Bachelor's Thesis

Implementing a ZigBee Protocol Stack and Light Sensor in TinyOS

Jacob Munk-Stander
jacob@munk-stander.dk

Martin Skovgaard
martin@vision-data.dk

Toke Nielsen
toke@mundt-stensgaard.dk

Department of Computer Science
University of Copenhagen

June 2005

Revised, October 2005

Abstract

The context of this thesis is the ZigBee wireless communication standard. The intended market space of applications using the ZigBee standard is home control, building automation and industrial automation. With these applications in mind the standard is optimized for low data rates, low power consumption, security and reliability.

In our thesis we describe, analyze and implement a reduced ZigBee protocol stack with a specific application in mind, namely the Light Sensor Monochromatic. Our objective is to implement the protocol stack and application on the Freescale MC13192-EVB platform, in less than 32,768 bytes. Obtaining this goal will cut current memory requirements in half, thus decreasing the cost of deploying ZigBee products. To minimize the size of the protocol stack, we analyze the required functionality of the light sensor application and implement both the protocol stack and application in TinyOS.

We have succesfully implemented the protocol stack and application, keeping the code size at 29,620 bytes—significantly below the 32,768 bytes limit. The implementation is not fully compliant with the ZigBee standard but it should be possible to achieve this without exceeding a code size of 32,768 bytes.

Even though the implementation is specialized to the light sensor application, the protocol stack in itself can be used to implement many other applications having similar requirements as the ones presented in this thesis.

Contents

List of Tables

List of Figures

1 Introduction

When deploying sensor networks, the choice of communication protocol depends on the context in which the network is used. The ZigBee protocol is designed for sensor networks used to control home lighting, security systems, building automation, etc.

In this thesis we will study the ZigBee protocol and implement a reduced version of the protocol stack specialized for use by a ZigBee light sensor.

1.1 Sensor Networks

In a time where focus in the computing industry is on computational power, the sensor networks paradigm takes a different approach. Where the personal computer is mostly about performing certain tasks in a controlled environment, sensor networks are all about the physical world and the inherent uncertainties that follow.

Sensor networks have already been deployed on wide scale and in a wide range of applications: from monitoring Leach's Storm Petrels' occupancy of small underground nesting burrows [1] to measuring sows in pig production [2] or alerting authorities of a developing forest fire [15]. It has become clear that the potential impact of sensor networks on our environment and daily lives is greater than ever, making it one of the most promising technologies of the decade [6].

Where personal computers are regarded as stable, inexpensive and computationally efficient, they have several disadvantages that prevent them from being deployed on a widespread scale in the physical world:

- **energy** – Without a permanent source of energy, the operating time of personal computers are measured in hours. This is a problem in sensor networks as a battery change in many cases would be infeasible, due to both locality and size of the network.
- **size** – Although the size of personal computers have decreased over the years, these cannot can be placed in a bird's nest, the collar of an animal or on a battlefield. Due to their sheer weight and physical dimensions, the placement would severely disrupt the environment in which the computer was placed.
- **cost** – While the cost of personal computers have decreased rapidly, sensor networks are targeting an entirely different price range. A cheap PC today cost in the order of a few hundred dollars while the price of a sensor node is about a tenth of this. This will allow sensor nodes to be deployed in massive numbers in new places, where a PC would be too expensive.

The approach of sensor networks is based on having a large amount of simple (often 8- or 16-bit processors and memory measured in kilobytes), small (down to 1 mm^3), inexpensive and computationally efficient (but slow) "nodes". Each node senses some parameter in the physical world. These measurements can be viewed in isolation or combined to solve a task that any one node could not solve.

Working nodes with limited capabilities have already been produced at the size of 1 mm^3, and as the progress in performance of computers has followed

Moore's Law the last decades, we can expect nodes living up to our expectations[1] in the near future.

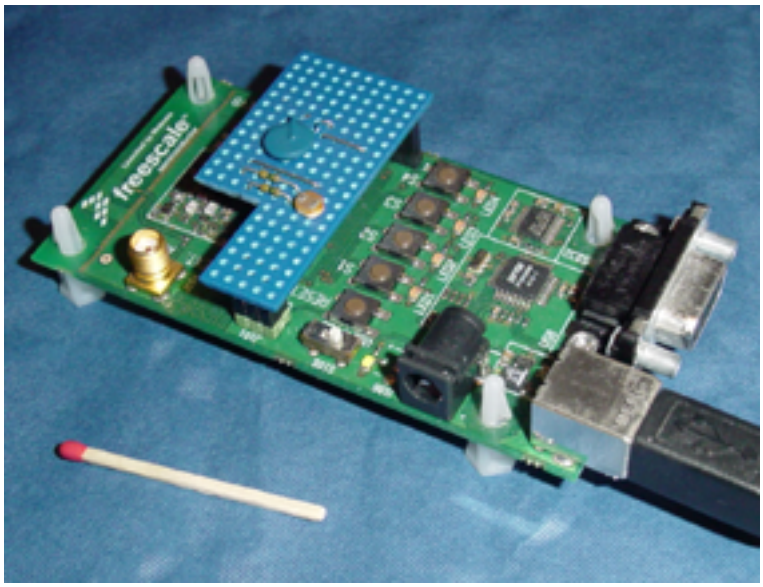


Figure 1: The Freescale Semiconductor MC13192-EVB node

1.2 Implementing ZigBee on Freescale nodes

The current physical size of nodes is the most apparent problem. We can implicitly address this problem by optimizing the software, and thereby reduce hardware requirements.

More specifically we will focus on utilization of the *Freescale MC13192-EVB*[2], configured with a temperature- and light sensor board. Our particular interest is in the use of the light sensor, along with the functionalities for wireless communication, using an IEEE 802.15.4 radio and the ZigBee protocol stack.

Size, cost and energy consumption are the main issues with regard to nodes, and we can reduce all of these by minimizing the memory footprint of our application. A smaller memory block means a smaller physical size, a cheaper node and less energy to maintain the state of the memory. Energy consumption can be reduced further by using duty cycling, i.e. only switching on components of the node as they are needed.

The goal of this thesis is to implement a ZigBee protocol stack and a light sensor application, which sends out light readings at specified intervals. Current implementations have a code size larger than 32KB, requiring FLASH memory blocks of 64KB. By implementing the ZigBee protocol stack and light sensor in less than 32KB, the cost of memory and the energy consumed by this can be cut in half.

[1] Nodes will become smaller, but have the same computational power.

[2] A node with, among other things, a USB port, push buttons, leds and an antenna for wireless communication.

To limit the size of our implementation we focus on implementing the mandatory primitives of the ZigBee standard, leaving out optional functionality where they are not needed in our application. Identifying necessary functionality will be an essential part of our analysis.

1.3 Approach

Since optimal memory usage and efficient code are our metric, we need a programming language and operating system that enhance these features. For this purpose we use the nesC programming language and the TinyOS operating system.

TinyOS is an open source operating system, implemented using nesC and designed for wireless embedded sensor networks. TinyOS and nesC are like two sides of a coin: nesC provides the language constructs on which TinyOS relies and TinyOS extends this model to a full operating system. The TinyOS core has a code size of 300-400 bytes and is component based. This means that only the necessary features of the operating system and application are included, thus limiting the code size.

An implementation of the IEEE 802.15.4 MAC layer is already provided by Freescale in a C library. To use this library in TinyOS our first task is to finish a nesC wrapper for the library. Work has already been made on this[3], but much is left to be implemented.

Having a working implementation of the IEEE 802.15.4 wrapper in nesC, we can continue to implement the ZigBee protocol stack on top.

We start off by analysing the requirements for our light sensor application and identifying the parts of the ZigBee protocol stack we consider necessary to support this. Afterwards we discuss how to implement these parts.

When implementing, we will take a bottom up approach. First we will implement the protocol stack with required functionality, based on the ZigBee specifications. When the protocol stack is complete, we implement the light sensor application, based on the ZigBee Light Sensor Monochromatic (LSM) device description [20].

1.4 Contribution

Given the context described up until now, our specific contributions to the field of sensor networks are:

- To analyze the ZigBee standard and its memory requirements in the context of a Light Sensor Monochromatic application.
- To implement a TinyOS wrapper for the IEEE 802.15.4 MAC layer provided by Freescale.
- To implement a minimal ZigBee protocol stack and Light Sensor Monochromatic application in TinyOS.

We believe these are novel contributions to the sensor networks community and will provide valuable insight into the use of ZigBee in sensor networks.

[3] By former Ph.D. student, Mads Bondo Dydensborg, at the Department of Computer Science, University of Copenhagen.

The rest of this thesis will be organized as follows:

In the first part we will provide an overview of the ZigBee standard. In the next part we will analyze the ZigBee protocol stack and how to reduce it to the needs of a light sensor. Before discussing the implementation, we will describe TinyOS, nesC and the hardware on which the protocol stack is implemented. We will then describe our implementation and finally present an evaluation of our protocol stack and light sensor with regard to functionality and code size.

Note, October 2005: At the time of writing the ZigBee specification consisted of several documents and was not openly available. Since then, the specification has been made available to the general public and combined into a single document, thus references to the specification in this thesis do not correspond to the combined ZigBee specification as is available from the ZigBee Alliance.

2 The ZigBee Standard

The ZigBee protocol is implemented on top of the IEEE 802.15.4 radio communication standard. The ZigBee specification is managed by a non-profit industry consortium of semiconductor manufacturers, technology providers and other companies, all together designated the ZigBee Alliance. The alliance currently numbers more than 150 members.

The ZigBee specification is designed to utilize the features supported by IEEE 802.15.4. In particular, the scope of ZigBee is applications with low requirements for data transmission rates and devices with constrained energy sources.

The intended market space for ZigBee products include home control and building automation. Imagine the intelligent building: controlling the lighting and temperature as needed, monitoring the building structure and performing surveillance tasks with a minimum of user interaction. This is the potential of ZigBee.

The overall ZigBee stack is illustrated in Figure 2.

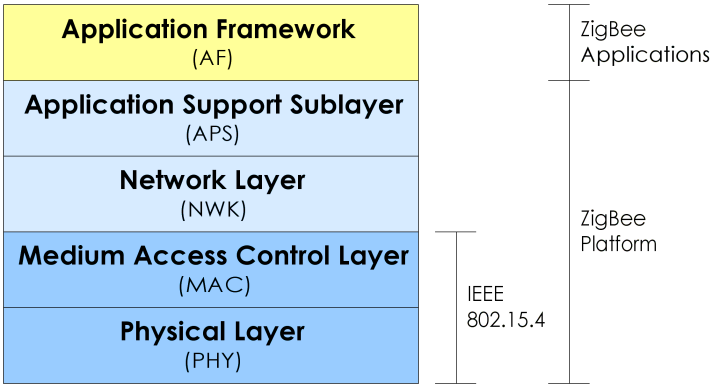


Figure 2: The overall ZigBee protocol stack

A comparison of prevalent wireless technologies is presented in Table 1. The use of Bluetooth in sensor networks is very limited [13], and the energy consumption of Wi-Fi makes this technology infeasible as well. Compared to these technologies, ZigBee is interesting and worth investigating further in the context of sensor networks.

	ZigBee	Bluetooth	Wi-Fi
Standard	802.15.4	802.15.1	802.11b
Memory requirements	4-32KB	250KB+	1MB+
Battery life	Years	Days	Hours
Nodes per master	65,000+	7	32
Data rate	250Kb/s	1Mb/s	11Mb/s
Range	300m	10m	100m

Table 1: Comparison of wireless technologies [24, 9]

Device Type	Description	Code Size
FFD	Full-blown FFD. Contains all 802.15.4 features including security.	37KB
FFDNGTS	Same as FFD but no GTS capability.	33KB
FFDNB	Same as FFD but no beacon capability.	28KB
FFDNBNS	Same as FFD but no beacon and no security capability.	21KB
RFD	Reduced function device. Contains 802.15.4 RFD features.	29KB
RFDNB	Same as RFD but no beacon capability.	25KB
RFDNBNS	Same as RFD but no beacon and no security capability.	18KB

Table 2: MAC/PHY software device type functionality [4, p. 1-1]

2.1 IEEE 802.15.4

The current IEEE 802.15.4 standard [11] was approved in 2003 and is managed by the Institute of Electrical and Electronics Engineers, IEEE. The standard differentiates itself from the more widespread 802.11 standard in focusing on lower data rates and lower power consumption [12].

In practise, this translates to data rates between 20 and 250 kbps depending on which of the three different radio frequencies that is used by the PHY layer[4].

The power management facilities of the standard enable battery-powered devices to operate for several months or years.

There are two overall types of devices defined by the standard: Reduced Function Devices (RFD) and Full Function Devices (FFD). These device types differ in their use and how much of the standard they implement.

Freescale provides seven C pre-compiled IEEE 802.15.4 MAC libraries, with varying degrees of functionality, cf. Table 2.

Our task is to implement an end device application (the light sensor) with the smallest memory footprint and since end devices can *optionally* be FFDs, cf. [17, p. 4], we can rule out the use of FFD-libraries. As our objective is to implement the ZigBee protocol stack using less than 32KB, we find it infeasible to use any other MAC library than the RFDNBNS, thus supporting neither beacons[5] nor security.

It should be noted that these design choices will not limit the use or compliance of our final implementation, rather the choices are made on the basis of requirements from the application we are implementing.

[4] 868 MHz (20 kbps, mainly Europe), 915 MHz (40 kbps, mainly North America and Australia) and 2.4 GHz (250 kbps, virtually anywhere).

[5] Beacons allow for synchronization in the network.

2.2 Core Concepts

A ZigBee network is called a *Personal Area Network* (PAN) and consists of one *coordinator*, one or more *end devices* and, optionally, one or more *routers*.

The coordinator is a *Full Function Device* (FFD), responsible for the inner workings of the ZigBee network. A coordinator sets up a network with a given PAN identifier which end devices can join. End devices are typically *Reduced Function Devices* (RFDs) to allow as cheap an implementation as possible. Routers can be used as mediators for the coordinator in the PAN, thus allowing the network to expand beyond the radio range of the coordinator. A router acts as a local coordinator for end devices joining the PAN, and must implement most of the coordinator capabilities. Hence a router is also an FFD device.

Commonly, coordinators and routers are mains powered and will in most cases have their radios on at all times. End devices, on the other hand, can be designed with very low duty cycling, allowing them long life expectancies, when battery powered.

Applications

The ZigBee Alliance provides a number of profiles that provide a framework for related applications to work within. This way, end devices from different vendors can interoperate as long as they adhere to the given profile.

One of these profiles is the *Home Control, Lighting Profile* [19]. This profile focuses on sensing and controlling light levels in the home environment. The profile defines different device descriptions which belong to the profile, e.g. Light Sensor Monochromatic, Switch Remote Control, Switching Load Controller and Dimmer Remote Control.

A profile can consist of 2^{16} device descriptors [18, p. 15] and can hold up to 256 *clusters*. Each cluster can contain up to 2^{16} attributes [18, p. 15]. A device description, contains a set of mandatory and optional input and output clusters from the profile.

Input clusters consist of attributes that can be set by other devices, e.g. the light sensor has an attribute called *ReportTime*, which controls the time interval between light readings. Output clusters consist of attributes that supply data to other devices, e.g. the Light Sensor Monochromatic (LSM) has one attribute in its output cluster, named *CurrentLevel*, which holds the current light sensor reading measured in lux[6].

Mandatory clusters (including every attribute within these) *must* be implemented by the appropriate end devices. Optional clusters *may* be implemented, but if a device supports an optional cluster, it *must* implement every attribute within that cluster.

Applications implement a device description. We will be focusing on the LSM device [20] in this thesis. The applications are implemented on different *endpoints* on an end device and are called *application objects*. Endpoints can be thought of as the port numbers used in TCP/IP.

To identify end devices, two address types exist. All end devices have a unique 64-bit IEEE address, also referred to as the *extended address*. Upon joining a PAN, an end device is assigned a 16-bit *short address* by the coor-

[6] *lux:* lumen per square meter.

dinator which is used as a sub-addressing mode, minimizing the overhead of addressing.

Application objects can send messages using *direct addressing*, also known as *unicast*, *indirect adressing* using bindings, see below, and *broadcast* addressing.

Two message types are defined:

1. **Key Value Pair (KVP) service** which uses a standardized way of representing messages using binary XML, or
2. **Message (MSG) service** which gives full control over the messages being sent for application specific needs

These message types are shown in Figure 8 and described further in Section 3.1.1 and Section 3.6.1, respectively.

Binding

Application objects at different end devices can initiate communication by a process known as *binding* which creates a logical link between application objects. More specifically, an entry is made in the *binding table* of the coordinator, identifying the endpoints of the application objects that requests a communication link. An application object can be bound with application objects at multiple end devices, as illustrated in Figure 3. Here switch 1 controls lamp 1, 2 and 3, while switch 2 only controls lamp 4. The concept of binding is similar to connecting two sockets in TCP/IP.

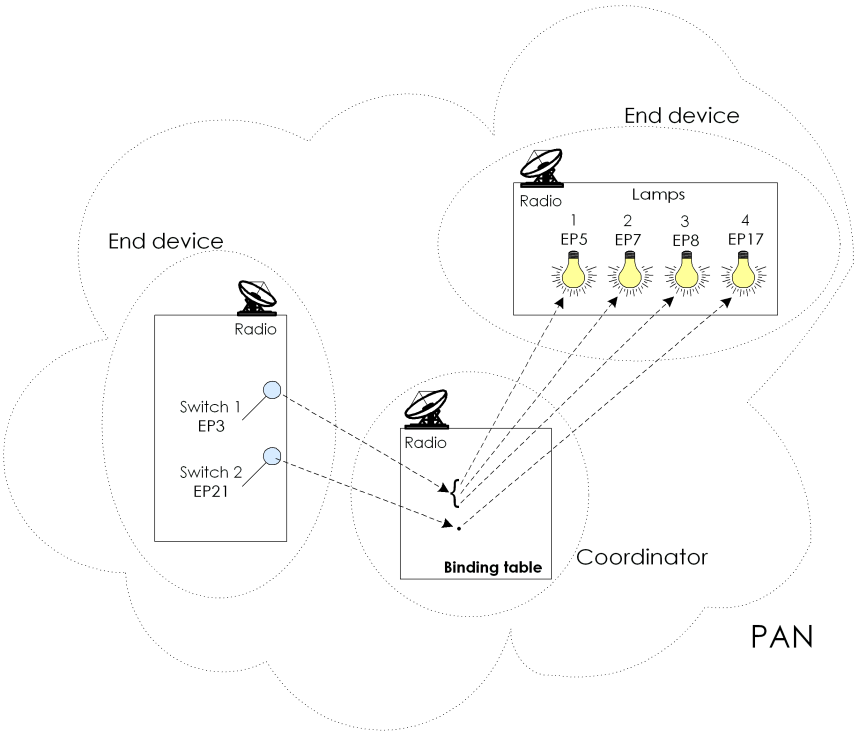


Figure 3: Binding several devices in the binding table

When two devices bind, the output cluster of one device is connected with the input cluster of another device. For example, the light sensor has one output cluster (`Output:LightLevelLSM`), and the switching load controller has the same cluster as input (`Input:LightLevelLSM`), thus the two devices can bind, ensuring that the light sensor will supply the switching load controller with periodical light sensor readings.

Binding can either be initiated by a coordinator/router directly, making the binding entry, or by the end devices themselves. The latter approach is known as *simple binding*, and is generally initiated by the press of a button on both of the two end devices wishing to bind two application objects.

When two bound application objects communicate, they do so via *indirect addressing*. The message is passed through the coordinator which identifies the recipient using the source address, source endpoint and cluster identifier. This way end devices need no knowledge of the addresses of the device(s) used in the communication.

To minimize power consumption (using duty cycling), end devices turn off their radio when it is not needed, e.g. after having binded. As messages can be sent to an end device at any time, the coordinator, with which the end device is joined, receives messages on behalf of the sleeping end devices. When an end device wakes up and is ready to receive a message, the end device *polls* the coordinator for available messages.

Descriptors
To describe the capabilities of devices within a network, the ZigBee protocol defines three mandatory descriptors: the node, power and simple descriptor. A descriptor is a set of attributes that other devices can request in order to obtain information about a device, e.g. the remaining power level or the services provided.

The three descriptors are characterized by:

- **Node Descriptor** – A node has one node descriptor which describes the type and capabilities of the node. The type of a node is either coordinator, router or end device. The capabilities of a node are properties such as frequency band, maximum buffer size, whether the receiver is on at all times or not, etc.
- **Power Descriptor** – A node has one power descriptor which describes the current power source in use, current power source level, etc.
- **Simple Descriptor** – A node has one simple descriptor for each endpoint. The simple descriptor holds information about the application residing on an endpoint. This includes the profile identifier, the number of input and output clusters, etc.

2.3 Network Stack

The ZigBee protocol stack has its origin in the Open Systems Interconnect (OSI) seven-layer model, initiated in the early 1980s by ISO and ITU-T[7]. A detailed

[7] International Organization for Standardization and International Telecommunication Union-Telecommunication

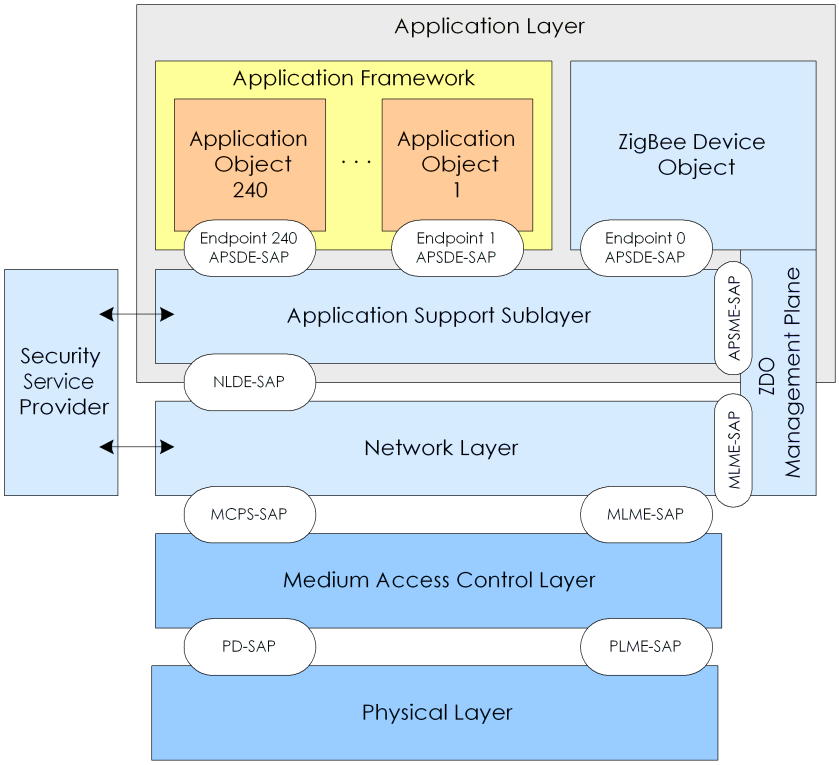


Figure 4: The detailed ZigBee protocol stack

ZigBee protocol stack is illustrated in Figure 4. The two lower layers are defined by the IEEE 802.15.4 standard while the remaining two layers are defined by the ZigBee Alliance:

1. **The Physical (PHY) Layer** is the lowest layer and is defined in the IEEE 802.15.4 standard [11]. It consists of two PHY-layers, operating in two separate frequency ranges: 868/915 MHz and 2.4 GHz.

2. **The Medium Access Control (MAC) Layer** is defined in the IEEE 802.15.4 standard [11]. The responsibility of the MAC layer is to control access to the radio channel using CSMA/CA[8]. The MAC layer provides support for transmitting beacon frames, network synchronization and reliable transmission using CRC and retransmissions.

3. **The Network (NWK) Layer** defined by the ZigBee Alliance [17], sends and receives data to and from the application layer. Furthermore, it performs the task of associating to and disassociating from a network, applying security and (on ZigBee coordinators) starting networks and assigning addresses. These services are provided through two interfaces—the *Network Layer Management Entity Service Access Point* (NLME-SAP) and the *Network Layer Data Entity Service Access Point* (NLDE-SAP).

4. **The Application (APL) Layer** is the top layer, and is defined by the ZigBee Alliance. It consists of the Application Support Sublayer (APS) [16], the ZigBee Device Object (ZDO) [22] and the Application Objects implemented on the given device:

   (a) **The Application Support Sublayer (APS)** provides two interfaces—the *APS Management Entity Service Access Point* (APSME-SAP) and the *APS Data Entity Service Access Point* (APSDE-SAP). The former is used to implement security, and by the ZDO of coordinators to retrieve information from the APS layer, while the latter is used by the application objects and the ZDO to send data.

   (b) **The ZigBee Device Object (ZDO)** provides an interface to the Application Objects used for discovering other devices and the services provided by these. Furthermore the ZDO sends responses to other devices requesting information about the device itself and the services provided by it. To support this, the ZDO uses the APSDE-SAP of the APS layer and the NLME-SAP of the NWK layer. The ZDO is a special Application Object, implemented on endpoint 0.

   (c) **Application Objects** are the actual manufacturer applications running on top of the ZigBee protocol stack. These adhere to a given profile approved by the ZigBee alliance and reside on endpoints numbered from 1-240. Endpoints, in conjunction with the address of the device, provide a uniform and unambiguous way of addressing individual application objects in the ZigBee network.

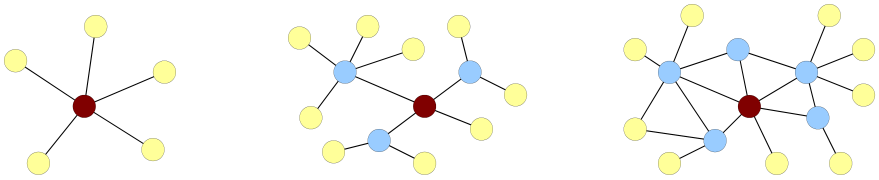
[8] Carrier Sense Multiple Access with Collision Avoidance

In addition to the above-mentioned layers, a Security Service Provider is optionally supported. This provider is used by both the NWK layer and APL layer. As we will not be implementing any security specific features, we will not describe this further.

2.4 Network Topologies

The nodes in a ZigBee network can be arranged using three different *network topologies*: star, tree and mesh.

The simplest of the three topologies is the *star* topology, shown in Figure 5(a). Here the ZigBee network contains one coordinator, no routers and a number of end devices. Each end device is within radio range of the coordinator.



(a) Star topology (b) Tree topology (c) Mesh topology

Figure 5: The ZigBee network topologies

In the *tree* topology, the communication routes are organized in such a way that there exists exactly one route from one device to another, see Figure 5(b). End devices may either communicate directly with the coordinator or with exactly one of a number of routers.

As with the tree topology, end devices in a *mesh* communicate either directly with the coordinator or with a router. Unlike the tree topology, there may be several routes between different routers in a mesh topology. This redundant routing is transparent to the end devices, and introduces some reliability in the network, at the cost of added complexity. An example of a mesh network can be seen in Figure 5(c).

As may be noticed, the star topology is a subset of the tree topology, which again is a subset of the mesh topology.

3 Analyzing the ZigBee Protocol Stack

The following sections describe the parts of the ZigBee protocol needed to implement a light sensor operating in a star network. We will first describe the Light Sensor Monochromatic device and thereby identify the functionality needed to implement this. Then we will consider how to provide this functionality, including how to send and receive data. Finally we will discuss issues concerning buffer management, concurrency and duty cycling.

The primitives mentioned in the following sections and their placement in the protocol stack are illustrated in Figure 6.

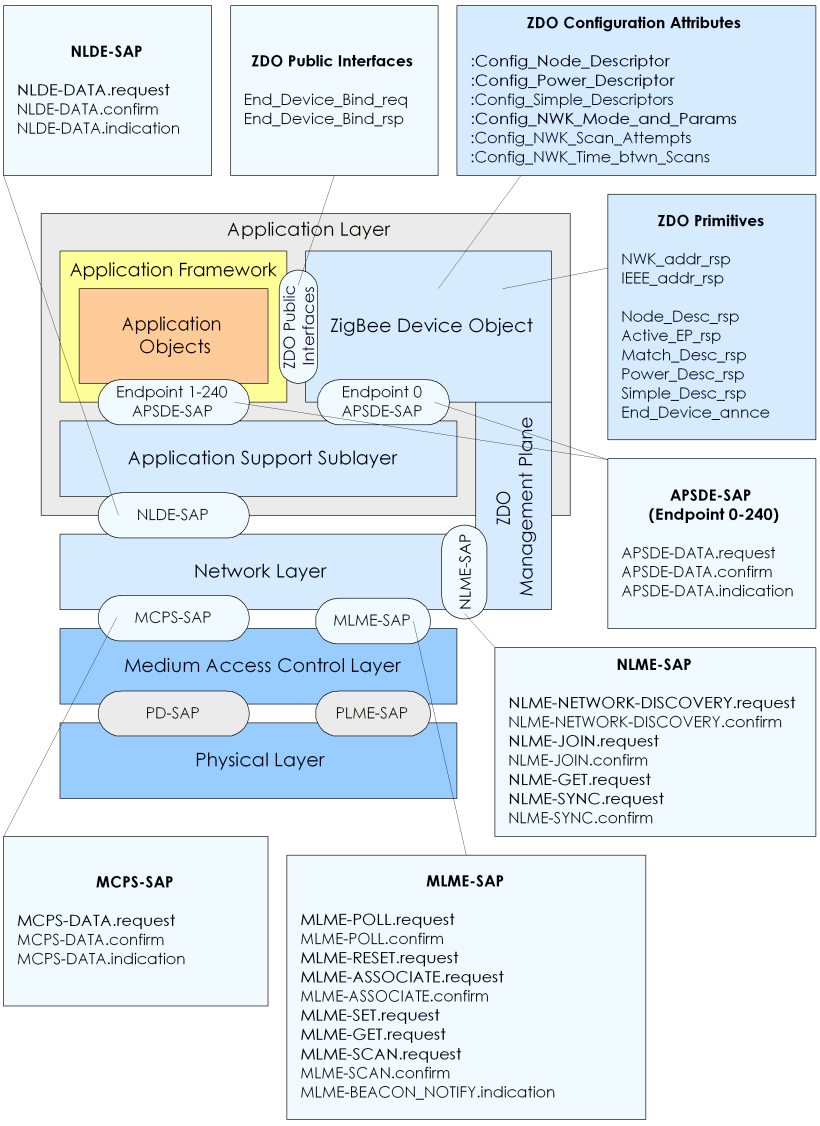


Figure 6: The primitives we will be needing

Cluster	Description
`Output:LightLevelLSM`	Mandatory cluster, containing one attribute, used for outputting the measured ambient light level to an external device.
`Input:ProgramLSM`	Optional cluster, containing eight attributes, used for programming the state of an LSM, allowing output behaviour and light level offset to be changed.

Table 3: Clusters defined in the LSM device description

3.1 Light Sensor Monochromatic

The device description for the Light Sensor Monochromatic [20] defines two clusters: one mandatory and one optional, see Table 3.

We have chosen to implement both the mandatory and the optional cluster, as:

1. It makes it possible to test whether our end device is capable of receiving (and handling) packets, as well as sending them.
2. It makes it possible to test binding of several clusters on the same device.
3. Testing would have been very limited with an LSM that only supports the mandatory output cluster.

To test our LSM, we will implement a simple *programming* device and a simple *consumer* device (both without any reference to official ZigBee profiles). These two devices are implemented with the sole purpose of testing our LSM device; hence we will not describe them in this section, merely note that they are implemented to send and receive data in the correct format and to utilize the two clusters of the LSM. For implementation specific details, see Section 6.4.

In the following we will describe the overall functionality required of the ZigBee procol stack, in order to implement the above mentioned clusters.

3.1.1 Required Functionality

Basically, there are three steps when deploying a light sensor:

- Joining a network.
- Binding devices.
- Operation (sending and receiving data).

For each of these, the application issues requests to the ZigBee protocol stack.

Attribute ID	Functionality
CurrentLevel	Current ambient light level measured by light sensor.

Table 4: Attributes in the `Output:LightLevelLSM` cluster

Attribute ID	Functionality
ReportTime	Interval in seconds between the output of light readings.
MinLevelChange	Level change needed before outputting a new value.
MinThreshold	Threshold to *drop below* before outputting a new value.
MaxThreshold	Threshold to *exceed* before outputting a new value.
Offset	Offset to add/subtract in all light readings.
Override	Disable all output.
Auto	Enable output.
FactoryDefault	Reset device to factory defaults.

Table 5: Attributes in the `Input:ProgramLSM` cluster

Joining a Network
After the device has been turned on and the radio and other components have been initialized, the first thing the device must do is to scan for available networks and subsequently join one of them.

Since an end device can only join one network, joining is handled by the ZDO and not by the individual application objects. Each application object will be notified by the ZDO when a network has been successfully joined.

Binding Devices
After a successful join, the application can issue a request to bind with other matching devices on the network. To bind with another application, we will need to send the identifier of the profile and the supported input and output clusters to the coordinator. This process is described further in Section 3.3.

When a device carries out a successful bind, it is ready to go into operation mode. Binding can also be initiated at any time during operation mode.

Sending Data
In the case of the LSM device, we need to output the light level with a given interval, specified by a factory default. The transmission of light level readings is the only mandatory part of the LSM, and also the only output specified in the device description (see Table 3 and Table 4).

The light reading will be sent in a message via the `APSDE-DATA.request` primitive. The destination of the packet will be all consumers within the Home Control Lighting (HCL) profile, that have bound with the LSM device's output cluster. An example of a consumer could be the Switching Load Controller, specified in [21].

Receiving Data
As mentioned in Section 3.1, we have implemented the optional `Input:ProgramLSM` cluster. The attributes defined in this cluster and the functionality they implement, are described in Table 5. There are some ambiguities

in the device description, concerning simultaneously enabled parameters. In the following we will describe how we choose to interpret them.

By setting *ReportTime* it is possible to decide how often the light sensor is read. If no other attributes are set, the light reading will also be outputted each time the sensor is read.

Setting the *MinLevelChange* attribute will cause the LSM to only output a reading if it has changed a certain percentage, since the last time a value was outputted. The change can be either positive or negative.

MinThreshold and *MaxThreshold* define two boundaries that must be dropped below or exceeded, respectively, before outputting a new light reading. Once the level drops below *MinThreshold*, it must exceed *MaxThreshold* to output a new light level, and vice versa. This way it is possible to provide hysteresis[9] for the light level, and hence avoid unnecessary outputs, when the light level is changing from one side of a boundary to another, see Figure 7.

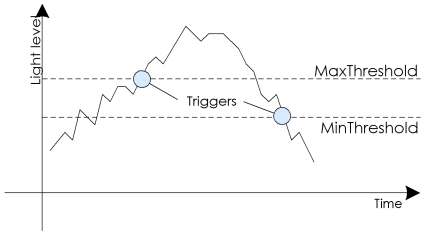


Figure 7: Hysteresis for the threshold attributes

If both *MinLevelChange* and *Min-/MaxThreshold* are enabled, there are a number of possible interpretations to choose from. The device description [20] does not clearly specify how to handle this, but we have identified these possible solutions:

- If *MinLevelChange* and *Min-/MaxThreshold* are both set, then both conditions *must* be satisfied in order to output a new value.
- If they are both set, just one of them should be satisfied before outputting a new light reading.

We believe it to be more likely that a consumer would rather receive some unnecessary light readings, than miss out on necessary ones. If an application object does not need a reading, it is free to simply discard it. Therefore we opt for the second choice. If life expectancy were the critical parameter, we might had opted for the first choice.

Another ambiguity in the device description concerns the priority of the *ReportTime* attribute, when it is enabled together with *MinLevelChange* or *Min/MaxThreshold*. This can be handled with two different approaches:

- The value of *ReportTime* should have precedence over other attributes, hence ensuring that a light level is outputted at least with the given interval.

[9] *hysteresis:* the lagging of an effect behind its cause.

- The value of *ReportTime* is used to determine how often to poll the light sensor and subsequently check the *MinLevelChange* and *Min/MaxThreshold* attributes, hence outputting at most with the interval specified by *ReportTime*.

The disadvantage of the first option is quiet clear; the user has no control over how often the attributes defined are checked. We expect the maximum time allowed between examination of attributes to vary significantly from one application to another. With the first option the only way to change this is by modifying the code for our LSM.

The problem with the second option is that the behavior of *MinLevelChange* and *Min/MaxThreshold* depends on *ReportTime*. We might imagine a sensor that only needs to output readings if radical changes occur, but should still output a reading at least with some given interval. With the second option this scenario is not possible.

Even though there are disadvantages of both options, we decide on the second. The primary reason for this is that we consider it a great advantage to be able to determine how often to examine the specified attributes.

The *Offset* attribute is used to correct all readings by a given offset. The offset can be either positive or negative, and it is taken into account *before* checking any attributes with regard to output constraints.

Override is used to disable all output, but otherwise keep the state of the device intact. Hence side effects of all functions will occur, even if *Override* is set, however no output is produced. The *Auto* attribute is used to enable output again.

The final attribute is *FactoryDefault*, which will reset everything to factory defaults.

KVP Commands

When end devices communicate, they do so by sending an *AF*[10] *command frame*. The AF command frame can hold a number of either MSG or KVP messages, known as transactions, see Figure 8. An AF command frame holding more than one transaction is called an *aggregated transaction*.

The LSM device uses KVP commands exclusively, when sending and receiving messages.

The *Transaction sequence number* increases by one everytime a device sends a transaction and is wrapped around when it reaches 255. This way it is possible to identify a given transaction, which can be necessary when handling acknowledgements.

The second field, *Command type identifier*, determines the type of transactions we are dealing with[11], and the *Attribute data type* specifies one of the ZigBee standard data types[12] of the attribute data. This field is somewhat redundant as the datatype is uniquely determined by the cluster and attribute identifier, see below, alone—but since it is defined in the KVP frame format, we set the correct data type, corresponding to the data type defined in the cluster.

[10] Application Framework

[11] Set, Event, Get/Set/Event with Acknowledgement or Get/Set/Event response.

[12] No data, uint8, int8, uint16, int16, semi-precision, absolute time, relative time, character string and octet string.

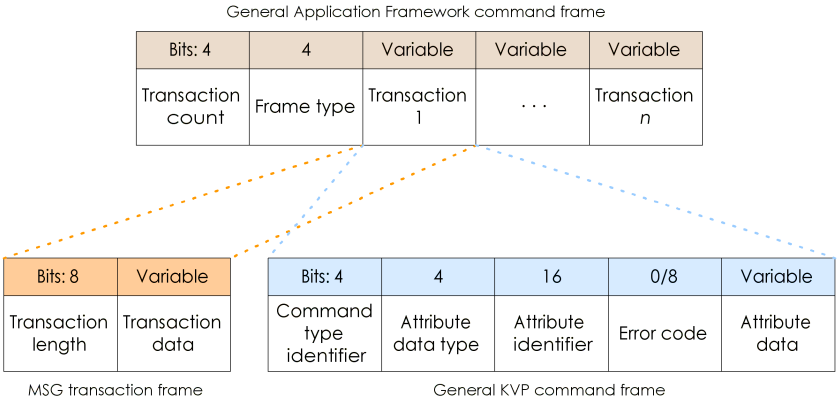


Figure 8: The Application Framework command frame

The *Attribute identifier* tells us which attribute within the cluster we are accessing. The optional *Error code* indicates the success or failure (and possible reason) of a request to another device. And the final field, *Attribute data*, is the payload of the package if there is any. The type of *Attribute data*, and thereby the way it is handled, is determined by the *Attribute data type* field.

3.1.2 Omitted Functionality

The ZigBee specifications include some functionality which we have chosen not to implement. The most important features that we have omitted, are described in the following.

Application Level Acknowledgements
It is possible to send KVP commands, asking for an acknowledgement. Since acknowledgements are supported in the APS layer, we have chosen not to support it in the application layer. Furthermore, all functions in our LSM device are idempotent, i.e. calling the same function one or more times with the same arguments, will not create invalid states or compromise the precision of our device.

If we were to handle application level acknowledgements, it would, as mentioned, rely on the transaction sequence numbers. Basically all that is needed is to send an acknowledgement to the sender of the original message, including the received sequence number and appropriate *Error code*. If a sender never receives an acknowledgement, it simply retransmits the lost message.

Semi-Precision Data Types
A deviation from the LSM device description is the absence of semi-precision data types[13]. From an optimization perspective it is always favourable to avoid floating point data types when an integer will do. But more importantly, in our case the readings returned from the light sensor are 10 bits and represent integer lux values (in the range from 0-1024). Hence we can easily handle our light levels as integers, without any loss of precision.

[13] A standard ZigBee data type, using two bytes to represent a floating point number.

This means that some devices may erroneously use the integer as a semi-precision number. However, because of the *Attribute data type* field, this should not happen.

Get on attributes
Besides setting an attribute in a given cluster, the standard also specifies a method to get the state of these attributes. This might be practical if, e.g. one LSM were to be programmed by several devices. But in our case we only have *one* programming device, which sets all the attributes. Therefore it should not be necessary to ask the LSM for the state of any attributes.

3.2 Joining a Network

In order to join a PAN the ZDO must issue an `NLME-NETWORK-DISCOVERY.request` to the NWK layer management entity. A constant defined in the ZDO, `:Config_NWK_Mode_and_Params`, specifies which channels to scan. The discovery request shall be issued the number of times specified in `:Config_NWK_Scan_Attempts` each separated in time by `:Config_NWK_Time_btwn_Scans`.

Scanning for PANs
The `NLME-NETWORK-DISCOVERY.request` will scan the specified channels using `MLME-SCAN.request` in the MAC layer management entity. There are two scan modes: active and passive scan.

When performing an active scan, the end device transmits a beacon[14] request and enables the receiver. Coordinators and routers, receiving the beacon request, will send back a beacon with a PAN descriptor, describing the properties of their PAN. This will be repeated on each of the channels specified. In contrast, a passive scan merely enables the receiver and listens for the periodic beacons that coordinators and routers send out. This scan mode will save some energy (avoiding the transmission of a beacon[15]), but in most cases the receiver will be enabled longer than in the active scan scenario.

The option to choose one over the other is a matter of weighing energy efficiency versus the time it takes to join a network. We have opted for the active scan to make the join procedure as quick as possible.

Beacons from a coordinator or router are received via the `MLME-BEACON-NOTIFY.indication` primitive. The PAN descriptor contains, among other things, the PAN identifier. The descriptor is saved in a local *network descriptor list* and is later used to determine which network to join. A *neighbour table* is also assembled as there could be multiple beacons for the same PAN identifier if there were routers within reach. The neighbour table will be used later to join the coordinator/router with the lowest link cost in the selected PAN.

The resulting `NLME-NETWORK-DISCOVERY.confirm` will supply the ZDO with a network descriptor list containing the list of active PANs.

This process is illustrated in Figure 9.

[14] The beacons described here are not to be confused with the beacon used for synchronization. These beacons are merely used to identify nearby coordinators and routers.
[15] To transmit a bit consumes more energy than receiving a bit.

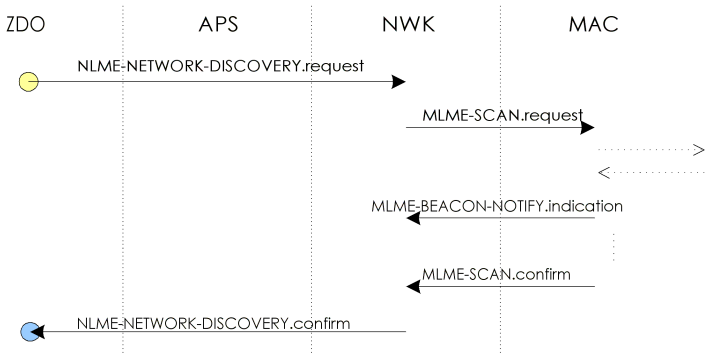


Figure 9: Scanning for PANs

Choosing a PAN to join
The ZDO will choose a PAN to join, based on whether the network accepts new devices, the security level and other factors. When a PAN is chosen, the ZDO performs an `NLME-NETWORK-JOIN.request` to the NWK layer, with the selected PAN identifier.

Here a link cost is calculated for each coordinator and router in the neighbour table. The link cost can be estimated in numerous ways as described in [17, pp. 82-83]. We have chosen the simplest way possible, i.e. hard coding the link cost, as we will only have one coordinator in our test set up.

After having selected a parent (coordinator or router) to join, the capabilities of the joining end device are assembled. These include how the device is powered (mains powered or by other means), whether the receiver is on when the end device is idle, whether MAC security is available etc.

A request to join the parent, including the capability information, is sent using the `MLME-ASSOCIATE.request` located in the MAC layer. Upon receipt of the `MLME-ASSOCIATE.confirm` primitive in the NWK layer, the short address assigned to the end device by the coordinator is saved and the relationship field of the selected parent is set to "parent" in the neighbour table. The `NLME-JOIN.confirm` primitive notifies the ZDO that the PAN has been joined.

Confirming join
To confirm the join, the ZDO sends an `End_Device_annce` to the parent, using the APS data entity with the short address and extended address of the end device. On receipt of the `End_Device_annce_rsp` from the parent, the network has been joined and all active endpoints are notified.

If an endpoint application tries to issue commands before the join process is completed, an error is returned.

Joining a PAN and confirming this, is illustrated in Figure 10.

3.3 Device Binding

Binding in the context of an end device is performed using a process known as simple binding. Typically two end devices bind in response to some user action,

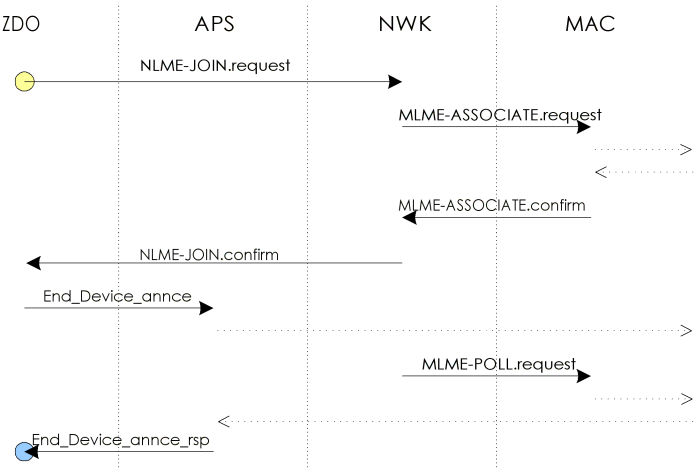


Figure 10: Choosing a PAN

e.g. the press of a button.

To bind, e.g. a Light Sensor Monochromatic device with a Switching Load Controller device, the application object on each end device issues an `End_Device_Bind_req` to the ZDO. When issuing the `End_Device_Bind_req`, the application object supplies the ZDO with its profile identifier and a list of its input and output clusters. The ZDO of each end device then sends a bind request to the coordinator.

When the coordinator receives the two bind requests, it compares the input and output clusters of the two application objects wishing to bind. Binding can only occur when a *match* between these is found, i.e. the input cluster(s) from one application object match the output cluster(s) of the other. If a match is found, an `End_Device_Bind_rsp` is sent to the two binding end devices with a `SUCCESS` status, otherwise the status is `NO_MATCH`.

If the coordinator only receives one bind request within a pre-configured time period, a `TIMEOUT` status is sent to the binding end device.

3.4 Data Transmisson

As mentioned in Section 2.2, the ZigBee protocol defines three addressing mechanisms: direct addressing, indirect addressing and broadcast addressing.

Direct addressing
Direct addressing, also known as normal unicast, is used to communicate from one device to another. To use this addressing mechanism, the sending device needs to know the short address or extended address of the recipient device. These addresses can be obtained using the primitives mentioned in Section 3.5.

Even though the APS specification specifies that an application object should have the option of using the extended address instead of the short address [16, p. 10], the underlying NWK layer only supports short addresses when sending data [17, p. 12]. Neither the NWK layer nor APS layer specifications specify how to resolve this inconsistency, so we have chosen to base our implementation

solely on the use of short addresses. A solution to this problem could be to use the device discovery primitives to translate the extended addresses to short addresses. However, we have chosen not to implement this.

Indirect addressing
Indirect addressing requires the sending and the receiving devices to have bound through the coordinator. When two devices are bound, they do not need to know the address of the other device, as the coordinator will orchestrate the delivery of messages. This allows for one-to-many and many-to-one relationships between participating end devices, cf. Figure 3.

Broadcast addressing
The ZigBee specification is highly ambiguous with regard to broadcast addressing. In the following we will lay out the possible interpretations of the specification.

In the NWK layer, a special network broadcast address is defined, namely `0xFFFF` [17, p. 47]. When sending a message to this address, all devices in the network will receive it [17, p. 96]. Furthermore [18, p. 15] defines a broadcast endpoint, `0xFF`[16]. When sending a message to this endpoint, the receiving device shall deliver it to *all* active endpoints. Based on this it can be concluded, that:

1. It is possible to send a message to a specific endpoint on all devices in the network.
2. It is possible to send a message to all endpoints on one device.
3. It is possible to send a message to all endpoints on all devices in the network.

In [18, p. 13] an *application broadcast* is defined as option 3 stated above. When issuing an application broadcast [18, p. 13] states that the destination address must be set to `0xFFFF` and the delivery mode of the APS header must be set to broadcast, see Section 3.6.2.

The only way to specify broadcast addressing in the application layer, is to set the destination address of a data request to `0xFFFF` [16, p. 10]. This, in turn, sets the delivery mode to broadcast. When setting the delivery mode to broadcast, the receiving device should deliver the message to the active endpoints with the profile identifier specified in the message [16, p. 26]. This rules out option 3, as a message cannot be issued to all active endpoints if they do not have the same profile identifier.

Furthermore, when setting the delivery mode to broadcast addressing, it is not possible to specify a specific destination endpoint, given the above description, thus ruling out option 1 as well.

This leads us to the conclusion that application broadcast, as defined by the ZigBee standard, is not possible. We could remedy this by setting the destination endpoint of the broadcast message to `0xFF`, and on delivery let this take precedence over the delivery mode in the APS header. However, this approach is not described in the ZigBee standard.

Given the above analysis, we have chosen to implement broadcast as follows:

[16] In [18, p. 6] endpoint 31 is said to be the broadcast endpoint, but this is not used anywhere else, thus we consider it an error in the specification.

- If the destination address is set to `0xFFFF`, we broadcast the message to all devices. Upon receipt it is delivered to all active endpoints matching the profile identifier.
- If the destination endpoint is set to `0xFF`, the unicast message is delivered to all active endpoints without filtering.

The ZigBee specification also specifies that upon receipt of a broadcast message, this message should be retransmitted to all nearby devices using broadcast. The device must then verify delivery of the broadcast message by keeping track of whether all nearby devices have broadcast the message themselves after receiving it. If not the message is retransmitted a limited number of times.

We have chosen not to implement this last behaviour as we will only be receiving message directly from the coordinator. Furthermore we question the use of this practice, given that all end devices are associated with a coordinator or router. These in turn must be in contact with each other to form the PAN, thus ensuring that the broadcast message can propagate throughout the network without the use of end devices. This makes the participation of end devices, in this process, superfluous.

3.4.1 Sending Data

When an application object needs to send data, it issues a data request to the APS layer through the APS data entity `APSDE-DATA.request`. The data is prepended with headers from each of the underlying layers, as illustrated in Figure 11. The format of each header is described in Section 3.6.

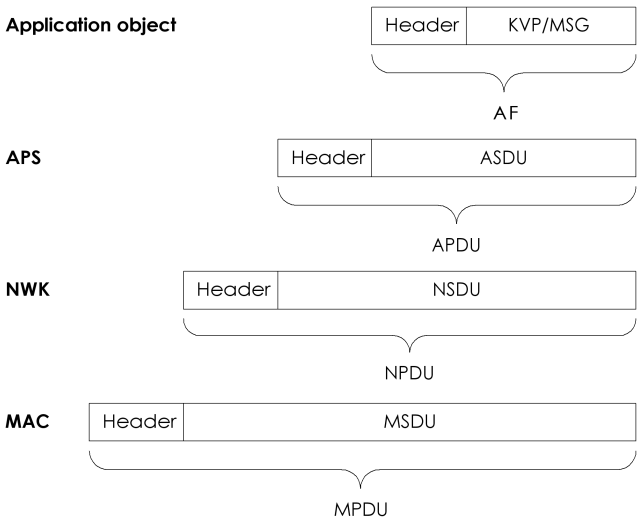


Figure 11: Constructing a data packet

3.4.2 Receiving Data

To receive data the ZDO layer sends a poll request to the coordinator, using the `NLME-SYNC.request` in the NWK management entity. This, in turn, issues

an `MLME-POLL.request` in the MAC layer.

If the coordinator has pending messages for the end device, it will send a response indicating this. If the MAC layer receives a response, indicating that there are pending messages, it will keep the radio receiver on. After sending the indication, the coordinator will send the first of the pending messages. Next time the coordinator is polled, it will send the next message and so forth.

If the coordinator indicates that it does not have any pending messages, the radio receiver of the end device is turned off immediately.

When the end device receives a message, the `MCPS-DATA.indication` is issued from the MAC layer, and tells the NWK layer that data is available. The NWK layer will issue an `NLDE-DATA.indication` to the APS layer. When the APS layer receives the `NLDE-DATA.indication`, it needs to analyze the header, to determine what kind of packet it is and what to do with it. Three kinds of packets are defined:

- normal unicast,
- broadcast, and
- acknowledgement.

As stated earlier, an end device can send a message using indirect addressing; messages sent this way go to the coordinator which in turn converts them to unicast messages and sends them to the relevant end device(s). Thus, the APS layer of an end device does not need to handle indirect packets, as they will be received as unicast messages.

If the message received is a unicast message, the APS layer shall issue an `APSDE-DATA.indication` to the relevant endpoint. If the message is addressed to endpoint `0xFF`, the `APSDE-DATA.indication` primitive shall be issued to all active endpoints. If the received message was broadcast, the `APSDE-DATA.indication` primitive shall be issued to all endpoints that match the profile identifier in the packet.

If the received message is an acknowledgement, the APS layer should issue an `APSDE-DATA.confirm` to the relevant endpoint.

3.4.3 Acknowledgements

When an application object requests an acknowledged transmission, the APS layer shall set the *acknowledge request bit* in the APS header accordingly, see Section 3.6.2.

To keep the code size of our implementation at a minimum, we have designed a very simple solution for acknowledged transmissions. We do not allow data transmission requests if we still have not received an acknowledgement for an acknowledged transmission. This means that if an application object has just requested the transfer of an acknowledged packet and immediately afterwards requests another transmission, acknowledged or not, the APS layer will return an `APSDE-DATA.confirm` with status set to `DATA_REQUEST_BUSY`. Note that this status value is not defined in the ZigBee standard, but we found it necessary to add it to our implementation, given the open interpretation of how multiple packets are handled. Of course the application developer needs to be made aware of this implementation specific detail.

Furthermore, if we are waiting to send an acknowledgement, we do not allow any data requests from application objects, thus prioritizing the acknowledgement. This is done to minimize the number of duplicate packets received.

If the APS layer receives an acknowledgement with the same cluster identifier and a source endpoint matching the destination endpoint in the original message, the transmission shall be assumed to be succesful, and an `APSDE-DATA.confirm`, with status set to `SUCCESS`, shall be issued to the application object.

If the APS layer does not receive an appropriate acknowledgement within `apscAckWaitDuration` seconds, it shall retransmit the original frame. This procedure is repeated `apscMaxFrameRetries` number of times. If all retransmissions fail, an `APSDE-DATA.confirm` primitive, with status set to `NO_ACK`, shall be issued to the requesting application object. Note that the use of retransmissions can lead to duplicate messages received in application objects. The APS layer specification explicitly states that a mechanism for handling duplicate packets should not be implemented in the APS layer and is therefore left to the application developer [16, p. 36].

3.5 Device and Service Discovery

To find other end devices and the services they offer, the ZDO provides several primitives. Issuing discovery requests is optional for end devices, whereas responses to the following device and service discovery requests are mandatory:

NWK_addr_req
This primitive is used to retrieve the short address of an end device, based on its extended address. The request is broadcast on the PAN and the end device with the requested extended address shall send a unicast response, `NWK_addr_rsp`, with its short address to the sender.

IEEE_addr_req
This primitive is used to retrieve the extended address of an end device based on its short address. The request is sent as a unicast message to the appropriate end device. The end device that receives this request should respond with a unicast message, `IEEE_addr_rsp`, containing its extended address.

Node_Desc_req, Power_Desc_req and Simple_Desc_req
These primitives are used to retrieve the three descriptors mentioned in Section 2.2 from an end device.

There is only one node descriptor and power descriptor per end device. There is one simple descriptor per endpoint, thus `Simple_Desc_req` shall include a specific endpoint to inquire about.

These requests are all unicast and upon receipt, the end device sends a unicast response, containing the appropriate descriptor.

Active_EP_req
This primitive is used to retrieve a list of active endpoints from an end device. The request is sent unicast, as is the response, `Active_EP_rsp`. The response contains a list of active endpoints on the end device.

Match_Desc_req
This primitive is used to find end devices implementing application objects matching a supplied profile identifier, input and output cluster list. The request can be either unicast or broadcast. Upon receipt the end device shall compare the profile identifier and list of input and output clusters to the simple descriptor of each active endpoint. If one or more endpoints match, a response, `Match_Desc_rsp`, is unicast and includes a list of the matching endpoints.

Though very useful in a larger setup, these service primitives will not be implemented as they are not needed for our LSM to bind, send and receive data. For full compliance with the ZigBee standard these primitives should be implemented.

3.6 Headers

The following sections describe the general frame format in the different layers of the protocol stack.

3.6.1 ZDO Header

The ZDO uses the MSG service to send messages. The MSG format only defines one header field which is 8 bits in length and holds the number of bytes contained in the rest of the packet. The contents of the rest of the packet depend on the service primive used and is described in [23]. The MSG format is illustrated in Figure 8.

3.6.2 APS

The general frame format of an APDU is illustrated in Figure 12.

<table>
<tr><th>Octets:1</th><th>0/1</th><th>0/1</th><th>0/2</th><th>0/1</th><th>Variable</th></tr>
<tr><td rowspan="2">Frame control</td><td>Destination endpoint</td><td>Cluster identifier</td><td>Profile identifier</td><td>Source endpoint</td><td rowspan="2">Frame payload</td></tr>
<tr><td colspan="4">Addressing fields</td></tr>
<tr><td colspan="5">APS header</td><td>APS payload</td></tr>
</table>

Figure 12: APDU frame format

<table>
<tr><th>Octets:1</th><th>0/1</th><th>1</th><th>1</th></tr>
<tr><td>Frame control</td><td>Destination endpoint</td><td>Cluster identifier</td><td>Source endpoint</td></tr>
<tr><td colspan="4">APS header</td></tr>
</table>

Figure 13: APS acknowledgement header format

Frame control
The frame control field is always present, as it is used to identify the contents of the rest of the header. The format of the frame control field is illustrated in Figure 14, and described below.

Bits: 0-1	2-3	4	5	6	7
Frame type	Delivery mode	Source endpoint present	Security	Ack. request	Reserved

Figure 14: APS frame control field

- *Frame type*
  This field can have one of the following values: *data*, *command* or *acknowledgement*. In our implementation we will only be using data and acknowledgement, since command is used as part of the security services [16, p. 29].
- *Delivery mode*
  Delivery mode can be either unicast, indirect addressing or broadcast transmission. We will be using all three.
- *Source endpoint present*
  This field indicates whether the source endpoint is included in the header. Since we always include the source endpoint, this field will always be set to true.
- *Security*
  This field indicates whether security is used or not. We have not implemented any security services, so this field will always be set to false.
- *Acknowledgement request*
  This field indicates whether an acknowledgement is requested or not. It is not possible to request an acknowledgement to another acknowledgement as this could lead to an infinite loop of acknowledgements. It is also not possible to request an acknowledgement to a broadcast message, since this could lead to flooding of the network.

Destination endpoint
The destination endpoint shall be included, unless the message is sent using indirect addressing. When using indirect addressing, the coordinator uses its binding table to find the destination endpoint.

Cluster identifier
The cluster identifier shall be included in data and acknowledgement frames. Since we only use data and acknowledgement frames, it will always be present in our implementation.

Profile identifier
The profile identifier shall only be included if broadcast addressing is used, since it is used to decide which endpoints should receive the message.

Source endpoint
According to [16, p. 26], the source endpoint shall be present in all data frames and should also be included in acknowledgement frames, see [16, p. 28], hence it will always be present in our implementation.

An acknowledgement frame only consists of a header. The format of the acknowlegement frame is illustrated in Figure 13

Frame control
The frame control is always present and has the same format as described above.

Destination endpoint
Destination endpoint shall only be omitted if indirect addressing is in use. As we only receive messages sent using direct addressing this field will always be present.

Cluster identifier and Source endpoint
Cluster identifier and source endpoint shall always be present.

3.6.3 NWK

The format of an NPDU is illustrated in Figure 15.

<table>
<tr><td>Octets: 2</td><td>2</td><td>2</td><td>0/1</td><td>0/1</td><td>Variable</td></tr>
<tr><td rowspan="2">Frame control</td><td>Destination address</td><td>Source address</td><td>Broadcast radius</td><td>Broadcast sequence number</td><td>Frame payload</td></tr>
<tr><td colspan="4">Routing fields</td><td rowspan="2">NWK payload</td></tr>
<tr><td colspan="5">NWK header</td></tr>
</table>

Figure 15: NPDU frame format

Frame control, Destination address and Source address
These values shall always be present. The frame control field is illustrated in Figure 16, and described below.

Bits: 0–1	2–5	6	7-8	9	10-15
Frame type	Protocol version	Discover route	Reserved	Security	Reserved

Figure 16: NWK frame control field

- *Frame type*
  Frame type can be either data or NWK command. In our implementation this field will only be set to data since NWK command is used for routing which is employed by routers and coordinators [17, p. 82].

- *Protocol version*
  Protocol version is set to 1.0.
- *Discover route*
  This field is set to false since this is an end device and thus does not employ routing.
- *Security*
  This field is set to false since we do not employ security.

Broadcast radius
This field shall only be present if the packet is a broadcast packet. This is an integer indicating how far in the network a packet should be broadcast, e.g. if set to 1 the message will only be broadcast to the immediate neighbours. Broadcast radius can be compared to the Time to Live (TTL) in TCP/IP.

Broadcast sequence number
This field is an integer and should be incremented by one for each broadcast. It is used as part of broadcast retransmission and will not be discussed further, cf.Section 3.4.

3.7 Buffer Management

When implementing a network protocol stack, there is a need to address the problem of sending and/or receiving multiple messages concurrently. There are two options to take into consideration:

- Allowing only one message to be sent/received at a time, or
- Allowing multiple messages to be sent/received concurrently, using buffer management.

When the application object, using the ZigBee protocol stack, is not intended to transmit concurrent messages, as in the case of the Light Sensor Monochromatic, it would be wasteful to implement a buffer manager. Not implementing a buffer manager could lead to messages not being sent, which the application programmer must be warned about.

In contrast, an application requiring concurrent message transmission or burst transmissions, e.g. object detection and tracking, would benefit from buffer management. Whether to use buffer management or not, depends on the usage of the protocol stack and network.

Generally speaking, the message throughput in a ZigBee network is kept at a minimum to minimize energy consumption. Especially battery powered end devices are optimized to send and receive few messages during normal operation, compared to other wireless network standards.

With this in mind, the use of the LSM, and our goal of minimizing the overall memory usage, we opt for the simpler option of only allowing a single message transmission at a time. This choice places the following constraints on our implementation:

1. If a message is currently being sent or the transmission of a message has not yet been confirmed, we must reject data requests from application objects and return an error.

2. If a message is currently being received, we cannot receive another message and therefore must reject the message received from the MAC layer.

3. If a message requiring acknowledgement is received and we are currently transmitting a message, we will buffer the acknowledgement and send it after the message has been transmitted.

As we have absolute control over the application object using our protocol stack, we can avoid the use of sending more than one message concurrently. If our implementation were to support either several application objects on the same end device, or an application object with a requirement of sending subsequent messages within a short time, a send buffer would be needed.

As we can only receive one message at a time, due to the use of polling, see Section 3.4.2, the lack of a receive buffer is not a problem. This is a design decision, based on the assumption that the end device is battery powered. If our implementation were to run on mains power, the radio receiver could be enabled at all times, which would increase the message throughput, but would require some kind of receive buffer manager, as messages could be received at any time, cf. [14].

Buffering the acknowledgement is a design decision we make as this incurs a smaller penalty than later having to receive a retransmission of the message we neglected to acknowledge.

As mentioned in Section 4.1, the nesC paradigm discourages the use of dynamic memory management similar to the use of `malloc`. Even though our implementation does not make use of buffer management to send and receive messages, there are several ways to implement the simpler solution we have opted for:

- Statically allocating a receive buffer in each layer, having the maximum size of the payload in each layer, and copying data between each buffer.

- Statically allocating a receive buffer in the NWK layer and a send buffer in the APS layer, each having the maximum size of the MAC message payload. A pointer can then be passed from layer to layer, minimizing the need to have memory allocated separately in each layer.

To avoid sharing memory between protocol layers, we prefer the first option, though this will consume more memory than needed. This is also the approach used in [8].

3.8 Concurrency

We have identified three main areas where concurrency issues can occur:

Receiving beacons when joining a network
As mentioned in Section 3.2, coordinators and routers send out beacons describing their given PAN, when an end device scans for networks. When scanning a channel, if there are more than one coordinator/router in the vicinity of the

joining end device, several beacons could be received simultaneously, thus introducing a potential concurrency issue.

As our test setup is based on the star topology, i.e. only consisting of one coordinator and several end devices, this will not be an issue to us, and we will not consider this further. If used in either tree or mesh topologies this should be addressed.

Sending messages
As mentioned, we limit the number of messages to be sent at a time to one. This design ensures that no concurrency issues can arise when sending data, as a request to send data when another request is pending, will be denied. This does however, limit the throughput of messages, and could be optimized by "pipelining" message data from layer to layer or by using a buffer manager.

Receiving messages
Receiving messages are based on polling, as mentioned in Section 3.4.2. We can therefore be certain that, as long as we can deliver a received message to the application layer faster than the time of the next poll, we will not have any issues of concurrency. Since we control the polling frequency, we do not consider this an issue.

3.9 Duty Cycling

One of the goals of ZigBee is to ensure that the life time of battery powered end devices is measurable in months or years. To reach this goal, the use of duty cycling is central. Duty cycling is the concept of only turning on parts of the end device as they are needed. In this way the radio and other parts of the end device can be turned off to reduce power consumption.

As mentioned in Section 3.4.2, polling is used by end devices to check for messages. The frequency of the polls is dependant on the use of the end device, and in the case of the LSM, the frequency of polls can be expected to be quite low, as the primary task at hand is to send out periodic light readings, not to be programmed (which requires messages to be received).

On the other hand if the device could expect messages to come in bursts, it would be reasonable to argue that two polling frequencies could be employed. When the coordinator indicates that no data is available the polling frequency should be low, while a higher frequency is used when the coordinator indicates that a message is available.

4 TinyOS, nesC and the Freescale nodes

Implementing the ZigBee protocol stack in TinyOS and nesC requires us to study the concepts and ideas behind both, in order to utilize them in our implementation. The following will be a brief introduction to TinyOS and nesC. Furthermore we will describe the Freescale MC13192-EVB platform and how we have implemented support for the switches (buttons) on this.

An application implemented in TinyOS is based on a number of *components*, e.g. Leds, Timers, ADCs (Analog-to-Digital Converter), etc. These components are reusable from one application to another. Applications are formed by wiring together components to suite the task at hand.

Components can be abstract concepts such as an implementation of directed diffusion (consisting of many different components) or a low level wrapper for a hardware component, such as the UART.

The implementation of components is based on *tasks*, *commands* and *events*. Long running computations should generally be deferred to tasks. Tasks are *posted* to a task queue, after which control is immediately returned to the posting component. The TinyOS task scheduler is based on simple FIFO task execution. When no tasks are pending to be executed, the scheduler puts the processor to sleep, until the next interrupt is received, cf. [10]. Tasks run to completion and cannot preempt each other, essentially making them synchronous with respect to other tasks. The use of tasks causes TinyOS to only have non-blocking operations.

Commands are *called* to execute a given functionality in another component.

Components wrapping hardware *signal* events in response to hardware interrupts. These events run to completion and can preempt tasks and other events. When events occur in response to interrupts, they are marked with the `async` keyword [7].

When long-latency operations are used, a technique of *split-phase* operation is employed. Here commands are used to initiate the requested action, e.g. *component*`.request`, essentially posting a task and returning immediately. Events are then *signaled* in response to the completion of the split-phase operation, e.g. typically using *component*`.requestDone`. These kinds of events do not preempt as those caused by hardware interrupts.

4.1 nesC

nesC uses two concepts to represent components: *modules* and *configurations*. Modules contain the code for a single component whereas a configuration is used to *wire* components together. An application can use a configuration wiring one or more components together as a component in itself. A *top-level configuration* wires all components in the application together.

A module implements one or more *interfaces*. Interfaces describe the commands and events provided by a component. Configurations will wire modules using a given interface to a component providing an implementation of this interface.

To allow for runtime event dispatching, *parameterized interfaces* can be employed. The configuration wires components to unique instances of a module,

identified by an integer. Events can then be signaled to components based on values obtained at runtime.

The concurrency model of nesC allows for static compile time detection of race conditions. These can be handled using `atomic` sections to turn off hardware interrupts in a block of code, or by converting the conflicting code into tasks, reinstating atomicity.

The static analysis prohibits some of the features used in regular C-programming, especially function pointers and dynamic memory allocation, i.e. the use of `malloc` [8].

As TinyOS is implemented in nesC, it consists of numerous modules. These are compiled with the application as needed, i.e. TinyOS allows for a timer to be used, but code for this will only be included if it is actually used in the application.

Our TinyOS based ZigBee protocol stack will be running on the Freescale MC13192-EVB platform, which we will briefly describe below.

4.2 Freescale MC13192-EVB

The Freescale MC13192-EVB is an evaluation board used to evaluate the platform. The MC13192-EVB is built around the Freescale MC13192 2.4 GHz transceiver which is controlled by the Freescale MC9S08GT60 microcontroller unit (MCU). The MCU has the following components:

- A 40-MHz HCS08 CPU.
- 4 KB RAM.
- 60 KB on-chip programmable FLASH memory.
- Analog to digital converter used by e.g. the light sensor.

We will use the CodeWarrior C-compiler developed by Metrowerks and use the USB port to transfer our compiled code to the unit.

The MC13192-EVB platform uses big endian, whereas IEEE 802.15.4 and ZigBee use little endian [4, p. 46] [17, p. 46]. We will have this in mind when implementing the protocol stack.

4.3 Implementing Switches

The Freescale MC13192-EVB has four switches which we have decided to implement support for as these will be useful to initiate simple binding and other functionalities, e.g. programming the LSM. The hardware specific details are based on [3, pp. 145-150].

No TinyOS reference interface existed, so as a start we defined a simple `HPLKBI`[17] interface, providing one command to initialize the switches and one event that is signaled when a switch is pressed.

Switches on the Freescale MC13192-EVB are implemented using direct paging registers assigned to setting up and using the keyboard interrupts (KBI).

[17] Hardware Presentation Layer, Keyboard Interrupt

Initialization of the KBI is done in `init()`, enabling interrupts for all four switches.

The keyboard interrupt handler is registered using the TinyOS `TOSH_SIGNAL` macro and will be signaled when a switch is pressed. Jitter can occur when pressing switches, which is due to the pressure on a switch fluctuating within a short period of time, triggering multiple interrupts for the same "intended" push on the switch. To prevent this, the value of `PTAD`, containing which switch is pressed, is saved and busy waiting is employed for 500 microseconds. After this, the value of `PTAD` is compared to the previously saved value. If these are the same, we signal the `switchDown()` event with the number of the switch pressed.

When receiving an interrupt for the KBI, no more KBI interrupts can be generated before an acknowledgement for the interrupt is registered. This is done by setting `KBISC_KBACK` to 1. This also means that the body of the interrupt handler does not need a surrounding `atomic` statement as we cannot receive any KBI interrupts while we are already in the process of handling one. This is very useful as we would otherwise have had to turn off all hardware interrupts, potentially missing interrupts from the radio or other components.

5 ZigBee Protocol Stack Implementation

In this section we will describe the overall structure of our implementation and the programming principles we follow. These principles are used extensively in the implementation, and knowing the rationale behind their use will ease the understanding of the implementation.

We have chosen to use the names of arguments and primitives as they are written in the ZigBee standard to have a clear connection between the standard and the implementation, cf. Figure 6.

The source code can be obtained by contacting the authors.

5.1 Overall Structure

We have chosen the approach of having a module per protocol stack layer. This leads to the component graph for our implementation, illustrated in Figure 17. Here the `Freescale802154C` is the MAC wrapper for the Freescale IEEE 802.15.4 library.

Black arrows represent commands being called from the next higher layer while white arrows represent events signaled to the next higher layer. A number next to an arrow indicates the number of commands or events in the interface, e.g. the `Timer` interface has eight commands and one event.

A single configuration wires the layers together in a configuration used by the actual application objects. This configuration is named `ZigBeeEndDevice` and represents the actual protocol stack. Application objects are wired to the protocol stack and each object "register" a simple descriptor used by the ZDO.

5.2 Call Depth

One of the problems when implementing the protocol stack was how to implement a series of subsequent events/commands, e.g. the indication of data in the MAC, NWK and APS layers and subsequently in the recipient application object(s).

Originally, we had considered a simple, blocking call flow, e.g. having several nested events. This approach turned out to be naïve as we experienced stack overflow on the Freescale MC13192-EVB when handling nested function calls. The call depth could easily reach four-five levels, which depleted the stack[18].

Following the guidelines in [8], we chose to redesign the functional flow to use tasks instead of nested calls. When a command is called or an event is signalled, a task is posted to do the processing and control is returned to the calling primitive, i.e. split-phase operation as described in Section 4.

The current version of nesC does not allow tasks to take arguments, it was therefore necessary to save the arguments given to the commands and events in global variables. These variables are set to the values of the command/event arguments after which the corresponding task is posted, to perform further processing and possibly call commands or signal events. In the next version of nesC and TinyOS (version 2.0), support for arguments to tasks will be available.

[18] This has later been identified to be a compiler issue.

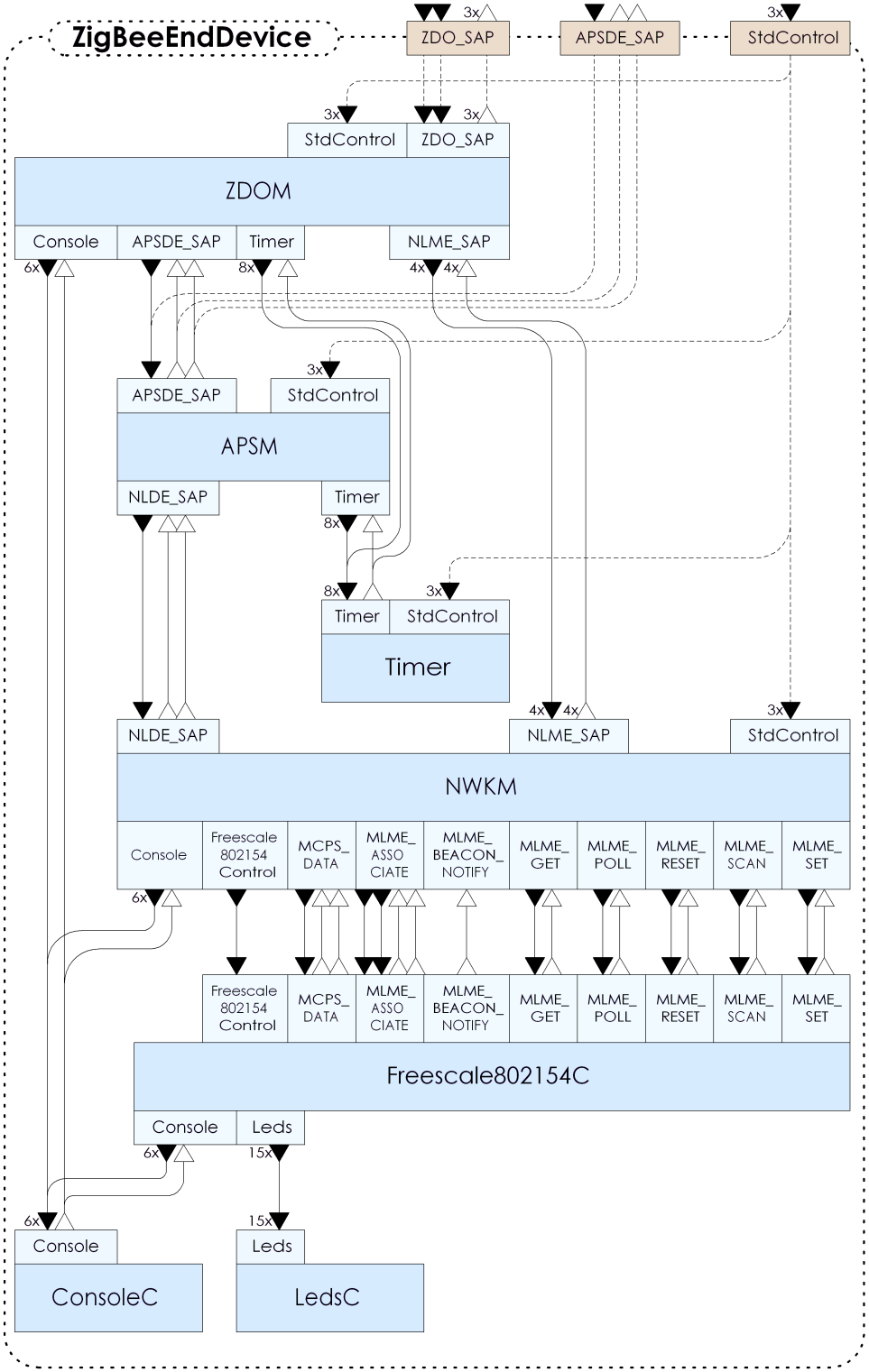


Figure 17: The ZigBee end device component graph

5.3 Data Structures

Where possible `unions` have been used instead of `structs` to save memory. Specifically, the saved arguments to the primitives used to join the network share the same memory space as these follow in sequence and cannot occur concurrently.

Another use of `unions` is when mapping headers to the message sent/received. In this way, it is possible to have access to the particular elements (short address, PAN identifier, etc.) of the different message headers in the code, using the same overall message structure, regardless of whether sending is to happen using unicast or broadcast addressing.

As ZigBee messages, including headers, are little endian we simplify our implementation by representing all internal ZigBee information received and sent using little endian, and requiring the application programmer to do this as well. This is also the approach followed by Freescale in their MAC layer library [4, p. 2-3].

5.4 Medium Access Control Layer

Before implementing the ZigBee protocol stack, a TinyOS wrapper for the Freescale MAC library was needed. Work on a wrapper had already been started as part of a sensor networks course at DIKU[19], but it was far from finished.

The interfaces provided by the TinyOS wrapper were originally defined by Joe Polastre at the University of California, Berkeley. These interfaces are direct translations of the primitives defined in [11] and form the basis for our implementation of the wrapper.[20]

At the time of writing (June 2005) a discussion, headed by the Sensor Networks group at DIKU, is taking place about the redefinition of these interfaces, making them more general and allowing for an easier way to only use the primitives needed from the IEEE 802.15.4 standard. When these interfaces have been finalized, it would be sensible to adapt our wrapper to the new interfaces.

The Freescale MC13192-EVB nodes we are using do not have an onboard EEPROM containing initialization for the MAC layer, including the IEEE extended address, thus this has to be initialized, which is done in `Control.init()`. This gives us complete control over the extended address of a device, which comes in handy when debugging.

The MAC library signals interrupts for the MLME-SAP and the MCPS-SAP with `MLME_NWK_SapHandler` and `MCPS_NWK_SapHandler`, respectively. When handling these interrupts, the message received is queued using mechanisms provided by the library. This in essense provides dynamic memory allocation, though it is intended for use in the MAC layer and thus we do not use it in other modules. After queuing, the message a task is posted to handle it, i.e. `processMlmeNwk` or `processMcpsNwk`.

[19] Department of Computer Science, University of Copenhagen

[20] We identified a minor bug in the MAC constants header file, `IEEE802154.h`, included in the TinyOS distribution while analyzing these interfaces. The `SuperframeSpec` element of the `PANDescriptor_t` was incorrectly declared as an 8 bit unsigned int (`uint8_t`), which should have been a 16 bit unsigned int (`uint16_t`). This has been reported to Joe Polastre.

In the tasks handling received messages, the message is de-queued and a simple `switch` statement checks the message type and the associated function is called in order to process the message. It should be noted that the use of the `switch` statement, which we cannot avoid, makes it difficult for the compile time analysis of nesC to perform optimizations. This is caused by the design of the library, and can only be avoided if the MAC library is re-implemented, preferably in TinyOS. However, this type of design is not uncommon for higher layer protocols in TinyOS, cf. [13].

Special care is needed with regard to beacon indications, as the Freescale library returns PAN descriptors in a non-standard order [4, p. 3-9]. To ensure compatibility between our implementation and other potential uses of the wrapper, we convert the Freescale PAN descriptor to follow the standard defined in the IEEE 802.15.4 standard [11, pp. 76-77].

We have only implemented the MAC primitives illustrated in Figure 6. All other primitives received will be discarded, as they are not needed in our implementation.

5.5 Network Layer

The network layer provides two interfaces, `NLME_SAP` and `NLDE_SAP`, corresponding to the service access points illustrated in Figure 6. The NWK module uses all the interfaces provided by the MAC layer module, see Figure 17.

In our implementation of the network layer we deviate from the standard with regards to the `NLME_GET_request` and `NLME_SET_request` primitives. These run synchronously and return the requested value, instead of signaling the associated confirm event. This is to avoid non-sequential program flow and is also the approach used by the Freescale library with regard to similar primitives in the MAC layer [4, pp. 3-3, 3-4].

As mentioned in Section 5.3, the arguments when calling a command and the subsequent posting of a task are shared in a `union` for:

- `MLME_ASSOCIATE_request`
- `MLME_ASSOCIATE_confirm`
- `MLME_POLL_confirm`
- `MLME_SCAN_confirm`
- `NLME_JOIN_request`
- `NLME_NETWORK_DISCOVERY_request`
- `NLME_SYNC_request`

These either occur in sequence or can in no way be used at the same time, e.g. `NLME_ASSOCIATE_request` does not conflict with `NLME_SYNC_request`, as the former must have been completed before the latter can occur.

The network descriptor list and neighbor table have a fixed size of five entries each, as this is the maximum number of PAN descriptors returned by the `MLME_SCAN.confirm` primitive in the Freescale MAC library [4, p. 3-6]. We have chosen to only scan for networks once. If we were to implement multiple network scans on joining a network, as described in [22, p. 26], we would have to extend these beyond five entries, thus consuming more memory.

5.6 Application Support Sub-Layer

We have modelled the Application Support Sub-Layer (APS) after Figure 6, thus providing the two interfaces `APSME_SAP` and `APSDE_SAP`, representing the service access points in the protocol stack. The latter interface is parameterized to allow for multiple application objects to request or receive data.

As described in Section 3.7, sending data using `APSDE_DATA_request` is limited to one message at a time, with constraints regarding pending acknowledgements. To enforce this we use two flags, `apsdeDataRequestBusy` and `ackResponsePending` respectively.

When `APSDE_DATA_request` is called, we first check, `apsdeDataRequestBusy`, to see whether we are already busy sending data or waiting for an acknowledgement. Secondly, we check, `ackResponsePending`, to see whether we are waiting to send an acknowledgement for a message received. If either of these are true, we reject the request to send data, by issuing `APSDE_DATA_confirm` with a status of `DATA_REQUEST_BUSY`, as mentioned in Section 3.4.3, otherwise we grant the data request and set `apsdeDataRequestBusy` to true.

The check of whether the data request is busy, is surrounded by an `atomic` statement, as data requests could potentially occur simultaneously.

Retransmission

When sending a message requesting acknowledgements, the `numFrameRetries` counter is initialized to 0, the `ackDataRequestBusy` is set to true and the `ackTimer`, used to retransmit messages, is set to fire after 15 seconds.

If the timer fires, we increment `numFrameRetries` and check if the number of retries exceeds `apscMaxFrameRetries` (default value is 3). If so the `APSDE_DATA_confirm` primitive is signaled to the requesting endpoint, with a status set to `NO_ACK`. If we need to retransmit the message, we merely have to post the `apsdeDataRequest()` task as the data will be stored in the arguments from the original `APSDE_DATA_request`, due to our one-message-only policy.

Acknowledgements are received, as other messages, in the `APSDE_DATA_indication` primitive. If the message received is an acknowledgement of the original message, we stop the `ackTimer` and signal the `APSDE_DATA_confirm` primitive to the endpoint requesting the original data transfer. Otherwise we consider the message to have been lost and signal `APSDE_DATA_confirm` with a status of `NO_ACK` [16, p. 36].

Acknowledgements

When receiving a message requesting acknowledgements through the `APSDE_DATA_indication` primitive, we use the `ackResponsePending` flag to check whether we are already waiting to send an acknowledgement.

If we do not have a pending acknowledgement to send, we save the values needed to send the acknowledgement in `ackFrameArguments` and set `ackResponsePending` to true. We then post the `ackFrame()` task which checks whether `apsdeDataRequestBusy` is true, indicating that a data transfer is taking place. If this is the case, the task will be reposted, otherwise `apsdeDataRequestArguments` is set up to construct the acknowledgement frame and the `apsdeDataRequest()` task is posted.

On the other hand, if we have a pending acknowledgement, we ignore the message. This is based on the assumption that the message will be retransmitted and that we have presumably finished sending the pending acknowledgement in the meantime.

Broadcast
When receiving a broadcast message, we use the profile identifier to check whether the broadcast was intended for the ZDO. If so, we only signal the `APSDE_DATA_indication` primitive to endpoint 0 as there can only be one ZDO. Otherwise we loop through the list of active endpoints, signaling the data indication to each endpoint with a matching profile identifier.

5.7 ZigBee Device Object

The ZigBee Device Object module provides a parameterized interface called `ZDO_SAP`. This interface is used to provide access to public primitives in the ZDO.

The only service primitive offered by the `ZDO_SAP` is the optional `End_Device_Bind_req` primitive, as this is required by the Light Sensor Monochromatic profile [20, p. 15]. In addition to the above mentioned service primitive, the `ZDO_SAP` interface provides two events: `End_Device_Bind_rsp` and `Network_Joined`, where the latter is not defined in the ZigBee standard, but is used to indicate that application objects can begin normal operation mode, including binding.

`ZDO_SAP` is parameterized as this simplifies the process of signaling `End_Device_Bind_rsp` to the respective endpoint and `Network_Joined` to all active endpoints.

To poll the coordinator for data we use the `syncTimer` and set the timer to fire every 10 seconds. The timer is started in the `NLME_JOIN_confirm` primitive as the `End_Device_annce` we send out after having joined the network is sent using the `APSDE_DATA_indication` primitive and hence the response must be polled from the coordinator.

There can only be one pending `End_Device_Bind_req` which is controlled using `endDeviceBindReqBusy`.

6 Light Sensor Monochromatic Implementation

In our implementation the Application Framework, see Figure 6, consists of one endpoint, namely the Light Sensor Monochromatic (LSM).

6.1 Device Configuration

In addition to the `ZigBeeEndDevice` component we will use following components:

`TimerC`	To trigger time-fixed events.
`Light`	To read the light sensor.
`HPLKBIC`	To handle switches being pressed.
`LedsC`	To indicate events on the node leds.
`ConsoleC`	To output debug information to the console.

Our final LSM component will rely on the `APSDE_SAP` and `ZDO_SAP` interfaces, described in Sections 5.6 and 5.7, and will in turn provide a `StdControl` interface, used for initializing the component.

The component graph for our light sensor is presented in Figure 18 and the complete graph for the entire application can be found in Figure 19.

6.2 Initial Considerations

When implementing an application on an end device, it is necessary to provide a simple descriptor, see Section 2.2. The contents of the LSM simple descriptor is as follows [20, 19]:

Application profile identifier	`0x0001`
Application device identifier	`0xFFFF`
Application input clusters	{`0x07`}
Application output clusters	{`0x06`}

To interact with the device, we use the switches, e.g. to perform simple binding, see Section 3.3.

6.3 Core Functionality

After these initial considerations we can proceed with a description of the light sensor implementation.

Network Joining
Joining a network is initiated by the ZDO, afterwhich all endpoints, are notified. A succesful join is signaled by the `ZDO_SAP.Network_Joined` event. Besides indicating a successful join, by turning on LED1, we do not carry out any further actions.

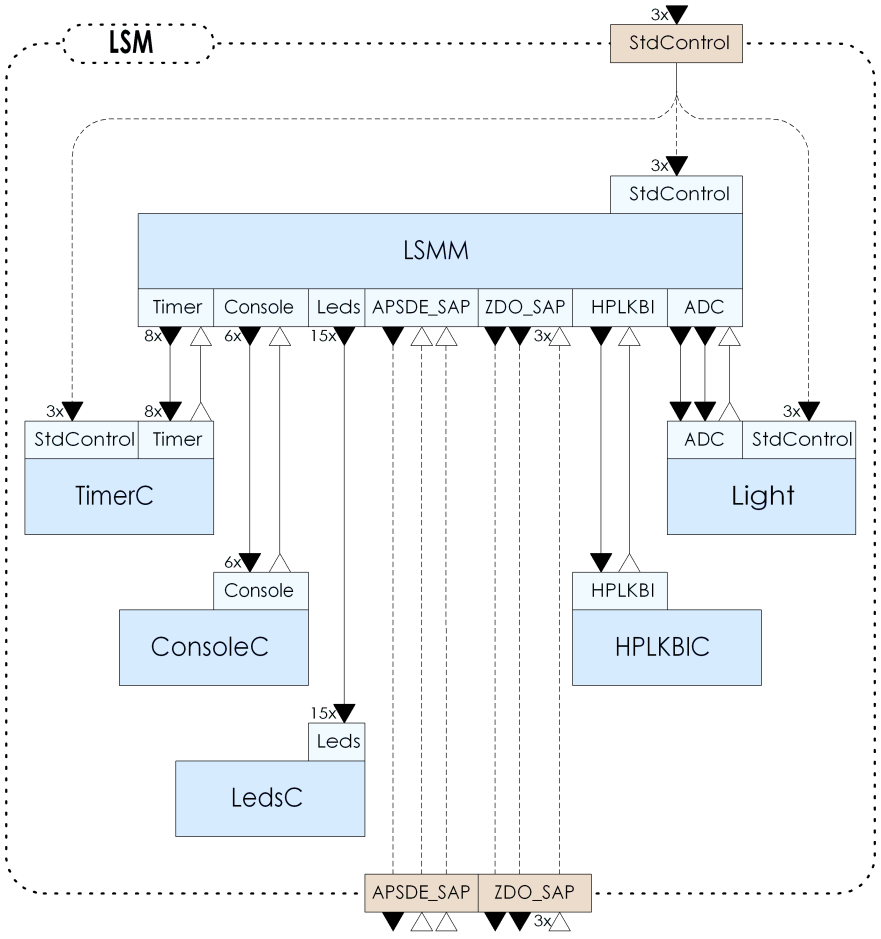


Figure 18: Light Sensor Monochromatic component graph

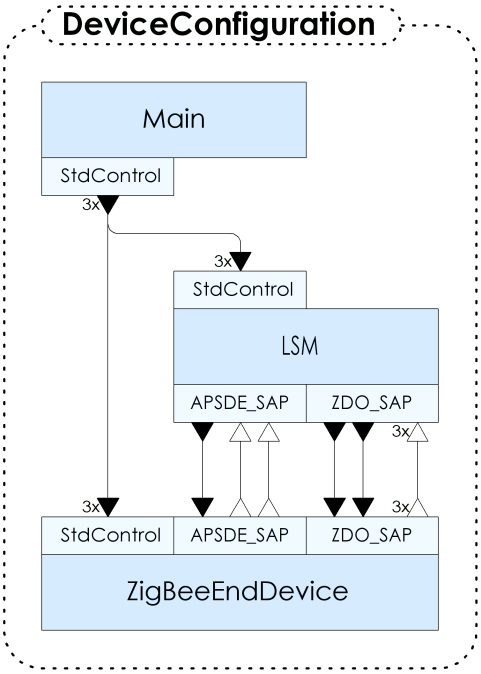


Figure 19: Entire application configuration

Device Binding
When the device has joined a network, device binding can be performed, using the `ZDO_SAP.End_Device_Bind_req`. Binding is performed manually by pressing switch 1 on the devices that need to bind.

When binding to another end device is successful, we receive a `ZDO_SAP.End_Device_Bind_rsp` event indicating `SUCCESS`. This is indicated by turning on LED2.

Data Indication
Received data from the APS layer is signaled by `APSDE_DATA_indication`. First our LSM checks if the specified cluster is valid, and discards the message if this is not the case. If we are not already handling a message, a task, `HandleAFFrame`, handling the received data is posted.

When this task is activated, it ensures that it is a KVP frame, see Figure 8, checks the number of included transactions, extracts them one by one and posts tasks handling each of the transactions. When all tasks have been posted new messages are allowed to arrive.

A function handling a KVP transaction, `HandleKVPTransaction`, will first check if the included attribute identifier is valid, and then use a `switch` to perform the appropiate action. Most of the attributes are straightforward to handle, but there are a few things worth noting:

- The time between light level reports is handled by the timer component, which will be described below.
- To determine if *MinLevelChange* or *Min/MaxThreshold* apply, we will

need to save the last outputted value. We also have to account for this when updates are made to the *Offset*.

- No guidelines about factory defaults are presented in the device description so we have chosen not to output, until a *ReportTime* has been signaled from a bound device.

Timer Events
The sensor readings are controlled by a timer component, `Timer`, which fires events at a given rate. Setting the frequency of the timer can control how often sensor readings occur. By simply stopping the timer we can ensure that no output is created since the sensor will not be read. This practice is suitable for implementing the *Override*-mode, specified in Section 3.1.1. Enabling output again (*Auto*-mode) is simply performed by starting the timer.

When the timer fires, we request a light level reading from the sensor using `Light.getData`. The light reading will eventually be returned from the light sensor.

Light Sensor Data Ready
When a light reading is done a `Light.dataReady` is signaled. Since this is an asynchronous event, we save the light data in a global variable and post the `handleLightReading` task to handle this.

When this task is activated, it checks whether the conditions for outputting the reading are met. We correct the reading according to the *Offset* and check whether the conditions for outputting are met. This is handled atomically to avoid the LSM from being programmed during these checks. If the conditions for output are satisfied, we create the data for the *LightLevelLSM* attribute, wrap it in a KVP message and send it using indirect addressing.

6.4 Test Applications

To test that our LSM device can bind, send and receive data, we implement two test applications objects and put these on two other nodes:

`LSMProgram`	Used for programming the LSM.
`LSMConsumer`	Used for receiving the sensor readings and outputting using the `Console` component.

6.4.1 LSMProgram

The component graph for this device is almost the same as the one for our LSM, since it basically performs the same actions, i.e. joining a network and binding with another device.

After successful binding the device should be able to program the LSM, having implemented the *Output:LSMProgram* cluster. By pressing one of switches 1-3 it is possible to send instructions to the LSM. The default values sent are:

Switch	Attribute	Action
1	n/a	Perform binding.
2	*ReportTime*	Set report time to 20 seconds.
3	*Min-/MaxThreshold*	1st press, sets min and max threshold to 150 and 750 (aggregated transaction).
	MinLevelChange	2nd press, sets min level change to 10%.
4	*Override*	1st press, stop transmission of readings.
	Auto	2nd press, start transmission of readings.

6.4.2 LSMConsumer

After having programmed the LSM, it should output values according to our initial intentions. To test this, we need a device that consumes the LSM output. For this purpose we have the LSMConsumer, which implements the *Input:LightLevelLSM*.

Like the LSMProgram device, the LSMConsumer is able to join a network and bind with devices. After this is done, it will be able to accept incoming light readings, verify the packet structure and output the received light level as a decimal number.

7 Evaluation

The evaluation of our implementation will focus on two areas:

- Testing of functionality, e.g. acknowledgements, retransmissions, broadcast messages, etc.
- Implementation size, i.e. memory usage and code size.

7.1 Testing Functionality

We will be testing our implementation using functional testing[21], as we have performed on-going tests for obvious issues during implementation.

In the following we will describe the test scenarios we have identified during the work of implementing the ZigBee protocol stack and light sensor application object. These scenarios are based on the analysis and design described in Section 3 and can be grouped into the following seven overall testing scenarios:

- Joining a network.
- Device binding.
- Sending data.
- Receiving data.
- Acknowledgements.
- Concurrency.
- Light Sensor Monochromatic.

The results of our evaluation are summarized in Table 6

Our test setup is a single ZigBee coordinator and one or more end devices. The coordinator is compiled from beta code found on Freescale Semiconductor's website during March 2005. The code was developed by "Figure 8 Wireless, Inc." and implements a beta version of ZigBee v. 0.92. This is the final draft before the standard was approved on the 14th of December 2004. We have not identified any differences in the documentation for ZigBee v. 0.92 and ZigBee v. 1.00, thus this should not be a problem[22] Some bugs and non-ZigBee standard behaviour have been observed in the coordinator, which we will note in our test evaluation below.

To debug and test our implementation we have used a packet sniffer to manually decode the messages sent. Regarding debugging our implementation, we have not had access to a Metrowerks CodeWarrior IDE and a debugging device, both of which are required to perform inline debugging. Instead we have been forced to use the `Console` module to dump data and print debug messages to the serial port, etc.

[21] Also known as *black box* testing.
[22] The code has later been removed from the Freescale website.

Test scenario	Result
Joining a PAN	✓
Binding between two end devices	✓
Sending a unicast message, direct addressing	✓
Sending a unicast message, indirect addressing	✓
Sending a unicast message, broadcast endpoint	✓
Sending a broadcast message	✓
Polling, no data available	✓
Polling, data available	✓
Sending a unicast message, requesting acknowledgement	✓
Retransmission of unacknowledged message	✓
Busy, already sending non-acknowledged message	✓
Busy, waiting for acknowledgement	✓
Busy, waiting to send acknowledgement	✓
LSM sends out light sensor readings	✓
LSM is programmable (start/stop, interval, thresholds, etc.)	✓

Table 6: Evaluation results of our implementation

7.1.1 Joining a Network

Description: To test the ability for one end device to join a PAN, we turn on the coordinator and let it start the PAN. We then turn on the end device which will automatically scan for a network and join it.

Result: The joining end device is informed of the PAN identifier of the coordinator, is assigned a unique short address and receives the `End_Device_annce_rsp`. The status returned by the coordinator for the `End_Device_annce_rsp` primitive is set to `0x01` which is not a valid ZigBee return value, cf. [23, p. 42]. We believe this is a bug in the coordinator.

7.1.2 Device Binding

Description: To test the ability to perform a simple bind between two end devices, we use two matching[23] application objects, i.e. the LSM and LSMConsumer mentioned in Section 6 and Section 6.4.2. An `End_Device_Bind_req` is issued from both end devices.

Result: When performing the simple bind, we correctly receive an `End_Device_Bind_rsp` primitive from the coordinator, with a status of success.

7.1.3 Sending Data

Sending a unicast message, direct addressing
Description: To test the ability to send a unicast message from one end device to another, we record the short addresses assigned to each end device and use these to send a message from one end device to the other. We enable the receiver to be on when idle, `macRxOnWhenIdle` [11, p. 137] as the message will

[23] With regard to input clusters and output clusters.

not go through the coordinator.

Result: We correctly receive an indication of data at the endpoint meant to receive the message.

Sending a unicast message, indirect addressing
Description: To test the ability to send a unicast message from one end device to another using indirect addressing we bind two application objects and send a message from one end device to the other. To receive the message we enable polling as specified in Section 3.4.2

Result: We correctly receive an indication of data at the endpoint meant to receive the message.

Sending a unicast message, broadcast endpoint
Description: To test the ability to send a message to the broadcast endpoint, we use direct addressing and send a message from one end device to another with the destination endpoint set to `0xFF`, cf. Section 3.4.

Result: We correctly receive the message at the application on the second end device (only one was implemented, but it did obviously not have `0xFF` as its endpoint).

Sending a broadcast message
Description: To test the ability to send a broadcast message, we let two end devices join the PAN having application objects with the same profile identifier. Then we send a message with the destination address set to `0xFFFF`, cf. Section 3.4.

Result: We correctly receive the message at the second end device. The sending end device also receives the broadcast message, which is because we have not implemented a filter on the broadcast sequence number also used to retransmit broadcasts, cf. Section 3.4.

7.1.4 Receiving Data

Polling, no data available
Description: To test polling, we let an end device join the network and start polling the coordinator.

Result: We correctly receive a status of `NO_DATA`, cf. [11, p. 111].

Polling, data available
Description: This has already been tested in the above-mentioned scenario for sending and receiving a message using indirect addressing. It should be noted that the coordinator sometimes, seemingly at random, stops sending data to end devices polling for data, though we can observe, using the packet sniffer, that data is sent to the coordinator which is intended for another device (using indirect addressing). We conclude that this is a bug in the coordinator, as there

is no difference in the message we send to the coordinator over time, and there is no pattern of when this happens.

7.1.5 Acknowledgements

Sending a unicast message, requesting acknowledgement
Description: To test the ability to send an acknowledgement we request the transfer of a unicast acknowledged message from one end device to another.

Result: We correctly receive an acknowledgement for the message sent.

Retransmission of unacknowledged message
Description: To test the ability to retransmit a message when an acknowledgement is not received we request the transfer of a unicast acknowledged message from one end device to another, using direct addressing. The destination address is set to an invalid short address.

Result: We correctly retransmit the message three times, afterwhich an `APSDE_SAP.APSDE_DATA_confirm` is signaled with a status of `NO_ACK` at the sending end device, cf. [11, p. 59].

7.1.6 Concurrency

Busy, already sending non-acknowledged message
Description: To test that the implementation only allows one message to be sent at a time, we perform two consecutive `APSDE_SAP.APSDE_DATA_request`s.

Result: We correctly send the first request and return an `APSDE_SAP.APSDE_DATA_confirm` with status set to `DATA_REQUEST_BUSY` to the second request.

Busy, waiting for acknowledgement
Description: To test that the implementation disallows an `APSDE_SAP.APSDE_DATA_request` when we have sent a message requesting acknowledgement that has not yet been confirmed, we send a message requesting acknowledgement to an invalid short address. We set up a timer to try to send another message after a short delay.

Result: We correctly send the first request and return an `APSDE_SAP.APSDE_DATA_confirm` with the status set to `DATA_REQUEST_BUSY` to the second request.

Busy, waiting to send acknowledgement
Description: To test that the implementation disallows an `APSDE_SAP.APSDE_DATA_request` if we are waiting to send an acknowledgement we send a message requesting acknowledgements to an end device on which we have disabled the actual sending of the acknowledgement. Upon receipt of the message at the recipient device we try to send a message.

Result: As expected the data request at the recipient device is followed by a `APSDE_SAP.APSDE_DATA_confirm` with the status set to `DATA_REQUEST_BUSY`.

7.1.7 Light Sensor Monochromatic

LSM sends out light sensor readings
Description: To test that the LSM sends out light sensor readings, we bind the LSM and LSMConsumer described in Section 6 and Section 6.4.2.

Result: We correctly receive light sensor messages on the LSMConsumer end device.

LSM is programmable (start/stop, interval, thresholds, etc.)
Description: To test that the LSM is programmable, we deploy the testbed described in Section 6.4, i.e. LSM, LSMConsumer and LSMProgram.

Result: We can correctly program the attributes in the LSM using the LSMProgram end device. This is reflected in the received light sensor readings in LSMConsumer.

7.2 Implementation Size

To evaluate the code size of our implementation, we use the output given by the compiler. This contains the data size (RAM), code size of our implementation without inlining and the total code size including the Freescale MAC library.

We have compiled our implementation using TinyOS v. 1.1.11, nesC v. 1.2 and the CodeWarrior HC(S)08 compiler. We have set compile flags to optimize for code size.

The final numbers obtained are:

- The code size for our implementation alone, is reported as 10,752 bytes including TinyOS, using 2,174 bytes RAM (including 200 bytes of buffer for the `Console` module and 229 bytes for application debug output).
- The total code size including, including the 18KB from the Freescale MAC library, is 29,620 bytes—significantly less than the 32,768 bytes goal.

The code size of the individual primitives and functions in the code can be obtained from the compiler. The numbers give an idea of the relative memory usage of each module, and are interesting in the context of future work. Table 7 summarizes the code size of the individual modules in our implementation, showing that the NWK layer has the largest code size. Also, as can be seen, we can save quite a lot of redundant memory for the MAC wrapper, if we had a TinyOS based implementation of IEEE 802.15.4.

We have looked into the code size for a Light Sensor Monochromatic end device[24], implemented by the authors of our coordinator, Figure 8 Wireless. The code size of their implementation is 50,246 bytes. This figure includes the Freescale MAC FFD library and support for outputting characters to a small

[24]For reference, the compiled file name is `ZStack_LSM02080 End Device - EVB DIG528.abs`

Component	Code size
Freescale802154	1,554 bytes
NWK	2,159 bytes
APS	1,730 bytes
ZDO	1,090 bytes
LSM	1,522 bytes
HPLKBI	47 bytes
Remaining (TinyOS)	2,650 bytes
MAC library	18,868 bytes
Total	29,620 bytes

Table 7: Code size of implemented modules

display, though this is no different than our implementation having the `Console` module in our implementation.

We consider our code size compared to the Figure 8 Wireless code to be proof that TinyOS can be a viable solution when implementing ZigBee end devices.

8 Conclusion

We have succesfully implemented a reduced ZigBee protocol stack and Light Sensor Monochromatic device supporting, most mandatory service primitives as well as optionals where needed. The implementation has a code size of 29,620 bytes thus meeting our goal of a code size less than 32KB.

The protocol stack and light sensor has been tested as shown in Section 7 and we are convinced that it is fully functional.

Our protocol stack is not fully compliant with the ZigBee standard as it does not support device and service discovery responses, retransmission of broadcasts and further analysis needs to be done before it can be deployed in other network topologies than a star topology. Furthermore the light sensor does not support application level acknowledgements or semi-precision values. We believe that it is possible to extend our implementation to be fully compliant and still keep the code size below 32KB.

Our implementation is not only usable by light sensors but could be used by many other ZigBee devices having similar requirements, e.g. temperature sensors, pressure sensors etc. However in a commercial context this would require that the implementation is made fully compliant with the ZigBee standard.

We believe that this thesis contains valuable information to other software developers implementing a ZigBee protocol stack. Not only have we analyzed and described the functionality needed for the light sensor, we have also proposed feasible interpretations of the ZigBee standard, cf. Section 3.4.

Furthermore, the use of IEEE 802.15.4 in the TinyOS community, is very limited making this thesis an important contribution on the use and applicability of this.

8.1 Future Work

- Extending the implementation to be fully compliant with the ZigBee standard by implementing the above mentioned missing functionality. Given the code size described in Section 7.2 it should be possible to implement the remaining functionality in the remaining 3KB.
- The way we transfer a message from one layer to another is not very efficient. At present they are copied from one memory space to another. This was a design decision which in retrospect seems unwarranted. This could be solved by transfer of ownership or using a buffer manager which could be configured to the size needs of the given application. The use of a buffer manager would allow the implementation to send more than one message at a time, including acknowledgements. This would make the protocol stack more robust in terms of reliability.
- Currently, the components and interfaces used and provided in our implementation are based entirely on the IEEE 802.15.4 and ZigBee standards. This is not necessarily the optimal way as can be seen by the discussions on the TinyOS IEEE 802.15.4 interfaces mentioned in Section 5.4. An

analysis of other ways to organize the protocol stack would be useful to provide a reference interface usable for other platforms.

- It would be interesting to perform power measurements on the LSM to see if the goal of battery life reaching years is within reach.
- Making a component to perform the handling of KVP messages in the application objects could be shared among applications and would make it easier to implement new profiles from scratch, if no reference code was available.
- Finally, implementing the IEEE 802.15.4 standard in TinyOS would allow nesC to perform further optimizations, presumably reducing the overall size of the implementation.

References

[1] College of the Atlantic, University of California at Berkeley and Intel Research Laboratory at Berkeley. *Habitat Monitoring on Great Duck Island*, 2004. `http://www.greatduckisland.net`.

[2] DTU, DIKU, KVL, NCPP and IO Technology. *Hogthrob – Networked on-a-chip nodes for sow monitoring.* `http://www.hogthrob.dk`.

[3] Freescale Semiconductor. *MC9S08GB/GT Data Sheet*, December 2004. `http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC9S08GB60.pdf`.

[4] Freescale Semiconductor. *802.15.4 MAC/PHY Software – Reference Manual*, May 2005. `http://www.freescale.com/files/rf_if/doc/ref_manual/802154MPSRM.pdf`.

[5] Freescale Semiconductor. *802.15.4 MAC/PHY Software – User's Guide*, May 2005. `http://www.freescale.com/files/rf_if/doc/user_guide/802154MPSUG.pdf`.

[6] Gartner, Inc. *Gartner Outlines Most Significant Technology Driven Shifts in the Next Decade*, March 2004. `http://www.gartner.com/press_releases/asset_62507_11.html`.

[7] David Gay, Philip Levis, David Culler, and Eric Brewer. *nesC 1.1 Language Reference Manual*, May 2003. `http://nescc.sourceforge.net/papers/nesc-ref.pdf`.

[8] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. *The* nesC *Language: A Holistic Approach to Networked Embedded Systems*, June 2003. `http://nescc.sourceforge.net/papers/nesc-pldi-2003.pdf`.

[9] Bob Heile. *Emerging Standards: Where does ZigBee fit.* ZigBee Alliance, October 2004. `http://www.zigbee.org/imwp/idms/popups/pop_download.asp?contentID=4524`.

[10] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. *System Architecture Directions for Networked Sensors*, 2000. `http://www.cs.berkeley.edu/~awoo/tos.pdf`.

[11] IEEE. *802.15.4, Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)*, October 2003. `http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf`.

[12] IEEE 802.15.4 WPAN-LR Task Group. *IEEE 802.15 WPAN^{TM} Task Group 4 (TG4)*, January 19th, 2005. `http://www.ieee802.org/15/pub/TG4.html`.

[13] Martin Leopold, Mads Bondo Dydensborg, and Philippe Bonnet. *Bluetooth and Sensor Networks: A Reality Check*, 2003. `http://www.diku.dk/~leopold/work/p072-leopold.pdf`.

[14] Klaus S. Madsen. The Case for Buffer Allocation in TinyOS. Technical report, DIKU, May 2005. `http://www.hogthrob.dk/output/technotes_files/buffer-management.pdf`.

[15] University of California at Berkeley. *Design and Construction of a Wildfire Instrumentation System Using Networked Sensors.* `http://firebug.sourceforge.net`.

[16] ZigBee Alliance. *Application Support Sub-Layer Specification, Version 1.00 – ZigBee Document 03244r9*, December 14th, 2004.

[17] ZigBee Alliance. *Network Specification, Version 1.00 – ZigBee Document 02130r10*, December 14th, 2004.

[18] ZigBee Alliance. *ZigBee Application Framework Specification, Version 1.00 – ZigBee Document 03525r6*, December 14th, 2004.

[19] ZigBee Alliance. *ZigBee Application Profile, Home Control, Lighting, Version 1.00 – ZigBee Document 03540r6*, December 14th, 2004.

[20] ZigBee Alliance. *ZigBee Device Description, Light Sensor Monochromatic, Version 1.00 – ZigBee Document 02080r12*, December 14th, 2004.

[21] ZigBee Alliance. *ZigBee Device Description, Switching Load Controller, Version 1.00 – ZigBee Document 03394r7*, December 14th, 2004.

[22] ZigBee Alliance. *ZigBee Device Objects, Version 1.00 – ZigBee Document 03265r10*, December 14th, 2004.

[23] ZigBee Alliance. *ZigBee Device Profile, Version 1.00 – ZigBee Document 03529r7*, December 14th, 2004.

[24] ZigBee Alliance. *Frequently Asked Questions*, 2005. `http://www.zigbee.org/en/about/faq.asp`.