UNIVERSITÀ DI PISA

FACOLTÀ DI INGEGNERIA

Corso di Laurea Specialistica in Ingegneria Informatica

Curriculum Networking e Multimedia

Implementation of IEEE 802.15.4 security on CC2420

Supervisors:
Prof. Gianluca Dini
Ing. Alessio Vecchio

Candidate:
Cristiano Carnicelli

Anno Accademico 2009-2010

Abstract

During the last ten years, the presence of sensors in common life has become pervasive. Sensor nodes are used in many areas of interest: from military fields to environment monitoring, from medical applications then to domotic uses. One of the most common radio communication protocol designed for Personal Area Network (PAN) is described by the IEEE 802.15.4 standard. Data communication among devices can be protected on a per frame basis, allowing to provide data authenticity and data confidentiality, and configure security mechanisms in a flexible and precise way.

With reference to the IEEE 802.15.4 MAC layer, the security sublayer was implemented during this thesis work, providing the main security features the standard describes. In particular, the TinyOS implementation for the TelosB mote was considered, as well as its CC2420 chipset features. The main goal of this work is exploit the IEEE 802.15.4 security mechanisms both sending and receiving ciphered frames, as well as integrate IEEE 802.15.4 security suite with the security features provided by the CC2420 chipset.

During this work, a simple two nodes network was considered. Besides, ciphering only mode was used, while all security options and parameters were statically set up. Finally, a simple application was developed, in order to test and evaluate the network activity.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last few years, sensors networks phenomenon has grown so much. These systems can be used for military facilities and war fields monitoring, as well as for environment monitoring and patients' record collection. Nowadays this technology can be used by anyone, with applications such as Global Position Systems or traffic control, office monitoring or white goods remote control.

A sensor node (or mote) is a small battery supplied device endowed with a sensing system able to collect data (temperature, moistness rate, position variations), a processing system which elaborates information and a communication system which sends and shares data with other motes.

One of the most important aspects regarding motes is power saving: typically, sensors are battery powered devices and batteries are not supposed to be changed frequently; thus, it is very important to increase sensors lifetime with smart techniques, like powering off the communication system when they do not transmit data or when they are not supposed to receive any.

Another interesting thing regarding motes concerns data transfer: since they have not much computational resources, data are sent via one or more hops to a Network Coordinator, which in turn can send them to a host able to cope with data processing and results storage. Communication between nodes must be reliable, in order to guarantee that frames are received uncorrupted and avoid retransmissions, saving battery power as well. Besides, it could be necessary or desirable to make communications safe and secure, that is guarantee messages provenance (authenticity), protect data from unauthorized accesses (confidentiality) and prevent unwanted replies of received messages (anti-replay). By doing so, it is more difficult for an adversary to alter or have access to data communications.

One technology suitable for sensors networks is described by the IEEE 802.15.4 Standard: it describes wireless Medium Access Control (MAC) and Physical (PHY) layers specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs). Two different device types can participate in an IEEE 802.15.4 network: Full-Function Devices (FFD) and Reduced-Function Devices (RFD). A network comprises at least one coordinator, that is a FFD capable of relaying messages from other devices. Plus, one coordinator is elected as the PAN Coordinator. An RFD is intended for extremely simple applications, such as light switching or passive infrared sensoring. Consequently, it can be implemented using minimal resources and memory capacity. In addition, an RFD is associated to a single PAN coordinator at a time.

Depending on the application requirements, a network may be organized in either of two topologies: the star topology in which all motes communicate directly and only with the network coordinator, or the peer-to-peer topology where all motes can communicate with each other within their radio range. Trasmissions are organized into frames, which have been designed trying to keep complexity at a minimum, while still assuring robustness for transmissions on a noisy channel.

The medium access exploits the Carrier Sense Multiple Access protocol, with Collision Avoidance. It can be used in an un-slotted version or in a time-slotted version with beacon frames to keep motes synchronized. In time-slotted version, between every two beacons each device competes with others by means of a slotted CSMA-CA mechanism during the Contention Access Period (CAP), while guaranteed time slots can be assigned during the Contention Free Period (CFP).

Moreover, IEEE 802.15.4 offers many security options: encryption only mode (CTR), authentication only mode (CBC-MAC) and a combination of them (CCM). Cryptography is based on AES (Advanced Encryption Standard) 128 bits symmetric-key cryptography and keys are provided by the higher layers. Security is handled by means of specific structures (containing security parameters) and the Auxiliary Security Header used to validate and decipher messages on reception.

In this thesis work, TelosB motes are used. TelosB is an open source platform designed for experimentation within the research community. Besides, it includes a CC2420 chipset which provides extensive hardware support for packet handling and security mechanisms. In particular, some security routines have been implemented, in order to create and use the Auxiliary Security Header within IEEE 802.15.4 frames, as provide security services in CTR mode, with reference to the nesC developing language and the TinyOS environment. Nevertheless, some simplified assumptions were made: cryptographic keys and security mode are set in a static way. Finally a simple application has been realized, in order to test the correctness of communications between two motes, with or without security.

The rest of the thesis is organized as follows: next chapter (chapter 2) provides a brief description about IEEE 802.15.4 networks, while chapter 3 describes the main features of TelosB motes and CC2420 chipset. Chapter 4 includes a description of the referential network scenario and discusses the implementation of the Auxiliary Security Header and the security routines as well. Finally, chapter 5 reports the conclusions.

Chapter 2

Overview on IEEE 802.15.4

2.1 Basic Concepts

In an IEEE 802.15.4 network, two kinds of device are admitted to participate: full-function devices (FFD) and reduce-function devices (RFD). FFD devices can operate as Personal Area Network coordinator, as coordinator or as device and they can also talk to RFDs or other FFDs. RFD devices can communicate only with a single FFD at time and are intended for simple applications (e.g. light switches). As already told in the previous chapter, two types of network can be created according to the IEEE 802.15.4 standard, that is star topology and peer to peer topology(fig 2.1).

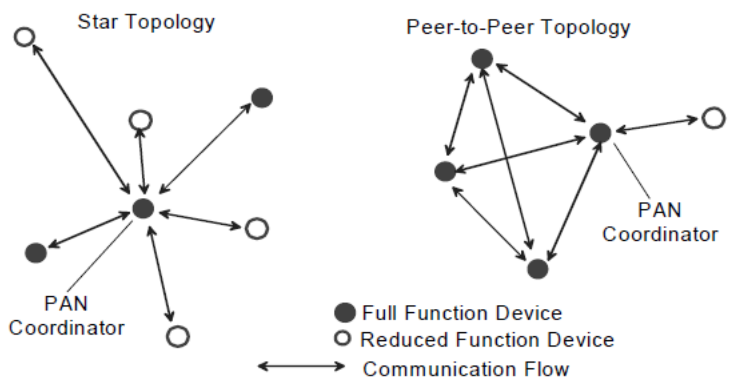


Figure 2.1: Topology in IEEE 802.15.4

In star topologies the communication is established between devices and a single central controller, called the PAN coordinator. A device has to associate with the PAN coordinator to be part of the network. Also in peer-to-peer topologies there is a PAN coordinator, but any device is allowed to communicate with any other device as long as they are in range of one another. Peer-to-peer topology allows more complex network formations to be implemented, such as mesh networking topology. Applications such as industrial control and monitoring, wireless sensor networks, asset and inventory tracking, intelligent agriculture, and security would benefit from such a network topology. The physical medium is accessed through a CSMA/CA protocol. Networks which are not using beaconing mechanisms utilize an unslotted variation which is based on the listening of the medium, leveraged by a random exponential backoff algorithm; acknowledgments do not adhere to this discipline. Common data transmission utilizes unallocated slots when beaconing is in use; again, confirmations do not follow the same process. Confirmation messages may be optional under certain circumstances, in which case a success assumption is made. Whatever the case, if a device is unable to process a frame at a given time, it simply does not confirm its reception: timeout-based retransmission can be performed a number of times, following after that a decision of whether to abort or keep trying. Because the predicted environment of these devices demands maximization of battery life, the protocols tend to favor the methods which lead to it, implementing periodic checks for pending messages, the frequency of which depends on application needs. IEEE 802.15.4 specify both PHY and MAC layer. The PHY layer activates and deactivates the radio transceiver, monitors energy detection and link quality indicator for received packets, controls the clear channel assessment (CCA) for carrier sense multiple access with collision avoidance (CSMA-CA), selects Channel Frequency and data transmission and reception [1]. The MAC layer allows the transmission of the MAC frames through the use of the physical channel. Besides the data service, it offers a management interface and manages access to the physical channel and network beaconing. It also controls frame validation, guarantees time slots and handles node associations. Finally, it offers hook points for secure services; this topic will be deepen later.

2.2 Frame Structure

The standard defines four kinds of frame: the four frame structures were designed to keep the complexity to a minimum as well as making them sufficiently robust for transmission on a noisy channel. Each further protocol layer is added to the structure with layer-specific headers and footers. Beacon frames are transmitted by a coordinator, data frames and acknowledgment frames are used for data transfers and to confirm successful frame reception respectively. Finally, MAC command frames are used to handle all MAC peer entity control transfers.

2.2.1 Beacon Frame

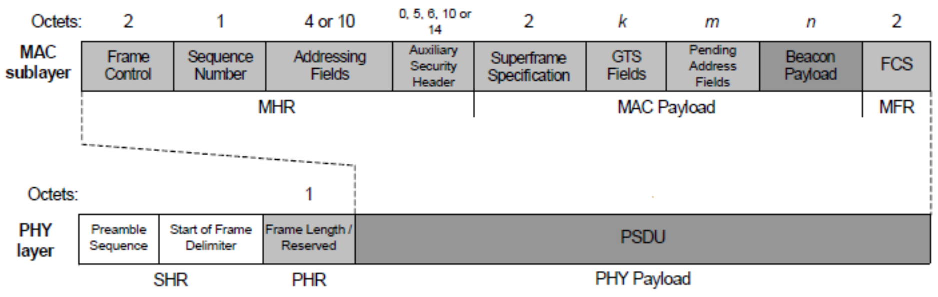


Figure 2.2: Beacon Frame and PHY Packet

The beacon frame (fig. 2.2) contains the MAC header (MHR), the MAC Payload and the MAC footer (MFR). These fields compose the PHY payload and the PHY packet is completed by the Synchronization Header (SHR) and the PHY header (PHR). The MAC header is composed by 2 octets Frame Counter Field, a single octet Sequence Number Field, a variable length Addressing Fields and, optionally, the Auxiliary Security Header. The MFR contains a 16-bit frame check sequence. This frame is used in beacon-enabled network to guarantee synchronization between RFDs and FFDs.

2.2.2 Data Frame

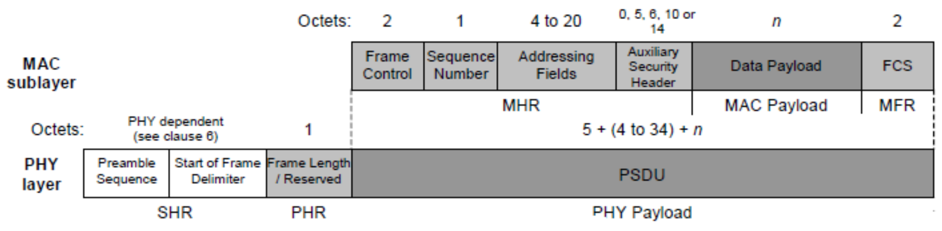


Figure 2.3: Data Frame and PHY Packet

The data frame content is originated by the upper layers. The MAC payload is prefixed with the MHR and appended with the MFR. The MHR contains the Frame Control Field, the Data

Sequence Number (DSN), the Addressing Fields, and, optionally, the Auxiliary Security Header. The MFR is composed of a 16-bit FCS. The MHR, the MAC payload, and the MFR together form the MAC data frame (fig 2.3). These fields compose the PHY payload and the PHY packet is completed by the Synchronization Header (SHR) and the PHY header (PHR). This frame is used to exchange data between RFDs and FFDs in star topology networks or between RFDs in peer-to-peer networks.

2.2.3 Acknowledgment Frame

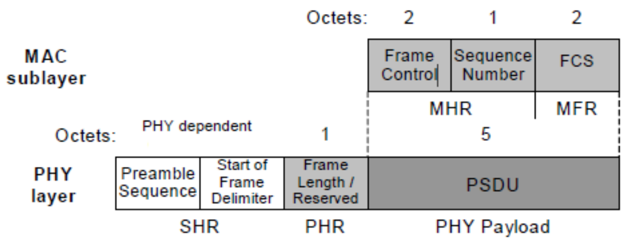


Figure 2.4: Acknowledgment Frame and PHY Packet

The MAC acknowledgment frame (fig. 2.4) is composed by the MHR and the MFR, and it has no MAC payload. The MHR contains the MAC Frame Control Field and the Sequence Number (DSN). The MFR is composed of a 16-bit FCS. The MHR and MFR together form the MAC acknowledgment frame, that is the payload of the PHY packet. The PHY payload is prefixed with the SHR, containing the Preamble Sequence and SFD fields, and the PHR containing the length of the PHY payload in octets. The SHR, the PHR, and the PHY payload together form the PHY packet. As we can see in figure, acknowledgment frames do not support the security mechanism; so they never have the Auxiliary Security Header. This frame, if requested, is sent to the data sender to notify that the frame it sent, was correctly received. This mechanism is asymmetric: if a single transmission attempt has failed and the transmission was indirect, the coordinator shall not retransmit the data or MAC command frame. Instead, the frame shall remain in the transaction queue of the coordinator and can only be extracted following the reception of a new data request command. If a new data request command is received, the originating device shall transmit the frame using the same DSN as was used in the original transmission. Otherwise if a single transmission attempt has failed and the transmission was direct, the device shall repeat the process of transmitting the data or MAC command frame and waiting for the acknowledgment, up to a maximum of macMaxFrameRetries times.

2.2.4 MAC Command Frame

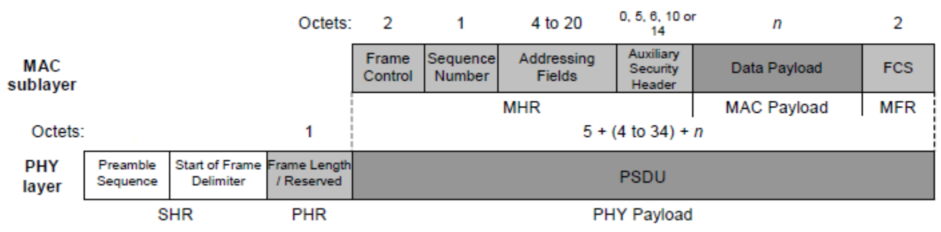
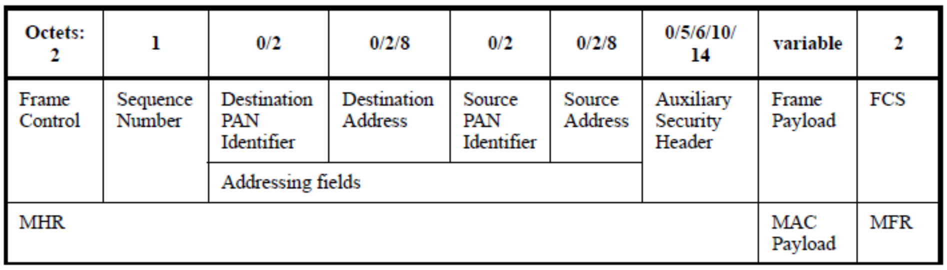


Figure 2.5: Command Frame and PHY Packet

MAC command frames(fig. 2.5) are entirely originated within the MAC sublayer. The MAC payload contains the Command Type field and the command payload, which is prefixed with the MHR and appended with the MFR. The MHR contains the MAC Frame Control field, the DSN, the Addressing Fields, and, optionally, the Auxiliary Security Header. The MFR contains a 16-bit FCS. The MHR, the MAC payload, and the MFR together form the MAC command frame and compose the PHY payload. The PHY payload is prefixed with an SHR, containing the Preamble Sequence and SFD fields, and a PHR containing the length of the PHY payload in octets. The preamble sequence allows the receiver to achieve symbol synchronization.

2.3 MAC Header and Auxiliary Security Header

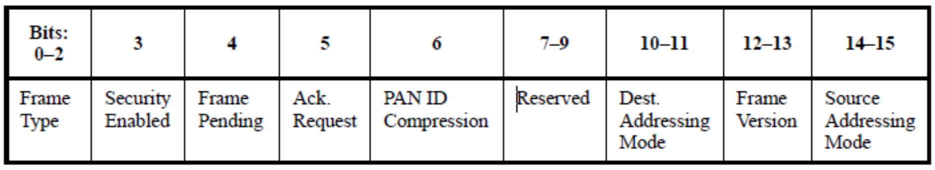


Octets: 2	**1**	**0/2**	**0/2/8**	**0/2**	**0/2/8**	**0/5/6/10/ 14**	**variable**	**2**
Frame Control	Sequence Number	Destination PAN Identifier	Destination Address	Source PAN Identifier	Source Address	Auxiliary Security Header	Frame Payload	FCS
		Addressing fields						
MHR							MAC Payload	MFR

Figure 2.6: MAC Frame

The MAC frame is composed of the MAC header, the MAC payload, and the MAC Footer. However, some fields like Addressing Fieds or Security Header might not be included in all frames, as shown in figure 2.6

2.3.1 Frame Control Field



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

Figure 2.7: Frame Control Field

The Frame Control Field is 2 octets field, containing information defining the frame type, addressing fields type, and other control flags. It is formatted as illustrated in figure 2.7.

2.3.1.1 Frame Type Subfield

The Frame Type subfield specifies the current frame type (command, beacon, acknowledgment or data).

2.3.1.2 Security Enabled Subfield

The Security Enabled subfield is set to one if the frame is protected by the MAC security sublayer and must be set to zero otherwise. This is an important bit, since the Auxiliary Security Header field of the MHR is present only if this subfield is set to one.

2.3.1.3 Frame Pending Subfield

The Frame Pending subfield is set to one if the device sending the frame has more data for the recipient.

2.3.1.4 Acknowledgment Subfield

The Acknowledgment Request subfield is 1 bit in length and specifies whether or not an acknowledgment is required from the recipient device, on receipt of a data or MAC command frame. If this subfield is set to one, the recipient device sends an acknowledgment frame only if, upon reception, the frame passes the third level of filtering. If this subfield is set to zero, the recipient device will never send acknowledgment frames.

2.3.1.5 PAN ID Compression subfield

The PAN ID Compression subfield specifies whether the MAC frame to be sent contains only one of the PAN identifier fields when both source and destination addresses are present. If this subfield is set to one and both the source and destination addresses are present, the frame has to contain only the Destination PAN Identifier field, and the Source PAN Identifier field is be assumed equal to the destination's.

2.3.1.6 Frame Version Subfield

The Frame Version subfield specifies the version number corresponding to the frame.

2.3.1.7 Addressing Mode Subfields

Finally, the Destination Addressing Mode and the Source Addressing Mode subfields indicate if Address fields contain 16-bit short addresses or 64-bit extended addresses.

2.3.2 Other MAC Header fields

2.3.2.1 Sequence Number

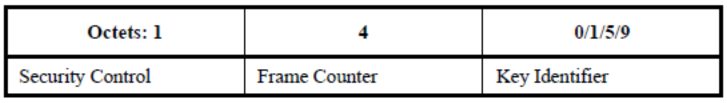
The Sequence Number field specifies the sequence identifier for the frame: for a Beacon frame, the Sequence Number field has to specify a Beacon Sequence Number. For Data, Acknowledgment, or MAC Command frame, the Sequence Number field has to specify a Data Sequence Number, used to match an acknowledgment frame to the data or MAC command frame.

2.3.2.2 Addressing fields

The addressing fields, if present, specify source and destination address and source and destination PAN ID, according with the PAN ID compression bit and destination and source addressing mode in the Frame Control Field.

2.3.3 Auxiliary Security Header

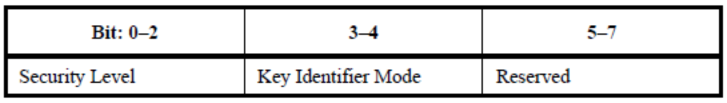
The Auxiliary Security Header has a variable length and contains information required for security processing, including the Security Control field, the Frame Counter field, and the Key Identifier field. The Auxiliary Security Header is present only if the Security Enabled subfield of the Frame Control field is set to one and is formatted as illustrated in figure 2.8.



Octets: 1	4	0/1/5/9
Security Control	Frame Counter	Key Identifier

Figure 2.8: Auxiliary Security Header

2.3.3.1 Security Control Field



Bit: 0–2	3–4	5–7
Security Level	Key Identifier Mode	Reserved

Figure 2.9: Security Control Subfield

The 8-bit Security Control field is used to provide information about what kind of protection is applied to the frame. The Security Control field has to be formatted as shown in figure 2.9. The Security Level subfield is 3 bit in length and indicates the actual frame protection that is provided. This value can be adapted on a frame-by-frame basis and allows for varying levels of data authenticity (to allow minimization of security overhead in transmitted frames where required) and for optional data confidentiality. Table 2.1 summarizes all security levels available.

Security Level Identifier	**Security Control Field**	**Security Attributes**	**Data confidentially**	**Data authenticity**
0x00	’000’	None	OFF	NO (M = 0)
0x01	’001’	MIC-32	OFF	YES (M = 4)
0x02	’010’	MIC-64	OFF	YES (M = 8)
0x03	’011’	MIC-128	OFF	YES (M = 16)
0x04	’100’	ENC	ON	NO (M =0)
0x05	’011’	ENC-MIC-32	ON	YES (M = 4)
0x06	’110’	ENC-MIC-64	ON	YES (M = 8)
0x07	’111’	ENC-MIC-128	ON	YES (M = 16)

Table 2.1: Security Level values and options

The Key Identifier Mode subfield is 2 bit in length and indicates whether the key used to protect the frame can be derived implicitly or explicitly. Furthermore, it is used to indicate the particular representations of the Key Identifier field, in case the key is derived explicitly. The Key Identifier Mode subfield is set according to Table 2.2. The Key Identifier field of the Auxiliary Security Header is present only if this subfield has a value not equal to 0x00.

Key Identifier mode	**Key Identifier Mode subfield**	**Description**	**Key Identifier field length** (octets)
0x00	’00’	Key is determined implicitly from the originator and receipient(s) of the frame as indicated in the frame header.	0
0x01	’01’	Key is determined from the 1-octet Key Index subfield of the Key Identifier field of the auxiliary security header in 6 conjunction with macDefaultKeySource.	1
0x02	’10’	Key is determined explicitly from the 4-octet Key Source subfield and the 1-octet Key Index subfield of the Key Identifier field of the auxiliary security header.	5
0x03	’11’	Key is determined explicitly from the 8-octet Key Source subfield and the 1-octet Key Index subfield of the Key Identifier field of the auxiliary security header.	9

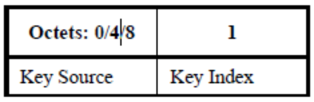
Table 2.2: Key Identifier values and options

2.3.3.2 Frame Counter Field

The Frame Counter field is 4 octets length field and represents the macFrameCounter attribute of the originator of a protected frame. It is used to provide semantic security of the cryptographic mechanism used to protect a frame and to assure replay protection.

2.3.3.3 Key Identifier Field

The Key Identifier field has variable length and identifies used for cryptographic protection of outgoing frames, either explicitly or in conjunction with implicitly defined side information. The Key Identifier field is present only if the Key Identifier Mode subfield of the Security Control field of the Auxiliary Security Header is set to a value different from 0x00. The Key Identifier field is formatted as illustrated in figure 2.10. The Key Source subfield, when present, is either 4 octets or 8 octets in length, according to the value specified by the Key Identifier Mode subfield of the Security Control field, and indicates the originator of a group key. The Key Index subfield is 1 octet in length and allows unique identification of different keys with the same originator.



Octets: 0/4/8	1
Key Source	Key Index

Figure 2.10: Key Identifier Subfield

2.4 Security Structures

The MAC sublayer is responsible for providing security services on specified incoming and outgoing frames, when requested by the higher layers. The information according to which is determined how to provide security is located in the security-related PIB (PAN Information Base) [2]. This security-related PIB is divided in 7 structures:

- Key Table.
- Device Table.
- Minimum security level table.
- Frame counter.
- Automatic request attributes.
- Default key source.
- PAN coordinator address.

Key table contains key-descriptors, that they are keys with related key-specific information that are required for security processing. The device table holds device-descriptors, containing device-specific addressing and security-related information which, combined with key-specific information from the key table, provide all the keying material needed to secure/unsecure frames. The minimum security level table holds information regarding the minimum security level the device expects to have been applied by the originator of a frame, depending on frame type and, if it concerns a MAC command frame, the command frame identifier. The 4-octets frame counter is used to provide replay protection and semantic security of the cryptographic building block used for securing outgoing frames. The Automatic Request table holds all the information needed to secure outgoing frames generated automatically and not as a result of a higher layer primitive, as is the case with automatic data requests. The default key source is commonly shared between originator and recipient (s) of a secured frame, so that, when combined with additional information explicitly contained in the requesting primitive or in the received frame, it allows an originator or a recipient to determine the key required for securing or unsecuring the frame, respectively. The address of the PAN coordinator is an information commonly shared between all devices in a PAN. All the security-related PIB attribute and their options are summarize in Table 2.3. Finally their implementation in the code can be seen in Appendix B.

2.5 Counter with CBC-MAC extension (CCM)

CCM is a generic authenticate-and-encrypt block cipher mode. CCM is only defined for use with 128-bit block ciphers, such as AES: the Advanced Encryption Standard. The AES ciphers have been analyzed extensively and are now used worldwide, as was the case with its predecessor, the Data Encryption Standard (DES). AES is based on a design principle known as a Substitution permutation network. It is fast in both software and hardware, Unlike its predecessor, DES, AES does not use a Feistel network [3]. The cipher operates on a 4×4 array of bytes, termed the state (versions of Rijndael with a larger block size have additional columns in the state). Most AES calculations are done in a special finite field. The AES cipher is specified as a number of repetitions of transformation rounds that convert the input plaintext into the final output of ciphertext. Each round consists of several processing steps, including one that depends on the encryption key. A set of reverse rounds are applied to transform ciphertext back into the original plaintext using the same encryption key. For Example, an eXtended Sparse Linearization attack, (XLS) relies on first analyzing the internals of a cipher and deriving a system of quadratic simultaneous equations. These systems of equations are typically very large, for example 8000 equations with 1600 variables for the 128-bit AES: it requiring 2^{100} operations (compared to 2^{128} possible keys) would be considered a break. For the generic CCM mode there are two parameter choices. The first choice is M, the size of the authentication field. The choice of the value for M involves a trade-off between message expansion and the probability that an attacker can undetectably modify a message. The second choice is L, the size of the length field. This value requires a trade-off between the maximum message size and the size of the Nonce. Different applications require different trade-offs, so L is a parameter. Regarding the strength of CCM. It is tested that shows that CCM provides a level of confidentiality and authenticity that is in line with other

Attribute	Identifier	Type	Range	Description	Default
macKeyTable	0x71	List of Key Descriptor entries.	—	A table of KeyDescriptor entries, each contains keys and related information required for secured communications.	(empty)
macKeyTable-Entries	0x72	Integer	Implementa-tion specific	The number of entries in macKeyTable.	0
macDeviceTable	0x73	List of Device-Descriptor entries	–	A table of Device-Descriptor entries, each indicating a remote device with which this device securely communicates.	(empty)
macDeviceTable-entries	0x74	Integer	Implementa-tion specific	The number of entries in macDeviceTable.	0
macSecurity-LevelTable	0x75	Table of SecurityLevel Descriptor entries	—	A table of SecurityLevel-Descriptor entries, each with information about minimum security level expected depending on incoming frame type and subtype.	(empty)
macSecurity-LevelTableEntries	0x76	Integer	Implementa-tion specific	The number of entries in macSecurityLevelTable	0
macFrameCounter	0x77	Integer	0x00000000-0xffffffff	The outgoing frame counter for this device.	0x00
macAutoRequest-SecurityLevel	0x78	Integer	0x00-0x07	The security level used for automatic data request.	0x06
macAutoRequest-KeyIdMode	0x79	Integer	0x00-0x03	The key identifier mode used for automatic data request. This Attribute is invaild if the macAuto-RequestSecurityLevel attribute is set to 0x00.	0x00
macAutoRequest-KeySource	0x7a	As specified by the mac-AutoRequest-KeyIDMode.	—	The originator of the key used for automatic data request.	All octets 0xff
macAutoRequest-KeyIndex	0x7b	Integer	0x01-0xff	The index of the key used for automatic data request.	All octets 0xff
macDefaultKey-Source	0x7c	Set of 8 octets	—	The originator of the default key used for key identifier mode 0x01.	All octets 0xff
macPANCoord-ExtendedAddress	0x7d	IEEE address	An extended 64-bit IEEE address	64-bit address of the PAN coordinator.	—
MacPANCoord-ShortAddress	0x7e	Integer	0x0000-0xffff	16-bit address of the PAN coordinator. 0xfffe means that PAN coordinator uses only 64-bit extended address.	0x0000

Table 2.3: MAC security-related PIB

proposed authenticated encryption modes, such as OCB mode. The only weakness of CCM is the trade-off between nonce size and counter size. For a general mode we want to support large messages. Some applications use only small messages, but would rather have a larger nonce. Introducing the L parameter solves this issue. The parameter M gives the traditional trade-off between message expansion and probability of forgery. For most applications, it is choosing M at least 8. Performance depends on the speed of the block cipher implementation.

In hardware, for large packets, the speed achievable for CCM is roughly the same as that achievable with the CBC encryption mode. Encrypting and authenticating an empty message, without any additional authentication data, requires two block cipher encryption operations. For each block of additional authentication data one additional block cipher encryption operation is required (if one includes the length encoding). Each message block requires two block cipher encryption operations. The worst-case situation is when both the message and the additional authentication data are a single octet. In this case, CCM requires five block cipher encryption operations. CCM results in the minimal possible message expansion; the only bit added are the authentication bit. Both the CCM encryption and CCM decryption operations require only the block cipher encryption function. In AES, the encryption and decryption algorithms have some significant differences. Thus, using only the encrypt operation can lead to a significant savings in code size or hardware size In hardware, CCM can compute the message authentication code and perform encryption in a single pass. That is, the implementation does not have to complete calculation of the message authentication code before encryption can begin. CCM was designed for use in the packet processing environment. The authentication processing requires the message length to be known at the beginning of the operation, which makes one-pass processing difficult in some environments. However, in almost all environments, message or packet lengths are known in advance [4].

Chapter 3

TelosB and CC2420

3.1 TelosB mote

A sensor node, also known as a 'mote' , is a node in a wireless sensor network that is capable of performing some processing, gathering sensory information and communicating with other connected nodes in the network. As we saw in figure 3.1, one point of strength about motes is their small dimension.

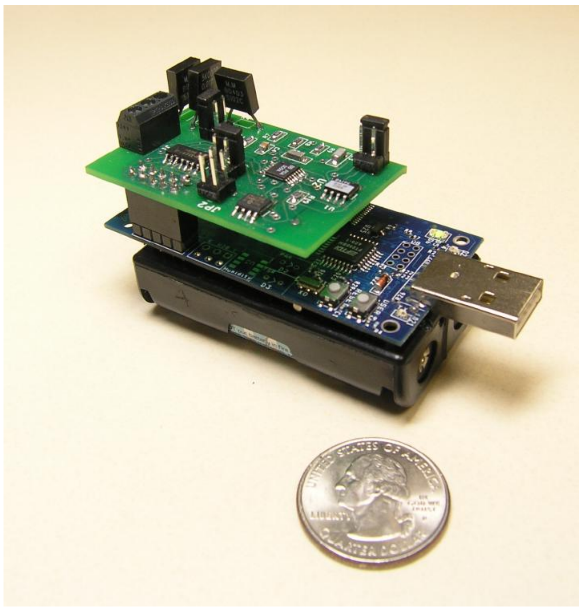


Figure 3.1: A simple mote

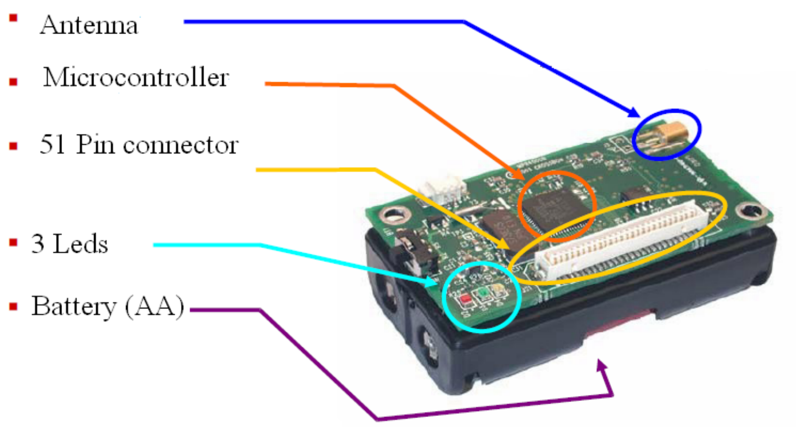


Figure 3.2: Components of a Mote

In the figure 3.2, we can see the components of the mote: an antenna, that permits to send and receive data; a microcontroller that performs tasks, processes data and controls the functionality of other components in the sensor node; 3 leds that can be use to signal certain situation or for debugging and a AA Battery slot. A generic sensor has 5 subsystems(fig 3.3), each of one with specific tasks:

- Sensing subsystem
- Processing subsystem
- Communication subsystem
- Actuation subsystem
- Power management subsystem.

The sensing subsystem is designed to gather information about the environment. It will process and store information that other subsystem use. The data collection system would capture data such as reflected light and sound. The processing subsystem is designated to take the information from e.g. the sensing subsystem and elaborate them in the way they can be used by other subsystems. The communication subsystem is designated for sending and receiving tasks. The power management subsystem concerns about all the operations about battery managing (e.g. power saving) and finally the actuation subsystem gathers information from sensing and processing and decide how to control and evolve the system. The mote we work with is TelosB: it is an open source platform designed to enable cutting-edge experimentation for the research community.

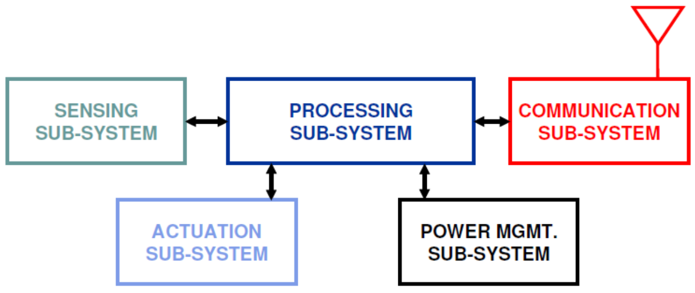


Figure 3.3: Generic Architecture

The TPR2400 bundles all the essentials for lab studies into a single platform including: USB programming capability, an IEEE 802.15.4 radio with integrated antenna, a low-power MCU with extended memory and an optional sensor suite (TPR2420).

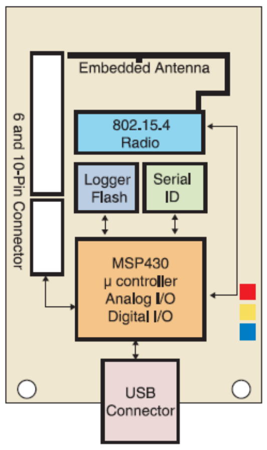


Figure 3.4: TPR2400CA Block Diagram

This suite offers many features, including:

- IEEE 802.15.4/ZigBee compliant RF transceiver
- Integrated onboard antenna
- Low current consumption
- 1 MB external flash for data logging
- Programming and data collection via USB

- Runs TinyOS 1.1.10 or higher

This platform delivers low power consumption allowing for long battery life as well as fast wakeup from sleep state. Though the TPR2400 is an uncertified radio platform, it is fully compatible with the open-source TinyOS distribution. TPR2400 is powered by two AA batteries. If the TPR2400 is plugged into the USB port for programming or communication, power is provided from the host computer. If the TPR2400 is always attached to the USB port no battery pack is needed. TPR2400 provides users with the capability to interface with additional devices. The two expansion connectors and onboard jumpers may be configured to control analog sensors, digital peripherals and LCD displays [5]. The platform with we work is TinyOS versione 2.0.2: it is a small, open-source, energy-efficient software operating system developed by UC Berkeley which supports large scale, selfconfiguring sensor networks. Other information can be found at TinyOS main site www.tinyos.org.

3.2 CC2420 Chipset

CC240 is the chipset that TelosB motes used. It can be used in several applications: Zigbee and TinyOS systems, home and building automation, industrial control and wireless sensor networks. The CC2420 is a true single-chip 2.4 GHz IEEE 802.15.4 compliant RF transceiver designed for low-power and low-voltage wireless applications. It provides extensive hardware support for packet handling, data buffering, burst transmissions, data encryption, data authentication, clear channel assessment, link quality indication and packet timing information. Between its many features, we can highlight the separate transmit and receive FIFOs, the IEEE 802.15.4 MAC hardware support (CRC-16 computation, Energy Detection, Link Quality detection, etc.) and IEEE 802.15.4 MAC hardware security (CTR encryption/decryption, CBC-MAC authentication CCM encryption /decryption and authentication, stand-alone AES encryption).

3.2.1 Configuration and Data Interface

There are 33 16-bit configuration and status registers, 15 command strobe registers, and two 8-bit registers to access to the separate transmit and receive FIFOs. Each of the 50 registers is addressed by a 6-bit address. The configuration registers can be read by the microcontroller via the same configuration interface. The R/W bit must be set high to initiate the data read-back. CC2420 then returns the data from the addressed register on the 16 clock cycles following the register address. During transfer of the register access byte or command strobes, the first RAM address byte and data transfer to the TXFIFO, the CC2420 status byte is returned on the SO pin. The status byte contains 6 status bit which are described in Table 3.1. Issuing a SNOP (no operation) command strobe may be used to read the status byte.

Command strobes may be viewed as single byte instruction to CC2420.

Bit#	**Name**	**Description**
7	—	Reserved, ignore value
6	XOSC16M_STABLE	Indicates whether the 16 MHz oscillator is running or not 0 : The 16 MHz crystal oscillator is not running 1 : The 16 MHz crystal oscillator is running
5	TX_UNDERFLOW	Indicates whether a FIFO underflow has occurred during transmission. It must be cleared manually with a SFLUSHTX command strobe. 0 : No underflow has occurred 1 : An underflow has occurred
4	ENC_BUSY	Indicates whether the encryption module is busy 0 : Encryption module is idle 1 : Encryption module is busy
3	TX_ACTIVE	Indicates whether RF transmission is active 0 : RF Transmission is idle 1 : RF Transmission is active
2	LOCK	Indicates whether the frequency synthesizer PLL is in lock or not 0 : The PLL is out of lock 1 : The PLL is in lock
1	RSSI_VALID	Indicates whether the RSSI value is valid or not. 0 : The RSSI value is not valid 1 : The RSSI value is valid, always true when reception has been enabled at least 8 symbol periods (128 us)
0	—	Reserved, ignore value

Table 3.1: Status Byte

By addressing a command strobe register internal sequences will be started.
These commands must be used to enable receive mode, start decryption etc. All command strobe can be viewed in Table 3.2.

Address	**Register**	**Description**
0x00	SNOP	No Operation (has no other effect than reading out status-bits)
0x01	SXOSCON	Turn on the crystal oscillator (set XOSC16M_PD = 0 and BIAS_PD = 0)
0x02	STXCAL	Enable and calibrate frequency synthesizer for TX; Go from RX / TX to a wait state where only the synthesizer is running.
0x03	SRXON	Enable RX
0x04	STXON	Enable TX after calibration (if not already performed) Start TX in-line encryption if SPI_SEC_MODE $\neq$ 0
0x05	STXONCCA	If CCA indicates a clear channel: Enable calibration, then TX. Start in-line encryption if SPI_SEC_MODE $\neq$ 0 else do nothing
0x06	SRFOFF	Disable RX/TX and frequency synthesizer
0x07	SXOSCOFF	Turn off the crystal oscillator and RF
0x08	SFLUSHRX	Flush the RX FIFO buffer and reset the demodulator. Always read at least one byte from the RXFIFO before issuing the SFLUSHRX command strobe
0x09	SFLUSHRX	Flush the TX FIFO buffer
0x0A	SACK	Send acknowledge frame, with pending field cleared.
0x0B	SACKPEND	Send acknowledge frame, with pending field set.
0x0C	SRXDEC	Start RXFIFO in-line decryption / authentication (as set by SPI_SEC_MODE)
0x0D	STXENC	Start TXFIFO in-line encryption / authentication (as set by SPI_SEC_MODE), without starting TX.
0x0E	SAES	AES Stand alone encryption strobe. SPI_SEC_MODE is not required to be 0, but the encryption module must be idle. If not, the strobe is ignored.

Table 3.2: Strobe configuration registers overview

3.2.2 RAM access

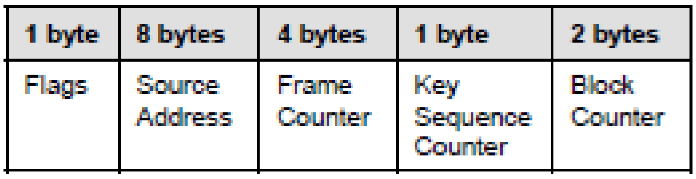
CC240 also has 368 bytes RAM that can be accessed through the SPI interface. It contains 1-1 mapping of the FIFO registers, the KEY0 and the KEY1 registers, the RXNONCE and the TXNONCE registers. In Table 3.3 we can see a little summarization of RAM. The TXFIFO is write only, but it may be read back using RAM access. Data is read and written one byte at a time, as with RAM access. The RXFIFO is both writeable and readable. The KEY0 and KEY1 registers contain a 16-bit key used for ciphering/deciphering operation. After a key is written in any of these registers, it is selected and used reading the SEC_TXKEYSEL/SEC_RXKEYSEL bit in SECCTRL0 register. TXNONCE and RXNONCE contain nonce for authentication.

Addressing	**Name**	**Description**
0x16F – 0x16C	–	Not used
0x16B – 0x16A	SHORTADR	16-bit Short address, used for address recognition.
0x169 0X168	PANID	16-bit PAN identifier, used for address recognition.
0x167 – 0x160	IEEEADR	64-bit IEEE address of current node, used for address recognition .
0x15F – 0x150	CBSTATE	Temporary storage for CBC-MAC calculations.
0x14F – 0x140	TXNONCE / TXCTR	Transmitter nonce for in-line authentication and transmitter counter for in-line encryption.
0x13F – 0x130	KEY1	Encryption Key 1.
0x12F – 0x120	SABUF	Stand-alone encryption buffer, for plaintext input and ciphertext output.
0x11F – 0x110	RXNONCE / RXCTR	Receiver nonce for in-line authentication or receiver counter for in-line decryption.
0x10F – 0x100	KEY 0	Encrypted Key 0.
0x0FF – 0x80	RXFIFO	128 bytes receive FIFO.
0x07F – 0x00	TXFIFO	128 bytes transmit FIFO.

Table 3.3: RAM Memory Space

3.2.3 Security Operation

CC2420 features hardware IEEE 802.15.4 MAC security operations. This includes counter mode (CTR) encryption/decryption, CBC-MAC authentication and CCM encryption and authentication. All security operations are based on AES encryption using 128 bit keys. Security operations are performed within the transmit and receive FIFOs on a frame basis. The SAES, STXENC and SRXDEC command strobes are used to start security operations in CC2420 as will be described in the following sections. The ENC_BUSY status bit may be used to monitor when a security operation has been completed. Security command strobes issued while the security engine is busy, will be ignored and the ongoing operation will be completed. The CC2420 RAM space has storage space for two individual keys (KEY0 and KEY1). Transmit, receive and stand-alone encryption may select one of these two keys individually in the SEC_TXKEYSEL, SEC_RXKEYSEL and SEC_SAKEYSEL control bits (SECCTRL0). A way of establishing the keys used for encryption and authentication must be decided for each particular application. IEEE 802.15.4 does not define how this is done, it is left to the higher layers of the protocol. The nonce must be correctly initialized before receiving or transmitting operation. The format of the nonce is shown in figure 3.5. The block counter must be set to 1. The key sequence counter is controlled by a layer above the MAC layer. The frame counter must be increased for each new frame by the MAC layer. The source address is the 64 bit IEEE address.



1 byte	8 bytes	4 bytes	1 byte	2 bytes
Flags	Source Address	Frame Counter	Key Sequence Counter	Block Counter

Figure 3.5: IEEE 802.15.4 Nonce

The other registers used on security operation are SECCTRL0(Table 3.4) and SECCNTRL1(Table 3.5): SECCTRL0 contains all security options(which key is used, what type of security it is applied to frame, other options); SECCNTRL1 contains the offset where encryption/decryption/authentication starts.

The key, nonce (does not apply to CBC-MAC), SECCTRL0 and SECCTRL1 control registers must be correctly set before starting any in-line security operation.

The in-line security mode is set in SECCTRL0.SEC_MODE to one of the following modes:

Bit	**Field Name**	**Reset**	**Description**
15:10	—	0	Reserved, write as 0
9	RXFIFO_PROTECTION	1	Protection enable of the RXFIFO (overflow)
8	SEC_CBC_HEAD	1	Defines what to use for the first byte in CBC-MAC (does not apply to CBC-MAC part of CCM): 0 : Use the first data byte as the first byte into CBC-MAC 1: : Use the length of the data to be authenticated for tx or using SEC_RXL
7	SEC_SAKEYSEL	1	Stand Alone Key select 0 : Key 0 is used - 1 : Key 1 is used
6	SEC_TXKEYSEL	1	TX Key select 0 : Key 0 is used - 1 : Key 1 is used
5	SEC_RXKEYSEL	1	RX Key select 0 : Key 0 is used - 1 : Key 1 is used
4:2	SEC_M[2:0]	1	Number of bytes in authentication field for CBC-MAC, bytes encoded as (M-2)/2. 0 : Reserved – 1: 4 – 2: 6 – 3: 8 4:10 – 5: 12 – 6:14 – 7: 16
1:0	SEC_MODE[1:0]	0	Security mode 0 : In-line security is disabled 1 : CBC-MAC 2 : CTR 3 : CCM

Table 3.4: Security Control0 Register

Bit	Field Name	Reset	Description
15	—	0	Reserved, write as 0
14:8	SEC_TXL	0	Multi-purpose length byte for TX in-line security operations: CTR : Number of cleartext bytes between length byte and the first byte to be encrypted CBC-MAC : Number of cleartext bytes between length byte and the first byte to be authenticated CCM : l(a), defining the number of bytes to be authenticated but not encrypted Stand-alone : SEC_TXL has no effect
7	—	0	Reserved, write as 0
6:0	SEC_RXL	0	Multi-purpose length byte for TX in-line security operations: CTR : Number of cleartext bytes between length byte and the first byte to be deencrypted CBC-MAC : Number of cleartext bytes between length byte and the first byte to be authenticated CCM : l(a), defining the number of bytes to be authenticated but not decrypted Stand-alone : SEC_RXL has no effect

Table 3.5: Security Control1 Register

- Disabled
- CBC-MAC (authentication)
- CTR (encryption / decryption)
- CCM (authentication and encryption/decryption)

When enabled, TX in-line security is started in one of two ways: issuing the STXENC command strobe, so in-line security will be performed within the TXFIFO, but a RF transmission will not be started. Ciphertext may be read back using RAM read operations; or issuing the STXON or STXONCCA command strobe, so in-line security will be performed within the TXFIFO and a RF transmission of the ciphertext is started. When enabled, RX in-line security is started as follows: issuing a SRXDEC command strobe, so the first frame in the RXFIFO is then decrypted/authenticated as set by the current security mode. RX in-line security operations are always performed on the first frame currently inside the RXFIFO, even if parts of this has already been read out over the SPI interface. This allows the receiver to first read the source address out to decide which key to use before doing authentication of the complete frame. In CTR or CCM mode it is of course important that bytes to be decrypted are not read out before the security operation is started. The frame in the RXFIFO may be received over RF or it may be written into the RXFIFO over the SPI interface for debugging or higher layer security operations.

3.2.3.1 CTR mode encryption / decryption

CTR mode encryption/decryption is performed by CC2420 on MAC frames within the TXFIFO/RXFIFO respectively. SECCTRL1.SEC_TXL/SEC_RXL sets the number of bytes between the length field and the first byte to be encrypted/decrypted respectively. This controls the number of plaintext bytes in the current frame. When encryption is initiated, the plaintext in the TXFIFO is then encrypted. The encryption module will encrypt all the plaintext currently available and it will wait if not everything is prebuffered. The encryption operation may also be started without any data in the TXFIFO at all, and data will be encrypted as it is written to the TXFIFO. When decryption is initiated with the SRXDEC command strobe, the ciphertext of the RXFIFO is then decrypted.

3.2.3.2 CBC-MAC

CBC-MAC in-line authentication is provided by CC2420 hardware. SECCTRL0.SEC_M sets the MIC length M, encoded as (M-2)/2. When enabling CBC-MAC in-line TXFIFO authentication, the generated MIC is written to the TXFIFO for transmission. The frame length must include the MIC. SECCTRL1.SEC_TXL/SEC_RXL sets the number of bytes between the length field and the first byte to be authenticated, normally set to 0 for MAC authentication. SECCTRL0.SEC_CBC_HEAD defines if the authentication length is used as the first byte of data to be authenticated or not. This bit should be set to 1. When enabling CBC-MAC in-line RXFIFO authentication, the generated MIC is compared to the MIC in the RXFIFO. The last byte of the MIC is replaced in the RXFIFO with: 0x00 if MIC is correct, 0xFF if MIC is incorrect.

3.2.3.3 CCM

CCM combines CTR mode encryption and CBC-MAC authentication in one operation. SECCTRL1.SEC_TXL/SEC_RXL sets the number of bytes after the length field to be authenticated but not encrypted. The MIC is generated and verified very much like with CBC-MAC [5].

Chapter 4

Security Implementation

4.1 Overview

As we said in the previous chapters, IEEE 802.15.4 offers several way to secure a frame: packets can be only encrypted, only authenticated or encryption and authenticated. In our scenario,we consider two motes: a RFD sender that transmits ciphered data and a FFD coordinator that deciphers messages and sends acks. When security is off, upper layers send to MAC no security parameters: when the frame is built, security routines are not called and the frame is sent in clear text. The coordinator, while parsing the frame, understands that it has no security and does not proceed to unsecure. This exchange can be seen in figure 4.1.

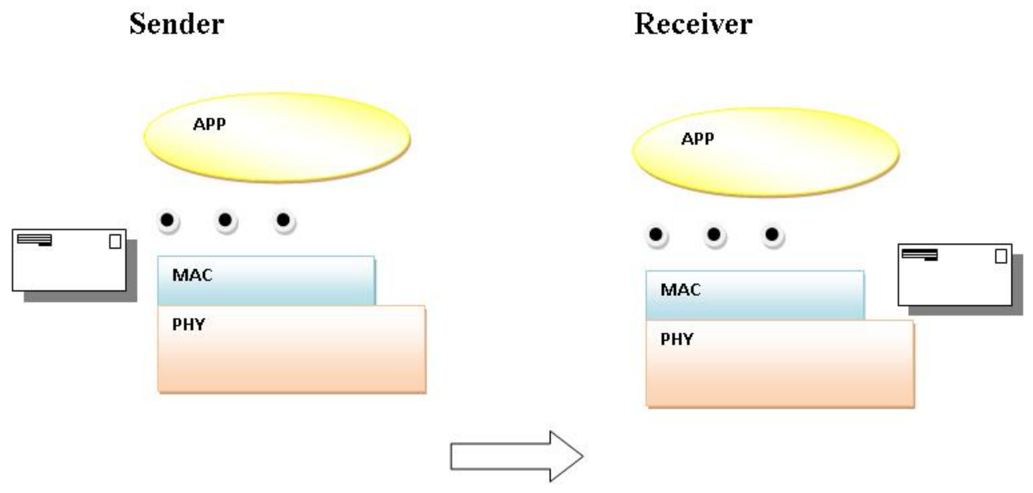


Figure 4.1: Frame transfer with no security

When security is on, upper layers send to MAC some security parameters: when the frame is built, Auxiliary Security Header is inserted in the frame and security routines are called.

Then security is applied to the frame and send. The coordinator, while parsing the frame, understands that it is not in clear text and does proceed to unsecure . This exchange can be seen in figure 4.2. Security parameters, from now we can refer to these as Security Structure, must

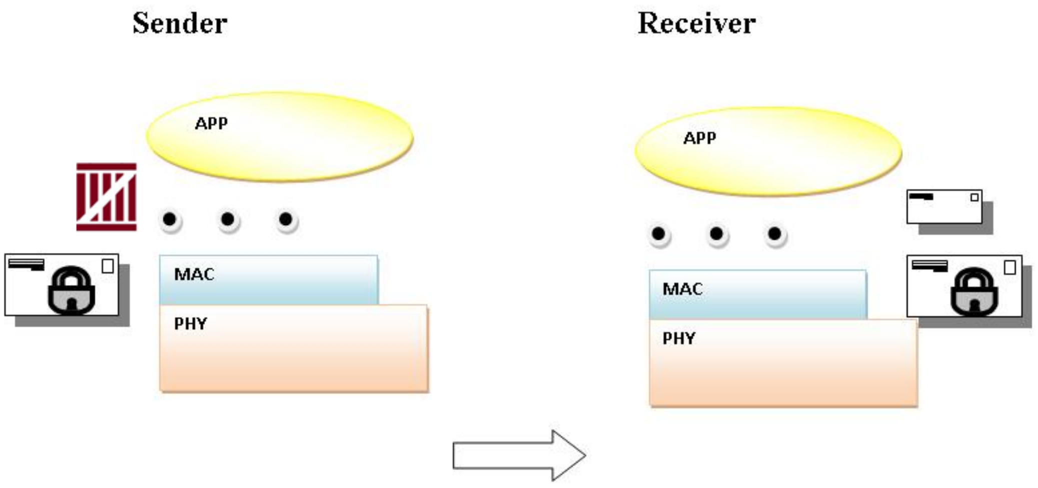


Figure 4.2: Frame transfer with security

contain the values in Table 4.1.

Name	**Type**	**Range**	**Description**
SecurityLevel	Integer	0x00 – 0x07	The security level to be used.(see Table 2.1)
KeyIdMode	Integer	0x00 – 0x03	The mode used to identify the key to be used. (see Table 2.2)
Key Source	Set of 0 4, or 8 octets	As specified by the KeyIdMode parameter	The originator of the key to be used. This parameter is ignored if the KeyIdMode parameter is ignored or set to 0x00.
KeyIndex	Integer	0x01 – 0xff	The index of the key to be used. This parameter is ignored if the KeyIdMode parameter is ignored or set to 0x00.

Table 4.1: Security Structure

This security structure is used by application layer to pass the security parameters when the frame is created. The implementation of the security structure is the following: [1]

```
//          tkn154/TKN154.h
typedef struct ieee154_security {
  uint8_t SecurityLevel;
  uint8_t KeyIdMode;
  uint8_t KeySource[8];
```

[1]All the code is Copyright (c) 2008, Technische Universitaet Berlin All rights reserved

```
  uint8_t KeyIndex;
} ieee154_security_t;
```

The actual frame protection provided can be adapted on a frame-by-frame basis and allows for varying levels of data authenticity (to minimize security overhead in transmitted frames where required) and for optional data confidentiality. For this reason all the routines handled by MAC management entity, called the MLME, has to manage at least one security structure. The MLME commands can be seen in Table 4.2. We will focus on the *MCPS-DATA.request*:

Name	**Description**
MLME-ASSOCIATE.request	Allows a device to request an association with a coordinator.
MLME-ASSOCIATE.indication	Indicates a reception of an association request command.
MLME-ASSOCIATE.response	Initiates a response to a MLME-ASSOCIATE.indication primitive.
MLME-ASSOCIATE.confirm	Informs the next higher layer of the initiating device whether its request to associate was successful or unsuccessful.
MLME-DISASSOCIATE.request	Notifies the coordinator of its intent to leave the PAN.
MLME-DISASSOCIATE.indication	The reception of a disassociation notification command.
MLME-GTS.indication	Indicates that a GTS has been allocated or that a previously allocated GTS has been deallocated.
MLME-ORPHAN.indication	Allows the MLME of a coordinator to notify the next higher layer of the presence of an orphaned device.
MLME-ORPHAN.response	Allows the next higher layer of a coordinator to respond to the MLME-ORPHAN.indication primitive.
MLME-SCAN.request	Is used to initiate a channel scan over a given list of channels.
MLME-COMM-STATUS.indication	Allows the MLME to indicate a communications status.
MLME-START.request	Allows the PAN coordinator to initiate a new PAN or to begin using a new superframe configuration.
MLME-SYNC-LOSS.indication	Indicates the loss of synchronization with a coordinator.
MLME-POLL.request	Prompts the device to request data from the coordinator.

Table 4.2: MLME commands

MAC common part sublayer (MCPS) data.request primitive requests the transfer of a data SPDU (i.e., MSDU) from a local SSCS (Service-specific convergence sublayer) entity to a single peer SSCS entity. In the Table 4.3 we can see the semantic of this command:

Name	**Type**	**Range**	**Description**
SrcAddrMode	Integer	0x00 – 0x03	The source addressing mode for this primitive and subsequent MPDU.
DstAddrMode	Integer	0x00 – 0x03	The destination addressing mode for this primitive and subsequent MPDU.
DstPANId	Integer	0x0000 – 0xffff	The 16-bit PAN identifier of the entity to which the MSDU is being transferred.
DstAddr	Device address	As specified by the DstAddrMode parameter	The individual device address of the entity to which the MSDU is being transferred.
msduLength	Integer	$\leq$ aMaxMACPayloadSize	The number of octets contained in the MSDU to be transmitted by the MAC sublayer entity.
msdu	Set of octets	—	The set of octets of MSDU to be transmitted by the MAC sublayer entity.
msduHandle	Integer	0x00 – 0xff	The handle associated with the MSDU to be transmitted by the MAC sublayer entity.
TxOptions	Bitmap	3-bit field	The 3 bits (b0, b1, b2) indicate the transmission options for this MSDU.
SecurityLevel	Integer	0x00 – 0x07	The security level to be used.(see Table 2.1)
KeyIdMode	Integer	0x00 – 0x03	The mode used to identify the key to be used. (see Table 2.2)
Key Source	Set of 0 4, or 8 octets	As specified by the KeyIdMode parameter	The originator of the key to be used. This parameter is ignored if the KeyIdMode parameter is ignored or set to 0x00.
KeyIndex	Integer	0x01 – 0xff	The index of the key to be used. This parameter is ignored if the KeyIdMode parameter is ignored or set to 0x00.

Table 4.3: MCPS-DATA.request primitive

4.2 Creation of data packets

4.2.1 MAC Header and Data Frame

As we can see in figure 4.3, a data frame contains the MAC Header, the MAC Payload and the Mac footer.

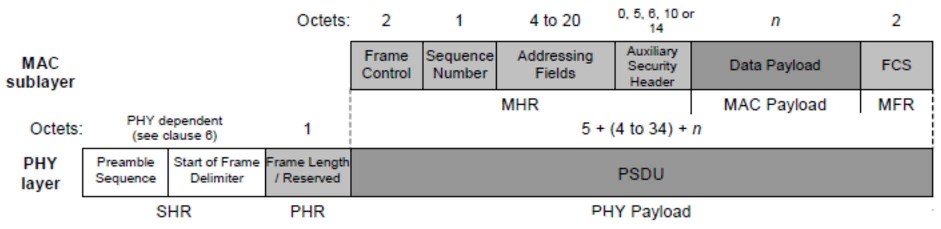


Figure 4.3: MAC Data Frame

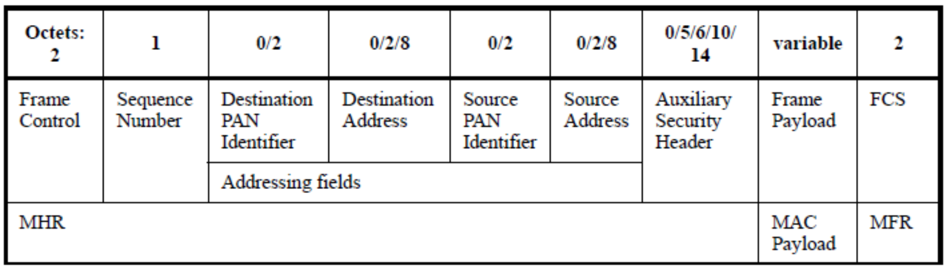
There are two main structures that describe the data packet: the generic message_t structure and the ieee154_txframe_t.

```
//        tos/types/message.h
typedef nx_struct message_t {
 nx_uint8_t header[sizeof(message_header_t)];
 nx_uint8_t data[TOSH_DATA_LENGTH];
 nx_uint8_t footer[sizeof(message_footer_t)];
 nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
//        tkn154/TKN154_MAC.h
typedef struct
{
 uint8_t client;
 uint8_t handle; // The set of octets forming the MSDU
 ieee154_header_t *header; //MAC Heder (MHR)
 uint8_t headerLen; // Length of MHR
 uint8_t *payload; //MAC payload
 uint8_t payloadLen;  // Length of Payload
 ieee154_metadata_t *metadata;
 ieee154_txframe_t;
```

The ieee154_txframe_t is a structure used for prepare the frame that will be sent. It contains the MAC header, the MAC payload and the MAC metadata; the MAC footer is added in a second time by CC2420 during transmission. Now we see the MAC Header must be and the how is implemented (fig. 4.4):

```
//        tkn154/TKN154_MAC.h
MHR_MAX_LEN = 37; // 23 + 14
typedef struct {
 uint8_t length;
 uint8_t mhr[MHR_MAX_LEN];
```

```c
} ieee154_header_t;
typedef struct {
 uint8_t rssi;
 uint8_t linkQuality;
 uint32_t timestamp;
} ieee154_metadata_t;
```

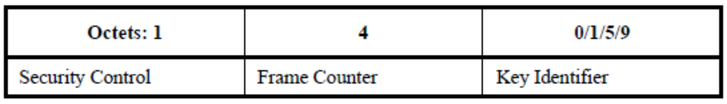


Octets: 2	1	0/2	0/2/8	0/2	0/2/8	0/5/6/10/14	variable	2
Frame Control	Sequence Number	Destination PAN Identifier	Destination Address	Source PAN Identifier	Source Address	Auxiliary Security Header	Frame Payload	FCS
		Addressing fields						
MHR							MAC Payload	MFR

Figure 4.4: MAC Frame

4.2.2 Auxiliary Security Header

The Auxiliary Security Header field shall be formatted as illustrated in figure 4.5 and ??.



Octets: 1	4	0/1/5/9
Security Control	Frame Counter	Key Identifier

Figure 4.5: Auxiliary Security Header

This is the implementation of the Auxiliary Security Header:

```c
//       tkn154/TKN154_MAC.h
typedef struct {
 uint8_t securityControl;
 uint32_t frameCounter;
 uint8_t keySource[8];
 uint8_t keyIndex;
} ieee154_security_header_t;
```

For setting the different subfields of Security Control field, can be used the macros in the same file.

```c
//       tkn154/TKN154_MAC.h
SEC_CNTL_LEVEL_POS      = 0,    // Position of Security Level in Security Control
SEC_CNTL_KEYIDMODE_POS  = 3,    // Position of KeyIDMode in Security Control
SEC_CNTL_RESERVED_POS   = 5,    // Position of Reserved in Security Control
```

4.2.3 Setting the MAC header

The MAC header is built by two different operations: the first will set all the fields regarding addressing and the security header; the second will set the other fields and prepare the ieee154_txframe_t packet that will be transfered.

4.2.3.1 setAddressingField function

Now we see the function that sets the addressing fields:

```
//        tkn154/PibP.nc
command error_t Frame.setAddressingFields(message_t* frame,
                                uint8_t srcAddrMode,
                                uint8_t dstAddrMode,
                                uint16_t dstPANId,
                                ieee154_address_t *dstAddr,
                                ieee154_security_t *security)
{
uint8_t *mhr = MHR(frame);
message_t* myframe = frame;
ieee154_address_t srcAddress;
ieee154_macPANId_t srcPANId = call MLME_GET.macPANId();
ieee154_security_t *temp = security;
mhr[MHR_INDEX_FC2] &= (FC2_DEST_MODE_MASK | FC2_SRC_MODE_MASK);
mhr[MHR_INDEX_FC2] |= dstAddrMode << FC2_DEST_MODE_OFFSET;
mhr[MHR_INDEX_FC2] |= srcAddrMode << FC2_SRC_MODE_OFFSET;
if (srcAddrMode == ADDR_MODE_SHORT_ADDRESS)
    srcAddress.shortAddress = call MLME_GET.macShortAddress();
else
    srcAddress.extendedAddress = call GetLocalExtendedAddress.get();
if (dstAddrMode>=ADDR_MODE_SHORT_ADDRESS &&
    srcAddrMode>=ADDR_MODE_SHORT_ADDRESS && dstPANId == srcPANId)
    mhr[MHR_INDEX_FC1] |= FC1_PAN_ID_COMPRESSION;
else
    mhr[MHR_INDEX_FC1] &= ~FC1_PAN_ID_COMPRESSION;
call FrameUtility.writeHeader(
            mhr,
            dstAddrMode,
            dstPANId,
            dstAddr,
            srcAddrMode,
            srcPANId,
            &srcAddress,
            (mhr[MHR_INDEX_FC1] & FC1_PAN_ID_COMPRESSION) ? TRUE: FALSE);

if (temp && (temp->SecurityLevel & SEC_CNTL_LEVEL))
{
#ifdef IEEE154_SECURITY_ENABLED
39     if (( temp->SecurityLevel & SEC_CNTL_LEVEL) == CTR)
40         call CC2420SecurityMode.setCtr(myframe,0,0,temp);
41     else if (( temp->SecurityLevel & SEC_CNTL_LEVEL) == CBC_MAC_4)
42         call CC2420SecurityMode.setCbcMac(frame,0, 0,MICLENGTH4,temp);
43     else if (( temp->SecurityLevel & SEC_CNTL_LEVEL) == CBC_MAC_8)
44         call CC2420SecurityMode.setCbcMac(frame,0, 0,MICLENGTH8,temp);
45     else if (( temp->SecurityLevel & SEC_CNTL_LEVEL) == CBC_MAC_16)
46         call CC2420SecurityMode.setCbcMac(frame,0, 0,MICLENGTH16,temp);
47    else if (( temp->SecurityLevel & SEC_CNTL_LEVEL) == CCM_4)
48         call CC2420SecurityMode.setCcm(frame,0, 0,MICLENGTH4,temp);
49    else if (( temp->SecurityLevel & SEC_CNTL_LEVEL) == CCM_8)
50         call CC2420SecurityMode.setCcm(frame,0, 0,MICLENGTH8,temp);
51    else if (( temp->SecurityLevel & SEC_CNTL_LEVEL) == CCM_16)
52         call CC2420SecurityMode.setCcm(frame,0, 0,MICLENGTH16,temp);
53 #else
54      status = IEEE154_UNSUPPORTED_SECURITY;
55 #endif
56 }
57  return SUCCESS;
58 }
```

The function *Frame.setAddressingFileds()* takes in these parameters:

- @ frame: The frame;
- @ srcAddrMode: The source addressing mode;
- @ dstAddrMode: The destination addressing mode;
- @ dstPANID: The 16 bit PAN identifier of the destination;
- @ dstAddr: Individual device address of the destination as per the dstAddrMode;
- @ security: the security options (NULL means security is disabled);

As we saw in the code, after declaration and initialization of local variables (lines 9 - 13), the subfields of the Frame Control Field regarding addressing are set (line 14 - 25) and then the addressing fields are written in the MHR by *FrameUtility.writeHeader()* (see Appendix A). Then, if the security structure that upper layer passed, is not NULL, the Security Level is not 0x00 and security is enabled (lines 36 - 38), depending of the value of Security Level, it is called a function that writes the Auxiliary Security Header in the MAC header. The CC2420SecurityMode interface takes the role of selecting the security mode and adding the security headers in the MAC Header by defining the commands *setCTR(), setCbcMac() and setCcm().* These commands are also extended with the security structure as a parameter.

command error_t setCtr() is called when the user wants to encrypt using the counter encryption mode. The command takes in three parameters as the following.

- **uint8_t setKey:** This parameter selects one of the two key registers.
- **uint8_t setSkip:** It is a CC2420 specific parameter that sets the number of bytes to skip in the payload without the affect of security functions.To be IEEE 802.15.4 compliant, the value should be set to 0.
- **ieee154_security_t *security:** Security values.

command error_t setCbcMac() is called when user wants to authenticate using the Cipher Block Chaining Message Authentication Code (CBC-MAC) mode. The command takes in four parameters as the following.

- **uint8_t setKey:** This parameter selects one of the two key registers.
- **uint8_t setSkip:** It is a CC2420 specific parameter that sets the number of bytes to skip in the payload without the affect of security functions.To be IEEE 802.15.4 compliant, the value should be set to 0.
- **uint8_t size:** This parameter sets the length of the MIC used in the authentication process. Sizes can be selected from 4, 8, and 16. If a different value is selected, the command returns a FAIL.
- **ieee154_security_t *security:** Security values.

command error_t setCcm() is called when user wants to encrypt and authenticate using the Counter with CBC-MAC (CCM) mode. The command takes in four parameters as the following.

- **uint8_t setKey:** This parameter selects one of the two key registers.
- **uint8_t setSkip:** It is a CC2420 specific parameter that sets the number of bytes to skip in the payload without the affect of security functions.To be IEEE 802.15.4 compliant, the value should be set to 0.
- **uint8_t size:** This parameter sets the length of the MIC used in the authentication process. Sizes can be selected from 4, 8, and 16. If a different value is selected, the command returns a FAIL.
- **ieee154_security_t *security:** Security values.

4.2.4 Setting the Auxiliary Security Header

Now we see how the Auxiliary Security Header is written in MHR with setCtr():

```
//      tkn154/PibP.nc
command error_t CC2420SecurityMode.setCtr(message_t* frame, uint8_t setKey,
                                          uint8_t setSkip,ieee154_security_t *security)
{
uint8_t *mhr = MHR(frame);
ieee154_security_t *mysec = security;
uint8_t len = 0;
uint8_t offset = 0;
ieee154_macFrameCounter_t macCounter;
if (setKey > 1 || setSkip > 7){
      return FAIL;
}
if( mysec -> SecurityLevel > 7 || mysec -> KeyIdMode > 4)
        return FAIL;
call  FrameUtility.getAddressingFieldsLength(mhr[0], mhr[1], &len);
offset = len;
mhr[MHR_INDEX_FC1] |=  FC1_SECURITY_ENABLED;
mhr[offset] = (mysec-> SecurityLevel << SEC_CNTL_LEVEL_POS);
mhr[offset] |= setSkip <<  SEC_CNTL_RESERVED_POS;
mhr[offset++] |= (mysec ->KeyIdMode <<  SEC_CNTL_KEYDMODE_POS);
macCounter = call MLME_GET.macFrameCounter();
if ( macCounter == 0xffffffff)
        return   IEEE154_COUNTER_ERROR;
*((nx_uint32_t*) (&(mhr[offset]))) = macCounter;
offset +=4;
call MLME_SET.macFrameCounter(++macCounter);
 if ( mysec->KeyIdMode & SEC_CNTL_KEYIDMODE ) {
        if ((mysec->KeyIdMode & SEC_CNTL_KEYIDMODE) == KEYIDMODE1) {
                mhr[offset++] = mysec->KeyIndex;
        }
        else if ( (mysec->SecurityLevel & SEC_CNTL_KEYIDMODE) == KEYIDMODE2) {
                mhr[offset++] = mysec->KeySource[0];
                mhr[offset++] = mysec->KeySource[1];
                mhr[offset++] = mysec->KeySource[2];
                mhr[offset++] = mysec->KeySource[3];
                mhr[offset++] = mysec->KeyIndex;
        }
        else {
                mhr[offset++] = mysec->KeySource[0];
                mhr[offset++] = mysec->KeySource[1];
                mhr[offset++] = mysec->KeySource[2];
                mhr[offset++] = mysec->KeySource[3];
                mhr[offset++] = mysec->KeySource[4];
                mhr[offset++] = mysec->KeySource[5];
                mhr[offset++] = mysec->KeySource[6];
                mhr[offset++] = mysec->KeySource[7];
47                    mhr[offset++] = mysec->KeyIndex;
48          }
49    }
50
51  return SUCCESS;
52  }
```

After consistency checks (line 10 - 14), the function *FrameUtility.getAddressingFieldsLength()* (see Appendix A) returns the addressing fields length (so the position where Auxiliary Security Header starts). Then the Security Enabled Bit in Frame Control Field and Security Control Field are set (lines 17 and 18-20 respectively), after it is set the macFrameCounter field and updated (lines 22-26). Finally depending of the value of KeyID mode, the others field of the Auxiliary Security Header are set by the value of the security structure. (lines 27 - 48). *CC2420SecurityMode.setCbcMac(), CC2420SecurityMode.setCCM()* and *Frame.writeSecurityMHR()* (another function that writes the Auxiliary Security Header in some cases) do the same things as *CC2420SecurityMode.setCtr()*. For detailed codes, see the Appendix A.

4.2.4.1 MCPS_DATA.request

The *MCPS_DATA.request()* takes in these parameters:

- @ message_t *frame: The frame;
- @ uint8_t payloadLen: The length of the payload;
- @ uint8_t msduHandle: Msdu parameter;
- @ uint8_t txOptions: Transmission options

Now we see how this command is implemented.

```
//      tkn154/DataP.nc
 command ieee154_status_t MCPS_DATA.request   (
                                message_t *frame,
                                uint8_t payloadLen,
                                uint8_t msduHandle,
                                uint8_t txOptions
                              )
{
uint8_t srcAddrMode = call Frame.getSrcAddrMode(frame);
uint8_t dstAddrMode = call Frame.getDstAddrMode(frame);
ieee154_address_t dstAddr;
ieee154_status_t txStatus;
ieee154_txframe_t *txFrame;
uint8_t sfType=0;
```

```
uint8_t *mhr;
if (payloadLen > call Packet.maxPayloadLength())
    txStatus = IEEE154_INVALID_PARAMETER;
else if ((!srcAddrMode && !dstAddrMode) ||
        (srcAddrMode > ADDR_MODE_EXTENDED_ADDRESS ||
        dstAddrMode > ADDR_MODE_EXTENDED_ADDRESS) ||
        (srcAddrMode == ADDR_MODE_RESERVED ||
        dstAddrMode  == ADDR_MODE_RESERVED))
                        txStatus = IEEE154_INVALID_ADDRESS;
else if (!(txFrame = call TxFramePool.get())) {
        txStatus = IEEE154_TRANSACTION_OVERFLOW;
 } else {
        txFrame->header = &((message_header_t*) frame->header)->ieee154;
        txFrame->payload = (uint8_t*) frame->data;
        txFrame->metadata = &((message_metadata_t*) frame->metadata)->ieee154;
        txFrame->payloadLen = payloadLen;
        mhr = txFrame->header->mhr;
        txFrame->headerLen = call Frame.getHeaderLength(frame);
        mhr[MHR_INDEX_FC1] &= ~(FC1_FRAMETYPE_MASK |  FC1_FRAME_PENDING | FC1_ACK_REQUEST);
        mhr[MHR_INDEX_FC1] |= FC1_FRAMETYPE_DATA;
        if (txOptions  & TX_OPTIONS_ACK)
           mhr[MHR_INDEX_FC1] |= FC1_ACK_REQUEST;
        mhr[MHR_INDEX_FC2] &= ~FC2_FRAME_VERSION_MASK;
        if (payloadLen > IEEE154_aMaxMACSafePayloadSize)
            mhr[MHR_INDEX_FC2] |= FC2_FRAME_VERSION_1;
         txFrame->handle = msduHandle;
        #ifdef IEEE154_SECURITY_ENABLED
        if ( mhr[MHR_INDEX_FC1] & FC1_SECURITY_ENABLED)
             mhr[MHR_INDEX_FC2] |=  FC2_FRAME_VERSION_1;
        #endif
        call Frame.getDstAddr(frame, &dstAddr);
        if (dstAddrMode == ADDR_MODE_SHORT_ADDRESS){
             if (dstAddr.shortAddress == call MLME_GET.macCoordShortAddress())
                        sfType = INCOMING_SUPERFRAME;
                     else
                        sfType = OUTGOING_SUPERFRAME;
      } else if (dstAddrMode == ADDR_MODE_EXTENDED_ADDRESS){
                    if (dstAddr.extendedAddress == call MLME_GET.macCoordExtendedAddress())
                        sfType = INCOMING_SUPERFRAME;
                    else
                        sfType = OUTGOING_SUPERFRAME;
                } else if (dstAddrMode == ADDR_MODE_NOT_PRESENT)
                        sfType = INCOMING_SUPERFRAME
         if (txOptions & TX_OPTIONS_GTS){
                           if (sfType == INCOMING_SUPERFRAME)
                        txStatus = call DeviceCfpTx.transmit(txFrame);
           else
                        txStatus = call CoordCfpTx.transmit(txFrame);
      } else if ((txOptions & TX_OPTIONS_INDIRECT) &&
64                      call IsSendingBeacons.getNow() &&
65                              (dstAddrMode >= ADDR_MODE_SHORT_ADDRESS)){
66                                  if (dstAddrMode == ADDR_MODE_SHORT_ADDRESS &&
67                                 dstAddr.shortAddress == 0xFFFF){
68                                    mhr[MHR_INDEX_FC1] &= ~FC1_ACK_REQUEST;
69                                     txStatus = call BroadcastTx.transmit(txFrame);
70                            } else
71                                    txStatus = call IndirectTx.transmit(txFrame);
72
73        } else {
74       if (sfType == INCOMING_SUPERFRAME)
75           txStatus = call DeviceCapTx.transmit(txFrame);
76          else
77                   txStatus = call CoordCapTx.transmit(txFrame);
78       if (txStatus != IEEE154_SUCCESS){
79          call TxFramePool.put(txFrame);
80        }
81      }
82 return txStatus
83 }
```

After checking the correctness of the arguments (line 16 - 23), the transmission frame is built (26 - 33); then AckPending and FrameVersion bits are set (lines 36 - 47). After that, in case a node is both, coordinator and device, it has to be decided whether the frame is to be sent in the incoming or outgoing superframe: it is done we do this by comparing the destination address to the coordinator address in the PIB, if they match the frame is sent in the incoming sf otherwise in the outgoing superframe (lines 49 - 61). Finally it is set Gts options (if enabled) and the transmitting mode (direct or indirect) in lines 73 -84.

4.3 Sending and Receiving

4.3.1 Transmitting packets

Transmission process involves two steps: firstly is inserting the frame in transmission buffer, secondly is sending what is contained in it.

4.3.1.1 Load TXFIFO

Transmission buffer is the register TXFIFO of CC2420. The TXFIFO is write only, but may be read back using RAM. This register can be used with CC2420Fifo interface that offers the HAL abstraction for accessing the FIFO registers of a ChipCon CC2420 radio. This interface provides a command that writes the FIFO and an event that signals the completion of a write operation.

async command cc2420_status_t write( ) is used to write byte in FIFOs.

async event void writeDone( ) is an event signaled when the write is finished.

- @ uint8_t* COUNT_NOK(length) data: pointer to data buffer;
- @ uint8_t length: byte to write/written;

See now how this routine is implemented.

```
1  //        chips/cc2420_tkn154/CC2420TransmitP
2  async command error_t CC2420Tx.loadTXFIFO(ieee154_txframe_t *data)
3  {
4  atomic {
5  if ( m_state != S_STARTED )
6          return FAIL;
7  m_state = S_LOAD;
8  m_frame = data;
9  m_frame->header->length = m_frame->headerLen + m_frame->payloadLen + 2; //2 for CRC
10 call CSN.clr();
11 call SFLUSHTX.strobe(); // flush out anything that was in TXFIFO
12 call CSN.set();
13 call CSN.clr();
14 call TXFIFO.write( &(m_frame->header->length), 1 );
15 }
16 return SUCCESS;
17 }
18
19 async event void TXFIFO.writeDone( uint8_t* tx_buf, uint8_t tx_len, error_t error)
20 {
21  atomic {
22 call CSN.set();
23 if (tx_buf == &(m_frame->header->length)){
24         call CSN.clr();
25         call TXFIFO.write( m_frame->header->mhr, m_frame->headerLen );
26         return;
27 } else if (tx_buf == m_frame->header->mhr) {
28         call CSN.clr();
29         call TXFIFO.write( m_frame->payload, m_frame->payloadLen );
30         return;
31         }
32     }
33 m_state = S_READY_TX;
34 signal CC2420Tx.loadTXFIFODone(m_frame, error);
35 }
```

Firstly, we must check if the radio is in the correct state; if so, it is switched in loading state. Afterward the TXFIFO is flushed and the length of the frame is written in transmission buffer(line

14): the CRC is written by CC2420 that needs two more bytes in FIFO, so the frame we will write, has to be two bytes longer (line 9). When we finished to write the length, and the *writedone()* is signaled, we write the header (line 27) and in the next iteration the payload (line 31). Finally the state changes in ready_to_transfer state and the end of operation is signaled.

4.3.1.2 Send

Loaded the TXFIFO, we choose the sending mode: we can use *RadioTx.transmit()*, transmitting withouCCA, *RadioTx.transmitUnslottedCsmaCa()*, transmittig with a single CCA or *RadioTx.transmitSlottedCsmaCa()*. Chosen one of that, with all implications, send command can be called.

```
//      chips/cc2420_tkn154/CC2420TransmitP
async command error_t CC2420Tx.send(bool cca){
cc2420_status_t status;
bool congestion = TRUE;
atomic {
if (m_state != S_READY_TX)
        return EOFF;
#ifdef IEEE154_SECURITY_ENABLED
if ((m_frame->header->mhr[0] & FC1_SECURITY_ENABLED)?  TRUE : FALSE  & (!ALREADY_CHIPERED)){
        securityCheck();
}
#endif
call CSN.set();
call CSN.clr();
status = cca ? call STXONCCA.strobe() : call STXON.strobe();
if ( !( status & CC2420_STATUS_TX_ACTIVE ) ) {
        status = call SNOP.strobe();
        if ( status & CC2420_STATUS_TX_ACTIVE ) {
                        congestion = FALSE;
          }
      }
call CSN.set();
if (congestion)
        return FAIL;
else {
            m_state = S_SFD;
        #ifdef IEEE154_SECURITY_ENABLED
        ALREADY_CHIPERED = 0;
        #endif
        call BackoffAlarm.start(CC2420_ABORT_PERIOD);
                return SUCCESS;
      }
    }
  }
```

If the radio is in the correct state (line 7), we look in the frame to see if it needs authentication/encryption (lines 11 - 13). We focus on this point in the next section. Then we effectively send the packet with the strobe command and we read the status byte. If we have no errors, the radio changes to the next state (line 34) and the ALREADY_CIPHERED bit is reset (it means that the next packet, eventually, is not already ciphered) or we assume that the channel is busy and we try to retransmit (if a encrypted packet has to be retransmit, we do not encrypt it again because it is already encrypted).

4.3.2 Receiving packets

Receiving packets operation consists on reading the receiving buffer RXFIFO of CC2420: it is both writeable and readable. Writing to the RXFIFO should however only be done for debugging or for using the RXFIFO for security operations (decryption/authentication). This register can be used with CC2420Fifo interface that provides the following read routines:
async command cc2420_status_t beginRead( ) is used for reading from FIFOs.
async command error_t continueRead( ) is used for reading from FIFOs. without sending address bytes.
async event void readDone() it is an event that signals the end of reading operation.

- @ uint8_t* COUNT_NOK(length) data : A pointer to the receive buffer.
- @ uint8_t length : Length number of bytes read.

The *beginRead()* routine starts reading from the FIFO, *continueRead()* continues reading from the FIFO without having to send the address byte again. Event *readDone()* will be signaled upon completion of both routines. Now we see how they are implemented.

```
1 //        chips/cc2420_tn154/CC2420ReceiveP
2 void receive() {
3 #ifdef IEEE154_SECURITY_ENABLED
4     header_parsing_function();
5 #endif
6 call CSN.clr();
7 call RXFIFO.beginRead( &((ieee154_header_t*) m_rxFramePtr->header)->length , 1 );
8 }
```

Receive process starts with the *receive()*: first operation that must be done, is understand if the packet we will read in the buffer, is secured or not. This is the job of *header_parsing_function()* (line 5) that will be analyzed in the next section. So it can start reading from the RXFIFO: the first read tells us the length byte, then it is signaled the *readDone()*.

```
1 async event void RXFIFO.readDone( uint8_t* rx_buf, uint8_t rx_len,
2                                    error_t error ) {
```

```
uint8_t* buf;
atomic {
buf = (uint8_t*) &((ieee154_header_t*) m_rxFramePtr->header)->length;
rxFrameLength = ((ieee154_header_t*) m_rxFramePtr->header)->length;
 switch( m_state ) {
        case S_RX_LENGTH:
         m_state = S_RX_FCF;
         m_mhrLen2 = 0;
         if ( rxFrameLength + 1 > m_bytes_left ) {
        // Length of this packet is bigger than the RXFIFO, flush it out.
                        flush();
                } else {
         if ( !call FIFO.get() && !call FIFOP.get() ) {
                        //m_bytes_left -= rxFrameLength + 1;
                        flush();
        }

        //if(rxFrameLength <= MAC_PACKET_SIZE) {
        if(rxFrameLength <= (sizeof(ieee154_header_t) - 1 + TOSH_DATA_LENGTH + 2)){
                        if(rxFrameLength > 0) {
                        if(rxFrameLength > SACK_HEADER_LENGTH) {
                // This packet has an FCF byte plus at least one more byte to read
                                call RXFIFO.continueRead(buf + 1,
                                SACK_HEADER_LENGTH);
                                break;
                        } else {
                // This is really a bad packet, skip FCF and get it out of here.
                         flush();
                        //m_state = S_RX_PAYLOAD;
                        //call RXFIFO.continueRead(buf + 1, rxFrameLength);
            }

          } else {
            // Length == 0; start reading the next packet
            flush();
/*          atomic receivingPacket = FALSE;*/
/*          call CSN.set();*/
/*          call SpiResource.release();*/
/*          waitForNextPacket();*/
          }

        } else {
          // Length is too large; we have to flush the entire Rx FIFO
          flush();
        }
      }
      break;
        case S_RX_FCF:
        if (call FrameUtility.getAddressingFieldsLength(buf[1], buf[2], &m_mhrLen)
            != SUCCESS || m_mhrLen > rxFrameLength - 2) {
            // header size incorrect
                    flush();
                    break;
            } else if (m_mhrLen > SACK_HEADER_LENGTH) {
                    m_state = S_RX_HEADER;
//Reading if the frame is secured
                    #ifdef IEEE154_SECURITY_ENABLED
                    if(buf[1] & FC1_SECURITY_ENABLED) {
                            m_state = S_RX_SECURITY;
                    //<C.C.> I have to read one more byte to know how tht ASH in long.
                    call RXFIFO.continueRead(buf + 1 + SACK_HEADER_LENGTH,
                            m_mhrLen - SACK_HEADER_LENGTH +1);
                            break;
}
                    #endif
            call RXFIFO.continueRead(buf + 1 + SACK_HEADER_LENGTH,
                m_mhrLen - SACK_HEADER_LENGTH);
            break;
          } else {

            // complete header has been read: fall through
          }
          // fall through
            case S_RX_HEADER:
          // JH: we are either using HW ACKs (normal receive mode) or don't ACK any
          // packets (promiscuous mode)
          // Didn't flip CSn, we're ok to continue reading
            if ((rxFrameLength - (m_mhrLen+m_mhrLen2) - 2) >
                    TOSH_DATA_LENGTH) // 2 for CRC
                            flush();
            else {
                     m_state = S_RX_PAYLOAD;
                             call RXFIFO.continueRead((uint8_t*) m_rxFramePtr->data,
                              rxFrameLength - (m_mhrLen+m_mhrLen2 ));
            }
                    break;

            case S_RX_SECURITY:
            if (call FrameUtility.getSecurityHeaderLength(buf[1],
             buf[m_mhrLen + 1], &m_mhrLen2)!= SUCCESS ||
              (m_mhrLen + m_mhrLen2) > rxFrameLength - 2) {
            // header size incorrect
                            flush();
                            break;
            }

            m_state =S_RX_HEADER;
            call RXFIFO.continueRead(buf + 2  + m_mhrLen, m_mhrLen2  - 1);
            break;

            case S_RX_PAYLOAD:
            call CSN.set();
            if(!m_missed_packets) {
                    // Release the SPI only if there are no more frames to download
                            call SpiResource.release();
            }
                if ( m_timestamp_size ) {
                    if ( rxFrameLength > 10 ) {
                    memcpy(&m_timestamp, &m_timestamp_queue[ m_timestamp_head ],
                    sizeof(ieee154_reftime_t) );
                    m_timestampValid = TRUE;
                            m_timestamp_head = ( m_timestamp_head + 1 ) %
                            TIMESTAMP_QUEUE_SIZE;
                            m_timestamp_size--;
            }
        } else {
/*                  metadata->time = 0xffff;*/
                    m_timestampValid = FALSE;
             }

        // We may have received an ack that should be processed by Transmit
        // buf[rxFrameLength] >> 7 checks the CRC
          if ( ( m_rxFramePtr->data[ rxFrameLength - (m_mhrLen+m_mhrLen2) - 1 ] >> 7 ) && rx_buf ) {
          uint8_t type = ((ieee154_header_t*) m_rxFramePtr->header)->mhr[0] & 0x07;
/*          signal CC2420Receive.receive( type, m_p_rx_buf );*/
          signal CC2420Receive.receive( type, m_rxFramePtr );
/*          if ( type == IEEE154_TYPE_DATA ) {*/
          if ( (type != IEEE154_TYPE_ACK ||
          call CC2420Config.isPromiscuousModeEnabled())
              && !m_stop) {
          post receiveDone_task();
          return;
          }
        }
waitForNextPacket();
break;
default:
atomic receivingPacket = FALSE;
call CSN.set();
call SpiResource.release();
if (m_stop){
            continueStop();
            return;
        }
break;
}
}
```

```
150 }
```

The first iteration of the *readDone()* (lines 19 - 57), after consistency checks about frame length, is reading Frame Control Fields and Sequence Number (3 bytes) of the MHR: they are read with *continueRead()* at line 37; then, if the packet has the addressing fields (line 64) we have two options: if Security Enabled flag is set, we will have to receive the the Auxiliary Security Header, so we will read all addressing fields and the first byte of Auxiliary Security Header(line 77); if Security Enabled is not set, we will not have the Auxiliary Security Header, so we will read only addressing fields. Assume that packet was ciphered, the next step is reading the security header in S_RX_SECURITY state (line 109 - 126): we discover how long is the Auxiliary Security Header (line 117) and read the same numbers of byte minus one from RXFIFO because the Security Control Field has been already read. Finally (line 128 - 162) , we receive payload and, if we receive an acknowledgment, the transmission process ends, if not, post *receiveDone_task()* that complete the receiving procedure.

4.4 Security Procedures

4.4.1 Ciphering

Ciphering/authentication process is done by *securitycheck()* as we saw in the previous section. This function sets all the parameters CC2420 needs to secure a packet recovering them from the Auxiliary Security Header and the internal security structure or, in this case, statically.

```
//       chips/CC2420_tkn154/CC2420TransmitP.nc
void securityCheck() {
uint8_t mode; //Variable that describe cipher/auth mode
bool key;  // "Ram" position of the Key
uint8_t micLength; //Auth parameter
uint16_t currentStatus;
cc2420_status_t status;
uint8_t AddLen; //Addressing field dimension
uint8_t SecLen; //Security field dimension
uint8_t SecLevel = 0; //Security Control Field
uint8_t start_CipAuth =0; //Position where Authentication/Ciphering start
if(SECURITYLOCK == 1){
        post waitTask();
}else {
      //Will perform encryption lock registers
atomic SECURITYLOCK = 1;
call FrameUtility.getAddressingFieldsLength(m_frame->header->mhr[MHR_INDEX_FC1],
m_frame->header->mhr[MHR_INDEX_FC2],&AddLen);
call FrameUtility.getSecurityHeaderLength(m_frame->header->mhr[MHR_INDEX_FC1],
m_frame->header->mhr[AddLen], &SecLen);
```

```
SecLevel = (m_frame->header->mhr[AddLen] & SEC_CNTL_LEVEL);
if (SecLevel == NO_SEC){
        mode = CC2420_NO_SEC;
        micLength = MICLENGTH4;
}else if (SecLevel == CBC_MAC_4){
        mode = CC2420_CBC_MAC;
        micLength = MICLENGTH4;
}else if (SecLevel == CBC_MAC_8){
        mode = CC2420_CBC_MAC;
        micLength = MICLENGTH8;
}else if (SecLevel == CBC_MAC_16){
        mode = CC2420_CBC_MAC;
        micLength = MICLENGTH16;
}else if (SecLevel == CTR){
        mode = CC2420_CTR;
        micLength = MICLENGTH4;
}else if (SecLevel == CCM_4){
        mode = CC2420_CCM;
        micLength = MICLENGTH4;
}else if (SecLevel == CCM_8){
        mode = CC2420_CCM;
        micLength = MICLENGTH8;
}else if (SecLevel == CCM_16){
        mode = CC2420_CCM;
        micLength = MICLENGTH16;
}else{
        //something is wrong here
        return;
}
key = KEYREG0; // Using the KEYO RAM location for key
start_CipAuth = AddLen + SecLen; //Payload Position
CTR_SECCTRL0 = ((mode << CC2420_SECCTRL0_SEC_MODE) |
                ((micLength-2)/2 << CC2420_SECCTRL0_SEC_M) |
                (key << CC2420_SECCTRL0_SEC_TXKEYSEL) |
                (1 << CC2420_SECCTRL0_SEC_CBC_HEAD));
CTR_SECCTRL1 = (start_CipAuth << CC2420_SECCTRL1_SEC_TXL);
//WRITE KEY0 REGISTER
 call CC2420Keys.setKey(key,mykey);
//WRITE TXNONCE REGISTER
nonce_building_function(AddLen,micLength);
call CSN.clr();
call TXNONCE.write(RAM_START, nonceValue, BITDIM16);
call CSN.set();
//WRITE SECCTRL0 REGISTER
call CSN.clr();
call SECCTRL0.write(CTR_SECCTRL0);
call CSN.set();
//WRITE SECCTRL1 REGISTER
call CSN.clr();
```

```
70 call SECCTRL1.write(CTR_SECCTRL1);
71 call CSN.set();
72 call CSN.clr();
73 status = call SNOP.strobe();
74 call CSN.set();
75 while(status & CC2420_STATUS_ENC_BUSY){
76         uwait(1*1024);
77         call CSN.clr();
78         status = call SNOP.strobe();
79         call CSN.set();
80 }
81 call CSN.clr();
82 call STXENC.strobe();
83 call CSN.set();
84 call CSN.clr();
85 call SECCTRL0.read(&currentStatus);
86 call CSN.set();
87 call CSN.clr();
88 call SECCTRL0.write(currentStatus && ~(3 << CC2420_SECCTRL0_SEC_MODE));
89 call CSN.set();
90 atomic SECURITYLOCK = 0;
91 ALREADY_CHIPERED = 1;
92   }
93 }
```

The first operation for setting security parameters is locking the registers: we wait until they are idle (line 14). After recovering the Auxiliary Security Header position from the frame (lines 20 - 21, see Appendix A for utility functions code), working mode and micLength are set from Security Control Field (lines 26 - 53). Then CTR_SECTRL0 is prepared for writing in SECCNTL0 register of CC2420 and in the same way SECCTRL1 (lines 75 - 83) : first contains working mode, key used, MIC length and other options; second the position where ciphering/authentication starts. (see Tables 3.4 and 3.5 in chapter 3.2.3 for detail). Afterwards *CC2420Keys.setkey()* command puts the static key in the appropriate register (see Appendix A for details), and *nonce_building_function()* (line 73) builds the nonce in according of CC2420 specification. If the security module is idle (lines 90 - 100), we can encrypt/authenticate the frame in TXFIFO. Finally, after clean up (lines 107 - 114), we unlock the registers and enabled the already_ciphered flag: if the frame is not sent for any reason, it does not need another encryption/authentication. When we send a encrypted frame, receiver is slower because it has more bytes to read (authentication) or it has to performs deciphering(encryption). So sender has to wait a little longer that receiver sends an acknowledgment to it. This can be done with the following code:

```
//        chips/cc2420_tkn154/CC2420TransmitP
#ifdef IEEE154_SECURITY_ENABLED
        if ((m_frame->header->mhr[MHR_INDEX_FC1] & FC1_SECURITY_ENABLED))
                call BackoffAlarm.start(200);
        else
                call BackoffAlarm.start(100);
#else
        call BackoffAlarm.start(100);
#endif
```

4.4.2 Deciphering

Deciphering/authentication is performed in two steps: initially we access the RAM to discover if the received packet is secured and so the real decipher procedure.

```
//        chips/cc2420_tkn154/CC2420ReceiveP.
void header_parsing_function ()
{
uint8_t temp = 0;
ieee154_header_t header;
atomic pos = (packetLength + pos) % RXFIFO_SIZE;
if (pos + (LENDIM + FCDIM) > RXFIFO_SIZE){
        temp = RXFIFO_SIZE - pos;
        call CSN.clr();
        atomic call RXFIFO_RAM.read(pos,(uint8_t*)&header, temp);
        call CSN.set();
        call CSN.clr();
        atomic call RXFIFO_RAM.read(RAM_START,(uint8_t*)&header+temp,
        (LENDIM + FCDIM)-temp);
        call CSN.set();
}else{
        call CSN.clr();
        atomic call RXFIFO_RAM.read(pos,(uint8_t*)&header, LENDIM + FCDIM );
        call CSN.set();
    }
if( header.mhr[MHR_INDEX_FC1] & FC1_SECURITY_ENABLED ) {
        dec();
        return;
}else {
        packetLength = header.length +1;
        return;
       }
}
```

Header_parsing_function() accesses to the RAM and reads how the packet is long and Frame Control Field. Dimension of packet is important because RAM is a circular buffer, so we have to know the replenish status of the RAM and the offset of the next packet; Frame Control Field tells us if the Security Enabled bit is set: in that case, *dec()* is called and deciphering operation is started.

```
//        chips/cc240_tkn154/CC2420ReceiveP.nc
void dec(){
uint8_t key =0 ,mode = 0;
uint8_t AddLen = 0; //Addressing field dimension
uint8_t SecLen = 0; //Security field dimension
uint8_t SecLevel = 0; //Security Control Field
uint8_t start_CipAuth =0; //Position where Authentication/Ciphering start
uint8_t temp = 0;
ieee154_header_t header;
cc2420_status_t status;
ieee154_address_t *srcAddress = NULL;
```

```
if(SECURITYLOCK == 1){
          post waitTask();
          return;
}else{
atomic SECURITYLOCK = 1;
if (pos + (LENDIM + FCDIM) > RXFIFO_SIZE){
          temp = RXFIFO_SIZE - pos;
          call CSN.clr();
          atomic call RXFIFO_RAM.read(pos,(uint8_t*)&header, temp);
          call CSN.set();
          call CSN.clr();
          atomic call RXFIFO_RAM.read(RAM_START,(uint8_t*)&header+temp,
          (LENDIM + FCDIM)-temp);
          call CSN.set();
}else{
          call CSN.clr();
          atomic call RXFIFO_RAM.read(pos,(uint8_t*)&header,
          LENDIM + FCDIM );
          call CSN.set();
}
call FrameUtility.getAddressingFieldsLength(header.mhr[MHR_INDEX_FC1],
header.mhr[MHR_INDEX_FC2],&AddLen);
// NOW I read FROM THE START to the first byte of Auxiliary Security Header
if (pos + ( AddLen + LENDIM + FCDIM) > RXFIFO_SIZE){
          temp = RXFIFO_SIZE - pos;
          call CSN.clr();
          atomic call RXFIFO_RAM.read(pos,(uint8_t*)&header, temp);
          call CSN.set();
          call CSN.clr();
          atomic call RXFIFO_RAM.read(RAM_START,(uint8_t*)&header+temp,
          ( AddLen + LENDIM + FCDIM) -temp);
          call CSN.set();
}else{
          call CSN.clr();
           atomic call RXFIFO_RAM.read(pos,(uint8_t*)&header, ( AddLen + LENDIM + FCDIM));
          call CSN.set();
}
call FrameUtility.getSecurityHeaderLength(header.mhr[MHR_INDEX_FC1],
header.mhr[AddLen], &SecLen);
//Now i read all until the end of Auxiliary Security Header
if (pos + ( AddLen + LENDIM + FCDIM + SecLen) > RXFIFO_SIZE){
          temp = RXFIFO_SIZE - pos;
          call CSN.clr();
          atomic call RXFIFO_RAM.read(pos,(uint8_t*)&header, temp);
          call CSN.set();
          call CSN.clr();
          atomic call RXFIFO_RAM.read(RAM_START,(uint8_t*)&header+temp,
          ( AddLen + LENDIM + FCDIM +SecLen ) -temp);
            call CSN.set();
}else{
        call CSN.clr();
        atomic call RXFIFO_RAM.read(pos,(uint8_t*)&header,
        ( AddLen + LENDIM + FCDIM + SecLen));
        call CSN.set();
 }
SecLevel = (header.mhr[AddLen] & SEC_CNTL_LEVEL);
if (SecLevel == NO_SEC){
        mode = CC2420_NO_SEC;
        micLength = MICLENGTH4;
}else if (SecLevel == CBC_MAC_4){
        mode = CC2420_CBC_MAC;
        micLength = MICLENGTH4;
}else if (SecLevel == CBC_MAC_8){
        mode = CC2420_CBC_MAC;
        micLength = MICLENGTH8;
}else if (SecLevel == CBC_MAC_16){
        mode = CC2420_CBC_MAC;
        micLength = MICLENGTH16;
}else if (SecLevel == CTR){
        mode = CC2420_CTR;
        micLength = MICLENGTH4;
}else if (SecLevel == CCM_4){
        mode = CC2420_CCM;
        micLength = MICLENGTH4;
}else if (SecLevel == CCM_8){
        mode = CC2420_CCM;
        micLength = MICLENGTH8;
}else if (SecLevel == CCM_16){
        mode = CC2420_CCM;
        micLength = MICLENGTH16;
}else{
        //something is wrong here
        return;
}
key = KEYREG0; // Using the KEYO RAM location for key
start_CipAuth = AddLen + SecLen;
key = KEYREG0; //Position of the Key
call CC2420Keys.setKey(key,mykey);
if (SecLevel == CBC_MAC_4 || SecLevel == CBC_MAC_8 || SecLevel == CBC_MAC_16){
      nonceValue[FLAGS_NONCE] |= (micLength -2 ) / 2 << CBCMAC_NONCE_FLAGS;
      nonceValue[FLAGS_NONCE] |= 1 << CBCADATA_NONCE_FLAGS;
}
else {
        nonceValue[FLAGS_NONCE] |= 0 << CBCMAC_NONCE_FLAGS;
        nonceValue[FLAGS_NONCE] |= 0 << CBCADATA_NONCE_FLAGS;
}
*((nxle_uint64_t*) (&(nonceValue[SOURCE_CLIENT_NONCE]))) =
 srcAddress->extendedAddress;
```

```
nonceValue[FLAGS_NONCE] |= 2 << L_NONCE_FLAGS;
nonceValue[COUNTER_NONCE] = header.mhr[AddLen + 1];
nonceValue[COUNTER_NONCE + 1] = header.mhr[AddLen + 2];
nonceValue[COUNTER_NONCE + 2] = header.mhr[AddLen + 3];
nonceValue[COUNTER_NONCE + 3] = header.mhr[AddLen + 4];
nonceValue[KSC_NONCE] = SecLevel;
nonceValue[BLOCK_COUNTER_NONCE] = 0;
nonceValue[BLOCK_COUNTER_NONCE + 1] = 1;
call CSN.clr();
atomic call RXNONCE.write(RAM_START, nonceValue, BITDIM16);
call CSN.set();
 if(mode == CC2420_CBC_MAC || mode == CC2420_CCM){

        CTR_SECCTRL0 = ((mode << CC2420_SECCTRL0_SEC_MODE) |
                        ((micLength-2)/2 << CC2420_SECCTRL0_SEC_M) |
                        (key << CC2420_SECCTRL0_SEC_RXKEYSEL) |
                        (RESET_BIT1 << CC2420_SECCTRL0_SEC_CBC_HEAD) |
                        (RESET_BIT1 << CC2420_SECCTRL0_RXFIFO_PROTECTION)) ;
        call CSN.clr();
        atomic call SECCTRL0.write(CTR_SECCTRL0);
        call CSN.set();
}else if (mode == CC2420_CTR )
{

         CTR_SECCTRL0 =  ((mode << CC2420_SECCTRL0_SEC_MODE) |
                        (RESET_BIT1 << CC2420_SECCTRL0_SEC_M) |
                        (key << CC2420_SECCTRL0_SEC_RXKEYSEL) |
                        (RESET_BIT1 << CC2420_SECCTRL0_SEC_CBC_HEAD) |
                        (RESET_BIT1 << CC2420_SECCTRL0_RXFIFO_PROTECTION)) ;
        call CSN.clr();
        atomic call SECCTRL0.write(CTR_SECCTRL0);
                calll CSN.set();
}

CTR_SECCTRL1 = (start_CipAuth << CC2420_SECCTRL1_SEC_RXL);
call CSN.clr();
atomic call SECCTRL1.write(CTR_SECCTRL1);
call CSN.set();
call CSN.clr();
atomic call SRXDEC.strobe();
call CSN.set();
packetLength = header.length + 1;
call CSN.clr();
status = call SNOP.strobe();
call CSN.set();
while(status & CC2420_STATUS_ENC_BUSY){
        uwait(1*1024);
        call CSN.clr();
        status = call SNOP.strobe();
159             call CSN.set();
160             }
161   call CSN.clr();
162   atomic call SECCTRL0.write((RESET_BIT0 << CC2420_SECCTRL0_SEC_MODE) |
163                              (RESET_BIT0 << CC2420_SECCTRL0_SEC_M) |
164                              (RESET_BIT0 << CC2420_SECCTRL0_SEC_RXKEYSEL) |
165                              (RESET_BIT1 << CC2420_SECCTRL0_SEC_CBC_HEAD) |
166                              (RESET_BIT1 << CC2420_SECCTRL0_RXFIFO_PROTECTION)) ;
167   call CSN.set();
168   SECURITYLOCK = 0;
169   return;
170   }
171
172   }
```

Dec() is very similar as *securitycheck()*:after locking the registers (lines 16 - 19), we read the security information with RAM access (lines 23 - 83); we set working mode (lines 86 - 111), decryption key (line 120), nonce (lines 121 - 144) and security control registers (lines 147 - 179). Finally *SRXDEC.strobe( )* performs decryption and, when security module is idle, SECCTRL0 is cleaned.

4.5 Test Application

Our scenario is made by two motes: a RFD that sends ciphered packets and a FFD that receives ciphered packets and deciphers them; RFD sends one packet for beacon received. Working mode is CTR because it is the easiest to verify and the key is static. In figure 4.6 can we see a simplification the communication paradigm.

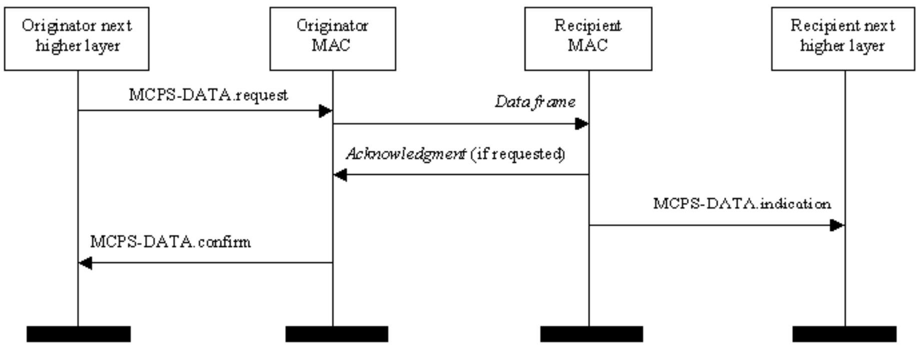


Figure 4.6: Data Exchange

4.5.1 Sender Application

This is the sender application.

```
#include "TKN154.h"
#include "app_profile.h"
//#include "printf.h"
module TestDeviceSenderC
{
  uses {
    interface Boot;
    interface MCPS_DATA;
    interface MLME_RESET;
    interface MLME_SET;
    interface MLME_GET;
    interface MLME_SCAN;
    interface MLME_SYNC;
    interface MLME_BEACON_NOTIFY;
    interface MLME_SYNC_LOSS;
    interface IEEE154Frame as Frame;
    interface IEEE154BeaconFrame as BeaconFrame;
    interface Leds;
    interface Packet;
  }
} implementation {

  message_t m_frame;
  uint8_t m_payloadLen;
  uint8_t myVar;
  ieee154_security_t mysec;
  ieee154_PANDescriptor_t m_PANDescriptor;
  bool m_ledCount;
  bool m_isPANDescriptorValid;
  bool m_sending;
  uint8_t u = 0;
  uint8_t d = 0;


  void startApp();
  task void packetSendTask();

  event void Boot.booted() {
    char payload[] = "00␣Packet␣sent␣from␣Device";



    uint8_t *payloadRegion;
    m_payloadLen = strlen(payload);
    payloadRegion = call Packet.getPayload(&m_frame, m_payloadLen);
    if (m_payloadLen <= call Packet.maxPayloadLength()){
      memcpy(payloadRegion, payload, m_payloadLen);
      call MLME_RESET.request(TRUE, BEACON_ENABLED_PAN);
    }
  }

  event void MLME_RESET.confirm(ieee154_status_t status)
  {
    if (status == IEEE154_SUCCESS)
      startApp();
  }

  void startApp()
  {
    ieee154_phyChannelsSupported_t channelMask;
    uint8_t scanDuration = BEACON_ORDER;

    m_isPANDescriptorValid = FALSE;
    call MLME_SET.macShortAddress(TOS_NODE_ID);
    channelMask = ((uint32_t) 1) << RADIO_CHANNEL;
    call MLME_SET.macAutoRequest(FALSE);
    mysec.SecurityLevel=CTR;
    mysec.KeyIdMode = 0;
   call MLME_SCAN.request  (
                            PASSIVE_SCAN,           // ScanType
                            channelMask,            // ScanChannels
                            scanDuration,           // ScanDuration
                            0x00,                   // ChannelPage
                            0,                      // EnergyDetectListNumEntries
                            NULL,                   // EnergyDetectList
                            0,                      // PANDescriptorListNumEntries
                            NULL,                   // PANDescriptorList
                            NULL             // security
```

```
                              );

}

event message_t* MLME_BEACON_NOTIFY.indication (message_t* frame)
{

       ieee154_phyCurrentPage_t page =  call MLME_GET.phyCurrentPage();
       ieee154_macBSN_t beaconSequenceNumber =  call BeaconFrame.getBSN(frame);
  if (beaconSequenceNumber & 1)
    call Leds.led2On();
  else
    call Leds.led2Off();
  if (!m_isPANDescriptorValid && call BeaconFrame.parsePANDescriptor(
        frame, RADIO_CHANNEL, page, &m_PANDescriptor) == SUCCESS){
    if (m_PANDescriptor.CoordAddrMode == ADDR_MODE_SHORT_ADDRESS &&
        m_PANDescriptor.CoordPANId == PAN_ID &&
        m_PANDescriptor.CoordAddress.shortAddress == COORDINATOR_ADDRESS){
        m_isPANDescriptorValid = TRUE;


    }
  }
  return frame;
}

event void MLME_SCAN.confirm    (
                              ieee154_status_t status,
                              uint8_t ScanType,
                              uint8_t ChannelPage,
                              uint32_t UnscannedChannels,
                              uint8_t EnergyDetectListNumEntries,
                              int8_t* EnergyDetectList,
                              uint8_t PANDescriptorListNumEntries,
                              ieee154_PANDescriptor_t* PANDescriptorList
                            )
{

  if (m_isPANDescriptorValid){


    call MLME_SET.macCoordShortAddress(m_PANDescriptor.CoordAddress.shortAddress);
    call MLME_SET.macPANId(m_PANDescriptor.CoordPANId);
    call MLME_SYNC.request(m_PANDescriptor.LogicalChannel,
                                                  m_PANDescriptor.ChannelPage, FALSE);
    call Frame.setAddressingFields(
        &m_frame,
        ADDR_MODE_SHORT_ADDRESS,         // SrcAddrMode,
        ADDR_MODE_SHORT_ADDRESS,         // DstAddrMode,
              m_PANDescriptor.CoordPANId,        // DstPANId,
              &m_PANDescriptor.CoordAddress,   // DstAddr,
              &mysec                           // security
              );
            post packetSendTask();
        } else {
          startApp();
}
 }

    task void packetSendTask()
    {
            if ( u + 1 > 9) {
                    m_frame.data[1] ='0';
                    u = 0;
                    if ( d + 1 > 9)   {
                            m_frame.data[0] = '0';
                            d = 0;
                    }else{
                            m_frame.data[0]++;
                            d++;}
            }else{
                    m_frame.data[1]++;
                    u++;
            }


            printf("My payload sent  %s\n",(char*) m_frame->data);
            printfflush();
        if (!m_sending && m_isPANDescriptorValid &&
            call MCPS_DATA.request  (
              &m_frame,                              // frame,
              m_payloadLen,                          // payloadLength,
              0,                                     // msduHandle,
              TX_OPTIONS_ACK                         // TxOptions,
              ) == IEEE154_SUCCESS)
          m_sending = TRUE;

    }


    event void MCPS_DATA.confirm     (
                              message_t *msg,
                              uint8_t msduHandle,
                              ieee154_status_t status,
                              uint32_t timestamp
                            )
    {
      m_sending = FALSE;
      if (status == IEEE154_SUCCESS ){
       call Leds.led1Toggle();
     }
    post packetSendTask();
    }

    event void MLME_SYNC_LOSS.indication(
                              ieee154_status_t lossReason,
                              uint16_t PANId,
                              uint8_t LogicalChannel,
                              uint8_t ChannelPage,
                              ieee154_security_t *security)
    {

      startApp();
    }

    event message_t* MCPS_DATA.indication (message_t* frame)
    {
      // we don't expect data
      return frame;
    }

}
```

The sender, after the *MLME_RESET.confirm()*, hears the channel with the *MLME_SCAN.request()* to understand if there are any coordinators. Then when it discovers a beacon, set the parameters in the frame and calls the *MCPS_DATA.request()*. Then when it receives acknowledgment, the *MCPS_DATA.confirm()*, toggles also the green led and ends the sending process.

4.5.2 Receiver application

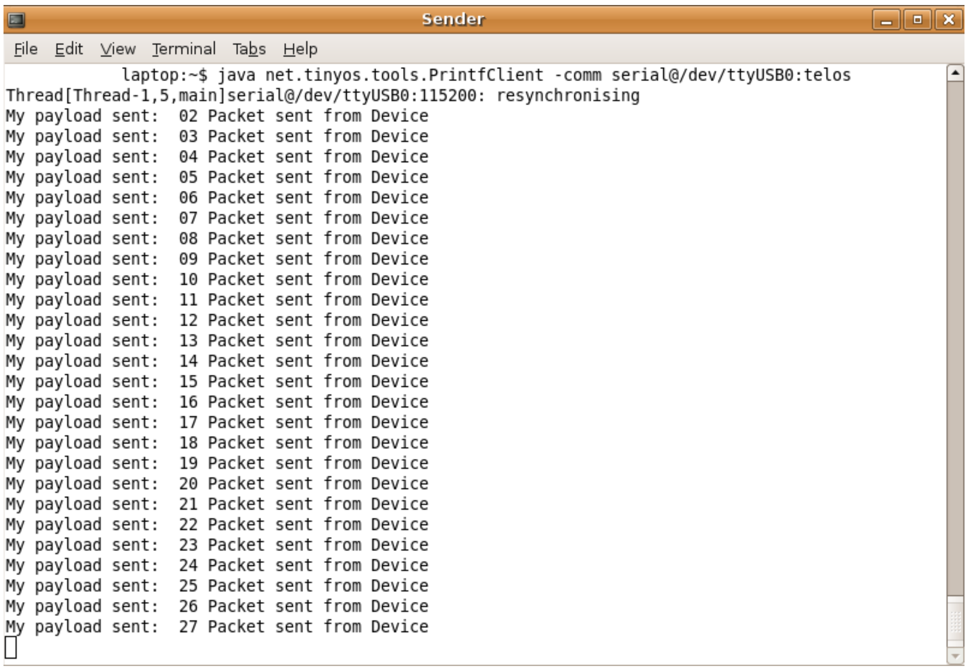
The receiver application is very similar.

```
#include "TKN154.h"
#include "app_profile.h"
#include "printf.h"
module TestCoordReceiverC
{
  uses {
    interface Boot;
    interface MCPS_DATA;
    interface MLME_RESET;
    interface MLME_START;
    interface MLME_SET;
    interface MLME_GET;
    interface IEEE154Frame as Frame;
    interface IEEE154TxBeaconPayload;
    interface Leds;
  }
} implementation {

  bool m_ledCount;
  char mypayload[50];

  event void Boot.booted() {
    call MLME_RESET.request(TRUE, BEACON_ENABLED_PAN);
  }

  event void MLME_RESET.confirm(ieee154_status_t status)
  {
    if (status != IEEE154_SUCCESS)
      return;
    call MLME_SET.macShortAddress(COORDINATOR_ADDRESS);
    call MLME_SET.macAssociationPermit(FALSE);
    call MLME_START.request(
                          PAN_ID,                 // PANId
                          RADIO_CHANNEL,          // LogicalChannel
                          0,                      // ChannelPage,
                          0,                      // StartTime,
                          BEACON_ORDER,           // BeaconOrder
                          SUPERFRAME_ORDER,       // SuperframeOrder
                          TRUE,                   // PANCoordinator
                          FALSE,                  // BatteryLifeExtension
                          FALSE,                  // CoordRealignment
                          0,                      // CoordRealignSecurity,
                          0                       // BeaconSecurity
                        );
  }

  event message_t* MCPS_DATA.indication ( message_t* frame )
  {
        uint8_t tempo=0;
        uint8_t len = ((ieee154_header_t*) frame->header)->length & FRAMECTL_LENGTH_MASK;
        call Leds.led1Toggle();
    for (tempo = 0; tempo  < len; tempo++)
                mypayload[tempo] = (char)frame->data[tempo];
        printf("My␣payload␣received:␣%s\n", mypayload);
        printfflush();

   for (tempo = 0; tempo < 50; tempo ++)
                mypayload[tempo] = 0;


    return frame;
  }

  event void MLME_START.confirm(ieee154_status_t status) {}

  event void MCPS_DATA.confirm(
                          message_t *msg,
                          uint8_t msduHandle,
                          ieee154_status_t status,
                          uint32_t Timestamp
                        ){



}

  event void IEEE154TxBeaconPayload.aboutToTransmit() { }

  event void IEEE154TxBeaconPayload.setBeaconPayloadDone(void *beaconPayload, uint8_t length) { }

  event void IEEE154TxBeaconPayload.modifyBeaconPayloadDone(
                                  uint8_t offset, void *buffer, uint8_t bufferLength) { }

  event void IEEE154TxBeaconPayload.beaconTransmitted()
  {
    ieee154_macBSN_t beaconSequenceNumber = call MLME_GET.macBSN();


    if (beaconSequenceNumber & 1)
      call Leds.led2On();
    else
      call Leds.led2Off();
  }
}
```

After *MLME_reset()*, the coordinator starts to send beacons and listens for incoming transmission. When it finds an incoming frame, starts the receiving procedure and, at the very end, the *MCPS_ data.indication()* is signaled and the application prints the payload of the frame the coordinator receives.

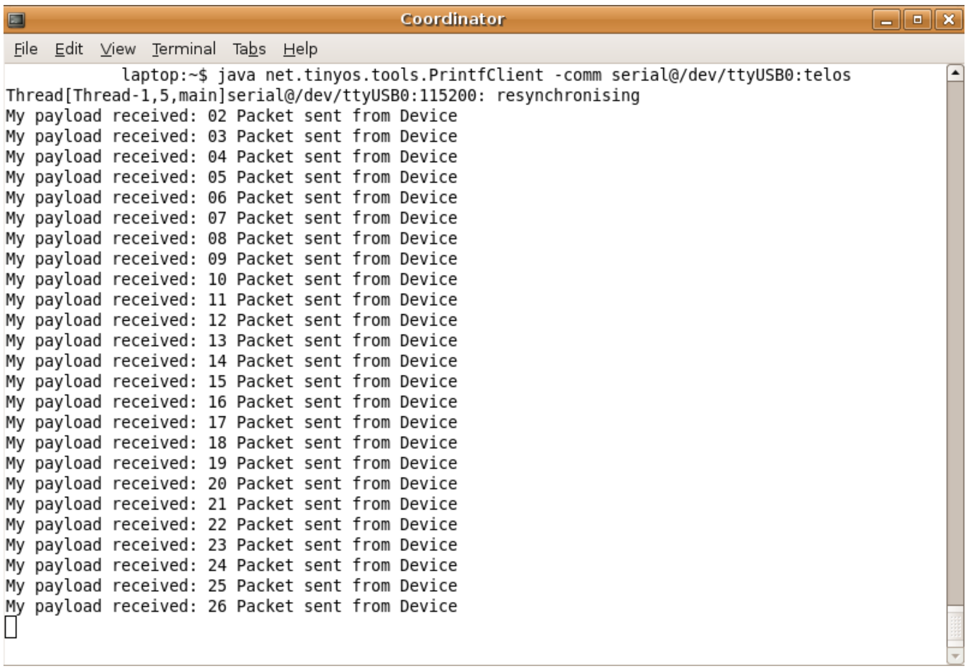
4.5.3 Results

In the figures 4.7 and 4.8 we can see the output of the sender and of the receiver: the payload has a dynamic part, the initial progressive counter that is incremented before single*MCPS_ data.request()* command on the sender, and a fixed part, the remain portion of the payload. Like we see in the picture, with a little delay, the payload that the senders sends to the coordinator, arrives on they receiver and the were correctly deciphered.



```
            laptop:~$ java net.tinyos.tools.PrintfClient -comm serial@/dev/ttyUSB0:telos
Thread[Thread-1,5,main]serial@/dev/ttyUSB0:115200: resynchronising
My payload sent:  02 Packet sent from Device
My payload sent:  03 Packet sent from Device
My payload sent:  04 Packet sent from Device
My payload sent:  05 Packet sent from Device
My payload sent:  06 Packet sent from Device
My payload sent:  07 Packet sent from Device
My payload sent:  08 Packet sent from Device
My payload sent:  09 Packet sent from Device
My payload sent:  10 Packet sent from Device
My payload sent:  11 Packet sent from Device
My payload sent:  12 Packet sent from Device
My payload sent:  13 Packet sent from Device
My payload sent:  14 Packet sent from Device
My payload sent:  15 Packet sent from Device
My payload sent:  16 Packet sent from Device
My payload sent:  17 Packet sent from Device
My payload sent:  18 Packet sent from Device
My payload sent:  19 Packet sent from Device
My payload sent:  20 Packet sent from Device
My payload sent:  21 Packet sent from Device
My payload sent:  22 Packet sent from Device
My payload sent:  23 Packet sent from Device
My payload sent:  24 Packet sent from Device
My payload sent:  25 Packet sent from Device
My payload sent:  26 Packet sent from Device
My payload sent:  27 Packet sent from Device
```

Figure 4.7: Sender Output



```
            laptop:~$ java net.tinyos.tools.PrintfClient -comm serial@/dev/ttyUSB0:telos
Thread[Thread-1,5,main]serial@/dev/ttyUSB0:115200: resynchronising
My payload received: 02 Packet sent from Device
My payload received: 03 Packet sent from Device
My payload received: 04 Packet sent from Device
My payload received: 05 Packet sent from Device
My payload received: 06 Packet sent from Device
My payload received: 07 Packet sent from Device
My payload received: 08 Packet sent from Device
My payload received: 09 Packet sent from Device
My payload received: 10 Packet sent from Device
My payload received: 11 Packet sent from Device
My payload received: 12 Packet sent from Device
My payload received: 13 Packet sent from Device
My payload received: 14 Packet sent from Device
My payload received: 15 Packet sent from Device
My payload received: 16 Packet sent from Device
My payload received: 17 Packet sent from Device
My payload received: 18 Packet sent from Device
My payload received: 19 Packet sent from Device
My payload received: 20 Packet sent from Device
My payload received: 21 Packet sent from Device
My payload received: 22 Packet sent from Device
My payload received: 23 Packet sent from Device
My payload received: 24 Packet sent from Device
My payload received: 25 Packet sent from Device
My payload received: 26 Packet sent from Device
```

Figure 4.8: Receiver Output

Chapter 5

Conclusions

This thesis work started considering IEEE 802.15.4 standard, that describes a radio communication protocol for wireless Personal Area Networks. It also provides a security suite by means of which is possible to create encrypted packets, authenticated packets and encrypted and authenticated packets. This functionalities can be integrated with the capabilities of the CC2420 chipset over TelosB motes.The chipset supports the IEEE 802.15.4 standard and features also a set of options to provide the in-line security operations planned by IEEE 802.15.4. The TelosB motes is IEEE 802.15.4 compliant and supports the TinyOs environment and NesC language.

The objective of this work was to create a secure communication between two motes. However, some simplifications were made during the development process: the communication occurs between a single sender that sends packets to a single coordinator, having care to not overload the network. In addition, the security parameters are statically set (there is not any key retrieval procedure) and only the encryption mode (CTR) was implemented and tested.

The implementation was verified by means of a simple application: the sender transmits an encrypted data frame each time a beacon frame has been received, while the coordinator receives packets, decrypts them and sends acknowledgments to confirm that the packet was correctly received. As shown by the application output, the payload of transmitted and received packets always coincides, proving that encryption/decryption operations work properly.

Further improvements can be added as future work: CBC-MAC and CCM mode can be easily implemented. Besides, full incoming/outgoing frame security procedures can be implemented as well, with incoming/outgoing frame key retrieval procedures, in order to cope with frame reception from multiple senders.

Appendix A

Utility Functions

A.1 writeSecurityMHR

- **Type**: Command
- **Provided by**: Frame
- **Implemented where**: tkn154/PibP.nc
- **Purpose**: writing Auxiliary Security Field when message_t *frame parameter is not available.
- **uint8_t* mhr**: Pointer to MAC Header.
- **uint8_t start:** Starting offset of Auxiliary Security Header.
- **ieee154_security_t *security**: Security values.

```
//       tkn154/PibP.nc
command error_t Frame.writeSecurityMHR( uint8_t* mhr,
                                               uint8_t start,
                                               ieee154_security_t *security)
{
ieee154_macFrameCounter_t macCounter;
ieee154_security_t *temp = security;
uint8_t offset = start;
 if( temp -> SecurityLevel > 7 || temp -> KeyIdMode > 4)
        return FAIL; //wrong values
mhr[offset] = (temp-> SecurityLevel << SEC_CNTL_LEVEL_POS);
mhr[offset] |= 0 << SEC_CNTL_RESERVED_POS;
mhr[offset++] |= (temp ->KeyIdMode << SEC_CNTL_KEYIDMODE_POS);
macCounter = call MLME_GET.macFrameCounter();
if ( macCounter >= 0xffffffff)
        return  IEEE154_COUNTER_ERROR;
*((nx_uint8_t*) (&(mhr[offset]))) = macCounter;
offset += 4;
call MLME_SET.macFrameCounter(++macCounter);
if ( temp->KeyIdMode & SEC_CNTL_KEYIDMODE ) {
    if ((temp->KeyIdMode & SEC_CNTL_KEYIDMODE) == KEYIDMODE1) {
        mhr[offset++] = temp->KeyIndex;
    }
    else if ( (temp->SecurityLevel & SEC_CNTL_KEYIDMODE) == KEYIDMODE2) {
        mhr[offset++] = temp->KeySource[0];
        mhr[offset++] = temp->KeySource[1];
        mhr[offset++] = temp->KeySource[2];
        mhr[offset++] = temp->KeySource[3];
        mhr[offset++] = temp->KeyIndex;
    }
    else {
        mhr[offset++] = temp->KeySource[0];
        mhr[offset++] = temp->KeySource[1];
        mhr[offset++] = temp->KeySource[2];
        mhr[offset++] = temp->KeySource[3];
        mhr[offset++] = temp->KeySource[4];
        mhr[offset++] = temp->KeySource[5];
        mhr[offset++] = temp->KeySource[6];
        mhr[offset++] = temp->KeySource[7];
        mhr[offset++] = temp->KeyIndex;
     }
}

return SUCCESS;
}
```

The commands *setCbcMac()* and *setCCM()*, provided by CC2420SecurityMode interface, have the same code of *writeSecurityMHR()*, excepts of the message_t *frame parameter instead of uint8_t* mhr.

A.2 writeHeader

- **Type**: Command
- **Provided by**: FrameUtility
- **Implemented where**: tkn154/PibP.nc

- **Purpose**: writing Addressing Fields of Header.
- **uint8_t* mhr**: Pointer to MHR.
- **uint8_t DstAddrMode**: Destination Addressing Mode.
- **uint8_t DstPANId**: Destination PAN Identifier.
- **ieee154_address_t* DstAddr**: Destination IEEE extended address.
- **uint8_t SrcAddrMode**: Source Addressing Mode.
- **uint8_t SrcPANId**: Source PAN Identifier.
- **const ieee154_address_t* SrcAddr**: Source IEEE extended address.
- **bool PANIdCompression**: PAN Id compression flag

```
//        tkn154/PibP.nc
async command uint8_t FrameUtility.writeHeader(
        uint8_t* mhr,
        uint8_t DstAddrMode,
        uint16_t DstPANId,
        ieee154_address_t* DstAddr,
        uint8_t SrcAddrMode,
        uint16_t SrcPANId,
        const ieee154_address_t* SrcAddr,
        bool PANIDCompression
)
{
uint8_t offset = MHR_INDEX_ADDRESS;
if (DstAddrMode == ADDR_MODE_SHORT_ADDRESS || DstAddrMode == ADDR_MODE_EXTENDED_ADDRESS){
        *((nxle_uint16_t*) &mhr[offset]) = DstPANId;
        offset += 2;
        if (DstAddrMode == ADDR_MODE_SHORT_ADDRESS){
          *((nxle_uint16_t*) &mhr[offset]) = DstAddr->shortAddress;
          offset += 2;
 } else {
          call FrameUtility.convertToLE(&mhr[offset], &DstAddr->extendedAddress);
          offset += 8;
        }
 }
if (SrcAddrMode == ADDR_MODE_SHORT_ADDRESS || SrcAddrMode == ADDR_MODE_EXTENDED_ADDRESS){
        if (DstPANId != SrcPANId || !PANIDCompression){
          *((nxle_uint16_t*) &mhr[offset]) = SrcPANId;
          offset += 2;
       }
        if (SrcAddrMode == ADDR_MODE_SHORT_ADDRESS){
          *((nxle_uint16_t*) &mhr[offset]) = SrcAddr->shortAddress;
          offset += 2;
        } else {
          call FrameUtility.convertToLE(&mhr[offset], &SrcAddr->extendedAddress);
          offset += 8;
        }
      }
  return offset;
   }
```

A.3 getAddressingFieldsLength

- **Type**: Command
- **Provided by**: FrameUtility
- **Implemented where**: tkn154/PibP.nc
- **Purpose**: Count how long is Addressing Fileds
- **uint8_t fcf1**: First Byte of Frame Control Field.
- **uint8_t fcf2**: Second Byte of Frame Control Field.
- **uint8_t *len**: Length of Header (except of Auxiliary Security Header, if present)

```
//       tkn154/PibP.nc
async command error_t FrameUtility.getAddressingFieldsLength(uint8_t fcf1,
uint8_t fcf2, uint8_t *len)
{
uint8_t idCompression;
 uint8_t offset = MHR_INDEX_ADDRESS;
 idCompression = (fcf1 & FC1_PAN_ID_COMPRESSION);
if (fcf2 & 0x08){
         offset += 4;
if (fcf2 & 0x04)
        offset += 6;
    }
if (fcf2 & 0x80){
        offset += 2;
        if (!idCompression)
                       offset += 2;
        if (fcf2 & 0x40)
                offset += 6;
    }
*len = offset;
return SUCCESS;
  }
```

A.4 getSecurityHeaderLength

- **Provided by**: FrameUtility
- **Implemented where**: tkn154/PibP.nc
- **Purpose**: Count how long is Auxiliary Security Header
- **uint8_t fcf1**: First Byte of Frame Control Field.
- **uint8_t SecurityControl**: SecurityControl Field.
- **uint8_t *len**: Length of Auxiliary Security Header.

```
//        tkn154/PibP.nc
async command error_t FrameUtility.getSecurityHeaderLength(uint8_t fcf1,
uint8_t SecurityControl, uint8_t *len)
  {
uint8_t offset=0;
if (fcf1 & FC1_SECURITY_ENABLED) {
          offset+=5;
          if (SecurityControl & AUX_KEYID) {
                if ((SecurityControl & AUX_KEYID) == AUX_KEYID1) {
                      offset +=1;
                      }
                else if ( (SecurityControl & AUX_KEYID) == AUX_KEYID2) {
                       offset +=5;
                      }
                else {
                       offset+=9;
                }
        }

}
*len = offset;
   return SUCCESS;
}
```

A.5 nonce_building_function

- **Provided by**: —

- **Implemented where**: chips/cc2420_tkn154/CC2420TransmitP
- **Purpose**: Fill the TXNONCE with correct values.
- **uint8_t stat**: Offset where Auxiliary Security Header start.
- **uint8_t Auth**: Authentication option.

```
//          chips/cc2420\_tkn154/CC2420TransmitP
void nonce_building_function(uint8_t start, uint8_t auth)
{
uint8_t SecLevel = m_frame->header->mhr[start];
ieee154_address_t srcAddress;
if (SecLevel == CBC_MAC_4 || SecLevel == CBC_MAC_8 || SecLevel == CBC_MAC_16){
        nonceValue[FLAGS_NONCE] |= auth << CBCMAC_NONCE_FLAGS;
        nonceValue[FLAGS_NONCE] |= 1 << CBCADATA_NONCE_FLAGS;
        }
else {
        nonceValue[FLAGS_NONCE] |= 0 << CBCMAC_NONCE_FLAGS;
        nonceValue[FLAGS_NONCE] |= 0 << CBCADATA_NONCE_FLAGS;
}
srcAddress.extendedAddress = call GetLocalExtendedAddress.get();
*((nxle_uint64_t*) (&(nonceValue[SOURCE_CLIENT_NONCE])))=srcAddress.extendedAddress;
nonceValue[FLAGS_NONCE] |= 2 << L_NONCE_FLAGS;
nonceValue[COUNTER_NONCE] = m_frame->header->mhr[start + 1];
nonceValue[COUNTER_NONCE + 1] = m_frame->header->mhr[start + 2];
nonceValue[COUNTER_NONCE + 2] = m_frame->header->mhr[start + 3];
nonceValue[COUNTER_NONCE + 3] = m_frame->header->mhr[start + 4];
nonceValue[KSC_NONCE] = SecLevel;
nonceValue[BLOCK_COUNTER_NONCE] = 0;
nonceValue[BLOCK_COUNTER_NONCE + 1] = 1;
}
```

A.6 setKey

- **Provided by**: CC2420Keys
- **Implemented where**: chips/cc2420_tkn154/CC2420ControlP
- **Purpose**: Set the Key in one Key Register of CC2420
- **uint8_t keyNo**: Index of the register
- **uint8_t* key**: Key.

```
//          chips/cc2420\_tkn154/CC2420ControlP
async command error_t CC2420Keys.setKey(uint8_t keyNo, uint8_t* key)
{
uint8_t *currentKey = key;
bool currentKeyNo = keyNo;
if (currentKey == NULL || keyNo > 1) {
        return FAIL;
 }
if (currentKeyNo) {
        call CSN.clr();
        call KEY1.write(0, currentKey, 16);
        call CSN.set();
} else {
        call CSN.clr();
        call KEY0.write(0, currentKey, 16);
        call CSN.set();
    }
        signal CC2420Keys.setKeyDone(currentKeyNo, currentKey);
        return SUCCESS;
}
```

Appendix B

Security Structures

B.1 Types Definitions

```
//        tkn154/TKN154.h
#define MAX_MAC_KEY_TABLE_ENTRIES                 16
#define MAX_MAC_DEVICE_TABLE_ENTRIES              16
#define MAX_MAC_SECURITY_LEVEL_TABLE_ENTRIES      16
#define MAX_MAC_KEY_ID_LOOKUP_LIST_ENTRIES        16
#define MAX_MAC_DEVICE_LIST_ENTRIES               16
#define MAX_MAC_KEY_USAGE_LIST_ENTRIES            16
#define MAX_DEVICE_TABLE_ENTRIES                  16
#define MAX_MAC_SECURITY_TABLE_ENTRIES            16
#define MAX_OCTETS                                 8

typedef uint8_t       ieee154_macKeyTableEntries_t;
typedef uint8_t       ieee154_macDeviceTableEntries_t;
typedef uint8_t       ieee154_macSecurityLevelTableEntries_t;
typedef uint8_t       ieee154_macAutoRequestSecurityLevel_t;
typedef uint8_t       ieee154_macAutoRequestKeyIdMode_t;
typedef uint8_t       ieee154_macAutoRequestKeyIndex_t;
typedef uint64_t      ieee154_macPANCoordExtendedAddress_t;
typedef uint16_t      ieee154_macPANCoordShortAddress_t;
typedef uint8_t       ieee154_KeyIdLookupListEntries_t;
typedef uint8_t       ieee154_KeyDeviceListEntries_t;
typedef uint8_t       ieee154_KeyUsageListEntries_t;
typedef uint16_t*     ieee154_Key_t;
typedef uint8_t       ieee154_FrameType_t;
typedef uint8_t       ieee154_CommandFrameIdentifier_t;
typedef uint8_t       ieee154_DeviceDescriptorHandle_t;
typedef bool          ieee154_UniqueDevice_t;
typedef bool          ieee154_Blacklisted_t;
typedef uint8_t       ieee154_SecurityMinimum_t;
typedef uint8_t          ieee154_DeviceOverrideSecurityMinimum_t;
typedef uint64_t         ieee154_ExtAddress_t;
typedef bool             ieee154_Exempt_t;
typedef uint8_t          ieee154_LookUpDataSize_t;
typedef uint8_t*         ieee154_LookUpData_t;
typedef uint32_          ieee154_macFrameCounter_t;
typedef uint8_t          ieee154_macAutoRequestKeySource_t;
typedef uint8_t          ieee154_macDefaultKeySource_t;

typedef struct ieee154_KeyUsageDescriptor_t {
          ieee154_FrameType_t *frametype;
          ieee154_CommandFrameIdentifier_t commandframeidentifier;
} ieee154_KeyUsageDescriptor_t;

typedef struct ieee154_KeyIdLookupDescriptor_t {
        ieee154_LookUpDataSize_t lookupdatasize;
        ieee154_LookUpData_t lookupdata;
} ieee154_KeyIdLookupDescriptor_t;

typedef struct ieee154_KeyDeviceDescriptor_t {
        uint8_t devicedescriptorhandle;
        ieee154_UniqueDevice_t uniquedevice;
        ieee154_Blacklisted_t  blackListed;
} ieee154_KeyDeviceDescriptor_t;

typedef struct   ieee154_KeyDescriptor_t {
        ieee154_KeyIdLookupDescriptor_t keyidlookupdescriptor
                                              [MAX_MAC_KEY_ID_LOOKUP_LIST_ENTRIES];
        ieee154_KeyDeviceDescriptor_t keydevicelist  [MAX_MAC_DEVICE_LIST_ENTRIES];
        ieee154_KeyUsageDescriptor_t keyusagelist[MAX_MAC_KEY_USAGE_LIST_ENTRIES];
        ieee154_Key_t key;
}   ieee154_KeyDescriptor_t;

typedef struct  ieee154_macKeyTable_t {
        ieee154_KeyDescriptor_t keydescriptor[MAX_MAC_KEY_TABLE_ENTRIES];
        bool valid[MAX_MAC_KEY_TABLE_ENTRIES];
} ieee154_macKeyTable_t;

typedef struct {
        ieee154_macPANId_t panid;
        ieee154_address_t address;
        ieee154_macFrameCounter_t framecounter;
        bool Exempt;
} ieee154_DeviceDescriptor_t;

typedef struct {
        ieee154_DeviceDescriptor_t devicedescriptor[MAX_MAC_DEVICE_TABLE_ENTRIES];
        bool valid[MAX_MAC_DEVICE_TABLE_ENTRIES];
} ieee154_macDeviceTable_t;
```

```

typedef struct {
        ieee154_FrameType_t frametype;
        ieee154_CommandFrameIdentifier_t commandframeidentifier;
        ieee154_SecurityMinimum_t securityminimum;
        ieee154_DeviceOverrideSecurityMinimum_t deviceoverridesecurity;
} ieee154_SecurityDescriptor_t;

typedef struct {
        ieee154_SecurityDescriptor_t securitydescriptor[MAX_MAC_SECURITY_TABLE_ENTRIES];
        bool valid[MAX_MAC_SECURITY_TABLE_ENTRIES];
} ieee154_macSecurityLevelTable_t;
```

```
//      tkn154/TKN154_MAC.h
        IEEE154_macKeyTable                        = 0x71,
        IEEE154_macKeyTableEntries                 = 0x72,
        IEEE154_macDeviceTable                     = 0x73,
        IEEE154_macDeviceTableEntries              = 0x74,
        IEEE154_macSecurityLevelTable              = 0x75,
        IEEE154_macSecurityLevelTableEntries       = 0x76,
        IEEE154_macFrameCounter                    = 0x77,
        IEEE154_macAutoRequestSecurityLevel        = 0x78,
        IEEE154_macAutoRequestKeyIdMode            = 0x79,
        IEEE154_macAutoRequestKeySource            = 0x7a,
        IEEE154_macAutoRequestKeyIndex             = 0x7b,
        IEEE154_macDefaultKeySource                = 0x7c,
        IEEE154_macPANCoordExtendedAddress         = 0x7d,
        IEEE154_macPANCoordShortAddress            = 0x7e,
```

```
//      tkn154/TKN154_PIB.h
        ieee154_macKeyTable_t macKeyTable;
        //0x72
        ieee154_macKeyTableEntries_t macKeyTableEntries;
        //0x73
        ieee154_macDeviceTable_t macDeviceTable;
        //0x74
        ieee154_macDeviceTableEntries_t macDeviceTableEntries;
        //0x75
        ieee154_macSecurityLevelTable_t macSecurityLevelTable;
        //0x76
        ieee154_macSecurityLevelTableEntries_t macSecurityLevelTableEntries;
        //0x77
        ieee154_macFrameCounter_t macFrameCounter;
        //0X78
        ieee154_macAutoRequestSecurityLevel_t macAutoRequestSecurityLevel;
        //0x79
        ieee154_macAutoRequestKeyIdMode_t macAutoRequestKeyIdMode;
        //0x7a
        ieee154_macAutoRequestKeySource_t macAutoRequestKeySource[8];
        //0x7b
        ieee154_macAutoRequestKeyIndex_t macAutoRequestKeyIndex;
        //0x7c
        ieee154_macDefaultKeySource_t macDefaultKeySource[8];
        //0x7d
        ieee154_macPANCoordExtendedAddress_t macPANCoordExtendedAddress;
        //0x7e
        ieee154_macPANCoordShortAddress_t macPANCoordShortAddress;



#ifndef         IEEE154_DEFAULT_MACKEYTABLEENTRIES
  #define       IEEE154_DEFAULT_MACKEYTABLEENTRIES      0
#endif


#ifndef         IEEE154_DEFAULT_MACKEYTABLE
  #define       IEEE154_DEFAULT_MACKEYTABLE             FALSE
#endif

#ifndef         IEEE154_DEFAULT_MACDEVICETABLEENTRIES
        #define IEEE154_DEFAULT_MACDEVICETABLEENTRIES   0
#endif

#ifndef         IEEE154_DEFAULT_MACDEVICETABLE
        #define IEEE154_DEFAULT_MACDEVICETABLE          FALSE
#endif

#ifndef         IEEE154_DEFAULT_MACSECURITYLEVELTABLEENTRIES
        #define IEEE154_DEFAULT_MACSECURITYLEVELTABLEENTRIES    0
#endif

#ifndef         IEEE154_DEFAULT_MACSECURITYLEVELTABLE
        #define IEEE154_DEFAULT_MACSECURITYLEVELTABLE   FALSE
#endif

#ifndef         IEEE154_DEFAULT_MACFRAMECOUNTER
        #define IEEE154_DEFAULT_MACFRAMECOUNTER         0x00000000
#endif

#ifndef         IEEE154_DEFAULT_MACAUTOREQUESTSECURITYLEVEL
        #define IEEE154_DEFAULT_MACAUTOREQUESTSECURITYLEVEL     0x06
#endif

#ifndef         IEEE154_DEFAULT_MACAUTOREQUESTKEYIDMODE
        #define IEEE154_DEFAULT_MACAUTOREQUESTKEYIDMODE         0X00
#endif

#ifndef         IEEE154_DEFAULT_MACAUTOREQUESTKEYSOURCE
```

```
        #define IEEE154_DEFAULT_MACAUTOREQUESTKEYSOURCE          0xff
#endif


#ifndef         IEEE154_DEFAULT_MACAUTOREQUESTKEYINDEX
        #define IEEE154_DEFAULT_MACAUTOREQUESTKEYINDEX           0xff
#endif

#ifndef         IEEE154_DEFAULT_MACDEFAULTKEYSOURCE
        #define IEEE154_DEFAULT_MACDEFAULTKEYSOURCE              0xff
#endif


#ifndef         IEEE154_DEFAULT_MACPANCOORDSHORTADDRESS
        #define IEEE154_DEFAULT_MACPANCOORDSHORTADDRESS          0x0000
#endif
```

Bibliography

[1] Wikipedia. Ieee 802.15.4-2006. http://en.wikipedia.org/wiki/802.15.4.

[2] I. of Electrical and I. Electronics Engineers. *EEE Std. 802.15.4-2006, IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Speci?c requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs).* Technical report, Institute of Electrical and Electronics Engineers, Inc., New York, September 2006

.

[3] R. Housley. Counter with cbc-mac (ccm). Technical report, RSA Laboratories, 918 Spring Knoll Drive Herndon, VA 20170, USA, 2002

.

[4] IETF. Rfc3610. Technical report, Internet engineering task force, 2003

.

[5] Chipcom. Cc2420 preliminary datasheet. Technical report, Chipcom - SmartRF, 2004

.