AN821

Advanced Encryption Standard Using the PIC16XXX

Author: *Caio Gubel*
Microchip Technology Inc.

INTRODUCTION

One of the most widely used block cipher algorithms is the Data Encryption Standard (DES), adopted in 1977 by the American National Standards Institute (ANSI).

After more than twenty years of use with continuous aging due to advances in cryptography, the National Institute of Standards and Technology (NIST) on September 12, 1997, started a process to stimulate the development and submission of alternatives to the DES. Twenty-one algorithms were analyzed in the first round and five algorithms were analyzed in the second round. On October 2, 2000 the NIST announced that the new encryption technique, named Advanced Encryption Standard (AES), would use the Rijndael algorithm, designed by two well-known specialists, Joan Daemen and Vincent Rijmen from Belgium. The new AES will be used to protect sensitive information of federal computer systems, as well as many private businesses.

FUNDAMENTAL ENCRYPTION OVERVIEW

Throughout history, mankind has faced the problem of storing and transmitting sensitive information in a way that could guarantee both reliable and easy access to authorized persons and prevent undue and illegal access. This has led to the development of many ingenious methods to cipher and decipher data.

The phenomenal development of computers and expansion of digital information exchange has led to a fundamental problem related to the availability, control, and security of data. To deal with this problem, several computer based encryption technologies and standards were developed. One of the most popular methods developed is the Block Cipher.

An algorithm that uses a key is, in general, much more secure. If the algorithm itself is secure in its design, then the data is secure (as long as the key is secure) even if the encryption algorithm is known. The only way to decipher the message is through the use of the correct key.

There are two basic types of keys:

1. Symmetric
2. Public

In a symmetric key algorithm, both encryption and decryption processes use the same key, which must be kept secret. In a public key system, two keys are used: one public (used to cipher messages) and another, private and secret (used to decipher the message).

In the AES (Symmetric Key) method, the plain text is broken in several blocks of the same size. For example, the following plain text:

"I pass death with the dying and birth with new washed baby, and am not contained between my hat and boots." (Walt Whitman)

This text is broken into 16-byte (arbitrary size) chunks as shown in Example 1:

EXAMPLE 1: PLAIN TEXT DIVIDED INTO 16-BYTE BLOCKS

`I pass death wit`	`h the dying and`	`birth with new w`	`ashed baby, and`
`am not contained`	`between my hat a`	`nd boots\0tuvwxyz`(1,2)	

Note 1: \0 is a single character and represents End of String.

2: Through a process called ‘padding’, an incomplete 16-byte text block may be completed using random characters like “tuvwxyz.”

Next, each block (plus an encryption key) is combined together using an algorithm that executes a complex function resulting in the production of a ciphered block. See Example 2.

EXAMPLE 2: PLAIN TEXT BLOCK TO CIPHERED BLOCK PROCESS

Plain Text Block	+	Key	Encryption Algorithm	Ciphered Block
`I pass death wit`	+	`waltwhitman,poet`	→	`dfkei5k7kkko23aq`

The deciphering process takes the encrypted block plus the encryption key and passes them through an algorithm that executes the reverse process, resulting in a plain text block. See Example 3.

EXAMPLE 3: CIPHERED BLOCK TO PLAIN TEXT BLOCK PROCESS

Ciphered Block	+	Key	Decryption Algorithm	Plain Text Block
`dfkei5k7kkko23aq`	+	`waltwhitman,poet`	→	`I pass death wit`

HOW THE AES ALGORITHM IS IMPLEMENTED IN A PIC16XXX MICROCONTROLLER

The AES Algorithm - An Overview

AES is a symmetric key block cipher algorithm that may use three different block and key sizes:

- 16-byte - 128 bits
- 24-byte - 192 bits
- 32-byte - 256 bits

The algorithm executes a series of rounds. The intermediate results of the rounds over the block are called **states**.

The number of round transformations is variable and a function of the sizes of the key and the text, shown as follows:

TABLE 1: ROUND TRANSFORMATIONS REQUIRED

	16-byte block	**24-byte block**	**32-byte block**
16-byte key	10*	12	14
24-byte key	12	12	14
32-byte key	14	14	14

* Chosen for implementation in this Application Note.

For these transformations, the state (block) and the key are both taken as matrixes, as shown in Table 2 and Table 3.

TABLE 2: BLOCK MATRIX

Block[0]	Block[4]	Block[8]	Block[12]	Block[16]	Block[20]	Block[24]	Block[28]
Block[1]	Block[5]	Block[9]	Block[13]	Block[17]	Block[21]	Block[25]	Block[29]
Block[2]	Block[6]	Block[10]	Block[14]	Block[18]	Block[22]	Block[26]	Block[30]
Block[3]	Block[7]	Block[11]	Block[15]	Block[19]	Block[23]	Block[27]	Block[31]

TABLE 3: KEY MATRIX

Key[0]	Key[4]	Key[8]	Key[12]	Key[16]	Key[20]	Key[24]	Key[28]
Key[1]	Key[5]	Key[9]	Key[13]	Key[17]	Key[21]	Key[25]	Key[29]
Key[2]	Key[6]	Key[10]	Key[14]	Key[18]	Key[22]	Key[26]	Key[30]
Key[3]	Key[7]	Key[11]	Key[15]	Key[19]	Key[23]	Key[27]	Key[31]

Note: In the 16-byte (128-bit) implementation, both matrixes are 4x4.

Key Schedule - Expansion and Selection: Encryption

In order to prepare for the round transformations, a “key schedule” operation must be executed. This operation uses the original key to create several round keys. Each round key, including the original one, will be used in one of the rounds.

This operation is performed in two steps:

1. Key Expansion -> takes the key from the previous round and expands it to create the key for the next round, according to the C code for 16-byte (128-bit key) shown below:
2. Round Key Selection -> takes the round buffer in blocks of 16 bytes (for 128-bit keys), so that the keys (taken in bytes) for a given round "i" are:

W[i] [0]	W[i] [4]	W[i] [8]	W[i] [12]
W[i] [1]	W[i] [5]	W[i] [9]	W[i] [13]
W[i] [2]	W[i] [6]	W[i] [10]	W[i] [14]
W[i] [3]	W[i] [7]	W[i] [11]	W[i] [15]

After the `enc_key_schedule`, an initial key addition must be executed:

Initial `key_addition`:

Before the first round of encryption, an initial `key_addition` is performed. This operation executes a simple XOR of the state with the initial round key. In C for 16-byte (128-bit) key and block:

```
for(i=0;i<16;i++)
    Block[i] ^= W[0][i];
```

The Structure of the Round Transformations: Encryption

In the encryption process, each of the ten rounds (with the exception of the last one) is composed of four stages:

- `byte_sub`
- `shift_row`
- `mix_column`
- `key_addition`

The last round doesn’t execute the `mix_column` stage, thus the sequence is:

- `byte_sub`
- `shift_row`
- `key_addition`

C Code for 16-Byte Key Expansion

```
KeyExpansion(byte Key[], byte W[][])
{
  byte rcon=1;                                 // initial value of round constant
  for (j=0;j<16;j++)                           // first key expansion no changed
       w[0][j] = key[j];
  for(i = 1; i<11; i++)
  {
   for(j = 0; j<16; j++)
   {
    if(j<4)                                    // calculate S_Box based values
       W[i][j] = W[i-1][j] ^ S_box(W[i-1][12+((j+1)%4)]];
    else
       W[i][j] = W[i-1][j] ^ w[i][j-4];         //
     if((j%4) == 0)
       W[i][j] ^= rcon;
  }
  rcon = xtime(rcon);                          // calculate rcon for next round
 }
}
```

with: `Rcon` = {0x36, 0x1B, 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01}; where `Rcon` represents a vector of round constants.

TABLE 4: S-BOX OR ENCRYPTION SUBSTITUTION TABLE (VALUES IN HEXADECIMAL)

		y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
x	**0**	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

DESCRIPTION OF ENCRYPTION STAGES:

`byte_sub`:

In this stage, each byte of the block matrix is replaced by the content of the S-box at the position defined by the byte that is going to be substituted. In this case, the S-box or substitution table may be seen as a 256 byte invertible vector/matrix used to map the substitution process.

This is equivalent to the following C language fragment:

```
for(i=0;i<BLOCKSIZE;i++)
   block[i]=S_box[block[i]];
```

EXAMPLE 4: S-BOX SUBSTITUTION

If block[0] = 0x41, then in the S-Box table go to 4 in ‘x’ axis and 1 in ‘y’ axis to get S-box[0x41] -> 0x83 thus the contents of block[0] = 0x83

`shift_row`:

The second stage of the round process executes a cyclical shift (rotate left) of the rows of the state table. The row number 0 is not affected, and the other rows are shifted according to Table 5:

TABLE 5: ENCRYPTION CYCLICAL SHIFT TABLE

	# shifts of row 1	**# shifts of row 2**	**# shifts of row 3**
16-byte block	1	2	3
24-byte block	1	2	3
32-byte block	1	3	4

EXAMPLE 5: `shift_row` TRANSFORMATION

For the 16-byte block and key version (the implemented one), a state table with the following content:

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Becomes the following after the `shift_row` transformation:

A	B	C	D
F	G	H	E
K	L	I	J
P	M	N	O

`mix_column`:

This operation, described in the AES Proposal (see Chapter 2 - Mathematical Preliminaries and Section 4.2.3 - The MixColumn Transformation), comprises the multiplication of each column a_i of the state by a fixed matrix c(x) following some special rules (Polynomials with coefficients in $GF(2^8)$), see Example 6.

The general form of this matrix multiplication is shown in the following equation:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \overset{\text{FIXED MATRIX } c(x)}{\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}} \mathrm{X} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

An example of matrix multiplication can be seen in Example 7.

`key_addition`:

This step takes the next round key and executes an XOR with the state in the form :

```
for(i=0;i<16;i++)
   Block[i] ^= W[round+1][i];
```

EXAMPLE 6: MATRIX MULTIPLICATION CONCEPTS USING SPECIAL RULES

First, let's define the `xtime` operation:

```
if(a<0x80)
    a<<=1;
else
    a=(a<<1)^0x1b;
```

From this we can see that the `xtime` operation for values lower than 0x80 is equivalent to a shift left (multiply by 2). For values bigger than or equal to 0x80, an extra XOR with 0x1B is necessary.

Multiply c[i][0]=0xA7 by a[i]=0x0D

Where

0xA7 ● 0x01=0xA7;

0xA7 ● 0x02=`xtime`(0xA7)=0x55;   (((0xA7)<<1) ^0x1B)

0xA7 ● 0x04=`xtime`(`xtime`(0xA7))=0xAA;   ((0x55)<<1)

0xA7 ● 0x08=`xtime`(`xtime`(`xtime`(0xA7)))=0x4F;   (((0xAA)<<1) ^0x1B)

Therefore, 0xA7 ● 0x0D may be written as:

(0xA7 ● 0x01) ⊕ (0xA7 ● 0x04) ⊕ (0xA7 ● 0x08) = 0xA7 ⊕ 0xAA ⊕ 0x42

The partial results are not added, but instead they are XORed to generate the new terms of the column.

Note: ⊕ means XOR.

EXAMPLE 7: MATRIX MULTIPLICATION

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \underbrace{\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}}_{} \times \begin{bmatrix} 5A \\ 11 \\ FD \\ 89 \end{bmatrix}$$

FIXED MATRIX c(x)

b_0 = 2 ● 5A ⊕ 3 ● 11 ⊕ 1 ● FD ⊕ 1 ● 89

where

2 ● 5A = xtime (5A) = B4

3 ● 11 = 11 ⊕ xtime (11) = 11 ⊕ 22 = 0x33

1 ● FD = FD

1 ● 89 = 89

therefore

b_0 = B4 ⊕ 33 ⊕ FD ⊕ 89 = F3

----

b_1 = 1 ● 5A ⊕ 2 ● 11 ⊕ 3 ● FD ⊕ 1 ● 89

where

1 ● 5A = 5A

2 ● 11 = xtime (11) = 22

3 ● FD = FD ⊕ xtime (FD) = FD ⊕ E1 = 1C

1 ● 89 = 89

therefore

b_1 = 5A ⊕ 22 ⊕ 1C ⊕ 89 = ED

----

b_2 = 1 ● 5A ⊕ 1 ● 11 ⊕ 2 ● FD ⊕ 3 ● 89

where

1 ● 5A = 5A

1 ● 11 = 11

2 ● FD = xtime (FD) = E1

3 ● 89 = 89 ⊕ xtime (89) = 80

therefore

b_2 = 5A ⊕ 11 ⊕ E1 ⊕ 80 = 2A

----

b_3 = 3 ● 5A ⊕ 1 ● 11 ⊕ 1 ● FD ⊕ 2 ● 89

where

3 ● 5A = 5A ⊕ xtime (5A) = 5A ⊕ B4 = EE

1 ● 11 = 11

1 ● FD = FD

2 ● 89 = xtime (89) = 09

therefore

b_3 = EE ⊕ 11 ⊕ FD ⊕ 09 = 0B

Now, the general form of b[i] = c[i][0] • a[0] ⊕ c[i][1] • a[1] ⊕ c[i][2] • a[2] ⊕ c[i][3] • a[3];

Observation1: The partial results are XORed (⊕) instead of added.

Observation2: In this multiplication (•), each time a carry bit occurs, the result must be XORed with 0x1B (xtime).

Key Schedule: Expansion and Selection: Decryption

In order to prepare for the round transformations, a "key schedule" operation must be executed. This function is basically the same as the one used in encryption; the difference is that, in the encryption process, the round keys are used in the direct order, W[0], W[1], W[2],...., while in the decryption process, they are used in the reverse order: W[10], W[9], W[8], ...

After the `dec_key_schedule`, an initial key addition must be executed:

Initial `key_addition`:

> Before the first round of decryption, an initial `key_addition` is performed. This operation executes a simple XOR of the state with the final round key. In C (for 128-bit key and block):
>
> ```
> for(i=0;i<16;i++)
>    Block[i] ^= W[10][i];
> ```

The Structure of the Round Transformations: Decryption

In the decryption process, each of the ten rounds (with the exception of the first one) is composed of four stages:

- `byte_sub`
- `shift_row`
- `inv_mix_column`
- `key_addition`

The first round doesn't execute the `inv_mix_column` stage, thus the sequence is:

- `byte_sub`
- `shift_row`
- `key_addition`

TABLE 6: Si-BOX OR DECRYPTION SUBSTITUTION TABLE (VALUES IN HEXADECIMAL)

		y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
x	**0**	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

DESCRIPTION OF DECRYPTION STAGES:

`byte_sub`:

In this stage, each byte of the block is replaced by the content of the Si-box at the position defined by the byte that is going to be substituted. In this case, the Si-box or substitution table, may be seen as a 256-byte invertible vector/matrix, used to map the substitution process in the inverse direction taken by the S-box.

This is equivalent to the following C language fragment:

```
for(i=0;i<BLOCKSIZE;i++)
   block[i]=Si_box[block[i]];
```

This relationship of boxes may be understood easily as follows:

```
S-box[i] = j
Si-box[j] = i
```

EXAMPLE 8: Si BOX SUBSTITUTION

If block[0] = 0x83, then in the Si-Box table, go to 8 in ‘x’ axis and 3 in ‘y’ axis to get Si-box[0x83] -> 0x41, thus the contents of block[0] = 0x41

`shift_row`:

The second stage of the round process executes a cyclical shift (rotate left) of the rows of the state. The row number 0 is not affected and the other rows are shifted according to Table 7:

TABLE 7: DECRYPTION CYCLICAL SHIFT TABLE

	# shifts of row 1	# shifts of row 2	# shifts of row 3
16 byte block	3	2	1
24 byte block	5	4	3
32 byte block	7	5	4

EXAMPLE 9: `shift-row` TRANSFORMATION

For the 16-byte block and key version (the implemented one), a state with the following content:

A	B	C	D
F	G	H	E
K	L	I	J
P	M	N	O

Becomes the following after the `shift_row` transformation:

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

`inv_mix_column`:

The implementation follows the same rules applied to the `mix_column` routine, except the `inv_mix_column` uses the fixed matrix c(x) shown below. See the `mix_column` section under the description of Encryption Stages for an explanation of the matrix multiplication used in the AES algorithm.

FIXED MATRIX c(x)

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} X \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

`key_addition`:

This step takes the next round key and executes an XOR with the state in the form: (`NUMROUNDS`=11 for 16-byte block).

```
for(i=0;i<16;i++)
   Block[i]^=W[NUMROUNDS- rounds][i];
```

Program Structure

The general structure of the encryption program is:

```
key_addition(block,key);   // initial key addition
 rounds =10;
while ( rounds-- )         // loop 10x
{
  substitution_S(block);
  enc_shift_row(block);
  if( rounds != 1 )        // last round is done without mix_column
     mix_column(block);
  enc_key_schedule(key);   // direct key_schedule executed on-the-fly
  key_addition(block,key);
}
```

The general structure of the decryption program is:

```
init_decryption_key(key)   // create the initial decryption key from initial key
rounds=10;
key_addition(block,key);   // initial
while ( rounds-- )         // loop 10x
{
  substitution_Si(block);  // substitution with Si_box table
  dec_shift_row(block);
  if( rounds != 10 )       // first round is done without inv_mix_column
     inv_mix_column(block);
  dec_key_schedule(key);   // inverse key_schedule executed on-the-fly
  key_addition(block,key);
}
```

`mix_column` Optimization

The original `mix_column` transformation, as described in the reference implementation of AES, is time consuming. Therefore, the following optimized equivalent form is recommended:

```
for(i=0;i<4;i++)
{
 Tmp = block[i+0] ^ block[i+0x1] ^ block[i+0x2] ^ block[i+0x3];
 Block[i=0x0] ^= Tmp ^ xtime(block[i+0x0]^block[i+0x1])
 Block[i+0x1] ^= Tmp ^ xtime(block[i+0x1]^block[i+0x2])
 Block[i+0x2] ^= Tmp ^ xtime(block[i+0x2]^block[i+0x3])
 Block[i+0x3] ^= Tmp ^ xtime(block[i+0x3]^block[i+0x0])
}
```

`inv_mix_column` Optimization

The original `inv_mix_column` transformation, as described in the reference implementation of AES, is time consuming, so the optimized equivalent form is used:

```
for(i=0;i<4;i++)
{
 Tmp0 = block[i+0] ^ block[i+0x1] ^ block[i+0x2] ^ block[i+0x3];
 Tmp1 = xtime(block[i+0] ^ block[i+0x2]);
 Tmp2 = xtime(block[i+1] ^ block[i+0x3]);
 Tmp3 = xtime( xtime( Tmp1 ^ Tmp2 )) ^ Tmp0;
 Block[i=0x0] ^= xtime(block[i+0x0]^block[i+0x1] ^ Tmp1) ^ Tmp3;
 Block[i+0x1] ^= xtime(block[i+0x1]^block[i+0x2] ^ Tmp2) ^ Tmp3;
 Block[i+0x2] ^= xtime(block[i+0x2]^block[i+0x3] ^ Tmp1) ^ Tmp3;
 Block[i+0x3]  = block[i+0x0] ^ block[i+0x1] ^ block[i+0x2] ^ Tmp0
}
```

On-The-Fly Key Schedule

The original key schedule functions use several RAM positions, in order to save all round keys used in the encryption/decryption process.

To reduce the RAM consumption, the implementation of the round keys was done on-the-fly. To do this, three different functions were added:

1. `enc_key_schedule`(key): This function takes the actual key and generates the next round key that is placed in the same RAM positions.
2. `dec_key_schedule`(key): This function takes the actual key and generates the previous round key that is placed in the same RAM positions.
3. `init_decryption_key`(key): This function takes the initial key used to encrypt the code and executes the `enc_key_schedule`(key) function `NUMROUNDS` times. The result is the last round key used in the encryption.

The reason behind this is that in the encryption process, the rounds use the scheduled keys (W) in the following sequence:

W[0] → W[1] → W[2] → W[3] → W[4] → W[5] → W[6] → W[7] → W[8] → W[9] → W[10]

While the decryption process uses the exact same scheduled keys in the reverse order:

W[10] → W[9] → W[8] → W[7] → W[6] → W[5] → W[4] → W[3] → W[2] → W[1] → W[0]

Given the generic round key:

K0	K4	K8	K12
K1	K5	K9	K13
K2	K6	K10	K14
K3	K7	K11	K15

`enc_key` Schedule:

The `enc_key` schedule may be understood in four steps:

1. Column 0 is transformed as follows:

K0 ^= s_box[K13]	K4	K8	K12
K1 ^= s_box[K14]	K5	K9	K13
K2 ^= s_box[K15]	K6	K10	K14
K3 ^= s_box[K12]	K7	K11	K15

After that:

```
 K0= K0 ^ Rcon
Rcon = xtime(Rcon)
```

The startup value of Rcon =0x01

2. Column 1 is XORed with column 0 as follows:

K4 ^= K0
K5 ^= K1
K6 ^= K2
K7 ^= K3

3. Column 2 is XORed with column 1 as follows:

K8 ^= K4
K9 ^= K5
K10 ^= K6
K11 ^= K7

4. Column 3 is XORed with column 2 as follows:

K12 ^= K8
K13 ^= K9
K14 ^= K10
K15 ^= K11

`dec_key` Schedule

The `dec_key` schedule may be understood in the exact same steps executed in reverse order:

1. Column 3 is XORed with column 2 as follows:

K12 ^= K8
K13 ^= K9
K14 ^= K10
K15 ^= K11

2. Column 2 is XORed with column 1 as follows:

K8 ^= K4
K9 ^= K5
K10 ^= K6
K11 ^= K7

3. Column 1 is XORed with column 0 as follows:

K4 ^= K0
K5 ^= K1
K6 ^= K2
K7 ^= K3

4. Column 0 is transformed as follows:

K0 ^= s_box[K13]	K4	K8	K12
K5 ^= s_box[K14]	K5	K9	K13
K6 ^= s_box[K15]	K6	K10	K14
K7 ^= s_box[K12]	K7	K11	K15

And after that:

```
K0=K0 ^ Rcon
if(Rcon &0x01)
   Rcon = 0x80
else
   Rcon >>1
```

This procedure is the exact inverse operation executed over K0 in the `enc_key` process (i.e., the xtime function applied to `Rcon`).

Source Code Example 1 (Encryption and Decryption)

The `aes_rijn.asm` source code first encrypts 16 bytes of data, then decrypts the 16 bytes of data that were just encrypted. Listed below is some important information you should know before using this source code:

1. Source code is written in Microchip Assembly language (MPASM™ Assembler).
2. Source code in `aes_rijn.asm` has been tested using MPLAB® 5.20.00:
    - Simulator testing has been done using a PIC16C622A device.
    - MPLAB ICD testing has been done using a PIC16F870 device. When using this device, the `tables.inc` memory locations need to be adjusted to accommodate the MPLAB ICD memory needs.
3. The `tables.inc` file is listed in Appendix F. This is where the S-TABLE & Si-TABLE can be found.
4. ROM Memory needed for Example #1 is:
    - (1416 x 14 bits) instructions
5. RAM Memory needed for Example #1 is:
    - encryption: 38 bytes total
      16 for the block cipher
      16 for key
      6 for loop control and partial result calculation
    - decryption: 41 bytes total
      16 for the block cipher
      16 for key
      9 for loop control and partial result calculation

Note: 41 bytes is the total needed; several registers are shared.

6. Execution Speed (in instruction cycles, calculated as the external clock/4):
    - encryption time: up to 5273 cycles
    - decryption schedule: up to 928 cycles
    - decryption time: up to 6413 cycles

Note: The number of cycles shown here were the largest found during simulations. Depending on your code implementation, these times may vary.

7. The 16-byte block vector is located in RAM locations 0x20 - 0x2F:
    - The `set_test_block` subroutine of the `aes_rijn.asm` code loads the 16 bytes of hard coded plain text data into the block vector. In order to change the initial block vector data, the `set_test_block` code needs to be changed.
    - The block vector is where the plain text data resides before the encryption process. The block vector is also where the encrypted text resides after the encryption process and before the decryption process, and finally, where the plain text data resides after the decryption process. It is important to be aware that the block vector locations are overwritten during code execution.
8. The 16-byte key vector is located in RAM locations 0x30 - 0x3F:
    - The `set_test_key` subroutine of the `aes_rijn.asm` code loads the 16 bytes of hard coded key data into the key vector. In order to change the initial key vector data, the `set_test_ key` code needs to be changed.
9. Test data can be found in the following files:
    - `ecbvt.txt` contains the encrypted results (CT) for changing plain text block vector data (PT) when the key vector data (KEY) is kept constant at KEY=`0000000000000000`.
    - `ecbvk.txt` contains the encrypted results (CT) for changing key vector data (KEY) when the plain text block vector data (PT) is kept constant at PT=`0000000000000000.`
10. The files you will need for this example are as follows:
    - `aes_rijn.asm`
    - `tables.inc`
    - `ecbvt.txt` (KEY constant)
    - `ecbvk.txt` (PT constant)

These files can be found with this Application Note on the Microchip web site:

www.microchip.com

Warning: United States federal regulations allow the Advanced Encryption Standard (AES) software code to be downloaded from the Microchip web site. The United States federal regulations restrict transfer of this Advanced Encryption Standard (AES) software by other means such as e-mail.

Source Code Example 2 (Encryption)

The `aes_encr.asm` source code encrypts 16 bytes of data. Listed below is some important information you should know before using this source code:

1. Source Code is written in Microchip Assembly language (MPASM Assembler).
2. Source Code in `aes_encr.asm` has been tested using MPLAB 5.20.00:
    - Simulator testing has been done using a PIC16C622A device.
    - MPLAB ICD testing has been done using a PIC16F870 device. When using this device, the `tables.inc` memory locations need to be adjusted to accommodate the MPLAB ICD memory needs.
3. The `s_table.inc` file is where the S-TABLE can be found.
4. ROM Memory needed for Example #2 is:
    - encryption: (728 x 14 bit) instructions
5. RAM Memory needed for Example #2 is:
    - encryption: 38 bytes total
      16 for the block cipher
      16 for key
      6 for loop control and partial result calculation
6. Execution Speed (in instruction cycles, calculated as the external clock/4):
    - encryption time: up to 5273 cycles

Note:	The number of cycles shown here were the largest found during simulations. Depending on your code implementation, these times may vary.

7. The 16-byte block vector is located in RAM locations 0x20 - 0x2F:
    - The `set_test_block` subroutine of the `aes_encr.asm` code loads the 16 bytes of hard coded plain text data into the block vector. In order to change the initial block vector data, the `set_test_block` code needs to be changed.
    - The block vector is where the plain text data resides before the encryption process. The block vector is also where the encrypted text resides after the encryption process. It is important to be aware that the block vector locations are overwritten during code execution.
8. The 16-byte key vector is located in RAM locations 0x30 - 0x3F:
    - The `set_test_key` subroutine of the `aes_encr.asm` code loads the 16 bytes of hard coded key data into the key vector. In order to change the initial key vector data, the `set_test_ key` code needs to be changed.
9. Test data can be found in the following files:
    - `ecbvt.txt` contains the encrypted results (CT) for changing plain text block vector data (PT) when the key vector data (KEY) is kept constant at KEY=`0000000000000000.`
    - `ecbvk.txt` contains the encrypted results (CT) for changing key vector data (KEY) when the plain text block vector data (PT) is kept constant at PT=`0000000000000000.`
10. The files you will need to run and test this Example are as follows:
    - `aes_encr.asm`
    - `s_table.inc`
    - `ecbvt.txt` (KEY constant)
    - `ecbvk.txt` (PT constant)

These files can be found with this Application Note on the Microchip web site:

www.microchip.com

Warning:	United States federal regulations allow the Advanced Encryption Standard (AES) software code to be downloaded from the Microchip web site. The United States federal regulations restrict transfer of this Advanced Encryption Standard (AES) software by other means such as e-mail.

Source Code Example 3 (Decryption)

The `aes_decr.asm` source code decrypts the 16 bytes of data. Listed below is some important information you should know before using this source code:

1. Source Code is written in Microchip Assembly language (MPASM Assembler).
2. Source Code `aes_decr.asm` has been tested using MPLAB 5.20.00:
   - Simulator testing has been done using a PIC16C622A device. When using this device, the `tables.inc` memory locations need to be adjusted to accommodate the MPLAB ICD memory needs.
   - ICD testing has been done using a PIC16F870 device.
3. The `tables.inc` file is listed in Appendix F. This is where the S-TABLE and Si-TABLE can be found.
4. ROM Memory needed for Example #3 is:
   - 1143 x 14 bit instructions
5. RAM Memory needed for Example #3 is:
   - decryption: 41 bytes total
     16 for the block cipher
     16 for key
     9 for loop control and partial result calculation
6. Execution Speed (in instruction cycles, calculated as the external clock/4):
   - decryption schedule: up to 928 cycles
   - decryption time: up to 6413 cycles

Note: The number of cycles shown here were the largest found during simulations. Depending on your code implementation, these times may vary.

7. The 16-byte block vector is located in RAM locations 0x20 - 0x2F:
   - The `set_test_block` subroutine of the `aes_decr.asm` code loads the 16 bytes of hard coded plain text data into the block vector. In order to change the initial block vector data, the `set_test_block` code needs to be changed.
   - The block vector is where the encrypted text data resides before the decryption process. The block vector is also where the plain text data resides after the decryption process. It is important to be aware that the block vector locations are overwritten during code execution.
8. The 16-byte key vector is located in RAM locations 0x30 - 0x3F:
   - The `set_test_key` subroutine of the `aes_decr.asm` code loads the 16 bytes of hard coded key data into the key vector.
9. Appendix H and Appendix I hold test data:
   - `ecbvt.txt` contains the encrypted results (CT) for changing plain text block vector data (PT) when the key vector data (KEY) is kept constant at KEY=`0000000000000000.`
   - `ecb_vk.txt` contains the encrypted results (CT) for changing key vector data (KEY) when the plain text block vector data (PT) is kept constant at PT=`0000000000000000.`
10. The files you will need to run and test this example are as follows:
    - `aes_decr.asm`
    - `tables.inc`
    - `ecbvt.txt` (KEY constant)
    - `ecbvk.txt` (PT constant)

These files can be found with this Application Note on the Microchip web site:

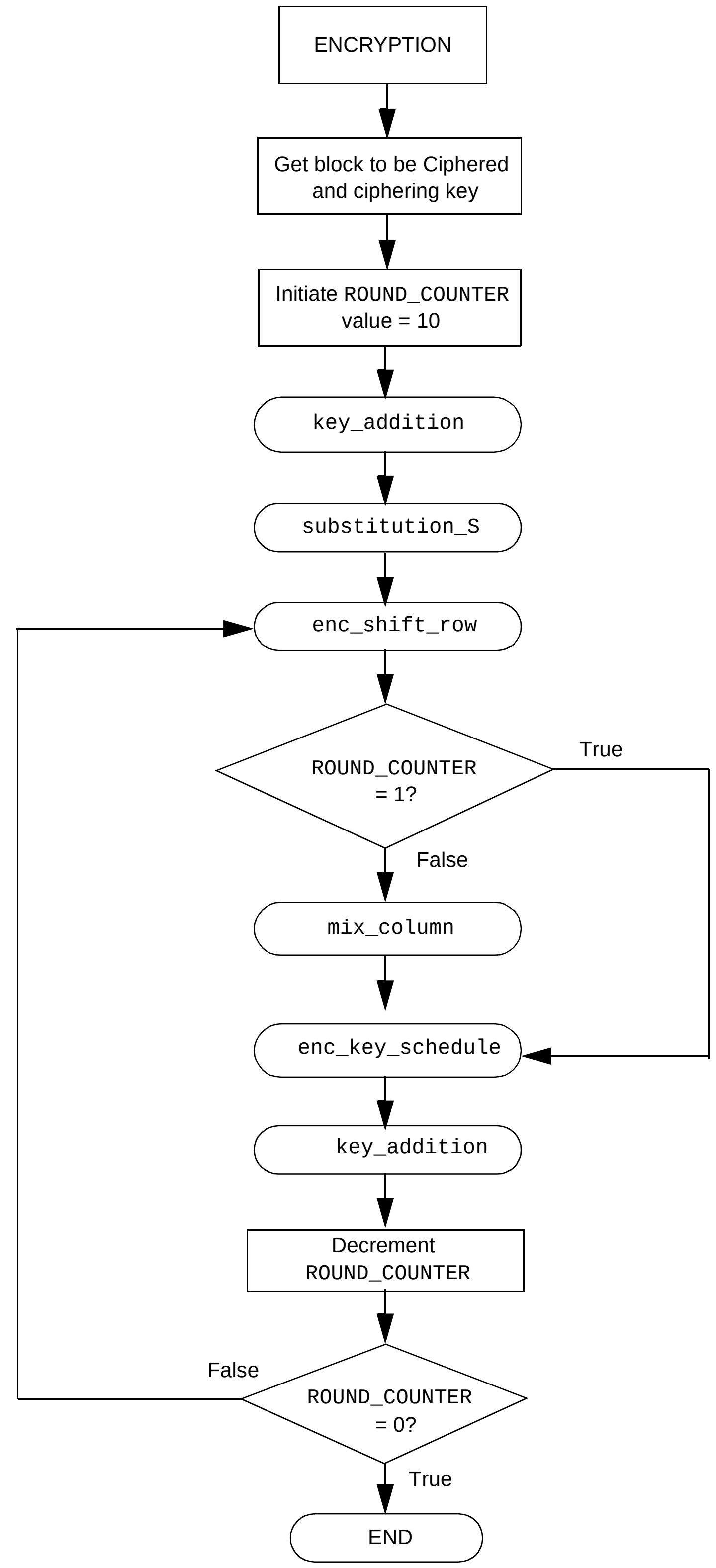
www.microchip.com

Warning: United States federal regulations allow the Advanced Encryption Standard (AES) software code to be downloaded from the Microchip web site. The United States federal regulations restrict transfer of this Advanced Encryption Standard (AES) software by other means such as e-mail.

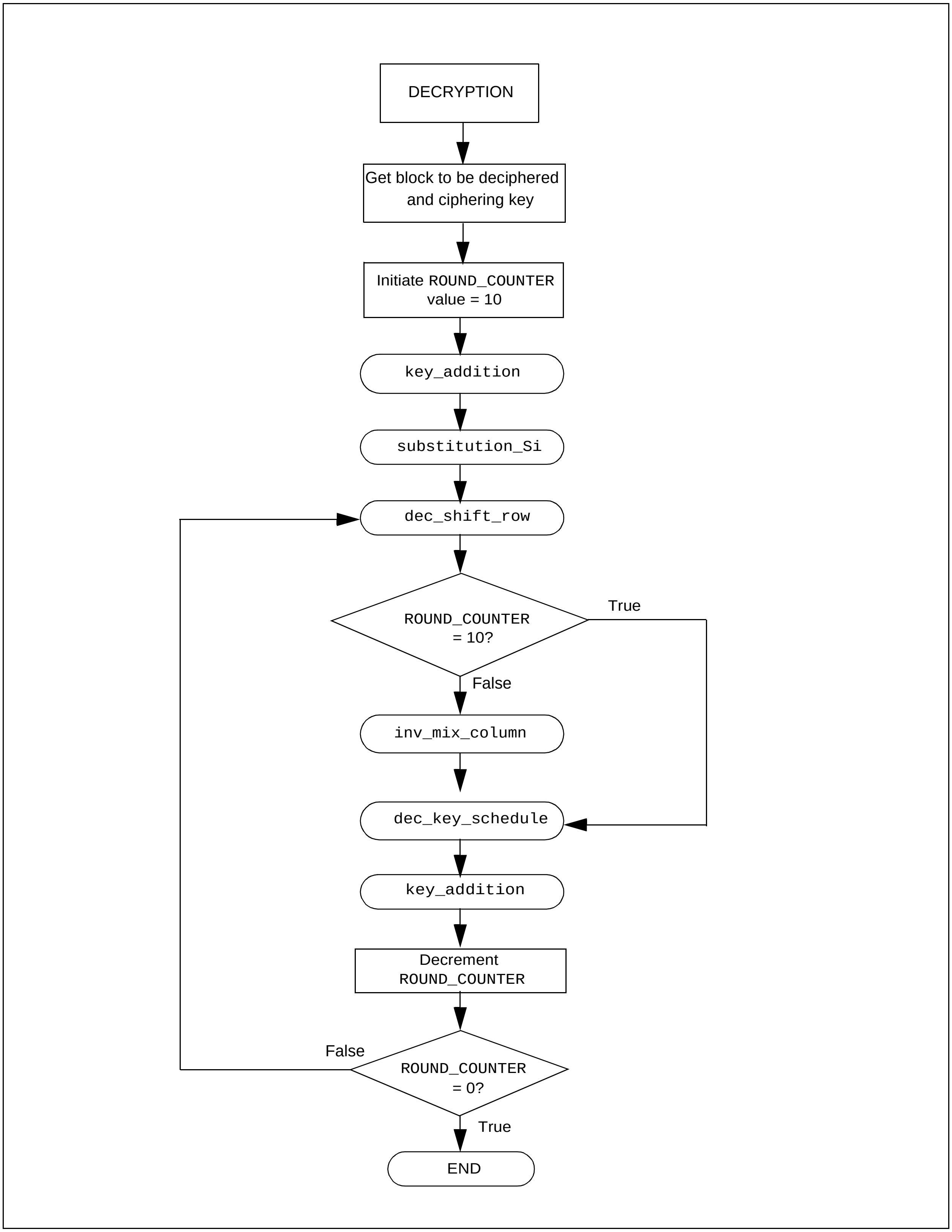
References

- Internet: Several good sources of information about Cryptography, in general, and AES/Rijndael were used in this Application Note:
  - NIST: http://csrc.nist.gov/encryption/aes/aes_home.htm
  - Rijndael home page:http://www.esat.kuleuven.ac.be/~rijmen/rijndael
  - Ritter: http://www.io.com/~ritter
  - Savard: http://home.ecn.ab.ca/~jsavard/crypto
- Book: "*Applied Cryptography*", Bruce Schneier, John Wiley & Sons, Inc., ISBN 0-471-11709-9
- Implementations: Excellent implementations were consulted and studied during the development process and to them our acknowledgement and gratitude:
  - Paulo Barreto, Dr. Vincent Rijmen and Antoon Bosselaers for their references and fast C versions of Rijndael.
  - Dr. Brian R. Gladman for his C++ implementation.
  - Mike Scott by his C version.
  - Rafael R. Sevilla for his 80x86 assembly version.
  - Robert G. Durnal for his 80x86 assembly version.

APPENDIX A: AES ENCRYPTION FLOW CHART



APPENDIX B: AES DECRYPTION FLOW CHART



NOTES:

Note the following details of the code protection feature on PICmicro® MCUs.

- The PICmicro family meets the specifications contained in the Microchip Data Sheet.
- Microchip believes that its family of PICmicro microcontrollers is one of the most secure products of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the PICmicro microcontroller in a manner outside the operating specifications contained in the data sheet. The person doing so may be engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable”.
- Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our product.

If you have any further questions about this matter, please contact the local sales office nearest to you.

Trademarks

The Microchip name and logo, the Microchip logo, FilterLab, KEELOQ, microID, MPLAB, PIC, PICmicro, PICMASTER, PICSTART, PRO MATE, SEEVAL and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

dsPIC, ECONOMONITOR, FanSense, FlexROM, fuzzyLAB, In-Circuit Serial Programming, ICSP, ICEPIC, microPort, Migratable Memory, MPASM, MPLIB, MPLINK, MPSIM, MXDEV, MXLAB, PICC, PICDEM, PICDEM.net, rfPIC, Select Mode and Total Endurance are trademarks of Microchip Technology Incorporated in the U.S.A.

Serialized Quick Turn Programming (SQTP) is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.

♻ Printed on recycled paper.

DNV Certification, Inc.
USA

ANSI • RAB
QMS
ACCREDITED

DNV

DNV MSC
The Netherlands
Accredited by the RvA

ISO 9001 / QS-9000
REGISTERED FIRM

Microchip received QS-9000 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona in July 1999 and Mountain View, California in March 2002. The Company’s quality system processes and procedures are QS-9000 compliant for its PICmicro® 8-bit MCUs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, non-volatile memory and analog products. In addition, Microchip’s quality system for the design and manufacture of development systems is ISO 9001 certified.

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200 Fax: 480-792-7277
Technical Support: 480-792-7627
Web Address: http://www.microchip.com

Rocky Mountain
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7966 Fax: 480-792-4338

Atlanta
500 Sugar Mill Road, Suite 200B
Atlanta, GA 30350
Tel: 770-640-0034 Fax: 770-640-0307

Boston
2 Lan Drive, Suite 120
Westford, MA 01886
Tel: 978-692-3848 Fax: 978-692-3821

Chicago
333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071 Fax: 630-285-0075

Dallas
4570 Westgrove Drive, Suite 160
Addison, TX 75001
Tel: 972-818-7423 Fax: 972-818-2924

Detroit
Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI 48334
Tel: 248-538-2250 Fax: 248-538-2260

Kokomo
2767 S. Albright Road
Kokomo, Indiana 46902
Tel: 765-864-8360 Fax: 765-864-8387

Los Angeles
18201 Von Karman, Suite 1090
Irvine, CA 92612
Tel: 949-263-1888 Fax: 949-263-1338

New York
150 Motor Parkway, Suite 202
Hauppauge, NY 11788
Tel: 631-273-5305 Fax: 631-273-5335

San Jose
Microchip Technology Inc.
2107 North First Street, Suite 590
San Jose, CA 95131
Tel: 408-436-7950 Fax: 408-436-7955

Toronto
6285 Northam Drive, Suite 108
Mississauga, Ontario L4V 1X5, Canada
Tel: 905-673-0699 Fax: 905-673-6509

ASIA/PACIFIC

Australia
Microchip Technology Australia Pty Ltd
Suite 22, 41 Rawson Street
Epping 2121, NSW
Australia
Tel: 61-2-9868-6733 Fax: 61-2-9868-6755

China - Beijing
Microchip Technology Consulting (Shanghai)
Co., Ltd., Beijing Liaison Office
Unit 915
Bei Hai Wan Tai Bldg.
No. 6 Chaoyangmen Beidajie
Beijing, 100027, No. China
Tel: 86-10-85282100 Fax: 86-10-85282104

China - Chengdu
Microchip Technology Consulting (Shanghai)
Co., Ltd., Chengdu Liaison Office
Rm. 2401, 24th Floor,
Ming Xing Financial Tower
No. 88 TIDU Street
Chengdu 610016, China
Tel: 86-28-86766200 Fax: 86-28-86766599

China - Fuzhou
Microchip Technology Consulting (Shanghai)
Co., Ltd., Fuzhou Liaison Office
Unit 28F, World Trade Plaza
No. 71 Wusi Road
Fuzhou 350001, China
Tel: 86-591-7503506 Fax: 86-591-7503521

China - Shanghai
Microchip Technology Consulting (Shanghai)
Co., Ltd.
Room 701, Bldg. B
Far East International Plaza
No. 317 Xian Xia Road
Shanghai, 200051
Tel: 86-21-6275-5700 Fax: 86-21-6275-5060

China - Shenzhen
Microchip Technology Consulting (Shanghai)
Co., Ltd., Shenzhen Liaison Office
Rm. 1315, 13/F, Shenzhen Kerry Centre,
Renminnan Lu
Shenzhen 518001, China
Tel: 86-755-2350361 Fax: 86-755-2366086

China - Hong Kong SAR
Microchip Technology Hongkong Ltd.
Unit 901-6, Tower 2, Metroplaza
223 Hing Fong Road
Kwai Fong, N.T., Hong Kong
Tel: 852-2401-1200 Fax: 852-2401-3431

India
Microchip Technology Inc.
India Liaison Office
Divyasree Chambers
1 Floor, Wing A (A3/A4)
No. 11, O'Shaugnessey Road
Bangalore, 560 025, India
Tel: 91-80-2290061 Fax: 91-80-2290062

Japan
Microchip Technology Japan K.K.
Benex S-1 6F
3-18-20, Shinyokohama
Kohoku-Ku, Yokohama-shi
Kanagawa, 222-0033, Japan
Tel: 81-45-471- 6166 Fax: 81-45-471-6122

Korea
Microchip Technology Korea
168-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea 135-882
Tel: 82-2-554-7200 Fax: 82-2-558-5934

Singapore
Microchip Technology Singapore Pte Ltd.
200 Middle Road
#07-02 Prime Centre
Singapore, 188980
Tel: 65-6334-8870 Fax: 65-6334-8850

Taiwan
Microchip Technology (Barbados) Inc.,
Taiwan Branch
11F-3, No. 207
Tung Hua North Road
Taipei, 105, Taiwan
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

EUROPE

Denmark
Microchip Technology Nordic ApS
Regus Business Centre
Lautrup hoj 1-3
Ballerup DK-2750 Denmark
Tel: 45 4420 9895 Fax: 45 4420 9910

France
Microchip Technology SARL
Parc d'Activite du Moulin de Massy
43 Rue du Saule Trapu
Batiment A - ler Etage
91300 Massy, France
Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79

Germany
Microchip Technology GmbH
Gustav-Heinemann Ring 125
D-81739 Munich, Germany
Tel: 49-89-627-144 0 Fax: 49-89-627-144-44

Italy
Microchip Technology SRL
Centro Direzionale Colleoni
Palazzo Taurus 1 V. Le Colleoni 1
20041 Agrate Brianza
Milan, Italy
Tel: 39-039-65791-1 Fax: 39-039-6899883

United Kingdom
Microchip Ltd.
505 Eskdale Road
Winnersh Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44 118 921 5869 Fax: 44-118 921-5820

Austria
Microchip Technology Austria GmbH
Durisolstrasse 2
A-4600 Wels
Austria
Tel: 43-7242-2244-399
Fax: 43-7242-2244-393

05/16/02