AN205500

F²MC-8FX Family MB95200H Series CRC Generation and Check

This application note introduces what is the CRC and how to realize the CRC by software.

Contents

1 Introduction

This document introduces what is the CRC and how to realize the CRC by software.

2 What's CRC

2.1 Background

The process of data transmission or storage usually contains the risk of unwanted modification of the data at the most physical level, caused by noisy or damaged transmission or storage media. (This does not include alteration by an intelligent third party like a malicious attacker.) To detect these errors, some error-detecting and even-correcting codes were invented, which calculate a value from the set of data and transmit or store it with the data. Any hash function can be used to perform this kind of error detection to a certain degree, and one of them is the "Cyclic Redundancy Check" (CRC). It's not a cryptographically secure hash and therefore can not reliably detect malicious changes in the transmitted data, but it can provably detect some common accidental errors like single-/two-bit or burst errors and can additionally be implemented very efficiently. There are different instances of the CRC which mainly differ in the polynomial on which they are based on, resulting in different sizes of the computed value. The most popular one is the CRC32, which computes a 32-bit value. But we use CRC16 or CRC8 on sometimes.

While most of the time you want to calculate the CRC of a given set of data, there are some situations where the CRC is given and you want to modify your data so that it computes to this CRC value afterwards. These scenarios include hard-wired checksums of firmware or calculating the CRC of a set of data which includes the CRC itself. An example of the latter case is the creation of a ZIP archive which includes itself as a file. We develop and analyse methods to calculate these modifications within the next sections.

2.2 CRC Theory

2.2.1 CRC Algorithm

Cyclic Redundancy Check is a way of providing error control coding in order to protect data by introducing some redundancy in the data in a controlled fashion. It is a commonly used and very effective way of detecting transmission errors during transmissions in various networks. Common CRC polynomials can detect the following types of errors:

- All single bit error
- All double bit errors
- All odd number of errors, provided the constraint length is sufficient
- Any burst error for which the burst length is less than the polynomial length
- Most large burst errors

The CRC encoding procedure can be described by equation 1.

$$V(x) = S(x) + x^{n-k}U(x) \quad (1)$$

V(x) is the n bit long data word transmitted and it consists of the original data and U(x) followed by a codeword S(x) called the CRC-sum. S(x) is the extra bits added to a message in order to provide redundancy so that errors during transmission can be detected. The length of the S(x) is denoted the constraint length. The constraint length of the most commonly used CRC polynomials are between 8 and 32 bits. S(x) is computed according to equation 3.

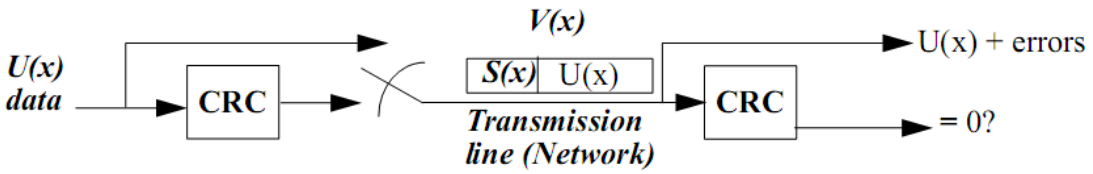
$$X^{n-k}U(x) = a(x)g(x) + S(x) \quad (2)$$

$$\frac{X^{n-k}U(x)}{g(x)} = a(x) + \frac{S(x)}{g(x)} \quad (3)$$

S(x) is by other words the reminder resulting from a division of the data stream and a generator polynomial g(x). Since all codewords are divisible with g(x), the remainder S(x) of the left hand side of (3) has to be zero for a real codeword.

The actual coding-procedure is the same on both the receiving and transmitting end of the line. The CRC encoding/decoding principle is illustrated by Figure 1.

Figure 1. Principle of Error Detection Using the CRC Algorithm



As can be seen in Figure 1, the receiving NT performs a CRC-check on the incoming message and if the result (S(x)) is zero, the transmission was error free. One more practical way of solving this is to compute the CRC only for the first part of the message U(x), and then do a bitwise 2-complements addition with the computed checksum S(x) on the transmission side. If the result is non-zero the receiver will order a retransmission from the sender.

2.2.2 Modulo-2 Binary Division

It turns out that once you start to focus on maximizing the "minimum Hamming distance across the entire set of valid packets," it becomes obvious that simple checksum algorithms based on binary addition don't have the necessary properties. A change in one of the message bits does not affect enough of the checksum bits during addition. Fortunately, you don't have to develop a better checksum algorithm on your own.

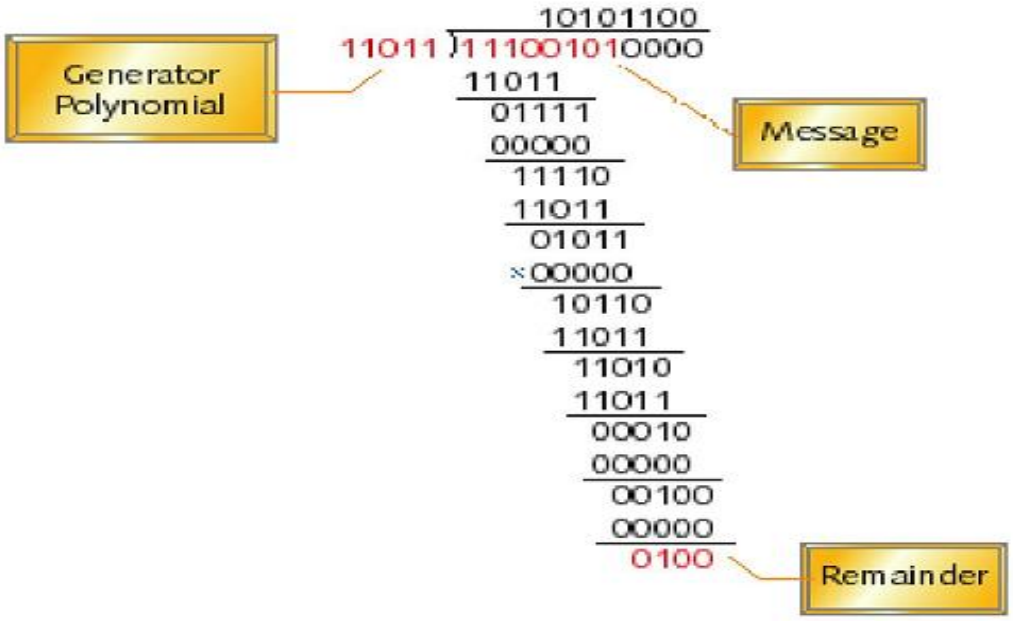
Researchers figured out long ago that modulo-2 binary division is the simplest mathematical operation that provides the necessary properties for a strong checksum.

All of the CRC formulas you will encounter are simply checksum algorithms based on modulo-2 binary division. Though some differences exist in the specifics across different CRC formulas, the basic mathematical process is always the same:

- The message bits are appended with c zero bits; this augmented message is the divided.
- A predetermined c+1-bit binary sequence, called the "generator polynomial", is the divisor.
- The checksum is the c-bit remainder that results from the division operation.

In other words, you divide the augmented message by the generator polynomial, discard the quotient, and use the remainder as your checksum. It turns out that the mathematically appealing aspect of division is that remainders fluctuate rapidly as small numbers of bits within the message are changed. Sums, products, and quotients do not share this property. To see what I mean, look at the example of modulo-2 division in Figure 2.

Figure 2. An Example of Modulo-2 Binary Division



In this example, the message contains eight bits while the checksum is to have four bits. As the division is performed, the remainder takes the values 0111, 1111, 0101, 1011, 1101, 0001, 0010, and, finally, 0100. The final remainder becomes the checksum for the given message.

For most people, the overwhelmingly confusing thing about CRCs is the implementation. Knowing that all CRC algorithms are simply long division algorithms in disguise doesn't help. Modulo-2 binary division doesn't map well to the instruction sets of general-purpose processors. So, whereas the implementation of a checksum algorithm based on addition is straightforward, the implementation of a binary division algorithm with an m+c-bit numerator and a c+1-bit denominator is nowhere close. For one thing, there aren't generally any m+c or c+1-bit registers in which to store the operands. For now, let's just focus on their strengths and weaknesses as potential checksums.

2.2.3 Generator Polynomial

Why is the predetermined c+1-bit divisor that's used to calculate a CRC called a generator polynomial? In my opinion, far too many explanations of CRCs actually try to answer that question. This leads their authors and readers down a long path that involves tons of detail about polynomial arithmetic and the mathematical basis for the usefulness of CRCs. This academic stuff is not important for understanding CRCs sufficiently to implement and/or use them and serves only to create potential confusion.

Suffice it to say here only that the divisor is sometimes called a generator polynomial and that you should never make up the divisor's value on your own. Several mathematically well-understood generator polynomials have been adopted as parts of various international communications standards; you should always use one of those. If you have a background in polynomial arithmetic then you know that certain generator polynomials are better than others for producing strong checksums. The ones that have been adopted internationally are among the best of these.

Table 1 lists some of the most commonly used generator polynomials for 16- and 32-bit CRCs. Remember that the width of the divisor is always one bit wider than the remainder. So, for example, you'd use a 17-bit generator polynomial whenever a 16-bit checksum is required.

Table 1. International Standard CRC Polynomials

Name	Generator Polynomial	Notation
CRC-CCITT	$G(x)=x^{16}+x^{12}+x^5+1$	0x1021
CRC-16	$G(x)=x^{16}+x^{15}+x^2+1$	0x8005
CRC-32	$G(x)=x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$	0x104C11DB7

As is the case with other types of checksums, the width of the CRC plays an important role in the error detection capabilities of the algorithm. Ignoring special types of errors that are always detected by a particular checksum algorithm, the percentage of detectable errors is limited strictly by the width of a checksum. A checksum of c bits can only take one of 2c unique values. Since the number of possible messages is significantly larger than that, the potential exists for two or more messages to have an identical checksum. If one of those messages is somehow transformed into one of the others during transmission, the checksum will appear correct and the receiver will unknowingly accept a bad message. The chance of this happening is directly related to the width of the checksum. Specifically, the chance of such an error is $1/2^c$. Therefore, the probability of any random error being detected is $1-1/2^c$.

To repeat, the probability of detecting any random error increases as the width of the checksum increases. Specifically, a 16-bit checksum will detect 99.9985% of all errors. This is far better than the 99.6094% detection rate of an eight-bit checksum, but not nearly as good as the 99.9999% detection rate of a 32-bit checksum. All of this applies to both CRCs and addition-based checksums. What really sets CRCs apart, however, is the number of special cases that can be detected 100% of the time. For example, two opposite bit inversions (one bit becoming 0, the other becoming 1) in the same column of an addition would cause the error to be undetected. Well, that's not the case with a CRC.

By using one of the mathematically well-understood generator polynomials like those in Table 1 to calculate a checksum, it's possible to state that the following types of errors will be detected without fail:

- A message with any one bit in error
- A message with any two bits in error (no matter how far apart, which column, and so on).
- A message with any odd number of bits in error (no matter where they are).
- A message with an error burst as wide as the checksum itself.

The first class of detectable error is also detected by an addition-based checksum, or even a simple parity bit. However, the middle two classes of errors represent much stronger detection capabilities than those other types of checksum. The fourth class of detectable error sounds at first to be similar to a class of errors detected by addition-based checksums, but in the case of CRCs, any odd number of bit errors will be detected. So the set of error bursts too wide to detect is now limited to those with an even number of bit errors. All other types of errors fall into the relatively high $1-1/2^c$ probability of detection.

3 CRC Software Realization

The CRC-16 is used to describe the CRC software.

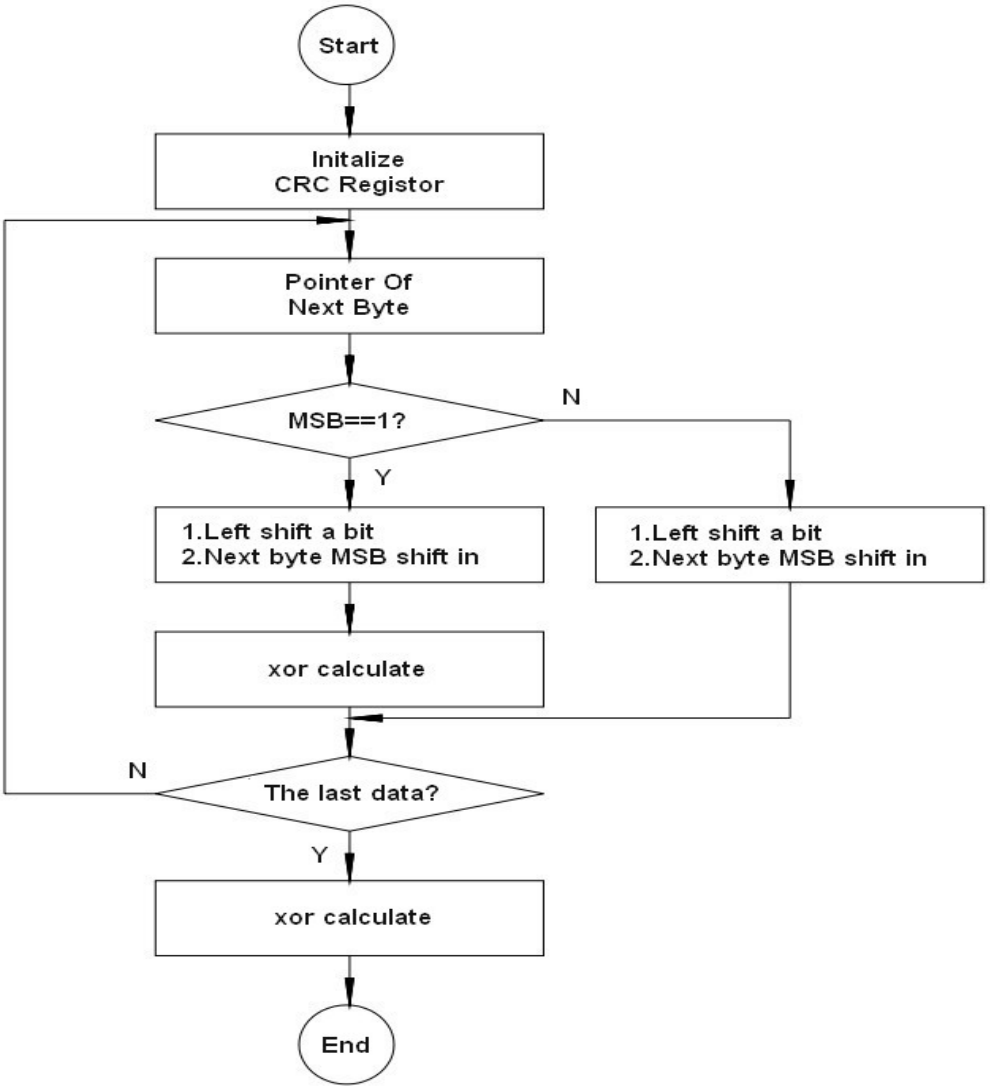
3.1 Bit-by-bit Realization

There are a lot of calculations for CRC in terms of bit-by-bit realization. Once a bit is shifted to left, a calculation will be performed.

Calculation Times=Total bits - 16

The following figure illustrates the bit-by-bit CRC calculation flow.

Figure 3. Bit-by-bit CRC Calculation



- UWORD CRC16_Bitbybit(UCHAR *pMsg,int len)
  Return : CRC value.
  Parameters : pMsg,data buffer.
  len,length of data.
  Description : Bit by bit calculate CRC.
  Example : UWORD Val;
  UCHAR datbuf[16];
  Val=CRC16_Bitbybit(datbuf,16);

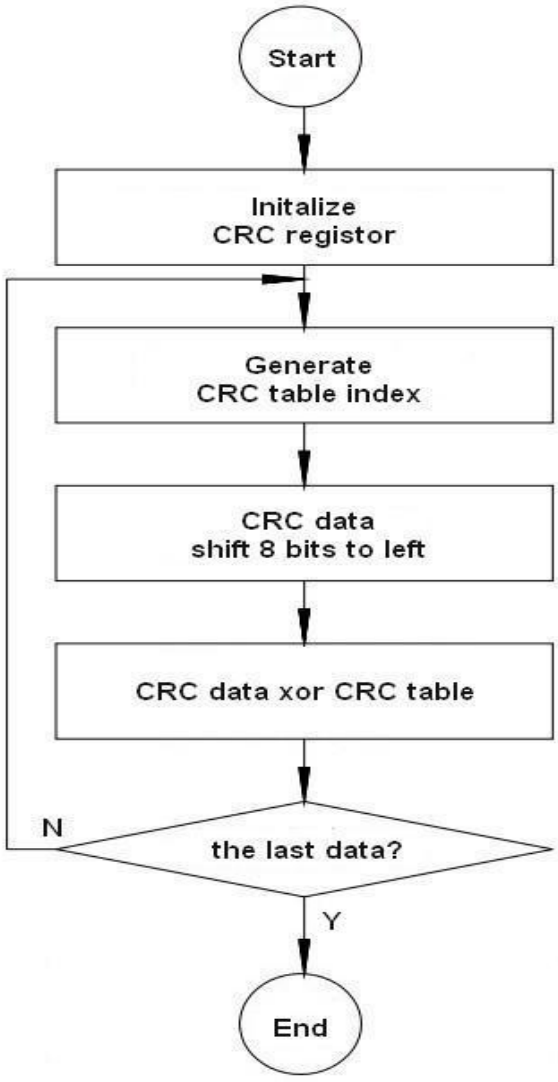
```
UWORD CRC16_Bitbybit(UCHAR *pMsg,int len)
{
  UCHAR i,j;
  UWORD CRC16=0;

  for(j=0;j<len;j++)
  {
    pMsg++;
    for(i=0;i<8;i++,*pMsg<<=1)
    {
      if(CRC16 & 0x8000)
      {
        CRC16<<=1;
        CRC16|=(*pMsg & 0x80)>>7;
        CRC16^=0x8005;
      }
      else
      {
        CRC16<<=1;
        CRC16|=0x8005;
      }
    }
  }

  return CRC16;
}
```

3.2 Table Look Realizing

8-bit table look-based CRC calculation can reduce a lot of duty for the MCU, thus making it operate faster.

Below is the flow of table look CRC calculation:

Figure 4. Table Look Realizing



3.2.1 CRC API

- UWORD CRC16_Table(UCHAR *pMsg,int len)

  Return : CRC value.
  Parameters : pMsg,data buffer.
  len,length of data.
  Description : Table Look calculate CRC.
  Example : UWORD Val;
  UCHAR datbuf[16];
  Val=CRC16_Table(datbuf,16);

```
UWORD CRC16_Table(UCHAR *pMsg, int len)
{
    UCHAR indx;
    UWORD CRC16=0;
    while(len--)
    {
        indx = CRC16 >> 8;
        CRC16 <<= 8;
        CRC16 ^= CRCTable[indx ^ *pMsg++];
    }

    return CRC16;
}
```

- void CRC16TableGen ()

  Return : None.
  Parameters : None.
  Description : Generate CRC table.
  Example : CRC16TableGen();

```
void CRC16TableGen()
{
  int i,j;
  UWORD CRC16;

  for(i=0;i<256;i++)
  {
    CRC16=i;
    CRC16<<=8;
    for(j=0;j<8;j++)
    {
      if(CRC16&0x8000)
      {
        CRC16<<=1;
        CRC16^=0x8005;
      }
      else
        CRC16<<=1;
}
CRCTable[i]=CRC16;
  }
}
```

3.2.2 CRC Table

```
UWORD CRCTable[256]=
{
    0x0000,0x8005,0x800F,0x000A,0x801B,0x001E,0x0014,0x8011,
    0x8033,0x0036,0x003C,0x8039,0x0028,0x802D,0x8027,0x0022,
    0x8063,0x0066,0x006C,0x8069,0x0078,0x807D,0x8077,0x0072,
    0x0050,0x8055,0x805F,0x005A,0x804B,0x004E,0x0044,0x8041,
    0x80C3,0x00C6,0x00CC,0x80C9,0x00D8,0x80DD,0x80D7,0x00D2,
    0x00F0,0x80F5,0x80FF,0x00FA,0x80EB,0x00EE,0x00E4,0x80E1,
    0x00A0,0x80A5,0x80AF,0x00AA,0x80BB,0x00BE,0x00B4,0x80B1,
    0x8093,0x0096,0x009C,0x8099,0x0088,0x808D,0x8087,0x0082,
    0x8183,0x0186,0x018C,0x8189,0x0198,0x819D,0x8197,0x0192,
    0x01B0,0x81B5,0x81BF,0x01BA,0x81AB,0x01AE,0x01A4,0x81A1,
    0x01E0,0x81E5,0x81EF,0x01EA,0x81FB,0x01FE,0x01F4,0x81F1,
    0x81D3,0x01D6,0x01DC,0x81D9,0x01C8,0x81CD,0x81C7,0x01C2,
    0x0140,0x8145,0x814F,0x014A,0x815B,0x015E,0x0154,0x8151,
    0x8173,0x0176,0x017C,0x8179,0x0168,0x816D,0x8167,0x0162,
    0x8123,0x0126,0x012C,0x8129,0x0138,0x813D,0x8137,0x0132,
    0x0110,0x8115,0x811F,0x011A,0x810B,0x010E,0x0104,0x8101,
    0x8303,0x0306,0x030C,0x8309,0x0318,0x831D,0x8317,0x0312,
    0x0330,0x8335,0x833F,0x033A,0x832B,0x032E,0x0324,0x8321,
    0x0360,0x8365,0x836F,0x036A,0x837B,0x037E,0x0374,0x8371,
    0x8353,0x0356,0x035C,0x8359,0x0348,0x834D,0x8347,0x0342,
    0x03C0,0x83C5,0x83CF,0x03CA,0x83DB,0x03DE,0x03D4,0x83D1,
    0x83F3,0x03F6,0x03FC,0x83F9,0x03E8,0x83ED,0x83E7,0x03E2,
    0x83A3,0x03A6,0x03AC,0x83A9,0x03B8,0x83BD,0x83B7,0x03B2,
    0x0390,0x8395,0x839F,0x039A,0x838B,0x038E,0x0384,0x8381,
    0x0280,0x8285,0x828F,0x028A,0x829B,0x029E,0x0294,0x8291,
    0x82B3,0x02B6,0x02BC,0x82B9,0x02A8,0x82AD,0x82A7,0x02A2,
    0x82E3,0x02E6,0x02EC,0x82E9,0x02F8,0x82FD,0x82F7,0x02F2,
    0x02D0,0x82D5,0x82DF,0x02DA,0x82CB,0x02CE,0x02C4,0x82C1,
    0x8243,0x0246,0x024C,0x8249,0x0258,0x825D,0x8257,0x0252,
    0x0270,0x8275,0x827F,0x027A,0x826B,0x026E,0x0264,0x8261,
    0x0220,0x8225,0x822F,0x022A,0x823B,0x023E,0x0234,0x8231,
    0x8213,0x0216,0x021C,0x8219,0x0208,0x820D,0x8207,0x0202
};
```

3.3 CRC Check

The methods of CRC check and generation are almost the same. In CRC check, if the generated CRC check code of whole data block received (include the CRC bytes) is zero, this indicates that all the data is correct.

4 Notes

In applications, users can also define different CRC generator polynomial to realize CRC-8, CRC-16 or CRC-32.

5 More Information

For more information on Cypress MB95200 products, please visit the following website:

www.cypress.com/documentation/application-notes/mb95200-crc-generating-and-checking

6 Bibliography

[1] Ross Williams, A Painless Guide to CRC Error Detection Algorithms.

Document URL: www.repairfaq.org/filipg/LINK/F_crc_v3.html

[2] Bertsekas, Dimitri and Robert Gallager. Data Networks, second ed. Inglewood Cliffs, NJ: Prentice-Hall, 1992, pp. 61-64.

Document History

Document Title: AN205500 - F²MC-8FX Family MB95200H Series CRC Generation and Check

Document Number: 002-05500

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	—	HUAL	04/08/2009	Initial release.
*A	5261676	HUAL	05/09/2016	Migrated Spansion Application note from MCU-AN-500032-E-10 to Cypress format.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Lighting & Power Control	cypress.com/powerpsoc
Memory	cypress.com/memory
PSoC	cypress.com/psoc
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless/RF	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

Cypress Developer Community

Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com