<table>
<thead>
<tr>
<th>Project</th>
<th>IEEE 802.16 Broadband Wireless Access Working Group <a href="http://ieee802.org/16">http://ieee802.org/16</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>CCM Endianness Disambiguation</td>
</tr>
<tr>
<td>Date Submitted</td>
<td>2005-04-22</td>
</tr>
<tr>
<td>Source(s)</td>
<td>David Johnston</td>
</tr>
<tr>
<td></td>
<td>Intel Corporation</td>
</tr>
<tr>
<td></td>
<td>Voice: (503) 264 3855</td>
</tr>
<tr>
<td></td>
<td>Email: <a href="mailto:dj.johnston@intel.com">dj.johnston@intel.com</a></td>
</tr>
<tr>
<td>Re:</td>
<td>Letter Ballot #17a, P802.16-2004/Cor1/D2</td>
</tr>
</tbody>
</table>

**Abstract**

**Purpose**

This document corrects out of scope text and resolves an ambiguity 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 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 endian MSB 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 ICVCiphertext 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.
‘\(T\text{len}\)’ shall equal 64 and \(t\) shall equal 8, meaning, the number of octets in the Message Authentication Code field \(M\) shall be set to 8. Consistent with the CCM specification the 3 bit binary encoding \([\frac{(t-2)}{2}]\) 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 \([q-1]\) 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.

<table>
<thead>
<tr>
<th>Octet Number</th>
<th>Field</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ... 4</td>
<td>Generic MAC Header</td>
<td>Omitting HCS</td>
</tr>
<tr>
<td>5 ... 8</td>
<td>Reserved</td>
<td>Packet number field from payload</td>
</tr>
<tr>
<td>9 ... 12</td>
<td>PN</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Byte within MIC_IV</th>
<th>Octet Number</th>
<th>0</th>
<th>1</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte Significance</td>
<td>Number of</td>
<td>msb</td>
<td>lsb</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of Bytes</td>
<td>Bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field Contents</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flag</td>
<td></td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Nonce</td>
<td></td>
<td>Counter</td>
<td></td>
</tr>
<tr>
<td>0x19</td>
<td></td>
<td></td>
<td>As Specified in Figure 135a</td>
<td>Length of plaintext payload data part not including padding</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 \(C_{\text{tr} i} \#i\) are formatted as shown in Figure 137.

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

<table>
<thead>
<tr>
<th>Byte within CTR_IV</th>
<th>Octet Number</th>
<th>0</th>
<th>1</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte Significance</td>
<td>Number of</td>
<td>msb</td>
<td>lsb</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of Bytes</td>
<td>Bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field Contents</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flag</td>
<td></td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Nonce</td>
<td></td>
<td>Counter</td>
<td></td>
</tr>
<tr>
<td>0x01</td>
<td></td>
<td></td>
<td>As Specified in Figure 135a</td>
<td>(i) Length of data part not including padding</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 137 – Construction of \textit{counter blocks} $C_{tr_j} A_i$.

[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

Ciphertext PDU where TEK = 0xB74EB0E4F81AD63D121B7E9ACCD268F 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];
unsigned char lookahead_table[1024] =

0x2e, 0x7a, 0x66, 0xb3, 0xb8, 0x4a, 0x61, 0xc4, 0x02, 0x1b, 0x68, 0x5d, 0x94, 0x2b, 0x6f, 0x2a,
0x05, 0x36, 0xd0, 0xba, 0x93, 0x06, 0xd7, 0xcd, 0x29, 0x57, 0xde, 0x54, 0xbf, 0x67, 0xd9, 0x23,
0x1c, 0xf2, 0xbd, 0xbd, 0x8a, 0xc2, 0xba, 0xca, 0x30, 0x93, 0xb3, 0x53, 0xa6, 0xa3, 0xb4, 0x24,
0x4a, 0x6a, 0xd1, 0xae, 0xdc, 0x5a, 0xd6, 0xd9, 0x66, 0x0b, 0xdf, 0x40, 0xf0, 0x3b, 0xd8, 0x37,
0x78, 0xe2, 0xa0, 0xee, 0xd2, 0x0d, 0xd7, 0x54, 0x83, 0x04, 0x4e, 0xc2, 0xb3, 0x03, 0x39,
0xff, 0x9e, 0x65, 0x8f, 0x69, 0xae, 0x62, 0xf8, 0xd3, 0xff, 0x6b, 0x61, 0x48, 0x01, 0xb9, 0x13,
0xd6, 0xd9, 0x2f, 0x02, 0x20, 0x60, 0x6b, 0x06, 0xf3, 0xd5, 0x62, 0x60, 0x0f, 0x16, 0x6f, 0x25,
0x4e, 0x6a, 0xd9, 0x2f, 0x02, 0x20, 0x60, 0x2f, 0x02, 0x20, 0x60, 0x0f, 0x16, 0x6f, 0x25,
0x4e, 0x6a, 0xd9, 0x2f, 0x02, 0x20, 0x60, 0x2f, 0x02, 0x20, 0x60, 0x0f, 0x16, 0x6f, 0x25,
0x4e, 0x6a, 0xd9, 0x2f, 0x02, 0x20, 0x60, 0x2f, 0x02, 0x20, 0x60, 0x0f, 0x16, 0x6f, 0x25,
0x4e, 0x6a, 0xd9, 0x2f, 0x02, 0x20, 0x60, 0x2f, 0x02, 0x20, 0x60, 0x0f, 0x16, 0x6f, 0x25,
0x4e, 0x6a, 0xd9, 0x2f, 0x02, 0x20, 0x60, 0x2f, 0x02, 0x20, 0x60, 0x0f, 0x16, 0x6f, 0x25,
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, 0xA5, 0x1F, 0x18, 0x11, 0x16,
  0x03, 0x04, 0x0D, 0x0A, 0x57, 0x50, 0x59, 0x5E,
  0x4B, 0x4C, 0x45, 0x42, 0x6F, 0x68, 0x61, 0x66,
  0x73, 0x74, 0x7A, 0x79, 0x87, 0x80, 0x99, 0x9C,
  0xB1, 0xB6, 0xBF, 0xB8, 0xAD, 0xAA, 0xA3, 0xA4,
  0xF9, 0xFE, 0xF7, 0xF0, 0xE5, 0xE2, 0xEB, 0xEC,
  0xC1, 0xC6, 0xC7, 0xC0, 0xC9, 0xCE, 0xDB, 0xDC,
  0xD5, 0xD2, 0xFF, 0xF8, 0xF1, 0xF6, 0xE3, 0xE4,
  0xB7, 0xB0, 0xB9, 0xBE, 0xA8, 0xA9, 0xA0, 0xA7,
  0xB2, 0xB5, 0xBC, 0xBB, 0x96, 0x91, 0x98, 0x9F,
  0x8A, 0x8D, 0x84, 0x83, 0xDE, 0xD9, 0xD0, 0xDF,
  0x3E, 0x39, 0x30, 0x37, 0x22, 0x25, 0x2C, 0x2B,
  0x06, 0x01, 0x08, 0x0F, 0x1A, 0x1D, 0x14, 0x13,
  0xAE, 0xA9, 0xA0, 0xA7, 0xB2, 0xB5, 0xB0, 0xBB,
  0x31, 0x18, 0x15, 0x43, 0x4D, 0x33, 0x85, 0x45,
  0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, 0x51,
  0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xe8,
  0xb2, 0x75, 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e,
  0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3,
  0x2f, 0x84, 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5,
  0xe4, 0xa9, 0x6b, 0x50, 0x3d, 0x64, 0x5d, 0x19,
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0xe0, 0x32,
  0xe5, 0x0a, 0x49, 0x2d, 0x0f, 0xb0, 0x54, 0xbb,
  0x16
};

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, 0x27, 0xb2, 0x75, 0x09,
  0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52,
  0x3b, 0x6b, 0xb3, 0x29, 0xe3, 0x2f, 0x84, 0x53,
  0xd1, 0x00, 0xed, 0x20, 0xc0, 0xb1, 0x5b, 0x6a,
  0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, 0x0d,
  0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45,
  0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, 0x51,
  0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xe8,
  0xb2, 0x75, 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e,
  0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3,
  0x2f, 0x84, 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5,
  0xe4, 0xa9, 0x6b, 0x50, 0x3d, 0x64, 0x5d, 0x19,
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0xe0, 0x32,
  0xe5, 0x0a, 0x49, 0x2d, 0x0f, 0xb0, 0x54, 0xbb,
  0x16
};

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%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 j;
    int i;
    unsigned char *ptr;

    ptr = payload;
    blocks = length/16;
    residue = length % 16;
    printf("%s\n", 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]);
```c
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_halves[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_halves[0] = in[2]; /* Swap halves */
    swap_halves[1] = in[3];
    swap_halves[2] = in[0];
    swap_halves[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[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_halves, 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);
        }
    }

}*/

void crc8(unsigned char *crc, unsigned char *data, int length)
{ int i;
```c
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)
{
    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[3] = table_entry[3];
    }
    crc[0] = crc[0] ^ 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)
{
    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++)
    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.                               */
/*************************************/

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];
unsigned char mic_temp[8];

unsigned char mic_remaining[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...
");
#endif

#ifdef SHOW_HEADER_FIELDS
        printf("---- Key =
0x%02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x
16\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,
                mic_remaining);
pn_vector
)

#ifdef SHOW_INIT_BLOCKS
printf("---- MIC_IB = (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);
    #=>show_DEBUG
    blockprint(" ---- Subsequent MIC input (lsb)", chain_buffer);
    #endif
    payload_index += 16;
aes128k128d(key, chain_buffer, aes_out);
    #=>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++];
    }
    #=>show_DEBUG
    blockprint(" ---- padded mic input ", padded_buffer);
    #endif
    bitwise_xor(aes_out, padded_buffer, chain_buffer);
    #=>show_DEBUG
    blockprint(" ---- final MIC input", chain_buffer);
    #endif
    aes128k128d(key, chain_buffer, aes_out);
    #=>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++)
{
    ciphertext[payload_index + j] = mic[j];
    payload_index += 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 (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 plaintext ", 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,
    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 < payload_remainder; j++)
{
    padded_buffer[j] = ciphertext[j+length-8];
}
aes128k128d(key, ctr_preload, aes_out);
blockprint(" ---- CTR_PRELOAD(0) cipherstream ", aes_out);
blockprint(" ---- Plaintext MIC ", padded_buffer);
blockprint(" ---- Ciphertext MIC ", chain_buffer);
}
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 aes_out[16];
    unsigned char chain_buffer[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_HEADERIELDS
    printf("--- Key = (msb) %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x \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(
        ctrpreload, 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\n", "
    i+1, ctrpreload[0],ctrpreload[1],ctrpreload[2],ctrpreload[3],
    ctrpreload[4],ctrpreload[5],ctrpreload[6],ctrpreload[7],
    ctrpreload[8],ctrpreload[9],ctrpreload[10],ctrpreload[11],
    ctrpreload[12],ctrpreload[13],ctrpreload[14],ctrpreload[15]);
#endif
aes128k128d(key, ctrpreload, aesout);
bitwise_xor(aesout, ciphertext_mpdu[payload_index], chain_buffer);
#ifdef SHOW_DEBUG
    blockprint(" ---- Cipherstream = ", aesout);
    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(
        ctrpreload, ciphertext_mpdu, pn_vector,
        num_blocks+1
    );
    for (j = 0; j < payload_remainder; 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);
#endif 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 (i=0; i<payload_remainder; i++) plaintext[payload_index++] = chain_buffer[j];

/* Decrypt the MIC */
construct_ctr_preload(
    ctr_preload,
    ciphertext_mpdu,
    pn_vector,
    0
);
#endif SHOW DEBUG
blockprint("  ---- CTR_PRELOAD(0) ", ctr_preload);
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);
#endif 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);
#endif 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);
#endif SHOW DEBUG
blockprint("---- Subsequent MIC input (lsb) ", chain_buffer);
#endif
payload_index += 16;
aes128k128d(key, chain_buffer, aes_out);
#endif 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,
                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 */
```c
int 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) & 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=18;
        if (i==18) length=19;
        if (i==19) length=20;
        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[7]; /* extract the pn */
        pn += 256*morphed mpdu[8];
        pn += 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("Ciphertext mpdu", ciphertext_mpdu, length);
    length = decrypt_mpdu(
        key,
        ciphertext_mpdu,
        length,
        decrypted_mpdu,
        test_case);
    blockprint_payload("Decrypted MPDU", decrypted_mpdu, length);
  }
  return 0;
}