Algorithm - Fast and Compact Unsigned Binary to BCD Conversion

AN2338

Author: Eugene Miyushkovich, Ryshtun Andrij
Associated Project: Yes
Associated Part Family: CY8C24x23A, CY8C24x94, CY8C27x43
Software Version: PSoC® Designer™ 5.1
Associated Application Notes: AN2112, AN2113

Application Note Abstract

This Application Note describes a fast and compact method to convert 8- and 16-bit binary numbers to BCD (binary-coded decimal). This algorithm differs from similar methods in execution speed and code size. Also discussed are functions to convert 8- and 16-bit binary numbers to string (analogous to the standard function 'itoa').

Introduction

Binary to BCD conversion is used when binary numbers must be displayed on text-based devices. Such devices include LCD displays, 7-digit LED indicators, printers, and in other applications where information must be displayed conveniently in decimal format for a human to read.

Division and subtraction are the two most common methods used for binary to BCD conversion. The first method, division, includes a series of divisions of the target binary number by 10 (see Application Note AN2113, "Math Programs"). The remainder holds the single decimal for each step. The number of division operations is equal to the number of decimal digits in the number. This method has several drawbacks. First, the use of the division is costly and second, the execution time rises sharply with increasing magnitude of the binary number to be converted. Therefore, the first method is only optimal for 8-bit number conversion.

The essence of the second method, subtraction, lies in serially repeated subtraction of the 10^n number from the target binary number, where 'n' is the number of the decimal digit (0, 1, 2, 3, 4). After each subtraction a check for zero takes place. The number of subtraction operations is equal to the number of digits in the resulting decimal number.

Very long execution time, even for an 8-bit number, is the main drawback of the subtraction method. The maximum number of subtraction operations it takes to convert a 16-bit number is 41. The worst-case number for this operation is 59999.

An interesting method of binary to BCD conversion is described in AN2112, "Binary To BCD Conversion." It uses minimal code, which increases slowly as the magnitude of the number increases from 8 to 24 bits. But it has the same drawback as the methods described above, long execution time.

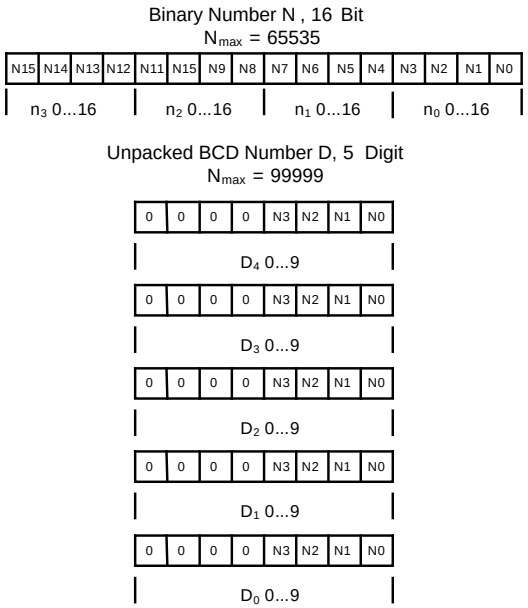
A very fast and compact 8- and 16-bit binary to BCD conversion method is proposed in this Application Note. It is based on the PSoC device Multiply Accumulate (MAC) and special mathematical methods to translate binary numbers to decimal base. This method is especially useful for 8- and 16-bit binary numbers but loses its advantage for numbers of greater magnitude. Fortunately, 8- and 16-bit binary numbers are the most common choice for use in embedded applications.

The Algorithm

A following algorithm was used to develop our fast and compact binary to BCD conversion method [1]. To understand the algorithm, we first consider the data structure. Let's consider a 16-bit number. The input is straightforward binary. The output is in unpacked BCD. See Figure 1.

For a 16-bit binary number, the maximum value is 65535. So the equivalent BCD number has 5 bytes: 6; 5; 5; 3; 5. Each byte consists of one decimal digit.

Figure 1. 16-Bit Binary and BCD Number Data Structure



A 16-bit binary number can be broken into 4 fields of 4 bits each. Let's call these fields n_3, n_2, n_1, n_0. Write the value of the number, using these coefficients:

$$n = 4096n_3 + 256n_2 + 16n_1 + n_0 \qquad \text{Equation 1}$$

Let's consider the decimal number $d_4d_3d_2d_1d_0$, where d_4, d_3, d_2, d_1 and d_0 are decimal digits. In the base 10^n, the value of n can be expressed as:

$$n = 10000d_4 + 1000d_3 + 100d_2 + 10d_1 + d_0 \qquad \text{Equation 2}$$

By combining equations (1) and (2), we can rewrite the original equation as:

$$n = n_3(4\times1000 + 0\times100 + 9\times10 + 6\times1) + \\ +n_2(2\times100 + 5\times10 + 6\times1) + \\ +n_1(1\times10 + 6\times1) + n_0(1\times1) \qquad \text{Equation 3}$$

If we distribute the n_i over the equations for each factor, we get the following:

$$n = 1000(4n_3) + 100(0n_3 + 2n_2) + \\ +10(9n_3 + 5n_2 + 1n_1) + \\ +1(6n_3 + 6n_2 + 6n_1 + 1n_0) \qquad \text{Equation 4}$$

We can use this to arrive at first estimates a_i for d_3 through d_0:

$$a_3 = 4n_3$$
$$a_2 = 0n_3 + 2n_2$$
$$a_1 = 9n_3 + 5n_2 + 1n_1 \qquad \text{Equation 5}$$
$$a_0 = 6n_3 + 6n_2 + 6n_1 + 1n_0$$

$$d_i = (a_i/10) \bmod 10 \qquad \text{Equation 6}$$

The values of a_i are not proper decimal digits because they are not properly bounded in the range $0 \le a_i \le 9$. Instead, given that each of n_i is bounded in the range $0 \le n_i \le 15$, the a_i are bounded as follows from Equation (5):

$$0 \le a_3 \le 60 \qquad (4*15)$$
$$0 \le a_2 \le 30 \qquad (2*15)$$
$$0 \le a_1 \le 225 \qquad (9*15 + 5*15 + 1*15) \qquad \text{Equation 7}$$
$$0 \le a_0 \le 285 \qquad (6*15 + 6*15 + 6*15 + 1*15)$$

This is actually quite promising because a_3 through a_1 is less than 256 and may therefore, be computed using the 8 bit PSoC arithmetic logic unit (ALU), and even a_0 may be computed in 8 bits if we can use the carry-out bit of the PSoC ALU to hold the high bit. Furthermore, if our interest is two's-complement arithmetic where the high bit is the sign bit, the first step in the output process is to print a minus sign and then negate, so the constraint on n_3 becomes $0 \le n_3 \le 7$ and we can conclude that:

$$0 \le a_0 \le 243 \qquad \text{Equation 8}$$

As we can see, operations on all considered numbers are executed on an 8-bit basis.

To illustrate the described algorithm, sample ‘C’ code is shown below:

Code 1.

```
unsigned short int n //n - bin number
unsigned char d4, d3, d2, d1, q;
//d4…d0 - decimal numbers
unsigned short int a0;
//Find n0…n3 numbers
d0 = n & 0xF;
d1 = (n>>4) & 0xF;
d2 = (n>>8) & 0xF;
d3 = (n>>12) & 0xF;
//Calculate d0…d4 numbers
d0 = 6*(d3 + d2 + d1) + d0;
q = d0 / 10;
d0 = d0 % 10;

d1 = q + 9*d3 + 5*d2 + d1;
q = d1 / 10;
d1 = d1 % 10;

d2 = q + 2*d2;
q = d2 / 10;
d2 = d2 % 10;

d3 = q + 4*d3;
q = d3 / 10;
d3 = d3 % 10;

d4 = q;
/*…Print d0…d4 numbers…*/
```

At first, the binary number ‘n’ that is to be converted to BCD is broken into 4 fields of 4 bits each. These fields are placed in variables $d_{0...}d_{3.}$ Next, values for $a_{0...}a_4$ are calculated according to Equation (5). The remainders from dividing the coefficients, $a_{0...}a_4$ by 10, are decimal numbers, which is what we need.

Routine for 8-Bit Binary to BCD Conversion

The standard division algorithm is used for 8-bit binary to BCD conversion. It works as follows:

1. Divide the binary input by 10.
2. Save the intermediate result.
3. Multiply the result by 10.
4. Subtract the multiplication result from binary input number.
5. Store the result in [units].
6. Divide the intermediate result by 10.
7. Store the result in [hundreds].
8. Multiply the hundreds by 10.
9. Subtract the multiplication result from intermediate result.
10. Store the result in [tens].

See the following example. The binary input number is 123. The BCD output is stored in D[0]…. D[2]:

1. 123/10=12
2. 12 →[D1]
3. 12*10=120
4. 123-120=3
5. 3→[D0]
6. 12/10=1
7. 1→[D2]
8. 1*10=10
9. 12-10=2
10. 2→[D1]

The assembly source code that converts an 8-bit binary number to BCD is shown below. In this implementation, the division instruction is represented by multiplication and shift instructions.

Code 2.

```
Byte2BCD:
; Divide binary input by 10
   mov   [D+0], A;5/2      store N in D[0]
   asr   A;   4/1    A = N div 2
   and   A, 7fh;     4/2    clear MSB of A
   mov   REG[MUL_X], 67h;  8/3
   mov   REG[MUL_Y], A;    5/2
   mov   A, REG[MUL_DH];   6/2
   asr   A;   4/1    A = N div 10
; Save intermediate result
   mov   [D+1], A;5/2      D[1] = N div 10
; The result is multiplied by 10
   mov   REG[MUL_Y],A;5/2  prepare to next
mul
   asl   A;   4/1    A =  2*(N div 10)
   asl   A;   4/1    A =  4*(N div 10)
   add   A, [D+1];6/2      A =  5*(N div
10)
   asl   A;   4/1    A = 10*(N div 10)
; Subtract multiplication result from
binary input number and store result in
[units]
   sub   [D+0], A;7/2      D[0]= N mod 10
;_____________________________Result 71/24
; Divide intermediate result by 10
   mov   REG[MUL_X], 1Ah;  8/3
   mov   A, REG[MUL_DH];   6/2
; Store the result in [hundreds]
   mov   [D+2], A; 5/2     D[2] = D[1] div
10
; The hundreds is multiplied by 10
   asl   A;   4/1    A =  2*D[2]
   asl   A;   4/1    A =  4*D[2]
   add   A, [D+2];6/2      A =  5*D[2]
   asl   A;   4/1    A = 10 * D[2]
; Subtract multiplication results from
intermediate result and store result in
[tens]
   sub   [D+1], A; 7/2     D[1] = (N div
10) mod 10
   ret;              8/1
;_____________________________Result 52/15
; Total Byte2BCD: 123 cycles / 38 bytes
```

This code is optimized for the PSoC ALU. It therefore is very compact and executes quickly.

A binary number is placed into ‘A’. After the conversion procedure, the BCD result is placed into global variables, $D_0 \ldots D_2$. The lowest digit of the decimal number and part of the next digit are defined in the first part of the code. The other digits are found in the second part of the code.

The PSoC MAC is used for both programs. This helps reduce code size and decrease execution time. When it is necessary to port to other PSoC devices that do not support MAC, the multiplication that the MAC executes can be replaced by several addition and shift operations. But even in this case, the method still yields compact code and executes quickly.

Routine for 16-Bit Binary to BCD Conversion

The algorithm for 16-bit binary to BCD conversion is shown ahead:

Code 3.

```
;Binary value - passed in X:A [MSB:LSB]
;returns Global variables [D+4], [D+3],
[D+2], [D+1] & [D+0]


Word2BCD:


   mov [D+0], A;    5/2    D[0] = 16n1 + n0
   asr   A;  4/1
   and   A, 78h;    4/2    A = 8n1
   sub   [D+0],     A; 7/2 D[0] = 8n1 + n0
   asr   A;  4/1    A = 4n1
   asr   A;  4/1    A = 2n1
   sub   [D+0],     A;7/2  D[0] = 6n1 + n0
   asr   A;  4/1    A = n1
   mov   [D+1], A;5/2      D[1] = n1
;_____________________________Result 44/14
   mov   A, X;      4/1    A = 16n3 + n2
   asr   A;  4/1
   and   A, 78h;    4/2    A = 8n3
   add   [D+1], A;7/2      D[1] = 8n3 + n1
   asr   A;  4/2    A = 4n3
   mov   [D+3], A;5/2      D[3] = 4n3
   add   [D+0], A;7/2      D[0] = 4n3 + 6n1
+ n0
   asr   A;  4/2    A = 2n3
   add   [D+0], A;7/2      D[0] = 6n3 + 6n1
+ n0
   asr   A;  4/2    A = n3
   add   [D+1], A;7/2      D[1] = 9n3 + n1
;_____________________________Result 57/20
   mov   A, X;      4/1    A = 16n3 + n2
   and   A, 0Fh;    4/2    A = n2
   add   [D+1], A;7/2      D[1] = 9n3 + n2
+ n1
   asl   A;  4/1    A = 2n2
   add   [D+0], A;7/2      D[0] = 6n3 + 2n2
+ 6n1+n0
   mov   [D+2], A;5/2      D[2] = 2n2
   asl   A;  4/1    A = 4n2
   add   [D+1], A;7/2      D[1] = 9n3 + 5n2
+ n1
   add   [D+0], A;7/2 C:D[0] = 6n3 + 6n2 +
6n1 + n0
   mov   [D+4], 0;8/3      clear D[4]
   mov   REG[MUL_X],67h;8/3       prepare
to MUL
   mov   X, -4;     4/2    set index / loop
counter
   jnc   loop;      5/2    skip correction
if C==0
;_____________________________Result 74/25
   add   [D+1], 20;9/3     D[1]+20 equ
C:D[0] - 200
   add   [D+0], 56;9/3     D[0] + 56
;_____________________________Result 18/06
loop:
   mov   A, [X+D+4];6/2    copy D[i] to A
   rrc   A;  4/1    A = D[i] div 2
   mov   REG[MUL_Y], A;    5/2
   mov   A, REG[MUL_DH];   6/2
   asr   A;  4/1    A = D[i] div 10
   add   [X+D+5], A;8/2
       D[i+1]=D[i+1]+(D[i]div10)
   asl   A;  4/1    A = 2(D[i] div 10)
   sub   [X+D+4], A; 8/2   D[i]= D[i] - A
   asl   A;  4/1    A = 4(D[i] div 10)
   asl   A;  4/1    A = 8(D[i] div 10)
   sub   [X+D+4], A;8/2    D[i]= D[i]mod10-
completed
   inc   X;  4/1    inc index/loop counter
   jnz   loop;      5/2    repeat 4 times
;_____________________Result 70*4 = 280/20
   ret;             8/1
; Total Word2BCD: 458(481) cycles / 82
bytes
```

The assembly language source code carries out the 16-bit binary to BCD conversion in the same manner as the ‘C’ source code. This source code is optimized for PSoC assembly language. After each M8C instruction, two numbers follow in the comment field. The first number shows the quantity of CPU machine cycles. The other number shows the size of the given instruction in bytes. This is useful for calculating and revising execution time and code size.

First, the coefficients $a_0 \ldots a_3$ are calculated. This is done in three virtually identical procedures. To calculate these coefficients, the addition and shift operations are used. Overrun correction is done at the end of these operations.

After the $a_0…a_3$ coefficients are calculated, the loop procedure is executed four times. The loop procedure executes the same function as the following ‘C’ code:

```
q = a2 / 10;
a2 = a2 % 10;
```

This function extracts the decimal digits and places them into $D_0…D_4$.

Routine for 8- and 16-Bit Binary to STRING Conversion

Very often binary number to string conversion is needed. For this task, the standard ‘C’ function ‘itoa’ is commonly used. The functions proposed ahead reduce execution time by a factor of 100 and code size by a factor of 3, relative to the iMAGEcraft library function.

An ‘itoa’ (integer to string) function is described ahead. For byte to string conversion, a ‘btoa’ function can be used. These two functions are similar, therefore, we will discuss only one.

Code 4.

```
;Binary value - passed in [SP-6]| [SP-5]
MSB|LSB
;Return zero terminated string was placed
from [SP-3]


ITOA:
_ITOA:
       mov X, SP
       mov A, [X-3]  ;read string pointer
       mov [pt], A         ;
       mov A, [X-5]  ;read input numberLSB
       mov X, [X-6]  ;read input number MSB
;----------------------------------------


       call Word2BCD;converting BIN to BCD
       mov [ct], 4   ;initialize loop
counter
       mov [fl], 0   ;initialize non-zero
flag
;----------------------------------------
.L:    mov X, [ct]
       mov A, [X+D]  ;read one BCD number
       add A, 48     ;convert to ASCII
       mvi [pt], A   ;save ASCII number to
str
       cmp A, 0x30   ;compare number with
zero
       jnz  .Z1            ;jump if non-
zero
.Z:    mov A, [fl]   ;check non-zero flag
       jnz .Z2             ;jump if non-
zero flag
       dec[pt]             ;decrement
string pointer
       jmp .Z2
.Z1:mov [fl], 1            ;set non-zero
flag
.Z2:mov A, [ct]            ;check loop
counter
jz .ex
       dec [ct]
       jmp .L
;----------------------------------------
.ex:                ;exit procedure
       mov A, [fl]
       jnz .Z3
       inc [pt]           ;increment string
pointer ;
;if non-zero flag is clear
.Z3:   mov A, 0            ;add nul to end
of string
       mvi [pt], A
       ret


;----------------------------------------
; Total ITOA: 925 (950)cycles /
58+82=140bytes
;----------------------------------------
```

After the ‘itoa’ function call, the parameters are read from the stack. The parameters are placed onto the stack according to the #Pragma Fastcall16 rules. After that, the ‘Word2BCD’ function is called. This function converts input data to BCD format. The procedure for converting data and writing it into an array in ASCII format is then performed five times. This procedure ignores leading zeros and does not write these zeros to the array. The following example illustrates this.

If we convert the number ‘123’ to ‘string’, we get the result: [0][0][1][2][3]. A string consists of [‘1’’2’’3’0h]. As we can see, the zeros in the high digits are not written into the string.

To mark the end of the string, a binary 0 should be added to the last ASCII character as string terminator.

‘C’ Prototype:

```
extern void ITOA(unsigned char *, unsigned
int );
extern void BTOA(unsigned char *, unsigned
char);
```

When we need to call this function from the assembler, the #Pragma Fastcall16 rules should be used.

‘ITOA’ Function Call:

```
     PUSH  X;
     MOV    A,[wTest]        ;LSB  of  input
value
     PUSH  A
     MOV    A,[wTest+1]     ;MSB  of  input
value
     PUSH  A
     MOV   A,0           ;page pointer
     PUSH  A
     MOV    A,28             ;Output string
pointer
     PUSH  A
     LCALL ITOA
     ADD   SP,252
     POP   X
```

‘BTOA’ Function Call:

```
     PUSH  X
     MOV   A,[bTest]     ;input value
     PUSH  A
     MOV   A,0           ;page pointer
     PUSH  A
     MOV    A,28             ;Output string
pointer
     PUSH  A
     LCALL BTOA
     ADD   SP,253
     POP   X
```

These functions were written only for the small memory model. The user can easily modify the associated project for a large memory model, which is supported in the CY8C29xxx device family.

The associated project is attached. This project includes a library file and source code for library testing. The testing consists of comparing the function result with a standard result. The comparison occurs in the range from 0 to ‘N’, where the ‘N’ is the maximum input value for the given function. The comparison results are sent to the serial port connected to the P1[1] pin. The transmission format is 115200 baud, no parity, and a 1 stop bit.

Summary

In this Application Note a fast and compact unsigned binary to BCD conversion method has been considered. On the basis of this method, integer to string and byte to string functions were written and well tested. As we can see from Table 2 and Table 3, they are smaller and faster than other functions. In the following tables a comparison of the described functions is shown. The drawback of the described method is a sharp decrease in effectiveness as number magnitude increases beyond 16 bits.

The data to estimate the complexity of the algorithm at 32-bit digit calculations are shown in Appendix 1. There is currently no implementation of this function.

Table 1. Code Size/Execution Time Details of Proposed Function Set

Function Name	Input	Output, (Global Variables)	Code Size in Bytes	Execution Time in CPU Cycles	
				Minimum	Maximum
Byte2BCD	A	([D+2], [D+1] & [D+0])	38	123	123
Word2BCD	MSB →X, LSB→A	([D+4], [D+3], [D+2], [D+1] & [D+0])	82	458	481
BTOA	unsigned char	unsigned char * (string)	82	438	438
ITOA	unsigned int	unsigned char * (string)	140	925	950

Table 2. Comparison Between Described Conversion Functions and Functions Described in AN2112

Source	Function Name	Code Size in Bytes	Execution Time in CPU Cycles	
			Minimum	Minimum
Proposed	Byte2BCD	38	123	123
AN2112	bin2bcd8	70	2045	2429
Proposed	Word2BCD	82	458	481
AN2112	bin2bcd16	77	5819	6971

Table 3. Comparison Between Described Binary to String Function and Standard ‘C’ ‘itoa’ Function

Source	Function Name	Code Size in Bytes	Execution Time in CPU Cycles
Proposed	ITOA	140	**925**
C compiler Library	Itoa	470	**94216**

References

1. http://www.cs.uiowa.edu/~jones/bcd/decimal.html

Appendix. 32-Bit Binary to BCD Conversion Arithmetic

Numbers in decimal form:

$$n = d_9 10^9 + d_8 10^8 + d_7 10^7 + d_6 10^6 + d_5 10^5 + d_4 10^4 + d_3 10^3 + d_2 10^2 + d_1 10^1 + d_0$$
$$n = 268435456 n_7 + 16777216 n_6 + 1048576 n_5 + 65536 n_4 + 4096 n_3 + 256 n_2 + 16 n_1 + n_0$$

$$\begin{aligned} n = & \\ & n_7(2*10^8 + 6*10^7 + 8*10^6 + 4*10^5 + 3*10^4 + 5*10^3 + 4*10^2 + 5*10^1 + 6) + \\ & n_6(1*10^7 + 6*10^6 + 7*10^5 + 7*10^4 + 7*10^3 + 2*10^2 + 1*10^1 + 6) + \\ & n_5(1*10^6 + 0*10^5 + 4*10^4 + 8*10^3 + 5*10^2 + 7*10^1 + 6) + \\ & n_4(6*10^4 + 5*10^3 + 5*10^2 + 3*10^1 + 6) + \\ & n_3(4*10^3 + 0*10^2 + 9*10^1 + 6) + \\ & n_2(2*10^2 + 5*10^1 + 6) + \\ & n_1(1*10 + 6) + \\ & n_0 \end{aligned}$$

Coefficients a_0…an:

$$a_8 = 2n_7$$
$$a_7 = 6n_7 + 1n_6$$
$$a_6 = 8n_7 + 6n_6 + 1n_5$$
$$a_5 = 4n_7 + 7n_6 + 0n_5$$
$$a_4 = 3n_7 + 7n_6 + 4n_5 + 6n_4$$
$$a_3 = 5n_7 + 7n_6 + 8n_5 + 5n_4 + 4n_3$$
$$a_2 = 4n_7 + 2n_6 + 5n_5 + 5n_4 + 0n_3 + 2n_2$$
$$a_1 = 5n_7 + 1n_6 + 7n_5 + 3n_4 + 9n_3 + 5n_2 + 1n_1$$
$$a_0 = 6n_7 + 6n_6 + 6n_5 + 6n_4 + 6n_3 + 6n_2 + 6n_1 + 1n_0$$

Maximum value of coefficients a_0…an:

$$0 \le a_8 \le 31$$
$$0 \le a_7 \le 106$$
$$0 \le a_6 \le 226$$
$$0 \le a_5 \le 166$$
$$0 \le a_4 \le 301$$
$$0 \le a_3 \le 426$$
$$0 \le a_2 \le 271$$
$$0 \le a_1 \le 466$$
$$0 \le a_0 \le 646$$

Decimal digits $d_{0...}d_9$:

$$d_i = (a_i / 10) \bmod 10$$

About the Authors

Name: Eugene Miyushkovich
Title: Post-Graduate Student
Background: Eugene earned his computer engineering diploma in 2001 from National University "Lvivska Polytechnika" (Lviv, Ukraine). He is furthering his studies at this university. His interests include embedded systems design and new technologies.
Contact: miyushk@svitonline.com

Name: Ryshtun Andrij
Title: Undergraduate Student
Background: Andrij is a 5th-year student pursuing an MS at National University “Lvivs`ka Polytechnika.”
Contact: ryshtun@gmail.com

Document History

Document Title: Algorithm - Fast and Compact Unsigned Binary to BCD Conversion

Document Number: 001-25639

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	1445143	SSFTMP4	09/07/2007	Recataloged Spec
*A	3088665	ANUP	10/17/2010	Updated to PSoC® Designer™ 5.1

In March of 2007, Cypress recataloged all of its Application Notes using a new documentation number and revision code. This new documentation number and revision code (001-xxxxx, beginning with rev. **), located in the footer of the document, will be used in all subsequent revisions.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone: 408-943-2600
Fax: 408-943-4730
http://www.cypress.com/