Project	IEEE 802.16 Broadband Wireless Access Working Group <**http://ieee802.org/16**>
Title	CCM Endianess Disambiguation
Date Submitted	2005-04-22
Source(s)	David Johnston Intel Corporation Voice: (503) 264 3855 Email: dj.johnston@intel.com
Re:	Letter Ballot #17a, P802.16-2004/Cor1/D2
Abstract	
Purpose	This document corrects out of scope text and resolves an abiguity in the D1 corrigendum text
Notice	This document has been prepared to assist IEEE 802.16. It is offered as a basis for discussion and is not binding on the contributing individual(s) or organization(s). The material in this document is subject to change in form and content after further study. The contributor(s) reserve(s) the right to add, amend or withdraw material contained herein.
Release	The contributor grants a free, irrevocable license to the IEEE to incorporate material contained in this contribution, and any modifications thereof, in the creation of an IEEE Standards publication; to copyright in the IEEE's name any IEEE Standards publication even though it may include portions of this contribution; and at the IEEE's sole discretion to permit others to reproduce in whole or in part the resulting IEEE Standards publication. The contributor also acknowledges and accepts that this contribution may be made public by IEEE 802.16.
Patent Policy and Procedures	The contributor is familiar with the IEEE 802.16 Patent Policy and Procedures <http://ieee802.org/16/ipr/patents/policy.html>, including the statement "IEEE standards may include the known use of patent(s), including patent applications, provided the IEEE receives assurance from the patent holder or applicant with respect to patents essential for compliance with both mandatory and optional portions of the standard." Early disclosure to the Working Group of patent information that might be relevant to the standard is essential to reduce the possibility for delays in the development process and increase the likelihood that the draft publication will be approved for publication. Please notify the Chair <mailto:chair@wirelessman.org> as early as possible, in written or electronic form, if patented technology (or technology under patent application) might be incorporated into a draft standard being developed within the IEEE 802.16 Working Group. The Chair will disclose this notification via the IEEE 802.16 web site <http://ieee802.org/16/ipr/patents/notices>.

CCM Endianess Disambiguation

David Johnston, Intel Corporation

1 Introduction

PN and ICV ordering

Changes to clause 7.5.1.2.1 and 7.5.1.2.2 in the corrigendum draft 1 replace ‘little endian’ with ‘MSB first’ for both the PN and ICV fields.

Figure 136 in the base document also describes the ordering of the bytes in the PN and ICV fields. However this is inconsistent with the new text.

In either case (little endian or big endian) the meaning with respect to the ICV in the NIST CCM and AES specifications could be misconstrued.

These problems can be fixed either by amending figure 136 to resolve the ambiguities, or by undoing the reversal of the ordering of the PN and ICV octets in 7.5.1.2.1 and 7.5.1.2.2 and expressing the order of transmission of the ICV bytes in terms of the byte index (0-15) used in the AES and CCM specifications.

The security of the CCM mode is not affected by the ordering decision; however the ordering must be the same between systems for them to interwork. Thus the existing text in the base document is not in error.

Accordingly, the change in to the base document corrects neither an error, inconsistency nor ambiguity. So it is out of scope for the text in 7.5.1.2.1 and 7.5.1.2.2 to be changed in the fashion currently in the corrigendum draft. This means that of the two options for fixing the corrigendum, the only one open to us is to remove the changes to 7.5.1.2.1 and 7.5.1.2.2 and resolve any ambiguities, thus restoring consistency in the text and removing the out of scope changes. Changing the order of transmission is not an in-scope option.

Also the comment that led to the changes in 7.5.1.2.1 and 7.5.1.2.2 is classified as editorial. This is not correct, the changing of the transmission order is very much a technical change.

Test Vectors to Resolve Ambiguities

Taken as it stands, may still be possible to make multiple interpretations of the text. Existing practice in implementing CCM in other 802 documents (802.11i) leads to the intended interpretation, however this is not expressed directly in the spec.

One way of disambiguating between all possible interpretations is to write a lot more clarifying text about every bit field and byte field. A much simpler way to disambiguate the text is to show example enciphered packets along with their plain text. This document proposes such text.

The 802.16e draft includes test vectors and test vector C code in Annex E.1. This is a problem, since logically, vectors included in 802.16-2004 would occur before additions introduced in 802.16. It is proposed to place the code and vectors in Annex E.1 and require that the numbering in 802.16e be changed to E.2 to accommodate this.

Consistency with NIST SP 800-38C

Between the NIST draft CCM specification referenced in 802.16-2004 and the subsequently published final CCM standard SP 300 38C, the names of parts of the CCM standard were changed. E.G. the ICV is now the Message Authentication Code. We cannot use 'MAC', since the term is already defined in 802. Contribution C80216maint-05/024 corrects for these changes and provides alternatives for figure 135 and 136. However document 024 assumes the big endian PN ordering of Corr1/D1. Also there were some minor technical errors, E.G. the rounding of the PDU length in figure 135 and the inclusion of the CRC in the payload example. Appropriate changes from proposal 024 have been included in the proposed text, but alterations have been made to address the above problems.

2 Proposed Text Changes

[Resolution Part 1]

[Modify the changed against section 7.5.1.2.1 to be as follows]

7.5.1.2 Data encryption with AES in CCM mode
7.5.1.2.1 PDU Payload Format

Change the first and third paragraph as indicated:

The PDU payload shall be prepended with a 4-byte PN (Packet Number). The PN shall be transmitted in little endian byte order. The PN shall not be encrypted. The ciphertext ~~ICV~~Message Authentication Code is transmitted such that byte index 0 (as enumerated in the NIST AES Specification) is transmitted first and byte index 7 is transmitted last. ~~in little endian byte order.~~

[Delete changes to section 7.5.1.2.2 from the corrigendum draft]

~~**7.5.1.2.2 PN (Packet Number)**~~

~~***Change the first sentence of the first paragraph as indicated:***~~

~~The PN associated with an SA shall be set to 1 when the SA is established and when a new TEK is installed.~~
~~The PN shall be transmitted in little endianMSB first order in the MAC PDU as described in 7.5.1.2.1.~~

[Resolution Part 2]

[Replace "Ciphertext ICV" in figure 135 with "Ciphertext Message Authentication Code"]

~~Ciphertext ICV~~Ciphertext Message Authentication Code

[Modify Labeling of Figure 135 in 7.5.1.2.1 as follows]

Figure 135 –~~TEK Management in BS and SS~~ Encrypted Payload Format in AES-CCM Mode

[Modify 7.5.1.2.3 as follows]

The NIST CCM specification defines a number of algorithm parameters. These parameters shall be fixed to specific values when used in SAs with a data encryption algorithm identifier of 0x02.

'*Tlen*' shall equal 64 and *t* shall equal 8, meaning, the ~~The~~ number of octets in the Message Authentication Code field ~~authentication field M~~ shall be set to 8. Consistent with the CCM specification the 3 bit binary encoding $[(t-2)/2)]_3$ of ~~M~~ bits 5, 4 and 3 of the 'Flags' octet in B_0 shall be 011.

The size *q* of the length field *Q* shall be set to 2. Consistent with the CCM specification, the 3-bit binary encoding $[q-1]_3$ of the *q* field in bits 2, 1 and 0 of the 'Flags' octet in B_0 shall be 001.

The length *a* of the ~~additional authenticated~~ Associated data string *A* ~~*l(a)*~~ shall be set to 0.

The nonce shall be 13 bytes long as shown in figure 135a. Bytes 0 through 4 ~~1 through 5~~ shall be set to the first five bytes of the Generic MAC Header ~~GMH~~ (thus excluding the HCS). The HCS of the Generic MAC Header is not included in the nonce since it is redundant. Bytes 5 through 8 ~~Bytes 6 through 9~~ are reserved and shall be set to 0x00000000. ~~Bytes 10 through 13~~ Bytes 9 through 12 shall be set to the value of the PN. The PN Bytes shall be ordered such that Byte 9 ~~10~~ shall take the least significant byte and byte 12 ~~13~~ shall take the most significant byte.

Octet Number	0 ... 4	5 ... 8	9 ... 12
Field	Generic MAC Header	Reserved	PN
Contents	Generic Mac Header omitting HCS	0x00000000	packet number field from payload

Figure 135a Nonce *N* Construction

[Modify 136 and following text of 7.5.1.2.1 to be as follows. Delete Other features of figure 136]

~~Byte within MIC_IV~~ Octet Number	0	1 ... 13	14 15
Byte Significance			msb lsb
Number of Bytes	1	13	2
Field	Flag	Nonce	*L*~~DLEN~~
Contents	0x19	As Specified in Figure 135a	Length of plaintext payload ~~data part not including padding~~

Figure 136 – Initial CCM Block B_0

Note the ~~big endian~~ ordering of the DLEN value is big endian , consistent ~~opposite that of the normal little endian representation. This is to remain compliant~~ with the ~~letter of the~~ NIST CCM specification.

~~The sixth byte of the GMH is not included in the nonce since it is redundant.~~

Consistent with the NIST CCM specification the counter blocks Ctr_i ~~*Ai*~~ are formatted as shown in Figure 137.

[Modify Figure 137of 7.5.1.2.1 as follows. Delete other features of figure 137]

~~Byte within *CTRi*~~ Octet Number	0	1 ... 13	14 15
Byte Significance			msb lsb
Number of Bytes	1	13	2
Field	Flag	Nonce	C*ounter*
Contents	0x01	As Specified in Figure 135a	*i* ~~Length of data part not including padding~~

Figure 137 – Construction of counter blocks Ctr_i ~~A_i~~

[Resolution Part 3]

[Insert a new annex E.1 "Cryptographic Method Examples"]

E.1 Cryptographic Method Examples

E.1.1 AES-CCM Mode Cryptographic Method Examples

E.1.1.1 Example PDUs Enciphered in AES-CCM Mode

The following examples show 802.16 MPDUs in both plaintext and enciphered form in transmission order. In addition, the post-decryption plaintext of the Message Authentication Code is shown.

The examples were generated by the Test Program in E.1.1.2 compiled using gcc version 2.96 and run on a little endian, 32 bit, Linux PC.

E.1.1.1.1 Example AES-CCM PDU #1

The following example corresponds to test case 15 generated by the Test Program in E.1.1.2.

Plaintext PDU
Generic MAC Header = 00 40 0A 06 C4 30
Payload = 00 01 02 03
Ciphertext PDU where TEK = 0xD50E18A844AC5BF38E4CD72D9B0942E5 and PN=0x2157F6BC
Generic MAC Header = 40 40 1A 06 C4 5A
PN Field = BC F6 57 21
Encrypted Payload = E7 55 36 C8
Encrypted Message Authentication Code =
27 A8 D7 1B 43 2C A5 48
CRC = CB B6 5F 48
After Decryption
Plaintext Message Authentication Code =
01 59 09 A0 ED CC 21 D3

E.1.1.1.2 Example AES-CCM PDU #2

The following example corresponds to test case 84 generated by the Test Program in E.1.1.2.

Plaintext PDU
Generic MAC Header = 00 40 27 7E B2 AD
Payload = 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
20
Ciphertext PDU where TEK = 0xB74EB0E4F81AD63D121B7E9AECCD268F and PN=0x78D07D08
Generic MAC Header = 40 40 37 7E B2 C7
PN Field = 08 7D D0 78
Encrypted Payload = 71 3F B1 22 B9 73 4F DB FD 68 2E AD 9D CA 9F 44
1F 62 FE 0F 4A 2C 45 B5 53 17 3D 66 5B 2D 53 C1
B3
Encrypted Message Authentication Code =
E7 E4 8D 2D B7 61 CF 94
CRC = 92 1B 32 41
After Decryption
Plaintext Message Authentication Code =
0B DB 85 3C 0A CA E6 5F

E.1.1.2 Test Program for AES-CCM Mode Cryptographic Examples

```
/*****************************************/
/* 802.16 AES-CCM Example MPDU Encryption   */
/* Author: David Johnston                   */
/* Email: david.johnston@ieee.org           */
/*****************************************/

int num_test_cases;
int enc decn;

int test case length[100];
int test case enc decn[100];

unsigned long int test_case_pnl[100];
unsigned char keys[100*16];
unsigned char test cases[16384];
```

```
/*************************************/
/********* Table for CRC 32 ************/
/*************************************/

 unsigned char lookahead_table[1024] =
 {
        0x00,0x00,0x00,0x00,0x96,0x30,0x07,0x77,0x2c,0x61,0x0e,0xee,0xba,0x51,0x09,0x99,
        0x19,0xc4,0x6d,0x07,0x8f,0xf4,0x6a,0x70,0x35,0xa5,0x63,0xe9,0xa3,0x95,0x64,0x9e,
        0x32,0x88,0xdb,0x0e,0xa4,0xb8,0xdc,0x79,0x1e,0xe9,0xd5,0xe0,0x88,0xd9,0xd2,0x97,
        0x2b,0x4c,0xb6,0x09,0xbd,0x7c,0xb1,0x7e,0x07,0x2d,0xb8,0xe7,0x91,0x1d,0xbf,0x90,
        0x64,0x10,0xb7,0x1d,0xf2,0x20,0xb0,0x6a,0x48,0x71,0xb9,0xf3,0xde,0x41,0xbe,0x84,
        0x7d,0xd4,0xda,0x1a,0xeb,0xe4,0xdd,0x6d,0x51,0xb5,0xd4,0xf4,0xc7,0x85,0xd3,0x83,
        0x56,0x98,0x6c,0x13,0xc0,0xa8,0x6b,0x64,0x7a,0xf9,0x62,0xfd,0xec,0xc9,0x65,0x8a,
        0x4f,0x5c,0x01,0x14,0xd9,0x6c,0x06,0x63,0x63,0x3d,0x0f,0xfa,0xf5,0x0d,0x08,0x8d,
        0xc8,0x20,0x6e,0x3b,0x5e,0x10,0x69,0x4c,0xe4,0x41,0x60,0xd5,0x72,0x71,0x67,0xa2,
        0xd1,0xe4,0x03,0x3c,0x47,0xd4,0x04,0x4b,0xfd,0x85,0x0d,0xd2,0x6b,0xb5,0x0a,0xa5,
        0xfa,0xa8,0xb5,0x35,0x6c,0x98,0xb2,0x42,0xd6,0xc9,0xbb,0xdb,0x40,0xf9,0xbc,0xac,
        0xe3,0x6c,0xd8,0x32,0x75,0x5c,0xdf,0x45,0xcf,0x0d,0xd6,0xdc,0x59,0x3d,0xd1,0xab,
        0xac,0x30,0xd9,0x26,0x3a,0x00,0xde,0x51,0x80,0x51,0xd7,0xc8,0x16,0x61,0xd0,0xbf,
        0xb5,0xf4,0xb4,0x21,0x23,0xc4,0xb3,0x56,0x99,0x95,0xba,0xcf,0x0f,0xa5,0xbd,0xb8,
        0x9e,0xb8,0x02,0x28,0x08,0x88,0x05,0x5f,0xb2,0xd9,0x0c,0xc6,0x24,0xe9,0x0b,0xb1,
        0x87,0x7c,0x6f,0x2f,0x11,0x4c,0x68,0x58,0xab,0x1d,0x61,0xc1,0x3d,0x2d,0x66,0xb6,
        0x90,0x41,0xdc,0x76,0x06,0x71,0xdb,0x01,0xbc,0x20,0xd2,0x98,0x2a,0x10,0xd5,0xef,
        0x89,0x85,0xb1,0x71,0x1f,0xb5,0xb6,0x06,0xa5,0xe4,0xbf,0x9f,0x33,0xd4,0xb8,0xe8,
        0xa2,0xc9,0x07,0x78,0x34,0xf9,0x00,0x0f,0x8e,0xa8,0x09,0x96,0x18,0x98,0x0e,0xe1,
        0xbb,0x0d,0x6a,0x7f,0x2d,0x3d,0x6d,0x08,0x97,0x6c,0x64,0x91,0x01,0x5c,0x63,0xe6,
        0xf4,0x51,0x6b,0x6b,0x62,0x61,0x6c,0x1c,0xd8,0x30,0x65,0x85,0x4e,0x00,0x62,0xf2,
        0xed,0x95,0x06,0x6c,0x7b,0xa5,0x01,0x1b,0xc1,0xf4,0x08,0x82,0x57,0xc4,0x0f,0xf5,
        0xc6,0xd9,0xb0,0x65,0x50,0xe9,0xb7,0x12,0xea,0xb8,0xbe,0x8b,0x7c,0x88,0xb9,0xfc,
        0xdf,0x1d,0xdd,0x62,0x49,0x2d,0xda,0x15,0xf3,0x7c,0xd3,0x8c,0x65,0x4c,0xd4,0xfb,
        0x58,0x61,0xb2,0x4d,0xce,0x51,0xb5,0x3a,0x74,0x00,0xbc,0xa3,0xe2,0x30,0xbb,0xd4,
        0x41,0xa5,0xdf,0x4a,0xd7,0x95,0xd8,0x3d,0x6d,0xc4,0xd1,0xa4,0xfb,0xf4,0xd6,0xd3,
        0x6a,0xe9,0x69,0x43,0xfc,0xd9,0x6e,0x34,0x46,0x88,0x67,0xad,0xd0,0xb8,0x60,0xda,
        0x73,0x2d,0x04,0x44,0xe5,0x1d,0x03,0x33,0x5f,0x4c,0x0a,0xaa,0xc9,0x7c,0x0d,0xdd,
        0x3c,0x71,0x05,0x50,0xaa,0x41,0x02,0x27,0x10,0x10,0x0b,0xbe,0x86,0x20,0x0c,0xc9,
        0x25,0xb5,0x68,0x57,0xb3,0x85,0x6f,0x20,0x09,0xd4,0x66,0xb9,0x9f,0xe4,0x61,0xce,
        0x0e,0xf9,0xde,0x5e,0x98,0xc9,0xd9,0x29,0x22,0x98,0xd0,0xb0,0xb4,0xa8,0xd7,0xc7,
        0x17,0x3d,0xb3,0x59,0x81,0x0d,0xb4,0x2e,0x3b,0x5c,0xbd,0xb7,0xad,0x6c,0xba,0xc0,
        0x20,0x83,0xb8,0xed,0xb6,0xb3,0xbf,0x9a,0x0c,0xe2,0xb6,0x03,0x9a,0xd2,0xb1,0x74,
        0x39,0x47,0xd5,0xea,0xaf,0x77,0xd2,0x9d,0x15,0x26,0xdb,0x04,0x83,0x16,0xdc,0x73,
        0x12,0x0b,0x63,0xe3,0x84,0x3b,0x64,0x94,0x3e,0x6a,0x6d,0x0d,0xa8,0x5a,0x6a,0x7a,
        0x0b,0xcf,0x0e,0xe4,0x9d,0xff,0x09,0x93,0x27,0xae,0x00,0x0a,0xb1,0x9e,0x07,0x7d,
        0x44,0x93,0x0f,0xf0,0xd2,0xa3,0x08,0x87,0x68,0xf2,0x01,0x1e,0xfe,0xc2,0x06,0x69,
        0x5d,0x57,0x62,0xf7,0xcb,0x67,0x65,0x80,0x71,0x36,0x6c,0x19,0xe7,0x06,0x6b,0x6e,
        0x76,0x1b,0xd4,0xfe,0xe0,0x2b,0xd3,0x89,0x5a,0x7a,0xda,0x10,0xcc,0x4a,0xdd,0x67,
        0x6f,0xdf,0xb9,0xf9,0xf9,0xef,0xbe,0x8e,0x43,0xbe,0xb7,0x17,0xd5,0x8e,0xb0,0x60,
        0xe8,0xa3,0xd6,0xd6,0x7e,0x93,0xd1,0xa1,0xc4,0xc2,0xd8,0x38,0x52,0xf2,0xdf,0x4f,
        0xf1,0x67,0xbb,0xd1,0x67,0x57,0xbc,0xa6,0xdd,0x06,0xb5,0x3f,0x4b,0x36,0xb2,0x48,
        0xda,0x2b,0x0d,0xd8,0x4c,0x1b,0x0a,0xaf,0xf6,0x4a,0x03,0x36,0x60,0x7a,0x04,0x41,
        0xc3,0xef,0x60,0xdf,0x55,0xdf,0x67,0xa8,0xef,0x8e,0x6e,0x31,0x79,0xbe,0x69,0x46,
        0x8c,0xb3,0x61,0xcb,0x1a,0x83,0x66,0xbc,0xa0,0xd2,0x6f,0x25,0x36,0xe2,0x68,0x52,
        0x95,0x77,0x0c,0xcc,0x03,0x47,0x0b,0xbb,0xb9,0x16,0x02,0x22,0x2f,0x26,0x05,0x55,
        0xbe,0x3b,0xba,0xc5,0x28,0x0b,0xbd,0xb2,0x92,0x5a,0xb4,0x2b,0x04,0x6a,0xb3,0x5c,
        0xa7,0xff,0xd7,0xc2,0x31,0xcf,0xd0,0xb5,0x8b,0x9e,0xd9,0x2c,0x1d,0xae,0xde,0x5b,
        0xb0,0xc2,0x64,0x9b,0x26,0xf2,0x63,0xec,0x9c,0xa3,0x6a,0x75,0x0a,0x93,0x6d,0x02,
        0xa9,0x06,0x09,0x9c,0x3f,0x36,0x0e,0xeb,0x85,0x67,0x07,0x72,0x13,0x57,0x00,0x05,
        0x82,0x4a,0xbf,0x95,0x14,0x7a,0xb8,0xe2,0xae,0x2b,0xb1,0x7b,0x38,0x1b,0xb6,0x0c,
        0x9b,0x8e,0xd2,0x92,0x0d,0xbe,0xd5,0xe5,0xb7,0xef,0xdc,0x7c,0x21,0xdf,0xdb,0x0b,
        0xd4,0xd2,0xd3,0x86,0x42,0xe2,0xd4,0xf1,0xf8,0xb3,0xdd,0x68,0x6e,0x83,0xda,0x1f,
        0xcd,0x16,0xbe,0x81,0x5b,0x26,0xb9,0xf6,0xe1,0x77,0xb0,0x6f,0x77,0x47,0xb7,0x18,
        0xe6,0x5a,0x08,0x88,0x70,0x6a,0x0f,0xff,0xca,0x3b,0x06,0x66,0x5c,0x0b,0x01,0x11,
        0xff,0x9e,0x65,0x8f,0x69,0xae,0x62,0xf8,0xd3,0xff,0x6b,0x61,0x45,0xcf,0x6c,0x16,
        0x78,0xe2,0x0a,0xa0,0xee,0xd2,0x0d,0xd7,0x54,0x83,0x04,0x4e,0xc2,0xb3,0x03,0x39,
        0x61,0x26,0x67,0xa7,0xf7,0x16,0x60,0xd0,0x4d,0x47,0x69,0x49,0xdb,0x77,0x6e,0x3e,
        0x4a,0x6a,0xd1,0xae,0xdc,0x5a,0xd6,0xd9,0x66,0x0b,0xdf,0x40,0xf0,0x3b,0xd8,0x37,
        0x53,0xae,0xbc,0xa9,0xc5,0x9e,0xbb,0xde,0x7f,0xcf,0xb2,0x47,0xe9,0xff,0xb5,0x30,
        0x1c,0xf2,0xbd,0xbd,0x8a,0xc2,0xba,0xca,0x30,0x93,0xb3,0x53,0xa6,0xa3,0xb4,0x24,
        0x05,0x36,0xd0,0xba,0x93,0x06,0xd7,0xcd,0x29,0x57,0xde,0x54,0xbf,0x67,0xd9,0x23,
        0x2e,0x7a,0x66,0xb3,0xb8,0x4a,0x61,0xc4,0x02,0x1b,0x68,0x5d,0x94,0x2b,0x6f,0x2a,
        0x37,0xbe,0x0b,0xb4,0xa1,0x8e,0x0c,0xc3,0x1b,0xdf,0x05,0x5a,0x8d,0xef,0x02,0x2d

  };

/*************************************/
```

```
/********* Table for CRC 8  ************/
/***************************************/

 unsigned char crc8 lookahead table[256] =
 {
  0x00, 0x07, 0x0E, 0x09, 0x1C, 0x1B, 0x12, 0x15, 0x38, 0x3F, 0x36, 0x31, 0x24, 0x23, 0x2A, 0x2D,
  0x70, 0x77, 0x7E, 0x79, 0x6C, 0x6B, 0x62, 0x65, 0x48, 0x4F, 0x46, 0x41, 0x54, 0x53, 0x5A, 0x5D,
  0xE0, 0xE7, 0xEE, 0xE9, 0xFC, 0xFB, 0xF2, 0xF5, 0xD8, 0xDF, 0xD6, 0xD1, 0xC4, 0xC3, 0xCA, 0xCD,
  0x90, 0x97, 0x9E, 0x99, 0x8C, 0x8B, 0x82, 0x85, 0xA8, 0xAF, 0xA6, 0xA1, 0xB4, 0xB3, 0xBA, 0xBD,
  0xC7, 0xC0, 0xC9, 0xCE, 0xDB, 0xDC, 0xD5, 0xD2, 0xFF, 0xF8, 0xF1, 0xF6, 0xE3, 0xE4, 0xED, 0xEA,
  0xB7, 0xB0, 0xB9, 0xBE, 0xAB, 0xAC, 0xA5, 0xA2, 0x8F, 0x88, 0x81, 0x86, 0x93, 0x94, 0x9D, 0x9A,
  0x27, 0x20, 0x29, 0x2E, 0x3B, 0x3C, 0x35, 0x32, 0x1F, 0x18, 0x11, 0x16, 0x03, 0x04, 0x0D, 0x0A,
  0x57, 0x50, 0x59, 0x5E, 0x4B, 0x4C, 0x45, 0x42, 0x6F, 0x68, 0x61, 0x66, 0x73, 0x74, 0x7D, 0x7A,
  0x89, 0x8E, 0x87, 0x80, 0x95, 0x92, 0x9B, 0x9C, 0xB1, 0xB6, 0xBF, 0xB8, 0xAD, 0xAA, 0xA3, 0xA4,
  0xF9, 0xFE, 0xF7, 0xF0, 0xE5, 0xE2, 0xEB, 0xEC, 0xC1, 0xC6, 0xCF, 0xC8, 0xDD, 0xDA, 0xD3, 0xD4,
  0x69, 0x6E, 0x67, 0x60, 0x75, 0x72, 0x7B, 0x7C, 0x51, 0x56, 0x5F, 0x58, 0x4D, 0x4A, 0x43, 0x44,
  0x19, 0x1E, 0x17, 0x10, 0x05, 0x02, 0x0B, 0x0C, 0x21, 0x26, 0x2F, 0x28, 0x3D, 0x3A, 0x33, 0x34,
  0x4E, 0x49, 0x40, 0x47, 0x52, 0x55, 0x5C, 0x5B, 0x76, 0x71, 0x78, 0x7F, 0x6A, 0x6D, 0x64, 0x63,
  0x3E, 0x39, 0x30, 0x37, 0x22, 0x25, 0x2C, 0x2B, 0x06, 0x01, 0x08, 0x0F, 0x1A, 0x1D, 0x14, 0x13,
  0xAE, 0xA9, 0xA0, 0xA7, 0xB2, 0xB5, 0xBC, 0xBB, 0x96, 0x91, 0x98, 0x9F, 0x8A, 0x8D, 0x84, 0x83,
  0xDE, 0xD9, 0xD0, 0xD7, 0xC2, 0xC5, 0xCC, 0xCB, 0xE6, 0xE1, 0xE8, 0xEF, 0xFA, 0xFD, 0xF4, 0xF3
 };

/*****************************/
/******** SBOX Table *********/
/*****************************/

    unsigned char sbox table[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,
        0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
        0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
        0x53, 0xd1, 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, 0xa8,
        0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
        0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
        0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,
        0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
        0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,
        0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
        0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
        0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
        0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
        0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
        0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,
        0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
        0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
        0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
        0xe1, 0xf8, 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, 0xbb, 0x16
    };


/*****************************/
/**** Function Prototypes ****/
/*****************************/

void bitwise xor(unsigned char *ina, unsigned char *inb, unsigned char *out);
void construct_mic_iv(
                        unsigned char *mic header1,
                        unsigned char *mpdu,
                        unsigned int payload length,
                        unsigned char * pn vector
```

```
                        );
void construct ctr preload(
                        unsigned char *ctr preload,
                        unsigned char *mpdu,
                        unsigned char *pn_vector,
                        int c
                        );
void get mpdu(      int test case,
                    unsigned char *plaintext);

int encrypt_mpdu(   unsigned char *key,
                    unsigned char *morphed mpdu,
                    int length,
                    unsigned char *ciphertext,
                    int test case);

void xor 128(unsigned char *a, unsigned char *b, unsigned char *out);
void xor 32(unsigned char *a, unsigned char *b, unsigned char *out);
unsigned char sbox(unsigned char a);
void next key(unsigned char *key, int round);
void byte sub(unsigned char *in, unsigned char *out);
void shift_row(unsigned char *in, unsigned char *out);
void mix column(unsigned char *in, unsigned char *out);
void add round key( unsigned char *shiftrow in,
                    unsigned char *mcol in,
                    unsigned char *block in,
                    int round,
                    unsigned char *out);
void aes128k128d(unsigned char *key, unsigned char *data, unsigned char *ciphertext);

/*********************************/
/* Routines to print hex fields */
/*********************************/

void blockprint_key(unsigned char* block)
{
          printf("Key = 0x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x\n",
                    block[0], block[1], block[2], block[3],
                    block[4], block[5], block[6], block[7],
                    block[8], block[9], block[10], block[11],
                    block[12], block[13], block[14], block[15]);
}

void blockprint(unsigned char *str, unsigned char* block)
{
          printf("%s = 0x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x\n", str,
                    block[15], block[14], block[13], block[12],
                    block[11], block[10], block[9], block[8],
                    block[7], block[6], block[5], block[4],
                    block[3], block[2], block[1], block[0]);
}

void blockprint_gmh(unsigned char *str, unsigned char *gmh)
{
          printf("%s = %02x %02x %02x %02x %02x %02x\n", str,
                    gmh[0], gmh[1], gmh[2], gmh[3], gmh[4], gmh[5]);
}

void blockprint_payload(unsigned char *str, unsigned char *payload, int length)
{
          int blocks;
          int residue;
          int i;
          int j;
          unsigned char *ptr;

          ptr = payload;
          blocks = length/16;
          residue = length % 16;

          printf("%s",str);
          if (blocks > 0)
          {
                    printf ("\t");
```

```
                    for (j=0;j<15;j++)
                    {
                              printf("%02x ",*ptr++);
                    }
                    printf("%02x\n",*ptr++);
          }

          for (i=1;i<blocks;i++)
          {
                    printf("\t\t\t");
                    for (j=0;j<15;j++)
                    {
                              printf("%02x ",*ptr++);
                    }
                    printf("%02x\n",*ptr++);
          }

          if (residue > 0)
          {
                    if (blocks != 0)
                    {
                              printf("\t\t\t");
                    }
                    else printf("\t");
                    for(i=0;i<(residue-1);i++)
                    {
                              printf("%02x ",*ptr++);
                    }
                    printf("%02x\n",*ptr++);
          }
}


/***************************************/
/* aes128k128d()                       */
/* Performs a 128 bit AES encrypt with  */
/* 128 bit data.                       */
/***************************************/
void xor 128(unsigned char *a, unsigned char *b, unsigned char *out)
{
     int i;
     for (i=0;i<16; i++)
     {
          out[i] = a[i] ^ b[i];
     }
}

void xor 32(unsigned char *a, unsigned char *b, unsigned char *out)
{
     int i;
     for (i=0;i<4; i++)
     {
          out[i] = a[i] ^ b[i];
     }
}
unsigned char sbox(unsigned char a)
{
     return sbox table[(int)a];
}

void next key(unsigned char *key, int round)
{
     unsigned char rcon;
     unsigned char sbox_key[4];
     unsigned char rcon table[12] =
     {
          0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
          0x1b, 0x36, 0x36, 0x36
     };

     sbox key[0] = sbox(key[13]);
     sbox key[1] = sbox(key[14]);
     sbox key[2] = sbox(key[15]);
     sbox key[3] = sbox(key[12]);
```

```
    rcon = rcon_table[round];

    xor_32(&key[0], sbox_key, &key[0]);
    key[0] = key[0] ^ rcon;

    xor_32(&key[4], &key[0], &key[4]);
    xor_32(&key[8], &key[4], &key[8]);
    xor_32(&key[12], &key[8], &key[12]);
}

void byte_sub(unsigned char *in, unsigned char *out)
{
    int i;
    for (i=0; i< 16; i++)
    {
        out[i] = sbox(in[i]);
    }
}

void shift_row(unsigned char *in, unsigned char *out)
{
    out[0] =  in[0];
    out[1] =  in[5];
    out[2] =  in[10];
    out[3] =  in[15];
    out[4] =  in[4];
    out[5] =  in[9];
    out[6] =  in[14];
    out[7] =  in[3];
    out[8] =  in[8];
    out[9] =  in[13];
    out[10] = in[2];
    out[11] = in[7];
    out[12] = in[12];
    out[13] = in[1];
    out[14] = in[6];
    out[15] = in[11];
}

void mix_column(unsigned char *in, unsigned char *out)
{
    int i;
    unsigned char add1b[4];
    unsigned char add1bf7[4];
    unsigned char rotl[4];
    unsigned char swap_halfs[4];
    unsigned char andf7[4];
    unsigned char rotr[4];
    unsigned char temp[4];
    unsigned char tempb[4];

    for (i=0 ; i<4; i++)
    {
        if ((in[i] & 0x80)== 0x80)
            add1b[i] = 0x1b;
        else
            add1b[i] = 0x00;
    }

    swap_halfs[0] = in[2];    /* Swap halfs */
    swap_halfs[1] = in[3];
    swap_halfs[2] = in[0];
    swap_halfs[3] = in[1];

    rotl[0] = in[3];         /* Rotate left 8 bits */
    rotl[1] = in[0];
    rotl[2] = in[1];
    rotl[3] = in[2];

    andf7[0] = in[0] & 0x7f;
    andf7[1] = in[1] & 0x7f;
    andf7[2] = in[2] & 0x7f;
    andf7[3] = in[3] & 0x7f;
```

```
    for (i = 3; i>0; i--)     /* logical shift left 1 bit */
    {
        andf7[i] = andf7[i] << 1;
        if ((andf7[i-1] & 0x80) == 0x80)
        {
            andf7[i] = (andf7[i] | 0x01);
        }
    }
    andf7[0] = andf7[0] << 1;
    andf7[0] = andf7[0] & 0xfe;

    xor_32(add1b, andf7, add1bf7);

    xor_32(in, add1bf7, rotr);

    temp[0] = rotr[0];          /* Rotate right 8 bits */
    rotr[0] = rotr[1];
    rotr[1] = rotr[2];
    rotr[2] = rotr[3];
    rotr[3] = temp[0];

    xor_32(add1bf7, rotr, temp);
    xor_32(swap halfs, rotl,tempb);
    xor_32(temp, tempb, out);
}

void aes128k128d(unsigned char *key, unsigned char *data, unsigned char *ciphertext)
{
    int round;
    int i;
    unsigned char intermediatea[16];
    unsigned char intermediateb[16];
    unsigned char round key[16];

    for(i=0; i<16; i++) round key[i] = key[i];

    for (round = 0; round < 11; round++)
    {
        if (round == 0)
        {
            xor_128(round key, data, ciphertext);
            next key(round key, round);
        }
        else if (round == 10)
        {
            byte sub(ciphertext, intermediatea);
            shift row(intermediatea, intermediateb);
            xor_128(intermediateb, round key, ciphertext);
        }
        else    /* 1 - 9 */
        {
            byte_sub(ciphertext, intermediatea);
            shift row(intermediatea, intermediateb);
            mix column(&intermediateb[0], &intermediatea[0]);
            mix column(&intermediateb[4], &intermediatea[4]);
            mix column(&intermediateb[8], &intermediatea[8]);
            mix column(&intermediateb[12], &intermediatea[12]);
            xor_128(intermediatea, round_key, ciphertext);
            next key(round key, round);
        }
    }

}

/****************************************/
/* CRC8()                               */
/* Calculates the CRC32 of a sequence   */
/* of octets.                           */
/****************************************/

void crc8(unsigned char *crc, unsigned char *data, int length)
{
    int i;
```

```
    int index;
    unsigned char ch;
    unsigned char table_entry;

    *crc = 0x00;
    for (i=0; i<length; i++)
    {
        ch = data[i];
        index = (((int)((*crc) ^ ch)) & 0xff);
        *crc = crc8_lookahead_table[index];
    }
}

/*****************************************/
/* CRC32()                               */
/* Calculates the CRC32 of a sequence    */
/* of octets.                            */
/*****************************************/

void crc32(unsigned char *crc, unsigned char *data, int length)
{

    int i;
    int index;
    unsigned char ch;
    unsigned char table_entry[4];

    crc[3] = 0xff;
    crc[2] = 0xff;
    crc[1] = 0xff;
    crc[0] = 0xff;

    for (i=0; i<length; i++)
    {
        ch = data[i];

        index = (((int)(crc[0] ^ ch)) & 0xff) * 4;

        table_entry[0] = lookahead_table[index];
        table_entry[1] = lookahead_table[index+1];
        table_entry[2] = lookahead_table[index+2];
        table_entry[3] = lookahead_table[index+3];

        crc[0] = crc[1] ^ table_entry[0];
        crc[1] = crc[2] ^ table_entry[1];
        crc[2] = crc[3] ^ table_entry[2];
        crc[3] = table_entry[3];

    }

    crc[0] = crc[0] ^ 0xff;
    crc[1] = crc[1] ^ 0xff;
    crc[2] = crc[2] ^ 0xff;
    crc[3] = crc[3] ^ 0xff;
}

/*************************************************/
/* construct_mic_iv()                            */
/* Builds the MIC IV from header fields and PN   */
/*************************************************/

void construct_mic_iv(
                        unsigned char *mic_iv,
                        unsigned char *mpdu,
                        unsigned int payload_length,
                        unsigned char *pn_vector
                        )
{
    int i;

    mic_iv[0] = 0x19;

    for (i = 1; i < 6; i++)
        mic_iv[i] = mpdu[i-1];  /* mic_iv[1:5] = GMH[0:4] = mpdu[0:4], little endian */
```

```
    for (i = 10; i < 14; i++)
        mic_iv[i] = pn_vector[i-10];    /* mic_iv[10:13] = PN[0:3], little endian*/

         mic_iv[6] = 0x00;
         mic_iv[7] = 0x00;
         mic_iv[8] = 0x00;
         mic_iv[9] = 0x00;

    mic_iv[14] = (unsigned char) (payload_length / 256); /* Big Endian DLEN */
    mic_iv[15] = (unsigned char) (payload_length % 256);
}

/*************************************************/
/* construct_ctr_preload()                       */
/* Builds the ctr_preload block                  */
/*************************************************/


void construct_ctr_preload(
                        unsigned char *ctr_preload,
                        unsigned char *mpdu,
                        unsigned char *pn_vector,
                        int c
                        )
{
    int i = 0;
    for (i=0; i<16; i++) ctr_preload[i] = 0x00;
    i = 0;

    ctr_preload[0] = 0x01;                                  /* flag */

    for (i = 1; i < 6; i++)
        ctr_preload[i] = mpdu[i-1];                     /* mic_iv[1:5] = GMH[0:4] = mpdu[0:4] */

    for (i = 10; i < 14; i++)
        ctr_preload[i] = pn_vector[i-10];           /* mic_iv[6:13] = PN[0:7] */
        ctr_preload[6] = 0x00;
        ctr_preload[7] = 0x00;
         ctr_preload[8] = 0x00;
         ctr_preload[9] = 0x00;

    ctr_preload[14] =  (unsigned char) (c / 256); /* Ctr - big endian */
    ctr_preload[15] =  (unsigned char) (c % 256);
}


/***********************************/
/* bitwise_xor()                   */
/* A 128 bit, bitwise exclusive or  */
/***********************************/

void bitwise_xor(unsigned char *ina, unsigned char *inb, unsigned char *out)
{
    int i;
    for (i=0; i<16; i++)
    {
        out[i] = ina[i] ^ inb[i];
    }
}


/*****************************************************/
/* int encrypt_mpdu()                                */
/* Encrypts a plaintext mpdu in accordance with      */
/* the 802.16D-2004 ccm specification. PN and MIC    */
/* insertion takes place.                            */
/* Returns the resulting length of the packet.       */
/*****************************************************/
int encrypt_mpdu(   unsigned char *key,
                    unsigned char *morphed_mpdu,
                    int length,
                    unsigned char *ciphertext,
                       int test_case)
```

```
{
    /* PN */
    unsigned char pn vector[4];
    unsigned char pn byte;
    /* MIC working variables */
    int i;
    int j;
    int payload length;
    int num blocks;
    int payload remainder;
    int payload_index;
    /* Initialization Blocks */
    unsigned char mic iv[16];
    unsigned char ctr preload[16];

    /* Length adjustment on header */
    unsigned char lengthlsb;
    unsigned char lengthmsb;

    /* Intermediate Buffers */
    unsigned char chain buffer[16];
    unsigned char aes_out[16];
    unsigned char padded buffer[16];

    /* MIC */
    unsigned char mic[8];

    for (i=0;i<16;i++)              /* Reset the buffers to zero */
    {
        mic iv[i] = 0x00;
        ctr preload[i] = 0x00;
        chain buffer[i] = 0x00;
        aes out[i] = 0x00;
        padded buffer[i] = 0x00;
    }

    #ifdef SHOW PHASES
    printf("--ENCRYPTING...\n");
    #endif

    #ifdef SHOW HEADER FIELDS
    printf("---- Key =             (msb) %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x
%02x %02x (lsb)\n",
                        key[0], key[1], key[2], key[3], key[4], key[5], key[6], key[7],
                        key[8], key[9], key[10], key[11], key[12], key[13], key[14], key[15]  );
    #endif


    for (i=0; i<4; i++)
    {
         pn vector[i] = morphed mpdu[6+i];
    }

    /* Copy plaintext to ciphertext */
    for (i=0; i<length; i++)
    {
        ciphertext[i] = morphed mpdu[i];
    };


    /* Turn the PN into a vector of bytes, taking care  */
    /* not to be dependent on the endianess of the      */
    /* architecture we are running on                   */

    #ifdef SHOW HEADER FIELDS
    printf("---- PN =              0x%08lx\n", pnl );
    #endif

    /* Calculate MIC */
    payload_length = length - 10; /* subtract header and pn length */
    construct mic iv(
                        mic iv,
                        morphed mpdu,
                        payload length,
```

```
                        pn vector
                        );

    #ifdef SHOW INIT BLOCKS
    printf("---- MIC_IV =          (lsb) ");
    for (i=0;i<16;i++) printf("%02x ", mic iv[i]);
    printf(" (msb)\n");

    #endif

    /* Calculate number of 16 byte blocks in MPDU */

    payload remainder = (payload length) % 16;
    num blocks = (payload length) / 16;

    /* Find start of payload */
    payload index = 10;

    /* Calculate MIC */
    aes128k128d(key, mic iv, aes out);
    #ifdef SHOW DEBUG
          blockprint("  ---- First MIC (lsb)", aes_out);
    #endif

    /* iterate through each 16 byte payload block */
    for (i = 0; i < num blocks; i++)
    {
        bitwise xor(aes out, &morphed mpdu[payload index], chain buffer);
          #ifdef SHOW_DEBUG
          blockprint("  ---- Subsequent MIC input (lsb)", chain buffer);
          #endif
        payload index += 16;
        aes128k128d(key, chain buffer, aes out);
          #ifdef SHOW DEBUG
          blockprint("  ---- Subsequent MIC (lsb)", aes_out);
          #endif
    }

    /* Add on the final payload block if it needs padding */
    if (payload_remainder > 0)
    {
        for (j = 0; j < 16; j++) padded buffer[j] = 0x00;
        for (j = 0; j < payload remainder; j++)
        {
            padded buffer[j] = morphed mpdu[payload index++];
        }
          #ifdef SHOW DEBUG
          blockprint("  ---- padded mic input ", padded buffer);
          #endif
        bitwise xor(aes out, padded buffer, chain buffer);
          #ifdef SHOW DEBUG
          blockprint("  ---- final MIC input", chain buffer);
          #endif
        aes128k128d(key, chain buffer, aes out);
          #ifdef SHOW DEBUG
          blockprint("  ---- final PT MIC   ", aes out);
          #endif
    }

    /* aes out contains padded mic, discard most significant    */
    /* 8 bytes to generate 64 bit MIC                           */

    for (j = 0 ; j < 8; j++) mic[j] = aes out[j];

    /* Insert MIC into payload */
    for (j = 0; j < 8; j++)
    {
        //printf(" mic putting index %d, value = 0x%02x\n",(payload index + j), mic[j]);
        ciphertext[payload index + j] = mic[j];
    }
    length += 8;
    payload length += 8;

    /**** Encrypt the payload and MIC ****/
```

```
    payload index = 10; /* header + PN header */
    for (i=0; i< num blocks; i++)
    {
        construct_ctr_preload(
                                    ctr preload,
                                    morphed mpdu,
                                    pn vector,
                                    i+1
                                );
        #ifdef SHOW_CTR_PRELOAD
        printf("---- CTR_PRELOAD(%i) = (lsb) %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x
%02x %02x %02x  (msb)\n",
                                i+1, ctr preload[0],ctr preload[1],ctr preload[2],ctr preload[3],
                                ctr preload[4],ctr preload[5],ctr preload[6],ctr preload[7],
                                ctr_preload[8],ctr_preload[9],ctr_preload[10],ctr_preload[11],
                                ctr preload[12],ctr preload[13],ctr preload[14],ctr preload[15]);
        #endif

        aes128k128d(key, ctr preload, aes out);
          bitwise xor(aes out, &ciphertext[payload index], chain buffer);
          #ifdef SHOW_DEBUG
          blockprint("  ---- Cipherstream = ", aes out);
          blockprint("  ---- Plaintext    = ", &ciphertext[payload index]);
          blockprint("  ---- Ciphertext   = ", chain buffer);
          #endif
        for (j=0; j<16;j++) ciphertext[payload index++] = chain buffer[j];
    }

    if (payload remainder > 0)          /* If there is a short final block, then pad it,*/
    {                                   /* encrypt it and copy the unpadded part back   */
        construct ctr preload(
                                    ctr preload,
                                    morphed mpdu,
                                    pn_vector,
                                    num blocks+1
                                );

        for (j = 0; j < 16; j++) padded buffer[j] = 0x00;
        for (j = 0; j < payload_remainder; j++)
        {
            padded buffer[j] = ciphertext[payload index+j];
        }
        aes128k128d(key, ctr preload, aes out);
          #ifdef SHOW DEBUG
          blockprint(" ---- final CTR_PRELOAD ", ctr_preload);
          blockprint(" ---- final cipherstream ", aes out);
          blockprint(" ---- final plaintest    ", padded buffer);
          #endif
        bitwise xor(aes out, padded buffer, chain buffer);
        for (j=0; j<payload remainder;j++) ciphertext[payload index++] = chain buffer[j];
    }

    /* Encrypt the MIC */
    construct ctr preload(
                            ctr preload,
                            morphed mpdu,
                            pn vector,
                            0
                            );
          #ifdef SHOW DEBUG
          blockprint("  ---- CTR PRELOAD(0) ", ctr preload);
          #endif
    for (j = 0; j < 16; j++) padded_buffer[j] = 0x00;
    for (j = 0; j < 8; j++)
    {
          padded buffer[j] = ciphertext[j+length-8];
    }
    aes128k128d(key, ctr preload, aes out);
    bitwise_xor(aes_out, padded_buffer, chain_buffer);
    #ifdef SHOW DEBUG
    blockprint(" ---- CTR PRELOAD(0) cipherstream ", aes out);
    blockprint(" ---- Plaintext MIC ", padded buffer);
    blockprint(" ---- Ciphertext MIC ", chain buffer);
```

```
    #endif
    for (j=0; j<8;j++) ciphertext[payload index++] = chain buffer[j];

    if ((ciphertext[1] & 0x40)==0x40) /* if CRC, compute it */
         {
                    crc32(&ciphertext[length], ciphertext, length);
                    length=length+4;
         }

    return length;
}


/*******************************************************/
/* int decrypt mpdu()                                  */
/* Decrypts a ciphertext mpdu in accordance with       */
/* the proposed 802.16 ccm specification.              */
/* Returns the resulting length of the packet.         */
/*******************************************************/
int decrypt mpdu(   unsigned char *key,
                    unsigned char *ciphertext mpdu,
                    int length,
                    unsigned char *plaintext,
                       int test case)
{
    /* PN */
    unsigned char pn vector[4];
    unsigned char pn byte;
    /* MIC working variables */
    int i;
    int j;
    int trailer length;
    int payload length;
    int num blocks;
    int payload remainder;
    int payload index;
    int crc ind;
    /* Initialization Blocks */
    unsigned char mic iv[16];
    unsigned char ctr_preload[16];

    /* Length adjustment on header */
    unsigned char lengthlsb;
    unsigned char lengthmsb;

    /* Intermediate Buffers */
    unsigned char chain buffer[16];
    unsigned char aes out[16];
    unsigned char padded buffer[16];

    /* MIC */
    unsigned char mic[8];

    /* detect CRC */
    crc ind=0;
    if ((ciphertext mpdu[1] & 0x40)==0x40) crc ind=1;

    for (i=0;i<16;i++)              /* Reset the buffers to zero */
    {
        mic iv[i] = 0x00;
        ctr preload[i] = 0x00;
        chain buffer[i] = 0x00;
        aes out[i] = 0x00;
        padded_buffer[i] = 0x00;
    }

    #ifdef SHOW PHASES
    printf("--DECRYPTING...\n");
    #endif

    #ifdef SHOW HEADER FIELDS
    printf("---- Key =             (msb) %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x
%02x %02x (lsb)\n",
                        key[0], key[1], key[2], key[3], key[4], key[5], key[6], key[7],
```

```
                        key[8], key[9], key[10], key[11], key[12], key[13], key[14], key[15]  );
    #endif


    for (i=0; i<4; i++)
    {
          pn vector[i] = ciphertext mpdu[6+i];
    }

    /* Copy ciphertext to plaintext */
    for (i=0; i<length; i++)
    {
        plaintext[i] = ciphertext mpdu[i];
    };

    /* Turn the PN into a vector of bytes, taking care  */
    /* not to be dependent on the endianess of the      */
    /* architecture we are running on                   */

    #ifdef SHOW HEADER FIELDS
    printf("---- PN =              0x%08lx\n", pnl );
    #endif

    payload length = length - 18;
    if (crc ind) payload length = length - 22; /* subtract header, pn, mic and crc length */

    /* Calculate number of 16 byte blocks in MPDU */

    payload_remainder = (payload_length) % 16;
    num blocks = (payload length) / 16;


    /**** Decrypt the payload and MIC ****/

    payload_index = 10; /* header + PN header */
    for (i=0; i< num blocks; i++)
    {
        construct ctr preload(
                                ctr preload,
                                ciphertext_mpdu,
                                pn vector,
                                i+1
                            );
        #ifdef SHOW_CTR_PRELOAD
        printf("---- CTR_PRELOAD(%i) = %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x
%02x %02x\n",
                            i+1, ctr preload[0],ctr preload[1],ctr preload[2],ctr preload[3],
                            ctr preload[4],ctr preload[5],ctr preload[6],ctr preload[7],
                            ctr preload[8],ctr preload[9],ctr preload[10],ctr preload[11],
                            ctr preload[12],ctr preload[13],ctr preload[14],ctr preload[15]);
        #endif

        aes128k128d(key, ctr_preload, aes_out);
          bitwise_xor(aes out, &ciphertext mpdu[payload index], chain buffer);
          #ifdef SHOW DEBUG
          blockprint("  ---- Cipherstream = ", aes out);
          blockprint("  ---- Plaintext    = ", &ciphertext[payload index]);
          blockprint("  ---- Ciphertext   = ", chain buffer);
          #endif
        for (j=0; j<16;j++) plaintext[payload index++] = chain buffer[j];
    }

    if (payload remainder > 0)          /* If there is a short final block, then pad it,*/
    {                                   /* encrypt it and copy the unpadded part back   */
        construct ctr preload(
                                ctr preload,
                                ciphertext mpdu,
                                pn vector,
                                num blocks+1
                            );

        for (j = 0; j < 16; j++) padded buffer[j] = 0x00;
        for (j = 0; j < payload remainder; j++)
        {
```

```
            padded buffer[j] = ciphertext mpdu[payload index+j];
        }
        aes128k128d(key, ctr preload, aes out);
          #ifdef SHOW DEBUG
          blockprint(" ---- final CTR_PRELOAD ", ctr_preload);
          blockprint(" ---- final cipherstream ", aes out);
          blockprint(" ---- final plaintest    ", padded buffer);
          #endif
        bitwise xor(aes out, padded buffer, chain buffer);
        for (j=0; j<payload remainder;j++) plaintext[payload index++] = chain buffer[j];
    }

    /* Decrypt the MIC */
    construct ctr preload(
                          ctr preload,
                          ciphertext_mpdu,
                          pn vector,
                          0
                          );
          #ifdef SHOW DEBUG
          blockprint("  ---- CTR PRELOAD(0) ", ctr preload);
          #endif
    for (j = 0; j < 16; j++) padded buffer[j] = 0x00;
    if (crc ind)
    {
          for (j = 0; j < 8; j++) padded buffer[j] = ciphertext mpdu[j+length-12];
    }
    else
    {
          for (j = 0; j < 8; j++) padded buffer[j] = ciphertext mpdu[j+length-8];
    }


    aes128k128d(key, ctr preload, aes out);
    bitwise_xor(aes_out, padded_buffer, chain_buffer);
    #ifdef SHOW DEBUG
    blockprint(" ---- CTR PRELOAD(0) cipherstream ", aes out);
    blockprint(" ---- Plaintext MIC ", padded buffer);
    blockprint(" ---- Ciphertext MIC ", chain buffer);
    #endif
    for (j=0; j<8;j++) plaintext[payload index++] = chain buffer[j];


    /* Calculate MIC */
    payload length = length - 18;
    if (crc_ind) payload_length = length - 22; /* subtract header, pn, mic and crc length */

    construct mic iv(
                          mic iv,
                          plaintext,
                          payload length,
                          pn vector
                          );

    payload index = 10;

    /* Calculate MIC */
    aes128k128d(key, mic iv, aes out);
    #ifdef SHOW_DEBUG
          blockprint("  ---- First MIC (lsb)", aes out);
    #endif

    /* iterate through each 16 byte payload block */
    for (i = 0; i < num_blocks; i++)
    {
        bitwise xor(aes out, &plaintext[payload index], chain buffer);
          #ifdef SHOW DEBUG
          blockprint("  ---- Subsequent MIC input (lsb)", chain buffer);
          #endif
        payload_index += 16;
        aes128k128d(key, chain buffer, aes out);
          #ifdef SHOW DEBUG
          blockprint("  ---- Subsequent MIC (lsb)", aes out);
          #endif
```

```
    }

    /* Add on the final payload block if it needs padding */
    if (payload remainder > 0)
    {
        for (j = 0; j < 16; j++) padded buffer[j] = 0x00;
        for (j = 0; j < payload remainder; j++)
        {
            padded buffer[j] = plaintext[payload index++];
        }
         #ifdef SHOW_DEBUG
         blockprint("  ---- padded mic input ", padded buffer);
         #endif
        bitwise xor(aes out, padded buffer, chain buffer);
         #ifdef SHOW DEBUG
         blockprint("  ---- final MIC input", chain_buffer);
         #endif
        aes128k128d(key, chain buffer, aes out);
         #ifdef SHOW DEBUG
         blockprint("  ---- final PT MIC   ", aes out);
         #endif
    }

    /* aes out contains padded mic, discard most significant    */
    /* 8 bytes to generate 64 bit MIC                           */

    for (j = 0 ; j < 8; j++) mic[j] = aes out[j];

    return(length);
}

/*************************************************/
/* get mpdu()                                    */
/* Copies an mpdu from the test case data        */
/*************************************************/
void get mpdu(      int test case,
                    unsigned char *plaintext)
{
    int i;
    unsigned char *ptr;

    ptr = test cases;
    for (i=0; i< (test case-1); i++)    /* Iterate through test cases */
    {
        ptr = ptr + test case length[i];
    }

    for (i=0; i< test case length[test case-1]; i++)
    {
        plaintext[i] = *ptr++;
    }

    crc8(&plaintext[5],plaintext,5);  /* Calculate the HCS */
}

/*******************************************************/
/* morph mpdu()                                        */
/* Turns a plaintext mpdu into one ready for encryption */
/* by adjusting the length to include the PN, by       */
/* extending the payload and moving the plaintext to   */
/* make space for the PN, setting the EC bit on and    */
/* recomputing the resulting HCS.                      */
/*******************************************************/

int morph mpdu (
         unsigned char *plaintext mpdu,
         int length,
         unsigned char *morphed mpdu,
         int testcase)
{
         /* prepare packet for entry into crypto block  */
         /* So add a PN, adjust length field for the PN */
         /* the crc (if there) and the mic. recomputie  */
         /* the HCS.                                    */
```

```
        int i;
        int morphed length;
        int gmh length;
        unsigned long int pn;

        /* copy header */
        for (i=0; i<6; i++) morphed mpdu[i] = plaintext mpdu[i];
        /* Shift payload right by 4 bytes to make room for the PN */
        for (i=length-1; i>5;i--) morphed mpdu[i+4]=plaintext mpdu[i];

        /* insert the PN */
        pn = test case pnl[testcase-1];
        morphed mpdu[6]=pn % 256;
        morphed mpdu[7]=(pn/256) % 256;
        morphed mpdu[8]=(pn/(256*256)) % 256;
        morphed mpdu[9]=(pn/(256*256*256)) % 256;

        morphed length = length+4;

        /* compute the length field in the GMH             */
        /* = header + payload + pn + mic + crc if present */

        gmh length = length + 4 + 8;
        if ((plaintext mpdu[1]&0x40)==0x40) gmh length+=4; /* add 4 for crc */

        morphed mpdu[2] = gmh length % 256;
        morphed mpdu[1] = plaintext mpdu[1] & 0xf8; /* clear lower 3 bits */
        morphed mpdu[1] = morphed mpdu[1] | (((gmh length/256) % 256) & 0x7);

        /* set the encryption bit on */
        morphed mpdu[0] = morphed mpdu[0] | 0x40;

        /* compute the HCS */
        crc8(&morphed mpdu[5],morphed mpdu,5);

        return (morphed length);
}


/*********************************************************/
/* create test case()                                    */
/* Generates a test case, of the specified payload length  */
/* with the CRC bit, key and pn as specified.            */
/* The test case num should increment.                   */
/* The returned pointer should be passed into *index so    */
/* it knows where in the test case array the next test     */
/* case should be placed.                                */
/* inc rndn controls whether the payload is random or      */
/* incrementing.                                         */
/*********************************************************/

unsigned int create test case(
                        int test_case_num,
                        unsigned int index,
                        int length,
                        int crc,
                        int enc decn,
                        int inc rndn
                    )
{
        int i;

        test case length[test case num-1] = length+6; /* add GMH length */
        test_case_enc_decn[test_case_num-1] = enc_decn;


        test case pnl[test case num-1]=random();

        for(i=0;i<16;i++)
        {
                keys[(16*(test case num-1))+i]=(unsigned char)(random() % 256);
        }

        /* Fill the header */
```

```
        test_cases[index]=0x00;
        if (crc) {
                test_cases[index+1] = 0x40;
        } else {
                test_cases[index+1] = 0x00;
        }

        test_cases[index+2] = (length+6) % 256;
        test_cases[index+1] |= (((length+6) / 256) % 256) & 0x7;

        test_cases[index+3] = (unsigned char)random();
        test_cases[index+4] = (unsigned char)random();
        test_cases[index+5] = 0x00;

        /* Fill the payload with random or sequential data */
        if (inc_rndn)
        {
                for (i=0;i<length;i++)
                {
                        test_cases[6+(index++)]=((unsigned char)i) % 256;
                }
        }
        else
        {
                for (i=0;i<length;i++)
                {
                        test_cases[6+(index++)]=((unsigned char)random()) % 256;
                }
        }

return(index+6);

}

/*************************************************/
/* Iterate around and make some test cases       */
/*************************************************/

void make_test_cases()
{
        int crc;
        int i;
        int enc_decn;
        int length;
        int index;
        int test_case_num;

        num_test_cases = 0;
        index = 0;
        test_case_num=1;

        for (i=0;i<21;i++)
        {
                if (i==0) length=1;
                if (i==1) length=2;
                if (i==2) length=3;
                if (i==3) length=4;
                if (i==4) length=5;
                if (i==5) length=6;
                if (i==6) length=7;
                if (i==7) length=8;
                if (i==8) length=9;
                if (i==9) length=10;
                if (i==10) length=11;
                if (i==11) length=12;
                if (i==12) length=13;
                if (i==13) length=14;
                if (i==14) length=15;
                if (i==15) length=16;
                if (i==16) length=17;
                if (i==17) length=23;
                if (i==18) length=31;
                if (i==19) length=32;
                if (i==20) length=33;
```

```
                    for (crc=0;crc<2;crc++)
                    {
                                for (enc decn=0;enc decn<2;enc decn++)
                                {
                                        num test cases++;
                                        index = create test case(
                                                                test case num++,
                                                                index,
                                                                length,
                                                                crc,
                                                                enc decn,
                                                                1
                                                        );
                                }
                    }
          }
}

/*************************************************/
/* main()                                        */
/* Iterate through the test cases, passing them  */
/* through the ccm algorithm to produce test     */
/* vectors                                       */
/*************************************************/

int main()
{
    int draft_output_length;
    int length;
    int plaintext length;
    int test case;
    int header length;
    int payload length;
    int num_blocks;
    int block remainder;
    int i;
    int j;
    int trailer length;
    unsigned char crc[4];
    unsigned char plaintext mpdu[3000];
    unsigned char morphed mpdu[3000];
    unsigned char ciphertext mpdu[3000];
    unsigned char decrypted mpdu[3000];
    unsigned char *key;
    unsigned int pn;

    make test cases();

    for (test case = 1; test case < (num test cases+1); test case++)
    {
        key = keys + (16 * (test case-1));
        get_mpdu(test_case, plaintext_mpdu);

          enc decn = test case enc decn[test case-1];
          plaintext length = test case length[test case-1];
        length = morph mpdu( plaintext mpdu,     /* Perform frame expansion and change GMH */
                    plaintext length,
                       morphed_mpdu,
                    test case);

          printf("EXAMPLE #%d\n", test case);
          blockprint payload("\tPlaintext MPDU",plaintext mpdu,plaintext length);
          blockprint_key(key);
          pn = morphed mpdu[6];                  /* extract the pn */
          pn += 256*morphed mpdu[7];
          pn += 256*256*morphed mpdu[8];
          pn += 256*256*256*morphed mpdu[9];
          printf("Packet Number = 0x%08x\n",pn);

        length = encrypt mpdu(
                        key,
                       morphed mpdu,
                            length,
```

```
                        ciphertext_mpdu,
                                test_case);

        blockprint_payload("\tCiphertext mpdu",ciphertext_mpdu,length);

        length = decrypt_mpdu(
                                key,
                                ciphertext_mpdu,
                                length,
                                decrypted_mpdu,
                                test_case);

        blockprint_payload("\tDecrypted MPDU",decrypted_mpdu,length);
    }
    return 0;
}
```