Micriμm

Empowering Embedded Systems

μC/CRC

V1.07

User Manual

www.Micrium.com

Disclaimer

Specifications written in this manual are believed to be accurate, but are not guaranteed to be entirely free of error. Specifications in this manual may be changed for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, Micrium assumes no responsibility for any errors or omissions and makes no warranties. Micrium specifically disclaims any implied warranty of fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies. Brand and product names are trademarks or registered trademarks of their respective holders.

Registration

Please register the software via email. This way we can make sure you will receive updates or notifications of updates as soon as they become available. For registration please provide the following information:

- Your full name and the name of your supervisor
- Your company name
- Your job title
- Your email address and telephone number
- Company name and address
- Your company's main phone number
- Your company's web site address
- Name and version of the product

Please send this information to: **licensing@micrium.com**

Contact address

Micrium
949 Crestview Circle
Weston, FL 33327-1848
U.S.A.
Phone : +1 954 217 2036
FAX : +1 954 217 2037
WEB : **www.micrium.com**
Email : **support@micrium.com**

Manual versions

If you find any errors in this document, please inform us and we will make the appropriate corrections for future releases.

Manual Version	Date	Description
V1.07	2010/10/25	Updated.
V1.06	2010/06/28	Updated.
V1.05	2009/08/16	Updated.
V1.03	2009/04/06	Updated.
V1.02	2008/12/10	Updated.
V1.02	2008/10/31	Added BCH code information.
V1.01	2008/08/16	Added Hamming code information.
V1.01	2008/07/26	Updated/corrected.
V1.00	2007/11/26	Released first version.

Table Of Contents

Introduction

µC/CRC is a stand-alone module for calculating checksums and error correcting codes. Currently, it includes functions to calculate 16- and 32-bit Cyclic Redundancy Checks (CRCs) on data sets and data streams, functions to calculate Hamming codes on large data blocks and functions to calculate BCH codes on large data blocks.

This document describes how to configure and use the µC/CRC module.

Required modules

The current version of µC/CRC requires the µC/LIB module. Please refer to the release notes document for version information.

Chapter 1

Directories and Files

The code and documentation of the µC/CRC module are organized in a directory structure according to "AN-2002, µC/OS-II Directory Structure". Specifically, the files may be found in the following directories:

\Micrium\Software\uC-CRC

This is the main directory for µC/CRC.

\Micrium\Software\uC-CRC\Doc

This directory contains the µC/CRC documentation files, including this one.

\Micrium\Software\uC-CRC\Cfg\Template

This directory contains a template of µC/CRC configuration.

\Micrium\Software\uC-CRC\Source

This directory contains the µC/CRC source code.

edc_crc.* implement CRC calculations in two platform-independent files.
ecc_hamming.* implement Hamming code calculations in two platform-independent files.
ecc_bch_4bit.* implement 4-bit correction BCH code calculations in two platform-independent files.
ecc_bch_8bit.* implement 8-bit correction BCH code calculations in two platform-independent files.

\Micrium\Software\uC-CRC\Ports

This directory contains optional assembly-language ports for core calculations in µC/CRC. A port is not necessary; if a port is available for your toolchain and processor, or you write one, then µC/CRC can be configured to use this to speed up the calculation. Otherwise, the C code in the source files will be used.

The port for CRC calculation is typically named *edc_crc_a.asm*.
The port for Hamming code calculation is typically named *ecc_hamming_a.asm*.
The port for BCH code calculation is typically named *ecc_bch_xbit_a.asm*.

\Micrium\Software\uC-CRC\Win32

This directory contains a Windows program which can generate a CRC table for a custom polynomial so that this can be compiled with your code into ROM, rather than RAM. The name of the executable is:

edc_crc_tblmake.exe

The location of the directory containing the example sample code application is dependent of the evaluation board and contains these files:

app.c
: Application code.

app_cfg.h
: Example application configuration file.

includes.h
: Master include file used by the application.

Chapter 2

Using µC/CRC for CRC Calculations

µC/CRC requires no initialization function for CRC calculation. However, to perform a CRC calculation, you must fill in the members of either a `CRC_MODEL_16` or `CRC_MODEL_32` `struct` (depending on whether your application is using a 16- or 32-bit CRC). These `struct`s are defined as follows:

```
typedef  struct  crc_model_16 {
    CPU_INT16U     Poly;
    CPU_INT16U     InitVal;
    CPU_BOOLEAN    Reflect;
    CPU_INT16U     XorOut;
    CPU_INT16U    *TblPtr;
} CRC_MODEL_16;
```

AND

```
typedef  struct  crc_model_32 {
    CPU_INT32U     Poly;
    CPU_INT32U     InitVal;
    CPU_BOOLEAN    Reflect;
    CPU_INT32U     XorOut;
    CPU_INT32U    *TblPtr;
} CRC_MODEL_32;
```

Typically, the first four members are dictated by the protocol or application for which the CRC will be generated; essentially, these will probably be defined in some reference work, manual or specification. At the very worst, if there is no specification, these probably can be determined by comparing the value of a reference CRC against values for standard CRCs. More information about the meaning of these parameters is covered in Section 2.01.

The fifth member of these `struct`s is determined by hardware and application constraints. Ideally, `TblPtr` will point to an array of pre-computed CRC values that allow the calculation to be accelerated. Eight tables for common parameter combinations are included in the code file, and a Windows program is provided to generate such a table for a different parameter combination. However, if the application will need to calculate a CRC for unknown parameter combination (which is unlikely) or if there is insufficient ROM for the table (which is also unlikely), then this should be a `NULL` pointer so that will know to use the slower method that uses no table for the calculation.

After filling in one of these structures, a CRC can be calculated in one of two ways. If a CRC will be calculated on an entire data set (say, a region in memory or a data buffer), then either `CRC_ChkSumCalc_16Bit()` or `CRC_ChkSumCalc_32Bit()` should be used:

```
CPU_INTxxU  CRC_ChkSumCalc_xxBit (CRC_MODEL_xx  *pmodel,
                                  void          *pbuf,
                                  CPU_INT32U     nbr_octets,
                                  CRC_ERR       *perr);
```

Where **xx** is either 16 or 32. This returns the checksum for the buffer. On the other hand, if a CRC will be calculated for a data stream (for example, for data read from an EEPROM or received serially), then a set of three or four functions will be used:

1. '`CRC_Open_xxBit()`' called to initialize '`CRC_CALC_xx`' structure.

2. '`CRC_WrBlock_xxBit()`' called for each block of bytes to factor into CRC.

   AND / OR

   '`CRC_WrOctet_xxBit()`' called for each octet to factor into CRC.

3. '`CRC_Close_xxBit()`' called to get final CRC value.

Where **xx** is either 16 or 32.

2.01 CRC theory & parameters

Before going into an example of µC/CRC usage, a few basic concepts about CRC calculation will be explained. These explanations attempt to be practical, rather than theoretical: though the µC/CRC module takes care of the calculation, the user must specify the parameters of the calculation. This provides sufficient information for specifying the parameters.

A CRC is actually the remainder of modulo-2 binary division of "data" by a constant divisor. The remainder is often just called the CRC. In this division, the "data" is the sequence of input bytes (or the single block of bytes) over which the checksum is calculated. Because this operation can be viewed as an operation on polynomials with binary coefficients, the constant divisor is typically called the "polynomial". For example, a common 16-bit CRC uses the polynomial

$$x^{16} + x^{12} + x^5 + 1$$

As a binary divisor, the polynomial is written as `0x1021`, which has its 12^{th}, 5^{th}, and 0^{th} bits set (which are the lower terms present in the polynomial). Since a 16-bit CRC always has a highest-order term of x^{16}, it is unnecessary to encode that into the binary polynomial. A comparable scheme may be used to transform a 32-bit polynomial into a binary divisor.

One defining characteristic of a CRC is its "width". Technically, this is the exponent of the highest-order term in the polynomial. Practically, this is the number of bits in the calculated CRC. µC/CRC supports the most common widths, 16 bits and 32 bits. In order to provide greater calculation speed, there are two sets of CRC functions. One set, which ends in "`_16Bit`", should be used for a 16-bit calculation; the other set, which ends in "`_32Bit`" should be used for a 32-bit calculation.

Some applications complicate the CRC calculation in order to provide greater security or some practical benefit. µC/CRC handles three common complications:

- Some CRCs specify a starting remainder (or "input value"). A typical value is `0xFFFF` (for a 16-bit CRC) or `0xFFFFFFFF` (for a 32-bit CRC).

- Some CRCs XOR the final remainder with constant value. A typical value is `0xFFFF` (for a 16-bit CRC) or `0xFFFFFFFF` (for a 32-bit CRC), which essentially complements the CRC.

- Some CRCs reverse the order of bits ("reflect") in the input bytes and/or in the output CRC. It is most common for this to be performed either on both the input and the output or on neither the input nor the output.

In summary, there are four input parameters for a CRC calculation:

- **Polynomial**. The polynomial, as a 16- or 32-bit integer (for a 16- or 32-bit CRC, respectively), which will be used in the calculation.

- **Input value.** A 16- or 32-bit integer (for 16- or 32-bit CRC, respectively) which is the initial CRC value.

- **Output XOR value.** A 16- or 32-bit integer (for 16- or 32-bit CRC, respectively) with which the final CRC is XOR'd.

- **Reflect.** A binary value which controls whether the input and output should be reflected.

2.02 CRC table generation & use

µC/CRC can calculate a CRC in three ways:

- **Non-table driven.** In the simplest case, the CRC is calculated bit-by-bit for each data byte. If this is used, the fifth member (`TblPtr`) of the `CRC_MODEL_16` or `CRC_MODEL_32` `struct` must be a NULL pointer. This method cannot be used with a data stream calculation (see Listing 2-1), but can be used with a block calculation (see Listing 2-3). The non-table driven implementation may be necessary if the CRC parameters will not be known at compile-time.

- **Table-driven**. The CRC calculation can be made faster by using a pre-computed a table of CRC values (optimally, stored in ROM). For a 16-bit CRC, this will be a table of 256 16-bit values; for a 32-bit CRC, this will be a table of 256 32-bit values. The table depends on only the polynomial and whether the input data is reflected. Several tables, as discussed below, are included in `crc.c`.

- **Optimized table-driven**. By coding the inner loop of the table-driven implementation in assembly, it is possible to speed up the calculation (sometimes, significantly). A port is provided for ARM processor (for IAR EW, though it could easily be adapted to other toolchains).

Table 2-1 contains the calculation time for a 1024 byte buffer using each of these methods on an 100-MHz ARM9 processor executing in ARM mode. Obviously, the table-driven calculation is much more efficient, particularly when reflection is necessary. Tables of pre-computed CRC values are included in *edc_crc.c* for four different polynomials, for use with or without reflection. Inclusion for each of these is controlled by an individual `#define` (see Section 2.03, Module Configuration); Table 2-2 contains the names of these tables and the name of the associated `#define`.

Width	Reflected?	Method	Calculation speed (MB/s)*	Calculation time (ms), 512-byte buffer
16- or 32 bit	yes or no	Table driven, optimized	5.82 MB/s	0.088 ms
16- or 32 bit	yes or no	Table driven, unoptimized	3.90 MB/s	0.131 ms
16- or 32-bit	no	Non-table driven	1.43 MB/s	0.358 ms
16- or 32-bit	yes	Non-table driven	1.24 MB/s	0.414 ms

Table 2-1. Example Calculation Speeds
***MB/s = 1,000,000 B/s**

Polynomial	Reflected?	Table	Include `#define`
`0x1021`	no	`CRC_TblCRC16_1021[]`	`EDC_CRC_CFG_CRC16_1021_EN`
`0x8005`	no	`CRC_TblCRC16_8005[]`	`EDC_CRC_CFG_CRC16_8005_EN`
`0x8048`	no	`CRC_TblCRC16_8048[]`	`EDC_CRC_CFG_CRC16_8048_EN`
`0x04C11DB7`	no	`CRC_TblCRC32[]`	`EDC_CRC_CFG_CRC32_EN`
`0x1021`	yes	`CRC_TblCRC16_1021_ref[]`	`EDC_CRC_CFG_CRC16_1021_REF_EN`
`0x8005`	yes	`CRC_TblCRC16_8005_ref[]`	`EDC_CRC_CFG_CRC16_8005_REF_EN`
`0x8048`	yes	`CRC_TblCRC16_8048_ref[]`	`EDC_CRC_CFG_CRC16_8048_REF_EN`
`0x04C11DB7`	yes	`CRC_TblCRC32_ref[]`	`EDC_CRC_CFG_CRC32_REF_EN`

Table 2-2. Provided CRC Tables

In case a CRC not covered in this table is to be used, a Windows program is provided to generate the table for a specified set of input parameters. The executable, *crc_tblmake.exe*, provided in the directory *\Micrium\Software\uC-CRC\Win32* should be invoked with five input parameters:

```
edc_crc_tblmake.exe  <bits 16/20> <poly> <initval> <reflect? YES/NO> <xorout>
```

where the parameters are

- `bits` should be either 16 (if the values are for a 16-bit CRC) or 32 (if the values are for a 32-bit CRC).
- `poly` should be the polynomial, in hexadecimal.
- `initval` should be the initial value, in hexadecimal.
- `reflect` should `YES` if the input bytes and result should be reflected.
- `xorout` is the value with which the result should be XOR'd, in hexadecimal.

An example invocation is

```
edc_crc_tblmake.exe  16  0x1021  0xFFFF  NO  0x0000
```

The program will output to the command line a CRC for the example input string "123456789". It will also create a file in the executable directory, *app_crctbl.c*, which contains an array named `App_CRCTbl[]` which can be copied into your application code file. You application should use a pointer to this table as the fifth member (`TblPtr`) of the `CRC_MODEL_16` or `CRC_MODEL_32` struct that is the first argument of the `Open()` and `ChkSumCalc()` functions.

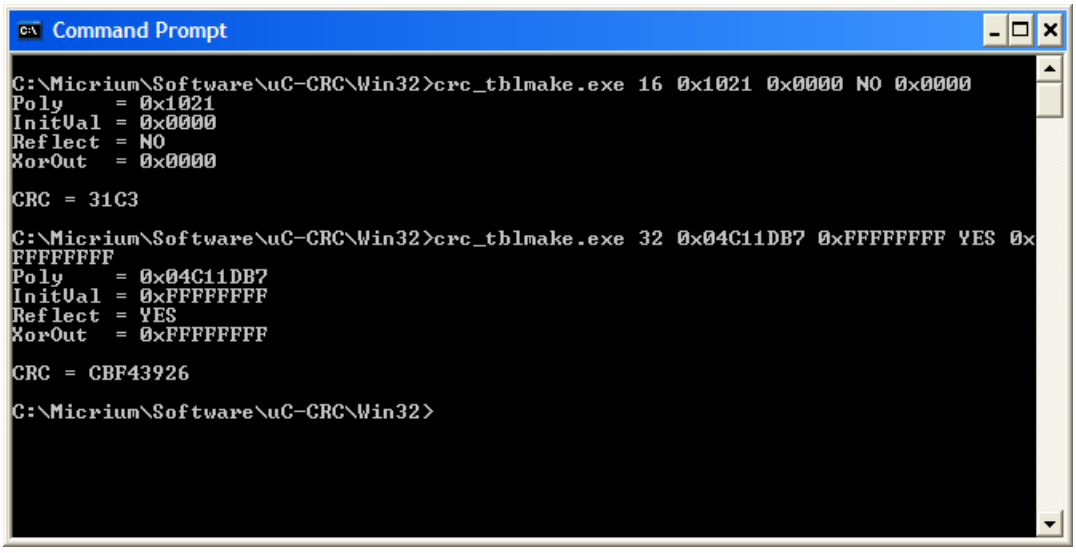

Figure 2-1. *edc_crc_tblmake.exe* Example Output

2.03 CRC example use

Listing 2-1. Example use, data stream calculation

```
static  CRC_MODEL_16  App_CRCModel = {
    0x1021,                                                          (1)
    0xFFFF,                                                          (2)
    DEF_NO,                                                          (3)
    0x0000,                                                          (4)
    (CPU_INT16U *)&CRC_TblCRC16_1021[0]                              (5)
};


void  App_ChkSumCalc (void)
{
    CPU_ERR      err;
    CPU_INT16U   crc;
    CRC_CALC_16  calc;
    CPU_INT32U   i;
    CPU_INT08U   octet;


    APP_TRACE_DEBUG(("Testing CRC, data stream calculation ...\n\r"));

    CRC_Open_16Bit(App_CRCModel, &calc, &err);

    if (err != CRC_ERR_NONE) {
        APP_TRACE_DEBUG(("... open failed.\n\r")
        return;
    }

    for (i = 0; i < 1024; i++) {
        octet = Ser_RdByte();                       /* Read byte from serial port.  */
        CRC_WrOctet_16Bit(&calc, octet);            /* Update CRC.                  */
    }

    crc = CRC_Close_16Bit(&calc);

    APP_TRACE_DEBUG(("... result = %04X.\n\r", crc));
}
```

L2-1(1) The input polynomial used for the calculation is `0x1021`; specified in polynomial form, this is $x^{16} + x^{12} + x^5 + 1$.

L2-1(2) The input value (starting CRC value) is `0xFFFF`.

L2-1(3) The input data and final CRC will NOT be reflected.

L2-1(4) The final CRC will be XOR'd with `0x0000`, which will not change the CRC value.

L2-1(5) A table will be used for the CRC calculation. The table `CRC_TblCRC16_1021[]` is provided in crc.c; for this to be used, `CRC_CFG_CRC16_1021_EN` must be `DEF_ENABLED`. Note that the third struct member, which controls reflection, must be properly specified. Since `CRC_TblCRC16_1021[]` is being used, the third member must be `DEF_NO`. If the table `CRC_TblCRC16_1021_ref[]` were being used, then the third member would need to be `DEF_YES`.

A table MUST be provided for a data stream calculation; the CRC can only be generated using the table-driven method.

Listing 2-2. Example use, data block calculation (table-driven)

```
static  CRC_MODEL_32  App_CRCModel = {
    0x04C11DB7,                                                    (1)
    0xFFFFFFFF,                                                    (2)
    DEF_YES,                                                       (3)
    0xFFFFFFFF,                                                    (4)
    (CPU_INT32U *)&CRC_TblCRC32_ref[0]                             (5)
};

static  CPU_INT08U  App_CRCBuf[1024];

void  App_ChkSumCalc (void)
{
    CPU_ERR     err;
    CPU_INT32U  crc;


    crc = CRC_ChkSumCalc_32Bit(App_CRCModel, App_CRCBuf, sizeof(App_CRCBuf), &err)

    if (err == EDC_CRC_ERR_NONE) {
        APP_TRACE_DEBUG((“CRC result = %08X.\n\r”, crc));
    } else {
        APP_TRACE_DEBUG((“CRC calculation failed.\n\r”));
    }
}
```

L2-2(1) The input polynomial used for the calculation is `0x04C11DB7`; specified in polynomial form, this is $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^5 + x^4 + x^2 + x + 1$.

L2-2(2) The input value (starting CRC value) is `0xFFFFFFFF`.

L2-2(3) The input data and final CRC WILL be reflected.

L2-2(4) The final CRC will be XOR’d with `0xFFFFFFFF`, which complements the CRC value.

L2-2(5) A table will be used for the CRC calculation. The table `CRC_TblCRC32_ref[]` is provided in crc.c; for this to be used, EDC_`CRC_CFG_CRC32_REF_EN` must be `DEF_ENABLED`. Note that the third struct member, which controls reflection, must be properly specified. Since `CRC_TblCRC32_ref[]` is being used, the third member must be `DEF_YES`. If the table `CRC_TblCRC32[]` were being used, then the third member would need to be `DEF_NO`.

The cast is necessary since the table is declare as `const`, but the struct member is not.

Listing 2-3. Example use, data block calculation (non-table-driven)

```
static  CRC_MODEL_32  App_CRCModel = {
    0x04C11DB7,
    0xFFFFFFFF,
    DEF_YES,
    0xFFFFFFFF,
    (CPU_INT32U *)0                                                    (1)
};

static  CPU_INT08U  App_CRCBuf[1024];

void  App_ChkSumCalc (void)
{
    CPU_ERR     err;
    CPU_INT32U  crc;


    crc = CRC_ChkSumCalc_32Bit(App_CRCModel, App_CRCBuf, sizeof(App_CRCBuf), &err)

    if (err == EDC_CRC_ERR_NONE) {
        APP_TRACE_DEBUG((“CRC result = %08X.\n\r”, crc));
    } else {
        APP_TRACE_DEBUG((“CRC calculation failed.\n\r”));
    }
}
```

L2-3(1) If you do not need to use a table (or cannot use a table) to take advantage of the speed of the table-driven calculation, then a non-table-driven calculation can be performed by assigning a `NULL` pointer to the fifth member of the `CRC_MODEL_16` or `CRC_MODEL_32 struct`.

Chapter 3

Using µC/CRC for Hamming Code Calculations

µC/CRC requires no initialization function for Hamming code calculations. Indeed, absolutely no setup is necessary. After a block of data is formed, the application should calculated a Hamming code with one of the functions:

`Hamming_Calc_004()`	Calculate Hamming code for 4-byte block.
`Hamming_Calc_256()`	Calculate Hamming code for 256-byte block.
`Hamming_Calc()`	Calculate Hamming code for *n*-byte block.

which are prototyped

```
void  Hamming_Calc_####(void        *p_buf,
                        CPU_SIZE_T   len,
                        CPU_INT08U  *p_ecc,
                        CPU_ERR     *p_err);
```

The Hamming code will be returned in `p_ecc`, which MUST point to an array of three bytes. When data integrity must be verified (for example, after the block is written to and read from a Flash medium), the respective of the check functions should be called:

`Hamming_Chk_004()`	Check Hamming code for 4-byte block.
`Hamming_Chk_256()`	Check Hamming code for 256-byte block.
`Hamming_Chk()`	Check Hamming code for *n*-byte block.

which are prototyped

```
CPU_INT08U  Hamming_Chk_####(void          *p_buf,
                             CPU_SIZE_T     len
                             CPU_INT08U    *p_ecc,
                             ECC_ERR_ADDR  *p_err_addr_tbl,
                             CPU_INT08U     max_errs,
                             CPU_ERR       *p_err);
```

The return parameters `p_err_addr_tbl` and `p_err` provide information about any corruption in the data. `p_err` takes one of the following values:

`ECC_ERR_NONE`	Hamming code verified (no error).
`ECC_ERR_CORRECTABLE`	Correctable error detected.
`ECC_ERR_UNCORRECTABLE`	Uncorrectable error detected.

If a correctable error is found, `p_err_addr_tbl->AddrOctet` and `p_err_addr_tbl->AddrBit` will specify the location of the bit error. To correct this, the application should complement that bit. For example:

```
Hamming_Chk_256(&buf[0],
                 256,
                 hamming_prev,
                &err_addr,
                 1,
                &err);

if (err == ECC_ERR_CORRECTABLE) {
    buf[err_addr.AddrOctet] ^= DEF_BIT(err_addr.AddrBit);
}
```

It is also possible to call directly one of the correct functions:

`Hamming_Correct_004()`	Check Hamming code for 4-byte block.
`Hamming_Correct_256()`	Check Hamming code for 256-byte block.
`Hamming_Correct()`	Check Hamming code for *n*-byte block.

which are prototyped

```
CPU_INT08U  Hamming_Correct_####(void          *p_buf,
                                 CPU_SIZE_T     len
                                 CPU_INT08U    *p_ecc,
                                 CPU_ERR       *p_err);
```

The return parameters `p_err` provide information about any corruption in the data. `p_err` takes one of the following values:

`ECC_ERR_NONE`	Hamming code verified (no error) or corrected (if needed).
`ECC_ERR_UNCORRECTABLE`	Uncorrectable error detected.

3.01 Hamming code theory

A Hamming code is an error-correcting code that can correct single-bit errors and detect double-bit errors. The heart of the underlying mathematics can be viewed as a novel use of parity checks. In some communication protocols (like RS-232), a parity bit may be appended to each data unit (for RS-232, each byte). The most common schemes require the transmitter assign this bit so that the total number of '1' bits in each data unit is even or odd. The receiver validates the data by counting the number of '1' bits; if not even (or odd, as required), an error has occurred, requiring the receiver wait for retransmission.

Note that a single parity bit cannot pinpoint the location of the error (the bit that was flipped). For a Hamming code, parity bits are calculated on overlapping bit groups within a data unit. A single bit error will cause a unique set of parity checks to fail, revealing the error location.

For large data blocks (longer than 128 bytes), the calculation is most efficient on data viewed byte-wise. As shown in Figure 3-1, the data bits are arranged as an array of eight columns (the bits) by a variable number of rows (the bytes). First, parities are calculated for each row (the intermediate byte parities) and column (the intermediate bit parities). Next, parities are calculated for overlapping subsets of the intermediate byte parities (`L004O`, `L004E`, `L002O`, `L002E`, `L001O` and `L001E`) and for overlapping subsets of the intermediate bit parities (`C4O`, `C4E`, `C2O`, `C2E`, `C1O` and `C1E`). These bits form the Hamming code.

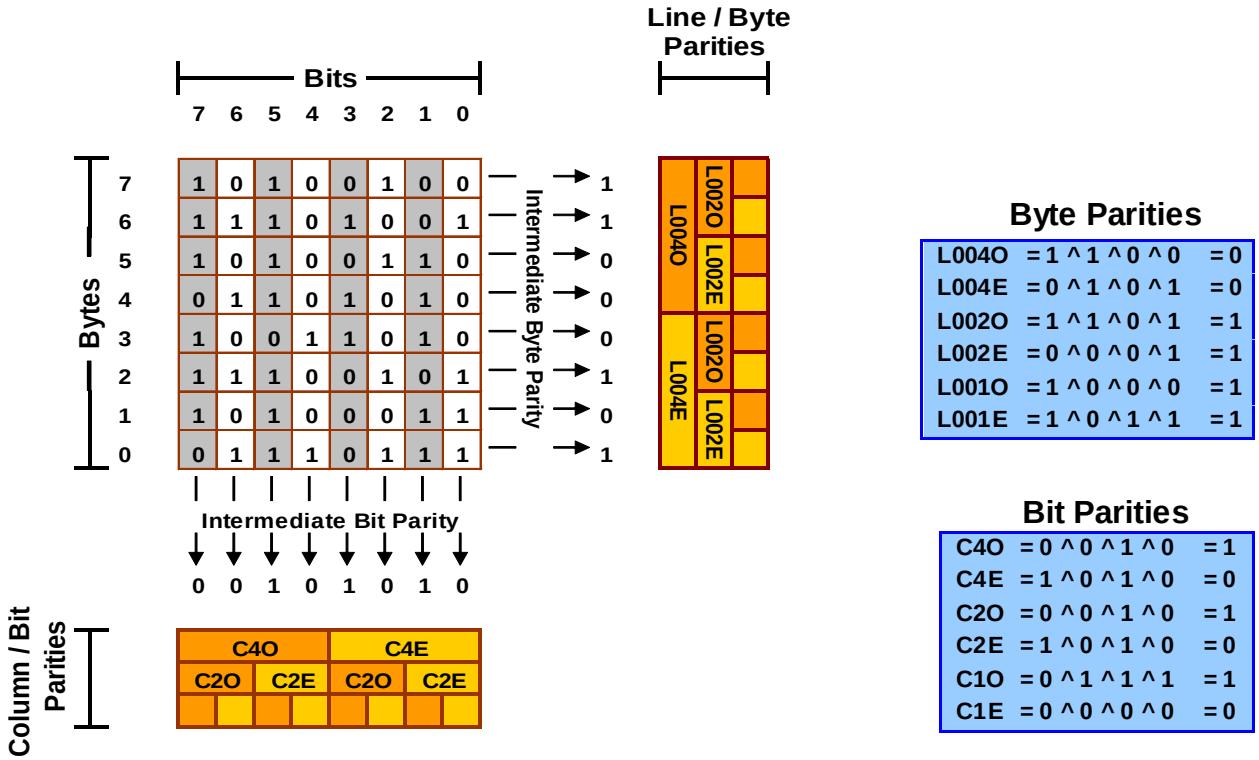

Figure 3-1. Hamming Code Parity Calculation

3.01.01 Hamming code implementation

The µC/CRC Hamming codes parity bits are organized as shown in Figure 3-2.

Bits 23-16:	**C4O**	**C4E**	**C2O**	**C2E**	**C1O**	**C1E**	**1**	**1**
Bits 15-8:	**L128O**	**L128E**	**L064O**	**L064E**	**L032O**	**L032E**	**L016O**	**L016E**
Bits 7-0:	**L008O**	**L008E**	**L004O**	**L004E**	**L002O**	**L002E**	**L001O**	**L001E**

Figure 3-2. Hamming code bit organization (256-byte buffers)

Each symbol represents the parity of one subset of data. For example, L001E represents the parity of even lines (bytes). L001O represents the parity of odd lines. L032O represents the parity of odd groups of 32 lines/bytes. C4O represents odd group of 4 bits/columns (the parity of the sum of bits 4, 5, 6, 7 for every byte).

3.02 Hamming code example use

Listing 3-1. Example use, Hamming code calculation

```
void  App_HammingCalc (void)
{
    CPU_ERR     err;
    CPU_INT32U  hamming;
    CPU_INT08U  buf[256];

    .
    .
    .

    Hamming_Calc_256(&buf[0], 256, (CPU_INT08U *)&hamming, &err);      /* (1) */

    .
    .
    .
}
```

L3-1(1) Calculate Hamming code for 256-byte buffer.

Listing 3-2. Example use, Hamming code check

```
void  App_HammingCalc (void)
{
    ECC_ERR_ADDR  err_addr;
    CPU_ERR       err;
    CPU_INT32U    hamming_prev;
    CPU_INT08U    p_buf;

    .
    .
    .

    Hamming_Chk_256((void         *) p_buf,                /* (1) */
                    (CPU_SIZE_T    ) 256,
                    (CPU_INT08U   *)&hamming_prev,
                    (ECC_ERR_ADDR *)&err_addr,
                    (CPU_INT08U    ) 1,
                    (CPU_ERR      *)&err);

    switch (err) {
        case ECC_ERR_NONE:
             APP_TRACE_INFO((“Data verified.\r\n”));
             break;

        case ECC_ERR_CORRECTABLE:
             APP_TRACE_INFO((“Correctable error @ bit %d of octet %d.\r\n”,
                             err_addr.AddrBit, err_addr.AddrOctet));
                                                          /* (2) */
             p_buf[err_addr.AddrOctet] ^= DEF_BIT(err_addr.AddrBit);
             break;

        default:
        case ECC_ERR_UNCORRECTABLE:
             APP_TRACE_INFO((“Uncorrectable error.\r\n”));  /* (3) */
             break;
    }

    .
    .
    .
}
```

L3-2(1) Check Hamming code for 256-byte buffer.

L3-2(2) Correct single-bit error.

L3-2(3) Uncorrectable error.

Chapter 4

Using µC/CRC for BCH Code Calculations

µC/CRC requires no initialization function for BCH code calculations. Indeed, absolutely no setup is necessary. After a block of data is formed, the application should calculated a Hamming code with one of the functions:

`BCH_4Bit_Calc()`	Calculate BCH code for 512- to 528-byte block for 4-bit correction.
`BCH_8Bit_Calc()`	Calculate BCH code for 512- to 528-byte block for 8-bit correction.

which are prototyped

```
void  BCH_####_Calc(void        *p_buf,
                    CPU_INT08U  *p_ecc,
                    CPU_ERR     *p_err);
```

The BCH code will be returned in `p_ecc`, which MUST point to an array of seven bytes or thirteen bytes, respectively. When data integrity must be verified (for example, after the block is written to and read from a Flash medium), the respective of the check functions should be called:

`BCH_4Bit_Chk()`	Check BCH code for 512- to 528-byte block for 4-bit correction.
`BCH_8Bit_Chk()`	Check BCH code for 512- to 528-byte block for 8-bit correction.

which are prototyped

```
CPU_INT08U  BCH_####_Chk(void          *p_buf,
                         CPU_SIZE_T     len,
                         CPU_INT08U    *p_ecc,
                         ECC_ERR_ADDR  *p_err_addr_tbl,
                         CPU_INT08U     max_errs,
                         CPU_ERR       *p_err);
```

The return parameters `p_err_addr_tbl` and `p_err` provide information about any corruption in the data. `p_err` takes one of the following values:

`ECC_ERR_NONE`	BCH code verified (no error).
`ECC_ERR_CORRECTABLE`	Correctable error(s) detected.
`ECC_ERR_UNCORRECTABLE`	Uncorrectable error detected.

If correctable error(s) are found, `p_err_addr_tbl[i].AddrOctet` and `p_err_addr_tbl[i].AddrBit` will specify the location of the bit error(s). The return value of the function will indicate the number of bit errors. To correct this, the application should complement those bit(s). For example:

```
err_cnt = BCH_8Bit_Chk(&buf[0],
                        512,
                       &p_ecc[0],
                       &err_addr_tbl,
                        8,
                       &err);

if (err == ECC_ERR_CORRECTABLE) {
    for (i = 0; i < err_cnt; i++) {
        buf[err_addr_tbl[i].AddrOctet] ^= DEF_BIT(err_addr_tbl[i].AddrBit);
    }
}
```

4.01 BCH code theory

A BCH (Bose-Chaudhuri-Hocquenghem) code is a random multiple-error-correcting code; µC/CRC implements a subclass called primitive BCH codes. Algorithms are provided for correcting up to 4 bit-errors and up to 8 bit-errors in a 512-byte buffer. The check code length for the 4 bit-error algorithm is 52-bits, which fits into a 7-byte buffer. The check code length for the 8-bit error algorithm is 104-bits, or 13-bytes.

4.02 BCH code implementation

The µC/CRC BCH codes are calculated using a table of pre-computed BCH code values. Similarly, in order to determine error locations, tables of Galois field elements must be used. These requirements introduce a significant ROM requirement, as listed in Table 4-1.

Module	Code Size (B)	Data ROM Size (B)
ecc_bch_4bit.c	3608 B	53248 B
*ecc_bch_4bit.c**	4112 B	36864 B
ecc_bch_8bit.c	4772 B	57344 B
*ecc_bch_8bit.c**	3568 B	40960 B

Table 4-1. µC/CRC BCH Code ROM Requirement
***Reduced footprint (slower execution)**

Example calculation speeds, benchmarked on an 100-MHz ARM9 platform executing in ARM mode, are given in Tables 4-2 and 4-3.

Function	Calculation speed (MB/s)*	Calculation time (ms), 512-B buffer	
		Typical***	Worst Case****
`BCH_8Bit_Calc()`	1.72 MB/s	0.298 ms	-----
`BCH_8Bit_Calc()**`	2.93 MB/s	0.175 ms	-----
`BCH_8Bit_Chk()` (with no errors)	2.83 MB/s	0.181 ms	-----
`BCH_8Bit_Chk()` (with 1 error)	1.92 MB/s	0.265 ms	-----
`BCH_8Bit_Chk()` (with 2 errors)	1.83 MB/s	0.280 ms	-----
`BCH_8Bit_Chk()` (with 3 errors)	-----	2.11 ms	5.73 ms
`BCH_8Bit_Chk()` (with 4 errors)	-----	3.48 ms	7.44 ms
`BCH_8Bit_Chk()` (with 5 errors)	-----	4.62 ms	9.12 ms
`BCH_8Bit_Chk()` (with 6 errors)	-----	5.66 ms	10.8 ms
`BCH_8Bit_Chk()` (with 7 errors)	-----	6.70 ms	12.5 ms
`BCH_8Bit_Chk()` (with 8 errors)	-----	7.64 ms	14.2 ms

Table 4-2. Example 8-Bit BCH Code Calculation Speeds
***MB/s = 1,000,000 B/s**
****Optimized assembly function**
*****For multiple error tests, errors spaced equally within buffer**
******For multiple error tests, errors at end of buffer**

Function	Calculation speed (MB/s)*	Calculation time (ms), 512-B buffer	
		Typical***	Worst Case****
`BCH_4Bit_Calc()`	1.90 MB/s	0.269 ms	-----
`BCH_4Bit_Calc()**`	3.19 MB/s	0.160 ms	-----
`BCH_4Bit_Chk()` (with no errors)	2.95 MB/s	0.173 ms	-----
`BCH_4Bit_Chk()` (with 1 error)	2.48 MB/s	0.207 ms	-----
`BCH_4Bit_Chk()` (with 2 errors)	2.39 MB/s	0.215 ms	-----
`BCH_4Bit_Chk()` (with 3 errors)	-----	2.04 ms	5.66 ms
`BCH_4Bit_Chk()` (with 4 errors)	-----	3.41 ms	7.36 ms

Table 4-3. Example 4-Bit BCH Code Calculation Speeds
***MB/s = 1,000,000 B/s**
****Optimized assembly function**
*****For multiple error tests, errors spaced equally within buffer**
******For multiple error tests, errors at end of buffer**

4.03 BCH code example use

Listing 4-1. Example use, BCH code calculation

```
void  App_BCH_Calc (void)
{
    CPU_ERR     err;
    CPU_INT08U  bch[13];
    CPU_INT08U  buf[512];

    .
    .
    .

    BCH_8Bit_Calc(&buf[0], 512, (CPU_INT08U *)&bch[0], &err);              /* (1) */

    .
    .
    .
}
```

L4-1(1) Calculate 8-bit error-correcting BCH code for 512-byte buffer.

Listing 4-2. Example use, BCH code check

```
void  App_BCH_Calc (void)
{
    ECC_ERR_ADDR  err_addr;
    CPU_ERR       err;
    CPU_INT08U    bch_prev[8];
    CPU_INT08U    p_buf;
    CPU_INT08U    err_cnt[8];
    CPU_INT08U    I;
    .
    .
    .

    err_cnt = BCH_8Bit_Chk((void         *) p_buf,           /* (1) */
                           (CPU_SIZE_T    ) 512,
                           (CPU_INT08U   *)&bch_prev[8],
                           (ECC_ERR_ADDR *)&err_addr,
                           (CPU_INT08U    ) 8,
                           (CPU_ERR      *)&err);

    switch (err) {
        case ECC_ERR_NONE:
             APP_TRACE_INFO((“Data verified.\r\n”));
             break;

        case ECC_ERR_CORRECTABLE:
             APP_TRACE_INFO((“%d correctable errors:\r\n”, err_cnt));
             for (i = 0; i < err_cnt; i++) {               /* (2) */
                 p_buf[err_addr[i].AddrOctet] ^= DEF_BIT(err_addr[i].AddrBit);
                 APP_TRACE_INFO((“    bit %d of octet %d\r\n”, err_addr[i].AddrBit,
                                                              err_addr[i].AddrOctet));
             }
             break;

        default:
        case ECC_ERR_UNCORRECTABLE:
             APP_TRACE_INFO((“Uncorrectable error.\r\n”));  /* (3) */
             break;
    }

    .
    .
    .
}
```

L4-2(1) Check 8-bit error-correcting BCH code for 512-byte buffer.

L4-2(2) Correct error(s).

L4-2(3) Uncorrectable error.

Appendix A

µC/CRC Application Programming Interface (API) Functions & Macros

Your application interfaces to µC/CRC using any of the functions or macro's described in this appendix.

A.01 CRC Functions

A.01.01 CRC_ChkSumCalc_xxBit()

These functions calculate a 16-/32-bit CRC over a buffer of data.

Prototypes

```
CPU_INT16U   CRC_ChkSumCalc_16Bit (CRC_MODEL_16  *p_model,
                                   void          *p_buf,
                                   CPU_INT32U     nbr_octets,
                                   CPU_ERR       *p_err);

CPU_INT32U   CRC_ChkSumCalc_32Bit (CRC_MODEL_32  *p_model,
                                   void          *p_buf,
                                   CPU_INT32U     nbr_octets,
                                   CPU_ERR       *p_err);
```

Arguments

`p_model` Pointer to model to use in calculation.

`p_buf` Pointer to data buffer over which CRC is calculated.

`nbr_octets` Number of data octets in buffer.

`p_err` Pointer to variable that will receive the return error code from this function :

`EDC_CRC_ERR_NONE`	No error.
`EDC_CRC_ERR_NULL_PTR`	Argument '`p_model`' or '`p_buf`' passed a `NULL` pointer.

Returned Value

16-/32-bit CRC.

Example

Listing A-1. Example use, data block calculation

```
static  CRC_MODEL_16  App_CRCModel = {
    0x1021,
    0xFFFF,
    DEF_NO,
    0x0000,
    (CPU_INT16U *)&CRC_TblCRC16_1021[0]
};

static  CPU_INT08U  App_CRCBuf[1024];

void  App_ChkSumCalc (void)
{
    CPU_ERR     err;
    CPU_INT16U  crc;


    crc = CRC_ChkSumCalc_16Bit(App_CRCModel, App_CRCBuf, sizeof(App_CRCBuf), &err)

    if (err == EDC_CRC_ERR_NONE) {
        APP_TRACE_DEBUG((“CRC result = %04X.\n\r”, crc));
    } else {
        APP_TRACE_DEBUG((“CRC calculation failed.\n\r”));
    }
}
```

A.01.02 CRC_Close_xxBit()

Closes (ends) a CRC calculation.

Prototypes

```
CPU_INT16U  CRC_Close_16bit (CRC_CALC_16  *p_calc);

CPU_INT32U  CRC_Close_32bit (CRC_CALC_32  *p_calc);
```

Arguments

`p_calc` Pointer to calculation.

Returned Value

16-/32-bit CRC.

Notes/Warnings

For more information, please see `CRC_Open_xxBit()`.

Example

See `CRC_Open_xxBit()`.

A.01.03 CRC_Open_xxBit()

Open (begin) a CRC calculation for a data stream.

Prototypes

```
void  CRC_Open_16Bit (CRC_MODEL_16  *p_model,
                      CRC_CALC_16   *p_calc,
                      CPU_ERR       *p_err);

void  CRC_Open_32Bit (CRC_MODEL_32  *p_model,
                      CRC_CALC_32   *p_calc,
                      CPU_ERR       *p_err);
```

Arguments

`p_model` Pointer to model to use in calculation.

`p_calc` Pointer to calculation.

`p_err` Pointer to variable that will receive the return error code from this function :

`EDC_CRC_ERR_NONE`	No error.
`EDC_CRC_ERR_NULL_PTR`	Argument '`p_model`' or '`p_calc`' passed a `NULL` pointer.

Returned Value

None.

Notes/Warnings

For a data stream CRC calculation, the CRC model must include a table of pre-computed CRC values.

A data stream CRC calculation proceeds as follows:

1. `CRC_Open_xxBit()` called to initialize `CRC_CALC_xx` structure.
2. `CRC_WrBlock_xxBit()` called for each block of bytes to factor into CRC.

   AND / OR

   `CRC_WrOctet_xxBit()` called for each octet to factor into CRC.
3. `CRC_Close_xxBit()` called to get final CRC value.

Example

Listing A-2. Example use, data strean calculation

```
static  CRC_MODEL_16  App_CRCModel = {
    0x1021,
    0xFFFF,
    DEF_NO,
    0x0000,
    (CPU_INT16U *)&CRC_TblCRC16_1021[0]
};


void  App_ChkSumCalc (void)
{
    CPU_ERR      err;
    CPU_INT16U   crc;
    CRC_CALC_16  calc;
    CPU_INT32U   i;
    CPU_INT08U   octet;


    APP_TRACE_DEBUG(("Testing CRC, data stream calculation ...\n\r"));

    CRC_Open_16Bit(App_CRCModel, &calc, &err);

    if (err != EDC_CRC_ERR_NONE) {
        APP_TRACE_DEBUG((“... open failed.\n\r”)
        return;
    }

    for (i = 0; i < 1024; i++) {
        octet = Ser_RdByte();                       /* Read byte from serial port.  */
        CRC_WrOctet_16Bit(&calc, octet);            /* Update CRC.                  */
    }

    crc = CRC_Close_16Bit(&calc);

    APP_TRACE_DEBUG((“... result = %04X.\n\r”, crc));
}
```

A.01.04 CRC_WrBlock_xxBit()

Process buffer for data stream CRC calculation.

Prototypes

```
void  CRC_WrBlock_16Bit (CRC_CALC_16  *p_calc,
                         void         *p_buf,
                         CPU_INT32U    nbr_octets);

void  CRC_WrBlock_32Bit (CRC_CALC_32  *p_calc,
                         void         *p_buf,
                         CPU_INT32U    nbr_octets);
```

Arguments

`p_calc`	Pointer to calculation.
`p_buf`	Pointer to data buffer that holds data for CRC calculation.
`nbr_octets`	Number of data octets in buffer.

Returned Value

None.

Notes/Warnings

For more information (and an example usage), please see `CRC_Open_xxBit()`.

The calculation must have been previously initialized with `CRC_Open_xxBit()`.

A.01.05 CRC_WrOctet_xxBit()

Process data octet for data stream CRC calculation.

Prototype

```
void  CRC_WrOctet_16Bit (CRC_CALC_16  *p_calc,
                         CPU_INT08U    octet);

void  CRC_WrOctet_32Bit (CRC_CALC_32  *p_calc,
                         CPU_INT08U    octet);
```

Arguments

`p_calc`	Pointer to calculation.
`octet`	Octet for CRC calculation.

Returned Value

None.

Notes/Warnings

For more information, please see `CRC_Open_xxBit()`.

The calculation must have been previously initialized with `CRC_Open_xxBit()`.

Example

See `CRC_Open_xxBit()`.

A.02 Hamming Code Functions

A.02.01 Hamming_Calc_xxx()

Calculate Hamming code for 4- or 256-byte buffer.

Prototype

```
void  Hamming_Calc_004 (void        *p_buf,
                        CPU_SIZE_T   len,
                        CPU_INT08U  *p_ecc,
                        CPU_ERR     *p_err);

void  Hamming_Calc_256 (void        *p_buf,
                        CPU_SIZE_T   len,
                        CPU_INT08U  *p_ecc,
                        CPU_ERR     *p_err);
```

Arguments

`p_buf` Pointer to buffer that contains the data.

`len` Length of buffer, in octets; MUST be 4 and 256, respectively.

`p_ecc` Pointer to buffer that will receive ECC.

`p_err` Pointer to variable that will receive return error code from this function:

`ECC_ERR_NONE`	Hamming code calculated.
`ECC_ERR_NULL_PTR`	Argument `p_buf` passed a NULL pointer.
`ECCR_INVALID_LEN`	Argument `len` passed an invalid length.

Returned Value

None.

Notes/Warnings

1. a. The return parameter p_ecc must point to a valid 4-byte buffer; this buffer need not be CPU_INT32U-aligned.
   b. The calculation is optimized for `CPU_INT32U`-aligned buffers, though any buffer alignment is acceptable.

2. The 22-bit Hamming code returned from `Hamming_Calc_256()` is stored in the 32-bit return variable. Data buffers larger than 256 bytes should either :

   a. Use other Hamming code calculation functions in this module:

`Hamming_Calc()`	Calculate Hamming code for *n*-byte data buffer.

b. Be divided into smaller segments, with the ECCs calculated for the individual segments concatenated to form the ECC for the entire buffer.

3. The 10-bit Hamming code returned from `Hamming_Calc_004()` is stored in the 16bit return variable.

A.02.02 Hamming_Calc()

Calculate Hamming code for *n*-byte buffer.

Prototype

```
void  Hamming_Calc (void        *pbuf,
                    CPU_SIZE_T   len,
                    CPU_INT08U  *p_ecc,
                    CPU_ERR     *perr);
```

Arguments

`pbuf` Pointer to buffer that contains the data.

`len` Length of buffer, in octets.

`p_ecc` Pointer to buffer that will receive ECC.

`p_err` Pointer to variable that will receive return error code from this function:

`ECC_ERR_NONE`	Hamming code calculated.
`ECC_ERR_NULL_PTR`	Argument `p_buf` passed a NULL pointer.
`ECCR_INVALID_LEN`	Argument `len` passed an invalid length.

Returned Value

None.

Notes/Warnings

1. a. The return parameter p_ecc must point to a valid 4-byte buffer; this buffer need not be CPU_INT32U-aligned.
   b. The calculation is optimized for `CPU_INT32U`-aligned buffers, though any buffer alignment is acceptable.

2. Certain buffer lengths only are supported:

   a. Buffer lengths greater than or equal to 128 that are also multiples of 128: 128, 256, 384, 512, ….
   b. A 2^n-bit data buffer requires a 2n-bit ECC. Consequently, the maximum data length that can be accomodated with a 32-bit ECC is

      $2^{32/2}$ bits = 2^{16} bits = 2^{13} byte = 8192 byte.

3. The 20- to 32-bit is stored in the 32-bit return variable.

A.02.03 Hamming_Chk_xxx()

Check previously computed Hamming code against current data for 4- or 256-byte buffer.

Prototypes

```
CPU_INT08U  Hamming_Chk_004 (void          *p_buf,
                             CPU_SIZE_T     len,
                             CPU_INT08U    *p_ecc,
                             ECC_ERR_ADDR  *p_err_addr_tbl,
                             CPU_INT08U     max_errs,
                             CPU_ERR       *p_err);

CPU_INT08U  Hamming_Chk_256 (void          *p_buf,
                             CPU_SIZE_T     len,
                             CPU_INT08U    *p_ecc,
                             ECC_ERR_ADDR  *p_err_addr_tbl,
                             CPU_INT08U     max_errs,
                             CPU_ERR       *p_err);
```

Arguments

`p_buf` Pointer to buffer that contains the data.

`len` Length of buffer, in octets; MUST be 4 and 256, respectively.

`p_ecc` Pointer to buffer that contains the ECC.

`p_err_addr_tbl` Pointer to table that will receie the addresses of any errors.

`max_errs` Size of `p_err_addr_tbl`; the maximum number of error locations that can be returned.

`p_err` Pointer to variable that will receive return error code from this function:

`ECC_ERR_NONE`	Hamming code verified.
`ECC_ERR_CORRECTABLE`	Correctable error detected in data.
`ECC_ERR_UNCORRECTABLE`	Uncorrectable error detected.
`ECC_ERR_INVALID_LEN`	Argument `len` passed an invalid length.
`ECC_ERR_NULL_PTR`	Argument `p_buf`, `p_ecc` or `p_err_addr_tbl` passed a NULL pointer.

Returned Value

The number of bit errors in the data.

Notes/Warnings

1. If a correctable erorr is found, `p_err_addr_tbl[0].AddrBit` and `p_err_addr_tbl[0].AddrOctet` will specify the location of the bit error. To correct this, the caller should complement that bit:

```
Hamming_Chk_256(&buf[0],
                 256u,
                &hamming_prev,
                &err_addr,
                 1u,
                &err);

if (err == ECC_ERR_CORRECTABLE) {
    buf[err_addr.AddrOctet] ^= DEF_BIT(err_addr.AddrBit);
}
```

2. The maximum number of errors that can be corrected is 1, so `max_errs` should be 1.

3. An uncorrectable error is one in which two or more bits of the data have changed and the error is detectable.

4. If more than two bits of data have changed, the error may be mis-diagnosed as a single bit error.

A.02.04 Hamming_Chk()

Check previously computed Hamming code against current data for *n*-byte buffer.

Prototype

```
CPU_INT08U  Hamming_Chk (void          *p_buf,
                         CPU_SIZE_T     len,
                         CPU_INT08U    *p_ecc,
                         ECC_ERR_ADDR  *p_err_addr_tbl,
                         CPU_INT08U     max_errs,
                         CPU_ERR       *p_err);
```

Arguments

`p_buf` Pointer to buffer that contains the data.

`len` Length of buffer, in octets.

`p_ecc` Pointer to buffer that contains the ECC.

`p_err_addr_tbl` Pointer to table that will receie the addresses of any errors.

`max_errs` Size of `p_err_addr_tbl`; the maximum number of error locations that can be returned.

`p_err` Pointer to variable that will receive return error code from this function:

`ECC_ERR_NONE`	Hamming code verified.
`ECC_ERR_CORRECTABLE`	Correctable error detected in data.
`ECC_ERR_UNCORRECTABLE`	Uncorrectable error detected.
`ECC_ERR_INVALID_LEN`	Argument `len` passed an invalid length.
`ECC_ERR_NULL_PTR`	Argument `p_buf`, `p_ecc` or `p_err_addr_tbl` passed a NULL pointer.

Returned Value

None.

Notes/Warnings

1. If a correctable erorr is found, `p_err_addr_tbl[0].AddrBit` and `p_err_addr_tbl[0].AddrOctet` will specify the location of the bit error. To correct this, the caller should complement that bit:

```
Hamming_Chk(&buf[0],
             len,
            &hamming_prev,
            &err_addr,
```

```
                1,
               &err);

    if (err == ECC_ERR_CORRECTABLE) {
        buf[err_addr.AddrOctet] ^= DEF_BIT(err_addr.AddrBit);
    }
```

2. The maximum number of errors that can be corrected is 1, so `max_errs` should be 1.

3. An uncorrectable error is one in which two or more bits of the data have changed and the error is detectable.

4. If more than two bits of data have changed, the error may be mis-diagnosed as a single bit error.

5. The value passed to the `len` argument MUST match the value passed to the `len` argument of `Hamming_Calc()`.

A.02.05 Hamming_Correct_xxx()

Check previously computed Hamming code against current data and correct any error(s), if possible, for 4- or 256-byte buffer.

Prototypes

```
void  Hamming_Correct_004 (void        *p_buf,
                           CPU_SIZE_T   len,
                           CPU_INT08U  *p_ecc,
                           CPU_ERR     *p_err);

void  Hamming_Correct_256 (void        *p_buf,
                           CPU_SIZE_T   len,
                           CPU_INT08U  *p_ecc,
                           CPU_ERR     *p_err);
```

Arguments

`p_buf` Pointer to buffer that contains the data.

`len` Length of buffer, in octets; MUST be 4 and 256, respectively.

`p_ecc` Pointer to buffer that contains the ECC.

`p_err` Pointer to variable that will receive return error code from this function:

`ECC_ERR_NONE`	Hamming code verified.
`ECC_ERR_CORRECTABLE`	Correctable error detected in data.
`ECC_ERR_UNCORRECTABLE`	Uncorrectable error detected.
`ECC_ERR_INVALID_LEN`	Argument `len` passed an invalid length.
`ECC_ERR_NULL_PTR`	Argument `p_buf` or `p_ecc` or passed a NULL pointer.

Returned Value

None.

Notes/Warnings

1. The maximum number of errors that can be corrected is 1.
2. An uncorrectable error is one in which two or more bits of the data have changed and the error is detectable.
3. If more than two bits of data have changed, the error may be mis-diagnosed as a single bit error.

A.02.06 Hamming_Correct()

Check previously computed Hamming code against current data and correct any error(s), if possible, for *n*-byte buffer.

Prototypes

```
void  Hamming_Correct (void        *p_buf,
                       CPU_SIZE_T   len,
                       CPU_INT08U  *p_ecc,
                       CPU_ERR     *p_err);
```

Arguments

`p_buf` Pointer to buffer that contains the data.

`len` Length of buffer, in octets.

`p_ecc` Pointer to buffer that contains the ECC.

`p_err` Pointer to variable that will receive return error code from this function:

`ECC_ERR_NONE`	Hamming code verified.
`ECC_ERR_CORRECTABLE`	Correctable error detected in data.
`ECC_ERR_UNCORRECTABLE`	Uncorrectable error detected.
`ECC_ERR_INVALID_LEN`	Argument `len` passed an invalid length.
`ECC_ERR_NULL_PTR`	Argument `p_buf` or `p_ecc` or passed a NULL pointer.

Returned Value

None.

Notes/Warnings

1. The maximum number of errors that can be corrected is 1.
2. An uncorrectable error is one in which two or more bits of the data have changed and the error is detectable.
3. If more than two bits of data have changed, the error may be mis-diagnosed as a single bit error.

A.03 BCH Code Functions

A.03.01 BCH_4Bit_Calc()

Calculate 4-bit correcting BCH code for 512- to 528-byte buffer.

Prototype

```
void  Hamming_4Bit_Calc (void        *p_buf,
                         CPU_SIZE_T   len,
                         CPU_INT08U  *p_ecc,
                         CPU_ERR     *p_err);
```

Arguments

`p_buf` Pointer to buffer that contains the data.

`len` Length of buffer, in octets; MUST be between 512 and 528, inclusive.

`p_ecc` Pointer to buffer that will receive ECC.

`p_err` Pointer to variable that will receive return error code from this function:

`ECC_ERR_NONE`	BCH code calculated.
`ECC_ERR_NULL_PTR`	Argument `p_buf` passed a NULL pointer.
`ECC_ERR_INVALID_LEN`	Argument `len` passed an invalid length.

Returned Value

None.

Notes/Warnings

1. The return parameter `p_ecc` must point to a valid 7-byte buffer.
2. The 52-bit BCH code is stored in the 7-byte buffer, with the bottom four bits of the 0^{th} byte unused.
3. Data buffers larger than 512 bytes should be divided into 512-byte segments, with ECCs calculated for the individual segments concatenated to form the ECC for the entire buffer.

A.03.02 BCH_8Bit_Calc()

Calculate 8-bit correcting BCH code for 512- to 528-byte buffer.

Prototype

```
void  Hamming_8Bit_Calc (void        *p_buf,
                         CPU_SIZE_T   len,
                         CPU_INT08U  *p_ecc,
                         CPU_ERR     *p_err);
```

Arguments

`p_buf` Pointer to buffer that contains the data.

`len` Length of buffer, in octets; MUST be between 512 and 528, inclusive.

`p_ecc` Pointer to buffer that will receive ECC.

`p_err` Pointer to variable that will receive return error code from this function:

`ECC_ERR_NONE`	BCH code calculated.
`ECC_ERR_NULL_PTR`	Argument `p_buf` passed a NULL pointer.
`ECC_ERR_INVALID_LEN`	Argument `len` passed an invalid length.

Returned Value

None.

Notes/Warnings

1. The return parameter `p_ecc` must point to a valid 13-byte buffer.

2. The 104-bit BCH code is stored in the 13-byte buffer.

3. Data buffers larger than 512 bytes should be divided into 512-byte segments, with ECCs calculated for the individual segments concatenated to form the ECC for the entire buffer.

A.03.03 BCH_4Bit_Chk()

Check previously computed 4-bit correcting BCH code against current data for 512- to 528-byte buffer.

Prototypes

```
CPU_INT08U  BCH_4Bit_Chk (void          *p_buf,
                          CPU_SIZE_T     len,
                          CPU_INT08U    *p_ecc,
                          ECC_ERR_ADDR  *p_err_addr_tbl,
                          CPU_INT08U     max_errs,
                          CPU_ERR       *p_err);
```

Arguments

`p_buf` Pointer to buffer that contains the data.

`len` Length of buffer, in octets; MUST be between 512 and 528, inclusive.

`p_ecc` Pointer to buffer that contains the ECC.

`p_err_addr_tbl` Pointer to table that will receive the addresses of any errors.

`max_errs` Size of `p_err_addr_tbl`; the maximum number of error locations that can be returned.

`p_err` Pointer to variable that will receive return error code from this function:

`ECC_ERR_NONE`	Hamming code verified.
`ECC_ERR_CORRECTABLE`	Correctable error detected in data.
`ECC_ERR_UNCORRECTABLE`	Uncorrectable error detected.
`ECC_ERR_INVALID_LEN`	Argument `len` passed an invalid length.
`ECC_ERR_NULL_PTR`	Argument `p_buf`, `p_ecc` or `p_err_addr_tbl` passed a NULL pointer.

Returned Value

The number of bit errors in the data.

Notes/Warnings

1. If correctable error(s) are found, `p_err_addr_tbl[n].AddrBit` and `p_err_addr_tbl[n].AddrOctet` will specify the location of the bit error. To correct this, the caller should complement that bit:

```
err_cnt = BCH_4Bit_Chk(&buf[0],
                        512,
                       &bch_prev[0],
                       &err_addr_tbl[0],
```

```
                                4,
                               &err);


    if (err == ECC_ERR_CORRECTABLE) {
        for (i = 0; i < err_cnt; i++) {
            buf[err_addr_tbl[i].AddrOctet] ^=
                                    DEF_BIT(err_addr_tbl[i].AddrBit);
    }
```

2. The maximum number of errors that can be corrected is four, so `max_errs` should be four.

3. An uncorrectable error is one in which five or more bits of the data have changed and the error is detectable.

4. If more than four bits of data have changed, the error may be mis-diagnosed as a correctabl error.

A.03.04 BCH_8Bit_Chk()

Check previously computed 8-bit correcting BCH code against current data for 512- to 528-byte buffer.

Prototypes

```
CPU_INT08U  BCH_8Bit_Chk (void          *p_buf,
                          CPU_SIZE_T     len,
                          CPU_INT08U    *p_ecc,
                          ECC_ERR_ADDR  *p_err_addr_tbl,
                          CPU_INT08U     max_errs,
                          CPU_ERR       *p_err);
```

Arguments

`p_buf` Pointer to buffer that contains the data.

`len` Length of buffer, in octets; MUST be between 512 and 528, inclusive.

`p_ecc` Pointer to buffer that contains the ECC.

`p_err_addr_tbl` Pointer to table that will receive the addresses of any errors.

`max_errs` Size of `p_err_addr_tbl`; the maximum number of error locations that can be returned.

`p_err` Pointer to variable that will receive return error code from this function:

`ECC_ERR_NONE`	Hamming code verified.
`ECC_ERR_CORRECTABLE`	Correctable error detected in data.
`ECC_ERR_UNCORRECTABLE`	Uncorrectable error detected.
`ECC_ERR_INVALID_LEN`	Argument `len` passed an invalid length.
`ECC_ERR_NULL_PTR`	Argument `p_buf`, `p_ecc` or `p_err_addr_tbl` passed a NULL pointer.

Returned Value

The number of bit errors in the data.

Notes/Warnings

1. If correctable error(s) are found, `p_err_addr_tbl[n].AddrBit` and `p_err_addr_tbl[n].AddrOctet` will specify the location of the bit error. To correct this, the caller should complement that bit:

```
err_cnt = BCH_8Bit_Chk(&buf[0],
                        512,
                       &bch_prev[0],
                       &err_addr_tbl[0],
```

```
                                8,
                               &err);


    if (err == ECC_ERR_CORRECTABLE) {
        for (i = 0; i < err_cnt; i++) {
            buf[err_addr_tbl[i].AddrOctet] ^=
                                    DEF_BIT(err_addr_tbl[i].AddrBit);
    }
```

2. The maximum number of errors that can be corrected is eight, so `max_errs` should be eight.

3. An uncorrectable error is one in which nine or more bits of the data have changed and the error is detectable.

4. If more than eight bits of data have changed, the error may be mis-diagnosed as a correctable error.

A.03.05 BCH_4Bit_Correct()

Check previously computed 4-bit correcting BCH code against current data and correct any error(s), if possible, for 512- to 528-byte buffer.

Prototypes

```
void  BCH_4Bit_Correct (void        *p_buf,
                       CPU_SIZE_T   len,
                       CPU_INT08U  *p_ecc,
                       CPU_ERR     *p_err);
```

Arguments

`p_buf` Pointer to buffer that contains the data.

`len` Length of buffer, in octets; MUST be between 512 and 528, inclusive.

`p_ecc` Pointer to buffer that contains the ECC.

`p_err` Pointer to variable that will receive return error code from this function:

`ECC_ERR_NONE`	Hamming code verified.
`ECC_ERR_CORRECTABLE`	Correctable error detected in data.
`ECC_ERR_UNCORRECTABLE`	Uncorrectable error detected.
`ECC_ERR_INVALID_LEN`	Argument `len` passed an invalid length.
`ECC_ERR_NULL_PTR`	Argument `p_buf` or `p_ecc` or passed a NULL pointer.

Returned Value

None.

Notes/Warnings

1. The maximum number of errors that can be corrected is four.
2. An uncorrectable error is one in which five or more bits of the data have changed and the error is detectable.
3. If more than four bits of data have changed, the error may be mis-diagnosed as a correctable error.

A.03.06 BCH_8Bit_Correct()

Check previously computed 8-bit correcting BCH code against current data and correct any error(s), if possible, for 512- to 528-byte buffer.

Prototypes

```
void  BCH_8Bit_Correct (void        *p_buf,
                       CPU_SIZE_T   len,
                       CPU_INT08U  *p_ecc,
                       CPU_ERR     *p_err);
```

Arguments

`p_buf` Pointer to buffer that contains the data.

`len` Length of buffer, in octets; MUST be between 512 and 528, inclusive.

`p_ecc` Pointer to buffer that contains the ECC.

`p_err` Pointer to variable that will receive return error code from this function:

`ECC_ERR_NONE`	Hamming code verified.
`ECC_ERR_CORRECTABLE`	Correctable error detected in data.
`ECC_ERR_UNCORRECTABLE`	Uncorrectable error detected.
`ECC_ERR_INVALID_LEN`	Argument `len` passed an invalid length.
`ECC_ERR_NULL_PTR`	Argument `p_buf` or `p_ecc` or passed a NULL pointer.

Returned Value

None.

Notes/Warnings

1. The maximum number of errors that can be corrected is eight.
2. An uncorrectable error is one in which nine or more bits of the data have changed and the error is detectable.
3. If more than eight bits of data have changed, the error may be mis-diagnosed as a correctable error.

Appendix B

µC/CRC Configuration

µC/CRC is configurable at compile time via severa `#defines` in an application's *crc_cfg.h* file. µC/CRC uses #defines because they allow code and data sizes to be scaled at compile time based on enabled features. In other words, this allows the ROM and RAM footprints of µC/FS to be adjusted based on your requirements.

Most of the #defines should be configured with the default configuration values. This leaves about a dozen or so values that should be configured with values that may deviate from the default configuration. The default configuration values are show in **bold**.

B.01 CRC Configuration

B.01.01 Optmization: EDC_CRC_CFG_OPTIMIZE_ASM_EN

EDC_CRC_CFG_OPTIMIZE_ASM_EN selects whether optimized assembly-language are used function to perform calculation:

DEF_ENABLED	Optimized assembly-language functions are used.
DEF_DISABLED	Optimized assembly-language functions are NOT used.

B.01.02 External Argument Checking: EDC_CRC_CFG_ARG_CHK_EXT_EN

EDC_CRC_CFG_ARG_CHK_EXT_EN allows code to be generated to check arguments for functions that can be called by the user and for functions which are internal but receive arguments from an API that the user can call:

DEF_ENABLED	External arguments checked.
DEF_DISABLED	External arguments NOT checked.

B.01.03 Enable/Disable CRC Table: EDC_CRC_CFG_CRCxx_ EN

Each of the following eight `#defines` control the inclusion of the table and sample model provided for the four most common CRCs. If a CRC will be used, then the corresponding `#define` should be `DEF_ENABLED`; otherwise, the corresponding `#define` should be `DEF_DISABLED`. Note that the sample models may not fit your application because the initial value or final XOR value might need to be changed.

EDC_CRC_CFG_CRC16_1021_EN controls inclusion of the table and sample model for the 16-bit CRC with polynomial `0x1021` for use without data reflection.

EDC_CRC_CFG_CRC16_8005_EN controls inclusion of the table and sample model for the 16-bit CRC with polynomial `0x8005` for use without data reflection.

EDC_CRC_CFG_CRC16_8048_EN controls inclusion of the table and sample model for the 16-bit CRC with polynomial `0x8048` for use without data reflection.

EDC_CRC_CFG_CRC32_EN controls inclusion of the table and sample model for the 32-bit CRC with polynomial `0x04C11DB7` for use without data reflection.

EDC_CRC_CFG_CRC16_1021_REF_EN controls inclusion of the table and sample model for the 16-bit CRC with polynomial `0x1021` for use with data reflection.

EDC_CRC_CFG_CRC16_8005_REF_EN controls inclusion of the table and sample model for the 16-bit CRC with polynomial `0x8005` for use with data reflection.

EDC_CRC_CFG_CRC16_8048_REF_EN controls inclusion of the table and sample model for the 16-bit CRC with polynomial `0x8048` for use with data reflection.

EDC_CRC_CFG_CRC32_REF_EN controls inclusion of the table and sample model for the 32-bit CRC with polynomial `0x04C11DB7` for use with data reflection.

B.02 Hamming Code Configuration

B.02.01 Optmization: ECC_HAMMING_CFG_OPTIMIZE_ASM_EN

ECC_HAMMING_CFG_OPTIMIZE_ASM_EN selects whether optimized assembly-language are used function to perform calculation:

DEF_ENABLED	Optimized assembly-language functions are used.
DEF_DISABLED	Optimized assembly-language functions are NOT used.

B.02.02 External Argument Checking: ECC_HAMMING_CFG_ARG_CHK_EXT_EN

ECC_HAMMING_CFG_ARG_CHK_EXT_EN allows code to be generated to check arguments for functions that can be called by the user and for functions which are internal but receive arguments from an API that the user can call:

DEF_ENABLED	External arguments checked.
DEF_DISABLED	External arguments NOT checked.

B.03 BCH Code Configuration

B.03.01 Optmization: ECC_BCH_xBIT_CFG_OPTIMIZE_ASM_EN

ECC_BCH_4BIT_CFG_OPTIMIZE_ASM_EN / **ECC_BCH_8BIT_CFG_OPTIMIZE_ASM_EN** select whether optimized assembly-language are used function to perform calculation:

DEF_ENABLED	Optimized assembly-language functions are used.
DEF_DISABLED	Optimized assembly-language functions are NOT used.

B.03.02 External Argument Checking: ECC_BCH_xBIT_CFG_ARG_CHK_EXT_EN

ECC_BCH_4BIT_CFG_ARG_CHK_EXT_EN / **ECC_BCH_8BIT_CFG_ARG_CHK_EXT_EN** allows code to be generated to check arguments for functions that can be called by the user and for functions which are internal but receive arguments from an API that the user can call:

DEF_ENABLED	External arguments checked.
DEF_DISABLED	External arguments NOT checked.