MINIMAL IMPLEMENTATION OF A SECURE REMOTE KEYLESS ENCRYPTION
PROTOCOL USING CMAC MODE OF AES

A Thesis
Presented to
The Graduate Faculty of The University of Akron

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

Dinesh Reddy Gade
May, 2010
MINIMAL IMPLEMENTATION OF A SECURE REMOTE KEYLESS ENCRYPTION
PROTOCOL USING CMAC MODE OF AES

Dinesh Reddy Gade

Thesis

Approved:  
Advisor  
Dr. Nathan Ida

Accepted:  
Dean of the College 
Dr. George K. Haritos

Committee Member 
Dr. Shiva Sastry

Dean of the Graduate School 
Dr. George R. Newkome

Committee Member  
Dr. George C. Giakos

Department Chair 
Dr. Alex De Abreu Garcia
Remote keyless entry systems gained prominence with the advancement of wireless communication technological applications. Early remote keyless entry systems operate by transmitting a unique identification key from the transmitter to the receiver. Security has become a major concern for these systems due to cloning of the key. Modern keyless entry systems make use of advanced theft control methods that generate rolling codes. The main purpose of this research was to implement a secure wireless encryption protocol that generates rolling codes using cryptography. This implementation makes use of the Advanced Encryption Standard (AES) and its Cipher-based Message Authentication Code (CMAC) mode of operation for transmitter authentication. A minimal implementation of this protocol is done for use in smaller devices. A simple and secure learning method was used for introducing new transmitters to the receiver.

The transmitter and receiver are programmed using PIC24 microcontrollers and C30 compiler. Two 16-bit microprocessors are used to implement the hardware. The microcontroller devices are programmed for minimal power consumption. A working implementation of the transmitter and receiver programs written in C language is included in the appendix.
ACKNOWLEDGEMENT

I would like to specially thank my advisor Dr. Ida for giving me an opportunity to work under his guidance and for providing consistent support and advice throughout my research work. I would like to thank my thesis committee members Dr. Nathan Ida, Dr. Shiva Sastry and Dr. George C. Giakos for their guidance and support throughout my Master’s program.

I would like to thank my department chair Dr. Alex De Abreu Garcia for providing me Teaching assistantship. Thanks to Dr. Grover for his suggestions regarding the PIC24F Programming. Thanks to Dr. Gerald W. Young, Dr. Joan Carletta, Mr. Mike Ward and Mrs. Gay J. Boden for their help during tough times. I take this opportunity to thank all teaching and non-teaching staff of Electrical and Computer Engineering department for their support.

My heartfelt regards to my family and friends for their consistent support and advice. I would like to thank my friends, Suraj Adhikari and Dileep Reddy Desai in this regard.
TABLE OF CONTENTS

Page

LIST OF TABLES .................................................................................................................. x

LIST OF FIGURES .................................................................................................................. xi

CHAPTER

I. INTRODUCTION ......................................................................................................................... 1

1.1 Overview of Remote Keyless System ....................................................................................... 1

1.2 Early Keyless Entry Systems .................................................................................................. 1

1.3 Modern Keyless Entry Systems ............................................................................................. 2

1.4 Cryptography ........................................................................................................................ 2

1.4.1 Secret Key Cryptography (SKC) ...................................................................................... 3

1.4.2 Public Key Cryptography (PKC) ..................................................................................... 3

1.4.3 Hash Functions ................................................................................................................ 4

1.5 System Goals ....................................................................................................................... 4

1.6 Message Contents ................................................................................................................ 5

1.6.1 Basic System with no Security ....................................................................................... 5

1.6.2 Upgraded System .......................................................................................................... 6

1.6.3 Secure System ............................................................................................................. 7

1.7 Rolling Window .................................................................................................................... 8
1.6 Thesis Outline ......................................................................................................................... 8

II. ADVANCED ENCRYPTION STANDARDS (AES) ................................................................. 10

2.1 AES ........................................................................................................................................ 10

2.2 AES Algorithm Definition ..................................................................................................... 10

2.2.1 SubBytes ( ) Transformation ............................................................................................ 12

2.2.2 InvSubBytes ( ) Transformation ......................................................................................... 13

2.2.3 ShiftRows ( ) Transformation .......................................................................................... 14

2.2.4 InvShiftRows ( ) Transformation ....................................................................................... 14

2.2.5 MixColumns ( ) Transformation ....................................................................................... 15

2.2.6 InvMixColumns ( ) Transformation ................................................................................... 16

2.2.7 AddRoundKey ( ) Transformation .................................................................................... 17

2.2.8 InvAddRoundkey ( ) Transformation ............................................................................... 17

2.3 Key Expansion ....................................................................................................................... 18

2.4 AES Encryption Flowchart .................................................................................................... 20

2.5 AES Decryption Flowchart .................................................................................................. 21

2.6 Conclusions ........................................................................................................................... 22

III. CIPHER BASED MESSAGE AUTHENTICATION CODE (CMAC) ............................... 23

3.1 CMAC ..................................................................................................................................... 23

3.2 Preliminary Symbols, Operations and Functions ............................................................... 23

3.2.1 $0'$ ....................................................................................................................................... 24

3.2.2 $X || Y$ ............................................................................................................................... 24

3.2.3 $X \oplus Y$ ........................................................................................................................... 24

3.2.4 $LSB_s (X)$ ...................................................................................................................... 24
3.2.5 $\text{MSB}_s(X)$ ..........................................................24

3.3 Block Cipher .............................................................................25

3.4 Subkeys .......................................................................................25

3.5 MAC Generation and Verification .............................................25

3.6 Input and Output for the MAC generation ...............................26

3.7 CMAC Specification .................................................................26
  3.7.1 Subkey Generation ..............................................................26
  3.7.2 MAC Generation ...............................................................27
  3.7.3 MAC Verification ...............................................................29

3.8 Conclusions ..............................................................................30

IV. MESSAGE FORMAT AND COMMUNICATION FLOWCHARTS ........31

  4.1 Message Format .................................................................31
    4.1.1 Serial Number ..............................................................31
    4.1.2 Counter Value ..............................................................32
    4.1.3 Command .................................................................32
    4.1.4 MAC ...........................................................................32

  4.2 Transmission format ............................................................32
    4.2.1 Manchester Encoding ..................................................33

  4.3 Encryption and Decryption .....................................................34

  4.4 Block Diagrams of Transmitter Side and Receiver Side ..........34
    4.4.1 Transmitter Side ..........................................................34
    4.4.2 Receiver Side .............................................................35
    4.4.3 EEPROM Record .........................................................35
4.5 Flowchart for Transmitter Session ..............................................................36
4.6 Flowchart for Receiver Session .................................................................38
  4.6.1 Normal Mode of Operation ...............................................................38
  4.6.2 Learn Mode of Operation .................................................................41
4.7 Learn Session ...........................................................................................41
4.8 Hardware and Software Development Tools .............................................43
  4.8.1 MPLAB IDE .......................................................................................43
  4.8.2 MPLAB C Compiler for PIC24 MCUs .................................................43
  4.8.3 MPLAB ICD 2 ...................................................................................44
  4.8.4 SLEEP Mode of Operation .................................................................45
4.9 Conclusions ..............................................................................................45

V. SYSTEM IMPLEMENTATION ....................................................................47

VI. CONCLUSIONS AND FUTURE WORK ................................................49

REFERENCES ...............................................................................................50

APPENDICES .................................................................................................52

APPENDIX A. MAIN PROGRAM FOR TRANSMITTER .............................53
APPENDIX B. AES PROGRAM FOR TRANSMITTER .................................57
APPENDIX C. CMAC PROGRAM FOR TRANSMITTER .............................63
APPENDIX D. TRANSMISSION PROGRAM FOR TRANSMITTER ............65
APPENDIX E. EEPROM PROGRAM FOR TRANSMITTER .......................67
APPENDIX F. COMMUNICATION PROGRAM FOR TRANSMITTER ........72
APPENDIX G. MESSAGE PROGRAM FOR TRANSMITTER .....................76
APPENDIX H. MAIN PROGRAM FOR RECEIVER ......................................78


**LIST OF TABLES**

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>S-box: Substitution values for the byte xy</td>
</tr>
<tr>
<td>2.2</td>
<td>Inverse S-box: Substitution values for the byte xy</td>
</tr>
<tr>
<td>4.1</td>
<td>EEPROM Record</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Picture depicting an eavesdropper cloning the secret code</td>
</tr>
<tr>
<td>1.2</td>
<td>Basic system with goal 1 satisfied</td>
</tr>
<tr>
<td>1.3</td>
<td>Upgraded system with goal 1, 2 and 3 satisfied</td>
</tr>
<tr>
<td>1.4</td>
<td>Secure System with goal 1, 2, 3 and 4 satisfied</td>
</tr>
<tr>
<td>1.5</td>
<td>Rolling window of acceptance</td>
</tr>
<tr>
<td>2.1</td>
<td>Input, state array and output</td>
</tr>
<tr>
<td>2.2</td>
<td>SubBytes ( ) applies to the S-box to each byte of the state</td>
</tr>
<tr>
<td>2.3</td>
<td>ShiftRows ( ) cyclically shifts the last three rows in the State</td>
</tr>
<tr>
<td>2.4</td>
<td>InverseShiftRows ( ) cyclically shifts the last three rows in the state</td>
</tr>
<tr>
<td>2.5</td>
<td>MixColumns ( ) operates on the state column-by-column</td>
</tr>
<tr>
<td>2.6</td>
<td>AddRoundKey ( ) XORs each column of the state with a word from the key schedule</td>
</tr>
<tr>
<td>2.7</td>
<td>Key expansion flowchart</td>
</tr>
<tr>
<td>2.8</td>
<td>AES encryption flowchart</td>
</tr>
<tr>
<td>2.9</td>
<td>AES decryption flowchart</td>
</tr>
<tr>
<td>3.1</td>
<td>Illustration of the two cases of the MAC generation</td>
</tr>
<tr>
<td>3.2</td>
<td>Generating MAC from the message</td>
</tr>
<tr>
<td>4.1</td>
<td>Message format</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>4.2</td>
<td>Manchester encoding</td>
</tr>
<tr>
<td>4.3</td>
<td>Transmission format</td>
</tr>
<tr>
<td>4.4</td>
<td>Block diagram of transmitter side</td>
</tr>
<tr>
<td>4.5</td>
<td>Block diagram of receiver side</td>
</tr>
<tr>
<td>4.6</td>
<td>Flowchart for transmitter session</td>
</tr>
<tr>
<td>4.7</td>
<td>Flowchart for receiver in normal mode of operation</td>
</tr>
<tr>
<td>4.8</td>
<td>Flowchart for receiver in learn mode of operation</td>
</tr>
<tr>
<td>4.9</td>
<td>Learn session</td>
</tr>
<tr>
<td>4.10</td>
<td>Explorer 16 development board, MPLAB ICD 2 debugger</td>
</tr>
</tbody>
</table>
CHAPTER I
INTRODUCTION

This chapter provides the basic knowledge required to understand this thesis topic. An overview of remote keyless systems is discussed. The problems with the early keyless entry systems are stated along with the methods to improve the security level. The goals of a secure system are mentioned along with the various system improvements implementing these goals. The cryptography technology which is a source for the AES algorithm, used in this thesis has been introduced.

1.1 Overview of Remote Keyless System

A remote keyless system is a system that can allow or deny access to automobiles or premises. Remote keyless systems operate by transmitting radio frequency signals from the transmitter to the receiver. They operate at a particular frequency. For example, 433.92 MHz is used in Europe and 315 MHz is used in North America.

1.2 Early Keyless Entry Systems

Early keyless entry systems used a fixed code for keyless entry. The transmitter transmits a secret code to the receiver. The receiver captures the code transmitted by the transmitter and performs the requested operation upon verification of the secret code. This fixed code method of transmission became unsecure with cloning of the secret code. The secret code can be captured by an eavesdropper and is used for unauthorized access.
Replacing fixed codes with the hopping codes will avoid this problem. A picture depicting an eavesdropper cloning the secret key is shown in the Figure 1.1.

![Figure 1.1 Picture depicting an eavesdropper cloning the secret code [15].](image)

1.3 Modern Keyless Entry Systems

Modern keyless entry systems use theft control methods that generate rolling codes for use in keyless entry systems. The transmitter generates a new code for transmission every time it needs to access the receiver. It should be highly impossible to predict the future codes using the previously transmitted codes. So cryptography is used.

1.4 Cryptography

Cryptography is the method of writing in secret code. The term cryptography is used when information is locked and made unavailable using keys. Correct keys are required to unlock the information. The unencrypted data usually called plaintext is encrypted to ciphertext by the transmitter. The ciphertext is decrypted to the plaintext by the receiver. There are three types of cryptographic schemes in general: secret key cryptography, public key cryptography and hash functions. Each of these schemes is discussed in the following subsections.
1.4.1 Secret Key Cryptography (SKC)

In secret key cryptography, a single key is used for both encryption and decryption. The transmitter uses a single key to encrypt the plaintext to the ciphertext and transmits to the receiver. The receiver has the knowledge of the key used by the transmitter. The receiver uses the same key to decrypt the ciphertext back to plaintext. Secret key cryptography is also called as symmetric encryption because a single key is used by both the transmitter and the receiver.

Secret key cryptography schemes are categorized into stream ciphers and block ciphers. A Stream cipher scheme processes a single bit at a time and uses a feedback mechanism so that the key changes for every transmission. A block cipher scheme uses the same key on each block and encrypts one block of data at a time.

There are several secret key cryptography algorithms that are readily available. Some of them are Data Encryption Standard (DES), Advanced Encryption Standard (AES), Secure and Fast Encryption Routine (SAFER), Seed. AES is one among the popular ones that suits best for the remote keyless applications.

1.4.2 Public Key Cryptography (PKC)

Public Key Cryptography makes use of the mathematical functions that are easy to compute whereas their inverse functions are hard to compute. Public Key Cryptography uses two keys that are mathematically related to each other. One key is referred as public key and the other is referred as private key. The public key is open and the private key is kept secret. The public key is used to encrypt the plaintext and the private key is used to decrypt the ciphertext.
1.4.3 Hash Functions

Hash functions do not use any key for encryption or decryption. A hash value of fixed length is computed using the plaintext. It will be impossible to recover the contents of the plaintext or the length of the plaintext from the hash value. Hash functions are used to encrypt passwords by many operating systems.

1.5 System Goals

The transmission protocol should contain two critical qualities as mentioned in Section 1.3 to make the system secure. They are shown below as referred from [3]:

1. No transmission is ever repeated which prevents a would-be thief from grabbing a message and retransmitting.
2. It is virtually impossible to predict message contents, even if previous messages are known. The receiver ignores all messages that have already been used.

The problem with transmitting the same key for every new transmission is discussed in Section 1.2. In order to prevent this problem any transmitted message should never be repeated. Even by knowing a few previous messages it should be impossible to predict the future messages. The receiver should not accept any message that has already been used. In order to satisfy the qualities mentioned above the system should achieve the following goals as referred from [3]:

1. One receiver should be associated with a limited number of transmitters
2. No two transmitted messages should be equal.
3. The receiver should ignore messages that have already been used.
4. It should be virtually impossible to predict the next message contents from a transmitter, even if previous message contents are known.
There should be a unique set of transmitters for a receiver. Only the transmitters that belong to this set should be able to access the receiver. The second goal prevents an eavesdropper from transmitting the captured message again for authentication. The third goal is necessary in order to ignore the messages that have already been used if retransmitted. The receiver must keep track of the old messages received and should ignore them if retransmitted by any. The fourth goal prevents an eavesdropper from predicting the contents of the next message using the previously grabbed messages. Even though each transmitted message is different, there will be some sequence known by both the transmitter and the receiver. This is because of the unidirectional communication from transmitter to the receiver.

1.6 Message Contents

The message transmitted to the receiver should contain elements of information that satisfy the system goals mentioned in Section 1.5. Messages that contain various elements of information providing different levels of security to the system are discussed in the following subsections. A final system that satisfies all of the goals is specified in Section 1.6.3.

1.6.1 Basic System with no Security

An example of a basic system with no security is shown in Figure 1.2. This system contains three transmitters and one receiver. Each transmitter is assigned a unique serial number. The serial numbers represent the identity of the transmitters. The receiver is associated with three transmitters satisfying the first goal. The receiver stores the ID’s of the transmitters that are learned. Any message received by the transmitters with an identity not in the list of accepted transmitters is ignored.
1.6.2 Upgraded System

In order to satisfy the second goal, the message should be upgraded with a content, sequential counter that increments after every transmission. So even if same command is transmitted again the sequential counter is incremented making the transmitted messages different from each other.

Figure 1.3 Upgraded system with goal 1, 2 and 3 satisfied [3].

The sequential number in the message will be smart enough to satisfy the third goal. If the receiver keeps track of the last used sequential numbers then the messages
that have already been sent which contain the old sequential numbers can be ignored by
the receiver.

1.6.3 Secure System

The upgraded system does not satisfy the fourth goal. By capturing the contents of
a few consecutive messages there is a possible chance to predict the next message. An
element that is dependent on a secret key and is different for each transmitter and is too
large to guess by trial-and-error should be added to the message contents. Such an
element is referred to as a Message Authentication Code (MAC). It is highly impossible
to guess a valid MAC even with the knowledge of the sequential number since the MAC
depends on the secret key, which is kept secret by the encoder and decoder devices. The
system with all the four goals satisfied is shown in Figure 1.4. There are a large variety of
algorithms available for generating the MAC bytes. Advanced Encryption standards
(AES) algorithm is chosen in this thesis.

Figure 1.4 Secure system with goal 1, 2, 3 and 4 satisfied [3].
1.7 Rolling Window

Ignoring the messages that contain old sequential numbers creates a problem if the counter value overflows and wraps back to 0. A rolling window can help to prevent the counter overflow. The counter values that falls within the range of the window are only accepted. The span of the window decides the number of messages that can be skipped from transmission. This rolling window of acceptance scheme prevents replay attacks even if old messages are transmitted. The rolling window of acceptance for counter values is shown in Figure 1.5. If each point on the circumference of the circle is considered as a sequential number with a starting point at zero and incremented in the clockwise direction ending with a maximum permitted value then the slice depicted will contain a list of accepted counter values. The slice moves clockwise for every valid transmission.

![Acceptance window](image)

Figure 1.5 Rolling window of acceptance.

1.8 Thesis Outline

The rest of the thesis is organized as follows. Chapter II describes the AES concepts with all the necessary background required to understand the AES algorithm.
The key expansion, AES encryption and decryption flowcharts are discussed in detail. Chapter III introduces the CMAC. The CMAC algorithm along with the preliminary symbols, operations and functions are discussed. The subkey generation, MAC generation and MAC verification algorithms are stated. The CMAC mode of AES is discussed. Chapter IV discusses the format of the message used in the system protocol. The block diagrams for transmitter and receiver sides are shown. Flowcharts for transmitter and receiver side in normal mode and learn mode of operation are discussed along with the learn session. The hardware and software developmental tools used in the implementation of the system are discussed. Chapter V discusses the functionality of the code with the stated results and conclusions are drawn in Chapter VI.
CHAPTER II
ADVANCED ENCRYPTION STANDARDS (AES)

This chapter gives an introduction to the Advanced Encryption Standards and discusses the AES algorithm. The flowcharts for the AES encryption process and AES decryption process are listed. Functions that operate on the state in the process of AES encryption and decryption are discussed. The key expansion algorithm required to generate the key schedule is discussed.

2.1 Advanced Encryption Standards (AES)

The Advanced Encryption Standards (AES) specifies a Federal Information Processing Standards (FIPS) approved cryptographic algorithm that can be used to protect electronic data [16]. The details of the AES algorithm with necessary flowcharts are specified in the following Sections.

2.2 AES Algorithm

The AES algorithm is a symmetrical block cipher that can encrypt and decrypt information. Encryption converts data to an unintelligible form called cipher text. Decrypting the cipher text converts the data back into its original form, called plaintext [16]. The input for the AES algorithm is a sequence of 128 bits called the plaintext. The AES algorithm encrypts the plaintext to cipher text using a secret key called the cipher key. The length of the cipher text and the cipher key is 128 bits each. The cipher key is
kept secret and is not revealed outside. The series of operations that convert plaintext to cipher text using the cipher key is called cipher.

All the operations in the AES algorithm are performed on a two dimensional array of bytes called the state. The two dimensional array is represented by $s[r,c]$. ‘s’ represents the name of the array, ‘r’ represents the row number and ‘c’ represents the column number. A byte is a sequence of 8 bits. 128 bits of plaintext can be represented by a group of 16 bytes. The dimensions of the array representing the state will be 4x4 with one byte accommodating in each block of the array.

Suppose the plain text is of the form $i_0 i_1 i_2 \ldots i_{126} i_{127}$ representing 128 bits. The byte representation will be of the form $i_n_0 i_n_1 i_n_2 \ldots i_n_{14} i_n_{15}$, where $i_n_0 = i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7$, $i_n_1 = i_8 i_9 i_{10} i_{11} i_{12} i_{13} i_{14} i_{15}$, $i_n_{15} = i_{120} i_{121} i_{122} i_{123} i_{124} i_{125} i_{126} i_{127}$.

The 16 bytes will be copied into the state array as shown in the Figure 2.1:

\[
\begin{array}{cccc}
\text{input bytes} & \text{State array} & \text{output bytes} \\
\hline
i_n_0 & i_n_4 & i_n_8 & i_n_{12} \\
i_n_1 & i_n_5 & i_n_9 & i_n_{13} \\
i_n_2 & i_n_6 & i_n_{10} & i_n_{14} \\
i_n_3 & i_n_7 & i_n_{11} & i_n_{15} \\
\end{array}
\]

Figure 2.1 Input, state array and output [16].

The AES algorithm processes the state array to convert the plain text in the input bytes to cipher text in the output bytes. The equations that transform the input bytes into the state array and from state array to the output bytes as referred from [16] are shown below:

\[
S[r,c] = in[r + 4 \cdot c] \text{ for } 0 \leq r \leq 4 \text{ and } 0 \leq c \leq 4 \quad (2.1)
\]

\[
\text{out}(r + 4 \cdot c) = S[r,c] \quad (2.2)
\]
AES algorithm operates on the state array using the functions SubBytes ( ), InvSubBytes ( ), ShiftRows ( ), InvShiftRows ( ), MixColumns ( ), InvMixColumns ( ), AddRoundkey ( ) and InvAddRoundkey ( ). The normal functions are performed at the transmitter side during encryption and inverse functions are performed at the receiver side during decryption. Each of these functions is explained in the following subsections.

2.2.1 SubBytes ( ) Transformation

![S-Box Diagram](image)

Figure 2.2 SubBytes ( ) applies to the S-box to each byte of the state [16].

In the SubBytes ( ) transformation each byte in the state array is substituted with one of the bytes in the S-Box shown in the Table 2.1. The substitution process takes place by selecting the corresponding row and column value in the two dimensional S-Box. For example if \( s_{0,0} \) is an element 57 in hexadecimal representation, \( s_{0,0} \) will be replaced by an element ‘5b’ in the S-box that corresponds to fifth row and seventh column. The SubBytes ( ) transformation is shown in the Figure 2.2.

The S-box shown in Table 2.1 is constructed from the multiplicative inverse over \( GF(2^8) \) that has good nonlinear properties. The SubBytes ( ) transformation provides the non linearity in the cipher text. Attacks based on simple algebraic properties are eliminated by using this transformation process.
Table 2.1 S-box: Substitution values for the byte xy [16].

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td>7c</td>
<td>77</td>
<td>7b</td>
<td>f2</td>
<td>6b</td>
<td>6f</td>
<td>c5</td>
<td>30</td>
<td>01</td>
<td>67</td>
<td>2b</td>
<td>fe</td>
<td>d7</td>
<td>ab</td>
<td>76</td>
</tr>
<tr>
<td>1</td>
<td>ca</td>
<td>e9</td>
<td>7d</td>
<td>fa</td>
<td>59</td>
<td>47</td>
<td>f0</td>
<td>ad</td>
<td>d4</td>
<td>a2</td>
<td>af</td>
<td>9c</td>
<td>a4</td>
<td>72</td>
<td>c0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b7</td>
<td>fd</td>
<td>93</td>
<td>26</td>
<td>3f</td>
<td>f7</td>
<td>cc</td>
<td>34</td>
<td>a5</td>
<td>e5</td>
<td>f1</td>
<td>71</td>
<td>d8</td>
<td>31</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>04</td>
<td>c7</td>
<td>23</td>
<td>c3</td>
<td>18</td>
<td>96</td>
<td>05</td>
<td>9a</td>
<td>07</td>
<td>12</td>
<td>80</td>
<td>e2</td>
<td>eb</td>
<td>27</td>
<td>b2</td>
<td>75</td>
</tr>
<tr>
<td>4</td>
<td>09</td>
<td>83</td>
<td>2c</td>
<td>1a</td>
<td>1b</td>
<td>6e</td>
<td>5a</td>
<td>a0</td>
<td>52</td>
<td>3b</td>
<td>d6</td>
<td>b3</td>
<td>29</td>
<td>e3</td>
<td>2f</td>
<td>84</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>d1</td>
<td>00</td>
<td>ed</td>
<td>20</td>
<td>Fc</td>
<td>b1</td>
<td>5b</td>
<td>6a</td>
<td>eb</td>
<td>be</td>
<td>39</td>
<td>4a</td>
<td>4c</td>
<td>58</td>
<td>cf</td>
</tr>
<tr>
<td>6</td>
<td>d0</td>
<td>ef</td>
<td>aa</td>
<td>fb</td>
<td>43</td>
<td>4d</td>
<td>33</td>
<td>85</td>
<td>45</td>
<td>f9</td>
<td>02</td>
<td>7f</td>
<td>50</td>
<td>3c</td>
<td>9f</td>
<td>a8</td>
</tr>
<tr>
<td>7</td>
<td>51</td>
<td>a3</td>
<td>40</td>
<td>8f</td>
<td>92</td>
<td>9d</td>
<td>38</td>
<td>f5</td>
<td>bc</td>
<td>b6</td>
<td>da</td>
<td>21</td>
<td>10</td>
<td>ff</td>
<td>f3</td>
<td>d2</td>
</tr>
<tr>
<td>8</td>
<td>cd</td>
<td>0c</td>
<td>13</td>
<td>ec</td>
<td>5f</td>
<td>97</td>
<td>44</td>
<td>17</td>
<td>c4</td>
<td>a7</td>
<td>7e</td>
<td>3d</td>
<td>64</td>
<td>5d</td>
<td>19</td>
<td>73</td>
</tr>
<tr>
<td>9</td>
<td>60</td>
<td>81</td>
<td>4f</td>
<td>dc</td>
<td>22</td>
<td>2a</td>
<td>90</td>
<td>88</td>
<td>46</td>
<td>ee</td>
<td>b8</td>
<td>14</td>
<td>de</td>
<td>5e</td>
<td>0b</td>
<td>db</td>
</tr>
<tr>
<td>a</td>
<td>e0</td>
<td>32</td>
<td>3a</td>
<td>0a</td>
<td>49</td>
<td>06</td>
<td>24</td>
<td>5c</td>
<td>e2</td>
<td>d3</td>
<td>ac</td>
<td>62</td>
<td>91</td>
<td>95</td>
<td>e4</td>
<td>79</td>
</tr>
<tr>
<td>b</td>
<td>e7</td>
<td>c8</td>
<td>37</td>
<td>6d</td>
<td>8d</td>
<td>d5</td>
<td>4e</td>
<td>a9</td>
<td>6c</td>
<td>56</td>
<td>f4</td>
<td>ea</td>
<td>65</td>
<td>7a</td>
<td>ae</td>
<td>08</td>
</tr>
<tr>
<td>c</td>
<td>ba</td>
<td>7b</td>
<td>25</td>
<td>2e</td>
<td>1c</td>
<td>a6</td>
<td>b4</td>
<td>c6</td>
<td>e8</td>
<td>dd</td>
<td>74</td>
<td>1f</td>
<td>4b</td>
<td>bd</td>
<td>8b</td>
<td>8a</td>
</tr>
<tr>
<td>d</td>
<td>70</td>
<td>3e</td>
<td>b5</td>
<td>66</td>
<td>48</td>
<td>03</td>
<td>f6</td>
<td>0e</td>
<td>61</td>
<td>35</td>
<td>57</td>
<td>b9</td>
<td>86</td>
<td>e1</td>
<td>1d</td>
<td>9e</td>
</tr>
<tr>
<td>e</td>
<td>e1</td>
<td>f8</td>
<td>98</td>
<td>11</td>
<td>69</td>
<td>d9</td>
<td>8e</td>
<td>94</td>
<td>9b</td>
<td>le</td>
<td>87</td>
<td>e9</td>
<td>ce</td>
<td>55</td>
<td>28</td>
<td>df</td>
</tr>
<tr>
<td>f</td>
<td>8c</td>
<td>a1</td>
<td>89</td>
<td>0d</td>
<td>bf</td>
<td>e6</td>
<td>42</td>
<td>68</td>
<td>41</td>
<td>99</td>
<td>2d</td>
<td>0f</td>
<td>b0</td>
<td>54</td>
<td>bb</td>
<td>16</td>
</tr>
</tbody>
</table>

2.2.2 InvSubBytes ( ) Transformation

Table 2.2 Inverse S-box: Substitution values for the byte xy [16].

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
<td>09</td>
<td>6a</td>
<td>d5</td>
<td>30</td>
<td>36</td>
<td>a5</td>
<td>38</td>
<td>bf</td>
<td>40</td>
<td>a3</td>
<td>9e</td>
<td>81</td>
<td>f3</td>
<td>d7</td>
<td>fb</td>
</tr>
<tr>
<td>1</td>
<td>7c</td>
<td>e3</td>
<td>39</td>
<td>82</td>
<td>9b</td>
<td>2f</td>
<td>ff</td>
<td>87</td>
<td>34</td>
<td>8e</td>
<td>43</td>
<td>44</td>
<td>c4</td>
<td>de</td>
<td>e9</td>
<td>cb</td>
</tr>
<tr>
<td>2</td>
<td>54</td>
<td>7b</td>
<td>94</td>
<td>32</td>
<td>a6</td>
<td>c2</td>
<td>23</td>
<td>3d</td>
<td>ee</td>
<td>4c</td>
<td>95</td>
<td>0b</td>
<td>42</td>
<td>fa</td>
<td>c3</td>
<td>4e</td>
</tr>
<tr>
<td>3</td>
<td>08</td>
<td>2e</td>
<td>a1</td>
<td>66</td>
<td>28</td>
<td>d9</td>
<td>24</td>
<td>b2</td>
<td>76</td>
<td>5b</td>
<td>a2</td>
<td>49</td>
<td>6d</td>
<td>8b</td>
<td>d1</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>72</td>
<td>f8</td>
<td>f6</td>
<td>64</td>
<td>86</td>
<td>68</td>
<td>98</td>
<td>16</td>
<td>d4</td>
<td>a4</td>
<td>5c</td>
<td>cc</td>
<td>5d</td>
<td>65</td>
<td>b6</td>
<td>92</td>
</tr>
<tr>
<td>5</td>
<td>6c</td>
<td>70</td>
<td>48</td>
<td>50</td>
<td>fd</td>
<td>ed</td>
<td>b9</td>
<td>da</td>
<td>5e</td>
<td>15</td>
<td>46</td>
<td>57</td>
<td>a7</td>
<td>8d</td>
<td>9d</td>
<td>84</td>
</tr>
<tr>
<td>6</td>
<td>90</td>
<td>d8</td>
<td>ab</td>
<td>00</td>
<td>8c</td>
<td>bc</td>
<td>d3</td>
<td>0a</td>
<td>f7</td>
<td>e4</td>
<td>58</td>
<td>05</td>
<td>b8</td>
<td>b3</td>
<td>45</td>
<td>06</td>
</tr>
<tr>
<td>7</td>
<td>d0</td>
<td>2c</td>
<td>1e</td>
<td>8f</td>
<td>ca</td>
<td>3f</td>
<td>0f</td>
<td>02</td>
<td>c1</td>
<td>af</td>
<td>bd</td>
<td>03</td>
<td>01</td>
<td>13</td>
<td>8a</td>
<td>6b</td>
</tr>
<tr>
<td>8</td>
<td>3a</td>
<td>91</td>
<td>11</td>
<td>41</td>
<td>4f</td>
<td>67</td>
<td>dc</td>
<td>ea</td>
<td>97</td>
<td>f2</td>
<td>cf</td>
<td>ce</td>
<td>f0</td>
<td>b4</td>
<td>e6</td>
<td>73</td>
</tr>
<tr>
<td>9</td>
<td>96</td>
<td>ac</td>
<td>74</td>
<td>22</td>
<td>e7</td>
<td>ad</td>
<td>35</td>
<td>85</td>
<td>e2</td>
<td>f9</td>
<td>37</td>
<td>e8</td>
<td>1c</td>
<td>75</td>
<td>df</td>
<td>6e</td>
</tr>
<tr>
<td>a</td>
<td>47</td>
<td>f1</td>
<td>1a</td>
<td>71</td>
<td>ld</td>
<td>29</td>
<td>c5</td>
<td>89</td>
<td>6f</td>
<td>b7</td>
<td>62</td>
<td>0e</td>
<td>aa</td>
<td>18</td>
<td>be</td>
<td>1b</td>
</tr>
<tr>
<td>b</td>
<td>fc</td>
<td>56</td>
<td>3e</td>
<td>4b</td>
<td>c6</td>
<td>d2</td>
<td>79</td>
<td>20</td>
<td>9a</td>
<td>db</td>
<td>c0</td>
<td>fe</td>
<td>78</td>
<td>cd</td>
<td>5a</td>
<td>f4</td>
</tr>
<tr>
<td>c</td>
<td>1f</td>
<td>dd</td>
<td>a8</td>
<td>33</td>
<td>88</td>
<td>07</td>
<td>c7</td>
<td>31</td>
<td>b1</td>
<td>12</td>
<td>10</td>
<td>59</td>
<td>27</td>
<td>80</td>
<td>ec</td>
<td>5f</td>
</tr>
<tr>
<td>d</td>
<td>60</td>
<td>51</td>
<td>7f</td>
<td>a9</td>
<td>19</td>
<td>b5</td>
<td>4a</td>
<td>0d</td>
<td>2d</td>
<td>e5</td>
<td>7a</td>
<td>9f</td>
<td>93</td>
<td>c9</td>
<td>9c</td>
<td>ef</td>
</tr>
<tr>
<td>e</td>
<td>a0</td>
<td>e0</td>
<td>3b</td>
<td>4d</td>
<td>ae</td>
<td>2a</td>
<td>f5</td>
<td>b0</td>
<td>c8</td>
<td>eb</td>
<td>bb</td>
<td>3c</td>
<td>83</td>
<td>53</td>
<td>99</td>
<td>61</td>
</tr>
<tr>
<td>f</td>
<td>17</td>
<td>2b</td>
<td>04</td>
<td>7e</td>
<td>ba</td>
<td>77</td>
<td>d6</td>
<td>26</td>
<td>e1</td>
<td>69</td>
<td>14</td>
<td>63</td>
<td>55</td>
<td>21</td>
<td>0c</td>
<td>7d</td>
</tr>
</tbody>
</table>
InvSubBytes ( ) transformation is the inverse of the SubBytes ( ) transformation. This transformation makes use of the Inverse S-box shown in Table 2.2. Each byte in the state array is substituted with one of the bytes in the inverse S-Box. The substitution process will be similar to that in the SubBytes ( ) transformation. For example if \( s_{0,0} = 32 \) (hexadecimal). \( s_{0,0} \) will be replaced by an element ‘a1’ in the inverse S-box that corresponds to third row and second column.

2.2.3 ShiftRows ( ) Transformation

The ShiftRows ( ) Transformation operates on the rows of the state array. The bytes in each row of the state array are shifted to the left cyclically by a different offset. The first row remains intact. The second, third and fourth rows are shifted by an offset of 1, 2 and 3 correspondingly. The ShiftRows ( ) Transformation process is shown in the Figure 2.3.

![Figure 2.3 ShiftRows ( ) cyclically shifts the last three rows in the State [16].](image)

2.2.4 InvShiftRows ( ) Transformation

The InvShiftRows ( ) transformation is the inverse of the ShiftRows ( ) transformation. The first row remains intact. The second, third and fourth rows are
cyclically shifted to the right by an offset of 1, 2 and 3 correspondingly. The InvShiftRows( ) transformation is shown in the Figure 2.4.

![Figure 2.4 InverseShiftRows( ) cyclically shifts the last three rows in the State [16].](image)

2.2.5 MixColumns( ) Transformation

![Figure 2.5 MixColumns( ) operates on the state column-by-column [16].](image)

The MixColumns( ) Transformation operate on the columns of the state array. The input for the function is a four byte column of the state array and the output is a four byte cipher text. Each column is multiplied modulo $x^4 + 1$ by a fixed polynomial $a(x)$ considering it as a polynomial over $GF(2^8)$. The transformation process is shown in the
Figure 2.5. The equation for polynomial \( a(x) \) and the matrix multiplication representation of the transformation as referred from [16] are:

\[
a(x) = \{03\}x^3 + \{01\}x^3 + \{01\}x + \{02\}
\]

\[
s'(x) = a(x) \otimes s(x)
\]

\[
\begin{bmatrix}
  s_{0,c}' \\
  s_{1,c}' \\
  s_{2,c}' \\
  s_{3,c}'
\end{bmatrix} = 
\begin{bmatrix}
  02 & 03 & 01 & 01 \\
  01 & 02 & 03 & 01 \\
  01 & 02 & 03 & 01 \\
  03 & 01 & 01 & 02
\end{bmatrix}
\begin{bmatrix}
  s_{0,c} \\
  s_{1,c} \\
  s_{2,c} \\
  s_{3,c}
\end{bmatrix}
\]

for \( 0 \leq c < 4 \) \hspace{1cm} (2.5)

ShiftRows ( ) operation followed by a MixColumns ( ) operation makes the cipher text change into an unpredictable random manner dissipating the redundancy in the plain text.

2.2.6 InvMixColumns( ) Transformation

The InvMixColumns ( ) Transformation operate on the columns of the state array. The input for the function is four bytes column of the state array and outputs four bytes. The process is similar to that of MixColumns ( ) transformation. Each column of the state is multiplied modulo \( x^4 + 1 \) by a fixed polynomial \( a^{-1}(x) \), considering it as a polynomial over GF(\( 2^8 \)). The equation for polynomial \( a^{-1}(x) \) and the matrix multiplication representation of the transformation referred from [16] are:

\[
a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}
\]

\[
s'(x) = a(x) \otimes s(x)
\]

\[
\begin{bmatrix}
  s_{0,c}' \\
  s_{1,c}' \\
  s_{2,c}' \\
  s_{3,c}'
\end{bmatrix} = 
\begin{bmatrix}
  0e & 0b & 0d & 09 \\
  09 & 0e & 0b & 0d \\
  0d & 09 & 0e & 0b \\
  0b & 0d & 09 & 0e
\end{bmatrix}
\begin{bmatrix}
  s_{0,c} \\
  s_{1,c} \\
  s_{2,c} \\
  s_{3,c}
\end{bmatrix}
\]

for \( 0 \leq c < 4 \). \hspace{1cm} (2.8)
2.2.7 AddRoundKey ( ) Transformation

Figure 2.6 AddRoundKey ( ) XORs each column of the state with a word from the key schedule [16].

A subkey is derived from the main key for each round in the AES encryption process. This operation is performed by the key expansion algorithm. The length of the subkey generated is 16 bytes. The subkey is divided into groups of four words and is performed bitwise exclusive-or operation with each column of the state array. \([w_i]\) represents the words in the key schedule. The range of round is \(0 \leq \text{round} \leq 4\). The AddRoundKey ( ) transformation process is shown in the Figure 2.6. The value of \(l\) represented in the Figure 2.6 is \(4 \cdot \text{round}\). The mathematical representation of the AddRoundKey ( ) transformation as referred from [16] is as shown below:

\[
[s_0,c', s_1,c', s_2,c', s_3,c'] = [s_0,c, s_1,c, s_2,c, s_3,c] \oplus [w_{4 \cdot \text{round}} + c] \text{ for } 0 \leq c \leq 4, \quad (2.9)
\]

2.2.8 InvAddRoundkey ( ) Transformation

The AddRoundkey ( ) transformation explained in Section 2.2.7 involves an exclusive-or operation with the key schedule. AddRoundkey ( ) transformation which is its own inverse can serve the purpose of the InvAddRoundkey ( ) transformation. Instead
of using the InvAddRoundkey ( ) transformation process AddRoundkey ( )
transformation with the key schedule taken in the reverse order is used.

2.3 Key Expansion

The cipher key is expanded to generate key schedule for use in the AddRoundkey
( ) transformation function. The AddRoundkey transformation function is used a multiple
number of times during the AES encryption and decryption process. Instead of using a
single key for all the times, a series of new keys generated by the key expansion routine
are used to increase the diffusion of the bits. The flowchart for the key expansion is
shown in Figure 2.7. The key expansion routine generates a total of ten keys from the
original cipher key. The initial round constant word is set to {0x01, 0x00, 0x00, 0x00}. The
round constant is multiplied by 2 for each round. For the second round the round
constant word will be {0x02, 0x00, 0x00, 0x00}.

The last four bytes in the key are copied into a temporary word. A left shift
operation is performed on the word. This operation is similar to the operation performed
on the second row of the state array in the ShiftRows ( ) transformation mentioned in
Section 2.3.3. The four bytes in the temporary word are substituted with the elements of
the S-box shown in Table 2.1. The substitution process is similar to the SubBytes ( )
operation mentioned in Section 2.2.1. An exclusive-or operation is performed on the
temporary word with the current round constant. A bitwise exclusive-or operation is
performed on the four words of the cipher key with this temporary word. The new 16-
byte sequence obtained is the expanded key. This key is used as the new cipher key in the
next round to generate the second key using the same procedure mentioned above. The
key expansion routine returns a total of ten keys for use in the AES encryption process.
Figure 2.7 Key expansion flowchart [2].
2.4 AES Encryption Flowchart

The flowchart for the AES encryption process is shown in the Figure 2.8. The encryption routine is performed at the transmitter side. The AddRoundKey(), SubBytes(), ShiftRows() and MixColumns() functions used in the AES encryption process are explained in Section 2.2. The 16 bytes plain text that is to be encrypted is copied in to the encrypt block. An initial AddRoundKey() operation is performed on the encrypt block.
using the first key generated from the key expansion routine. The SubBytes ( ), ShiftRows ( ), MixColumns ( ) and AddRoundKey ( ) transformations are performed in a loop for eight rounds. The sequence of steps that are executed in this loop is referred to as encryption round. A new key from the key expansion routine is used for each round. Upon exit from the loop the SubBytes ( ), ShiftRows ( ) and AddRoundKey ( ) transformations are performed single time using the last key from the key expansion routine. The output from the AES encryption algorithm is a 16-byte cipher text.

2.5 AES Decryption Flowchart

The flowchart for the AES decryption process is shown in the Figure 2.9. The decryption routine is performed at the receiver side. The AddRoundKey ( ), InvSubBytes ( ), InvShiftRows ( ) and InvMixColumns ( ) functions used in the AES decryption process are explained in Section 2.2. The cipher text that is to be decrypted is initially copied in to the decrypt block. The key schedule for the decryption process is in the reverse order of the key schedule used in the encryption process. An initial AddRoundKey ( ) operation is performed on the decrypt block using the first key generated from the key expansion routine. The InvShiftRows ( ), InvSubBytes ( ), AddRoundKey ( ) and InvMixColumns ( ) transformations are performed in a loop for eight rounds. The sequence of steps that are executed in this loop is referred to as decryption round. A new key from the key expansion routine is used for each round. Upon exit from the loop the InvShiftRows ( ), InvSubBytes ( ), AddRoundKey ( ) transformations are performed single time using the last key from the key expansion routine. The output from the AES decryption algorithm is a 16-byte plain text recovered from the ciphertext.
2.6 Conclusions

The AES concepts have been introduced. The rudiments essential for understanding the AES algorithm is discussed. The transformation functions used in the AES encryption and decryption algorithms are discussed. The flowcharts for the key expansion, AES encryption and AES decryption algorithms are discussed.
CHAPTER III

CIPHER BASED MESSAGE AUTHENTICATION CODE (CMAC)

This chapter discusses the Cipher Based Message Authentication Code. Preliminary symbols, operations and functions required for understanding the CMAC algorithm are discussed. The block cipher, inputs and outputs for the algorithm are mentioned. Subkey generation, MAC generation and MAC verification algorithms are explained.

3.1 CMAC

CMAC is a cipher based message authentication code. It is based on symmetric key block cipher, Advanced Encryption Standard (AES) algorithm. CMAC can be considered as a mode of operation of the block cipher [1]. CMAC provides stronger assurance of data integrity than a checksum or an error detecting code [1]. CMAC is capable of detecting unauthorized modification of data.

3.2 Preliminary Symbols, Operations and Functions

In order to understand the CMAC algorithm it is important to know the preliminary symbols, operations and functions used in the algorithm. Some of them like $0^5, X \parallel Y, X \oplus Y, LSB_s(X), MSB_s(X)$ which are used for the message authentication code are explained in detail in the following subsections. An example is shown for each operation for a better understanding.
3.2.1 $0^s$

If $s$ is a positive integer then $0^s$ is a bit string that contain $s$ ‘0’ bits. For example, $0^6$ represents a bit string that contain 6 zeros i.e. 000000. $0^s$ is used in the representation of $R_b$.

3.2.2 $X \parallel Y$

Concatenation refers to the operation of appending one sequence to the other. The concatenation operator is denoted by $\parallel$. If $X$ and $Y$ are two bit strings then the concatenation operation between $X$ and $Y$ is denoted by $X \parallel Y$. For example, $010 \parallel 10011 = 01010011$.

3.2.3 $X \oplus Y$

The bitwise exclusive-OR operator is denoted by $\oplus$. If the length of the two bit strings is equal then the exclusive-OR operation performed on the bit strings is denoted by modulo-2 addition. For example, $11001 \oplus 10011 = 01010$.

3.2.4 $LSB_s(X)$

If $X$ is a bit string of length greater than or equal to $s$, the function $LSB_s(X)$ gives the $s$ least significant bits (right-most bits) in the bit string $X$. For example, $LSB_3(10010101) = 101$.

3.2.5 $MSB_s(X)$

If $X$ is a bit string of length greater than or equal to $s$, the function $MSB_s(X)$ gives the $s$ most significant bits (left-most bits) in the bit string $X$. For example, $MSB_3(10010101) = 100$. 
3.3 Block Cipher

A Block Cipher is a method of encrypting plaintext to ciphertext in which a cryptographic key and algorithm are applied to a block of data at once as a group rather than to one bit at a time [19]. The CMAC algorithm chosen in this thesis depends on the symmetric key block cipher (AES). Hence it is a mode of operation of the AES. The key used for the CMAC and the AES is the system secret key. The forward cipher function is a permutation on bit strings of a fixed length; the strings are called blocks [1]. If the forward cipher function of the block cipher is denoted as $CIPH_k(\cdot)$, the function implements the AES algorithm on the block of length $k$.

3.4 Subkeys

Two secret keys denoted as $K1$ and $K2$ are derived from the block cipher. The length of the generated subkeys will be same as that of the block size i.e. 128 bits. The secret keys generated and any intermediate values computed in the generation process like $CIPH_k(0^b)$ will be kept secret. One of the elements of the subkey generation process is a bit string, denoted by $R_b$, $b$ represents the number of bits in the bit string. $R_{128}$ represents a series of 120 zero bits followed by a bit string $1000011$ as referred from [1] is shown in equation 3.1.

$$R_{128} = 0^{120} \cdot 1000011 \quad (3.1)$$

3.5 MAC Generation and Verification

MAC is generated at the transmitter side by means of applying the MAC generation process to the data to be authenticated. The generated MAC is transmitted to the receiver along with the unencrypted data. MAC verification process is performed at the receiver end for transmitter authentication.
3.6 Input and Output for the MAC Generation

For the AES block cipher and secret key, the input and output to the MAC generating function are denoted as $M$ and $T$ correspondingly. The input is a sequence of nine bytes which includes serial number, counter value and command. The length of the output $T$ (MAC) is denoted $Tlen$ and is four bytes in size.

3.7 CMAC Specification

Subkey generation, MAC generation, and MAC verification are specified in the following Subsections. The specification for the CMAC contains the inputs, the outputs, a notation for the function used, the sequence of steps and a summary.

3.7.1 Subkey Generation

The prerequisites for the subkey generation are block cipher $CIPH$ with block size $b$ and key $K$. The outputs are the subkeys $K1$ and $K2$. The notation for the subkey generation function is $SUBK(K)$. The sequence of steps for the subkey generation process as referred from [1] are listed below:

1. Let $L = CIPH_k(0^b)$.

2. If $MSB_1(L) = 0$, then $K1 = L \ll 1$;
   
   Else $K1 = (L \ll 1) \oplus R_b$;

3. If $MSB_1(K1) = 0$, then $K2 = (K1 \ll 1)$
   
   Else $K2 = (K1 \ll 1) \oplus R_b$.

4. Return $K1, K2$.

In Step 1, the AES algorithm is applied to a bit string that contain entirely of 128 ‘0’ bits. In Step 2, a left shift operation is performed on the output from the AES algorithm, $L$ to obtain the first key. If the most significant bit of $L$ is zero then an
exclusive-or operation is performed with the bit string $R_{128}$ represented in equation 3.1. In Step 3, the same process in Step 2 is applied on the subkey $K1$ to derive the subkey $K2$. The subkeys are derived one time and are stored in the memory for future use.

3.7.2 MAC Generation

The prerequisites for the MAC generation are block cipher CIPH with block size $b$ and key $K$. The input is the message $M$ of bit length $Mlen$. The output is the MAC $T$ of bit length $Tlen$. The notation for the MAC generation function is CMAC $(K, M, Tlen)$. The sequence of steps for the MAC generation process as referred from [1] are listed below:

1. Apply the subkey generation process to $K$ to produce $K1$ and $K2$.

2. If $Mlen=0$, let $n=1$; else, let $n = \lceil \frac{Mlen}{b} \rceil$.

3. Let $M_1, M_2, ..., M_{n-1}, M_n^*$ denote the unique sequence of bit strings such that

   $M = M_1 \parallel M_2 \parallel \cdots \parallel M_{n-1} \parallel M_n^*$ where $M_1, M_2, ..., M_{n-1}$ are composite blocks.

4. If $M_n^*$ is a complete block, let $M_n = K1 \oplus M_n^*$; else, let $M_n = K2 \oplus (M_n^* \parallel 10^{j})$, $j=nb-Mlen-1$.

5. Let $C_0 = 0^b$.

6. For $i=1$ to $n$, let $C_i = CIPH_k(C_{i-1} \oplus M_i)$.

7. Let $T = MSB_{Tlen}(C_n)$.

8. Return $T$.

In Step 1, the subkey generation process is applied to the key $K$ to generate subkeys $K1$ and $K2$ using the algorithm specified in Section 3.7.1. In Step 2, ceiling operation is performed on the length of the input message to determine the number of possible composite blocks. The length of the message used in this thesis is 72 bits. In
Step 3, the message is partitioned into a possible number of composite blocks with each block having a length of $k$. For the AES algorithm the value of $k$ is 128 bits.

In Step 4, an exclusive-or operation is performed on the last block in the message $M$, denoted by $M_n^*$ with the subkey $K1$ if it is a composite block. If $M_n^*$ is not a composite block a bit string ‘1’ followed by a minimum number of zeros is appended and an exclusive-or operation is performed with the subkey $K2$. The two cases of MAC generation are shown in the Figure 3.1. In this application $M_n^*$ is a bit string of 72 bits length. A bit string 1 followed by 55 zero bits is appended to the message to form a sequence of 128 bits.

In Step 5, an initial vector containing $b$ zeros is chosen. In Step 6 the cipher block chaining (CBC) technique, with the initial vector in step 5 is applied to the formatted message. Steps 5 and 6 are not necessary since composite blocks are not available. Cipher block chaining is not applicable since the length of the message is smaller than a single block. In Step 7, the final CBC output is truncated to a length of $Tlen$ and is specified as MAC for the output. The 4 most significant bytes in the output from the AES encryption are specified as MAC in this application.

![Diagram](image.png)

Figure 3.1 Illustration of the two cases of the MAC generation [1].
In summary the 72 bit message containing the 32 bit serial number, 32 bit counter value and an 8 bit command is appended with a single ‘1’ bit followed by 55 zeros to change the length of the message to 128 bits. An exclusive-or operation is performed on this 128 bit sequence with the subkey $K_2$ derived from the Section 3.7.1. The result is used as a plaintext for the AES algorithm and encrypted to ciphertext using the secret key. The output of the AES algorithm is a sequence of 16 bytes. The four most significant bytes are chosen as MAC and transmitted along with the 9 byte message. The MAC generation process is better illustrated in Figure 3.2.

![Figure 3.2 Generating MAC from the message](image)

3.7.3 MAC Verification

The prerequisites for the MAC verification process are the Block cipher CIPH with block size $b$, key $K$, subkeys $K_1$ and $K_2$ and MAC length $Tlen$. The input is the message $M$ of bit length $Mlen$ and MAC $T$. The output is determined as either valid or
invalid. The notation for the MAC verification function is \( \text{VER} (K, M, T') \). The sequence of steps for the MAC verification process referred from [1] is shown below:

1. Apply the MAC generation process to \( M \) to produce \( T \).

2. If \( T=T' \), return \text{VALID}; else, return \text{INVALID}.

In Step 1, the receiver performs the MAC generation process on the message received from the transmitter after excluding the MAC bytes. In Step 2, the MAC generated is compared with the MAC transmitted by the transmitter for validation. The same secret key is used for generating the MAC at the receiver end.

3.8 Conclusions

This chapter discusses the MAC. The preliminary symbols, operations and functions necessary for understanding the algorithms are discussed. The sequence of steps for the subkey generation, MAC generation and MAC verification algorithms are discussed.
CHAPTER IV
MESSAGE FORMAT AND COMMUNICATION FLOWCHARTS

This chapter discusses the message format and the transmission format. The block diagrams for transmitter and receiver are shown. The flowcharts for the transmitter and receiver in normal mode and learn mode of operations are explained. The flow chart for the learn process is explained in detail. The hardware and software developmental tools used in this thesis are discussed.

4.1 Message Format

The format of the message in a secure system mentioned in Section 1.6.3 with all the four goals satisfied shall contain a serial number, sequential counter, command and MAC as shown in the Figure 1.4. The message contents with respect to their length and position in the transmission word are shown in Figure 4.1. A total of 13 bytes are used for the message.

4.1.1 Serial Number

Each transmitter should have a unique serial number. The serial number is fixed and does not change throughout the process. Since no two transmitters share a same serial
number, the length of the serial number should be large enough to accommodate all the transmitters in use. A 32 bit serial number is smart enough.

4.1.2 Counter Value

The counter value will be incremented on each button press. Even if the transmitter is not in the receivers range, a button press detected on the key fob will increment the counter. The counter value is chosen in such a way that the skipped messages fall within the acceptance window. A 32 bit counter value is chosen.

4.1.3 Command

Command is used to send the button information to the receiver. Command is not transmitted in the learn mode of operation. Based on the button press the command bits are activated. The length of the command depends on the number of buttons on the transmitter. An 8 bit command is chosen.

4.1.4 MAC

The size of the MAC determines the level of security of the system. The length of the MAC as specified in [17] should be in the range \( \frac{L}{2} \leq t \leq L \), where \( L \) is the length of the output of the MAC function and \( t \) is the length of the MAC used. A 32 bit MAC is chosen.

4.2 Transmission Format

This section describes the format of the signal transmission. The signal contains a preamble, a header and a message. A guard period is provided after each signal transmission. Manchester encoding, which is used in the signal transmission to represent the logic “0” and logic “1” is discussed in the following Subsection 4.2.1 using appropriate figures.
4.2.1 Manchester Encoding

Manchester encoding is used to represent the binary bits 0 and 1. Logic “0” and logic “1” are represented as shown in the Figure 4.2. $T_E$ represents the length of the on time for logic “1” or the off time for logic “0”. $T_{BP}$ represents the bit period.

![Manchester Encoding Diagram]

Figure 4.2 Manchester encoding.

The transmission format for the wireless communication is as shown in the Figure 4.3. Each transmission begins with a 50% duty cycle preamble, a header, followed by an encrypted and fixed code portions. Guard time is provided at the end of each transmission before another transmission can begin.

The length of the Preamble is $23T_E$ and that of header is $10T_E$. The length of fixed and encrypted code portions vary for normal mode of operation and for learn mode of operation. For normal mode the length of encrypted portion is $96T_E$ and that of fixed
portion is 216Te. For learn mode the length of encrypted portion is 384Te and that of fixed portion is 96Te. The length of the guard time is 39Te.

4.3 Encryption and Decryption

The basic operation performed on the transmitting side is encryption and transmission, and that of receiving side is reception and decryption. Encryption is the method of encoding a message or data so that its contents are hidden from outsiders [2]. The plain text in its original form is encrypted to cipher text in order to hide the contents of the message. Decryption is the method of retrieving the original message or data and typically cannot be performed without knowing the proper key [2]. The secret key used to decrypt the cipher text is stored in the EEPROM and is made available to the microcontroller during the decryption process.

4.4 Block diagrams of transmitter side and receiver side

The block diagrams of the transmitter side and the receiver side that are required to understand the basic operations performed at the transmitter and receiver are discussed in the following Subsections.

4.4.1 Transmitter Side

The block diagram for transmitter operation is shown in the Figure 4.4. The buttons are the inputs to the device. The microcontroller performs the key operation of encrypting the data sequence. The EEPROM contains certain information like the serial number, secret key, counter value and S-box that are necessary to be kept secret. The RF transmitter performs the operation of transmitting the Manchester encoded bits to the receiver by means of an RF Signal.
4.4.2 Receiver Side

The block diagram for receiver operation is shown in the Figure 4.5. The RF receiver performs the operation of receiving the signal from the transmitter. The microcontroller performs the key operation of decrypting the received data. The EEPROM contains the records of transmitters, S-box, and InvS-box. The output of the receiver contains LED’s to be turned on when signaled by the microcontroller.

4.4.3 EEPROM Record

Each of the EEPROM record contains a set of data as shown in the Table 4.1. Each of the learned transmitter information is stored in the EEPROM as a record. Each record contains the latest button information (FCode), serial number (IDHi-IDMi2-
IDMi1-IDLo), synchronization counter value (HopHi-HopMi2-HopMi1-HopLo) and the secret key (KEY15-KEY14-KEY13-KEY12-KEY11-KEY10-KEY9-KEY8-KEY7-KEY6-KEY5-KEY4-KEY3-KEY2-KEY1-KEY0) up to date.

Table 4.1 EEPROM record.

<table>
<thead>
<tr>
<th>DatoHi</th>
<th>DatoLo</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCode</td>
<td>Reserved</td>
<td>+0</td>
</tr>
<tr>
<td>IDMi1</td>
<td>IDLo</td>
<td>+2</td>
</tr>
<tr>
<td>IDHi</td>
<td>IDMi2</td>
<td>+4</td>
</tr>
<tr>
<td>HopMi1</td>
<td>HopLo</td>
<td>+6</td>
</tr>
<tr>
<td>HopHi</td>
<td>HopMi2</td>
<td>+8</td>
</tr>
<tr>
<td>KEY1</td>
<td>KEY0</td>
<td>+10</td>
</tr>
<tr>
<td>KEY3</td>
<td>KEY2</td>
<td>+12</td>
</tr>
<tr>
<td>KEY5</td>
<td>KEY4</td>
<td>+14</td>
</tr>
<tr>
<td>KEY7</td>
<td>KEY6</td>
<td>+16</td>
</tr>
<tr>
<td>KEY9</td>
<td>KEY8</td>
<td>+18</td>
</tr>
<tr>
<td>KEY11</td>
<td>KEY10</td>
<td>+20</td>
</tr>
<tr>
<td>KEY13</td>
<td>KEY12</td>
<td>+22</td>
</tr>
<tr>
<td>KEY15</td>
<td>KEY14</td>
<td>+24</td>
</tr>
</tbody>
</table>

4.5 Flowchart for Transmitter Session

The flowchart for the transmitter session is shown in the Figure 4.6. When the transmitter device is powered up the peripherals are initialized. The transmitter spends most of the time in sleep mode, waiting for a button press. If a learn button press is detected key schedule is calculated using shared key. Secret key is encrypted using the AES encryption algorithm with shared key and is transmitted along with the serial number to the receiver and enters the sleep mode.

If a normal button press is detected the transmitter calculates the key schedule using the secret key and generates the subkeys for use in the CMAC generation process. The sequential counter is incremented by 1.
Device powered up

Initialize peripherals

Sleep mode of operation

Read command from keypad

Teach command

Yes

Calculate keyschedule using shared key

No

First time transmission

Yes

Encrypt secret key

No

Increment counter

Encrypt sno, counter, command

Transmit serial number + encrypted secret key

Transmit sno, counter, command and MAC

Calculate keyschedule for secret key, subkey for CMAC generation

Figure 4.6 Flowchart for transmitter session.
The serial number, sequential counter value and button information is encrypted using the CMAC mode of AES. The serial number, counter value and button information along with the MAC bytes are transmitted to the receiver. The key schedule and subkeys are calculated and stored in the EEPROM for the first time and is used for the rest of the normal button press.

4.6 Flowchart for Receiver Session

The flowchart for the receiver operation is discussed separately for normal mode of operation and learn mode of operation. The learn mode of operation is performed prior to the normal mode of operation. The normal mode of operation is explained in Section 4.6.1 and the learn mode of operation is explained in Section 4.6.2.

4.6.1 Normal Mode of Operation

The flowchart for receiver in normal mode of operation is shown in Figure 4.7 below. The receiver spends most of the time in sleep mode and continuously polls the receiver pin for the signal. Up on receiving the message from the transmitter, the receiver verifies the identity of the transmitter by checking the serial number. If the serial number is not in the list of accepted transmitters the receiver ignores the message. If the serial number is matched the receiver verifies the validity of the counter value by comparing with the range of accepted counter values as mentioned in Section 1.7. The receiver generates the MAC bytes using the fixed code portion of the received signal and compares with that of the transmitted MAC bytes. Up on validation of the MAC bytes the counter value and the command are updated in the EEPROM and the requested operation is performed.
Interrupt driven polling

Message received from transmitter

Serial No Match

Yes

Counter with in range

No

Yes

Construct MAC bytes

MAC compare

True

Update counter, value, command in EEPROM

Perform requested function

Figure 4.7 Flowchart for receiver in normal mode of operation.
Learn button press detected

10 sec Button press

250 msec Normal Button press

Receive learn message from transmitter <18 sec

Check if Serial no is in EEPROM

Yes

Transmitter already learnt

No

Check if empty record present in EEPROM

Yes

Reached max no of transmitters

No

Decrypt message for secret key using shared key

Update serial number, secret key in EEPROM record

End of learn session

Figure 4.8 Flowchart for receiver in learn mode of operation
4.6.2 Learn mode of Operation

The flowchart for the receiver in learn mode of operation is shown in the Figure 4.8. The receiver is programmed to learn mode by means of an external switch called the learn button. Pressing the learn button for more than 10 seconds will erase all the previously learned transmitter records. The receiver can no longer accept the previously learned transmitters. If the learn button on the receiver is pressed for less than 10 seconds the receiver will be made ready to add new transmitters, keeping the old transmitter records. There is a 250 msec normal button debounce.

Up on receiving the message from the transmitter the receiver compares the serial number received with that of the previously stored serial numbers database. If there is a match in the serial numbers the receiver identifies that the transmitter is already learnt to the receiver and terminates the session. If the serial number is not found by the receiver it will check for the availability of an empty record in the EEPROM. If an empty record is found, the receiver decrypts the message using the AES decryption algorithm and shared key. The decrypted message contains the secret key. The serial number and the secret key are stored in the EEPROM record and exits the learn mode.

4.7 Learn Session

The flowchart for learn session is shown in Figure 4.9. The receiver is made to enter the learn mode of operation by an external switch called learn button. Pressing the learn button for more than 10 seconds will erase all the previously learned transmitter records. This receiver can no longer accept the previously learned transmitters. If the receiver’s learn button is pressed for less than 10 seconds the receiver is ready to add new transmitters, keeping the old transmitter records. The receiver expects the learn sequence
Learning Session

Receiver in Learn mode (using learn switch)

- Learn button press >10 sec
  - Yes: Erase all previously learned transmitters
  - No: Receive learn sequence from transmitter <18 sec

- Yes: Learn session timeout
- No: Need to teach more transmitters

- Yes: Learn session trigger
- No: Receiver is ready to accept messages from learned transmitters

- End of learn session

Figure 4.9 Learn session.
from the transmitter in 18 seconds. If the receiver doesn’t receive the learn sequence with in 18 seconds then the lean process is terminated. If the receiver receives a learn sequence from the transmitter, the transmitter will be learned to the receiver after decrypting and verifying its identity. Once the transmitter is learned it is ready to use. The receiver waits for another 18 seconds for learning more transmitters and terminates its session automatically if learn sequence is not received from any of the transmitters in the system.

4.8 Hardware and Software Development Tools

The hardware and software development tools used in the thesis are discussed in the following subsections. The PIC24FJ128GA010 16-bit microcontroller is programmed in C language with the help of MPLAB IDE and the MPLAB C30 language suite.

4.8.1 MPLAB IDE

MPLAB IDE is a Windows Operating System (OS) software program that runs on a PC to develop applications for Microchip microcontrollers and digital signal controllers [18]. It is called an Integrated Development Environment, or IDE, because it provides a single integrated "environment" to develop code for embedded microcontrollers [18].

4.8.2 MPLAB C Compiler for PIC24 MCUs

The MPLAB C Compiler for PIC24 MCUs is a full-featured ANSI compliant C compiler for the Microchip PIC24 family of 16-bit devices [18]. MPLAB C is fully compatible with Microchip’s MPLAB Integrated Development Environment (IDE), allowing source level debugging with the MPLAB REAL ICE Emulator, MPLAB ICD 2 In-Circuit Debugger and MPLAB SIM Simulator [18].
4.8.3 MPLAB ICD 2

The MPLAB ICD 2 is a low-cost in-circuit debugger (ICD) and in-circuit serial programmer (ICSP) [5]. MPLAB ICD 2 is intended to be used as an evaluation, debugging and programming aid [5]. A picture of MPLAB ICD 2 is shown in Figure 4.10. The MPLAB ICD 2 offers the following features as referred from [5]:

1. Real-time and single-step code execution
2. Breakpoints, Register and Variable Watch/Modify
3. In-circuit debugging
4. Target Vdd monitor
5. Diagnostic LEDs
6. MPLAB IDE user interface
7. RS-232 serial or USB interface to a host PC

Figure 4.10 Explorer 16 development board, MPLAB ICD 2 debugger [12].
4.8.4 SLEEP mode of Operation

The PIC24FJ128GA010 family of devices provides the ability to manage power consumption by selectively managing clocking to the CPU and peripherals [5]. PIC24 devices can operate on two power saving modes, sleep mode and idle mode. In sleep mode of operation the clock is stopped and all the code executions are halted. Some of the sleep mode features as referred from [5] are listed below:

1. The device current consumption will be reduced to a minimum provided that no I/O pin is sourcing current.
2. The system clock source is shut down. If an on-chip oscillator is used, it is turned off.
3. Some device features or peripherals may continue to operate in the sleep mode. This includes input change notification on the I/O ports, or peripherals that use an external clock input.

The device needs to be wake-up after entering the sleep mode of operation in order to perform any function. The device will wake-up from sleep mode on a few occasions. The events that can wake-up the device as referred from [5] are listed below:

1. On any interrupt source that is individually enabled.
2. On any form of device reset.
3. On a WDT time-out.

4.9 Conclusions

The components of the message, the basic block diagrams for transmitter side and receiver side are discussed. The flow charts for the encryption and decryption in normal
mode and learn mode of operation are discussed. The hardware and software tools used for the implementation of the code are discussed.
CHAPTER V
SYSTEM IMPLEMENTATION

This chapter gives details of the test run performed to verify the functionality of the program. The encoder, decoder softwares are programmed in C using MPLAB IDE v8.14. MPLAB ICD 2 device is used as the tool to debug the software. After successfully building the software, a test run is conducted for verifying the functionality of the encoder and decoder software programs. The 32-bit shared key, 32-bit secret key and the 8-bit serial number used for the test run are shown below in hexadecimal notation.

1. Shared key=0x7F3A8A2D8F501CCBB9CAB7727B8F6B07
2. Secret key=0x2B7E151628AED2A6ABF7158809CF4F3C
3. Serial number=0x54545667

Both the transmitter and receiver microcontrollers are in SLEEP mode of operation initially. The transmitter waits for a button press and the receiver polls the receiver input pin awaiting for a signal from the transmitter. The decoder has no knowledge of the transmitters since no transmitter is learned to the receiver initially, hence could not accept any transmitter commands.

The learn process is initiated by pressing the learn button on the receiver microcontroller. The receiver is acknowledged with the button press to expect the learn sequence from the transmitter within 18 seconds. The learn button on the transmitter microcontroller aids with the construction and transmission of the learn sequence.
The transmitter encodes the secret key using the shared key and transmits the serial number and the encrypted secret key. The receiver decrypts the encrypted portion of the received learn sequence for generating secret key using the AES decryption algorithm along with shared key and stores the serial number and the secret key in the EEPROM record. Hence the transmitter is learned to the receiver. Both the transmitter and the receiver exits the learn mode of operation and enters the sleep mode waiting for a button press and signal reception correspondingly.

The Normal mode of operation is initiated with the normal button press detection at the transmitter side. The transmitter increments the counter by 1 and encrypts the button information and counter value using the AES and constructs the MAC bytes. The transmitter transmits the command, the serial number and the MAC. The receiver decrypts the encrypted portion of the received signal. Up on validation of authenticity the receiver changes the corresponding output bit to 1 and exits the loop.

The sequence of process is observed by means of inserting break points in the code and single stepping through the program. The results are viewed in the watch window. The test transmitter was successfully learned to the test receiver. The EEPROM record values are viewed and verified. The normal mode of operation is successfully verified. The transmitter is erased from the decoder record and reprogrammed.
CHAPTER VI
CONCLUSIONS AND FUTURE WORK

The importance of security in remote keyless systems has been discussed. The AES and CMAC algorithms are discussed in detail. A secure remote keyless system using the CMAC mode of AES authentication was implemented for use in small devices. A simple and secure learning feature is used for introducing new transmitters to the receiver. The software used for programming the PIC24FJ128GA010 microcontroller was coded in MPLAB IDE v8.14 C30 compiler. The advantage of SLEEP mode of operation was utilized to reduce the power consumption. The program was successfully tested for correct functionality using an encoder and a decoder device.

No two learned transmitters are assumed to be operated at the same time. All the transmitters are given equal priority. A mechanism to avoid conflicts when two learned transmitters are operating at the same time can be designed by assigning priority to the transmitters.
REFERENCES


APPENDICES
APPENDIX A. MAIN PROGRAM FOR TRANSMITTER

//MAIN.C

#include <p24fj128ga010.h>
#include "message.h"
#include "transmission.h"
#include "communication.h"
#include "eeprom.h"
#include "aes.h"
#include "main.h"

int i = 0;

_CONFIGURATION(JTAGEN_OFF & GCP_OFF & GWRP_OFF & BKBUG_OFF & COE_OFF &
FWDTEN_OFF)
_CONFIGURATION2(FCKSM_CSDCMD & OSCIOFNC_ON & POSCMOD_HS & FNOSC_PRI)

union SharedBlock sharedBlock;
void getsecretKey(void);

void __attribute__ ((interrupt, no_auto_psv)) _CNInterrupt (void){
    _CNIF = 0; // clear external interrupt
    if((~RD6&&~RD7&&~RD13)==0){
        COMMAND_CODE_TYPE lastButtonState;
        COMMAND_CODE_TYPE newButtonState;
        unsigned char equalCount = 0;
        Nop();
        Nop();
        lastButtonState = (PORTD & (0xFFFF));
        do {
            msDelay( 6 ); // Allow for debounce and human slowness.
            newButtonState = PORTD & (0xFFFF);
            if( newButtonState == lastButtonState ) {
                ++equalCount;
            } else {
                equalCount = 0;
                lastButtonState = newButtonState;
            }
        } while( equalCount < 10 ); // 10 is empirical value.
        Nop();
        Nop();
        i = i + 1;
    }
}
COMMAND_CODE_TYPE sleepAndGetCommand(void)
{
    COMMAND_CODE_TYPE buttonState=0;

    // Enable Pin Change Interrupt.
    msDelay(1); // Wait for RF TX ctrl pins to settle.
    _CNIF0 = 1;
    _CNIP1 = 1;
    _CNIP2 = 0;
    _CN15IE = 1;
    _CN16IE = 1;
    _CN19IE = 1;
    _CNIE = 1;
    while(!i); // Wait for Pin-Change Interrupt.
    // Read pins and mask unused inputs according pin change mask.
    if(!_RD6&&_RD7&&_RD13)
        buttonState = (PORTD & (0xFFFF));
    else
        buttonState = 0;
    _CNIE = 0;
    Nop();
    Nop();
    i = 0;
    return buttonState;
}

// initialize switch(D6),switch (D7), switch (D13) as inputs
void initButtons(void)
{
    TRISD = 0;
    PORTD = 0;
    TRISDbits.TRISD6=1;
    TRISDbits.TRISD7=1;
    TRISDbits.TRISD13=1;
}

byte cryptoBlock[BLOCK_SIZE + 1];
byte secretKey[16];

void getsecretKey()
{
    for(i=0;i<16;i++)
    {
        int temp;
        temp=iReadNVM(myAddr+i);
        secretKey[i]=temp;
        i++;
        secretKey[i]=temp>>8;
    }
}

byte sharedKey[16];
void getsharedKey()
{
    for(i=0;i<16;i++)
    {
        int temp;
        temp=iReadNVM(myAddr+24+i);
        sharedKey[i]=temp;
        i++;
        sharedKey[i]=temp>>8;
    }
}

byte block[16];

void getblock()
{
    int i;
    for(i=0;i<16;i++)
    {
        int temp;
        temp=iReadNVM(myAddr+i);
        block[i]=temp;
        i++;
        block[i]=temp>>8;
    }
}

SEQ_COUNTER_TYPE counter;

int Learn=0;

int main(void)
{
    TRISB=0;
    PORTBbits.RB0=0;
    T1CON=0x8020;
    initButtons();
    InitNVM();
    iwriteEEPROM();
    getsecretKey();
    getsBox();
    initializeMessage();//serial number from EEPROM
    int first=0;
    for(;;)
    {
        COMMAND_CODE_TYPE command =sleepAndGetCommand();
        if(command == 0)
            command = sleepAndGetCommand();
        if(command == 0xc0)
{ // Learn Algorithm
    Learn=1;
    getsharedKey();
    getblock(); // obtain block[16] -> secret key
    calcKeySchedule(sharedKey);
    calcCMACSkey(sharedKey);
    cipher(block);
    prepare_transmit1();
    transmit();
}
else
{
    if(first==0)
    {
        calcKeySchedule(secretKey);
        calcCMACSkey(cryptoBlock);
        first+=1;
    }
    updateCounter();
    createMessage(command);
    calcCMAC(const byte *)&sharedBlock.messagePayload.serialNo, cryptoBlock);
    prepare_transmit();
    transmit();
}
//for
return 0 ;
//main

// MAIN.H

void getsecretKey();
void getsharedKey();
void getblock();
eextern int Learn;
eextern byte block[16];
APPENDIX B. AES PROGRAM FOR TRANSMITTER

//AES.C

//Some statements in this function are referred from [3]

#include<p24fj128ga010.h>
#include "message.h"
#include "transmission.h"
#include "eeprom.h"

// Lower 8 bits of AES polynomial (x^8+x^4+x^3+x+1), ie. (x^4+x^3+x+1).
#define BPOLY 0x1b

void getsBox(void);

static const byte keyScheduleInSRAM[ 320 ];

// S-Box lookup table.
static byte sBox[256];

void getsBox(void)
{
    int i;
    for(i=0;i<256;i++)
    {
        int temp;
        temp=iReadNVM(myAddr+40+i);
        sBox[i]=temp;
        i=i+1;
        sBox[i]=temp>>8;
    }
}

void copyBytes( byte * destination, const byte * source, byte count)
{
    // Copy to temporary variables for optimization.
    byte * tempDest = destination;
    const byte * tempSrc = source;
    byte tempCount = count;

    do {
        *tempDest++ = *tempSrc++;
    } while( --tempCount );
}
// Cycle a 4-byte array left once.
static void cycleLeft( byte * row )
{
    byte temp1 = row[0];
    row[0] = row[1];
    row[1] = row[2];
    row[2] = row[3];
    row[3] = temp1;
}

// Perform AES mixcolumn operation on the 'column' buffer.
static void mixColumn( byte * column )
{
    byte result0, result1, result2, result3;
    byte column0, column1, column2, column3;

    column0 = column[0];
    column1 = column[1];
    column2 = column[2];
    column3 = column[3];

    // Partial sums (modular addition using XOR).
    result0 = column1 ^ column2 ^ column3;
    result1 = column0 ^ column2 ^ column3;
    result2 = column0 ^ column1 ^ column3;
    result3 = column0 ^ column1 ^ column2;

    // Multiply column bytes by 2 modulo BPOLY.
    if( column0 & 0x80 ) {
        column0 = (column0 << 1) ^ BPOLY;
    } else {
        column0 = (column0 << 1);
    }
    if( column1 & 0x80 ) {
        column1 = (column1 << 1) ^ BPOLY;
    } else {
        column1 = (column1 << 1);
    }
    if( column2 & 0x80 ) {
        column2 = (column2 << 1) ^ BPOLY;
    } else {
        column2 = (column2 << 1);
    }
    if( column3 & 0x80 ) {
        column3 = (column3 << 1) ^ BPOLY;
    } else {
        column3 = (column3 << 1);
    }
}
// Final sums stored into original column bytes.
column[0] = result0 ^ column0 ^ column1;
column[1] = result1 ^ column1 ^ column2;
column[2] = result2 ^ column2 ^ column3;
column[3] = result3 ^ column0 ^ column3;
}

// Perform AES column mixing for the whole AES block.
static void mixColumns( byte * state )
{
    mixColumn( state + 0*4 );
mixColumn( state + 1*4 );
mixColumn( state + 2*4 );
mixColumn( state + 3*4 );
}

// Substitute 'count' bytes in the 'bytes' buffer using the S-Box.
static void subBytes( byte * bytes, byte count )
{
    byte * tempPtr = bytes;
    byte tempCount = count;
    do {
        *tempPtr = sBox[*tempPtr]; // Substitute every byte in state
        ++tempPtr;
    } while( --tempCount );
}

// Perform AES shiftRows operation for the whole AES block.
static void shiftRows( byte * state )
{
    byte temp;
    // Cycle second row left one time.
    temp = state[ 1 + 0*4 ];
    state[ 1 + 0*4 ] = state[ 1 + 1*4 ];
    state[ 1 + 1*4 ] = state[ 1 + 2*4 ];
    state[ 1 + 2*4 ] = state[ 1 + 3*4 ];
    state[ 1 + 3*4 ] = temp;

    // Cycle third row left two times.
    temp = state[ 2 + 0*4 ];
    state[ 2 + 0*4 ] = state[ 2 + 2*4 ];
    state[ 2 + 2*4 ] = state[ 2 + 3*4 ];
    state[ 2 + 3*4 ] = temp;
    state[ 2 + 3*4 ] = state[ 2 + 1*4 ];
state[ 2 + 3*4 ] = temp;

// Cycle fourth row left three times.
    temp = state[ 3 + 3*4 ];
    state[ 3 + 3*4 ] = state[ 3 + 2*4 ];
    state[ 3 + 2*4 ] = state[ 3 + 1*4 ];
    state[ 3 + 1*4 ] = state[ 3 + 0*4 ];
    state[ 3 + 0*4 ] = temp;

void addConstant( byte * bytes, const byte * constant, byte count )
{
    byte * tempDestination = bytes;
    const byte * tempSource = constant;
    byte tempCount = count;
    byte tempValue;

    do {
        // Add in GF(2), ie. XOR.
        tempValue = *tempDestination ^ *tempSource++;
        *tempDestination++ = tempValue;
    } while( --tempCount );

}

// XOR 'count' bytes from 'constant' buffer into 'bytes' buffer.

static void addConstantAndSubstitute( byte * bytes, const byte * constant, byte count )
{
    // Copy to temporary variables for optimization.
    byte * tempDestination = bytes;
    const byte * tempSource = constant;
    byte tempCount = count;
    byte tempValue;

    do {
        // Add in GF(2), ie. XOR.
        tempValue = *tempDestination ^ *tempSource++;
        *tempDestination++ = sBox[ tempValue ];
    } while( --tempCount );

}

void cipher( byte * block1 )
{
    byte * keySchedule2 = keyScheduleInSRAM;
    byte round = 0;

for( ; round < ROUNDS-1; ++round )
{
    addConstantAndSubstitute( block1, keySchedule2, BLOCK_SIZE );
    shiftRows( block1 );
    mixColumns( block1 );
    keySchedule2 += BLOCK_SIZE;
}

addConstantAndSubstitute( block1, keySchedule2, BLOCK_SIZE );
shiftRows( block1 );
keySchedule2 += BLOCK_SIZE;
addConstant( block1, keySchedule2, BLOCK_SIZE );

void calcKeySchedule( byte * key )
{
    byte schedulePos; // Current position inside schedule.
    byte temp[4]; // Temporary word when expanding the key.
    byte * roundConstant[4] = { 0x01, 0x00, 0x00, 0x00 }; // Copy entire key to start of schedule.
    copyBytes( keyScheduleInSRAM, key, KEY_SIZE );
    keySchedule1 += KEY_SIZE;
    // Copy last 4 bytes of key to temp word.
    copyBytes( temp, keySchedule1+4, 4 );
    // Expand key into schedule buffer 1 first.
    schedulePos = KEY_SIZE;
    while( schedulePos < SCHEDULE_SIZE )
    {
        // Multiple of key size?
        if( (schedulePos % KEY_SIZE) == 0 )
        {
            cycleLeft( temp ); // Cycle left one byte.
            subBytes( temp, 4 ); // Substitute each byte.
            addConstant( temp, roundConstant, 4 );
            // Modular doubling of round constant's first byte.
            if( roundConstant[0] & 0x80 )
            {
                roundConstant[0] <<= 1;
                roundConstant[0] ^= BPOLY;
            }
            else
            {
                roundConstant[0] <<= 1;
            }
        }
    }
    // Add with data KEY_SIZE backwards in schedule.
    addConstant( temp, keySchedule1 - KEY_SIZE, 4 );
    copyBytes( keySchedule1, temp, 4 );
    keySchedule1 += 4;
    schedulePos += 4;
extern void getsBox(void);
APPENDIX C. CMAC PROGRAM FOR TRANSMITTER

//CMAC.C

//Some statements in this function are referred from [3].

#include<p24fj128ga010.h>
#include "message.h"
#include "transmission.h"

static byte CMACSubkey[ CMAC_SUBKEY_SIZE ];

void calcCMACSubkey( byte * cryptoBlock )
{
    byte blockSize;
    byte * blockPtr;
    byte overflow;

    // Fill crypto block with all zeros.
    blockSize = BLOCK_SIZE;
    blockPtr = cryptoBlock;
    do {
        *blockPtr++ = 0;
    } while( --blockSize );

    // Encrypt zeros with secret key (assuming
    // key schedule has been created with secret key)
    cipher( cryptoBlock );

    // Multiply by 2 modulo (0b1^120 0b10000111).
    blockSize = BLOCK_SIZE - 1;
    blockPtr = cryptoBlock;
    overflow = blockPtr[0] & 0x80;
    do {
        blockPtr[0] = (blockPtr[0] << 1) | (blockPtr[1] >> 7);
        ++blockPtr;
    } while( --blockSize );
    blockPtr[0] <<= 1;
    if( overflow ) {
        blockPtr[0] ^= 0x87;
    }

#if MESSAGE_WO_MAC < BLOCK_SIZE
    // Multiply by 2 modulo (0b1^120 cat 0b10000111) again.
    blockSize = BLOCK_SIZE - 1;
    blockPtr = cryptoBlock;
#endif
overflow = blockPtr[0] & 0x80;
do {
    blockPtr[0] = (blockPtr[0] << 1) | (blockPtr[1] >> 7);
    ++blockPtr;
} while ( --blockSize );
blockPtr[0] <<= 1;
if (overflow) {
    blockPtr[0] ^= 0x87;
}
#endif

copyBytes( CMACSubkey, cryptoBlock, BLOCK_SIZE );

void calcCMAC( const byte * message, byte * MACStorage )
{
    // Copy message to MACStorage, which should be BLOCK_SIZE.
    copyBytes( MACStorage, message, MESSAGE_WO_MAC );

    #if MESSAGE_WO_MAC < BLOCK_SIZE
    // Pad message before calculating CMAC.
    #if MESSAGE_WO_MAC + 1 < BLOCK_SIZE
    byte pos = MESSAGE_WO_MAC + 1;
    do {
        MACStorage[ pos ] = 0x00;
    } while ( ++pos < BLOCK_SIZE );
    #endif
    MACStorage[ MESSAGE_WO_MAC ] = 0x80;
    #endif

    // Add (XOR) subkey.
    addConstant( MACStorage, CMACSubkey, BLOCK_SIZE );

    // Encrypt, assuming secret key schedule is precalculated.
    cipher( MACStorage );
}
APPENDIX D. TRANSMISSION PROGRAM FOR TRANSMITTER

//TRANSMISSION.C

#include<p24fj128ga010.h>
#include "message.h"
#include "transmission.h"
#include "eeprom.h"

void initializeMessage(void)
{
    //serial number from EEPROM
    unsigned long int serialNo_lsb, serialNo_msb, serialNo_tot;
    serialNo_lsb = iReadNVM(myAddr+20);
    serialNo_msb = iReadNVM(myAddr+22);
    serialNo_msb = serialNo_msb<<16;
    serialNo_tot = serialNo_msb | serialNo_lsb;
    sharedBlock.messagePayload.serialNo = serialNo_tot; // 0x54545667
}

void createMessage( COMMAND_CODE_TYPE commandCode )
{
    sharedBlock.messagePayload.commandCode = commandCode;
    //counter from eeprom
    sharedBlock.messagePayload.counterValue = counter;
}

void updateCounter(void)
{
    //update counter in eeprom - 4 bytes
    unsigned long int counter_lsb, counter_msb;
    counter_lsb = iReadNVM(myAddr+16);
    counter_msb = iReadNVM(myAddr+18);
    counter_msb = counter_msb<<16;
    counter = counter_msb | counter_lsb;
    if(counter==0xffffffff)
        counter=0;
    else
        counter=counter+1;
    iWriteNVM(myAddr+16, counter);
    iWriteNVM(myAddr+18, counter>>16);
}

void msDelay(unsigned int ms){
    unsigned long i;
    for (i=(unsigned long) (ms+1)*SCALE;i>0;i--) Nop();
}
//TRANSMISSION.H

#ifndef TRANSMISSION_H
#define TRANSMISSION_H

#define SCALE 333L

// Common workspace for encrypting and decrypting.
extern byte cryptoBlock[ BLOCK_SIZE + 1 ];

struct MessagePayload {
    //byte preamble;
    SERIAL_NO_TYPE serialNo;
    SEQ_COUNTER_TYPE counterValue;
    COMMAND_CODE_TYPE commandCode;
};

union SharedBlock {
    struct MessagePayload messagePayload;
    byte cryptoBlock[ BLOCK_SIZE ];
};

extern union SharedBlock sharedBlock;

// Prepare const fields for message payload in 'sharedBlock'.
void initializeMessage(void);
// Prepare non-const fields for message payload in 'sharedBlock'.
void createMessage( COMMAND_CODE_TYPE commandCode );
// Update Counter
void updateCounter(void);
// Milli second delay
void msDelay(unsigned int ms);
#endif
APPENDIX E. EEPROM PROGRAM FOR TRANSMITTER

//EEPROM.C

#include<p24fj128ga010.h>
#include "eeprom.h"

void InitNVM(void)
{
    // init the SPI peripheral
    CSEE = 1; // de-select the Serial EEPROM
    TCSEE = 0; // make SSEE pin output
    SPI2CON1 = SPI_MASTER; // select mode
    SPI2STAT = SPI_ENABLE; // enable the peripheral
}

// send one byte of data and receive one back at the same time
int WriteSPI2( int i)
{
    SPI2BUF = i; // write to buffer for TX
    while( !(SPI2STATbits.SPIRBF)); // wait for transfer to complete
    Nop();
    return SPI2BUF; // read the received value
}

int ReadSR( void)
{
    // Check the Serial EEPROM status register
    int i;
    CSEE = 0; // select the Serial EEPROM
    WriteSPI2( SEE_RDSR); // send a READ STATUS COMMAND
    i = WriteSPI2( 0); // send/receive
    CSEE = 1; // deselect to terminate command
    return i;
}

// read a 16-bit value starting at an even address
int iReadNVM( int address)
{ 
int lsb, msb;
// wait until any work in progress is completed
while ( ReadSR() & 0x03); // check the two lsb WEN and WIP
// perform a 16-bit read sequence (two byte sequential read)
CSEE = 0; // select the Serial EEPROM
WriteSPI2( SEE_READ); // read command
WriteSPI2( address>>8); // address MSB first
WriteSPI2( address & 0xfe); // address LSB (word aligned)
msb = WriteSPI2( 0); // send dummy, read msb
lsb = WriteSPI2( 0); // send dummy, read lsb
CSEE = 1;
return ( (msb<<8)+ lsb);
}//iReadNVM

void WriteEnable( void)
{
  // send a Write Enable command
  CSEE = 0; // select the Serial EEPROM
  WriteSPI2( SEE_WEN); // write enable command
  CSEE = 1; // deselect to complete command
}//WriteEnable

// write a 16-bit value starting at an even address
void iWriteNVM( int address, int data)
{
int lsb, msb;

// wait until any work in progress is completed
while ( ReadSR() & 0x03); // check the two lsb WEN and WIP
// Set the Write Enable Latch
WriteEnable();
// perform a 16-bit write sequence (2 byte page write)
CSEE = 0; // select the Serial EEPROM
WriteSPI2( SEE_WRITE); // write command
WriteSPI2( address>>8); // address MSB first
WriteSPI2( address & 0xfe); // address LSB (word aligned)
WriteSPI2( data >>8); // send msb
WriteSPI2( data & 0xff); // send lsb
CSEE = 1;
}//iWriteNVM

void iwriteEEPROM(void)
{
  // 2-bytes for each location
  // KEY 16-bytes , SYNC 4-bytes , s/n 4-bytes ,
  iWriteNVM( myAddr,KEY_0);
  iWriteNVM( myAddr+2,KEY_1);

68
iWriteNVM( myAddr+4,KEY_2);
iWriteNVM( myAddr+6,KEY_3);
iWriteNVM( myAddr+8,KEY_4);
iWriteNVM( myAddr+10,KEY_5);
iWriteNVM( myAddr+12,KEY_6);
iWriteNVM( myAddr+14,KEY_7);

iWriteNVM( myAddr+16,SYNC_0);
iWriteNVM( myAddr+18,SYNC_1);
// serial number
iWriteNVM( myAddr+20,SNO_0);
iWriteNVM( myAddr+22,SNO_1);
// shared key
iWriteNVM( myAddr+24,SHK_0);
iWriteNVM( myAddr+26,SHK_1);
iWriteNVM( myAddr+28,SHK_2);
iWriteNVM( myAddr+30,SHK_3);
iWriteNVM( myAddr+32,SHK_4);
iWriteNVM( myAddr+34,SHK_5);
iWriteNVM( myAddr+36,SHK_6);
iWriteNVM( myAddr+38,SHK_7);
//sbox
writesBox();
}

void writesBox(void)
{
    int i;
    for(i=0;i<256;i++)
    {
        int temp1,temp2,temp;
        temp1=sBoxeeprom[i];
        i=i+1;
        temp2=sBoxeeprom[i];
        temp2=temp2<<8;
        temp=temp1|temp2;
        iWriteNVM( myAddr+38+i+1,temp);
    }
}

//EEPROM.H

#include "message.h"
define TRUE 1
define myAddr 0x0000

//Shared Key
define SHK_0 0x3a7f
define SHK_1 0x2d8a
define SHK_2 0x508f
define SHK_3 0xcb1c
#define SHK_4 0xcab9
#define SHK_5 0x72b7
#define SHK_6 0x8f7b
#define SHK_7 0x076b

#define KEY_0 0x7E2B
#define KEY_1 0x1615
#define KEY_2 0xAE28
#define KEY_3 0xA6D2
#define KEY_4 0xF7AB
#define KEY_5 0x8815
#define KEY_6 0xCF09
#define KEY_7 0x3C4F

// Synchronization Counter
#define SYNC_0 0x0000
#define SYNC_1 0x0000

// Serial Number
#define SNO_0 0x5667
#define SNO_1 0x5454

// 25LC256 Serial EEPROM commands

// I/O definitions for PIC24 + Explorer16 demo board
#define CSEE_RD12 // select line for Serial EEPROM
#define TCSEE_TRISD12 // tris control for CSEE pin

// peripheral configurations
#define SPI_MASTER 0x0122 // select 8-bit master mode, CKE=1, CKP=0, 4MHz
#define SPI_ENABLE 0x8000 // enable SPI port, clear status

#define SEE_WRSR 1 // write status register
#define SEE_WRITE2 // write command
#define SEE_READ 3 // read command
#define SEE_WDI 4 // write disable
#define SEE_RDSR 5 // read status register
#define SEE_WEN 6 // write enable

static const byte sBoxeeprom[256] = {
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
  0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
  0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,
  0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,
  0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
void InitNVM(void);
int WriteSPI2( int i);
int ReadSR( void);
int iReadNVM( int address);
void WriteEnable( void);
void iWriteNVM( int address, int data);
void iwriteEEPROM(void);
void writesBox(void);

0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
0x52, 0x3b, 0xdd, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
0x53, 0x11, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa0,
APPENDIX F. COMMUNICATION PROGRAM FOR TRANSMITTER

//COMMUNICATION.C

#include<p24fj128ga010.h>
#include "communication.h"
#include "message.h"
#include "transmission.h"
#include "main.h"
byte r[161];

void prepare_transmit()
{
    byte mask, temp;
    int i, j, p;
    p=0;
    //MAC
    for(j=0; j<4; j++)
    {
        temp=cryptoBlock[j];
        mask=0b00000001;
        for(i=0; i<8; i++)
        {
            r[p]=(temp&mask)>>i;
            p=p+1;
            mask=mask<<1;
        }
    }
    // Command
    temp=sharedBlock.cryptoBlock[8];
    mask=0b00000001;
    for(i=0; i<8; i++)
    {
        r[p]=(temp&mask)>>i;
        p=p+1;
        mask=mask<<1;
    }
    //counter value
    for(j=4; j<8; j++)
    {
        temp=sharedBlock.cryptoBlock[j];
        mask=0b00000001;
        for(i=0; i<8; i++)
        {
            r[p]=(temp&mask)>>i;
            p=p+1;
        }
    }
}
void prepare_transmit1(void)
{
    // LSB -> MSB { [0] -> [max] }
    // SNO+ENCRYPTED SECRET KEY -> 20 bytes

    byte mask, temp;
    int i, j, p;
    p = 0;
    for (j = 0; j < 16; j++)
    {
        temp = block[j];
        mask = 0b00000001;
        for (i = 0; i < 8; i++)
        {
            r[p] = (temp & mask) >> i;
            p = p + 1;
            mask = mask << 1;
        }
    }
    // serial number
    for (j = 0; j < 5; j++)
    {
        temp = sharedBlock.cryptoBlock[j];
        mask = 0b00000001;
        for (i = 0; i < 8; i++)
        {
            r[p] = (temp & mask) >> i;
            p = p + 1;
            mask = mask << 1;
        }
    }
}

mask = mask << 1;
}

// serial number
for (j = 0; j < 4; j++)
{
    temp = sharedBlock.cryptoBlock[j];
    mask = 0b00000001;
    for (i = 0; i < 8; i++)
    {
        r[p] = (temp & mask) >> i;
        p = p + 1;
        mask = mask << 1;
    }
}

// LSB -> MSB { [0] -> [max] }
// SNO+ENCRYPTED SECRET KEY -> 20 bytes

byte mask, temp;
int i, j, p;
p = 0;
for (j = 0; j < 16; j++)
{
    temp = block[j];
    mask = 0b00000001;
    for (i = 0; i < 8; i++)
    {
        r[p] = (temp & mask) >> i;
        p = p + 1;
        mask = mask << 1;
    }
}

// serial number
for (j = 0; j < 5; j++)
{
    temp = sharedBlock.cryptoBlock[j];
    mask = 0b00000001;
    for (i = 0; i < 8; i++)
    {
        r[p] = (temp & mask) >> i;
        p = p + 1;
        mask = mask << 1;
    }
}
void transmit(void)
{
    int i, LENGTH;
    if(Learn == 1)
        LENGTH=161;
    else
        LENGTH=105;
    for(i=0; i<4; i++)
    {
        // preamble
        int k;
        for(k=0; k<12; k++)
        {
            PORTBbits.RB0=1;
            TMR1=0;
            while(TMR1<DELAY2)
            {
            
            }
            // if(k==11)
            // break;
            PORTBbits.RB0=0;
            TMR1=0;
            while(TMR1<DELAY2)
            {
            
            }
        }
    }
    // header
    PORTBbits.RB0=0;
    TMR1=0;
    while(TMR1<DELAY2*9)
    {
    
    }
    // encrypted fixed
    for(k=0; k<LENGTH; k++) // no of bits to be transmitted : 104/160
    {
        if(r[k]==0)
        {
            PORTBbits.RB0=1;
            TMR1=0;
            while(TMR1<DELAY1)
            {
            }
            PORTBbits.RB0=0;
            TMR1=0;
            while(TMR1<DELAY2)
            {
            }
        }
        else
        {=}
PORTBbits.RB0=1;
TMR1=0;
while(TMR1<DELAY2)
{
}
PORTBbits.RB0=0;
TMR1=0;
while(TMR1<DELAY1)
{
}
}
//guard
PORTBbits.RB0=0;
TMR1=0;
while(TMR1<DELAY2*39)
{
}
}

COMMUNICATION.H

#define DELAY1 50
#define DELAY2 25

void prepare_transmit(void);
void prepare_transmit1(void);
void transmit(void);
APPENDIX G. MESSAGE PROGRAM FOR TRANSMITTER

//MESSAGE.H

#ifndef MESSAGE_H
#define MESSAGE_H

typedef signed char int8_t;
typedef unsigned char uint8_t;
typedef short int16_t;
typedef unsigned short uint16_t;
typedef long int32_t;
typedef unsigned long uint32_t;
typedef uint8_t byte;

#define SERIAL_NO_BYTES 4
#define COMMAND_BYTES 1
#define COUNTER_BYTES 4
#define MESSAGE_WO_MAC (SERIAL_NO_BYTES+COMMAND_BYTES+COUNTER_BYTES)
#define SERIAL_NO_TYPE uint32_t
#define SERIAL_NO_MAX 0xffffffff
#define SEQ_COUNTER_TYPE uint32_t
#define COMMAND_CODE_TYPE byte
#define COMMAND_CODE_MAX 0xff
#define BLOCK_SIZE 16
#define ROUNDS 10
#define KEY_SIZE 16
#define SCHEDULE_SIZE_BLOCKS (ROUNDS+1)
#define SCHEDULE_SIZE (SCHEDULE_SIZE_BLOCKS*BLOCK_SIZE)
#define CMAC_SUBKEY_SIZE BLOCK_SIZE

extern SEQ_COUNTER_TYPE counter;

// Copy 'count' bytes from 'source' buffer to 'destination' buffer
void copyBytes( byte * destination, const byte * source, byte count);

// XOR 'count' bytes from 'constant' buffer into 'bytes' buffer
void addConstant( byte * bytes, const byte * constant, byte count );

//Calculates Key Schedule for given key
void calcKeySchedule( byte * key );

//uses key schedule to encode the data
void cipher( byte * block1 );
//uses key schedule to decode the data
void invCipher( byte * block1 );

//Calculates CMACSubkey
void calcCMACSubkey( byte * cryptoBlock );

//Calculates CMAC using CMAC Subkey
void calcCMAC( const byte * message, byte * MACStorage );

#endif
APPENDIX H. MAIN PROGRAM FOR RECEIVER

//MAIN.C

// Some statements in this function are referred from [3]

#include<p24fj128ga010.h>
#include "communication.h"
#include "transmission.h"
#include "eeprom.h"
#include "aes.h"
#include "main.h"
#include <string.h>

_CONFIG1(JTAGEN_OFF & GCP_OFF & GWRP_OFF & BKBUG_OFF & COE_OFF &
FWDTEN_OFF)
_CONFIG2(FCKSM_CSDCMD & OSCIOFNC_ON & POSCMOD_HS & FNOSC_PRI)

word Dato;
word Ind;
byte B[20];
byte Output[3];
byte messageBlock[20];
SEQ_COUNTER_TYPE counter;
int NBIT=104;
int LENGTH=13;

union SharedBlock sharedBlock;

byte secretKey[16];
byte sharedKey[16];
byte Block[16];

void getsecretKey()
{
  int i;
  for(i=0;i<16;i++)
  {
    int temp;
    temp=iReadNVM(myAddr+Ind+10+i);
    secretKey[i]=temp;
    i++;
    secretKey[i]=temp>>8;
  }
}
void getsharedKey()
{
    int i;
    for(i=0;i<16;i++)
    {
        int temp;
        temp=iReadNVM(myAddr+122+i);
        sharedKey[i]=temp;
        i++;
        sharedKey[i]=temp>>8;
    }
}

void KeyUpdate()
{
    //update extracted key in learn mode to eeprom
    int i;
    word temp;
    for(i=15;i>=0;i--)
    {
        temp=Block[i];
        temp=temp<<8;
        i--;
        temp=temp|Block[i];
        iWriteNVM(myAddr+Ind+10+i,temp);
    }
}

void counterUpdate()
{
    //update the counter value in the eeprom
    word data;
    data=((word)HopMi1<<8)|HopLo;
    iWriteNVM(myAddr+Ind+6,data);
    data=((word)HopHi<<8)|HopMi2;
    iWriteNVM(myAddr+Ind+8,data);
}

void functionupdate()
{
    word temp;
    temp=messageBlock[4];
    temp=(temp<<8);
    iWriteNVM(myAddr+Ind+0,temp);
}

int compareMAC(void)
{
    //compare constructed MAC with that of received buffer
    if((cryptoBlock[0]!=messageBlock[12])|(cryptoBlock[1]!=messageBlock[11])|(cryptoBlock[2]!=messageBlock[10])|(cryptoBlock[3]!=messageBlock[9]))
return 1;
}
else
return 0;
}

void performfunction(void)
{
byte temp=messageBlock[4];
if(temp==0x80)
Output[0]=1;
if(temp==0x40)
Output[1]=1;
if(temp==0xc0) //require keypad
Output[2]=1;
}

void __attribute__((interrupt,no_auto_psv)) _T1Interrupt (void)
{
// this routine gets called every time TMR1 overflows
RFBit = RFIn; // sampling RF pin verify
TMR1 -= PERIOD; // reload
_T1IF = 0;
XTMR++; // extended 16 long timer update
if (RFFull) // avoid overrun
return;

switch( RFstate) // state machine main switch
{
  case TRFUNO:
    if ( RFBit == 0) // falling edge detected
    {
    //
    //
    //
    //
    RFstate= TRFZERO;
    }
    else
    {
    // while high
    RFcount--;
    if ( RFcount < HIGH_TO)
    RFstate = TRFreset; // reset if too long
    }
    break;

  case TRFZERO:
    if ( RFBit)
    {
    // rising edge detected
    //
    //
    RFstate = TRFreset;
    }
    break;

  case TRFONE:
    if ( RFBit) // rising edge detected
    {
    //
    //
    RFstate = TRFreset;
    }
    break;

  case TRFONE:
    if ( RFBit) // falling edge detected
    {
    //
    //
    RFstate = TRFreset;
    }
    break;

  case TRZFIVE:
  // This case is not shown in the above switch statements.
  break;
}
}
RFstate = TRFUNO;
B[Bptr] >>= 1; // rotate
if (RFcount >= 0)
{
    B[Bptr] = 0x80; // shift in bit
}
RFcount = 0; // reset length counter

if ( ( ++BitCount & 7) == 0)
    Bptr++; // advance one byte
if (BitCount == NBIT)
{
    RFstate = TRFreset; // finished receiving
    RFFull = TRUE;
}
else
{
    RFcount++;
    if (RFcount >= LOW_TO) // too long low
    {
        RFstate = TRFSYNC; // RFSYNC state
        Bptr = 0; // reset pointers, while keep counting on
        BitCount = 0;
    }
}
break;

case TRFSYNC:
    if (RFBit)
    {
        // rising edge detected +----+ +----+
        //       |    <-Theader->    |
        //              +----------+
        //
        if ( (RFcount < SHORT_HEAD) || (RFcount >= LONG_HEAD))
        {
            RFstate = TRFreset;
            break; // too short/long, no header
        }
    else
    {
        RFcount = 0; // restart counter
        RFstate = TRFUNO;
    }
}
else
{
    // still low
    RFcount++;
}
break;

case TRFreset:
default:
RFstate = TRFSYNC; // reset state machine in all other cases
RFcount = 0;
Bptr = 0;
BitCount = 0;
break;
}
}

void InitReceiver()
{
    _T1IP=4;
    TMR1=0;
    PR1=PERIOD-1;
    T1CON=0x8030; // 1:256 prescaler
    _T1IF=0;
    _T1IE=1;
    RFstate = TRFreset; // reset state machine in all other cases
    RFFull = 0; // start with buffer empty
    XTMR = 0; // start extended timer
}

void Remote()
{
    // A frame was received and is stored in the receive buffer
    // Move it to decryption Buffer, and restart receiving

    memcpy( messageBlock,B,LENGTH);
    RFFull = FALSE; // ready to receive a new frame

    if ( FLearn)
    {
        // Learn Mode

        if ( Find()== FALSE)
        { // could not find the Serial Number in memory
            if ( !Insert()) // look for new space
                return; // fail if no memory available
        }

        // ASSERT Ind is pointing to a valid memory location
        IDWrite(); // write Serial Number+button in memory
        HopLearn(); // write Synchronization counter in memory
        memcpy( Block,B,16); // decrypt for secret key.
        getsharedKey();
        calcKeySchedule(sharedKey);
        invCipher(Block);//16 bytes of received data
        KeyUpdate(); // write secret key in memory from Block
        CFlash = 32; // request Led flashing
        CTFlash = TFLASH; // load period timer
        Led = TRUE; // start with Led on
    }
FLearn = FALSE; // terminate successfully Learn
// change length of buffer and number of bits to receive
LENGTH=13;
NBIT=104; //number of bits=1
}
// Learn
else // Normal Mode of operation
{
// look for s/n in EEPROM
if ( Find()== FALSE)
    return;
if(!HopCHK()) //check for Hopping code integrity
    return;
getsecretKey(); //secret key form the table
initializeMessage(); // serial no from received message
updateCommand();
updateCounter(); //counter value update
calcKeySchedule(secretKey);
calcCMACSubkey(cryptoBlock);
createmessage(command);
calcCMAC( (const byte *)
    &sharedBlock.messagePayload.serialNo,cryptoBlock );
if(compareMAC()==TRUE)
{
    counterUpdate(); //update counter in eeprom
    functionupdate(); //update command in eeprom
    performfunction(); //perform requested function
}
}

int main(void)
{
    TRISA = MASKPA; // set i/o config.
    PORTA = 0;
    TRISDbits.TRISD6=1;
    PORTD = 0;
    InitReceiver(); // enable and init the receiver state machine
    InitNVM();
    iwriteEEPROM();
    ClearMem(); // clear the eeprom space for table of records
    getsBox(); // sbox from eeprom
    getsBoxinv(); // invsbox from eeprom
    // main loop
    while ( TRUE)
    {
        if ( RFFull) // buffer contains a message
        {
            Remote();
        }
        // loop waiting 512* period = 72ms
        if ( XTMR < 512)
            continue; // main loop
        // once every 72ms

XTMR=0;

// re-init fundamental registers
AD1CON1 = 0xffff; // disable analog inputs
TRISA = MASKPA; // set i/o config.
TRISB = MASKPB;
TRISC = MASKPC;
_T1IE = 1;

// poll learn
if (!RD6) //(!Learn) // low -> button pressed
{
    // change length of buffer and number of bits to receive
    LENGTH=20;
    NBIT=160; // number of bits-1
    CLearn++;
    // pressing Learn button for more than 10s -> ERASE ALL
    if (CLearn == 128) // 128 * 72 ms = 10s
    {
        Led = OFF; // switch off Learn Led
        while (!Learn); // wait for button release
        Led = ON; // signal Led on
        ClearMem(); // erase all command!
        COut = TOUT; // single long flash pulse time
        // timer will switch off Led
        CLearn = 0; // reset learn debounce
        FLearn = FALSE; // exit learn mode
    }
}

// normal Learn button debounce
if (CLearn == 4) // 250ms debounce
{
    FLearn = TRUE; // enter learn mode command!
    CTTLEARN = TLEARN; // load timeout value
    Led = ON; // turn Led on
}
else CLearn=0; // reset counter

// outputs timing
if (COut > 0) // if timer running
{
    COut--;
    if (COut == 0) // when it reach 0
    {
        Led = OFF; // all outputs off
        int i;
        for(i=0;i<4;i++)
            OUT[i] = OFF;
    }
}

// Learn Mode timeout after 18s (TLEARN * 72ms)
if ( CTLearn > 0)
{
    CTLearn--;                    // count down
    if ( CTLearn == 0)           // if timed out
    {
        Led = OFF;                // exit Learn mode
        FLearn = FALSE;
    }
}

// Led Flashing
if ( CFlash > 0)
{
    CTFlash--;                  // count down
    if ( CTFlash == 0)          // if timed out
    {
        CTFlash = TFLASH;        // reload timer
        CFlash--;                // count one flash
        Led = OFF;               // toggle Led
        if ( CFlash & 1)
        
            Led = ON;
    }
}

return 0;
}

// FIND Routine
//
// search through the whole table the given a record whose ID match
//
// INPUT:
//  IDHi, IDMi2, IDMi1, IDLo,   serial number to search
//
// OUTPUT:
//  Ind                 address of record (if found)
//  EHop                sync counter value
//  ETemp               second copy of sync counter
// RETURN:
//  TRUE if matching record found
byte Find()
{
    byte Found;
    Found = FALSE;                // init to not found

    for (Ind=0; Ind < (EL_SIZE * MAX_USER); Ind+=EL_SIZE)
    {
        RDword( Ind);              // read first Word
        FCode = (Dato>>8);          // check if 1111xxxx
        if ( (FCode & 0xf0) == 0xf0)
continue;  // empty
RDWord( Ind+2);
if ( (IDLo != (Dato & 0xff)) || ((IDMi1 != (Dato>>8))) )
    continue;  // fails match
RDWord( Ind+4);  // read next word
if ( ( (Dato & 0xff) == IDMi2) && ( (Dato>>8) == IDHi))
{
    Found = TRUE;  // match
    break;
}
}  // for

if (Found == TRUE)
{
    RDWord( Ind+6);  // read HopMi1/Lo
    EHop1 = Dato;
    RDWord( Ind+8);  // read HopHi/Mi2
    EHop2=Dato;
    EHop=((uint32_t)EHop2<<16)+EHop1;
}

return Found;

//INSERT Routine
//search through the whole table for an empty space
//INPUT:  
//  IDHi,IDMi2, IDMi1, IDLo,   serial number to insert
//OUTPUT: 
//  Ind                 address of empty record
//RETURN:               FALSE if no empty space found
byte Insert()
{
    for (Ind=0; Ind < (EL_SIZE * MAX_USER); Ind+=EL_SIZE)
    {
        RDword(Ind);     // read first Word
        FCode = (Dato>>8);
        // check if 1111xxxx
        if ( (FCode & 0xf0) == 0xf0)
            return TRUE;  // insert point found
    }  // for

    return FALSE;  // could not find any empty slot
}  // Insert

//Function IDWrite
//  store IDHi,Mi2,Mi1,Lo + XF at current address Ind
//INPUT:  
//  Ind                 point to record + offset 0
//  IDHi, IDMi, IDLo    Serial Number
//  XF                  function code
//OUTPUT:  

byte IDWrite()
{
    if (!FLearn) return FALSE;  //Guard statement: check if Learn ON
    Dato =0xc0;  //write button info- but button info is not
    //not transmitted so use a fixed value 0xc0
    WRword(Ind+0);
    Dato = IDMi1;
    Dato = (Dato<<8) + IDLo;
    WRword(Ind+2);  // write first word
    Dato = IDHi;
    Dato = (Dato<<8) + IDMi2;
    WRword(Ind+4);  // write second word

    return TRUE;
} // IDWrite

//Function HopUpdate
// update sync counter of user record at current location
//INPUT:
//  Ind     record + offset 0
//  Hop     current sync counter
//OUTPUT:
//  none

byte HopUpdate()
{
    if (!FHopOK) return FALSE;  // Guard statement: check if Hop update
    Hop = ((word)HopMi1<<8) + HopLo;
    Dato = Hop;
    WRword(Ind+6);  // write at offset +4
    Hop = ((word)HopHi<<8) + HopMi2;
    Dato = Hop;
    WRword(Ind+8);  // back up copy at offset +6
    FHopOK = FALSE;  // disable updating hopping code
    return TRUE;
} // HopUpdate

byte HopCHK()
{
    FHopOK = FALSE;  // Hopping Code is not verified yet
    FSame = FALSE;  // Hop Code is not the same as previous

    // make it a 32 bit signed integer
    Hop =((uint32_t)HopHi<<24)+((uint32_t)HopMi2<<16)+((uint32_t)
        HopMi1<<8)+HopLo;

    if ( F2Chance)
        if ( NextHop == Hop)
{  
    F2Chance = FALSE; // resync success
    FHopOK = TRUE;
    return TRUE;
}

// main comparison
ETemp = Hop - EHop; // subtract last value from new one
if ( ETemp < 0) // locked region
    return FALSE; // fail
else if ( ETemp > 16) // resync region
    return ReqResync();
else // 0>= ETemp >16 ; // open window
{
    if ( ETemp == 0) // same code (ETemp == 0)
        FSame = TRUE; // rise a flag
        FHopOK = TRUE;
        return TRUE;
    }
} // HopCHK

byte ReqResync()
{
    F2Chance = TRUE; // flag that a second (sequential) transmission
    NextHop = Hop+1; // is needed to resync receiver
    return FALSE; // cannot accept for now
}

//Function - clear memory
void ClearMem(void)
{
    word i;
    for (i=0; i < (EL_SIZE * MAX_USER); i+=EL_SIZE)
    {
        Dato = 0xffff;
        WRword( i);
    }
} // ClearMem

//function - update hopping code in learn mode
void HopLearn(void)
{
    Dato=0x0000;
    WRword(Ind+6);
    WRword(Ind+8);
    FHopOK = TRUE; // enable updating Hopping Code
}

//MAIN.H

extern byte messageBlock[20];
extern byte Buffer[13];
extern word Dato;
extern word Ind;
int compareMAC(void);
extern SEQ_COUNTER_TYPE counter;
void getsharedKey(void);
void KeyUpdate(void);
void getsecretKey(void);
void counterUpdate(void);
void functionupdate(void);
int compareMAC(void);
void performfunction(void);
APPENDIX I. AES PROGRAM FOR RECEIVER

//AES.C

// Some statements in this function are referred from [3]

#include<p24fj128ga010.h>
#include "message.h"
#include "transmission.h"
#include "eeprom.h"

// Lower 8 bits of AES polynomial (x^8+x^4+x^3+x+1) ->(x^4+x^3+x+1).
#define BPOLY 0x1b
static const byte keyScheduleInSRAM[320];

// S-Box and invS-Box lookup tables.
static byte sBox[256];
static byte sBoxInv[256];

void getsBox(void)
{
   int i;
   for(i=0;i<256;i++)
   {
      int temp;
      temp=iReadNVM(myAddr+138+i);
      sBox[i]=temp;
      i=i+1;
      sBox[i]=temp>>8;
   }
}

void getsBoxInv(void)
{
   int i;
   for(i=0;i<256;i++)
   {
      int temp;
      temp=iReadNVM(myAddr+398+i);
      sBoxInv[i]=temp;
      i=i+1;
      sBoxInv[i]=temp>>8;
   }
}
void copyBytes( byte * destination, const byte * source, byte count)
{
    // Copy to temporary variables for optimization.
    byte * tempDest = destination;
    const byte * tempSrc = source;
    byte tempCount = count;
    do {
        *tempDest++ = *tempSrc++;
    } while( --tempCount );
}

// Cycle a 4-byte array left once.
static void cycleLeft( byte * row )
{
    byte temp1 = row[0];
    row[0] = row[1];
    row[1] = row[2];
    row[2] = row[3];
    row[3] = temp1;
}

// AES column mix operation on the 4 bytes in 'column' buffer.
static void mixColumn( byte * column )
{
    byte result0, result1, result2, result3;
    byte column0, column1, column2, column3;
    column0 = column[0];
    column1 = column[1];
    column2 = column[2];
    column3 = column[3];

    // Partial sums (modular addition using XOR).
    result0 = column1 ^ column2 ^ column3;
    result1 = column0 ^ column2 ^ column3;
    result2 = column0 ^ column1 ^ column3;
    result3 = column0 ^ column1 ^ column2;

    // Multiply column bytes by 2 modulo BPOLY.
    if( column0 & 0x80 ) {
        column0 = (column0 << 1) ^ BPOLY;
    } else {
        column0 = (column0 << 1);
    }
    if( column1 & 0x80 ) {
        column1 = (column1 << 1) ^ BPOLY;
    } else {
        column1 = (column1 << 1);
    }
    if( column2 & 0x80 ) {
        column2 = (column2 << 1) ^ BPOLY;
    } else {
        column2 = (column2 << 1);
    }
    if( column3 & 0x80 ) {
        column3 = (column3 << 1) ^ BPOLY;
    } else {
        column3 = (column3 << 1);
    }
}
if( column2 & 0x80 ) {
    column2 = (column2 << 1) ^ BPOLY;
} else {
    column2 = (column2 << 1);
}
if( column3 & 0x80 ) {
    column3 = (column3 << 1) ^ BPOLY;
} else {
    column3 = (column3 << 1);
}

// Final sums stored into original column bytes.
column[0] = result0 ^ column0 ^ column1;
column[1] = result1 ^ column1 ^ column2;
column[2] = result2 ^ column2 ^ column3;
column[3] = result3 ^ column0 ^ column3;
}

// Perform AES column mixing for the whole AES block.
static void mixColumns( byte * state )
{
    mixColumn( state + 0*4 );
mixColumn( state + 1*4 );
mixColumn( state + 2*4 );
mixColumn( state + 3*4 );
}

// Substitute 'count' bytes in the 'bytes' buffer using the S-Box.
static void subBytes( byte * bytes, byte count )
{
    // Copy to temporary variables for optimization.
    byte * tempPtr = bytes;
    byte tempCount = count;

do {
    *tempPtr = sBox[ *tempPtr ]; // Sub every byte in state.
    ++tempPtr;
} while( --tempCount );
}

// Perform AES shift rows operation for the whole AES block.
static void shiftRows( byte * state )
{
    byte temp;

    // Cycle second row left one time.
    temp = state[ 1 + 0*4 ];
    state[ 1 + 0*4 ] = state[ 1 + 1*4 ];
    state[ 1 + 1*4 ] = state[ 1 + 2*4 ];
    state[ 1 + 2*4 ] = state[ 1 + 3*4 ];
state[1 + 3*4] = temp;

// Cycle third row left two times.
temp = state[2 + 0*4];
state[2 + 0*4] = state[2 + 2*4];
state[2 + 2*4] = temp;
state[2 + 1*4] = state[2 + 3*4];
state[2 + 3*4] = temp;

// Cycle fourth row left three times, ie. right once.
temp = state[3 + 3*4];
state[3 + 3*4] = state[3 + 2*4];
state[3 + 2*4] = state[3 + 1*4];
state[3 + 1*4] = state[3 + 0*4];
state[3 + 0*4] = temp;

void addConstant( byte * bytes, const byte * constant, byte count )
{
    // Copy to temporary variables for optimization.
    byte * tempDestination = bytes;
    const byte * tempSource = constant;
    byte tempCount = count;
    byte tempValue;

    do {
        // Add in GF(2), ie. XOR.
        tempValue = *tempDestination ^ *tempSource++;
        *tempDestination++ = tempValue;
    } while( --tempCount );
}

static void addConstantAndSubstitute( byte * bytes, const byte * constant, byte count )
{
    // Copy to temporary variables for optimization.
    byte * tempDestination = bytes;
    const byte * tempSource = constant;
    byte tempCount = count;
    byte tempValue;

    do {
        // Add in GF(2), ie. XOR.
        tempValue = *tempDestination ^ *tempSource++;
        *tempDestination++ = sBox[tempValue];
    } while( --tempCount );
}
void cipher( byte * block1 )
{
    byte * keySchedule2 = keyScheduleInSRAM;
    byte round = 0;

    for( ; round < ROUNDS-1; ++round ) {
        addConstantAndSubstitute( block1, keySchedule2, BLOCK_SIZE );
        shiftRows( block1 );
        mixColumns( block1 );
        keySchedule2 += BLOCK_SIZE;
    }

    addConstantAndSubstitute( block1, keySchedule2, BLOCK_SIZE );
    shiftRows( block1 );
    keySchedule2 += BLOCK_SIZE;
    addConstant( block1, keySchedule2, BLOCK_SIZE );
}

void calcKeySchedule( byte * key )
{
    byte schedulePos; // Current position inside schedule.
    byte temp[4]; // Temporary word when expanding the key.
    byte roundConstant[4] = { 0x01, 0x00, 0x00, 0x00 };  
    byte * keySchedule1 = keyScheduleInSRAM;
    // Copy entire key to start of schedule.
    copyBytes( keySchedule1, key, KEY_SIZE );
    keySchedule1 += KEY_SIZE;
    // Copy last 4 bytes of key to temp word.
    copyBytes( temp, keySchedule1-4, 4 );
    // Expand key into schedule buffer 1 first.
    schedulePos = KEY_SIZE;
    while( schedulePos < SCHEDULE_SIZE ) {
        // Multiple of key size?
        if( (schedulePos % KEY_SIZE) == 0 ) {
            cycleLeft( temp ); // Cycle left one byte.
            subBytes( temp, 4 ); // Substitute each byte.
            addConstant( temp, roundConstant, 4 );
            // Modular doubling of round constant's first byte.
            if( roundConstant[0] & 0x80 ) {
                roundConstant[0] <<= 1;
                roundConstant[0] ^= BPOLY;
            } else {
                roundConstant[0] <<= 1;
            }
        }
    }

    // Add with data KEY_SIZE backwards in schedule.
    addConstant( temp, keySchedule1 - KEY_SIZE, 4 );
}
copyBytes( keySchedule1, temp, 4 );
keySchedule1 += 4;
schedulePos += 4;
}

// Inverse AES

static void invMixColumn( byte * column )
{
    byte result0, result1, result2, result3;
    byte column0, column1, column2, column3;

    column0 = column[0];
    column1 = column[1];
    column2 = column[2];
    column3 = column[3];

    // Partial sums (modular addition using XOR).
    result0 = column1 ^ column2 ^ column3;
    result1 = column0 ^ column2 ^ column3;
    result2 = column0 ^ column1 ^ column3;
    result3 = column0 ^ column1 ^ column2;

    // Multiply column bytes by 2 modulo BPOLY.
    if( column0 & 0x80 ) {
        column0 = (column0 << 1) ^ BPOLY;
    } else {
        column0 = (column0 << 1);
    }
    if( column1 & 0x80 ) {
        column1 = (column1 << 1) ^ BPOLY;
    } else {
        column1 = (column1 << 1);
    }
    if( column2 & 0x80 ) {
        column2 = (column2 << 1) ^ BPOLY;
    } else {
        column2 = (column2 << 1);
    }
    if( column3 & 0x80 ) {
        column3 = (column3 << 1) ^ BPOLY;
    } else {
        column3 = (column3 << 1);
    }

    // More partial sums.
    result0 ^= column0 ^ column1;
    result1 ^= column1 ^ column2;
    result2 ^= column2 ^ column3;
    result3 ^= column0 ^ column3;

    // Multiply by 2.
if( column0 & 0x80 ) {
    column0 = (column0 << 1) ^ BPOLY;
} else {
    column0 = (column0 << 1);
}
if( column1 & 0x80 ) {
    column1 = (column1 << 1) ^ BPOLY;
} else {
    column1 = (column1 << 1);
}
if( column2 & 0x80 ) {
    column2 = (column2 << 1) ^ BPOLY;
} else {
    column2 = (column2 << 1);
}
if( column3 & 0x80 ) {
    column3 = (column3 << 1) ^ BPOLY;
} else {
    column3 = (column3 << 1);
}

// More partial sums.
result0 ^= column0 ^ column2;
result1 ^= column1 ^ column3;
result2 ^= column0 ^ column2;
result3 ^= column1 ^ column3;

// Multiply by 2.
if( column0 & 0x80 ) {
    column0 = (column0 << 1) ^ BPOLY;
} else {
    column0 = (column0 << 1);
}
if( column1 & 0x80 ) {
    column1 = (column1 << 1) ^ BPOLY;
} else {
    column1 = (column1 << 1);
}
if( column2 & 0x80 ) {
    column2 = (column2 << 1) ^ BPOLY;
} else {
    column2 = (column2 << 1);
}
if( column3 & 0x80 ) {
    column3 = (column3 << 1) ^ BPOLY;
} else {
    column3 = (column3 << 1);
}

// Final partial sum.
column0 ^= column1 ^ column2 ^ column3;

// Final sums stored in to original column bytes.
column[0] = result0 ^ column0;
column[1] = result1 ^ column0;
column[2] = result2 ^ column0;
column[3] = result3 ^ column0;
}

static void invMixColumns( byte * state )
{
    invMixColumn( state + 0*4 );
    invMixColumn( state + 1*4 );
    invMixColumn( state + 2*4 );
    invMixColumn( state + 3*4 );
}

// Perform AES inverse shift rows operation for the whole AES block.
static void invShiftRows( byte * state )
{
    byte temp;

    // Cycle second row right one time.
    temp = state[ 1 + 3*4 ];
    state[ 1 + 3*4 ] = state[ 1 + 2*4 ];
    state[ 1 + 2*4 ] = state[ 1 + 1*4 ];
    state[ 1 + 1*4 ] = state[ 1 + 0*4 ];
    state[ 1 + 0*4 ] = temp;

    // Cycle third row right two times.
    temp = state[ 2 + 0*4 ];
    state[ 2 + 0*4 ] = state[ 2 + 2*4 ];
    state[ 2 + 2*4 ] = temp;
    temp = state[ 2 + 1*4 ];
    state[ 2 + 1*4 ] = state[ 2 + 3*4 ];
    state[ 2 + 3*4 ] = temp;

    // Cycle fourth row right three times.
    temp = state[ 3 + 0*4 ];
    state[ 3 + 0*4 ] = state[ 3 + 1*4 ];
    state[ 3 + 1*4 ] = state[ 3 + 2*4 ];
    state[ 3 + 2*4 ] = state[ 3 + 3*4 ];
    state[ 3 + 3*4 ] = temp;
}

static void invSubBytes( byte * bytes, byte count )
{
    // Copy to temporary variables for optimization.
    byte * tempPtr = bytes;
    byte tempCount = count;

    do {
        *tempPtr = sBoxInv[ *tempPtr ]; // Sub every byte in state.
void invCipher( byte * block1 )
{
byte * scheduleBuffer = keyScheduleInSRAM+ROUNDS*BLOCK_SIZE;
byte round;

addConstant( block1, scheduleBuffer, BLOCK_SIZE);
for( round = 0; round < ROUNDS-1; ++round ) {
    invShiftRows( block1 );
    invSubBytes( block1, BLOCK_SIZE);
    scheduleBuffer -= BLOCK_SIZE;
    addConstant( block1, scheduleBuffer, BLOCK_SIZE);
    invMixColumns( block1 );
}

invShiftRows( block1 );
invSubBytes( block1, BLOCK_SIZE);
scheduleBuffer -= BLOCK_SIZE;
addConstant( block1, scheduleBuffer, BLOCK_SIZE);
}

//AES.H

void getsBox(void);
void getsBoxinv(void);
APPENDIX J. CMAC PROGRAM FOR RECEIVER

//CMAC.C

//Some statements in this function are referred from [3]

#include<p24fj128ga010.h>
#include "message.h"
#include "transmission.h"

static byte CMACSubkey[ CMAC_SUBKEY_SIZE ];
void calcCMACSubkey( byte * cryptoBlock )
{
    byte blockSize;
    byte * blockPtr;
    byte overflow;

    // Fill crypto block with all zeros.
    blockSize = BLOCK_SIZE;
    blockPtr = cryptoBlock;
    do {
        *blockPtr++ = 0;
    } while( --blockSize );

    // Encrypt zeros with secret key (assuming
    // key schedule has been created with secret key)
    cipher( cryptoBlock );

    // Multiply by 2 modulo (0b1^120 cat 0b10000111).
    blockSize = BLOCK_SIZE - 1;
    blockPtr = cryptoBlock;
    overflow = blockPtr[0] & 0x80;
    do {
        blockPtr[0] = (blockPtr[0] << 1) | (blockPtr[1] >> 7);
        ++blockPtr;
    } while( --blockSize );
    blockPtr[0] <<= 1;
    if( overflow ) {
        blockPtr[0] ^= 0x87;
    }

    #if MESSAGE_WO_MAC < BLOCK_SIZE
    // Multiply by 2 modulo (0b1^120 cat 0b10000111) again.
    blockSize = BLOCK_SIZE - 1;
    blockPtr = cryptoBlock;
    #endif
}
overflow = blockPtr[0] & 0x80;
do {
    blockPtr[0] = (blockPtr[0] << 1) | (blockPtr[1] >> 7);
    ++blockPtr;
} while( --blockSize );
blockPtr[0] <<= 1;
if( overflow ) {
    blockPtr[0] ^= 0x87;
}
#endif

void calcCMAC( const byte * message, byte * MACStorage )
{
    // Copy message data to MACStorage, which is of BLOCK_SIZE.
copyBytes( MACStorage, message, MESSAGE_WO_MAC );

#if MESSAGE_WO_MAC < BLOCK_SIZE
    // Pad message before calculating CMAC.
    #if MESSAGE_WO_MAC + 1 < BLOCK_SIZE
        byte pos = MESSAGE_WO_MAC + 1;
        do {
            MACStorage[ pos ] = 0x00;
        } while( ++pos < BLOCK_SIZE );
    #endif
    MACStorage[ MESSAGE_WO_MAC ] = 0x80;
#endif
    // Add (XOR) subkey.
    addConstant( MACStorage, CMACSubkey, BLOCK_SIZE );
    // Encrypt, assuming secret key schedule is precalculated.
cipher( MACStorage );
}
APPENDIX K. TRANSMISSION PROGRAM FOR RECEIVER

//TRANSMISSION.C

#include <p24fj128ga010.h>
#include "message.h"
#include "transmission.h"
#include "eeprom.h"
#include "main.h"

byte cryptoBlock[BLOCK_SIZE + 1];

void initializeMessage(void)
{
    //serial number from EEPROM
    SERIAL_NO_TYPE serialNo_0, serialNo_1, serialNo_2, serialNo_3, serialNo_tot;
    serialNo_0 = messageBlock[9];
    serialNo_0 = serialNo_0 & 0x000000ff;
    serialNo_1 = messageBlock[10];
    serialNo_1 = serialNo_1 & 0x000000ff;
    serialNo_1 = serialNo_1 << 8;
    serialNo_2 = messageBlock[11];
    serialNo_2 = serialNo_2 & 0x000000ff;
    serialNo_2 = serialNo_2 << 16;
    serialNo_3 = messageBlock[12];
    serialNo_3 = serialNo_3 & 0x000000ff;
    serialNo_3 = serialNo_3 << 24;
    serialNo_tot = (serialNo_3 | (serialNo_2 | (serialNo_1 | serialNo_0));
    sharedBlock.messagePayload.serialNo = serialNo_tot; // 0x54545667
    Nop();
}

void createMessage(COMMAND_CODE_TYPE commandCode)
{
    sharedBlock.messagePayload.commandCode = commandCode;
    //counter from eeprom
    sharedBlock.messagePayload.counterValue = counter;
}

void updateCounter(void)
{
    //extract counter from message and assign to counter and update

101
SEQ_COUNTER_TYPE counter_0, counter_1, counter_2, counter_3;
counter_0 = messageBlock[5];
counter_1 = messageBlock[6];
counter_1 = counter_1 & 0x000000ff;
counter_1 = counter_1 << 8;
counter_2 = messageBlock[7];
counter_2 = counter_2 & 0x000000ff;
counter_2 = counter_2 << 16;
counter_3 = messageBlock[8];
counter_3 = counter_3 & 0x000000ff;
counter_3 = counter_3 << 24;
counter = counter_0 | counter_1 | counter_2 | counter_3;
sharedBlock.messagePayload.counterValue = counter;
iWriteNVM(myAddr + 16, counter);
iWriteNVM(myAddr + 18, counter >> 16);
}

void updateCommand(void)
{
    sharedBlock.messagePayload.commandCode = messageBlock[4];
}

void msDelay(unsigned int ms)
{
    unsigned long i;
    for (i = (unsigned long) (ms + 1) * SCALE; i > 0; i /= 1) Nop();
}

//TRANSMISSION.H

#ifndef TRANSMISSION_H
#define TRANSMISSION_H

#define SCALE 333L

//Common workspace for encrypting and decrypting.
extern byte cryptoBlock[BLOCK_SIZE + 1];

struct MessagePayload
{
    //byte preamble;
    SERIAL_NO_TYPE serialNo;
    SEQ_COUNTER_TYPE counterValue;
    COMMAND_CODE_TYPE commandCode;
};

union SharedBlock
{
    struct MessagePayload messagePayload;
    byte cryptoBlock[BLOCK_SIZE];
};

extern union SharedBlock sharedBlock;

102
void updateCommand(void);
void initializeMessage(void);
void createMessage( COMMAND_CODE_TYPE commandCode );

// Update Counter
void updateCounter(void);

// Milli second delay
void msDelay(unsigned int ms);
#endif
APPENDIX L. EEPROM PROGRAM FOR RECEIVER

//EEPROM.C

#include<p24fj128ga010.h>
#include "eeprom.h"
#include "main.h"

// init the SPI peripheral
void InitNVM(void)
{
    CSEE = 1; // de-select the Serial EEPROM
    TCSEE = 0; // make SSEE pin output
    SPI2CON1 = SPI_MASTER; // select mode
    SPI2STAT = SPI_ENABLE; // enable the peripheral
}

// send one byte of data and receive one back at the same time
int WriteSPI2( int i)
{
    SPI2BUF = i; // write to buffer for TX
    while( !(SPI2STATbits.SPIRBF)); // wait for transfer to complete
    Nop();
    return SPI2BUF; // read the received value
}

// Check the Serial EEPROM status register
int ReadSR( void)
{
    int i;
    CSEE = 0; // select the Serial EEPROM
    WriteSPI2( SEE_RDSR); // send a READ STATUS COMMAND
    i = WriteSPI2( 0); // send/receive
    CSEE = 1; // deselect to terminate command
    return i;
} //ReadSR

// read a 16-bit value starting at an even address
int iReadNVM( int address)
{
    int lsb, msb;
    // wait until any work in progress is completed
    while ( ReadSR() & 0x03); // check the two lsb WEN and WIP
// perform a 16-bit read sequence (two byte sequential read)
CSEE = 0;  // select the Serial EEPROM
WriteSPI2( SEE_READ);  // read command
WriteSPI2( address>>8);  // address MSB first
WriteSPI2( address & 0x0fe);  // address LSB (word aligned)
msb = WriteSPI2( 0);  // send dummy, read msb
lsb = WriteSPI2( 0);  // send dummy, read lsb
CSEE = 1;
return ( (msb<<8)+ lsb);
} //iReadNVM

// send a Write Enable command
void WriteEnable( void)
{
    CSEE = 0;  // select the Serial EEPROM
    WriteSPI2( SEE_WEN);  // write enable command
    CSEE = 1;  // deselect to complete the command
} //WriteEnable

// write a 16-bit value starting at an even address
void iWriteNVM( int address, int data)
{
    int lsb, msb;

    // wait until any work in progress is completed
    while ( ReadSR() & 0x03);  // check the two lsb WEN and WIP

    // Set the Write Enable Latch
    WriteEnable();

    // perform a 16-bit write sequence (2 byte page write)
    CSEE = 0;  // select the Serial EEPROM
    WriteSPI2( SEE_WRITE);  // write command
    WriteSPI2( address>>8);  // address MSB first
    WriteSPI2( address & 0x0fe);  // address LSB (word aligned)
    WriteSPI2( data >>8);  // send msb
    WriteSPI2( data & 0xff);  // send lsb
    CSEE = 1;
} //iWriteNVM

void RDword(word Ind)
{
    Dato=iReadNVM(Ind);
}

void WRword(word Ind)
{
    iWriteNVM(Ind,Dato);
}
void iwriteEEPROM(void)
{
    // 2-bytes for each location
    // shared key
    iWriteNVM( myAddr+122, SHK_0); //SHK 16-bytes
    iWriteNVM( myAddr+124, SHK_1);
    iWriteNVM( myAddr+126, SHK_2);
    iWriteNVM( myAddr+128, SHK_3);
    iWriteNVM( myAddr+130, SHK_4);
    iWriteNVM( myAddr+132, SHK_5);
    iWriteNVM( myAddr+134, SHK_6);
    iWriteNVM( myAddr+136, SHK_7);
    //sBoxeeeprom
    writesBox();
    //sBoxinveeprom
    writesBoxinv();
}

void writesBox(void)
{
    int i;
    for(i=0; i<256; i++)
    {
        int temp1, temp2, temp;
        temp1=sBoxeeeprom[i];
        i=i+1;
        temp2=sBoxeeeprom[i];
        temp2=temp2<<8;
        temp=temp1|temp2;
        iWriteNVM( myAddr+136+i+1, temp);
    }
}

void writesBoxinv(void)
{
    int i;
    for(i=0; i<256; i++)
    {
        int temp1, temp2, temp;
        temp1=sBoxInveeprom[i];
        i=i+1;
        temp2=sBoxInveeprom[i];
        temp2=temp2<<8;
        temp=temp1|temp2;
        iWriteNVM( myAddr+396+i+1, temp);
    }
}
//EEPROM.H

#include "message.h"
#define TRUE 1
#define myAddr 0x0000

//Table structure definition:
//the EEPROM is filled with an array of MAX_USER user records
//starting at address 0x0000
//each record is EL SIZE byte and contains the following fields
//EEPROM access is in 16 bit words for efficiency

#define SHK_0 0x3a7f
#define SHK_1 0x2d8a
#define SHK_2 0x508f
#define SHK_3 0xcb1c
#define SHK_4 0xcab9
#define SHK_5 0x72b7
#define SHK_6 0x8f7b
#define SHK_7 0x076b
// 25LC256 Serial EEPROM commands

// I/O definitions for PIC24 + Explorer16 demo board
#define CSEE _RD12 // select line for Serial EEPROM
#define TCSEE _TRISD12 // tris control for CSEE pin

// peripheral configurations
#define SPI_MASTER 0x0122 // select 8-bit master mode, CKE=1, CKP=0, 4MHz
#define SPI_ENABLE 0x8000 // enable SPI port, clear status
#define SEE_WRSR 1 // write status register
#define SEE_WRITE 2 // write command
#define SEE_READ 3 // read command
#define SEE_WDI 4 // write disable
#define SEE_RDSR 5 // read status register
#define SEE_WEN 6 // write enable

static const byte sBoxeeeprom[256] = {
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
  0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
  0xca, 0x82, 0x97, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
  0xad, 0xd8, 0x31, 0x15, 0xc4, 0x4d, 0x33, 0x85,
  0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
  0x64, 0xe5, 0x0f, 0x17, 0xc4, 0xa7, 0x7e, 0x3d,
  0x60, 0x81, 0x9f, 0xdf, 0x22, 0x2a, 0x90, 0x88,
  0x46, 0x8e, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
  0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
  0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xb0, 0x16
};
static const byte sBoxInveeprom[256] = {
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38,
    0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87,
    0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d,
    0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2,
    0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x92,
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda,
    0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0xda, 0xd3,
    0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89,
    0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20,
    0x9a, 0xdb, 0x0c, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31,
    0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xc9, 0x5f,
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d,
    0x02, 0xe5, 0x9a, 0x9f, 0x93, 0xc0, 0x9c, 0xef,
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0,
    0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26,
    0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0xf4, 0xb1,
    0x12, 0x10, 0x59, 0x30, 0x36, 0xa5, 0x38, 0x3b,
    0xb0, 0xc9, 0x9c, 0xef, 0xa0, 0xe0, 0x3b, 0x4d,
    0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c,
    0x83, 0x53, 0x99, 0x61, 0x17, 0x2b, 0x04, 0x7e,
    0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63,
    0x55, 0x21, 0xf4, 0xb1, 0x12, 0x10, 0x59, 0x30,
    0x36, 0xa5, 0x38, 0x3b, 0xb0, 0xc9, 0x9c, 0xef,
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0,
    0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26,
    0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0xf4, 0xb1,
};

void InitNVM(void);
int WriteSPI2(int i);
int ReadSR(void);
int iReadNVM(int address);
void WriteEnable(void);
void iWriteNVM(int address, int data);
void iwriteEEPROM(void);
void RDword(word Ind);
void WRword(word Ind);
void writesBox(void);
void writesBoxinv(void);
APPENDIX M.COMMUNICATION PROGRAM FOR RECEIVER

//COMMUNICATION.H

#include "message.h"
#include "main.h"

#define MAX_USER 4 // max number of TX that can be learned
#define EL_SIZE 26 // single record size in bytes

#define command messageBlock[4]
#define HopLo messageBlock[5] //sync counter
#define HopMi1 messageBlock[6]
#define HopMi2 messageBlock[7]
#define HopHi messageBlock[8]
#define IDLo messageBlock[LENGTH-4] //S/N LSB
#define IDMi1 messageBlock[LENGTH-3] //S/N
#define IDMi2 messageBlock[LENGTH-2] //S/N
#define IDHi messageBlock[LENGTH-1] //S/N MSB

#define MASKPA 0xffff // port A I/O config (all input)
#define MASKPB 0xffff // port B I/O config (all input)
#define MASKPC 0xffff // port C I/O config (all input)

#define RFIn PORTAbits.RA1 // i radio signal input
#define Learn PORTAbits.RA4 // i learn button

#define HIGH_TO -10 // longest high Te
#define LOW_TO 10 // longest low Te
#define SHORT_HEAD 20 // shortest Thead accepted
#define LONG_HEAD 45 // longest Thead accepted

#define TRFreset 0
#define TRFSYNC 1
#define TRFUNO 2
#define TRFZERO 3

static byte RFstate; // receiver state
static int8_t RFcount; // timer counter
static byte Bptr; // receive buffer pointer
static byte BitCount; // received bits counter
word XTMR;                   // 16 bit extended timer

volatile int16_t RFFull;

volatile int16_t RFBit;      // sampled RF signal

#define TRUE    1
#define FALSE   0
#define ON      1
#define OFF     0

#define TRFreset     0
#define TRFSYNC      1
#define TRFUNO       2
#define TRFZERO      3

#define PERIOD 1125 /512

//byte Flags;                 // various flags
byte CLearn, CTLearn;        // learn timers and counter
byte CFlash, CTFlash;        // led flashing timer and counter
byte COut;                   // output timer
byte FCode;                  // function code

//flags defines

int16_t FHopOK;              // Hopping code verified OK
int16_t FSame;               // Same code as previous
int16_t FLearn;              // Learn mode active
int16_t F2Chance;            // Resync required

#define TOUT    5           //   5 * 71ms = 350ms output delay
#define TFLASH  2           //   4 * 71ms = 280ms half period
#define TLEARN  255         // 255 * 71ms = 18s learn timeout

int Led;
int OUT[4];

uint32_t Hop;                // hopping code sync counter
word EHop1;                  // lsb value of sync counter (from EEPROM)
word EHop2;                  // msb value of sync counter
uint32_t EHop;

word NextHop;                // resync value for 2 Chance
word ETemp;                  // second copy of sync counter

// function definitions

void InitReceiver();
void Remote();
byte Find();
byte Insert();
byte IDWrite();
byte HopUpdate();
byte HopCHK();
byte ReqResync();
void ClearMem(void);
void HopLearn(void);
APPENDIX N. MESSAGE PROGRAM FOR RECEIVER

//MESSAGE.H

#ifndef MESSAGE_H
#define MESSAGE_H

typedef signed char int8_t;
typedef unsigned char uint8_t;
typedef short int16_t;
typedef unsigned short uint16_t;
typedef long int32_t;
typedef unsigned long uint32_t;
typedef signed int word;
typedef uint8_t byte;
#endif

#define SERIAL_NO_BYTES 4
#define COMMAND_BYTES 1
#define COUNTER_BYTES 4
#define MESSAGE_WO_MAC (SERIAL_NO_BYTES+COMMAND_BYTES+COUNTER_BYTES)
#define SERIAL_NO_TYPE uint32_t
#define SERIAL_NO_MAX 0xffffffff
#define SEQ_COUNTER_TYPE uint32_t
#define COMMAND_CODE_TYPE byte
#define COMMAND_CODE_MAX 0xff
#define BLOCK_SIZE 16
#define ROUNDS 10
#define KEY_SIZE 16
#define SCHEDULE_SIZE_BLOCKS (ROUNDS+1)
#define SCHEDULE_SIZE (SCHEDULE_SIZE_BLOCKS*BLOCK_SIZE)
#define CMAC_SUBKEY_SIZE BLOCK_SIZE

extern SEQ_COUNTER_TYPE counter;

// Copy 'count' bytes from 'source' buffer to 'destination' buffer
void copyBytes( byte * destination, const byte * source, byte count);

// XOR 'count' bytes from 'constant' buffer into 'bytes' buffer.
void addConstant( byte * bytes, const byte * constant, byte count);

//Calculates Key Schedule for given key
void calcKeySchedule( byte * key);
// uses key schedule to encode the data
void cipher( byte * block1 );

// Calculates CMACSubkey
void calcCMACSubkey( byte * cryptoBlock );

// Calculates CMAC using CMAC Subkey
void calcCMAC( const byte * message, byte * MACStorage );

// uses key schedule to decode the data
void invCipher( byte * block1 );

#endif