDLE` signal returns to zero. The SignalTap II output that results from this change is shown in Figure 5.9. It is clear that even though the ISR is still being triggered, it is no longer causing a premature termination of a running ED scan as the IDLE signal remains high for the expected length of time.

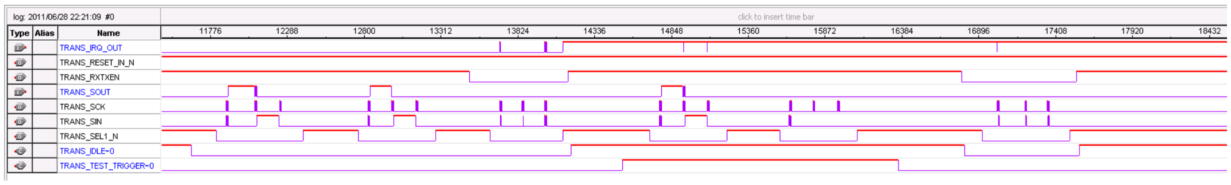


Figure 5.9: IDLE line controlled ED scan with spurious IRQ

While using the IDLE line as a trigger did enable the ED scan to be successfully executed, there is potential that critical interrupts are being ignored. One critical interrupt is the pll_lock_irq. While the triggering of this interrupt is expected to be rare, failure to correctly handle it can cause the system to hang. Should this interrupt occur during while an action is being executed, the action will be aborted immediately. In the case of the ED scan, this could mean the scan is terminated before the scanning time has elapsed. It may also stop the PHY driver from setting the current transceiver mode back to cIdle from cRxED, resulting in a loss of synchronization as the transceiver has reverted back to Idle mode. It is clear that finding the cause of the slow IRQ line is critical.

While the value of the IRQ line pull-up resistor has been considered as a source of the problem and eliminated, it was revisited and a schematic of the entire signal path was drawn. This schematic is shown in Figure 5.10. When reviewing the schematic for the MC13192 daughter card [38]that was used, it was noted that the tri-state buffer connection on the IRQ line differed from those of the other output signals. Specifically, the input was also connected to the active-low enable. As a result, when the IRQ line transitions from a 0 to a 1, the buffer is disabled. This leaves the FPGA side of the line in a high-Z state, as the included pull-ups are both on the transceiver side of the buffer. To solve this problem a 10 kΩ pull-up resistor was added to the FPGA side of the buffer. Subsequent testing with SignalTap II showed that this did indeed result in `IRQ` line behaviour consistent with the transceiver documentation. With the `IRQ` now behaving as expected, it was possible to test stream transmit and steam receive functionality.

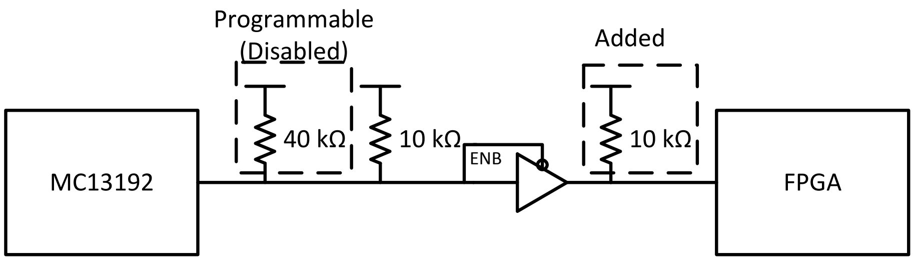


Figure 5.10: Schematic Diagram of IRQ Line

A combination of SignalTap II and the software debugging environment were used to confirm the functionality of the PLME primitives. In many cases it was possible to use the special test signal line that had been added to the processor to trigger the recording of the desired data. From this data the SPI bus transactions were decoded, confirming the proper transactions were taking place. In the case of `PLME-GET` and `PLME-SET`, it was possible to read a register containing information normally found in the PHY PIB, modify the value using the set function, and finally perform a second read to confirm the change had indeed taken place. The result of both reads was printed to the console, allowing multiple registers to be checked very quickly.

Chapter 6

Analysis and Results

The primitives and functions discussed previously in Chapter 4 and Chapter 5 are the building blocks used to implement the more complex core functions of the system. In this initial implementation, two core functions have been implemented and tested: ED scanning and polling-based data transactions. The stream transmit and stream receive functionality are used in combination with one another to support polling-based data transactions, which allows both core functions to be tested simultaneously. In both cases, the actual implementation varied from the message sequence that has been defined by the standard. These variations, as well as the justification for them, are discussed in detail for each function.

6.1 Energy Detection (ED) Scan

As discussed previously in Section 4.5, the ED scan is one of four defined scan types. This particular one is only used by a coordinator on startup when it is attempting to determine the best channel on which to operate the network. The sequence of message calls to be used according to the standard is shown in Figure 6.1. As discussed before, the well defined SAP type interface between the PHY and MAC layers is not practical due to timing constraints as well as the physical interface. As a result, calls to PHY primitives are carried out synchronously and there is no need to use confirm primitives. One exception to this is the PLME-ED.confirm primitive. Since the transceiver generates an interrupt when the scan completes, it is called by the interrupt handler and sets a global variable used as a handshake signal to tell the `MLME-ED.request` handler that the current scan is complete. Multiple scans are executed on a single channel until the time alloted for scanning has elapsed, with the highest reading being saved as the value for that channel. This process is then repeated on each of the other channels selected for scanning. When all scans are complete, the array containing the results is passed back to the network layer as part of the `MLME-Scan.confirm` message. It is then up to the network layer to process the results and determine the best channel to use for the network. The message sequence chart for the function as it has actually been implemented is shown in Figure 6.2.

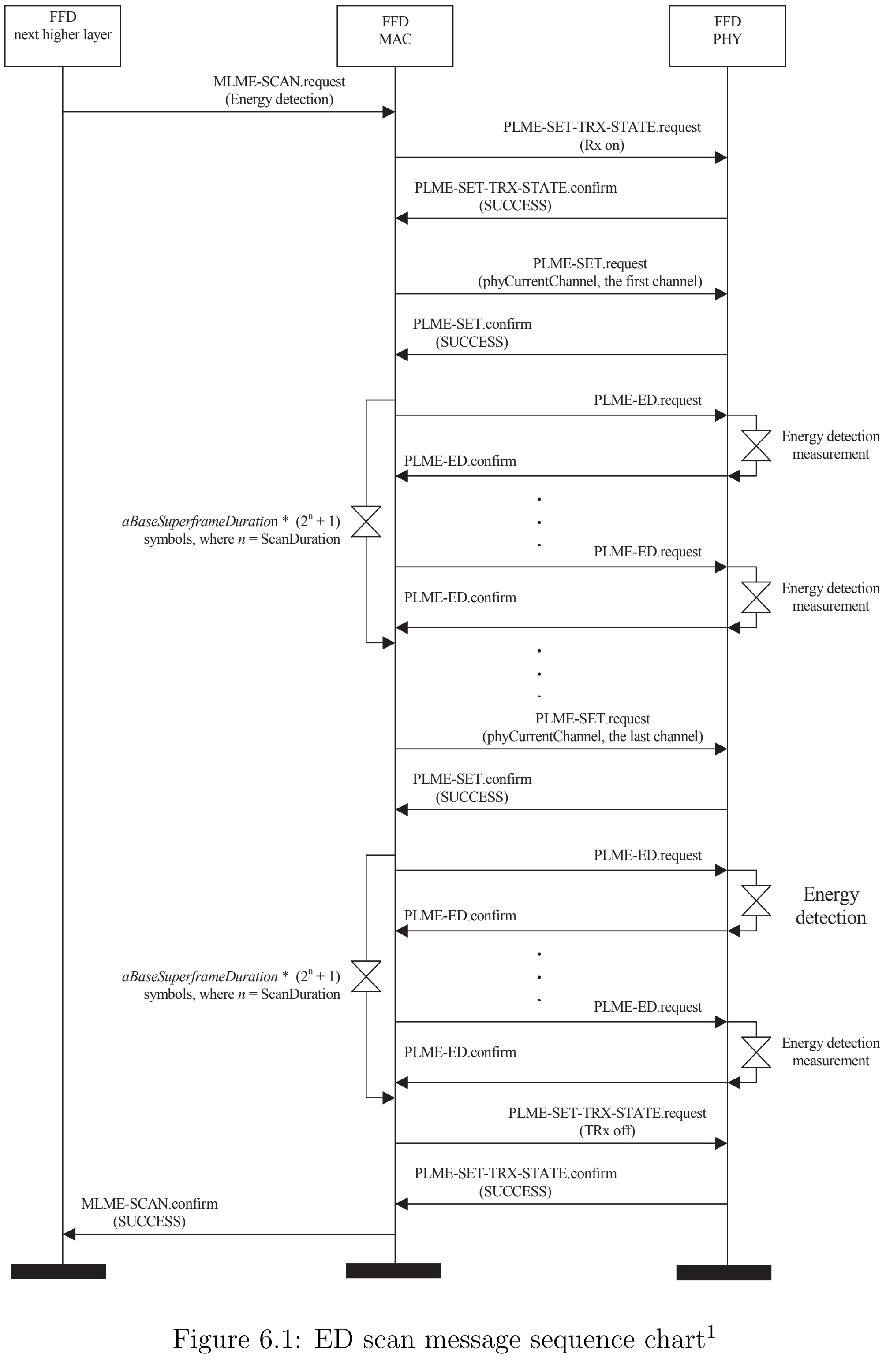

FFD
next higher layer
FFD
MAC
FFD
PHY
MLME-SCAN.request
(Energy detection)
PLME-SET-TRX-STATE.request
(Rx on)
PLME-SET-TRX-STATE.confirm
(SUCCESS)
PLME-SET.request
(phyCurrentChannel, the first channel)
PLME-SET.confirm
(SUCCESS)
PLME-ED.request
Energy detection measurement
PLME-ED.confirm
aBaseSuperframeDuration * (2^n + 1) symbols, where n = ScanDuration
PLME-ED.request
Energy detection measurement
PLME-ED.confirm
PLME-SET.request
(phyCurrentChannel, the last channel)
PLME-SET.confirm
(SUCCESS)
PLME-ED.request
Energy detection
PLME-ED.confirm
aBaseSuperframeDuration * (2^n + 1) symbols, where n = ScanDuration
PLME-ED.request
Energy detection measurement
PLME-ED.confirm
PLME-SET-TRX-STATE.request
(TRx off)
PLME-SET-TRX-STATE.confirm
(SUCCESS)
MLME-SCAN.confirm
(SUCCESS)

Figure 6.1: ED scan message sequence chart[1]

[1]From IEEE Std 802.15.4-2006 Copyright 2006, by IEEE. All rights reserved.

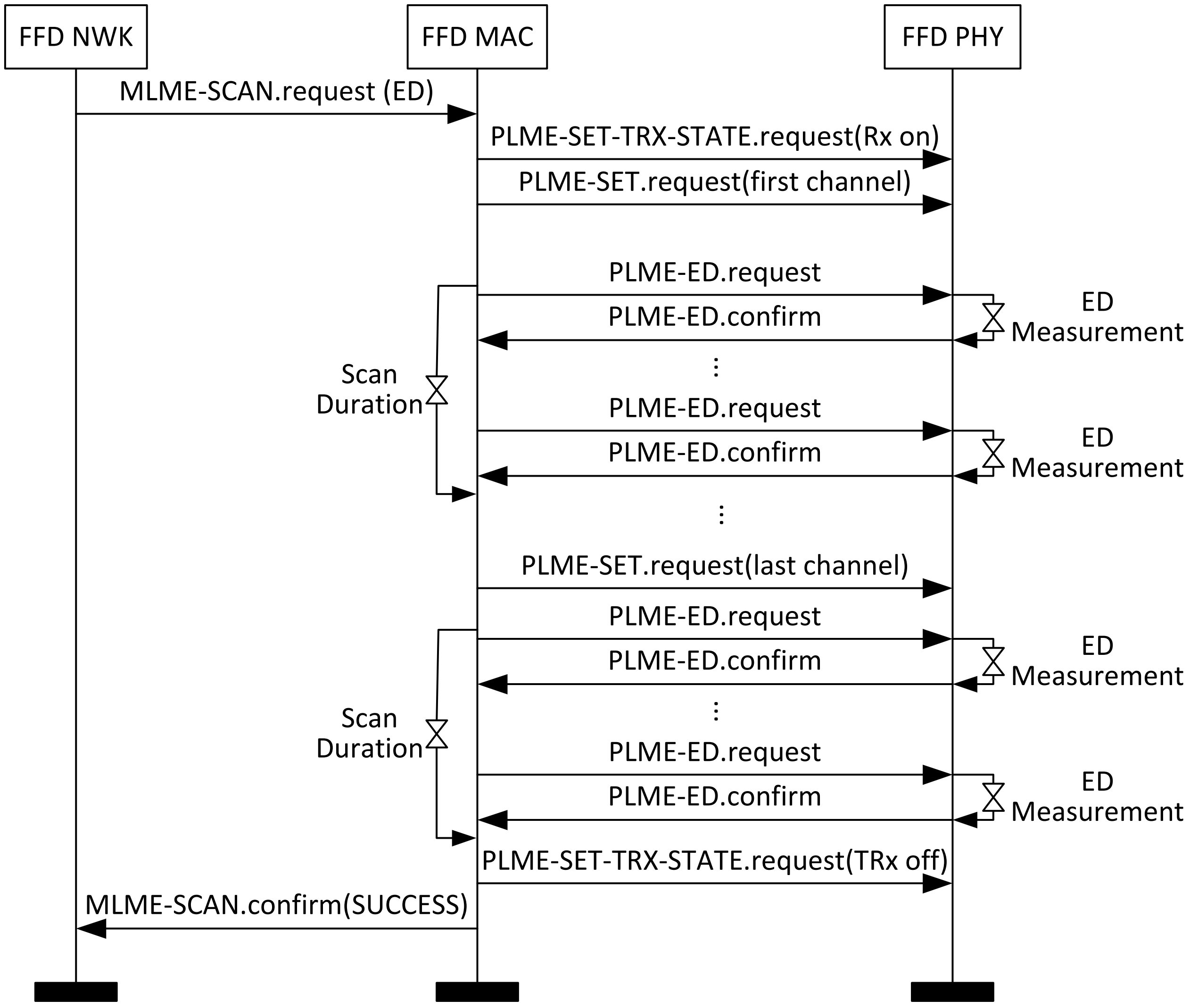


Figure 6.2: Message sequence chart of an ED scan as implemented

6.2 Polling Based Data Transmission

The execution of a polling-based data transmission involves primitives from both the MLME and MCPS cores of the MAC, as well as involvement from the NWK and PHY layers on both the device and the coordinator. This makes it an excellent candidate for testing the basic functionality of this implementation. This communication mechanism is often used in sensor networks where the device periodically polls the coordinator to determine if there is any new data available for it. Available data, if any, is then transmitted to the device. This arrangement allows the device to go into a power saving mode between checks. This is advantageous in situations where low power use is critical, such as a battery-powered device. The alternative is to have the device enabled as a receiver whenever it is idle. This allows the coordinator to send new data to the device as soon as it becomes available. While this decreases the latency of new data being received and processed by the device, it also increases the power consumption of the device considerably. While this arrangement was not tested by the research presented here, it requires only changes to the application in the NWK layer. As a result, there is no reason to believe the MAC/PHY layers would not be fully functional in this scenario as well.

For the purposes of testing this implementation, the NWK layer of the device was configured to enter an infinite loop after startup, where the following procedure is repeated approximately every 0.5 s. This loop sends a `MLME-POLL.request` from the NWK layer to the MLME. The MLME handles the request by issuing a Data Request MAC command, which is passed to the PHY for transmission over the RF medium. This command is then received by the coordinator PHY, which passes the command to the coordinator's MAC layer for processing. The MAC layer then checks the list of pending data frames for any addressed to the requesting device. If one is found, an Ack is sent with Frame Pending = 1. If there is no pending data for the device, the Ack is sent with Frame Pending = 0. When the polling device receives the Ack, it compares it to its list of pending Acks. When the match is found, the `MLME-POLL.confirm` primitive is sent to the NWK layer, with a status of *NO_DATA* if Frame Pending = 0. If there is data, the MLME waits until it has been received and sends the Ack to the coordinator before issuing the `MLME-POLL.confirm` primitive with a status of *SUCCESS*. The data itself is passed to the NWK layer via the `MCPS-DATA.indication` primitive. The message sequence chart for the procedure described is shown in Figure 6.3.

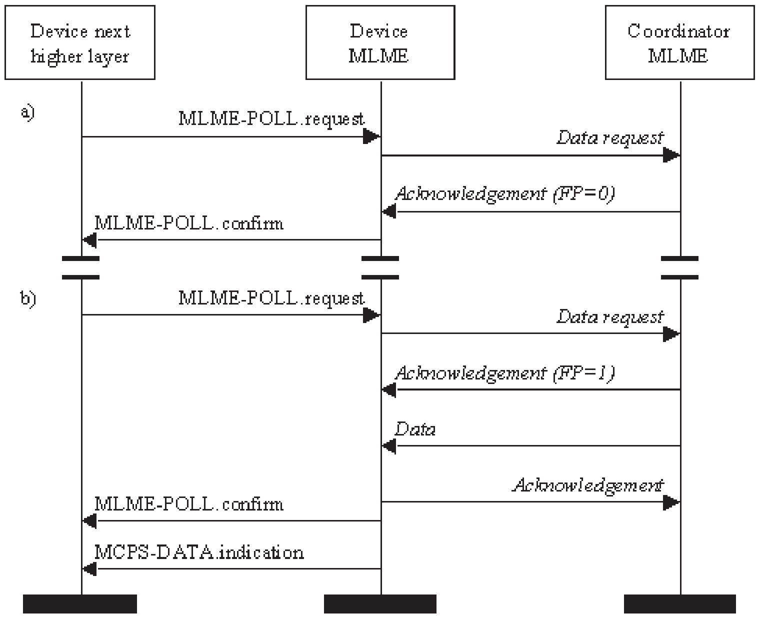


Figure 6.3: Message sequence chart for requesting data from the coordinator[2]

The coordinator has been configured to generate a new pending message for the device every time the current one is transmitted in response to the poll. The data payload of this message is only one byte. It is the value of a counter that is incremented each time a new message is generated, making it possible to observe the number of polling transactions that have taken place. For the data to be available in the pending message list, the `MCPS-DATA.request` must be passed from the NWK layer to the MLME. The *txOptions* settings tell the MLME to queue the message instead of transmitting it right

[2]From IEEE Std 802.15.4-2006 Copyright 2006, by IEEE. All rights reserved.

away. Once a Data Request MAC command is received for data in the pending list, the MCPS commences the transmit procedure message sequence shown in Figure 6.4 for the coordinator and Figure 6.5 for the device.

For various reasons, some of the primitives and procedures specified by the standard are not used in this implementation. For the purpose of the initial testing of a simple network, the CCA calls were not used. However, they could easily be added to the current procedure at a later date. The sequence chart for the full polling procedure as it is currently implemented is shown in Figure 6.6 for the device and Figure 6.7 for the coordinator.

Achieving successful stream transmissions proved to be a much more complex and time consuming process than originally thought. As a result, despite all the code needed to execute the polling procedure described above being written, it has not been fully tested. This is a result of stream receives not being successfully executed at this time. The message sequence has been fully verified up to the transmission of the first MAC command.

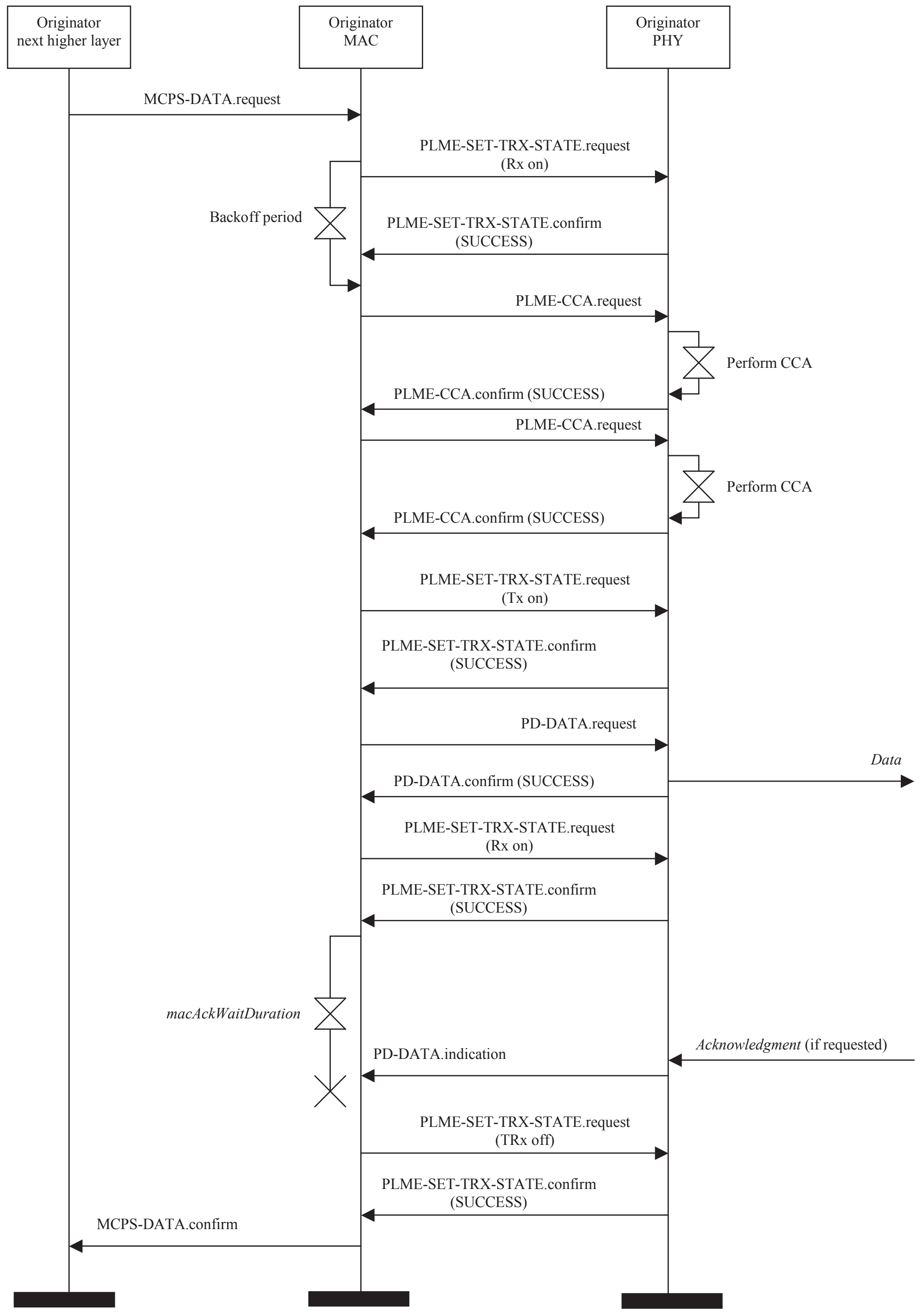

Originator next higher layer
Originator MAC
Originator PHY
MCPS-DATA.request
PLME-SET-TRX-STATE.request (Rx on)
Backoff period
PLME-SET-TRX-STATE.confirm (SUCCESS)
PLME-CCA.request
Perform CCA
PLME-CCA.confirm (SUCCESS)
PLME-CCA.request
Perform CCA
PLME-CCA.confirm (SUCCESS)
PLME-SET-TRX-STATE.request (Tx on)
PLME-SET-TRX-STATE.confirm (SUCCESS)
PD-DATA.request
Data
PD-DATA.confirm (SUCCESS)
PLME-SET-TRX-STATE.request (Rx on)
PLME-SET-TRX-STATE.confirm (SUCCESS)
macAckWaitDuration
Acknowledgment (if requested)
PD-DATA.indication
PLME-SET-TRX-STATE.request (TRx off)
PLME-SET-TRX-STATE.confirm (SUCCESS)
MCPS-DATA.confirm

Figure 6.4: Data transmission message sequence chart - originator[3]

[3]From IEEE Std 802.15.4-2006 Copyright 2006, by IEEE. All rights reserved.

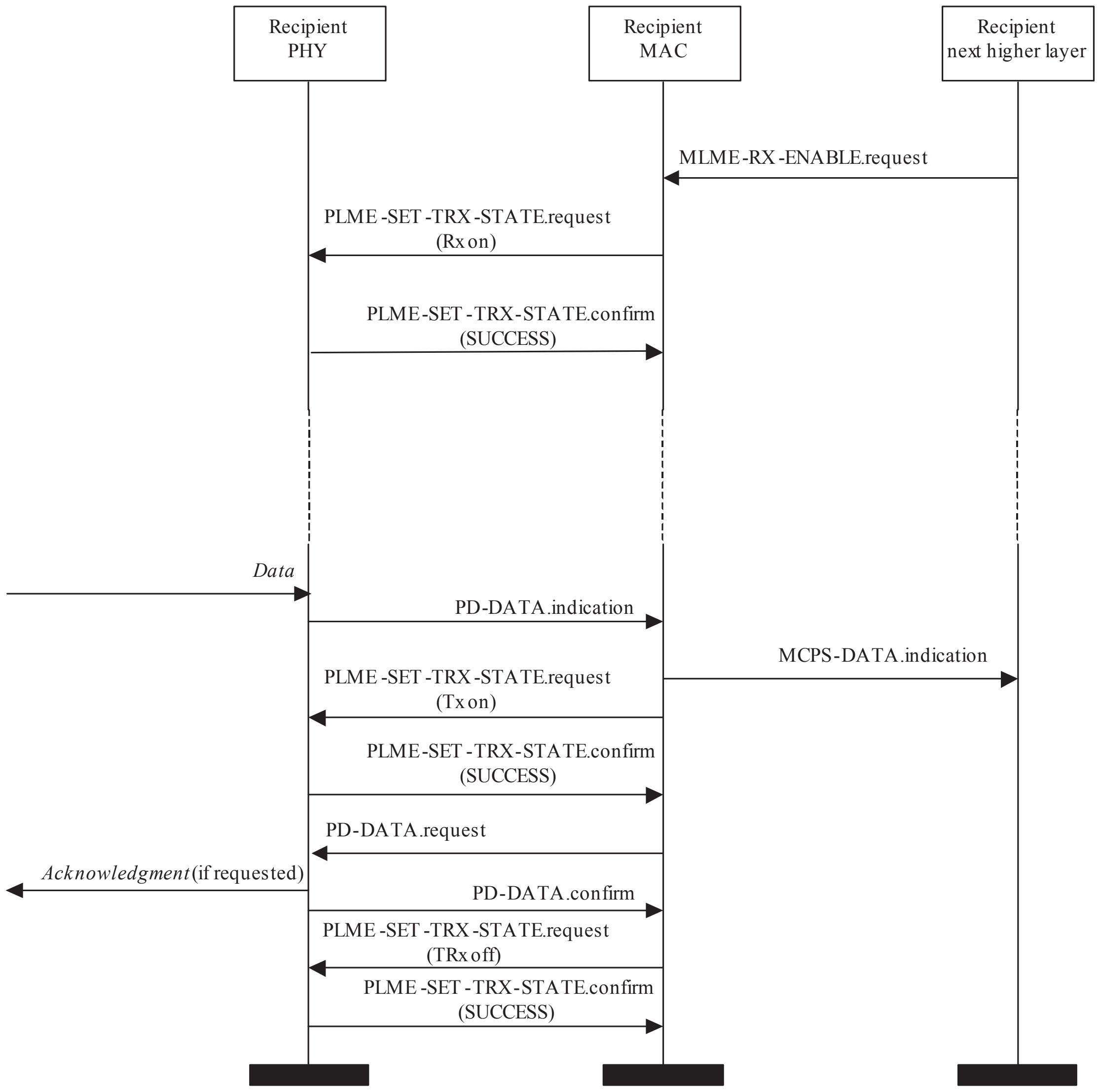

Recipient PHY
Recipient MAC
Recipient next higher layer
MLME-RX-ENABLE.request
PLME-SET-TRX-STATE.request (Rx on)
PLME-SET-TRX-STATE.confirm (SUCCESS)
Data
PD-DATA.indication
MCPS-DATA.indication
PLME-SET-TRX-STATE.request (Tx on)
PLME-SET-TRX-STATE.confirm (SUCCESS)
PD-DATA.request
Acknowledgment (if requested)
PD-DATA.confirm
PLME-SET-TRX-STATE.request (TRx off)
PLME-SET-TRX-STATE.confirm (SUCCESS)

Figure 6.5: Data transmission message sequence chart - recipient[4]

[4]From IEEE Std 802.15.4-2006 Copyright 2006, by IEEE. All rights reserved.

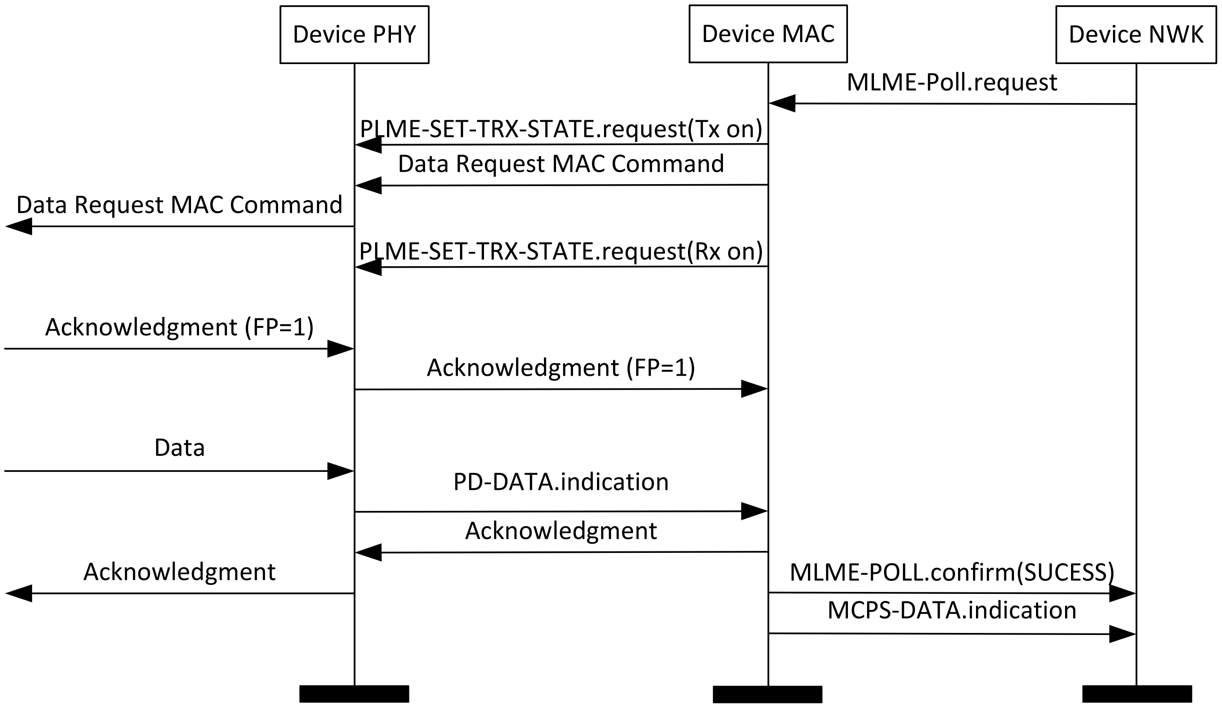


Figure 6.6: Polling message sequence chart - device as implemented

6.3 Resource Usage

One of the main goals of the standard is to support systems with limited resource capabilities. For example, the sensors used as end devices generally have limited memory and processor capabilities. Given this, the resource usage of the current system can be reviewed from both a hardware and software perspective. The unused resources available are also of interest as they will determine the feasibility of incorporating additional functionality in the future.

As discussed previously in Section 3.2, an Altera DE2 board with a Cyclone II 2C35 provides the hardware support for the system. The hardware, namely the Nios II processor and a supporting PLL, was compiled for two cases. The first case includes all the interfaces, as well as the SignalTap II support needed for debugging. In order to more accurately

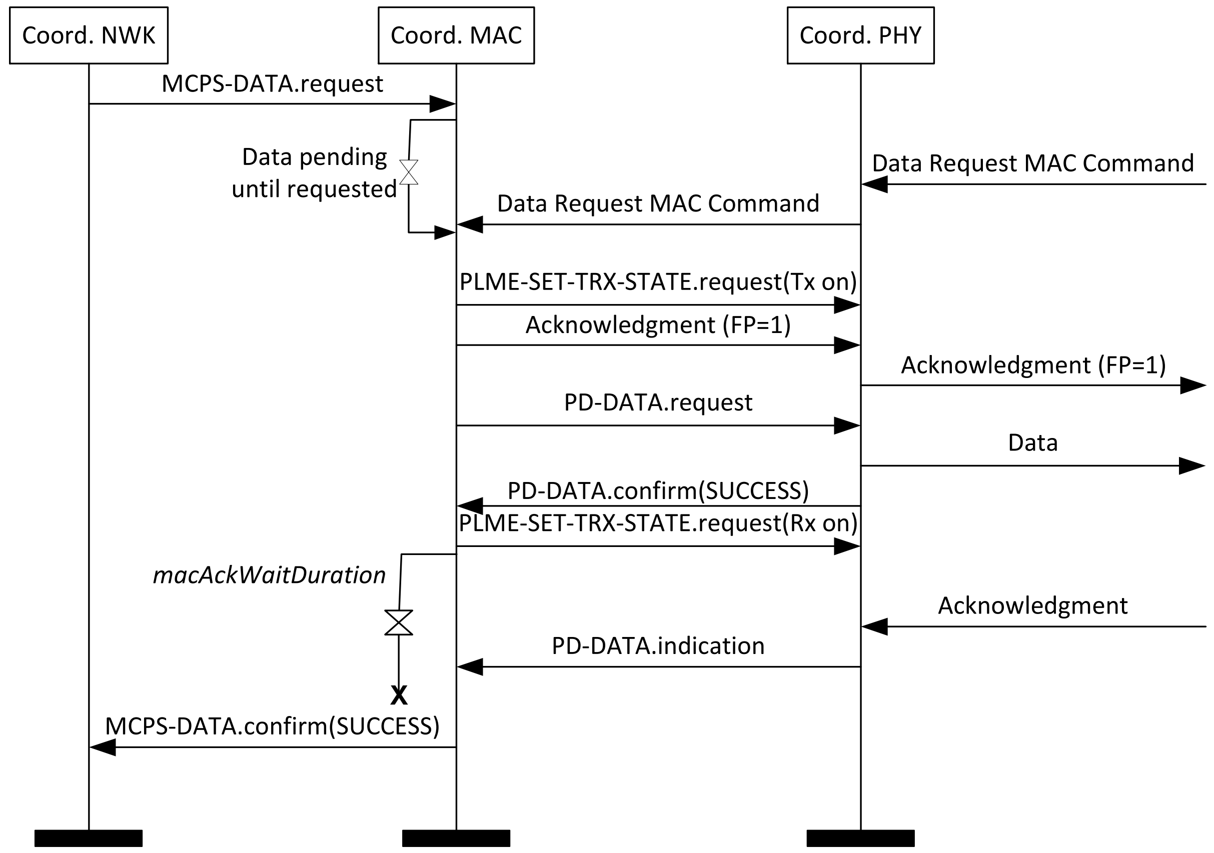


Figure 6.7: Polling message sequence chart - coordinator as implemented

estimate the size of a final design, a second case without debugging support and using the smaller Nios II/e processor was compiled. The results of both cases are shown in Table 6.1. The resources available are also shown for comparison. It is clear that a large portion of currently used resources, especially available memory bits, are used by debugging support. In particular, SignalTap II is responsible for the majority of the memory usage in the debug case. For comparison purposes, the minimum resource for a functional processor system has also been shown. This system consists of the Nios$\tilde{\text{I}}$I/e with SysID, JTAG and SRAM interface modules.

The software used to implement the MAC layer and PHY driver is stored in the 512 kB SRAM chip on the DE2 board. This chip is external to the FPGA. As with the hardware,

Table 6.1: FPGA Resource Usage

Resource	**Available**	**Debug**	**Final**	**Minimal**
Logic Elements	33,216	4,067 (14 %)	1,977 (6 %)	1,433 (4 %)
Memory bits	483,840	341,632 (71 %)	11,264 (2 %)	11,264 (2 %)
PLLs	4	1	1	0

the code footprint can be considered for two cases: debug and final. In the case of the final code, it was compiled using two different optimization level settings. In the first case, no optimization was used, while optimized size was used in the second. Code optimization is know to interfere with debugging, hence only the no optimization was used. In both cases a FFD was compiled. The resource usage is shown in Table 6.2. From the results it is clear that sufficient resources are still available for future enhancements of the current implementation.

Table 6.2: Code Usage of 512 kB SRAM

Memory Usage	**Debug**	**Final**	
	No Optimization	No Optimization	Optimized (size)
Program (code + initialized data)	151	149	125
Free (stack/heap)	353	354	378

Chapter 7

Conclusions and Future Work

The goal of this research was to implement an 802.15.4 compliant MAC/PHY on an FPGA. The work has addressed a void in availability of open-source code for FPGA. While even basic compliance with the standard has yet to be fully realized, the implementation presented here could easily achieve basic functionality with a small amount of additional code and testing. While successful stream transmission allowed some functions to be tested, the lack of successful stream reception prevented full functionality from being validated.

The current implementation supports only the most basic features of the 802.15.4 standard. There are several key areas of functionality that could be added to make the device more adaptable and flexible in its implementation. First, only security level 0 has been implemented. The 802.15.4 standard defines 7 other levels of security. Second, features such as the time-stamping of messages could be supported.

There are also deviations from the standard present in the current implementation that are a direct result of using the MC13192 transceiver as the PHY layer. For example, the standard states that the CRC calculation is done in the MAC layer. However, this is implemented in hardware by the transceiver. To make this implementation compatible with transceivers without this feature, an optional module should be added to the code. This would provide the CRC functionality when needed. Another variation is that the MC13192 does not require a `channelPage` to be set as it only supports the 2.4 GHz frequency band. Compliance with the standard, as well as flexibility to use other transceivers, requires that this be added where appropriate in the code.

Significant changes to the PHY/MAC interface were also necessary. The SAPs defined by the standard, as well as some of the primitives, have been replaced with a physical SPI bus and IRQ line. In order to accommodate transceivers using alternative interfaces, it will be necessary to adapt the PHY layer driver. Since Altera provides support in the Nios II core for most common interfaces, the most difficult part will be adapting to the registers and command sequences used by the particular transceiver.

This work provides an opportunity for future work to be done in expanding the system to support beaconed networks. While the beaconed networks require a considerably more complex coordinator and some additional functionality in devices, the current implementation provides a framework to build on. All additional functionality is in the MAC and NWK layers. This means future work could fully leverage the PHY layer, without having to necessarily understand every detail of it.

Overall this implementation serves as a framework for future research. It provides an opportunity for another student to use for exploration of optimization and refinement within a specific function or primitive. Alternatively, it could also be used as the backbone in a larger system. Possible systems include sensor networks and remote node configurations.

References

[1] LAN/MAN Standards Committee, *IEEE-802.15.4-2006 - Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*, IEEE Computer Society Std., 2006. 1, 8, 23, 31, 38, 50, 97

[2] B. K. Choi, B. S. Kim, S. S. Lee, Y. J. Kim, and D. J. Chung, "Implementation of IEEE 802.15.4 LR-WPAN 868/915 MHz, 2.4 GHz multimode baseband," in *Communications and Information Technology, 2009. ISCIT 2009. 9th International Symposium on*, September 2009, pp. 1055 –1056. 1

[3] R. Ahmad, O. Sidek, and S. Mohd, "Development of the CRC block for ZigBee standard on FPGA," in *Technical Postgraduates (TECHPOS), 2009 International Conference for*, December 2009, pp. 1 –4. 1, 6

[4] J. Flora and P. Bonnet, "Tiny15four: A portable, yet efficient 802.15.4 stack," in *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on*, October 2009, pp. 842 –849. 2, 5

[5] ZigBee Alliance. (2011) ZigBee Specification Overview. [Online]. Available: http://www.zigbee.org/Specifications/ZigBee/Overview.aspx 3, 8, 15, 97

[6] ABI Research. (2011) IEEE 802.15.4 IC Vendor Matrix. [Online]. Available: https://www.abiresearch.com/research/1002833?lr 3

[7] Texas Instruments. (2011) Broadband RF/IF and Digital Radio: ZigBee. [Online]. Available: http://focus.ti.com/analog/docs/gencontent.tsp?familyId=367&genContentId=24190 3

[8] NXP. (2010, July) NXP acquires Jennic to extend leadership in low power RF solutions for wireless applications. [Online]. Available: http://www.nxp.com/news/content/file_1741.html 3

[9] NXP. (2011) ZigBee PRO protocol stack. [Online]. Available: http://www.jennic.com/products/protocol_stacks/zigbee_pro 3

[10] Freescale Semiconductor. (2011) IEEE 802.15.4 (ZigBee) protocol stacks and tools. [Online]. Available: http://www.freescale.com/webapp/sps/site/homepage.jsp?code=802-15-4_HOME 4

[11] Freescale Semiconductor. BeeKit Wireless Connectivity Toolkit. [Online]. Available: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=BEEKIT_WIRELESS_CONNECTIVITY_TOOLKIT 4

[12] Freescale Semiconductor, *Freescale HCS08 802.15.4 MAC 2006 CodeBase Version 2.0.0*, 2010. 4, 16

[13] Atmel. (2011) IEEE 802.15.4 MAC. [Online]. Available: http://www.atmel.com/dyn/products/tools_card.asp?category_id=163&family_id=676&subfamily_id=2124&tool_id=4675# 4

[14] Atmel. (2011) MCU wireless. [Online]. Available: http://www.atmel.com/products/zigbee/?category_id=163&family_id=676 4

[15] TinyOS Alliance. Tinyos. [Online]. Available: www.tinyos.net 5

[16] Freescale Semiconductor, *MC13192 Reference Manual*, 1st ed., April 2008. 6, 47, 61, 64, 68, 69, 70

[17] D. Rohm, M. Goyal, H. Hosseini, A. Divjak, and Y. Bashir, "Configuring beaconless ieee 802.15.4 networks under different traffic loads," in *Advanced Information Networking and Applications, 2009. AINA '09. International Conference on*, May 2009, pp. 921 –928. 6

[18] A.-J. Garcia-Sanchez, F. Garcia-Sanchez, and J. Garcia-Haro, "Optimized orphan algorithm for IEEE 802.15.4 networks," in *Wireless Communication Systems, 2009. ISWCS 2009. 6th International Symposium on*, September 2009, pp. 428 –432. 6

[19] H. Li and Z.-Q. He, "The realization of mac controller in zigbee chip," in *Information Engineering and Electronic Commerce (IEEC), 2010 2nd International Symposium on*, july 2010, pp. 1 –3. 7

[20] C. Chauvenet, B. Tourancheau, and D. Genon-Catalot, "802.15.4, a MAC layer solution for PLC," in *Computer Systems and Applications (AICCSA), 2010 IEEE/ACS International Conference on*, May 2010, pp. 1 –8. 7

[21] M. Gorev and P. Ellervee, "Fpga based system for video compression and transmission over bluetooth," in *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, aug. 2010, pp. 367 –370. 7

[22] H. Tana, A. Sazish, A. Ahmad, M. Sharif, and A. Amira, "Efficient fpga implementation of a wireless communication system using bluetooth connectivity," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 30 2010-june 2 2010, pp. 1767 –1770. 7

[23] LAN/MAN Standards Committee, *802.15.1-IEEE Standard for Information Technology- Telecommunications and Information Exchange*, IEEE Computer Society Std., 2002. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7932 7

[24] LAN/MAN Standards Committee, *IEEE-802.15.4-2003 - Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*, IEEE Computer Society Std., 2003. 8

[25] Y. Xiao and Y. Pan, *Emerging wireless LANs, wireless PANs, and wireless MANs : IEEE 802.11, IEEE 802.15, IEEE 802.16 wireless standard family.* Wiley, Hoboken NJ, 2009. 9

[26] Federal Communications Commision (US Gov.). What we do. [Online]. Available: http://www.fcc.gov/what-we-do 13

[27] Industry Canada. (2011, June) Low-power Licence-exempt Radiocommunication Devices: Frequently Asked Questions. [Online]. Available: http://www.ic.gc.ca/eic/site/smt-gst.nsf/eng/sf08655.html 13

[28] Product Safety Engineering. (2009, September) Frequently asked questions. [Online]. Available: http://www.pseinc.com/faq.htm 14

[29] ABI Research. (2011) ZigBee Technology. [Online]. Available: http://www.zigbee.org/About/AboutTechnology/ZigBeeTechnology.aspx 15

[30] Freescale Semiconductor. AP13192USLK ZigBee-ready RF transceiver student learning kit. online. Freescale. [Online]. Available: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=ZIGBEESLK&fsrch=1 16

[31] Axiom Manufacturing, *Application Module for the Freescale MC13192 2.4 GHz RF Transceiver - Hardware User Guide*, July 2006. 17

[32] Altera, *DE2 Development and Education Board User Manual*, 1st ed., 2008. 18

[33] Altera. Low cost FPGAs. [Online]. Available: www.Altera.Com 18, 19

[34] Altera, *Nios II Processor Reference Handbook*, July 2010. 20

[35] Altera, *Nios II Software Developer's Handbook*, 9th ed., March 2009. 20

[36] Altera, *Embedded Peripheral IP User Guide*, 10th ed., 101 Innovation Drive San Jose, CA 95134, July 2010, uG-01085-10.0.0. 20

[37] Freescale Semiconductor, *802.15.4 MAC PHY Software Reference Manual, Rev 2.4*, 03 2010. 23, 24, 27

[38] Axiom Manufacturing. (2006, July) MC13192U Module AXM-0361, Rev F. [Online]. Available: http://www.axman.com/?q=node/334 72

APPENDICES

Appendix A

C Code for Testing MAC Data Primitives

C Code for the generation of `MCPS-DATA.request` test messages

```
void MacMcpsDataRequestTest(uint8_t dstAddrMode, uint8_t srcAddrMode){
    nwkToMcpsMessage_t *pMsg;
    uint8_t data[2] = {0xAA,0xBB};

    MyMacPIBTable.aMPibPanId[0] = 0xEF;
    MyMacPIBTable.aMPibPanId[1] = 0xBE;

    pMsg = MSG_AllocType(nwkToMcpsMessage_t);

    pMsg->msgType = gMcpsDataReq_c;
    pMsg->msgData.dataReq.dstAddrMode = dstAddrMode;
    pMsg->msgData.dataReq.srcAddrMode = srcAddrMode;
    pMsg->msgData.dataReq.txOptions = 0x1;

    switch(dstAddrMode){
    case 0:
        break;
    case 2:
        pMsg->msgData.dataReq.dstPanId[0] = MyMacPIBTable.aMPibPanId[0];
        pMsg->msgData.dataReq.dstPanId[1] = MyMacPIBTable.aMPibPanId[1];
        DstShortAddress(pMsg);
        break;
    case 3:
        pMsg->msgData.dataReq.dstPanId[0] = MyMacPIBTable.aMPibPanId[0];
        pMsg->msgData.dataReq.dstPanId[1] = MyMacPIBTable.aMPibPanId[1];
        DstLongAddress(pMsg);
        break;
    }

    switch(srcAddrMode){
        case 0:
            break;
        case 2:
            pMsg->msgData.dataReq.srcPanId[0] = 0xBA;//MyMacPIBTable.
               aMPibPanId[0];
            pMsg->msgData.dataReq.srcPanId[1] = 0xBE;//MyMacPIBTable.
               aMPibPanId[1];
            SrcShortAddress(pMsg);
            break;
        case 3:
            pMsg->msgData.dataReq.srcPanId[0] = 0xBA;//MyMacPIBTable.
               aMPibPanId[0];
            pMsg->msgData.dataReq.srcPanId[1] = 0xBE;//MyMacPIBTable.
               aMPibPanId[1];
            SrcLongAddress(pMsg);
            break;
    }

    pMsg->msgData.dataReq.msduLength = 2;  // 0-102
    pMsg->msgData.dataReq.msduHandle = MyMacPIBTable.aMPibDsn;
    pMsg->msgData.dataReq.securityLevel = 0;
    pMsg->msgData.dataReq.pMsdu = data;

    MacMcpsDataRequest(pMsg);
    MM_Free(pMsg);
}
```

C Code for the generation of `MCPS-DATA.confirm` test messages

```
void MacMcpsDataConfirmTest(){
    uint8_t status;
    uint8_t msduHandle;

    msduHandle = 5;
    status = 0; //success
    MacMcpsDataConfirm( msduHandle,status);

}
```

Glossary

6LoWPAN:	Common name for IPv6 over Low-power Wireless Personal Area Networks
CAP:	Contention Access Period
CFP:	Contention Free Period
CRC:	Cyclic Redundancy Check
CSMA-CA:	Carrier Sense Multiple Access with Collision Avoidance, an algorithm used for access coordination on wireless networks
FCC:	Federal Communications Commission
FCS:	Frame Check Sequence
FFD:	Full Function Device
FPGA:	Field Programmable Gate Array
GTS:	Guaranteed Time Slot
IEEE 802.15.4:	A short range, low data rate wireless transmission protocol developed by the Institute of Electrical and Electronics Engineers
ISR:	Interrupt Service Routine
MAC:	Medium Access Control Layer
MCPS:	MAC Common Part Sublayer
MLME:	MAC Layer Management Entity
MSB:	Most Significant Bit

NWK:	Abbreviation used to refer to the application/network layer
PD:	PHY Data
PHY:	Physical Layer
PIB:	PAN Information Base
PLME:	Physical-Layer Management Entity
POS:	Personal Operating Space - The space about a person or object that is typically about 10 m in all directions and envelops the person or object whether stationary or in motion[1].
RAM:	Random Access Memory
RF:	Radio Frequency
RFD:	Reduced Function Device - A device that is not capable of acting as a coordinator[1].
RISC:	Reduced Instruction Set Computing
SAP:	Service Access Point
SDRAM:	Synchronous Dynamic Random Access Memory
SOC:	System-on-chip
SPI:	Serial Peripheral Interface
WPAN:	Wireless Personal Area Network
ZigBee:	An application framework standard developed on top of the 802.15.14 MAC/PHY layers by the ZigBee Alliance[5]